

Артур Гриффитс



КОМПИЛЯТОР GNU №1 в мире

Настольная книга пользователей,
программистов и системных администраторов

Компиляция любого программного обеспечения с открытым исходным кодом, включая Linux, FreeBSD, GNOME, StarOffice, OpenOffice, Apache Web Server, и любых приложений для них. Перенос программ на любую операционную среду и десятки аппаратных платформ.

Разработка и
перенос
программного
обеспечения на
любую платформу
Unix, включая
Linux и BSD.
Компиляция для
Windows

Инсталляция и
использование
вашего
собственного
компилятора
языков C, C++,
Objective-C,
Fortran, Java и Ada

Генерация
стандартного
исполняемого
кода для
различных
платформ

книга-почтой e-магазин

www.diasoft.kiev.ua



издательство
DiaSoft



OSBORNE

GCC:

The Complete Reference

Arthur Griffith

McGraw-Hill/Osborne
New York Chicago San Francisco
Lisbon London Madrid Mexico City
Milan New Delhi San Juan
Seoul Singapore Sydney Toronto

GCC.

Настольная книга пользователей, программистов и системных администраторов

Артур Гриффитс



торгово-издательский дом
DiaSoft

Москва • Санкт-Петербург • Киев
2004

УДК 004.4'422
ББК 32.973-018.2

Г 85

Гриффитс Артур

Г 85 *GCC. Настольная книга пользователей, программистов и системных администраторов:* Пер. с англ./Артур Гриффитс. — К.: ООО «ТИД «ДС», 2004.— 624 с.

ISBN 966-7992-34-9

GCC — основной компилятор проекта **GNU**. Он поддерживает набор всех наиболее используемых языков программирования и обеспечивает перенос программ на десятки аппаратных платформ. Все свободно распространяемое программное обеспечение, включая и компиляторы, на том или ином уровне основываются на **GCC**.

В книге даются подробные сведения о получении, конфигурировании, установке и тестированию компилятора. Представлено построение кросс-компилятора и создание встраиваемых систем, детально описывается компиляция программ на языках C, C++, Objective-C, Fortran, Java и Ada. А также сочетание в одной программе нескольких языков программирования и включение в нее частей, написанных на ассемблере или языках системного уровня. В этой книге можно найти практически любые сведения, достаточные не только для разрешения ваших проблем, но и для участия в разработке и поддержке самого компилятора **GCC**.

Книга будет полезна: программистам-разработчикам и руководителям программных проектов; администраторам и системным программистам, которым приходится заниматься переносом программного обеспечения и приложений; пользователям, заинтересованным в использовании программ с открытым исходным кодом. И всем сторонникам развития движения по созданию свободно распространяемых программ.

ББК 32.973-018.2

Original edition copyright © 2002 by McGraw-Hill/Osborne, as set forth in copyright notice of Proprietor's edition. All rights reserved.

Russian language edition copyright © 2004 by DiaSoft Publishing House. All rights reserved.

Лицензия предоставлена издательством McGraw-Hill/Osborne.

Все права зарезервированы, включая право на полное или частичное воспроизведение в какой бы то ни было форме.

Материал, изложенный в данной книге многократно проверен. Но поскольку вероятность технических ошибок все равно остается, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

Все торговые знаки, упомянутые в настоящем издании, зарегистрированы. Случайное неправильное использование или пропуск торгового знака или названия его законного владельца не должно рассматриваться как нарушение прав собственности.

ISBN 966-7992-34-9 (рус.) © Перевод на русский язык. ООО «ТИД «ДС», 2004

ISBN 0-07-222405-3 (англ.) © McGraw-Hill/Osborne, 2002

© Оформление. ООО «ТИД «ДС», 2004

Питиеническое заключение № 77.99.6.953.П.438.2.99 от 04.02.1999

Оглавление

Введение	16
Часть I. Свободно распространяемый компилятор	19
Глава 1. Введение в GCC	20
GNU	20
Сравнение компиляторов	21
Опции командной строки	22
Платформы	23
Что делает компилятор	24
Языки программирования	25
Язык программирования C — фундаментальный язык GCC	25
Первым добавлением был язык C++	26
Язык Objective-C	26
Добавление языка Fortran	26
Добавление языка Java	27
Добавление языка Ada	27
Язык Chill покинул семейство GCC	28
Список частей компилятора GCC	28
Контакты	31
Глава 2. Получение и установка компилятора GCC	33
Загрузка готовой к запуску скомпилированной версии	34
Загрузка исходного кода по FTP	35
Загрузка исходного кода через систему CVS	37
Предыдущие выпуски	38
Экспериментальная версия	39
Компиляция и установка GCC	39
Процедура инсталляции	40
Опции конфигурации	41
Пакет binutils	53
Установка прекомпилированной версии для Microsoft Windows	54
Суґвін	54
Инсталляция Cygwin	55
Запуск проверочного набора	56
Часть II. Использование Сборного Компилятора	59
Глава 3. Препроцессор cpp	60
Директивы препроцессора	60
#define	61
#error и #warning	64
#if, #elif, #else и #endif	64
#endif, #ifndef, #else и #endif	66

#include	66
#include_next	68
#line	68
Директивы #pragma и оператор_Pragma	69
#undef	70
##	70
Предопределенные макросы	71
Включение заголовочного файла единственный раз	73
Включение информации о расположении кода в сообщения об ошибках	74
Временное удаление части исходного кода	75
Создание компоновочных файлов (makefiles)	75
Опции командной строки и переменные окружения	76
Глава 4. Компиляция программ на языке С	77
Базовая компиляция	77
Преобразование отдельного исходного файла в готовую к запуску программу	78
Переработка исходного файла в объектный модуль	79
Преобразование нескольких исходных файлов в готовую к запуску программу	79
Обработка исходного кода препроцессором	80
Выработка ассемблерного кода	80
Создание статической библиотеки	80
Создание разделяемой библиотеки	82
Замещение соглашений об именах	83
Поддержка стандартов языка	84
Расширения GNU языка С	84
Выравнивание	85
Безымянные (анонимные) объединения	85
Массивы переменной длины	86
Массивы нулевой длины	87
Атрибуты	88
Составные операторы, возвращающие значение	93
Условный пропуск операнда	94
Неполные перечисляемые типы	95
Построение аргументов функций	95
Расширение вызовов функций подстановкой кода	96
Имена функций	97
Использование вложенных функций	98
Прототипы функций	99
Адреса возврата из функций и кадры стэка	99
Идентификаторы	100
Целые числа	100
Альтернативные формы ключевых слов	101
Адреса меток	101
Локально объявляемые метки	102
Составные выражения в левой части оператора присваивания, Lvalue	103
Макроопределения с переменным количеством аргументов	103
Строки	104

Арифметические действия над указателями	104
Операторы Switch и Case	105
Создание имени определяемого типа	105
Ссылки на типы переменных	106
Приведение типов объединения	107
Глава 5. Компиляция программ на языке C++	108
Базовая компиляция	108
Компиляция отдельного исходного файла в готовую к запуску программу	109
Преобразование нескольких исходных файлов в готовую к запуску программу ...	110
Компиляция исходного кода в объектный	111
Предобработка	111
Выработка компилятором ассемблерного кода	112
Создание статической библиотеки	112
Создание разделяемой библиотеки	114
Расширения языка C++	116
Атрибуты	116
Включаемые заголовочные файлы	117
Имена функций	117
Объявление класса и код его реализации (Interface и Implementation)	118
Операторы <? и >	119
Ограничение указателей	120
Действия компилятора	121
Библиотеки	121
Представление символических имен в объектном коде	122
Компоновка программ	124
Экземпляры компилируемого шаблона	125
Глава 6. Компиляция программ на языке Objective-C	127
Базовая компиляция	127
Компиляция отдельного исходного файла в готовую к запуску программу	128
Компиляция программ, использующих объекты	129
Создание и использование статической библиотеки	131
Создание разделяемой библиотеки	133
Общие замечания, касающиеся языка Objective-C	133
Предопределенные типы	134
Создание интерфейсного объявления	134
Присвоение символических имен и их представление в объектном коде	135
Глава 7. Компиляция программ на языке Fortran	136
Базовая компиляция	136
Преобразование отдельного исходного файла в готовую машинную программу	137
Преобразование нескольких исходных файлов в исполняемый файл	138
Генерирование ассемблерного кода	139
Предобработка	139
Создание и использование статической библиотеки	140
Создание разделяемой библиотеки	141
Ratfor	142

Особенности и расширения GNU Fortran	143
Встроенные функции	144
Формат исходного кода	144
Комментарии	145
Знак доллара	145
Заглавные и строчные буквы	145
Особенности Fortran 90	147
Глава 8. Компиляция программ на языке Java	152
Базовая компиляция	152
Преобразование отдельного исходного файла в машинную программу	153
Компиляция отдельного исходного файла в байт-код класса виртуальной машины Java	154
Двоичный объектный файл из отдельного исходного файла на языке Java	154
Преобразование байт-кода интерпретатора Java в машинную программу	155
Компиляция нескольких исходных файлов Java в запускаемую программу	155
Компиляция нескольких входных файлов интерпретатора JVM в машинный код	157
Выработка ассемблерного кода	157
Создание статической библиотеки	158
Создание разделяемой (динамической) библиотеки	159
Создание Java-архива .jar	159
Утилиты компилятора Java	160
gij	160
jar	161
gcjh	163
jcf-dump	164
jv-scan	165
jv-convert	166
grepjar	167
RMI	168
rmic	169
rmiregistry	170
Свойства	171
Глава 9. Компиляция программ на языке Ada	173
Инсталляция	173
Базовая компиляция	176
Преобразование отдельного исходного файла в исполняемый код	176
Выработка готовой программы из нескольких исходных файлов	178
Преобразование исходного файла на языке Ada в ассемблерный код	179
Опции компиляции	179
Утилиты, связанные с компиляцией программ на языке Ada	183
gnatbind	183
gnatlink	186
gnatmake	187
gnatchop	190
gnatxref	191

gnatfind	191
gnatkr	192
gnatprep	193
gnats	194
gnatsys и gnatsta	196
Глава 10. Совмещение языков	197
Совмещение C++ и C	198
Вызовы функций С из кода на языке C++	198
Вызовы функций C++ из кода на языке С	199
Совмещение языков Objective-C и C	200
Вызов С-функции из программы на языке Objective-C	200
Вызов функции Objective-C из программы на языке С	200
Сочетание языков Java и C++	202
Создание Java-объектов и вызов статических методов	202
Загрузка Java-класса и создание его экземпляров	204
Обработка исключений	206
Типы данных интерфейса CNI	207
Совмещение языков Java и C	207
Применение системно-ориентированного метода в классе Java	207
Передача аргументов методу, реализованному на языке системного уровня	209
Обратный вызов метода Java-класса из системно-ориентированного метода на языке С	211
Совмещение языков Fortran и C	213
Вызов функции С из кода на языке Fortran	213
Вызов из программы на языке С подпрограммы на языке Fortran	214
Совмещение языков Ada и C	216
Вызов кода С из программы на языке Ada	216
Вызов с аргументами функции С из программы на языке Ada	218
Глава 11. Интернационализация	220
Пример программы для применения перевода	221
Создание нового файла ".po"	223
Использование функций gettext()	225
Статические строки	226
Перевод строки, имеющейся в другом домене	226
Перевод из другого домена в указанной категории	226
Множественное число	227
Формы множественного числа из другого домена	227
Формы множественного числа из другого домена в указанной категории	227
Объединение двух файлов	227
Создание двоичного файла ".mo" из файла ".po"	229
Часть III. Внутренняя структура и окружение	231
Глава 12. Использование библиотек и способы компоновки	232
Объектные файлы и библиотеки	232
Объектные файлы в каталоге диска	232

Объектные файлы в статической библиотеке	233
Объектные файлы в динамических библиотеках	236
Оболочка компоновщика	236
Размещение библиотек	236
Поиск библиотек во время компоновки	237
Поиск библиотек во время выполнения программ	237
Загрузка функций из разделяемой библиотеки	238
Утилиты для работы с объектными файлами и библиотеками	241
Конфигурирование поиска разделяемых библиотек	241
Вывод из объектных файлов имен программных символов	243
Удаление неиспользуемой информации из объектных файлов	245
Вывод списка зависимостей, связанных с разделяемыми библиотеками	246
Вывод внутренней информации объектного файла	247
Глава 13. Использование отладчика GNU	250
Форматы отладочной информации	250
Формат STABS	251
Формат DWARF	252
Формат COFF	252
Формат XCOFF	253
Компиляция программы для отладки	253
Загрузка программы в отладчик	255
"Посмертный" анализ	258
Подключение отладчика к выполняемой программе	260
Сводная таблица команд отладчика	263
Глава 14. Утилиты Make и Autoconf	265
Утилита make	265
Внутренние определения	267
Как написать make-файл	269
Опции утилиты make	270
Утилита Autoconf	273
Глава 15. Ассемблер GNU	278
Управление ассемблированием из командной строки	278
Абсолютная и относительная адресация, выравнивание адресов	280
Вставка ассемблерного кода	281
Конструкция asm	282
Директивы ассемблера GNU	285
Глава 16. Перекрестная компиляция и перенос программ на систему MS Windows	298
Целевые платформы	298
Построение кросс-компилятора	299
Установка начального компилятора	300
Построение отдельного набора binutils, ориентированного на целевую платформу	300

Установка файлов из целевой системы	301
Конфигурируемая библиотека <i>libgcc1.a</i>	302
Компоновка кросс-компилятора	302
Запуск кросс-компилятора	303
MinGW — компилятор для Windows	303
Компилятор проекта <i>Cygwin</i>	304
Компиляция в <i>Cygwin</i> консольной программы	304
Компиляция в <i>Cygwin</i> программы, использующей объекты Windows GUI	305
Глава 17. Встраиваемые системы	306
Настройка компилятора и компоновщика	306
Выбор языка	308
Средства GCC для разработки встраиваемого программного обеспечения	309
Опции командной строки	309
Диагностика	310
Ассемблерный код	310
Библиотеки	311
Сокращение стандартной библиотеки	311
Библиотека, предназначенная для встраиваемых систем	312
Язык сценариев компоновщика <i>ld</i>	312
Пример сценария компоновщика № 1	313
Пример сценария компоновщика № 2	314
Некоторые другие команды сценариев компоновщика <i>ld</i>	315
Глава 18. Выход компилятора	316
Сведения о программе	316
Дерево синтаксического разбора	316
Заголовочные файлы	318
Необходимый программе объем памяти	318
Время компиляции программы	319
Промежуточное дерево компиляции C++	320
Иерархия классов в программах на языке C++	320
Информация для включения в сценарий <i>Makefile</i>	321
Информация о самом компиляторе	322
Отчет о времени компиляции	322
Ключи подпроцессов	323
Расширенная отладочная информация компилятора	324
Информация о файлах и каталогах	326
Глава 19. Реализация алгоритмического языка	328
Этапы компиляции программы	329
Лексический анализ	330
Пример построения простого lex-сканера	331
Построение lex-сканера с использованием регулярных выражений	331
Синтаксический разбор	332
Построение дерева синтаксического разбора	338
Подключение нижнего уровня компилятора к верхнему уровню	339

Глава 20. Язык регистрового переноса	342
Инструкции языка RTL	342
Шесть базовых кодов выражений	342
Типы и содержимое инструкций	344
Список инструкций RTL	347
Режимы и классы режимов	370
Флаги	373
Глава 21. Машино-зависимые опции Компилятора	375
Перечень аппаратных платформ	375
Опции командной строки для запуска компилятора GCC	376
Опции для платформ Alpha	376
Опции для процессоров Alpha/VMS	382
Опции для процессоров ARC	383
Опции для процессоров ARM	383
Опции для платформы AVR	390
Опции для платформы CRIS	391
Опции для платформы D30V	394
Опции для поддержки платформы H8/300	395
Опции для платформы HPPA	395
Опции компилятора для платформы IA-64	397
Опции для платформ Intel 386 и AMDx86-64	399
Опции для платформы Intel 960	406
Опции для платформы M32R/D	408
Опции для платформы M680x0	409
Опции для платформы M68HC1x	413
Опции для платформы M88K	414
Опции для платформы MCORE	417
Опции для платформы MIPS	418
Опции для платформы MMIX	426
Опции для платформы MN10200	427
Опции для платформы MN10300	427
Опции для платформы NS32K	428
Опции для платформы PDP-11	430
Опции для платформ RS/6000 и PowerPC	432
Опции для платформы RT	443
Опции для платформ S/390 и zSeries	444
Опции для платформы SH	445
Опции для платформы SPARC	447
Опции для платформы System V	452
Опции для платформы TMS320C3x/C4x	453
Опции для платформы V850	456
Опции для платформы VAX	457
Опции для платформы Xstormy16	457

Часть IV. Приложения	459
Приложение А. Генеральная Общественная Лицензия GNU	460
Перевод на русский язык Генеральной Общественной Лицензии GNU	461
Оригинальный текст Генеральной Общественной Лицензии GNU	469
Приложение Б. Переменные окружения	475
Приложение В. Перекрестная совместимость опций компилятора GCC	479
Совместимость опций	479
Приложение Г. Опции командной строки компилятора GCC	489
Префиксы опций	490
Порядок следования опций	491
Типы файлов	491
Алфавитный список опций	492
Словарь терминов	584
Предметный указатель	605

Посвящается Марии

Об авторе этой книги

Артур Гриффитс (Arthur Griffith) участвовал в работах по созданию таких программ, как трансляторы, компиляторы, компоновщики и ассемблеры с 1977 года. Именно тогда начался трудовой путь теперь уже широко известного программиста и автора многих книг. Как участник того проекта, он писал ассемблер и компоновщик для вычислительных машин специального назначения. Затем он вступил в группу, которая занималась поддержкой компилятора языка PL\EXUS, который имеет схожую с GCC внутреннюю структуру. Следующим был проект интерактивного интерпретатора и компилятора SATS.

Следующими разработками, в которых участвовал Гриффитс, были интерпретатор Forth и расширения для компилятора COBOL. Он также участвовал в разработке некоторых особых интерпретируемых языков для управления промышленным оборудованием. Одним из них был интерактивный командный язык для распределенного наземного управления системой космической связи.

Последние 5 лет Артур Гриффитс делает книги по компьютерной тематике, занимается дистанционным обучением программированию и пишет программы на языке Java. Его книги охватывают огромный тематический диапазон от "Java, XML и Jaxp" до "Язык COBOL для начинающих". Он много раз использовал компилятор GCC в работе над своими программами. И после включения в GCC языка Java он сам решил написать для вас эту книгу.

Благодарности

Прежде всего я должен выразить благодарность Вэнди Райнальди (Wendy Ranaldi), редактору издательства McGraw-Hill/Osborne, за возможность написания этой книги, и за проявленное ею огромное терпение. Особенно в первые дни, когда мне еще казалось, что впереди — целая вечность, и рабоча никуда не убежит.

Я хочу поблагодарить Кэти Конли (Katy Conley) за то, что она не давала мне сбиться с дороги и всегда мне указывала правильный путь. Она обладает уникальной способностью сохранять ровное настроение и отношение к работе при редактировании различных частей такой большой книги, как эта. Мы с Бартом Ридом (Bart Reed) имеем совершенно разные взгляды на английский язык — его английский куда более читаемый и правильный, чем мой. Я также хочу поблагодарить Пола Гарланда (Paul Garland) за проверку технической корректности этой книги и за то, что он смело указывал те места, где фантазия вводила меня в противоречие фактам.

Я весьма обязан Марго Мэйли (Margot Maley) за то, что она вовремя возвращала мои ноги на землю, а руки на клавиатуру.

Для написания этой книги мне очень пригодилось понимание технологии работы компиляторов. Хочу поблагодарить Дэйва Роджерса (Dave Rogers) за то, что он много лет тому назад познакомил меня с языком программирования C и сделал наброски для написания мною своего компилятора этого языка. Также благодарю Рона Саудера (Ron Souder) и Тревиса Митчела (Travis Mitchel) за привлечение меня к участию в достаточно странных проектах, которые, однако, помогли мне проникнуть в скрытые закоулки и узкие места в обработке исходного языка и выработке объектного кода.

И, возможно, более всех других я должен, пусть и слишком поздно, поблагодарить покойного Фреда Льюиса (Fred Lewis) за то, что он открыл мне захватывающий мир компиляторов, ассемблеров и объектных компоновщиков.

Введение

Следует заметить, что нынешний подъем движения по созданию свободно распространяемых программ является наиболее важным обстоятельством на современном этапе применения вычислительной техники. Мы находимся в середине принципиального перехода от использования дорогостоящего программного обеспечения, поддерживаемого жестоко конкурирующими корпорациями, к широкому выбору свободно распространяемых и бесплатно поддерживаемых программ, получатели которых могут использовать их для своих целей без каких-либо ограничений. Если при этом учесть, что практически все свободно распространяемые программы компилируются в GCC, то становится ясно, что компилятор GCC является наиболее важной частью современного программного обеспечения во всем мире. Конечно, существует большое количество алгоритмических языков, используемых для создания свободно распространяемых программ. Но большая часть компиляторов этих языков написаны и скомпилированы с использованием GCC. Так что, на том или ином уровне все свободно распространяемое программное обеспечение основывается на GCC. Некоторые компании, занимающиеся разработкой программ, прекращают поддержку собственных компиляторов и вместо них используют GCC. Его всегда можно получить, это просто и бесплатно. К тому же расширение и поддержка GCC не прекращается никогда.

С добавлением в семейство GCC языков *Java* и *Ada*, область применения свободно распространяемого компилятора GCC стала еще больше. Начиная с версии 3.1, количество компилируемых в нем языков достигло шести: *C*, *C++*, *Objective-C*, *Fortran*, *Java* и *Ada*. Пополнение семейства GCC будет продолжаться и дальше. Вскоре, очевидно, будет добавлен язык *COBOL* (после того как он будет обеспечен достаточной поддержкой).

Пройденный путь

Проект GNU был запущен в 1984 году в целях создания свободно распространяемой операционной системы. Основателем этого проекта стал Ричард Столмен (Richard Stallman), он же является и автором первоначальной версии GCC.

Начальный выпуск первой тестовой версии GCC — релиза 0.9, состоялся 22-го марта 1987 года. Первая действующая версия — 1.0 — появилась 23-го мая 1987 года. С тех пор вышло 108 релизов до выпуска 3.2 от 5-го мая 2002 года, на котором и основывается настоящая книга. Получается, что каждый новый выпуск GCC выходил в среднем один раз в 1,7 месяца.

Что в этой книге?

Эта книга содержит необходимую начальную информацию по применению GCC разработчиками программного обеспечения и руководителями проектов. Здесь много полезных сведений и для тех, кто собирается принять участие в поддержке и развитии самого компилятора. Но главная идея,ложенная в основу создания этого Руководства, — помочь читателю на этапах установки компилятора, его настройки и дальнейшего использовании для разработки своих программ. По любым меркам GCC — очень большая система. И, подобно многим большим программным системам, он содержит некоторые полезные свойства, которые пользователь может применить только когда он точно знает, как их использовать. Мы постарались дать вам достаточные представления о применении многих именно таких особенностей компилятора GCC. И в этом заключается главная цель настоящей книги.

Книга состоит из трех частей. Часть первая, "Свободно распространяемый компилятор" описывает замыслы, положенные в основу GCC, и содержит инструкции для его получения и установки в вашей системе. Часть вторая, "Использование Сборного Компилятора" содержит подробные инструкции по использованию GCC. Эта часть книги состоит из глав, каждая из которых посвящена отдельному языку программирования и содержит примеры программ. Специально отведенные главы описывают препроцессор и способы объединения в одной программе различных частей, написанных на разных языках программирования. Часть третья, "Внутренняя структура и окружение" состоит из глав, посвященных компоновке, отладке и перекрестной компиляции программ, сценариев компоновки и ассемблеру GNU. В этой части также содержатся сведения, раскрывающие тонкости взаимодействия верхнего и нижнего уровней компилятора.

GCC — чемпион мира по количеству доступных опций командной строки. Опции в алфавитном порядке представлены в Приложении Г. Приложение Б посвящено их взаимной совместимости. Глава 21 содержит дополнительные опции команд, ориентированные на выработку машинного кода для специфического оборудования, поддерживаемого компилятором.

Ниже представлено краткое описание каждой главы для того, чтобы дать более полное представление о темах, раскрываемых в книге:

- **Глава 1** является общим введением в замысел GCC, она содержит список частей компилятора и поддерживаемых им языков.
- **Глава 2** описывает процедуры получения и инсталляции GCC.
- **Глава 3** описывает работу препроцессора и способы его использования для обработки исходного кода программ.

- **Глава 4** содержит примеры компиляции и компоновки программ, написанных на языке *C*.
- **Глава 5** содержит примеры компиляции и компоновки программ на языке *C++*.
- **Глава 6** содержит примеры компиляции и компоновки программ на языке *Objective-C*.
- **Глава 7** содержит примеры компиляции и компоновки программ на языке *Fortran*.
- **Глава 8** содержит примеры компиляции и компоновки программ на языке *Java*.
- **Глава 9** содержит примеры компиляции и компоновки программ на языке *Ada*.
- **Глава 10** описывает способы сочетания исходного кода на двух языках программирования для получения одного готового к запуску файла.
- **Глава 11** поясняет применение средств GNU для интернационализации и локализации скомпилированных в GCC программ.
- **Глава 12** содержит примеры создания и применения статических и разделяемых объектных библиотек.
- **Глава 13**. Здесь поясняются основы использования отладчика GNU.
- **Глава 14** описывает использование `make` и других связанных с ней утилит.
- **Глава 15**. В этой главе обсуждается ассемблер GNU и поясняется, как вы можете его использовать с GCC.
- **Глава 16** описывает процесс конфигурирования GCC для компиляции и компоновки программ, предназначенных для другой машины.
- **Глава 17** объясняет, как можно использовать GCC для выработки кода, выполняемого во встраиваемых (*embedded*) системах.
- **Глава 18** содержит примеры генерации компилятором другой полезной информации, кроме объектного кода.
- **Глава 19** дает представление об использовании утилит `lex` и `yacc` для построения пользовательского верхнего уровня компилятора.
- **Глава 20** описывает промежуточный язык RTL, который вырабатывается верхним уровнем компилятора и используется для передачи кода на нижний уровень.
- **Глава 21** предоставляет список доступных опций командной строки для применения различных версий компилятора на различном оборудовании.
- **Приложение А** содержит копию Общественной Лицензии GNU (и ее перевод на русский язык).
- **Приложение Б** перечисляет переменные окружения, оказывающие влияние на работу GCC.
- **Приложение В** содержит справочник перекрестной совместимости опций командной строки по их категориям.
- **Приложение Г** содержит алфавитный список опций командной строки.
- **Словарь терминов** содержит словарь терминов и определений этой книги.

*Полное
руководство*



Часть I

**Свободно распространяемый
компилятор**



Глава 1

Введение в GCC

Сборный Компилятор *GNU* ("GNU compiler collection", сокращенно — "GCC") является важнейшей частью всего мирового программного обеспечения с открытым исходным кодом (open source software). Все остальные программы с открытым исходным кодом, на том или ином уровне основываются на нем. Даже другие языки программирования, такие как Perl и Python, написаны на языке C, и компилируются в GCC.

Компилятор GCC имеет весьма занимательную историю. Она — нечто большее, чем простое перечисление дат и событий. Эта часть программного обеспечения имеет наиболее фундаментальное значение для всего движения создания свободно распространяемых программ (free software). Создание системы Linux стало возможным единственно благодаря использованию при ее разработке компилятора GCC.

В этой вводной части предлагается обзор возможностей, которые включает в себя сборный компилятор *GNU*, здесь дается представление о составляющих частях его окружения. Кроме средств для собственно компиляции имеются также средства трассировки исходного кода и программы для редактирования файлов, программы управления процессом компиляции и средства предоставления отладочной информации.

Вводная часть завершается списком составляющих частей GCC и описанием его процессов. Список содержит описания файлов и программ, которые входят в состав Сборного Компилятора *GNU*. За этим списком следует пошаговое описание процессов преобразования исходных файлов в скомпонованную и готовую к запуску программу.

GNU

GCC является продуктом проекта *GNU*. Этот проект был начат в 1984 году с целью создания свободно распространяемой операционной системы, подобной UNIX.

("GNU — Not Unix".) Как и любой проект такого масштаба, он претерпел некоторые изгибы и прошел через кое-какие повороты, но цель все же была достигнута. Сейчас имеется действительно полнофункциональная UNIX-подобная операционная система, известная всему миру под названием Linux, успешно применяемая некоммерческими организациями, правительствами и частными лицами. И эта система вместе со всеми своими утилитами и приложениями основана на Сборном Компиляторе GNU.

Количество свободно распространяемых программ для Linux и других систем огромно и оно растет с каждым днем. Программное обеспечение, разработанное как часть общего проекта GNU в целях создания свободно распространяемой версии операционной системы UNIX, перечислено в "Каталоге свободно распространяемых программ" (Free Software Directory) по адресу <http://www.gnu.org/directory>.

Тысячи программистов участвовали в различных работах проекта GNU, так же как и в других проектах свободно распространяемого программного обеспечения, и практически все они на каком-либо уровне своей работы основывались на GCC.

Сравнение компиляторов

Компиляторы можно сравнивать по скорости компиляции, по скорости работы и по величине генерируемого кода. Вряд ли можно придумать еще какие-нибудь способы их померять. Конечно, придумать можно много разных характеристик, но придать им хоть какой-нибудь смысл — задача не всегда простая. Например, количество обрабатываемых исходных файлов различного типа (сборочных, конфигурационных файлов, включаемых заголовочных файлов, выполнимого кода, и т.п.) может превышать 15000. Преобразование исходных файлов в объектные файлы, библиотеки и выполнимые программы увеличивает это количество еще на несколько тысяч. Подсчет строк в исходном коде — т.е. количество строк текста в 15000 и более файлах — дает число, превышающее 3700000. Какие бы критерии мы не применили, следует признать, что речь идет о действительно больших программах.

Из-за привлечения в разработку такого большого количества программистов, качество кода лежит в широких пределах. Количество и качество внутренней документации также подвержено изменениям, ведь она в основном составлена из комментариев, включаемых в исходный код. К счастью, большинство программистов, работавших над кодом, постепенно улучшили и сам код и комментарии к нему. Вам не придется читать встроенные комментарии для того, чтобы использовать компилятор. Однако, если вы решите поработать над усовершенствованием самого компилятора, то, конечно, придется потратить какое-то время и на чтение включенных в исходный код комментариев.

Единственный путь по-настоящему оценить качество компилятора — узнать мнение о нем специалистов, которые его используют. Невозможно определить количество таких людей во всем мире (таково уж свойство свободно распространяемых программ), но это количество должно быть огромным. Этот компилятор применяется как основной на некоторых версиях UNIX, имеющих и свои довольно неплохие средства, поддерживаемые поставщиками. Мне известно, что даже один из крупных поставщиков UNIX, имеющий свой прекрасный компилятор, применяет GCC для многих своих "домашних" проектов.

Разработка компилятора не прекращается. Как описано во второй главе, вы можете устанавливать выпущенные версии GCC, "выкачивая" себе исходный код интересующего вас выпуска, или загружать последнюю (экспериментальную) версию. Экспериментальная версия обновляется раз в несколько минут — она изменяется непрерывно. Некоторые исправления поправляют обнаруженные ошибки, другие — добавляют новые языки и возможности, и некоторые — убирают более не применяемые средства. Если вам приходилось раньше работать с GCC, и через некоторое время возвращаетесь к нему снова, то вы непременно заметите некоторые изменения.

Опции командной строки

Каждая опция командной строки начинается со знака дефиса ("-", hyphen) либо пары дефисов. К примеру, следующая команда компилирует программу на языке *C* `muxit.c`, написанную в соответствии со стандартом ANSI и создает не скомпонованный объектный файл с именем `muxit.o`:

```
gcc -ansi -c muxit.c -o muxit.o
```

Опция, состоящая из одной буквы, за которой следует соответствующее имя, не обязательно должно включать пробел между буквой опции и параметром. Например, опция `-omuxit.o` тождественна `-o muxit.o`.

Следующая команда использует `-v` для вывода подробных описаний и `--help` для распечатывания списка доступных опций компилятора, по ней будет выведен исчерпывающий список опций командной строки, включающий и особые опции для каждого языка:

```
$ gcc -v --help
```

Возможно построение команды с таким набором опций, который не вызывает никаких действий. Например, следующая команда загружает в компилятор объектный файл и затем определяет `-c`, чтобы предотвратить задействование компоновщика:

```
$ gcc -c brookm.o
```

Все опции командной строки приблизительно укладываются в три категории:

- Специфичные к языку. Компилятор GCC способен компилировать несколько языков, и некоторые опции применяются только к одному или двум из них.
- Специфичные к платформе. Компилятор GCC способен генерировать объектный код для нескольких платформ, некоторые опции применяются лишь тогда, когда создается код для особой платформы. К примеру, если целевой платформой является Intel 386, то может быть использован набор опций `-fpr -ret -in -387` чтобы определить, что числа с плавающей точкой, возвращаемые вызываемыми функциями, должны сохраняться в аппаратных регистрах с плавающей точкой.
- Общие. Немало опций имеют общее значение для всех языков и всех платформ. Например, опция `-O` указывает компилятору оптимизировать выводимый объектный код.

Назначение опции, неизвестной компилятору, всегда будет приводить к сообщению об ошибке. Назначение опции, не соответствующей целевой платформе, также будет приводить к сообщению об ошибке.

Сама программа-драйвер `gcc` обрабатывает все известные ей опции и передает оставшиеся опции процессу, который компилирует особый язык. Если опция, переданная компилятору специфического языка, не известна ему, то будет выдано сообщение об ошибке.

Опции, указывающие `gcc` на выполнение только определенных действий, как то компоновка (linking) или предобработка (preprocessing), также означают, что остальные флаги, обычно имеющие значение, будут проигнорированы. Несмотря на применение опции `-w`, используемой для выработки дополнительных предупреждений, распознаваемые, но не применяемые флаги игнорируются.

Платформы

Сборный Компилятор GCC работает на многих платформах. Платформа (platform) здесь и далее понимается как сочетание определенного компьютерного процессора с запускаемой на нем определенной операционной системой.

Несмотря на то, что GCC переносится на тысячи таких аппаратно-системных сочетаний, лишь несколько основных платформ используются для тестирования правильности работы выпускаемых версий. Эти основные платформы, перечисленные в таблице 1.1, выбраны из-за их популярности, и еще потому, что они вполне представляют свойства остальных поддерживаемых GCC платформ.

Со всей возможной тщательностью разработчики добиваются правильной работы GCC на основных платформах, перечисленных в таблице 1.1. Немалое внимание также уделяется и вторичным платформам, перечисленным в таблице 1.2.

Таблица 1.1. Основные платформы тестирования GCC

<i>Процессор</i>	<i>Операционная система</i>
Alpha	Red Hat Linux 7.1
HPPA	HPUX 11.0
Intel x86	Debian Linux 2.2, Red Hat Linux 6.2 и FreeBSD 4.5
MIPS	IRIX 6.5
PowerPC	AIX 4.3.3
Sparc	Solaris 2.7

Таблица 1.2. Вторичные платформы тестирования GCC

<i>Процессор</i>	<i>Операционная система</i>
PowerPC	Linux
Sparc	Linux
ARM	Linux
Intel x86	MS Win32 + Cygwin

Причиной первичного и вторичного тестирования на таком ограниченном количестве платформ является недостаток человеческих ресурсов. Несмотря на то, что ваша платформа может быть не представлена в этих перечнях, вы вполне можете расчитывать на то, что компилятор прекрасно пойдет и на вашей системе. Полный набор средств для тестирования компилятора находится в его комплекте с исходным кодом. С его помощью вы можете проверить, насколько правильно работает у вас GCC. Другой подход состоит в том, чтобы в качестве добровольца тестировать на своей системе каждую очередную версию компилятора до ее выпуска.

Что делает компилятор

Компилятор действует как переводчик. Он читает набор инструкций, записанных в одной форме (обычно — текст на языке программирования), и переводит его в набор инструкций (обычно — последовательность машинных команд в двоичном формате), выполняемых компьютером.

Грубо говоря, Компилятор можно разделить на две части: верхний и нижний его уровни. *Верхний уровень компилятора* (front end) читает исходный код программы и преобразует все, что имеет соответствия в таблице, резидентно загруженной в память, в некоторую древовидную структуру. Когда же это дерево построено, *нижний уровень компилятора* (back end) читает информацию, записанную в этой передаваемой ему древовидной структуре, и преобразует ее в код на ассемблерном языке, соответствующем целевой машине.

Далее представлена взятая, так сказать, "с высоты птичьего полета" последовательность шагов, предпринимаемых для преобразования вашего исходника в готовую для выполнения компьютером программу:

- В самом начале *верхнего уровня* компилятора находится *лексический анализ* (иначе говоря, разложение текста по грамматическим правилам для транслируемого исходного языка). При этомчитываются знаки из входного потока и по тому, как они сгруппированы, определяются составляемые ими программные символы, числа и пунктуация.
- Процесс *синтаксического разбора* (parsing process) читает поток символов (лексем), передаваемый ему лексическим сканером, и, следуя своему набору правил, определяет отношения между ними.
- Разделенная древовидная структура переводится на псевдо-ассемблерный язык, называемый *Языком Регистрового Переноса* (Register Transfer Language, RTL).
- Работа *нижнего уровня* компилятора начинается с анализа кода на языке RTL и выполнения некоторых действий по его оптимизации. При этом убираются избыточные и не используемые секции кода. Некоторые части кода могут быть перемещены по дереву в другое местоположение, чтобы предотвратить выполнение соответствующего набора инструкций большее число раз, чем это необходимо. В целом проводится больше дюжины различных оптимизаций, и некоторые из них проходят по обрабатываемому коду несколько раз.

- *RTL*-код, то есть код на Языке Регистрового Переноса переводится на ассемблерный язык целевой машины.
- Задействуется ассемблер, который транслирует ассемблерный код в объектный файл. Этот файл еще не имеет выполнимого формата — он содержит исполняемый объектный код, но еще не в той форме, которая годится для его запуска на выполнение машиной. Кроме того, он может содержать неразрешимые ссылки на подпрограммы (*routines*) и данные, находящиеся в других модулях.
- *Компоновщик* (*linker*) связывает и объединяет ассемблированные объектные файлы (некоторые из них могут храниться в библиотеках объектного кода) в выполнимую машиной программу.

Обратите внимание на полное разделение верхнего и нижнего уровней компилятора. Любой язык, обеспеченный *синтаксическим разделителем* (*парсером*), может быть использован для создания компилируемой нижним уровнем GCC древовидной структуры. Кроме того, любая машина, обеспеченная программой, которая транслирует промежуточную древовидную структуру GCC в ассемблерный код, способна компилировать программы с любого из языков, поддерживаемого верхним уровнем компилятора.

На самом деле кухня GCC работает не так просто, как представлено вам в этом упрощенном описании. Но главное — это то, что она вполне справляется со своей работой.

ЯЗЫКИ ПРОГРАММИРОВАНИЯ

GCC компилирует исходный код на нескольких языках программирования. Между этими языками существуют глубинные внутренние взаимоотношения. Процедуры синтаксического разбора, или *парсеры* (*parsers*) могут значительно отличаться друг от друга, ведь набор правил для каждого языка уникален, но с каждым шагом компиляционного процесса все большая часть кода преобразуется к подобным базовым структурам. Как уже объяснялось в предыдущих разделах, Сборный Компилятор GNU способен обрабатывать входной поток в форме любого из нескольких поддерживаемых языков программирования и вырабатывать, выполнимый код для ряда различных поддерживающих платформ.

ЯЗЫК ПРОГРАММИРОВАНИЯ С – ФУНДАМЕНТАЛЬНЫЙ ЯЗЫК GCC

Основным языком, поддерживаемым GCC, является алгоритмический язык С. Вся эта система начиналась как компилятор C, и со временем к ней добавлялись другие языки. Это сложилось весьма удачно, потому что C — язык системного уровня, способный оперировать с простейшими элементами компьютерных программ, которые относительно упрощают построение на их основе внутренних структур других компиляторов.

Если вы программируете на любом отличном от *C* языке, то как только вы ближе узнаете GCC, то обнаружите, что многие привычные для вас понятия имеются и в языке *C*. С некоторым допущением можно утверждать, что язык *C* является чем-то вроде ассемблера высокого уровня, главным механизмом GCC. Кстати, большая часть компиляторов была создана именно на языке *C*.

Первым добавлением был язык *C++*

Язык программирования *C++* является прямым расширением (с небольшими модификациями) языка программирования *C*. Поэтому он стал превосходным кандидатом для первого расширения GCC. Все, что делается в *C++*, возможно и в *C*, поэтому не возникло потребности в изменении базового нижнего уровня компилятора. Понадобилось только добавить в верхний уровень новый синтаксический разделитель (parser) и семантический анализатор. Раз уж промежуточную структуру можно сгенерировать в том же виде, то остальная часть компилятора может оставаться той же, что и для компилятора *C*.

Язык *Objective-C*

Objective-C, не являясь столь же популярным языком, как *C* и *C++*, стал следующим удобным кандидатом для расширения компилятора GCC. Он происходит от языка *C* и основывается на нем. Его называют "языком *C* с объектами", и если вам приходилось его изучать, то вы хорошо понимаете, что при этом имеется в виду. В большинстве случаев можно написать всю программу на языке *C*, скомпилировать ее в *Objective-C*, и не почувствовать при этом никакой разницы. Особый синтаксис, явным образом отличающий этот язык от *C*, используется для определения объектных структур, поэтому не возникает ни путаницы ни конфликтов с любыми частями кода на чистом *C*.

Добавление языка *Fortran*

Язык *Fortran* умеет делать кое-что такое, чего не умеет *C*, а именно — точные математические расчеты. Стандартная библиотека функций *Fortran*, известная также как набор встроенных функций языка "Fortran intrinsics" (встроенные функции действуют как часть самого языка), имеет довольно сильные и глубокие возможности, эта библиотека совершенствовалась и расширялась в течение многих лет. Сейчас *Fortran* широко используется для серьезных научных расчетов благодаря своим изначальным возможностям быстро и точно производить сложные вычисления. *Fortran* даже имеет в своем наборе простейших типов данных комплексные числа. И все простейшие типы могут объявляться с очень высокой точностью.

Сама структура языка несколько громоздка и представляется довольно устаревшей по сравнению с гибкостью современных языков, но она все же включает объявляемые процедуры, подпрограммы и функции, необходимые для структурного программирования. Познейший стандарт языка *Fortran* расширяет эти возможности до уровня вполне современного языка.

Добавление языка Java

Самым молодым языком, включенным в GCC, является *Java*. Подобно C++, *Java* основывается на языке *C*, однако в нем применен несколько иной подход к синтаксису описания классов. Хотя в этом вопросе C++ более гибок, в *Java* применением строго недвусмысленных форм описания исключена свойственная C++ некоторая "неоднозначность" структур конструкторов, деструкторов и наследования.

Java весьма отличается от других включенных в GCC языков благодаря особым формам представления объектного кода. Исходный код на языке *Java* компилируется в объектный код особого формата, известный как "байтовый код", или байт-код (bytecodes), который выполняется интерпретатором, известным под названием *Виртуальная Машина Java* (*Java Virtual Machine, JVM*). Все программы, написанные на *Java*, могут выполняться этим способом. Но компилятор GCC имеет дополнительные опции, которые позволяют вырабатывать системно-ориентированный выполнимый машиной код (native code). При этом подключается расширение *верхнего уровня* GCC для языка *Java* к существующему общему *нижнему уровню* GCC и генерирует объектный код, ориентированный на целевую платформу. В дополнение к этому создано расширение *верхнего уровня*, которое способно считывать байтовый код, пригодный для выполнения в Виртуальной Машине *Java*, в качестве исходного для последующей выработки из него ориентированного на нужную платформу исполнимого двоичного кода.

Добавление языка Ada

Новейшее дополнение семейства GCC — язык *Ada*. Он был добавлен как полнофункциональный компилятор "GNAT Ada" отдельной оригинальной разработки компании "Ada Core Technologies" и бесплатно передан проекту GCC в октябре 2001 года.

Верхний уровень компилятора *Ada* отличается от других расширений GCC, он сам написан на языке *Ada*. Это очень удобно, когда вы применяете некоторую своеобразную разновидность компилятора *Ada*, установленную в вашей системе. Однако, следует иметь в виду, что при этом на ряде систем требуется применение специальной установочной процедуры. Все остальные подключаемые языки написаны на языке *C* или *C++*, поэтому применение GCC в таких случаях позволяет достичь почти идеальной *переносимости* программных работ с одной платформы на другую.

Язык *Ada* разработан специально для задействования разных программистов в разработке больших программ. Когда компилируется любая часть программы на языке *Ada*, то присходит перекрестная сверка на корректность с другими частями программы. Правила синтаксиса этого языка требует объявления функций и процедур вместе с их принадлежностью к пакету. При компиляции происходит проверка модулей на соответствие конфигурации указанного пакета. В языках *C* и *C++* используются прототипы для объявления локально адресуемых функций, а в *Java* для этих целей применяются соглашения об именах файлов. Однако ни один из подобных языков не применяет настолько строгих способов согласования, как *Ada*.

Язык *Chill* покинул семейство GCC

Начиная с версии 3.0, в GCC прекращена поддержка языка программирования *Chill*. К выходу версии 3.1 из GCC был исключен весь исходный код, связанный с компиляцией этого языка. Однако компилятор GCC очень сложен и поддержка им языка *Chill* имеет долгую историю, так что еще некоторое время можно будет видеть в исходном коде и в документации комментарии, связанные с *Chill*. Эта книга написана в переходный период отказа от поддержки языка *Chill*, поэтому и в ней имеются указания на опции компилятора и файловые типы для этого языка.

Список частей компилятора GCC

Компилятор GCC состоит из многих компонентов. В таблице 1.3 представлен список основных частей GCC, однако вам следует знать о том, что не все из них всегда присутствуют. Некоторые из них специфичны к языку программирования, поэтому, если поддержка для соответствующего им языка у вас не установлена, то вы и не увидите их в вашей системе.

Таблица 1.3. Различные устанавливаемые части компилятора GCC

Часть	Описание
c++	Версия <code>gcc</code> , устанавливающая C++ в качестве языка по умолчанию и автоматически подключающая стандартные библиотеки C++ при компоновке. То же, что и <code>g++</code> .
cc1	Действующий компилятор C.
cc1plus	Действующий компилятор C++.
collect2	На системах, не использующих компоновщик GNU, необходимо запускать <code>collect2</code> для генерирования определенного кода инициализации среды Окружения (global initialization code), подобно конструкторам и деструкторам в языке C.
configure	Выполнимый сценарий, находящийся в корневом каталоге дерева исходных файлов GCC. Используется для установки значений конфигурационных величин и создания компоновочных файлов (makefiles), необходимых для компиляции GCC.
crt0.o	Код инициализации и завершения, особый для каждой системы, компилируется в этот файл. Он затем компонуется в каждый исполняемый файл для обеспечения необходимых начальных и завершающих действий.
cygwin1.dll	Разделяемая библиотека для Microsoft Windows, которая обеспечивает интерфейс API (Application Programming Interface), эмулирующий системные вызовы UNIX.
f77	Драйверная программа, используемая для компиляции программ на языке Fortran.
f771	Действующий компилятор Fortran.
g++	Версия <code>gcc</code> , устанавливающая C++ в качестве языка по умолчанию и при компоновке автоматически подключающая стандартные библиотеки C++. То же, что и <code>c++</code> .
gcc	Основная драйверная программа, координирующая выполнение компиляторов и компоновщиков (linkers) для выработки желаемого вывода. Поддерживает большое количество опций.

Часть	Описание
<code>gcj</code>	Драйверная программа, используемая для компиляции программ на языке <i>Java</i> .
<code>gnat1</code>	Действующий компилятор <i>Ada</i> .
<code>gnatbind</code>	Утилита, используемая для связывания (подшивки) пакетов кода на языке <i>Ada</i> (<i>binding</i>). <i>Биндер Ada</i> .
<code>gnatlink</code>	Утилита, используемая для компоновки (<i>linking</i>) программ на языке <i>Ada</i> .
<code>jc1</code>	Действующий компилятор <i>Java</i> .
<code>libgcc</code>	Эта библиотека содержит функциональные блоки кода (<i>routines</i>), которые могут считаться частью компилятора, т.к. они компонуются практически в каждую исполняемую программу. Это специальные подпрограммы (<i>routines</i>), компонуемые в выполнимую программу для выполнения таких коренных задач, как арифметические действия над числами с плавающей точкой и т.п. Эти функциональные блоки зачастую зависят от платформы.
<code>libgcj</code>	Динамическая библиотека (содержит подпрограммы, компонуемые к программе на этапе ее выполнения, <i>Real Time Library, RTL</i>), содержащая классы ядра <i>Java</i> .
<code>Libobjc</code>	Динамическая библиотека (<i>RTL</i>), для всех программ на языке <i>Objective-C</i> .
<code>libstdc++</code>	Динамическая библиотека (<i>RTL</i>), содержащая классы и функции <i>C++</i> , определенные как часть стандарта языка.

В таблице 1.4 перечислено программное обеспечение, работающее в одной упаковке с `gcc` для обеспечения процесса компиляции. Некоторые совершенно необходимы (такие, как `as` и `ld`), другие при всей своей полезности не строго обязательны. Несмотря на то, что эти средства доступны и включены в наборы стандартных утилит систем *UNIX*, вы можете многие из них получить в пакете *GNU*, называемом `binutils`. Процедура установки (инсталляции) пакета утилит `binutils` подробно описана в главе 2.

Таблица 1.4. Программные средства, используемые совместно с *GCC*

Средство	Описание
<code>addr2line</code>	Предоставляя адреса, находящиеся в исполняемом файле, <code>addr2line</code> использует отладочную информацию, находящуюся в файле, для преобразования адресов в имя исходного файла и номер строки. Эта программа входит в пакет <code>binutils</code> .
<code>ar</code>	Программа для поддержки библиотек. Добавляет, удаляет и извлекает файлы из архивов. Наиболее часто используется для создания и обслуживания объектных библиотек, используемых компоновщиком. Входит в пакет <code>binutils</code> .
<code>as</code>	Ассемблер <i>GNU</i> . В действительности представляет собой целое семейство ассемблеров, он способен работать с одной из нескольких различных платформ.
<code>autoconf</code>	Вырабатывает сценарии для программного окружения, которые автоматически конфигурируют пакеты исходного кода так, чтобы они могли компилироваться в назначенней версии <i>UNIX</i> .
<code>c++filt</code>	Программа, воспринимающая сжатые (<i>mangled</i>) компилятором <i>C++</i> имена. Имена сжимаются при их замещении, перегрузке (<i>overloading</i>). Программа <code>c++filt</code> преобразовывает эти замененные имена в их исходную форму. Входит в пакет <code>binutils</code> .
<code>f2c</code>	Программа-транслятор с языка <i>Fortran</i> на язык <i>C</i> . В пакет <code>binutils</code> не входит.

Средство	Описание
<code>gcov</code>	Профилирующее средство, используемое совместно с <code>gprof</code> для определения участков программы, отнимающих наибольшую часть времени выполнения программы.
<code>gdb</code>	Отладчик (debugger) GNU. Может использоваться для проверки действий и значений величин во время выполнения программы.
<code>GNATS</code>	Система трассировки (<i>трекинга</i>) ошибок, применяемая в GNU. Встроенная система для поиска ошибок компилятора <code>gcc</code> и другого программного обеспечения GNU.
<code>gprof</code>	Это средство обеспечивает наблюдение за выполнением программы, скомпилированной со встроенным в нее профилирующим кодом. Вырабатывает профиль для оптимизации, сообщая время выполнения каждой функции. Входит в пакет <code>binutils</code> .
<code>ld</code>	Редактор компоновочных связей (компоновщик) GNU. Собирает объектные файлы в готовую выполнимую программу. Входит в пакет <code>binutils</code> .
<code>libtool</code>	Сценарий поддержки базовой библиотеки, используемый в компоновочных файлах (<i>makefiles</i>) для упрощения использования <i>разделяемых динамических библиотек</i> (<i>shared libraries</i>).
<code>make</code>	Утилита, читающая компоновочный сценарий (<i>makefile script</i>) для определения частей программы, требующих компиляции и компоновки, и затем выдающая команды, необходимые для таких действий. Она считывает сценарий, называемый <i>компонентовочным файлом</i> (<code>makefile</code> или <code>Makefile</code>), содержащий определения отношений и зависимостей файлов.
<code>nlmconv</code>	Преобразовывает переносимый объектный файл в загружаемый модуль системы NetWare (NetWare Loadable Module, NLM). Входит в пакет <code>binutils</code> .
<code>nm</code>	Перечисляет список символических имен, назначаемых в объектном файле. Входит в пакет <code>binutils</code> .
<code>objcopy</code>	Транслирует объектные файлы из одного двоичного формата в другой. Входит в пакет <code>binutils</code> .
<code>objdump</code>	Показывает различного типа информацию, содержащуюся внутри одного или нескольких объектных файлов. Входит в пакет <code>binutils</code> .
<code>ranlib</code>	Создает и добавляет индексы в архивный файл типа <code>.a</code> . То есть, именно те индексы, которые используются программой <code>ld</code> для поиска модулей в библиотеках. Входит в пакет <code>binutils</code> .
<code>ratfor</code>	Препроцессор Ratfor (Rational Fortran). Он может вызываться из GCC, но в пакет <code>binutils</code> не входит.
<code>readelf</code>	Показывает информацию, содержащуюся в объектном файле формата ELF. Входит в пакет <code>binutils</code> .
<code>size</code>	Перечисляет имена и размер всех разделов (секций) объектного файла. Входит в пакет <code>binutils</code> .
<code>strings</code>	Считывает файл любого типа и извлекает из него все определяемые строки буквенных символов (<i>character strings</i>). Входит в пакет <code>binutils</code> .
<code>strip</code>	Убирает символьные таблицы (<i>symbol tables</i>) вместе со всей прочей отладочной информацией из объектного файла или архивной библиотеки. Входит в пакет <code>binutils</code> .
<code>vprof</code>	Просмотрщик Ratfor (Rational Fortran), представляющий текстовый файл в виде графа. Эта утилита не предоставляется как часть GCC. Однако, в GCC предусмотрена особая опция <code>-dv</code> для генерирования данных для оптимизации в формате, поддерживаемом программой <code>vprof</code> .
<code>windres</code>	Компилятор для файлов ресурсов окон (Window Resource Files). Входит в пакет <code>binutils</code> .

Контакты

Домашняя страница проекта GNU находится в сети Интернет по адресу — <http://www.gnu.org>, и домашняя страница GCC — <http://www.gnu.org>.

Компилятор GCC может быть представлен довольно разнообразно — от простых утилит для конвейерной загрузки (*batch utility programs*) до систем с исходным кодом в несколько миллионов строк. В общем случае при существенном увеличении программного проекта или его специализации любого рода, возникают ситуации, при которых обнаруживаются необычные проблемы. Иногда это связано с ошибками, а иногда с различного рода недоразумениями, но обязательно приходит время, когда становится необходимым кое-что выяснить или, по крайней мере, посоветоваться и получить поддержку. К счастью, помочь вполне доступна и вы можете узнать о GCC все что пожелаете.

Основным источником информации являются почтовые рассылки. Все открытые почтовые рассылки (т.е. такие, в которых подписчики могут как получать, так и отправлять сообщения — по типу конференций) имеют преимущество скорости получения нужной информации и удобны для ведения обсуждений. Если вам нужна именно помочь, то я вам предлагаю подписаться на рассылку `gcc-help`. Обсуждение в открытой почтовой рассылке может продолжаться до полного выяснения ситуации и решения проблемы. Таблица 1.5 содержит краткое описание всех открытых почтовых рассылок GCC. В таблице 1.6 перечислены все закрытые рассылки, то есть такие в которых не могут быть помещены письма обычных получателей.

Таблица 1.5. Открытые почтовые рассылки GCC

Название подписки	Описание
<code>gcc</code>	Общая область обсуждений, связанная с разработкой GCC. Содержит ряд тем, на которые можно подписаться дополнительно. Помогает узнать последние новости и получить последние разработки, понять "чем дышат" разработчики. По этой рассылке идет большой объем сообщений.
<code>gcc-bugs</code>	Сообщения об обнаруженных ошибках, обсуждение различных "багов". Большой объем сообщений.
<code>gcc-help</code>	Подписка для тех, кто ищет ответы на свои вопросы. Большой объем сообщений.
<code>gcc-patches</code>	Исправления исходного кода, обсуждение рассылаемых с сообщениями "патчей". Большой объем сообщений.
<code>gcc-testresults</code>	Сообщения о результатах различных испытаний, обсуждение способов тестирования и посылаемых с сообщениями результатов тестов.
<code>java</code>	Обсуждение, связанное с разработкой верхнего уровня (<i>front end</i>) компилятора GCC для языка Java, и с динамическими библиотеками (<i>Real Time Libraries, RTL</i>) для Java-программ.
<code>java-patches</code>	Рассылка исправлений исходного кода верхнего уровня компилятора GCC для языка Java, и кода динамических библиотек Java. Обсуждение рассылаемых "патчей" и дополнений.
<code>libstdc++</code>	Обсуждение, связанные с разработкой и поддержкой стандартных библиотек языка C++.

Таблица 1.6. Почтовые рассылки GCC "только для чтения" (Read-Only)

Название подписки	Описание
gccadmin	Подписка на сообщения, посылаемые с административного аккаунта <code>gcc.gnu.org</code> .
gcc-announce	Рассылка небольшого объема, объявления о новых версиях и выпусках, а также других важных событиях GCC.
gcc-cvs	По этой подписке приходят сообщения о каждой регистрации новой документации в репозитории <i>CVS</i> .
gcc-cvs-wwwdocs	По этой подписке приходят сообщения о каждой регистрации новой документации в формате HTML в репозитории <i>CVS</i> .
gcc-prs	Сообщения о каждой обнаруженной ошибке в базе данных <i>GNATS</i> .
gcc-regression	Результаты тестирования GCC на поддержку предыдущих версий.
java-announce	Рассылка небольшого объема. Новости о верхнем уровне компилятора GCC для языка Java, и о динамических RTL-библиотеках Java.
java-cvs	В эту рассылку (и также в <code>gcc-prs</code>) направляются сообщения о каждой регистрации новой документации по компилятору Java и по динамическим RTL-библиотекам Java в репозитории <i>CVS</i> .
java-prs	В эту рассылку (и также в <code>gcc-prs</code>) направляются сообщения каждый раз при возникновении новых сообщений в базе данных <i>GNATS</i> о проблемах, связанных с языком Java.
Libstdc++-cvs	Сообщения о каждой регистрации новых добавлений документации по <code>libstdc++</code> в репозиторий <i>CVS</i> .

Все подписки доступны на web-сайте <http://www.gnu.org/software/gcc/lists.html>. На этой странице можно как подписаться на новую рассылку, так и отказаться от уже получаемой. Также каждая рассылка имеет свой собственный сайт, на котором возможен поиск и просмотр информации в архивах сообщений. Имена таких сайтов составляются следующим образом: в начале указывается `gcc.gnu.org/ml/` и далее следует название нужной рассылки. Например, чтобы найти web-сайт с архивом подписки `gcc-announce`, зайдите на <http://gcc.gnu.org/ml/gcc-announce>.

Глава 2



Получение и установка компилиатора GCC

Несмотря на доступность любой скомпилированной и готовой к запуску версии GCC, наиболее употребляемым способом установки является загрузка его исходного кода и последующая компиляция. Сам процесс компиляции GCC вполне устоялся, поскольку он совершенствовался в течение ряда лет. Тот же основной процесс инсталляции используется для установки всего программного обеспечения GNU. Упрощенно его можно представить следующей последовательностью действий:

- Загрузка исходного кода и сохранение его в отдельном каталоге.
- Создание отдельного рабочего каталога, используемого для компиляции исходных файлов.
- Выполнение из рабочего каталога сценария конфигурации `configure`, который создает дерево каталогов, содержащее подборку файлов. Эти файлы, зависимые от аппаратно-операционной платформы, управляют процессом компиляции.
- Выполнение команды `make` для компиляции исходных файлов в объектный код.
- Выполнение команды `make install` для установки свежих скомпилированных программ и библиотек в вашей системе.

Существует два пути получения исходного кода: вы можете получать сжатые подборки файлов в архивах формата `tar` по протоколу FTP, либо можете получать отдельные файлы, используя систему *CVS* (Concurrent Version System). По FTP вы можете получать выпущенные и устоявшиеся версии компилятора. А использование CVS предоставляет доступ как к выпущенным, так и к текущей экспериментальной

версии. Форма FTP более приспособлена для пользователей компилятора, в то время как форма CVS предназначена для тех, кто занимается разработкой и поддержкой GCC. Однако сама процедура установки для обоих способов мало отличается.

Если на компьютере еще не установлено программное обеспечение GNU, то, возможно, что вы сочтете необходимым вначале установить пакет **binutils**. Этот пакет включает программные утилиты, используемые GCC вместе с *ассемблером* и *компоновщиком* (linker), специально разработанные для непосредственной работы с GCC. Конечно, существует возможность использования ассемблера и компоновщика, комплектных установленной у вас системе, однако ассемблер и компоновщик GNU специально приспособлены для работы с компиляторами GNU. Процедура их инсталляции в основном такая же, как и для GCC, разница только в том, что процесс установки **binutils** легко осуществим на машине, еще не имеющей установленного пакета **binutils**, в то время как установка GCC требует наличия утилит установленного набора **binutils**.

Как и почти все программное обеспечение GNU, компилятор написан на языке C. Поэтому для его компиляции требуется наличие уже установленного на машине компилятора C. Если же на машине его нет, то для компиляции устанавливаемого компилятора GNU нужно перекрестно скомпилировать этот компилятор на другой машине специально для этого сконфигурированным и скомпилированным компилятором. (Неплохо сказано!) Процедура компиляции на другой машине подробно объясняется в главе 16.

Загрузка готовой к запуску скомпилированной версии

Если у вас нет установленного компилятора C, то вы можете сделать одно из двух: загрузить исходный код на другой компьютер, имеющий компилятор C, и перекрестно скомпилировать полученную версию для целевой машины; либо можете загрузить уже готовую прекомпилированную версию. Сама группа GNU не предоставляет прекомпилированных версий компилятора GCC, но некоторые версии доступны из других источников. Существует огромное разнообразие сочетаний аппаратных платформ и операционных систем, для которых можно найти готовый машинный код компилятора, в таблице 2.1 представлен список наиболее распространенных, и потому — наиболее доступных из них.

Таблица 2.1. Прекомпилированные версии GCC

Платформа	Наименование и расположение
AIX	"Бычий" большой архив свободно распространяемого и условно-бесплатного программного обеспечения для AIX. http://freeware.bull.net Библиотека общественных программ домена Университета Южной Калифорнии (University of Southern California). http://aixpdslib.seas.ucla.edu
DOS	"DJGPP". http://www.delorie.com/djgpp

Платформа	Наименование и расположение
HP-UX	Центр компьютерных технологий Университета штата Висконсин (University of Wisconsin). http://hpx.cae.wisc.edu Центр архивов и переносимых программ для HP-UX в штате Юта. http://hpx.cs.utah.org Центр архивов и переносимых программ для HP-UX в Великобритании. http://hpx.connect.org.uk Официальный сайт компании Sun Microsystems в Центральной Европе. ftp://sunsite.informatic.rwth-aachen.de/pub/packages/gcc_hpx
Solaris 2	Проект свободного программного обеспечения для систем Solaris (как на базе Intel, так и на Spark). http://www.sunfreeware.com
SGI	Свободное программное обеспечение для SGI. http://freeware.sgi.com
UnixWare	"Скунсовый" сайт программ (Skunkware). ftp://ftp2.caldera.com/pub/skunkware/w7/Packages
Windows	Сайт проекта Cygwin. http://sources.redhat.com/cygwin

Каждый из представленных в таблице сайтов предлагает подробные указания по загрузке и установке. Компилятор GCC является *переносимым* (т.е. его можно применять на различных платформах), однако первоначально планировалась его переносимость только между операционными системами UNIX. DOS-версия компилятора представляет собой простейшее средство переноса на эту платформу программ и для запуска требует только загрузки на машину, работающую под системой *DOS*. Правда, эта версия ограничена возможностью компиляции только языков *C* и *C++*. Компилятор для Microsoft Windows — *Cygwin* (*Cygwin Project*), является полнокомплектным средством переноса программ, включающим не только компилятор, но и набор утилит, обеспечивающий полную эмуляцию рабочего окружения UNIX.

Загрузка исходного Кода по FTP

Целый ряд сайтов предоставляет анонимный FTP-доступ к исходному коду GCC. Есть возможность скачивания полного сборного компилятора GNU либо выбора отдельного языка программирования (или нескольких), который вы желаете установить. Файлы перечислены в таблице 2.2, хотя и нет необходимости устанавливать все из них. Вы можете выбирать из двух вариантов:

- загрузить только ядро компилятора и затем выбрать любой из языков, с которым будет работать ядро;
- загрузить компилятор полностью, что равнозначно загрузке ядра, всех языков и проверочного пакета.

Проверочный пакет устанавливать не обязательно. Он состоит из исходных текстов программ, которые используются для оценки правильности работы загруженного и скомпилированного вами GCC.

Таблица 2.2. FTP-файлы, содержащие исходный код GCC

Имя файла	Содержимое
gcc-3.1.tar.gz	Весь компилятор, включая ядро и все компоненты.
gcc-ada-3.1.tar.gz	Компилятор <i>Ada</i> .
gcc-core-3.1.tar.gz	Ядро сборного компилятора GNU, включающее компилятор C и общие для всех компиляторов модули.
gcc-g++-3.1.tar.gz	Компилятор C++.
gcc-g77-3.1.tar.gz	Компилятор <i>Fortran</i> .
gcc-java-3.1.tar.gz	Компилятор <i>Java</i> .
gcc-objc-3.1.tar.gz	Компилятор <i>Objective-C</i> .
gcc-testsuite-3.1.tar.gz	Проверочный набор.

Далее описывается последовательность действий, предпринимаемых для загрузки исходного кода, подготовки его к компиляции и последующей инсталляции из него GCC:

1. Выберите FTP-сайт. FTP-сайт GNU находится по адресу URL <ftp://ftp.gnu.org/gnu>, однако его зеркалируют тысячи сайтов по всему миру, и, возможно, что вам будет удобнее воспользоваться одним из них. Текущий список сайтов-зеркал представлен на <http://www.gnu.org/order/ftp.html>. Для беспроблемной загрузки лучше выбрать ближайший к вам сайт.
2. Загрузите файлы в рабочий каталог. Это может быть тот же каталог, из которого вы будете компилировать GCC. Хотя обычно создают отдельный временный каталог для загрузки этих файлов, чтобы сохранить их, поскольку в ходе инсталляции исходные файлы могут быть удалены после извлечения из них исходного кода. Необходимо устанавливать параметры загрузки FTP на загрузку бинарных файлов, а не текстовых. Это сжатые файлы, и загрузка в режиме для текстовых файлов им повредит из-за неправильной интерпретации их содержимого и преобразования некоторых байтовых сочетаний в знаки кодировки ASCII.
3. Выберите или создайте каталог, который будет использоваться как корневой для дерева исходных каталогов, создаваемого при распаковке полученных файлов таким образом, чтобы вы могли выбирать между инсталляционными каталогами различных вариантов и версий. Например, если вы выбираете установку исходных файлов в расположение с именем `/usr/local/src/`, то при распаковке дерево исходных каталогов GCC будет создано в каталоге `/usr/local/src/gcc-3.1`.
4. Распакуйте файлы. Если ваш архиватор `tar` поддерживает формат `gzip` (параметр `z`), то можете распаковать так (предполагается что вы скачали файл `gcc-core-3.1.0.tar.gz` в расположение `/tmp/download`):

```
cd /usr/local/src
tar -xvzf /tmp/download/gcc-core-3.1.0.tar.gz
```
5. Если ваш архиватор `tar` не поддерживает формат `gzip`, то следует добавить еще одну команду:

```
cd /usr/local/src  
gunzip /tmp/download/gcc-core-3.1.0.tar.gz  
tar -xvzf /tmp/download/gcc-core-3.1.0.tar
```

Таким способом будет создан каталог `/usr/local/src/gcc-3.1`, содержащий исходники. Если вы решили загрузить несколько файлов, то вам нужно применить указанную процедуру к каждому из них. Если вдруг окажется, что у вас нет `gunzip`, то вы можете получить его готовую к запуску версию, соответствующую вашей системе, по адресу <http://www.gzip.org>.

Загрузка исходного кода через систему CVS

В некотором отношении загрузка через *Систему Конкурирующих Версий* (Concurrent Versions System, *CVS*) проще, чем скачивание по FTP. Эта система позволяет загружать различные версии GCC. Система CVS используется разработчиками программного обеспечения GNU для получения последних испытательных версий и позволяет им быть в курсе последних обновлений. Являясь архивом исходного кода, CVS предоставляет вам возможность получения любой версии компилятора, включая находящуюся в стадии разработки.

Существуют некоторые различия между загрузкой исходных файлов в архивах формата `tar` и загрузкой исходных файлов через CVS. Для компиляции из исходников с помощью CVS требуется разделитель `Bison` (Bison parser) и установленная утилита `Texinfo` четвертой или более поздней версии для выработки некоторых промежуточных файлов. Эти промежуточные файлы содержатся в уже сгенерированном виде в архивах `tar`, но не поставляются с файлами CVS. Другое отличие состоит в том, что некоторые конфигурационные параметры имеют значения по умолчанию, установленные так, чтобы обеспечить наибольшие возможности диагностики при загрузке по CVS.

Хранилище, или *репозиторий CVS* отслеживает каждое изменение, внесенное в исходный код. Когда приходит время выпуска готовой версии, создается *тэг*, отмечающий этот выпуск (версию). Тэг связан с текущей или выбранной версией каждого модуля в хранилище. Когда вы желаете загрузить какую-либо версию компилятора, то указываете имя тэга своей локальной утилите `cvs`, и она получает для вас все исходные файлы заказанной версии. Исходные файлы загружаются в сжатом виде (если вами указана соответствующая опция в команде), распаковываются и сохраняются в соответствующем каталоге по их прибытию. В результате вы получаете тот же набор каталогов и файлов, что и при распаковке архивов, полученных по FTP.

Далее описана последовательность действий, которой вы должны следовать при получении желаемой версии компилятора GCC:

1. Убедитесь, что в вашей системе установлена утилита `cvs`, введя команду:

```
cvs -v
```

По ней будет выведен номер версии `cvs` вместе с некоторой дополнительной информацией. Если у вас нет `cvs`, либо если она версии 1.10.4 или старше, то вам придется получить свежую версию `cvs`. Это можно сделать по адресу <http://www.cvshome.org>.

2. Укажите наименование удаленного хранилища (*репозитория*) CVS. Для этого проще всего назначить переменную окружения следующей командой:

```
cvsroot=:pserver:anoncvs@subversions.gnu.org:/cvsroot/gcc
export CVSROOT
```

Значением назначенной переменной является расположение удаленного репозитория CVS. Утилита **cvs**, если не определен параметр **-d**, считывает значение переменной окружения, указанной ей в командной строке. При желании можно использовать параметр **-d**, чтобы назначить адрес репозитория в командной строке **cvs**, в таком случае этот параметр должен стоять первым в списке параметров командной строки:

```
cvs -d :pserver:anoncvs@subversions.gnu.org:/cvsroot/gcc
```

3. Подключитесь к системе CVS. При назначенной переменной окружения **CVSROOT** вы можете подключиться непосредственно к удаленному репозиторию следующей командой:

```
cvs login
```

Вам будет предложено ввести пароль. Для анонимного доступа с правами только для чтения просто нажмите клавишу ввода. При успешном подключении текстовый интерфейс операционной системы перейдет в режим ожидания следующей команды **cvs**.

4. Загрузите исходные файлы. Назначьте родительский каталог, который будет содержать дерево исходных каталогов GCC. По следующей команде **cvs** загрузит все исходные файлы назначенного выпуска и сохранит их в новом каталоге с именем **gcc**:

```
cvs -z 9 checkout -r gcc_3_1_0_release gcc
```

Параметр **-z 9** указывает **cvs** сжимать файлы при передаче, что сокращает время их загрузки. Будут сжиматься файлы или нет, конечный результат будет тем же, **cvs** распаковывает файлы при записи, когда получает их в сжатом виде.

5. Используя тот же тэг, вы можете также получать/обновлять и документацию, соответствующую версии компилятора. Она сохраняется в каталоге с именем **wwwdocs** в виде файлов в формате HTML. Команда для загрузки документации ненамного отличается от той, что вы использовали для получения исходных файлов:

```
cvs -z 9 checkout -r gcc_3_1_0_release wwwdocs
```

Предыдущие выпуски

Вероятно, вы захотите получить последнюю версию компилятора по загрузке через систему CVS. Однако есть возможность использовать перечисленные ниже тэги для получения более ранних выпусков:

gcc_3_0_3_release	gcc_2_95_2-release	egcs_1_1_release
gcc_3_0_2_release	gcc_2_95_1-release	egcs_1_0_3_release
gcc_3_0_1_release	gcc_2_95-release	egcs_1_0_2_release

gcc_3_0_release	egcs_1_1_2_release	egcs_1_0_1_release
gcc_2_95_3	egcs_1_1_1_release	egcs_1_0_release

Экспериментальная версия

Если не указано имя тэга, то вы получаете текущий вариант последней экспериментальной версии GCC. Для загрузки экспериментальной версии подается следующая команда:

```
cvs -z 9 checkout gcc
```

Полученный этим путем исходный код является последним и новейшим вариантом экспериментальной версии компилятора, поэтому он может работать некорректно. В действительности нет никакой гарантии, что вы вообще сможете его скомпилировать.

Однажды получив все файлы, представляющие собой экземпляр последней версии, вы в дальнейшем можете поддерживать их в свежем состоянии, используя команду **cvs** для получения обновлений в любое время, когда пожелаете. По отданной команде **cvs** проведет сравнение версий файлов в репозитории с версиями файлов на локальном диске и загрузит только те файлы, которые следует обновить. Команда для обновления выглядит так:

```
cvs -z 9 update
```

По мере обновления файлов в локальных каталогах выводится список, в котором имени каждого файла предшествует один символ, обозначающий предпринятое по этому файлу действие. Буква "Р" показывает, что имеющийся файл соответствует последней версии. Буква "U" означает, что файл заменен обновленной версией. Знак вопроса "?" появляется перед именем локального файла, не имеющего соответствия в репозитории CVS.

По мере разработки программ компилятора GCC обновляется также и соответствующая документация на них. Используя **cvs**, вы можете загрузить последнюю версию этой документации очень простым способом:

```
cvs -z 9 checkout wwwdocs
```

После того как вы решитесь загрузить документацию, она в дальнейшем будет обновляться каждый раз, когда вы будете обновлять с помощью **cvs** исходные файлы.

Компиляция и установка GCC

Установка GCC выполнялась тысячи раз в течение ряда лет на многих различных платформах, и она, конечно, вполне устоялась и усовершенствовалась за это время. Если вы собираетесь и скомпилировать и установить компилятор GCC на машине, уже имеющей его используемую версию, то это будет совсем просто. Если же вам нужно сделать что-то особенное, то опции командной строки, поддерживаемые компилятором, вам доступны во всем своем их изобилии.

Процедура инсталляции

Далее вам предлагается последовательность основных действий, необходимых для установки GCC.

1. Убедитесь в доступности вашего текущего компилятора. Исполняемый файл `cc` либо `gcc` должен находиться в текущем каталоге, или должна быть назначена переменная окружения с именем `CC`, содержащая имя компилятора и его расположение.
2. Проверьте, установлена ли утилита `make`. Наверняка сборочный компоновщик иной версии сработает вполне прилично, хотя могут и возникнуть проблемы. Если вы решите использовать компоновщик другой версии и при этом обнаружите несколько странных сообщений об ошибках, то следует переустановить `make` и попытаться снова. Чтобы убедиться, что компоновщик GNU у вас установлен, введите следующую команду, по которой тот сообщит номер своей версии:

```
make -v
```

3. Создайте конфигурационный каталог. Он будет корневым по отношению к дереву каталогов, содержащему все компоновочные и создаваемые при компоновке объектные файлы. Настойчиво рекомендуется не компилировать GCC в каталогах, содержащих исходные файлы.
4. Выберите желаемые опции компиляции и поместите их в сценарий `configure`. У вас есть довольно большой их выбор, и все они описаны в следующем разделе этой главы. Каждая опция имеет значение по умолчанию, так что вам остается только указать особые опции, исходя из своей ситуации. Наиболее часто применяемой является опция `--prefix`, она указывает имя каталога, корневого для инсталляции двоичного исполняемого кода GCC. После завершения инсталляции указанный в этой опции каталог (*префиксный каталог*) содержит все исполняемые и прочие файлы компилятора GCC в подкаталогах `bin`, `include`, `info`, `lib`, `man` и `share`. Префиксный каталог по умолчанию установлен как `/usr/local`. Одним из самых интересных свойств сценария `configure` является его способность почти безошибочно точного распознавания операционной системы и аппаратной базы, на которой она запущена. Он это делает с помощью вызова сценария `config.guess`. При желании вы можете выполнить этот сценарий из командной строки и убедиться, что он правильно идентифицирует вашу систему.
5. Выполните из текущего каталога сценарий `configure`. В том случае, когда вы запускаете сценарий из другого расположения, необходимо указать полный путь к нему. Например, если вы загрузили исходное древо в `/opt/gnu/gcc`, объектный каталог называется `/opt/build` и вы желаете сохранить исполняемые результаты компиляции в `/opt/usr/local`, можете выполнить сценарий `configure` следующим образом:

```
cd /opt/build  
/opt/gnu/gcc/configure --prefix=/opt/usr/local
```

6. Скомпилируйте GCC. После успешной отработки `configure` в объектном каталоге создаются некоторые каталоги и файлы, в том числе и компоновочный сценарий `Makefile`. Для того, чтобы все это скомпилировать и скомпоновать введите команду:

```
make
```

После выполнения компиляции вы, возможно, увидите некоторые предупреждения и сообщения об ошибках. Они обычно не имеют критического значения, поскольку компилятор игнорирует их и продвигается дальше к следующему файлу. Некоторые из таких ошибок и предупреждений предполагаемы, и только ошибки, приводящие к остановке процесса компиляции, действительно заслуживают внимания.

7. Можете проверить компилятор. Собственно, запуск тестового набора не обязательен. Это даже может потребовать загрузки некоторых дополнительных программ. Но если вы пожелаете запустить тестовый набор, то последовательность предпринимаемых для этого действий вы найдете в конце этой главы — в разделе "Запуск проверочного набора".
8. Установите компилятор в системе. Все скомпилированное хозяйство устанавливается следующей командой:

```
make install
```

9. Установите каталог в пути доступа. Для возможности непосредственного использования Сборного Компилятора GNU необходимо включить каталог, содержащий исполняемые файлы GCC в список путей, содержащийся в переменной окружения `PATH`. Хотя, если вы изменяли предполагаемые значения опций расположения в сценарии `configure`, то может оказаться, что переменная `PATH` уже корректно установлена.
10. Если вы желаете построить *кросс-компилятор* (т.е. компилятор, применяемый для перекрестной компиляции — тот, который запускается на одной машине и вырабатывает исполняемый код для другой машины), то сначала следует построить "родной" компилятор для той машины, на которой он будет запускаться. Затем создать на нем кросс-компилятор, следуя указаниям, приведенным в главе 16.
11. Если нужно получить компилятор языка *Ada*, который не до конца создается в результате описываемого здесь процесса, то вам следует выполнить процедуру, приведенную в главе 9.

Опции конфигурации

Параметры установки, или инсталляционные опции, указываются в командной строке сценария `configure`. Он является предписанием, в соответствии с которым генерируются файлы, управляющие процессом компиляции и установки. Каждая опция имеет предполагаемое значение, отвечающее задаче создания компилятора (набора компиляторов) для вашей локальной машины. Но бывают обстоятельства, когда в некоторые опции должны быть внесены поправки. Далее приводится описание применяемых опций.

- **ENABLE и DISABLE.** Все опции, которые начинаются с `--enable`, имеют соответствующую обратную им опцию, начинающуюся с `--disable`. Какая из них применяется по умолчанию, зависит от платформы. В дальнейшем алфавитном списке опций описаны их варианты, начинающиеся с `--enable`.
- **WITH и WITHOUT.** Опции, которые начинаются с `--with` имеют свои соответствия, начинающиеся с `--without`. Какая опция из такой пары применяется по умолчанию, зависит от платформы. В дальнейшем алфавитном списке описаны их варианты, начинающиеся с `--with`.
- **ЯЗЫКИ.** (*Languages.*) По умолчанию сценарий `configure` приготовит к компиляции все языки, имеющиеся в инсталляции, но вы можете указать в опции `--enable-languages`, какие из них будут скомпилированы.
- **ПРЕФИКСНЫЙ КАТАЛОГ.** (*Prefix directory.*) Части компилятора устанавливаются в набор каталогов, имеющих устоявшиеся стандартные имена. Но вы можете переназначить их имена как угодно. Даже если вы пожелаете изменить имена каталогов, вам в редких случаях придется использовать другие опции, кроме `--prefix`. *Префиксный каталог* является корневым для всех каталогов инсталляции. Он является приставкой (префиксом) для составления их полных имен. Вам следует знать, что использование каталогов исходного дерева для инсталляции объектных файлов не рекомендуется, это может приводить к конфликтам, вызывающим при инсталляции проблемы.
- **ИМЕНА ФАЙЛОВ.** Возможно переназначение имен файлов, которые составляют комплект устанавливаемого компилятора. Это иногда помогает при разработке собственного компилятора или установке нескольких версий GCC.
- **БИБЛИОТЕКИ.** *Разделяемые библиотеки* (*runtime libraries*) для различных языков, содержащие функции, используемые во время выполнения программ, входят в состав GCC. Все необходимые библиотеки, как разделяемые динамические (*shared*), так и *статические библиотеки* (*static*), создаются как собственная часть GCC при его компиляции. Некоторые из них являются обязательными, другие нет.
- **АССЕМБЛЕР и КОМПОНОВЩИК.** (*Assembler and linker.*) Для определения расположения имен задействуемых ассемблера и компоновщика (*Linker*) может быть использован набор соответствующих параметров. Если же вы ими не воспользуетесь, то конфигурационная процедура для их поиска выполняет два действия:
 1. Конфигурационный сценарий проводит их поиск в каталоге `exec-prefix/lib/gcc-lib/target/version`, где: `exec-prefix` предполагается `/usr/local`, если он не установлен опциями установки каталогов `--prefix` или `--exec-prefix`; `target` — название целевой системы, `version` — номер версии GCC.
 2. Конфигурационный сценарий ищет их в специфических для операционных систем каталогах (таких как `/usr/ccs/bin` для *Solaris*, или `/usr/bin` для *Linux*).
- **ВЫРАБОТКА ОБЪЕКТНОГО КОДА.** (*Code Generation.*) Существует две категории параметров генерации кода. Первая определяет тип объектного кода, из

которого составляется сам компилятор. Другая — тип кода, который должен вырабатываться составляемым компилятором.

- **ПЛАТФОРМА.** (Platform, target, host.) Некоторые опции применяются для того, чтобы определенные программы запускались на определенных операционных системах. Несмотря на то, что сценарий `config.guess` почти всегда может правильно распознать платформу, которую вы используете, встречаются и такие платформы, которых он может и не распознать. Некоторые системы можно считать идентичными, но на деле они могут иметь некоторые различия.

--bindir=*directory*

По умолчанию `exec-prefix/bin`. Полное имя каталога, в котором должны создаваться исполняемые файлы. Обычно это имя содержит переменная окружения `PATH`, благодаря этому команды компилятору можно подавать из командной строки независимо от текущего расположения.

--build=*host*

Создает конфигурацию компилятора, запускаемого на указанной "базовой" платформе `host`. Предполагаемое значение этой опции содержится в переменной `--host`, по умолчанию устанавливаемой сценарием `config.guess`.

--cache-file=*filename*

Сценарий `configure` выполняет многочисленные проверки для определения конфигурации и возможностей локальной машины. Файл с указанным именем содержит результаты проверок.

--datadir=*directory*

По умолчанию равно значению `prefix/share`. Полное имя каталога, содержащего файлы данных, такие как информация локализации (locale information).

--enable-alitvec

Указывает PowerPC в качестве целевой платформы, машину PowerPC имеющую аппаратную поддержку усиления векторных функций AltiVec (AltiVec vector enhancements), и, когда это возможно, назначает генерацию кода с использованием инструкций набора AltiVec.

--enable-checking=[*check*[,*check*,...]]

По этому параметру включается выработка кода, выполняющего некоторые внутренние проверки компилятора. Проверки порождают вывод данных диагностики и увеличивают время компиляции, но не имеют никакого дополнительного влияния на результаты работы компилятора.

--enable-cpp

Назначает установку доступной для пользователя версии препроцессора C — программы `cpp`. Этот параметр по умолчанию включен. Также см.

`--with-cpp-install-dir`.

--enable-languages=*language*[,*language*,...]

Определяет, что при построении Сборного Компилятора GNU будут подключаться только компиляторы указанных языков. Доступны следующие имена языков: `ada`, `c`, `c++`, `f77`, `java`, `objc` и `chill`. Если эта опция не используется, то подключаются все языки. Для подключения языка *Ada* требуются некоторые дополнительные действия, описанные в главе 9. Язык *CHILL* в настоящее время уже не поддерживает, но может правильно компилироваться в старших версиях *GCC*.

--enable-libgcj

Назначает построение динамической библиотеки функций (*runtime library*), загружаемых во время выполнения программ на языке *Java*. Этот параметр по умолчанию включен. Назначение опции `--disable-libgcj` делает возможным создание компилятора *Java* с использованием такой библиотеки, полученной из другого источника.

--enable-maintainer-mode

Опция назначает создание файла `gcc.pot` из исходного кода заново. Этот файл содержит главный каталог сообщений об ошибках и сообщений-предупреждений, вырабатываемых компилятором. Этот файл используется для интернационализации, подробнее об этом — в главе 11.

Примечание

Для корректной работы этой опции требуется полный вариант дерева исходников *GCC* и свежая версия утилиты `gettext`.

--enable-multilib

На многих системах применяется по умолчанию. Опция назначает построение многокомпонентных библиотек для целевой машины. Обычно такие библиотеки создаются для поддержки компилятором различных вариантов целевых платформ (эмуляция операций над числами с плавающей точкой, поддержка стандартов вызова функций и т.д.). Если желательно подавление генерации ряда библиотек, то вместо полного отключения `multilib` на ряде платформ, перечисленных в таблице 2.3, возможно подавление выработки отдельных библиотек по их имени. Например, для платформ `arc-*-*-elf*` можно использовать опцию `--disable-biendian` для подавления создания соответствующей библиотеки.

Таблица 2.3. Предусмотренные для ряда платформ варианты подавления генерации библиотек

Платформа	Наименование библиотеки
<code>arc-*-*-elf*</code>	<code>biendian</code>
<code>arm-*-*</code>	<code>fpu</code> , <code>26bit</code> , <code>underscore</code> , <code>interwork</code> , <code>biendian</code> , <code>nofmult</code>
<code>m68*-*-*</code>	<code>softfloat</code> , <code>m68881</code> , <code>m68000</code> , <code>m68020</code>
<code>mips*-*-*</code>	<code>single-float</code> , <code>biendian</code> , <code>softfloat</code>

Платформа	Наименование библиотеки
powerpc*-*-*	aix64, pthread, softfloat, powercpu, powerpccpu, powerpcos, biendian, sysv,aix
rs6000*-*-*	aix64, pthread, softfloat, powercpu, powerpccpu, powerpcos, biendian, sysv,aix

--enable-nls

Назначает подключение к сборному компилятору GNU в качестве его части системы поддержки национальных языков *NLS* (National Language Support), что позволяет получать сообщения об ошибках и предупреждения компилятора на других языках, кроме "родного" языка GCC — американского диалекта английского языка.

--enable-shared

Действует по умолчанию. При назначении **--disable-shared** будет проведено построение только статических библиотек (построение разделяемых библиотек будет отключено).

--enable-shared[=package[,package ...]]

Для **gcc** версии 2.95 и старших эта опция необходима для построения разделяемых библиотек. В последующих версиях такие библиотеки собираются по умолчанию для всех поддерживающих языков.

Назначение списка имен пакетов указывает, что разделяемые библиотеки будут создаваться только для указанных пакетов. Распознаваемыми именами пакетов являются **libgcc**, **libstdc++**, **libffi**, **zlib**, **boehm-gc** и **libjava**.

--enable-target-optspace

Указывает оптимизировать размер библиотек за счет их быстродействия.

--enable-threads

Для некоторых платформ эта опция действует по умолчанию. Указывает, что целевая платформа поддерживает *потоки* (*threads*). Это влияет на библиотеки для *Objective-C* и на перехват исключений для *C++* и *Java*. Если целевая платформа не поддерживает потоки или если компилятор не способен вырабатывать для целевой платформы код, поддерживающий потоки, то действует опция **--disable-threads** или аналогичная ей **--enable-threads=single**.

--enable-threads=library

Назначает указываемую библиотеку в качестве библиотеки поддержки потоков. Список имен доступных библиотек приводится в таблице 2.4.

Таблица 2.4. Имена, используемые для выбора библиотеки поддержки потоков.

Библиотека	Описание
aix	Поддержка потоков для AIX.
dce	Поддержка потоков для DCE.

Библиотека	Описание
mach	Поддержка последовательных потоков для MACH. Требует наличия копии заголовочного файла <code>gthr-mach.h</code> .
no	То же, что и <code>single</code> .
posix	Стандартная поддержка потоков для POSIX.
rtems	Поддержка потоков для RTEMS.
single	Отключение поддержки потоков.
solaris	Поддержка потоков для Sun Solaris 2.
vxworks	Поддержка потоков для VxWorks.
win32	Поддержка потоков для Microsoft Win32.

--enable-version-specific-runtime-libs

Указывает, что заголовочные файлы, содержащие определенные библиотеки подключаемых во время выполнения функций (runtime libraries) вместо их обычного расположения, установлены в каталоге, название которого соответствует платформе и версии. Эти библиотеки установлены в `libdir/gcc-lib/target/version`, включаемые для `libstdc++` файлы установлены в `libdir/gcc-lib/target/version/include/g++`, если не определено другое расположение опцией `--with-gxx-include-dir`.

--enable-win32-registry

Использование этой опции без параметра указывает, что версия GCC для *Microsoft Win32* не использует для определения пути инсталляции компилятора и его библиотек значение ключа *Регистра* (Win32 Registry):

```
HKEY_LOCAL_MACHINE\SOFTWARE\Free Software Foundation\key
```

Значение ключа по умолчанию устанавливается номером версии GCC. Значение этого ключа может также и назначаться параметром опции `--enable-win32-registry`.

--enable-win32-registry=key

Указывает, что версия GCC для *Microsoft Win32* определяет пути инсталляции с помощью *Регистра* (Win32 Registry), используя значение следующего ключа:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Free Software Foundation\key
```

Если вы не используете эту опцию, то по умолчанию в качестве значения ключа используется номер версии GCC. Подобное применение реестра позволяет одновременно использовать различные версии GCC, устанавливая их в различное расположение.

--exec-prefix=directory

По умолчанию имеет значение, соответствующее `prefix`. Это имя верхнего каталога для расположения всех каталогов и файлов, связанных с различными платформами.

--help

По этой опции будет выдан список всех опций командной строки и остановлено выполнение сценария `configure` без каких-либо дополнительных действий.

--host=*host*

Назначает имя платформы "домашнего" компьютера (*host*), т.е. того, на котором компилируется GCC. По умолчанию используется вывод сценария `config.guess`. Компилятор GCC может запускаться на разнообразных платформах.

--includedir=*directory*

По умолчанию имеет значение `prefix/include`. Полное имя каталога, который должен содержать заголовочные файлы, включаемые в программы на языке C.

--infodir=*directory*

По умолчанию имеет значение `prefix/info`. Полное имя каталога для хранения документации в формате утилиты `info`.

--libdir=*directory*

По умолчанию имеет значение `exec-prefix/lib`. Полное имя каталога, который должен содержать статические библиотеки и другие внутренние части компилятора GCC.

--libexecdir=*directory*

По умолчанию имеет значение `exec-prefix/libexec`. Полное имя каталога, который должен содержать некоторые программы в исполняемом коде, связанные с библиотеками.

--localstatedir=*directory*

По умолчанию имеет значение `prefix/etc`. Полное имя каталога, который должен содержать изменяемые данные, специфичные для машины. См. также `--sysconfdir`.

--mandir=*directory*

По умолчанию имеет значение `prefix/man`. Полное имя каталога, который должен содержать Руководство Пользователя в UNIX-формате "manual pages".

--nfp

Указывает, что машина не имеет аппаратной поддержки операций с плавающей точкой. Применяется только для `m68k-sun-sunos*` и `m68k-isix-bsd`.

--no-create

Конфигурационный сценарий запускается, но не создает необходимых для компиляции выходных файлов.

--norecursion

В каждом каталоге исходного дерева имеется свой сценарий **configure**. Если вы не назначите эту опцию, то выполнение каждого сценария **configure** вызовет также и выполнение сценариев **configure** во всех подкаталогах.

--prefix=directory

По умолчанию `/usr/local`. *Префиксный каталог* инсталляции. Полное имя верхнего каталога, корневого для всей инсталляции GCC. Составляющие инсталляцию каталоги по умолчанию располагаются внутри каталога **directory** и имеют стандартные имена **bin**, **include**, **info**, **lib**, **man** и **share**. Указание префикса **directory** назначает полный путь к каждому из устанавливаемых каталогов, если не применены другие опции, особо изменяющие полные имена отдельных каталогов.

--program-prefix=ProgramPrefix

По умолчанию не используется. **ProgramPrefix** добавляется к началу имен исполняемых файлов, создаваемых в каталоге **bin**. Например, чтобы изменить имя устанавливаемой программы компилятора Java с **gcj** на **stim-gcj**, нужно использовать следующую опцию:

```
--program-prefix=stim-
```

--program-suffix=ProgramSuffix

По умолчанию не используется. **ProgramSuffix** добавляется к концу имен исполняемых файлов, которые создаются в каталоге **bin**. Например, чтобы изменить имя устанавливаемой программы компилятора Java с **gcj** на **gcj-v4**, нужно использовать следующую опцию:

```
--program-suffix=-v4
```

--program-transform-name=pattern

Указанный шаблон в формате регулярного выражения UNIX редактора **sed** применяется для преобразования имен файлов, размещаемых в каталоге **bin**. Использование регулярных выражений делает возможным отдельное изменение имени каждого файла в каталоге. Например, для изменения имени устанавливаемой программы компилятора Java с **gcj** на **gjava** и имени **g++** на **gcplus** без изменения имен других файлов, следует применить следующую опцию:

```
--program-transform-name='s/^gcj$/gjava; s/^g++$/gcplus/'
```

--sbindir=directory

По умолчанию **exec-prefix/sbin**. Полное имя каталога, который должен содержать системные исполняемые файлы.

--silent

Подавляет вывод результатов выполнения сценария `configure` на экран. Обычно, когда эта опция не указывается, выводится список выполненных тестов. Эта опция равносильна `--quiet`.

--srcdir=directory

Каталог, в котором находится сценарий `configure.in`, предоставляющий для `configure` особую информацию об именах и расположении исходных файлов.

--sysconfdir=directory

По умолчанию `prefix/etc`. Полное имя каталога, который содержит данные "только для чтения", специфичные для машины. См. также `--localstatedir`.

--target=target

Имя целевой платформы (`target`), т.е. той для которой компилируется GCC. По умолчанию используется вывод сценария `config.guess`.

--tmpdir=directory

Назначает полное имя каталога, используемого сценарием `configure` для хранения временных файлов.

--version

Выводит номер утилиты `autoconf`, используемой для создания сценариев `configure`. Никаких дополнительных действий при этом не предпринимается.

--with-as=pathname

Указывает путь расположения программы ассемблера. Применяется в тех случаях, когда стандартная процедура поиска, предпринимаемого сценарием `configure`, не дает результата, или при наличии в системе нескольких программ ассемблеров и необходимости точного указания на ту из них, которую следует использовать.

--with-catgets

В случае, когда подключена система поддержки национальных языков (*NLS*) опцией `--enable-nls`, но на "домашней" (`host`) машине не установлена утилита `settext`, компилятор будет использовать `catgets`.

--with-cpp-install-dir=directory

Указывает, что копия `cpp` (программы препроцессора *C*) следует установить в каталог `prefix/directory` в дополнение его установки в каталоге, указанном опцией `--bindir` (по умолчанию `exec-prefix/bin`). Также см. `--disable-cpp`.

--with-cpu(cpu)

Указывает тип центрального процессорного блока (*CPU*, Central Processor Unit) целевой машины. При назначении типа процессора GCC имеет возможность выра-

ботки более качественного кода (оптимального для указанной марки процессора), чем тот код, который предполагает совместимость с целым семейством процессоров. В таблице 2.5 предлагается список CPU, поддерживаемых в текущей версии GCC. Новые типы процессоров добавляются постоянно, поэтому если вы не найдете нужную марку в таблице, то посмотрите, нет ли ее в конфигурационном файле `config.gcc`.

Таблица 2.5 Типы конфигураций процессоров, выбираемые по названию

Семейство	Назания процессоров
arm*-*-*	xarm2, xarm3, xarm6, xarm7, xarm8, xarm9, xarm250, xarm600, xarm610, xarm700, xarm7m, xarm7dm, xarm7dmi, xarm7tdmi, xarm7100, xarm7500, xarm7500fe, xarm810, xxscale, xstrongarm, xstrongarm110, xstrongarm1100
powerpc*-*-*	xcommon, xpower, xpower2, xpower3, xpowerpc, xpowerpc64, xrios, xrios1, xrios2, xrsc, xrsc1, xrs64a, x401, x403, x405, x505, x601, x602, x603, x603e, x604, x604e, x620, x630, x740, x750, xx801, x821, x823, x8607400, x7450, xec603e
sparc*-*-*	supersparc, hypersparc, ultrasparc, v7, v8, v9

--with-dwarf2

Назначает формат информации для отладки программ, которую будет вырабатывать устанавливаемый компилятор, как *DWARF2*.

--with-gnu-as

Устанавливает режим использования *ассемблера GNU* независимо от обнаружения в комплектного системе ассемблера. На системах, чувствительных к такой ситуации, возможны проблемы, когда назначена эта опция и найденный системный ассемблер не является ассемблером GNU. Также могут возникать проблемы при назначении этой опции и тогда, когда найденный ассемблер является частью компилятора GNU. Далее следует список платформ, к которым это может относиться:

hppa1.0-*_*	m68k-sony-bsd
hppa1.1-*_*	m68k-altos-sysv
i386-*_sysv	m68000-hp-hpx
i386-*_isc	m68000-att-sysv
1860-*_bsd	*-lynx-lynxos
m68k-bull-sysv	mips-*
m68-hp-hpx	

Если в вашей системе имеется несколько ассемблеров, то вы должны указать, который из них следует использовать, опцией `--with-as`. На следующих системах при применении ассемблера GNU также можно применять и комплектный ему *компоновщик* (linker) GNU (и, конечно же, указывать его опцией `--with-ld`):

i386-*_sysv	m68k-altos-sysv
i386-*_bsd	m68000-hp-hpx

m68k -bull-sysv	m68000-att-sysv
m68k -hp-hpx	*-lynx-lynxos
m68k-sony-bsd	mips-* (except mips-sgi-irix5-*)

--with-gnu-ld

Такая же опция, как **--with-as**, но только для компоновщика (linker).

--with-gxx-include-dir=directory

Опция назначает полное имя каталога для стандартных заголовочных файлов `g++`. По умолчанию имеет значение `prefix/include/g++-v3`. Также см. **--enable-version-specific-runtime-libs**.

--with-headers=directory

Назначает каталог для заголовочных файлов целевой системы при сборке *кросс-компилятора*. Эта опция необходима при отсутствии каталога `prefix/target/sys-include`. Заголовочные файлы копируются в инсталляционный каталог GCC и модифицируются для обеспечения совместимости с целевой платформой. Также см. **--with-newlib** и **--with-libs**.

--with-included-gettext

При подключении системы поддержки национальных языков *NLS* опцией **--enable-nls** эта опция указывает, что процесс компоновки будет пытаться использовать собственную копию утилиты `gettext` в первую очередь, до использования той версии `gettext`, которая установлена в системе.

--with-ld=pathname

То же, что и опция **--with-as**, но для компоновщика (linker).

--with-libs="directory [directory ...]"

Эта опция применяется для построения *кросс-компилятора*. Библиотеки функций из перечисленных каталогов будут скопированы в инсталляционный каталог GCC. Также см. **--with-headers** и **--with-newlib**.

--with-local-prefix=directory

По умолчанию `/usr/local`. Это префикс для каталога включаемых по директиве `#include` исходных файлов (`include directory`), в этом каталоге компилятор будет искать локально установленные подключаемые `include`-файлы. Опцию следует использовать только в том случае, если в вашей системе уже установлено правило использования некоторых других каталогов, кроме `/usr/local/include`, для локальных заголовочных файлов (header files). В качестве назначаемого опцией каталога не следует использовать `/usr`, потому что в этом случае устанавливаемые заголовочные файлы будут перемешаны с системными заголовочными файлами и появятся конфликты, из-за которых некоторые программы не будут компилироваться.

Назначение опции **--prefix** не влияет на каталог, назначаемый рассматриваемой опцией. Опция **--prefix** указывает, куда устанавливать GCC, а рассматриваемая опция сообщает компилятору, где искать включаемые заголовочные файлы во время его работы.

--with-newlib

Применяется при сборке *кросс-компилятора*. Библиотека **newlib** используется как библиотека стандартных функций языка *C* целевой машины. Функция **__fprintf** не включена в **libgcc.a**, предполагается что она предоставляется в библиотеке **newlib**. Также см. **--with-headers** и **--with-libs**.

--with-slibdir=directory

По умолчанию **libdir**. Полное имя каталога, в котором будут содержаться разделяемые библиотеки (shared libraries).

--with-stabs

Назначает формат **STABS** в качестве формата информации для отладки программ, которую будет вырабатывать устанавливаемый компилятор, взамен стандартного формата, применяемого для этого на "домашней" (host) системе. Обычно по умолчанию GCC использует для отладки программ информацию в стандарте **ECOFF**, но использование этой опции заменяет его на применяемый в системе BSD стандарт **STABS**. Эта опция назначает предустановку компилятора, действие которой может быть отменено применением в команде на запуск компилятора параметра **-gcoff** или **-gstabs**.

Стандарт ECOFF содержит недостаточно информации для отладки программ на других языках, кроме *C*. Формат STABS несет больше информации, однако обычно он требует применения отладчика **gdb**.

--with-system-zlib

Указывает, что при установке компилятора следует использовать уже установленную в системе утилиту **zlib** вместо создания новой. Опция применяется только при установке компилятора *Java*.

--with-x

Указывает, что будет применяться система *X Window*.

--x-includes=directory

Имя каталога, содержащего включаемые файлы для системы *X Window* (X include files).

--x-libraries=directory

Имя каталога, содержащего библиотеки для системы *X Window* (X libraries).

Пакет binutils

Несмотря на предусмотренную возможность использования GCC с комплектными ассемблерами, утилитами и компоновщиками тех систем, на которых он применяется, все же он лучше работает и более совместим с ассемблером, компоновщиком и другими утилитами GNU. Все программы, составляющие пакет **binutils**, кратко описаны вместе с остальными средствами GCC в таблице 1.4. Вот список имен утилит пакета **binutils**:

addr2line	gprof	objcopy	size
ar	ld	objdump	strings
as	nlmconv	ranlib	strip
c++filt	nm	readelf	windres

Некоторые из этих утилит считывают и записывают информацию внутри объектных файлов. Эти возможности обеспечиваются благодаря применению библиотеки *BFD* (Binary File Descriptor library), поставляемой вместе с исходным кодом пакета **binutils**. Библиотека предоставляет набор функций, которые распознают несколько различных форматов объектного кода и могут использоваться для различных действий с ним. Это делает возможным однообразно компилировать и применять каждую из утилит пакета на различных платформах.

Для загрузки исходного кода **binutils** и установки его в готовом для компиляции виде следует предпринять следующие действия:

1. Выбрать FTP-сайт. FTP-сайт GNU находится по адресу <ftp://ftp.gnu.org/gnu>, но возможно, что вам удобнее будет воспользоваться одним из тысяч его "зеркал", расположенных по всему миру. Текущий список сайтов-зеркал представлен на <http://www.gnu.org/order/ftp.html>. Для беспроблемной загрузки лучше подобрать ближайший к вам сайт.
2. Загрузите файл **binutils-2.9.tar.gz** в рабочий каталог. Номер версии файла может отличаться, пакет постоянно совершенствуется и обновляется. Необходимо устанавливать параметры загрузки FTP на загрузку бинарных файлов, а не текстовых. Это сжатые файлы и загрузка в режиме для текстовых файлов им повредит из-за неправильной интерпретации их содержимого и преобразования некоторых байтовых сочетаний в знаки кодировки ASCII.
3. Выберите те опции, которые вы желаете использовать в сценарии **configure**. Применяемые в нем опции в основном те же, что и в подобном сценарии для компиляции GCC. Точно также, как и сценарий **configure** исходного пакета GCC, сценарий **configure** пакета **binutils** может быть запущен и вовсе без опций, хотя проще всего использовать опцию **--prefix=prefixdir** для назначения имени каталога, в который будет инсталлироваться исполняемый вариант пакета. Каталог, назначенный как **prefixdir** будет содержать подкаталоги **bin**, **include**, **info**, **man** и **share**. Если **prefixdir** не назначен, то по умолчанию он будет иметь значение **/usr/local**.
4. Выполните сценарий **configure** из рабочего каталога. Если вы запускаете его из другого текущего расположения, то необходимо указать полный путь к нему.

Например, если вы распаковали полученное по загрузке дерево исходных файлов в каталог `/opt/gnu/binutils`, объектный каталог называется `/opt/bubuild` и вы хотите сохранить исполняемый результат в `/opt/usr/local`, то вам следует запустить сценарий такой командой:

```
cd /opt/bubuild  
/opt/gnu/binutils/configure --prefix=/opt/usr/local
```

5. Если это не было сделано раньше, то поместите новый каталог `bin` в переменную окружения `PATH` чтобы утилиты могли быть найдены системой при их выполнении из любого расположения.

Существует альтернатива FTP, вы можете получить копию текущей рабочей версии `binutils` при помощи системы конкурирующих версий *CVS*. Этот способ обычно используется программистами, имеющими намерение внести изменения в исходный код, и это — единственная возможность использования самых последних усовершенствований. Процедура доступа через CVS подобна уже описанной для получения GCC. Сначала устанавливаете значение `CVSROOT`:

```
cvsroot=:pserver:anoncvs@sources.redhat.com:/cvs/src  
export CVSROOT
```

Затем используете для подключения к серверу CVS следующую команду (и вводите имя "anoncvs" в ответ на запрос при входе):

```
cvs login
```

После этого загружаете полное дерево исходных каталогов командой:

```
cvs -z 9 checkout binutils
```

После выполнения этой процедуры можно получать обновления в любое время, подавая после подключения такую команду:

```
cvs -z 9 update
```

Установка прекомпилированной версии для Microsoft Windows

Для запуска сборного компилятора GNU на операционной системе *Microsoft Windows*, вы можете получить уже скомпилированную и готовую к выполнению на этой системе версию GCC. Подробности о компиляторе *Cygwin* можно узнать на сайте <http://cygwin.com>.

Cygwin

Средства разработки программ GNU могут использоваться на системах *Microsoft Windows* благодаря применению разделляемой библиотеки `cygwin1.dll`. Эта библиотека содержит *API* (интерфейс разработки приложений, Application Programming Interface), эмулирующий среду окружения UNIX. Это работает на всех версиях, начиная с *Microsoft Windows 95* (за исключением *Windows CE*). Применение этих

средств позволяет разрабатывать не только *консольные*, но также и приложения, которые работают в *GUI* (Graphic User Interface). Написание GUI-приложений требует применения *Win32 API*, в то время как консольные могут быть написаны на одних только функциях библиотеки *Cygwin*.

Хоть это и свободно распространяемое программное обеспечение (free software), лицензирование *Cygwin* — сборная солянка. На часть его распространяется лицензия *GNU GPL*, часть — под стандартной лицензией *X11*, и часть является собственностью *Public Domain*. Ничего из этого не является условно бесплатным программным обеспечением (*shareware*). Поэтому никому ничего не нужно платить при использовании *Cygwin* ДЛЯ НЕКОММЕРЧЕСКИХ ЦЕЛЕЙ. Однако вы должны быть предупреждены о некоторых лицензионных требованиях, которые действуют в случае использования его В КОММЕРЧЕСКИХ изделиях (имеется в виду предназначенные для продажи программы, использующие при выполнении библиотеку *cygwin1.dll*). Условия получения коммерческой лицензии можно получить по электронной почте, отправив запрос по адресу: sales@cygwin.com.

В системах *Microsoft Windows* выполняются два типа программ: *консольные приложения* (запускаемые из командной строки и не порождающие связанных с графической оболочкой окон) и оконные *GUI-приложения* (которые могут запускаться из консоли, но используют окна графического пользовательского интерфейса — *GUI*). Компиляция программ этих типов несколько отличается между собой.

Консольное приложение может быть скомпилировано и скомпоновано такой командой:

```
gcc helloworld.c -o helloworld.exe
```

Также есть возможность использовать компилятор *GCC* вместе с *Win32 API* и соответствующими утилитами пакета *Cygwin* для создания оконных программ для систем *Microsoft Windows* и разделяемых *DLL*-библиотек . Этот процесс описан в главе 16.

Инсталляция *Cygwin*

Существует специальная инсталляционная программа *setup.exe*. Она может быть использована не только для начальной загрузки и инсталляции, но также для загрузки и установки выпускаемых обновлений. Одной из основных причин загрузки и применения этой инсталляционной утилиты является очень большой размер полного пакета при отсутствии необходимости применения всех его частей. Программа *setup.exe* управляет загрузкой и предоставляет возможность выбора различных вариантов установки пакета *Cygwin*.

Далее приводится последовательность шагов, в общем описывающая процесс инсталляции. Эта процедура почти полностью автоматизирована, после запуска программы, вам остается только отвечать на вопросы.

1. Создайте инсталляционный каталог. Это кое-что большее, чем просто каталог *GCC*, доступный из *Cygwin*. Лучше назвать каталог как-то наподобие *c:\cygwin*, что является его именем по умолчанию. Инсталляция создает набор каталогов (таких как *bin* и *etc*), так что вы создаете корневой каталог для целого дерева.

2. Загрузите в этот новый каталог **setup.exe**. Для этого зайдите на web-сайт <http://cygwin.com/download.html>, где вы найдете свежую информацию и ссылку для загрузки, или сразу наберите запрос <http://cygwin.com/setup.exe> — по его выполнению ваш броузер сразу же предложит выбрать путь для закачки программы **setup.exe**.
3. Запустите на выполнение программу **setup.exe**. вам будет предложено выбирать: инсталлировать программное обеспечение прямо из Интернет, или загрузить его и сохранить в назначенному вами каталоге. Также есть вариант его установки из такого каталога, если вы уже имеете загруженную из Интернет копию. Есть возможность выбрать установку сразу из Интернет, а можно только загрузить выбранные пакеты в каталог, отложив установку до другого раза.
4. Выберите ближайший к вам сайт-“зеркало” из предложенного списка. Если он окажется перегружен и загрузка с него не пойдет, то придется выбрать другой.
5. Выберите устанавливаемые компоненты. вам будет предложен список категорий утилит, все эти программы включены в пакет Cygwin, откомпилированы и готовы к запуску. Можете выбрать сколько угодно утилит из любых категорий, но выбрав категорию “**Devel**”, вы увидите список средств для разработки, включающий также и компилятор GCC. По умолчанию многие пакеты имеют отметку “Пропустить” (“**Skip**”), это означает, что они не будут загружены на ваш компьютер. При выборе мышью этой отметки появляется меню с номерами версий. Дальше, когда вы выбрали загрузить конкретную версию в исполняемом коде, появляется еще возможность поставить “птичку”, которая означает что вы также желаете получить и ее исходный код.
6. Если вы выбираете устанавливать программы пакета Cygwin непосредственно из Интернет, то на этом все можно считать сделанным. Если же вы только загрузили файлы из Интернет, то придется запустить **setup.exe** снова и заказать ему установку с использованием каталога, куда вы загрузили файлы.

Запуск проверочного набора

Перед окончанием установки свежескомпилированной версии GCC у вас есть возможность запуска на ней набора тестов, чтобы убедиться в правильности работы GCC. Это необязательная часть работы. Вообще говоря, если вам удалось скомпилировать GCC так, что он запускается, то можно уже не сомневаться, что он будет работать правильно. Главным образом этими тестами пользуются разработчики, когда нужно проверить, не повредило ли сделанное ими усовершенствование остальным функциям компилятора.

Для запуска тестового набора нужно выполнить только несколько простых шагов:

1. Если вы еще этого не сделали, то следует загрузить проверочный набор и установить его в том же каталоге, где находятся остальные части GCC. Для проверки, загружен ли уже проверочный набор, проверьте, есть ли у вас в исходном дереве каталог **gcc/testsuite**.

2. Установите последнюю версию тестового набора **DejaGnu**. Удостоверьтесь, что у вас свежая версия, потому что версии 1.3 и старше работать не будут.
3. Установите переменные окружения. Если инсталляционные каталоги для утилит DejaGnu **runtst** и **expect** прописаны в установках переменной **PATH**, то, возможно, что вам уже не нужно их туда добавлять. Если нет, то вам, учитывая что набор DejaGnu инсталлирован в **/usr/local**, следует установить две следующие переменные окружения:

```
TCL_LIBRARY=/usr/local/tcl8.0
DEJAGNULIBS=/usr/local/dejagnu
```

4. Запустите проверку. Для этого перейдите в тот же каталог, который вы использовали для компиляции GCC и запустите любой тест. Для запуска всего проверочного набора (это может занять довольно много времени) введите команду:

```
make -k check
```

Опция **-k** указывает утилите **make** игнорировать аварийное завершение теста и переходить к следующей проверке. Для запуска только тестов *верхнего уровня* (front end) для компилятора C команда должна быть такой:

```
make -k check-gcc
```

Для пуска только тестов компилятора C++ — такой:

```
make -k check-g++
```

5. Просмотрите результаты тестов. После выполнения проверок в подкаталогах проверочного набора вы обнаружите созданные в ходе этих проверок файлы. Файлы с расширением **.log** содержат подробный листинг действий, предпринятых в ходе тестов. Файлы с расширением **.exit** содержат отчеты о выполненных проверках, каждой из которых соответствует один из кодов завершения, перечисленных в таблице 2.6.

Таблица 2.6. Коды завершения тестов в отчетах о результатах

Результат	Описание
PASS	Ождалось успешное прохождение теста и он был успешно пройден.
XPASS	Успешное прохождение теста не ожидалось, но он был успешно пройден.
FAIL	Ождалось успешное прохождение теста, но тест не пройден.
XFAIL	Успешное прохождение теста не ожидалось и тест не пройден.
UNSUPPORTED	Прохождение теста не поддерживается на этой платформе.
ERROR	Обнаружена проблема при выполнении теста.
WARNING	Определена возможная проблема при выполнении теста.

*Полное
руководство*



Часть II

**Использование Сборного
Компилиатора**



Глава 3

Препроцессор `cpp`

Препроцессор *CPP* изначально задумывался как часть языка программирования *C*. Препроцессор считывает исходный код и, отвечая на размещенные в нем директивы, производит модифицированную версию этого исходного кода, которая уже затем передается компилятору. Препроцессор по-прежнему является важной частью языков *C*, *C++* и *Objective-C*, но он также используется и для реализации условной компиляции в программах на языках *Fortran* и *Java*.

В терминологии GNU препроцессор представлен как CPP (сокращение от "C Preprocessor"). Соответствующая исполняемая программа GNU имеет имя `cpp`.

Директивы препроцессора

Инструкции препроцессору размещаются в исходном коде в форме директив и легко выделяются из него, благодаря тому, что они начинаются с символа "#" (hash) в первой не пустой позиции строки. Обычно "#" стоит в первой колонке и сразу же за ним — ключевое слово директивы. Все директивы перечислены и описаны в таблице 3.1. Предусмотрена возможность препроцессора модифицировать строки, не выделенные как директивы, но только тогда, когда уже определены директивы, предписывающие такие действия.

Таблица 3.1. Директивы, обрабатываемые препроцессором GNU

Директива	Описание
<code>#define</code>	Определяет имя макроса, которое препроцессор расширяет в исходном коде подстановкой его значения каждый раз, когда оно используется.
<code>#elif</code>	Задает альтернативное условие, проверяемое после директивы <code>#if</code> .
<code>#else</code>	Предоставляет альтернативный блок кода, который компилируется когда условия <code>#if</code> , <code>#ifdef</code> или <code>#ifndef</code> ложны.
<code>#error</code>	Выдает сообщение об ошибке и останавливает препроцессор.

Директива	Описание
#if	Код, находящийся между этой директивой и соответствующей #endif , компилируется при условии ненулевого результата арифметического выражения условия.
#ifdef	Код между этой директивой и соответствующей #endif компилируется только тогда, когда макрос с указанным в параметре директивы именем определен.
#ifndef	Код между этой директивой и соответствующей #endif компилируется только тогда, когда макрос с указанным именем не определен.
#include	Проводит поиск в назначенных каталогах файла с указанным именем, затем помещает в код содержимое этого файла.
#include_next	Работает подобно #include , только поиск включаемого файла начинается с каталога, стоящего в списке следующим после того, в котором файл с указанным именем уже был найден.
#line	Сообщает компилятору номер строки и, возможно, имя файла для помещения отладочной информации в объектном файле.
#pragma	Стандартный метод предоставления дополнительной информации об особенностях имеющегося компилятора или аппаратной платформы.
#undef	Удаляет определение, созданное ранее директивой #define .
#warning	Производит предупредительное сообщение препроцессора.
##	Оператор объединения строк, используемый внутри макрона.

#define

Директива **#define** определяет *макрос*, т.е. создает *макроопределение*. Макрос имеет имя, которое заменяется препроцессором в обрабатываемом тексте строкой символов, определенных как значение макрона. Имеется возможность назначать параметры в макроопределении, используемые как часть расширения макрона.

Большинство макроопределений действуют подобно именованным константам. Традиционно имена макросов набираются большими буквами. К примеру, следующее макроопределение создает макрос с именем **ARRAY_SIZE**, который вызывает подстановку этого имени строкой 512 в любом месте кода, где оно используется:

```
#define ARRAY_SIZE 512
```

Этот макрос может в дальнейшем быть использован для объявления массива на-значеннной величины:

```
Int valarray[ARRAY_SIZE];
```

Следующий пример является широко известным макросом, использующим па-раметры для создания выражения, которое возвращает наименьшее значение из двух заданных величин:

```
#define min(a,b) ((a) < (b) ? (a) : (b))
```

Имя этого макрона может затем быть расширено в исходном коде с использова-нием значений, подставляемых для **a** и **b**:

```
result = min(44,uplim);
```

Развернутый препроцессором из этого макроса код будет выглядеть следующим образом:

```
result = ((44) < (uplim) ? (44) : (uplim));
```

Далее перечислены правила и требования, которым должны удовлетворять макроопределения:

- Макроопределение размещается в одной строке. При необходимости можно разместить его в нескольких строках (для лучшей читаемости либо из-за большой длины), это можно сделать, используя символ обратной косой черты "\" ("backslash") в качестве символа продолжения строки. Так сделано в следующем примере выражения, которое возвращает случайную величину из заданного интервала:

```
#define ran(low,high) \
    ((int)random() % (high-low +1)) \
    + low
```

- Препроцессор обрабатывает текст последовательно и может делать подстановки лишь после макроопределения. К примеру, в следующих 4-х строках кода макрос `B` использован один раз до того как был определен и один раз после:

```
#define A 100
sum = A + B;
#define B 200
sum = A + B;
```

На выходе препроцессора мы получим:

```
Sum = 100 + B;
Sum = 100 + 200;
```

- Подстановки могут выполняться рекурсивно, то есть они могут вкладываться друг в друга. Это значит, что после первой подстановки препроцессор обрабатывает тот же текст снова для выполнения дальнейших подстановок. Следующий пример показывает как один макрос подставляется в другой:

```
#define TANKARD TSIZE
#define TSIZE 100
tank1 = TANKARD;
#define TSIZE 200
tank2 = TANKARD;
```

Обработка этих пяти строк препроцессором даст следующий результат:

```
tank1 = 100;
tank2 = 200;
```

- Для изменения уже определенного макроса необходимо удалить его и определить заново. Пример:

```
#define MKEYVAL 889
#undef MKEYVAL
#define MKEYVAL 890
```

- Для определения макроса с параметрами, список параметров должен помещаться в скобках сразу после имени макроса. Между именем макроса и скобкой

не должно быть пробелов. В следующем примере один макрос определяется с параметрами и один с простой строкой для подстановки:

```
#define showint(a) printf("%d\n",a)
#define incrint(a) a++
showint(300);
incrint(bb1s);
```

В результате обработки препроцессором этого кода получаем:

```
printf("%d\n",300);
bb1s++;
```

- Имена макросов не подставляются внутри строковых литералов, как в следующем примере:

```
#define BLOCK 8192
printf("The BLOCK number.\n");
```

Программа выведет следующее:

```
The BLOCK number
```

- Аргумент, передаваемый макросу, может преобразовываться в строку, когда имени этого аргумента предшествует символ "#". В примере макрос с именем MONK содержит преобразованную к строковому типу переменную, объединяемую с другими строками, которые находятся рядом с ним в макроопределении:

```
#define MONK(ARGTERM) \
    printf("The term " #ARGTERM " is a string\n")
MONK(A to B);
```

В выводе программы мы увидим:

```
The term A to B is a string
```

- Макрос может быть определен без его значения. Несмотря на то, что он не имеет связанного с ним значения для подстановки, он, тем не менее, считается определенным и может использоваться в качестве проверяемого флага.
- *Вариативный макрос* (variadic macro) содержит переменное число аргументов. Аргументы, представленные троеточием "...", сохраняются в строчной форме через запятую в стандартной переменной __VA_ARGS__, расширяемой внутри макроса. В следующем примере макрос принимает любое количество аргументов.

```
#define err(...) fprintf(stderr,__VA_ARGS__)
err("%s %d\n","The error code ", 48);
```

Вот результат на выходе препроцессора после обработки этих двух строк:

```
fprintf(stderr,"%s %d\n","The error code ", 48);
```

Вариативный макрос может иметь настолько большой список параметров, насколько позволяет величина содержащей его переменной. Далее — пример макроса, имеющего два фиксированных аргумента, за которыми следует открытый список параметров:

```
#define errout(a,b,...) \
    fprintf(stderr,"File %s      Line %d\n",a,b); \
    fprintf(stderr,__VA_ARGS__)
```

Пример использования этого макроса:

```
errout(__FILE__,__LINE__,"Unexpected termination\n");
```

Во всех предыдущих формах вариативных макросов требовалось присутствие хотя бы одного параметра, чтобы удовлетворялись требования к переменному списку параметров. Потому что, когда мы использовали функцию `fprint()` внутри макроса, перед `__VA_ARGS__` стояла запятая. Можно помещать `__VA_ARGS__` в список аргументов особым оператором объединения строк, чтобы в случае пустого списка аргументов запятая была удалена. Как в таком примере:

```
fprintf(stderr, ##__VA_ARGS__)
```

#error и #warning

Директива `#error` вызывает сообщение препроцессора о критической ошибке и его остановку. Она может использоваться для перехвата известных условий, при которых скомпилированный код будет заведомо неработоспособным. К примеру, следующий код будет успешно скомпилирован только в том случае, когда определен некоторый макрос с именем `__unix__`:

```
#ifndef __unix__
#error "This section will only work on UNIX systems"
#endif
```

Директива `#warning` действует подобно директиве `#error`, за исключением того, что она не является критической, препроцессор продолжает работу после вывода сообщения.

#if, #elif, #else и #endif

Директива `#if` вычисляет и проверяет результат арифметического выражения условия. В случае, если он не равен нулю, то условие считается истинным. В этом случае *условный код*, стоящий в блоке условной директивы до соответствующей закрывающей директивы (`#elif`, `#else` или `#endif`), передается на выход. В противном случае условие считается ложным и код блока не передается на выход препроцессора и, соответственно, не компилируется. В примере строковая переменная объявляется только тогда, когда значение `COUNT` не равно нулю.

```
#if COUNT
char *desc = "The count is non-zero";
#endif
```

Далее перечислены правила и условия, которые применяются к условным директивам и к выражениям условий.

- Выражение условия может включать целочисленные константы и имена не-пустых макросов, т.е. макросов, содержащих значение.
- Для определения порядка вычисления выражения могут применяться скобки.

- Выражение может включать в себя арифметические действия в форме операторов `+`, `-`, `*`, `/`, `<<` и `>>`, которые действуют также, как такие же целочисленные арифметические операторы в языке С. Действия производятся над числами, формат которых соответствует наибольшему целочисленному формату, поддерживаемому целевой платформой. Обычно 64 бита.
- Выражение может содержать операторы сравнения `>`, `<`, `>=`, `<=` и `==`, действующие так же, как соответствующие им в языке С.
- Выражение может включать логические операторы `&&` и `||`.
- Для логического инвертирования результата выражения может быть использован оператор отрицания `!"`. К примеру, следующее выражение истинно, если `LIMXP` не более 12:

```
#if !(LIMXP > 12)
```

- Если нужно проверить, определен ли какой-либо макрос, для этого может быть использован оператор `defined`. Следующее выражение истинно, только если определен макрос с именем `MINXP`:

```
#if defined(MINXP)
```

- Оператор отрицания `!"` часто используется в сочетании с `defined` для того, чтобы проверить, что макрос с интересующим именем не назначен, например:

```
#if !defined(MINXP)
```

- Идентификатор, не определенный как макрос всегда возвращает нулевое значение.
- Для выдачи сообщения в случае использования такого индентификатора в выражениях можно использовать опцию командной строки `-Wundef`.
- Имена макросов, определенных как имеющие аргументы, всегда возвращают ноль.
- Для выдачи препроцессором сообщения в подобном случае также используют опцию `-Wundef`.
- Директива `#else` может быть использована для представления альтернативного кода, который компилируется тогда, когда выражение условия ложно. Применение этой директивы показано в следующем примере:

```
#if MINTXT <= 5
#define MINTLOG 11
#else
#define MINTLOG 14
#endif
```

- Директива `#elif` используется для предоставления одного или больше альтернативных выражений, как в этом примере:

```
#if MINTXT <=5
#define MINTLOG 11
#elif MINTXT == 6
#define MINTLOG 12
#elif MINTXT ==7
#define MINTLOG 13
```

```
#else
#define MINTLOG 14
#endif
```

#ifdef, #ifndef, #else и #endif

Строки кода находящиеся после директивы `#ifdef` компилируются при условии, что макрос с указанным именем определен. Действие директивы `#ifdef` заканчивается следующей за ней в тексте программы директивой `#endif`. В примере массив объявляется при условии, что макрос `MINTARRAY` определен:

```
#ifdef MINTARRAY
int xarray[20]
#endif /* MINTARRAY */
```

Комментарий в строке с директивой `#endif` не обязателен. Он только служит для улучшения читаемости кода.

Противоположной по отношению к `#ifdef` является директива `#ifndef`, она служит для условной компиляции стоящего после нее кода, когда указанный ею макрос не определен.

Директива `#else` может использоваться после `#ifdef` для определения альтернативного кода. В приводимом примере, если макрос `MINTARRAY` определен, то массив будет объявлен типа `int`, в противном случае он будет иметь тип `char`:

```
#ifdef MINTARRAY
int xarray[20];
#else
char xarray[20];
#endif /* MINTARRAY */
```

Другие директивы могут включаться в код, компилируемый по условию. Это относится также и к директивам `#ifdef`, `#ifndef` и `#if`, однако каждой из подобных директив должна быть должным образом сопоставлена соответствующая ей `#endif`.

#include

По директиве `#include` препроцессор проводит поиск файла с указанным именем и вставляет его содержимое в текст совершенно так же, как он мог бы быть туда помещен при помощи текстового редактора. Файл, включаемый таким образом, в общем случае считается заголовочным файлом C, его имя обычно имеет суффикс `.h`, хотя это может быть любой текстовый файл с любым именем.

Директива `include` имеет две формы. Первая, используемая более для *системных заголовочных файлов*, помещает имя файла в угловые скобки ("angle brackets"). Другая, чаще применяемая для вставки *пользовательских заголовочных файлов*, — в двойные кавычки. Пример для обеих форм:

```
#include <syshead.h>
#include "syshead.h"
```

Далее следует список свойств и правил, которые действуют для директивы **#include**:

- Если имя файла окружает угловые скобки, то поиск файла начинается с каталогов, указанных с использованием опции **-I**, и затем продолжается в стандартном наборе системных каталогов.
- Если имя файла окружает кавычки, то поиск начинается с текущего каталога (т.е того, в котором находится исходный файл) и затем продолжается по правилам, действующим для директивы с угловыми скобками.
- В системах UNIX стандартный набор системных каталогов выглядит следующим образом:

```
/usr/local/include  
/usr/lib/gcc-lib/target/version/include  
/usr/target/include  
/usr/include
```

- Для поиска заголовочных файлов используются два различных списка каталогов. Стандартные системные заголовочные файлы находятся в каталогах из второго списка. Оция командной строки **-I** добавляет каталоги в список для поиска первой очереди. Опции **-prefix**, **-withprefix** и **-idirafter** предназначены для манипуляций с именами каталогов во втором списке каталогов для поиска файлов.
- В случае, когда GCC компилирует программу на языке C++, каталог **/usr/include/g++v3** проверяется на наличие указанного файла прежде всех других стандартных каталогов.
- Относительный путь к каталогу может использоваться вместе с именем файла. Например, если вы определите директиву **#include <sys/time.h>**, то препроцессор будет искать файл **time.h** в подкаталогах **sys** всех системных каталогов.
- Символ наклонной черты "/" ("slash") всегда воспринимается как разделитель имен в пути к каталогу, даже в тех системах, которые используют для этой цели другой символ, например, обратную наклонную черту "\ ("backslash").
- Имя файла воспринимается в таком виде, как оно указано. Внутри имени файла не подставливаются ни имена макросов, ни символы, имеющие специальное значение. Если в директиве указанное имя содержит "*" ("звездочку") или обратную косую черту "\", то файл будет найден только в том случае, если его имя содержит именно эти символы.
- Директива **#define** может быть использована для указания имени включаемого заголовочного файла, как в следующем примере:

```
#define BOGHEADER "bog_3.h"  
#include BOGHEADER
```
- Если в строке, которая содержит директиву **#include**, содержится хоть что-нибудь, кроме комментария, то это воспринимается как ошибка.
- Директива **#line** не может изменить текущий каталог в целях поиска файлов.

- Опция **-I** может применяться для указания, как именно действуют опции **-I**, определяющие каталоги для поиска файлов. Подробные сведения об этом содержатся в приложении 4.

#include_next

Директива **#include_next** применяется только в особых случаях. Она используется внутри заголовочного файла, включаемого в другой файл, и вызывает поиск нового файла. Этот поиск начинается с каталога, следующего за тем, где был найден текущий файл.

Например, действуют такие условия поиска, при котором каталоги просматриваются в таком порядке: **A, B, C, D, E**. Допустим, текущий файл был найден в каталоге **B**. Директива **#include_next** в текущем файле вызовет поиск заголовочного файла с новым именем в каталоге **C**, потом в **D**, и затем в **E**.

Эта директива может применяться для добавления или модификации определений в системных заголовочных файлах без изменения самих файлов. Например, системный заголовочный файл **/usr/include/stdio.h** содержит определение функции **getc**, которая считывает один символ из входного потока. Чтобы заменить эту функцию "болваном", который всегда возвращает один и тот же символ, и при этом оставить все остальное хозяйство стандартного заголовочного файла как есть, вы можете создать собственную версию файла **stdio.h**, который будет содержать следующий код:

```
#include_next "stdio.h"
#undef getc
#define getc(fp) ((int)'x')
```

Использование этого заголовочного файла вызовет вставку стандартного файла **stdio.h** и переназначение макроса **getc**.

#line

Программы-отладчики (debuggers) нуждаются в том, чтобы связывать имена файлов и номера строк исходников с элементами данных и исполняемым кодом программы. Поэтому препроцессор помещает эту информацию в вывод, передаваемый компилятору. Необходимо *трассировать* (т.е. построчно выполнять) каждый исходный файл, в то время как препроцессор создает один файл из нескольких. Компилятор использует эти данные для построения таблиц, помещаемых в объектный код.

В обычной ситуации позволить препроцессору определять номера строк простым подсчетом — это как раз то, что нужно сделать. Однако, бывает и так, что некоторые другие обработчики вызывают отключение передачи этой информации. Например, общепринятый метод передачи SQL-запросов заключается в том, чтобы записать запрос в виде макроса, который затем преобразовывается специальным обработчиком в вызовы SQL-функций более низкого уровня. Это может расширить код, передаваемый компилятору на несколько строк и привести к тому, что счет строк изменится. Обработчик SQL может поправить это, помещая директивы **#line** в свой вывод таким образом, чтобы препроцессор вел подсчет строк, ориентированный на первичный исходный код.

На директивы `#line` действуют следующие правила и условия:

- При определении директивы `#line` с числом препроцессор заменяет свой текущий счет строк заданным числом. В этом примере номер текущей строки устанавливается на 137:

```
#line 137
```

- Определение директивы `#line` с числом и именем файла указывает препроцессору изменить как счет строк, так и имя текущего файла. Например, следующая директива изменяет текущую позицию на первую строку файла `muggles.h`:

```
#line 1 "muggles.h"
```

- Директива `#line` изменяет значение предопределенных макросов `__LINE__` и `__FILE__`.
- Директива `#line` не влияет на имена файлов или каталогов, используемые директивой `#include`.

Директивы `#pragma` и оператор `_Pragma`

Директива `#pragma` предоставляет стандартный метод передачи информации, которая может быть специфичной для компилятора. В соответствии со стандартом компилятор может присоединять любое сообщение, имеющее для него значение, к директиве `#pragma`.

В GCC *прагмы* (прагма-директивы) определяются как два слова. Первое из них "GCC", второе — имя особой прагмы.

`#pragma GCC dependency`

Прагма зависимости `dependency` проверяет отметку времени (timestamp) текущего файла и сравнивает его с отметкой времени другого файла. Если другой файл оказывается более новым, выдается предупреждение. Следующий пример прагмы сравнивает отметку времени текущего файла с отметкой времени файла с именем `lexgen.tbl`:

```
#pragma GCC dependency "lexgen.tbl"
```

В случае, если файл `lexgen.tbl` новее, чем текущий файл, препроцессор выдает следующее сообщение:

```
warning: current file is older than "lexgen.tbl"
```

В директиву `pragma` может добавляться другой текст. Он будет включен как часть предупредительного сообщения, как в следующем примере:

```
#pragma GCC dependency "lexgen.tbl" Header lex.h needs to be updated
```

Эта директива может порождать выдачу следующих сообщений:

```
show.c:26: warning: current file is older than "lexgen.tbl"  
show.c:26: warning: Header lex.h neeeds to be updated
```

#pragma GCC poison

"Испорченная" прагма (poison pragma) может применяться для того, чтобы при любом использовании указанного имени выдавалось предупреждение. Вы можете этим воспользоваться, чтобы иметь гарантию, что определенная функция не будет вызываться. Следующая прагма выдает предупреждение при любом вызове функций копирования памяти (memory-to-memory copy functions):

```
#pragma GCC poison memcpuy memmove
memcpuy(target, source, size);
```

Этот код вызовет следующее предупредительное сообщение:

```
show.c:38:9: attempt to use poisoned "memcpuy"
```

#pragma GCC system_header

Код, стоящий после директивы прагмы `system_header` до конца файла, считается кодом системного заголовочного файла. Такой код компилируется несколько иначе, потому что библиотеки, подключаемые во время выполнения программ, (runtime libraries) не могут быть написаны в полном строгом соответствии со стандартом языка C. Все предупреждения препроцессора, кроме выдаваемых по директиве `#warning`, подавляются на этом участке кода. В частности, некоторые макроопределения и расширения устойчивы к предупредительным сообщениям.

Оператор _Pragma

Обычная директива `#pragma` не может быть включена в макроопределения, поэтому придумали оператор `_Pragma` чтобы генерировать сообщения прагмы внутри макросов. Чтобы создать "испорченную" прагму (poison pragma) внутри макроса поставьте туда такой оператор:

```
_Pragma("GCC poison printf")
```

Обратная наклонная черта "\" ("backslash"), используется в качестве *escape-символа* (escape character). Это делает возможным использование кавычек для создания прагмы зависимости:

```
_Pragma("GCC dependency \"lexgen.tbl\"")
```

#undef

Директива `#undef` используется для удаления макроопределения, ранее созданного директивой `#define`. Это может применяться, когда макроопределение более не нужно, или его нужно переопределить другим значением.

##

Директива *конкатенации* (объединения) "##" используется внутри макроса для объединения значений, содержащихся в двух лексемах исходного кода (source code tokens), в одну строку. Она может использоваться для сборки имен, которые иначе

могут быть неправильно истолкованы программой синтаксического разбора (парсером). В примере два макроса выполняют конкатенацию:

```
#define PASTE1(a)##house
#define PASTE2(a,b) a##b
result = PASTE1(farm);
result = PASTE1(ranch);
result = PASTE2(front,back);
```

После препроцессора получим следующий код:

```
result = farmhouse;
result = ranchhouse;
result = frontback;
```

Предопределенные макросы

Компилятор GCC предопределяет довольно много макросов. В точности, какие из них определены, и какие они содержат значения, зависит от языка компиляции, назначаемых опций командной строки, используемой "домашней" платформы, предназначенной целевой платформы, версии запускаемого компилятора и установленных переменных окружения. Можете использовать опцию препроцессора `-dM`, чтобы увидеть весь их список. Подаваемая для этого команда должна выглядеть примерно так:

```
cpp -E -dM myprog.c | sort | more
```

Список, выводимый по этой команде, содержит директивы `#define` для каждого макроса, определенного препроцессором после обработки указанного исходного файла и всех включаемых им заголовочных файлов.

Таблица 3.2 содержит список макросов, почти всегда определяемых, и описание каждого из них.

Таблица 3.2. Основной набор предопределенных макросов

Имя макроса	Описание
<code>__BASE_FILE__</code>	Строка в кавычках, содержащая полный путь к каталогу, где находится исходный файл, назначенный командной строкой (не обязательно файл, в котором используется макрос). См. также <code>__FILE__</code> .
<code>__CHAR_UNSIGNED__</code>	Макрос назначаемый, чтобы показывать что символьный тип, действующий на целевой машине, является беззнаковым. Он используется в <code>limits.h</code> для определения значений <code>CHAR_MIN</code> и <code>CHAR_MAX</code> .
<code>__cplusplus</code>	Определяется только тогда, когда исходный код является программой на языке C++. Он определен как "1", если компилятор не вполне соответствует полному набору правил стандарта; в противном случае определяется месяцем и годом выпуска стандарта так же, как <code>__STDC_VERSION__</code> для языка C.
<code>__DATE__</code>	Заключенная в кавычки строка из 11 символов, содержащая календарную дату, когда программа была скомпилирована. Имеет такой формат: "may 3 2002"

Имя макроса	Описание
<code>__FILE__</code>	Строка в кавычках, содержащая имя исходного файла, в котором применен макрос. См. также <code>__BASE_FILE__</code> .
<code>__func__</code>	То же, что и <code>__FUNCTION__</code> .
<code>__FUNCTION__</code>	Строка в кавычках, содержащая имя текущей функции.
<code>__GNUC__</code>	Макрос, всегда определяемый как старший номер версии компилятора. Например, если номер версии компилятора 3.1.2, то этот макрос определяется значением "3".
<code>__GNUC_MINOR__</code>	Макрос, всегда определяемый как младший номер версии компилятора (т.е. номер выпуска). Например, если номер версии компилятора 3.1.2, то этот макрос определяется значением "1".
<code>__GNUC_PATCHLEVEL__</code>	Макрос, всегда определяемый как номер редакции выпуска компилятора. Например, если полный номер версии компилятора 3.1.2, то этот макрос определяется значением "2".
<code>__GNUG__</code>	Определяется при использовании компилятора C++. Этот макрос определяется в любом случае, независимо от того, что <code>__cplusplus</code> и <code>__GNUC__</code> также определены.
<code>__INCLUDE_LEVEL__</code>	Целочисленное значение, показывающее текущую глубину включаемого (include) файла. Для основного (назначенного в командной строке) файла равно "0", и увеличивается на единицу внутри каждого следующего файла, включаемого в предыдущий файл имеющейся в нем директивой <code>#include</code> .
<code>__LINE__</code>	Номер строки файла, в котором используется макрос.
<code>__NO_INLINE__</code>	Определяется как "1" в любом случае отсутствия inline-функций, которые должны подставляться кодом их определения, как из-за отсутствия оптимизации, так и в силу намеренной блокировки возможности подстановки кода таких функций.
<code>__OBJC__</code>	Определяется как "1" в случае, если код компилируется как программа на языке Objective-C.
<code>__OPTIMIZE__</code>	Макрос определен как "1", если назначен любой уровень оптимизации.
<code>__OPTIMIZE_SIZE__</code>	Макрос определен значением "1", когда назначена оптимизация размера программы за счет быстродействия.
<code>__REGISTER_PREFIX__</code>	Этот макрос — не строка, а указатель (token). Он указывает префикс для зарегистрированных программ. Может применяться для написания ассемблерного кода, переносимого на различные варианты окружения.
<code>__STDC__</code>	Определяемый как "1", показывает, что компилятор соответствует правилам стандарта языка C. Макрос не определяется при компиляции программ на языках C++ и Objective-C, и когда назначена опция <code>-traditional</code> .
<code>__STDC_HOSTED__</code>	Определяется как "1" в "дружественном" (hosted) окружении (таком окружении, в котором доступны все стандартные библиотеки функций языка C)
<code>__STDC_VERTION__</code>	Значение типа <code>long integer</code> , определяющее версию стандарта в форме года и месяца его выпуска. Например, версия стандарта 1999 года будет представлена значением "199901L". Этот Макрос не определяется при компиляции программ на языках C++ и Objective-C, и когда назначена опция <code>-traditional</code> .

Имя макроя	Описание
<code>_STRICT_ANSI_</code>	Определяется только тогда, когда в командной строке назначена хотя бы одна из опций <code>-ansi</code> и <code>-std</code> . Используется в заголовочных файлах (header files) GNU для ограничения применяемых определений условиями стандарта.
<code>_TIME_</code>	Строка в кавычках из восьми символов, содержащая время компиляции программы в формате "18:10:34".
<code>_USER_LABEL_PREFIX_</code>	Этот макрос — не строка, а указатель (token). Он указывает префикс перед символическими именами в ассемблерном коде. Указатель меняется в зависимости от платформы, но обычно ссылается на символ Подчеркивания "_".
<code>_USING_SJLJ_EXCEPTIONS_</code>	Определяется как "1", если механизм обработки исключений установлен <code>jmp</code> и <code>longjmp</code> .
<code>_VERSION_</code>	Полный номер версии. Для этой информации нет особого формата представления, однако она содержит по крайней мере старший номер версии и номер выпуска.

Таблица 3.3 содержит подборку ключевых языка C++, заменяющих имена операторов, обычно набираемых символами пунктуации. Они воспринимаются препроцессором так же, как если бы они были макросами, определенными с помощью директив `#define`. Если вас интересуют их определения, то подобные им макросы для C и Objective-C определяются в заголовочном файле `iso646.h`.

Таблица 3.3. Именная форма логических операторов в языке C++

Имя оператора	Эквивалентная ему форма с применением знаков пунктуации
<code>and</code>	<code>&&</code>
<code>and_eq</code>	<code>&=</code>
<code>bitand</code>	<code>&</code>
<code>bitor</code>	<code> </code>
<code>compl</code>	<code>~</code>
<code>not</code>	<code>!</code>
<code>not_eq</code>	<code>!=</code>
<code>or</code>	<code> </code>
<code>or_eq</code>	<code> =</code>
<code>xor</code>	<code>^</code>
<code>xor_eq</code>	<code>^=</code>

Включение заголовочного файла единственный раз

В связи с тем, что одни заголовочные файлы включают другие, может оказаться что вы обзавелись программой, которая включает один и тот же заголовок более одного раза. Это может приводить к появлению сообщений об ошибках, связанных с тем, что уже определенные элементы кода будут определяться снова. Для предотвращения подобных приключений заголовочный файл должен иметь особый код, который "отлавливает" такую ситуацию, определяет, не вставлялся ли уже текущий обрабатываемый файл.

```
/* myheader.h */
#ifndef MYHEADER_H
#define MYHEADER_H
/* Тело заголовочного файла */
#endif /* MYHEADER_H */
```

В этом примере, заголовочный файл называется `myheader.h`. Первая строка проверяет, определялся ли ранее макрос `MYHEADER_H`. Если так, то собственно условный код тела заголовочного файла пропускается препроцессором.

Этот прием применяется во всех системных заголовочных файлах. Все определяемые в них имена начинаются знаком подчеркивания, чтобы они не конфликтовали с именами, которые назначаются пользователями. Соглашение, действующее на имена, определяемые для предотвращения повторного включения, состоит в том, что имя должно состоять из прописных букв и содержать в себе имя файла.

Препроцессор GNU распознает эти конструкции и ведет список заголовочных файлов, которые их применяют. Таким способом он оптимизирует обработку кода заголовочных файлов, распознавая имена файлов, и даже не читает файлы, когда они уже включены в программу.

Включение информации о расположении кода в сообщения об ошибках

Предопределенные макросы могут использоваться для автоматизации построения сообщений об ошибках, которые содержат подробную информацию о расположении и типе найденной ошибки. Предопределенные макросы `__FILE__`, `__LINE__` и `__func__` содержат необходимую информацию, но они должны быть применены в том месте, где создается сообщение. Следовательно, если вы напишите функцию, которая содержит все эти макросы, то сообщение будет выдано с указанием имени функции, файла и номера строки.

Прекрасным решением этой задачи будет определение макроса, который содержит все эти предопределения. Таким образом, после того, как препроцессор произведет подстановку для этих макросов, все они займут соответствующие им места, и будут содержать точную информацию. Далее следует пример макроса, выдающего сообщение в случае стандартной ошибки:

```
#define msg(str) \
    fprintf(stderr,"File: %s Line: %d Function: %s\n%s\n", \
    __FILE__, __LINE__, __func__, str);
```

Для вызова этого макроса из любого места кода, нужно только определить строку описывающую ошибку:

```
msg("There is an error here.");
```

Другое преимущество такого способа состоит в том, что ваш метод обработки ошибочных условий может быть изменен простым изменением макроса. Его можно преобразовывать для того, чтобы "пробиваться" через исключения, или для ведения записей об ошибках в файле. Сообщение, производимое в рассмотренном нами примере, будет выглядеть примерно так:

```
File: Hamlink.c Line: 822 Function: hashdown
There is an error here
```

Временное удаление части исходного кода

Во время разработки программного обеспечения, зачастую возникает потребность в удалении блоков кода таким образом, чтобы позднее, если понадобится, их можно было бы восстановить. Код можно просто закомментировать, но это может вызвать проблемы, потому что в языке C комментарии не могут быть вложенными один в другой. А ведь в исключаемом из обработки коде может находиться какое-то количество комментариев, которые в таком случае нужно убирать. Чистый и безопасный способ исключения кода дает применение директив `#if` примерно следующим образом:

```
#if 0
    /* Закрываемый участок кода */
#endif
```

Такой способ позволяет не только свободно обращаться с комментариями, но и вполне целенаправленно убирать из области действия препроцессора тот или иной участок кода.

Создание компоновочных файлов (makefiles)

Препроцессор можно использовать для чтения исходного файла и создания строк определения зависимостей, передаваемых в **компоновочный файл** (или *make*-файл, англ. "makefile"). Для примера, следующая команда использует опцию `-E` для передачи указания компилятору вызвать препроцессор и затем остановиться без выполнения компиляции и компоновки (linking). Опция `-M` указывает препроцессору выдать полную строку действующих зависимостей:

```
gcc -E -M trick.c
```

Исходный файл `trick.c` содержит установки `#include` для системного заголовочного файла `<stdio.h>` и локального заголовочного файла `"barrow.h"`, однако список зависимостей включает не только эти файлы, но и все файлы, в свою очередь включаемые ими. Результирующая строка зависимостей примет такой вид:

```
trick.o: trick.c /usr/include/stdio.h \
/usr/include/features.h \
/usr/include/sys/cdefs.h /usr/include/gnu/stubs.h \
/usr/lib/gcc-lib/i386-redhat-linux/2.96/include/stddef.h \
/usr/include/bits/types.h \
/usr/include/bits/pthreadtypes.h \
/usr/include/bits/sched.h \
/usr/include/libio.h \
/usr/include/libio.h /usr/include/_G_config.h \
/usr/include/wchar.h \
```

```
/usr/include/bits/wchar.h /usr/include/gconv.h \
/usr/lib/gcc-lib/i386-redhat-linux/2.96/include/stdarg.h \
/usr/include/bits/stdio_lim.h barrow.h
```

Как разъясняется в приложении Г, опции **-MD**, **-MMD**, **-MF**, **-MG**, **-MP**, **-MQ** и **-MT** могут использоваться для создания зависимостей другими способами и в другом формате, отличном от применяемого при опции **-m**. Примеры использования этих опций для создания компоновочных файлов можно найти в главе 14.

Опции командной строки и переменные окружения

Существует набор опций командной строки **gcc**, определяющих способ действия препроцессора. Они приводятся в следующем списке и подробно описываются в приложении Г.

-A	-MF
-A-	-MG
--assert	-MM
-C	-MMD
-D	-MP
--define-macro	-MQ
--dependencies	-MT
-fident	-no-line-commands
-fpreprocessed	--no-standard-includes
-H	-nostdinc
-I	-nostdinc++
-I-	-P
-idirafter	--preprocess
-imacros	--print-missing-file-dependencies
-include	-remap
--include-barrier	--trace-includes
--include-directory	-trigraphs
--include-directory-after	-U
--include-prefix	-undef
--include-with-prefix	--undefined-macro
--include-with-prefix-after	--user-dependencies
--include-with-prefix-before	-Wp
-iprefix	--write-dependencies
-isystem	--write-user-dependencies
-iwithprefix	-Wsystem-headers
-iwithprefixbefore	-Wundef
-M	-Wunknown-pragmas
-MD	

Далее следует список переменных окружения, которые могут быть использованы в тех инструкциях препроцессора, которые связаны с именами каталогов и путей доступа. Переменные окружения описаны в приложении Б.

```
C_INCLUDE_PATH, CPATH, CPLUS_INCLUDE_PATH, DEPENDENCIES_OUTPUT,
OBJC_INCLUDE_PATH, SUNPRO_DEPENDENCIES
```

Глава 4



Компиляция программ на языке С

В этой главе описываются команды и опции команд, используемые для компиляции программ, написанных на алгоритмическом языке *C*, в объектные файлы, готовые к запуску программы и библиотеки. В главу также включено общее описание различных поддерживаемых в GCC стандартов языка программирования *C*, а также описание всех уникальных для GCC расширений языка *C*.

В системах UNIX оригинальный компилятор *C* называется `cc` ("C Compiler"). От него произошло название оригинального компилятора *C* проекта GNU, который был назван `gcc` ("GNU C Compiler"). Этот акроним остался до сих пор, но теперь его значение изменилось. Сейчас "GCC" означает "GNU Compiler Collection", что можно перевести как "сборный компилятор GNU" или как "набор компиляторов GNU". Это произошло в результате принципиального расширения компилятора — включения в него дополнительных языков. Однако базовой структурой GCC по-прежнему остается компилятор *C*. К счастью, структура языка *C* собственно основана на весьма низкоуровневых и довольно похожих на аппаратные действиях. Это позволяет в верхнем уровне программного обеспечения для компиляции кода надстраивать структуры компиляторов других языков на основе, поддерживающей язык *C*.

Базовая компиляция

В таблице 4.1 приводится список *суффиксов имен файлов* (иногда говорят "расширений имен файлов"), которые так или иначе связаны с компиляцией и компоновкой программ на языке *C*. Полный список распознаваемых GCC суффиксов приведен в приложении Г.

Таблица 4.1. Суффиксы имен файлов, связанные с языком C

Суффикс имени	Содержимое файла
.a	Статическая объектная библиотека (архив).
.c	Исходный код на языке C, подлежащий обработке препроцессором.
.h	Исходный код на языке C заголовочного файла (header file).
.i	Исходный код на языке C, не подлежащий предобработке. Файлы этого типа являются промежуточным продуктом компиляции.
.o	Объектный файл в формате, поддерживаемом компоновщиком (linker). Файлы этого типа являются промежуточным продуктом компиляции.
.s	Ассемблерный код. Файлы этого типа являются промежуточным продуктом компиляции.
.so	Разделяемая объектная библиотека. (Shared object library)

Преобразование отдельного исходного файла в готовую к запуску программу

Вот пример исходного кода простейшей программы "hello, world":

```
/* helloworld.c */
#include <stdio.h>
int main(int argc,char *argv[])
{
    printf("hello, world\n");
    return(0);
}
```

Простейшим и самым прямым способом компиляции этой программы в готовую к запуску форму является следующий способ. Сохранить этот исходный код в файле с именем **helloworld.c** и подать такую команду:

```
$ gcc helloworld.c
```

Проверив суффикс имени файла, компилятор определит, что указанный файл содержит исходный код. Согласно предопределенной по умолчанию последовательности действий GCC компилирует исходный файл в объектный, компонует его в готовый к запуску исполняемый файл и затем удаляет временный объектный файл. В команде не назначается имя файла для вырабатываемой выполнимой версии программы, поэтому компилятор назовет файл по умолчанию **a.out** и сохранит его в текущем каталоге. Ввод этого имени в командной строке вызовет запуск программы и вывод ею сообщения:

```
$ a.out
hello, world
```

Можно использовать опцию **-o** для назначения имени файла готовой к запуску программы, вырабатываемого компилятором. По следующей команде будет выработана выполнимая программа с именем **howdy**:

```
$ gcc helloworld.c -o howdy
```

Ввод этого имени программы в командной строке вызовет ее запуск и распечатает ее вывод:

```
$ howdy  
hello, world
```

Переработка исходного файла в объектный модуль

Опция `-c` указывает `gcc` скомпилировать исходный код и оставить на диске объектный файл, пропуская этап компоновки объектного файла в готовую к запуску программу. В этом случае имя выводимого файла по умолчанию будет таким же, как имя исходного, но только с суффиксом `.o`. Например, следующая команда выработает объектный файл с именем `helloworld.o`:

```
$ gcc -c helloworld.c
```

Возможно использование опции `-o` для переназначения имени вырабатываемого объектного файла. По следующей команде будет создан файл с именем `harumph.o`:

```
$ gcc -c helloworld.c -o harumph.o
```

Для построения объектных библиотек или просто, чтобы создать набор объектных файлов для их последующей компоновки, вы можете использовать одну команду для выработки объектных файлов из нескольких исходных. По следующей команде будут созданы объектные файлы с именами `arglist.o`, `ponder.o` и `listsort.o`:

```
$ gcc -c arglist.c ponder.c listsort.c
```

Преобразование нескольких исходных файлов в готовую к запуску программу

GCC выполняет компоновку автоматически даже при компиляции нескольких исходных файлов. К примеру, приведенный ниже исходный код сохранен в файле `hellomain.c` и он вызывает функцию `sayhello()`:

```
/* hellomain.c */  
void sayhello(void);  
int main (int argc,char *argv[]){  
    sayhello();  
    return(0);  
}
```

Далее — исходник функции `sayhello()`, который находится в файле `sayhello.c`:

```
/* sayhello.c */  
#include <stdio.h>  
void sayhello()
```

```
{
    printf("hello, world\n");
}
```

По следующей команде будет выполнена компиляция двух этих исходных файлов в объектные, компоновка их в готовую к запуску программу с именем `hello` и, затем, удаление временных объектных файлов:

```
$ gcc hellomain.c sayhello.c -o hello
```

Обработка исходного кода препроцессором

Опция `-E` указывает компилятору `gcc` запустить только препроцессор `cpp`. Следующая команда выполнит предобработку (preprocessing) исходного файла `helloworld.c` и выведет результат предобработки на стандартное устройство вывода.

```
$ gcc -E helloworld.c
```

Возможно использование опции `-o` для направления вывода предобработки кода в файл. Как ранее было показано в таблице 4.1, не нуждающийся более в предобработке исходный код на языке C сохраняется в файле с суффиксом `.i`. Это может быть выполнено в результате применения такой команды:

```
$ gcc -E helloworld.c -o helloworld.i
```

Выработка ассемблерного кода

Опция `-S` указывает компилятору выработать код на языке ассемблера и на этом остановиться. Следующая команда создаст файл с именем `helloworld.s` на ассемблерном языке текущей целевой машины из исходного файла `helloworld.c`:

```
$ gcc -S helloworld.c
```

Разновидность ассемблерного языка зависит от выбора целевой платформы. При компиляции нескольких исходных файлов вырабатывается отдельный модуль ассемблерного кода для каждого из них.

Создание статической библиотеки

Статическая библиотека является набором файлов типа `.o`, вырабатываемых компилятором обычным путем. Компоновка программы с объектными модулями библиотеки — то же самое, что и компоновка ее с объектными файлами, находящимися в каталоге. Другое название статической библиотеки — *архив*. Утилита, которая управляет содержимым статической библиотеки, называется `ar`.

Для построения статической библиотеки сначала необходимо скомпилировать все объектные модули, которые должны в нее войти. К примеру, следующий исходный код программы содержится в двух файлах `hellofirst.c` и `hellosecond.c`:

```
/* hellofirst.c */
#include <stdio.h>
```

```
void hellofirst()
{
    printf("The first hello\n");
}

/* hellosecond.c */
#include <stdio.h>
void hellosecond()
{
    printf("The second hello\n");
}
```

Два этих исходных файла могут быть скомпилированы в соответствующие им объектные файлы (с суффиксом .o) следующей командой:

```
$ gcc -c hellofirst.c hellosecond.c
```

Для создания новой библиотеки и вставки в нее объектных файлов следует использовать утилиту **ar** с опцией **-r**. Опция **-r** создает указанную библиотеку, если ее еще нет, и добавляет в нее перечисленные объектные модули, при необходимости заменяя уже присутствующие модули их новыми версиями. По следующей команде создается библиотека **libhello.a**, содержащая два объектных модуля:

```
$ ar -r libhello.a hellofirst.o hellosecond.o
```

Теперь библиотека вполне закончена и готова к использованию. Следующий пример программы, находящейся в файле **twohellos.c**, вызывает обе функции из созданной библиотеки:

```
/* twohellos.c */
void hellofirst(void);
void hellosecond(void);
int main(int argc,char *argv[])
{
    hellofirst();
    hellosecond();
    return(0);
}
```

Программа **twohellos** может быть откомпилирована и скомпонована одной командой компилятору с указанием библиотеки в командной строке:

```
$ gcc twohellos.c libhello.a -o twohellos
```

Существует соглашение об именах статических библиотек: они должны начинаться с трех букв **lib** и заканчиваться суффиксом **.a**. Все системные библиотеки соответствуют этому соглашению, что позволяет использовать в командной строке сокращенную форму имен библиотек, применяя опцию **-l** ("эль"). Следующая командная строка отличается от предыдущей только расположением, в котором **gcc** будет проводить поиск библиотеки **libhello.a**:

```
$ gcc twohellos.c -lhello -o twohellos
```

Если указан полный путь расположения, то компилятор будет искать библиотеку только в указанном каталоге. Имя библиотеки может быть назначено как с або-

лютым путем расположения (например, `/usr/worklibs/libhello.a`), так и относительно текущего каталога (например, `../lib/libhello.a`). Опция `-l` не дает возможности назначить путь расположения, вместо этого она указывает компилятору искать библиотеку в стандартных каталогах расположения системных библиотек.

Создание разделяемой библиотеки

Разделяемая библиотека (*shared library*) представляет собой набор объектных файлов, вырабатываемых компилятором особым образом. Все адреса (ссылки на переменные и вызовы функций) внутри объектных модулей являются относительными, а не абсолютными, что позволяет загружать и выполнять разделяемые модули динамически во время выполнения программы.

Для построения разделяемой библиотеки (*shared library*) сначала необходимо особым образом скомпилировать исходные файлы объектных модулей, которые должны в нее войти. К примеру, следующий исходный код содержится в двух исходных файлах `shellofirst.c` и `shellosecond.c`:

```
/* shellofirst.c */
#include <stdio.h>
void shellofirst()
{
    printf("The first hello from a shared library\n");
}

/* shellosecond.c */
#include <stdio.h>
void shellosecond()
{
    printf("The second hello from a shared library\n");
}
```

Два этих исходных файла могут быть скомпилированы в объектные следующей командой:

```
$ gcc -c -fpic shellofirst.c shellosecond.c
```

Опция `-c` указывает компилятору выработать объектные файлы. Применение опции `-fpic` назначает, что выходные объектные модули будут вырабатываться с использованием перемещаемой (*relocatable*) адресации. Акроним "pic" — сокращение от "position independent code", что означает "независимый от положения код".

Следующая команда `gcc` использует эти объектные файлы для построения разделяемой библиотеки `hello.so`:

```
$ gcc -shared shellofirst.o shellosecond.o -o hello.so
```

Опция `-o` присваивает имя выходному файлу и при этом суффикс `.so` дополнительно сообщает `GCC`, что объектные файлы должны быть скомпонованы в разделяемую библиотеку. Обычно компоновщик (*linker*) находит и использует функцию `main()` как точку входа в программу, однако эти объектные модули не имеют такой

точки входа, поэтому необходимо применение опции **-shared**, чтобы предотвратить вывод сообщения об ошибке.

Компилятор распознает исходный файл программы на языке C по суффиксу **.c** и он знает, как его скомпилировать в объектный файл. Благодаря этому две предыдущие команды можно объединить в одну. Исходные модули будут скомпилированы и сохранены непосредственно в разделяемой библиотеке по следующей команде:

```
$ gcc -fpic -shared shellofirst.c shellosecond.c -o hello.so
```

Следующая программа, сохраненная в файле **stwohellos.c**, — образец программы, вызывающей функции разделяемой библиотеки:

```
/* stwohellos.c */
void shellofirst(void);
void shellosecond(void);
int main(int argc,char *argv[])
{
    shellofirst();
    shellosecond();
    return(0);
}
```

Эта программа может быть скомпилирована и скомпонована с нашей разделяемой библиотекой по следующей команде:

```
$ gcc stwohellos.c hello.so -o stwohellos
```

Теперь программа **stwohellos** готова к запуску. Однако, для правильного выполнения она должна быть способной находить разделяемую библиотеку **hello.so**, потому что во время выполнения программы необходима динамическая загрузка модулей подпрограмм, хранящихся в этой библиотеке. Информация о правилах размещения разделяемых библиотек содержится в главе 12.

Замещение соглашений об именах

В случае обстоятельств, требующих использования исходных файлов C с отличным от **.c** суффиксом имени, применяется опция **-x** для замещения предполагаемого по умолчанию суффикса. Эта опция применяется для указания языка. В следующем примере команды компилируется исходный код на языке C из файла **helloworld.jxj** и создается готовая к запуску программа с именем **helloworld**:

```
$ gcc -xc helloworld.jxj -o helloworld
```

Обычно, без применения опции **-x**, предполагается, что любой исходный файл с нераспознанным суффиксом имени известен компоновщику, и этот файл передается ему с тем же неизмененным именем. Опция **-x** применяется ко всем встречающимся в командной строке именам файлов с неизвестными расширениями. Например, в следующей команде оба файла **align.zzz** и **types.xxx** обрабатываются как исходные файлы на языке C:

```
$ gcc -c -xc align.zzz types.xxx
```

Поддержка стандартов языка

Используя опции командной строки, вы можете компилировать любые программы на языке *C* от написанных в первоначально известном синтаксисе "K&R C" (часто сейчас называемом "традиционным *C*") до самых последних стандартов языка с подключением расширений GNU. По умолчанию GCC компилирует исходный код, используя правила последнего известного стандарта с подключением всех расширений GNU. Доступные опции перечислены в таблице 4.2. Приложение Г содержит более детальное описание каждой из этих опций.

Таблица 4.2. Опции управления применяемой версией языка *C*

Опция	Описание
<code>-ansi</code>	Компилирует программы, как соответствующие стандартам, так и использующие расширение GNU.
<code>-pedantic</code>	Выдает предупреждения при любых отклонениях от строгого соответствия стандартам.
<code>-std=c89</code>	Стандарт ISO C-89.
<code>-std=c99</code>	Стандарт ISO C-99.
<code>-std=gnu89</code>	Стандарт ISO C-89 с расширениями GNU и некоторыми свойствами ISO C-99.
<code>-traditional</code>	Строгое соответствие правилам традиционного стандарта языка <i>C</i> ("K&R C").

Наиболее фундаментальные различия между соответствующими стандарту и несоответствующими ему программами на языке *C* лежат в формах передачи аргументов вызываемым функциям и в присутствии или отсутствии прототипов функций. Для подавления подобных проблем в GCC имеется опция `-aux-info`, которая может быть использована для автоматической генерации прототипов функций. Далее следует пример команды, которая создает заголовочный файл `slmwrk.h`, содержащий прототипы для всех определяемых в исходном файле `slmwrk.c` функций:

```
$ gcc slmwrk.c -aux-info slmwrk.h
```

А следующая команда может быть использована для создания прототипов (они будут помещены в `prototypes.h`) для функций всех исходных файлов на языке *C* в текущем каталоге:

```
$ gcc *.c -aux-info prototypes.h
```

Функции в программах на языке *C* могут быть преобразованы к форме стандарта "ANSI C" использованием утилиты `protoize`. Это более подробно описано в главе 14.

Расширения GNU языка *C*

Компилятор GCC не только может быть установлен на работу в соответствии с правилами одного из стандартов языка *C* использованием таких опций как `-ansi` или `-std`, но также и может использовать некоторые собственные расширения. Многие из расширений GNU языка *C* прошлых версий компилятора GCC вошли в новые стандарты языка *C*. В следующих разделах рассмотрены лишь те расширения GCC,

которые не являются частью известных стандартов. За исключением нескольких особых случаев, рассматриваемые расширения языка применяются только в GCC.

При назначении опции `-pedantic`, равно как и некоторых других, использование расширений GNU вызовет выдачу компилятором предупредительных сообщений. Однако, вы можете подавлять выдачу таких предупреждений, если используете ключевое слово `__extension__` перед расширенными выражениями.

Благодаря особой внутренней структуре GCC, многие из описываемых здесь расширений применимы в *C++* и *Objective-C*, так же как и в *C*. Компиляторы *C++* и *Objective-C* используют части компилятора *C*, а это значит, что дополнения языка *C* и препроцессора CPP в ряде случаев распространяются также и на другие языки. Однако, некоторые из расширений языка *C* конфликтуют с основными определениями других языков, поэтому они отключены или несколько изменены в языках *C++* и *Objective-C*.

Выравнивание (Alignment)

Оператор `__alignof__` возвращает границы выравнивания типа или отдельного элемента данных. Следующая программа показывает выравнивание каждого из типов данных:

```
/* align.c */
#include <stdio.h>
typedef struct {
    double dvalue;
    int ivalue;
} showal;

int main(int argc,char *argv[])
{
    printf("__alignof__(char)=%d\n", __alignof__(char));
    printf("__alignof__(short)=%d\n", __alignof__(short));
    printf("__alignof__(int)=%d\n", __alignof__(int));
    printf("__alignof__(long)=%d\n", __alignof__(long));
    printf("__alignof__(long long)=%d\n", __alignof__(long long));
    printf("__alignof__(float)=%d\n", __alignof__(float));
    printf("__alignof__(double)=%d\n", __alignof__(double));
    printf("__alignof__(showal)=%d\n", __alignof__(showal));
    return(0);
}
```

Действующие правила выравнивания типов изменяются в зависимости от возможностей применяемой аппаратной базы. Они могут либо определяться абсолютными требованиями применяемого оборудования либо устанавливаться в настройках предлагаемых границ выравнивания для более эффективного доступа к данным.

Безымянные (анонимные) объединения (Anonymous Unions)

Внутри структуры (`struct`) возможно объявление объединения (`union`) без имени, что делает возможным адресоваться к членам этого объединения напрямую, так

как если бы они были членами структуры. В следующем примере один блок из четырех байт обеспечивается двумя именами и двумя типами данных:

```
struct {
    char code;
    union {
        char chid[4];
        int numid;
    };
    char *name;
} morx;
```

Члены этой структуры могут быть адресованы как `morx.code`, `morx.child`, `morx.numid` и `morx.name`.

Массивы переменной длины (Arrays of Variable Length)

Массив может быть объявлен так, что его размер определяется во время выполнения программы. Это достигается использованием выражения в качестве вложенного в объявление сценария. Например, следующая функция принимает две строки и объединяет их в одну, вставляя между ними пробел:

```
void combine(char *str1,char *str2)
{
    char outstr[strlen(str1) + strlen(str2) + 2];
    strcpy(outstr,str1);
    strcat(outstr," ");
    strcat(outstr,str2);
    printf("%s\n",outstr);
}
```

Массив переменной длины может передаваться как аргумент, как это показано в следующем примере:

```
void fillarray(int length,char letters[length])
{
    int i;
    char character = 'A';
    for(i=0; i<length; i++)
        letters[i] = character++;
}
```

Порядок аргументов может быть обращен благодаря выполнению упреждающего объявления таким образом, что тип `length` известен к тому времени, когда читается массив `letters`, что показано в следующем примере:

```
void fillarray (int length; char letters[length], int length)
```

Вы можете иметь столько подобных упреждающих объявлений, сколько вам нужно (разделенных запятыми или точкой с запятой; после последнего из них должна стоять точка с запятой).

Массивы нулевой длины (Arrays of Zero Length)

Имеется расширене GNU языка C, которое допускает объявление массивов нулевой длины, что обеспечивает создание структур с переменной длиной. Это имеет смысл только в том случае, когда массив нулевой длины объявляется как последний член структуры (struct). Размер массива может назначаться простым резервированием необходимой под него памяти. Этот способ демонстрируется в следующей программе:

```
/* zarray.c */
#include <stdio.h>
typedef struct {
    int size;
    char string[0];
} vlen;
int main(int argc,char *argv[])
{
    int i;
    int count = 22;
    char letter = 'a';
    vlen *line = (vlen *)malloc(sizeof(vlen) + count);
    line->size = count;
    for(i=0; i<count; i++)
        line->string[i] = letter++;
    printf("sizeof(vlen)=%d\n", sizeof(vlen));
    for(i=0; i<line->size; i++)
        printf("%c ",line->string[i]);
    printf("\n");
    return(0);
}
```

В этом примере стандартная функция `printf()` выводит значение 4 потому что оператор `sizeof()` может определить только размер типа `int` объявленного поля структуры. Вывод программы `zarray` выглядит примерно так:

```
sizeof(vlen)=4
a b c d e f g h i j k l m n o p q r s t u v
```

То же самое может быть достигнуто и определением массива как неполного типа. Этот подход имеет не только преимущество стандартного способа языка C, но также он может использоваться таким же способом, как и массив в предыдущем примере. Дополнительное преимущество состоит в том, что массив может быть определен в инициализаторах. В этом примере размер массива устанавливается равным четырем буквенным символам:

```
/* incarray.c */
#include <stdio.h>
typedef struct {
    int size;
    char string[];
}
```

```

} vlen;
vlen initvlen = { 4, { 'a', 'b', 'c', 'd' } };
int main(int argc,char *argv[])
{
    int i;
    printf("sizeof(vlen)=%d\n",sizeof(vlen));
    printf("sizeof(initvlen)=%d\n",sizeof(initvlen));
    for(i=0; i<initvlen.size; i++)
        printf("%c ",initvlen.string[i]);
    printf("\n");
    return(0);
}

```

Далее приводится вывод этой программы:

```

sizeof(vlen)=4
sizeof(initvlen)=4
a b c d

```

Атрибуты (Attributes)

Ключевое слово `__attribute__` может использоваться для назначения атрибутов при объявлении функций или данных. Основной целью назначения атрибутов является предоставление компилятору возможности выполнения оптимизации. Атрибуты функции назначаются в объявлении прототипа, как в следующем примере:

```

void fatal_error() __attribute__ ((noreturn));
. . .
void fatal_error(char *message)
{
    fprintf(stderr,"FATAL ERROR: %s\n",message);
    exit(1);
}

```

В этом примере атрибут `noreturn` сообщает компилятору, что эта функция не возвращает управление вызывающей ее структуре. Благодаря этому обычно вставляемый код возврата из функции будет пропущен при оптимизации.

В одном объявлении может быть назначено несколько атрибутов. Они помещаются в список (через запятую). В следующем примере объявление функции назначает атрибуты, сообщающие компилятору, что эта функция не изменяет глобальных переменных и что она не должна в дальнейшем подставляться `inline`.

```
int getlim() __attribute__ ((pure,noinline));
```

Атрибуты могут назначаться отдельно переменным и полям структур. Например, чтобы гарантировать определенное выравнивание поля внутри структуры, его можно объявить следующим образом:

```
struct mong {
    char id;
```

```

    int code __attribute__ ((align(4)));
}

```

Таблица 4.3 перечисляет набор атрибутов функций. Таблица 4.4 содержит атрибуты, допустимые при объявлении данных, в таблице 4.5 перечислены атрибуты, которые могут назначаться при объявлении типов данных.

Таблица 4.3. Атрибуты, используемые при объявлении функций

Атрибут	Описание
<code>alias</code>	Объявление функции с этим атрибутом приводит к тому что это объявление становится неприоритетной альтернативой (weak alias) другой функции. Для этого он может применяться в сочетании с атрибутом <code>weak</code> , как в следующем примере, где функция <code>centon()</code> создается как альтернатива для <code>__centon()</code> :
	<pre> int __centon() { return(100); } void centon() __attribute__ \ ((weak,alias("__centon"))); </pre> <p>В языке C++ при объявлении основной функции должно быть также указано имя ее заменяющей (mangled name of the target). Этот атрибут действует не для всех целевых машин.</p>
<code>always_inline</code>	Функция, объявленная как <code>inline</code> , и имеющая этот атрибут будет всегда расширяться подстановкой кода даже в том случае, когда не назначается никакой оптимизации. Обычно подстановка кода функций производится во время оптимизации. Далее — пример прототипа функции, вызовы которой всегда будут расширяться подстановкой кода:
	<pre>inline void infn() __attribute__ ((always_inline));</pre>
<code>const</code>	Функция с таким атрибутом имеет те же свойства, что и с атрибутом <code>pure</code> , но кроме этого она не считывает никаких значений из глобальной памяти. Это дает оптимизатору большую свободу чем атрибут <code>pure</code> , потому что отпадает необходимость в проверке изменения значений глобальных величин перед вызовом функции.
<code>constructor</code>	Функция с этим атрибутом вызывается автоматически до передачи управления главной подпрограмме <code>main()</code> . См. также атрибут <code>destructor</code> .
<code>deprecated</code>	Не актуальные более функции (либо по каким-нибудь причинам не соответствующие) целесообразно компилировать с этой опцией. При каждом вызове такой функции будет выдаваться предупредительное сообщение, включающее информацию о расположении такой (<code>deprecated</code>) функции и сообщающее пользователю источник более подробных сведений.
<code>destructor</code>	Функция с этим атрибутом вызывается автоматически после возврата из процедуры <code>main()</code> либо сразу после вызова функции <code>exit()</code> . См. также атрибут <code>constructor</code> .
<code>format</code>	Функция с таким атрибутом имеет один аргумент, являющийся строкой формата, и переменное число аргументов, к которым применяется этот формат. Это дает компилятору возможность проверять содержимое аргументов из списка на соответствие его типа применяемому формату. Существует несколько типов форматирования, поэтому необходимо также указать стиль формата. Стили соответствуют применяемым в языке C для стандартных функций <code>printf()</code> , <code>scanf()</code> , <code>strftime()</code> и <code>strfmon()</code> . Например, следующий атрибут указывает, что второй аргумент, передаваемый функции, является форматирующей строкой, и что эта форматирующая строка имеет тип формата, соответствующий команде <code>printf()</code> , а также, что список аргументов переменной длины начинается с третьего аргумента:

Атрибут	Описание
	<pre>int logprintf (void *log, char *fmt, ...) \ __attribute__ ((format.printf, 2, 3)));</pre> <p>Предупреждения о несоответствии форматирующей строки указанному типу вырабатываются при назначении опции компилятора <code>-Wformat</code>.</p>
<code>format_arg</code>	<p>Функция с таким атрибутом принимает в одном из аргументов форматирующую строку и вносит такие изменения в передаваемую ею строку, чтобы результат мог быть затем передан функции типа <code>printf()</code>, <code>scanf()</code>, <code>strftime()</code>, <code>strfmon()</code>. Использование этого атрибута подавляет предупреждения, вырабатываемые при установке опции компилятора <code>-Wformat-nonliteral</code> (опция применяется для обнаружения неконстантных форматирующих строк). Следующий пример показывает установку этого атрибута для функции, имеющей вторым аргументом строку формата:</p> <pre>void fedit (int ndx, const *fmt) \ __attribute__ ((format_arg(2)));</pre>
<code>malloc</code>	<p>Этот атрибут сообщает компилятору, что функция может применяться подобно стандартной функции <code>malloc()</code>. При выполнении оптимизации компилятор учитывает, что возвращаемый указатель не должен иметь синонимов (aliases).</p>
<code>no_instrument_function</code>	<p>Функция с таким атрибутом не будет инструментирована и в нее не будет включаться профилирующий код, вставляемый компилятором. Даже при применении опции <code>-finstrument-functions</code>.</p>
<code>noinline</code>	<p>Функция с таким атрибутом не будет расширяться подстановкой кода (<code>inline</code>).</p>
<code>noreturn</code>	<p>Функция с этим атрибутом не возвращает управление вызывающей ее структуре.</p>
<code>pure</code>	<p>Функция с таким атрибутом не оказывает влияния на другие величины, кроме возвращаемого ею значения. То есть, она не изменяет глобальных переменных, передаваемых в аргументах адресов, содержимого файлов. В отличие от атрибута <code>const</code> функция с этим атрибутом использует значения глобальных переменных, но все же не изменяет их. Применение этих условий дает компилятору возможность выполнения общей оптимизации применяемых в функции выражений, так как гарантируется постоянство используемых в них внешних значений.</p>
<code>section</code>	<p>Функция с таким атрибутом будет иметь именованные разделы (секции) ассемблерного кода (вместо одного раздела <code>text</code> по умолчанию). Вот пример заголовка функции, имеющей раздел с именем <code>specials</code>:</p> <pre>void mspec (void) __attribute__ ((section("specials")));</pre> <p>Этот атрибут будет игнорироваться в системах, не поддерживающих секционирование.</p>
<code>used</code>	<p>Этот атрибут указывает компилятору генерировать код тела функции даже тогда, когда он определяет, что функция нигде не используется. Это может пригодиться для функций, вызываемых из вставляемого ассемблерного кода.</p>
<code>weak</code>	<p>Функция с таким атрибутом будет иметь замещаемое символьическое имя вместо глобального символьического имени. В основном это применяется для библиотечных функций, которые могут замещаться пользовательским кодом.</p>

Таблица 4.4. Атрибуты, используемые при объявлении данных

Атрибут	Описание
<code>aligned</code>	Переменная с этим атрибутом имеет выравнивание к адресу, равное целой части указанного числа. Например, следующее объявление указывает, что к <code>alivalue</code> применяется выравнивание по границе 32-бита: <code>int alivalue __attribute__ ((aligned(32)));</code> Применение выравнивания удобно в некоторых системах для размещения определенных ассемблерных инструкций. Оно также применяется для полей структуры, которые должны принимать форматированные данные из файла. Если число для выравнивания не указано, то компилятор применяет выравнивание к границе наиболее длинного типа данных, обрабатываемого процессором, как в следующем примере:
<code>short shlist [312] __attribute__ ((align));</code>	
<code>deprecated</code>	Каждая ссылка на переменную, объявленную с этим атрибутом, будет вызывать выдачу компилятором предупредительного сообщения.
<code>mode</code>	Переменной с этим атрибутом назначается размер, соответствующий указанному в параметре атрибута типу (<code>byte</code> , <code>word</code> или <code>pointer</code>). Атрибут <code>mode</code> также определяет и тип данных. Например, так объявляется переменная целочисленного типа размером в 1 байт: <code>int x __attribute__ ((mode(byte)));</code>
<code>nocommon</code>	Переменная с таким атрибутом не размещается в общей области данных, вместо этого ей выделяется собственное пространство. Значение переменной при этом инициализируется нулями. Назначение опции компилятора <code>-fno-common</code> приводит к применению этого атрибута ко всем переменным.
<code>packed</code>	С этим атрибутом переменная имеет наименьшее возможное выравнивание. Переменная будет отделена от предшествующего ей поля не более чем одним байтом. В структуре поле с таким атрибутом будет размещаться без промежутка между ним и следующим за ним полем. Например, в следующей структуре начало массива с именем <code>zar</code> выравнивается со сдвигом точно на один байт от вершины структуры: <code>struct zrecord { char id; int zar [32] __attribute__ ((packed)); };</code> См. также опции <code>-fpack-struct</code> и <code>-Wpacked</code> в приложении Г.
<code>section</code>	Переменная с этим атрибутом будет помещена в именованный раздел вместо разделов по умолчанию <code>data</code> или <code>bss</code> . Далее — пример функции, помещаемой в секцию именем <code>domx</code> : <code>struct domx __attribute__ ((section("domx")))= {0}; int trigger __attribute__ ((section("MONLOG")))= 0 ;</code> Вследствие способа действий над данными, применяемого компоновщиком, данные, размещаемые в именованных секциях, должны иметь первоначальное значение. Этот атрибут будет игнорироваться в системах, не поддерживающих секционирования. Принудительная инициализация переменных может быть включена при необходимости опцией командной строки <code>-fno-common</code> .
<code>unused</code>	С таким атрибутом переменная может нигде в программе не использоваться, и по этому поводу компилятор не будет выдавать никаких предупреждений.

Атрибут	Описание
<code>vector_size</code>	Переменной с таким атрибутом будет отведено пространство, общий размер которого определяется как размер комплексного числа. Вот пример объявления вектора данных с плавающей точкой: <pre>float fvec __attribute__ ((vector_size(32)));</pre> Учитывая, что тип данных <code>float</code> имеет длину 4 байта, мы видим, что это объявление создает блок общей длиной 32 байта, который может содержать 8 чисел с плавающей точкой. Этот атрибут применим только для целых и вещественных скалярных чисел.
<code>weak</code>	Переменная с таким атрибутом будет иметь имя, выделенное как замещаемый (слабый) символ вместо глобального имени. В основном это применяется для библиотечных объектов, которые могут быть замещены пользовательским кодом.

Таблица 4.5. Атрибуты, используемые при объявлении типов данных

Атрибут	Описание
<code>aligned</code>	Тип, объявленный с этим атрибутом, имеет выравнивание в памяти, равное целой части числа, которое указанно в параметре атрибута. Например, поля следующей структуры будут выравниваться по границе 32 бит: <pre>struct blockm { charj [3]; } __attribute__ ((aligned(32)));</pre> To же выравнивание может быть получено при применении атрибута <code>aligned</code> к первому члену структуры. Атрибут <code>aligned</code> может использоваться только для увеличения выравнивания, не для его уменьшения. Некоторые компоновщики могут устанавливать компилятору ограничение на наибольшее возможное значение выравнивания.
<code>deprecated</code>	Тип, объявленный с таким атрибутом, вызывает предупредительное сообщение компилятора, выдаваемое каждый раз, когда этот тип используется в объявлениях. Сообщение включает информацию о расположении объявления этого типа.
<code>packed</code>	Структура (<code>struct</code>) или объединение (<code>union</code>), объявленные с этим параметром, занимают наименьший возможный размер в памяти. Это равнозначно применению атрибута <code>packed</code> для каждого члена структуры или объединения.
<code>transparent_</code> - <code>union</code>	Объединение (<code>union</code>), объявляемое с этим атрибутом, и используемое как тип параметра при объявлении функции, делает для этой функции возможным воспринимать в качестве аргумента любой из типов данных, определенных в таком объединении. Далее предлагается пример использования "прозрачного" объединения для вызова одной и той же функции с тремя различными типами аргументов: <pre>/* transp.c */ #include <stdio.h> typedef union { float *f; int *i; } fourbytes __attribute__ ((transparent_union)); void showboth(fourbytes fb);</pre>

Атрибут	Описание
	<pre> int main(int argc,char *argv[]) { int ivalue = 2562; float fvalue = 898.44; fourbytes fb; fb.i = &ivalue; showboth(&ivalue); showboth(&fvalue); showboth(fb); return(0); } void showboth(fourbytes fb); { printf("The int value: %d\n",*fb.i); printf("The float value: %f\n",*fb.f); } </pre> <p>Функция <code>showboth()</code> в приведенном примере объявлена как требующая объединения <code>fourbytes</code> в качестве аргумента. Однако, благодаря тому, что это объединение объявлено с атрибутом <code>transparent_union</code>, любые типы, объявленные в объединении, также могут передаваться в аргументе функции. Этот пример содержит вызовы указанной функции, передающие ей адрес числа с плавающей точкой, адрес целого числа и адрес собственно объединения.</p>
<code>unused</code>	Применение такого атрибута при объявлении типа приводит к тому, что использование данных этого типа не ожидается. Поэтому какие-либо предупреждения из-за их неиспользования не будут выдаваться компилятором.

Составные операторы, возвращающие значение (Compound Statements Returning a Value)

Составные операторы (compound statement) — это блок операторов, заключенный в фигурные скобки. Они имеют собственную область действия и в них могут объявляться собственные локальные переменные. Пример:

```
{
    int a = 5;
    int b;
    b = a + 5;
}
```

В GNU C составной оператор, окруженный скобками, вырабатывает возвращаемое значение; как в следующем примере, где возвращаемое значение равно 8:

```
rslt = ({
    int a = 5;
    int b;
    b = a + 3;
});
```

Возвращаемое значение имеет тот же тип и значение результата, что и выражение в последнем операторе блока.

Эта конструкция может быть весьма удобна при написании макросов. Иногда могут возникать проблемы, например, когда макрос имеет в качестве аргумента выражение, определяемое более одного раза. В следующем примере макрос возвращает значение, равное либо большее заданному, при необходимости инкрементируя его значение:

```
#define even(x) (2*(x / 2) == x ? x : x + 1)
```

Это будет работать, несмотря на возможность побочного эффекта вычисления значения **x**. Например, следующее выражение может давать неопределенный результат:

```
int nexteven = even(value++);
```

Следующее макроопределение выполняет ту же задачу, но делает это, определяя значение выражения **x** один раз и сохраняя его во временной переменной:

```
#define even(x) \  
({ int y = x; \  
    (2*(y / 2) == y ? y : y + 1); \  
})
```

Следует заметить, что описываемое расширение не будет корректно работать в C++. Если вы будете использовать этот прием в заголовочных файлах, включаемых в программы на языке C++, то это может вызвать проблемы. Трудности происходят оттого, что деструкторы временных переменных, используемых внутри макроса, запускаются раньше, чем это действует в inline-функциях, расширяемых подстановкой кода.

Условный пропуск операнда (Conditional Operand Omission)

В условных выражениях ложное либо истинное условие определяется как нулевой либо ненулевой результат проверяемого выражения. Может случится так, что проверяемое выражение определяется и как результат всего условного выражения. Например, в следующем операторе, **x** будет присвоено значение **y**, если **y** не равно нулю:

```
x = y ? y : z;
```

Выражение **y** будет вычисляться второй раз, если при первом вычислении оно оказалось отличающимся от нуля. Есть способ пропустить повторное вычисление выражения, как это показано в следующем примере:

```
x = y ? : z;
```

Это может очень пригодиться, если вычисление выражения **y** имеет побочный эффект и не должно определяться более одного раза в пределах одного оператора.

Неполные перечисляемые типы (Enum Incomplete Types)

Ярлык `enum` может быть объявлен без указания списка значений, так же как и имя структуры может объявляться без назначения содержащихся в ней полей. Неполные перечисления могут использоваться в прототипах функций и для объявления указателей.

В примере показано объявление неполного перечисления, сопровождающееся затем его действительным объявлением:

```
enum color_list;
.
.
.
enum color_list { BLACK, WHITE, BLUE },
```

Построение аргументов функций (Function Argument Construction)

Три приводимые встроенные функции могут использоваться для прямой передачи аргументов текущей функции в другую функцию и последующего возвращения результата в первоначальную вызывающую структуру. Для функции, передающей аргументы, вовсе не обязательно что-либо о них знать.

Следующая функция передает и записывает описывающую аргументы информацию:

```
void * __builtin_apply_args(void);
```

Когда информация об аргументах записана, может быть использована следующая функция, чтобы построить стэк информации, необходимой для вызова, и, собственно, осуществить вызов нужной функции:

```
void * __builtin_apply(void (*func) (), void *arguments, int size);
```

Первый аргумент — адрес функции, передаваемый как адрес функции, не имеющей аргументов и не возвращающей значение. Второй — результат процесса записи, выполненного функцией `__builtin_apply_args()`. Аргумент `size` — количество байт для копирования из текущего кадра стэка (stack frame) в новый кадр стэка, он должен иметь достаточный размер для передачи всех аргументов вместе с возвращаемым адресом.

После того как вызвана функция `__builtin_apply()` возвращаемое ею значение будет позиционировано в стэке. Следующая функция выравнивает кадр стэка и возвращает управление первоначальной вызывающей структуре:

```
__builtin_return (void *result);
```

Приведенная далее в примере программа вызывает функцию `pastthrough()`, использующую описанные встроенные функции для вызова `average()` и возвращения ее результата:

```
/* args.c */
#include <stdio.h>
```

```

int passthrough();
int average();

int main(int argc,char *argv[])
{
    int result;

    result = passthrough(1,7,10);
    printf("result=%d\n",result);
    return(0);
}

int passthrough(int a,int b,int c)
{
    void *record;
    void *playback;
    void (* fn)() = (void (*)())average;

    record = __builtin_apply_args();
    playback = __builtin_apply(fn,record,128);
    __builtin_return(playback);
}

int average(int a,int b,int c) {
    return((a + b + c) / 3);
}

```

Обратите внимание, что функции `passthrough()` известны аргументы и возвращаемое значение только потому, что это ее собственные аргументы и результат. Эта функция может быть преобразована в оболочку для обобщенного вызова функций, если применить троеточие для создания открытого списка аргументов и объявить указатель на элемент данных (void pointer) для передачи возвращаемого значения. Тогда `passthrough()` можно применять для передачи произвольного набора аргументов в функцию, известную по ее адресу. Возвращаемое значение также будет произвольного типа.

Для написания такой обобщенной оболочки (generalized wrapper), необходимо учитывать, что величина `size`, сообщаемая функции `__builtin_apply()`, должна быть достаточной для того, чтобы вместить все аргументы.

Расширение вызовов функций подстановкой кода (Function Inlining)

Функция может быть объявлена как расширяемая подстановкой (`inline`), и ее код будет подставляться в месте ее вызова подобно тому, как подставляются макросы. Для объявления такой функции используется ключевое слово `inline`:

```

inline int halve(double x)
{
    return (x / 2.0);
}

```

Далее приводится список правил, действующих на расширяемые подстановкой функции:

- Ни одна функция не будет в действительности подставливаться своим кодом определения, если вы не примените при компиляции опцию `-O`, назначающую тот или иной уровень оптимизации. Это правило действует для упрощения применения программы-отладчика (debugger). Несмотря на это можно принудительно включать обязательное применение подстановки атрибутом `always_inline`.
- В результате объявления функции как расширяемой подстановкой кода, объем программы может увеличиться или уменьшиться в зависимости от размера кода функции, сложности установки кадра стека для вызова функции (`call frame`), и количества вызовов функции в коде программы.
- Некоторые функции не могут быть расширены подстановкой. Среди таких — функции, использующие переменное количество аргументов, `alloca`, массивы переменной длины, удаленные (не локальные) переходы по `goto`, а также имеющие вложенные функции. Также возникают сложности в рекурсивных и имеющих ссылки на собственный адрес функциях. Опция командной строки `-winline` выдает предупреждения во всех случаях невозможности подстановки кода функций, объявленных как `inline`.
- В программах на языке C стандарта "ISO C" вы можете использовать ключевое слово `__inline__` вместо `inline`.
- Можно использовать опцию командной строки `-finline-functions` для предоставления компилятору возможности самостоятельно принимать решение о применении подстановки для функций, отвечающих действующим для этого условиям.
- Если `inline`-функция не объявлена как статическая (`static`), то объектный код ее тела все-же должен быть сгенерирован компилятором на случай вызова этой функции из другого модуля. Обявление функции одновременно как `inline`, так и `static`, приведет к тому, что она будет подставливаться во всех случаях, поэтому отдельный объектный код для нее вырабатываться не будет. Опция `-fkeep-inline-functions` замещает это правило и тогда тело функции всегда создается.
- Обявление функции в заголовочном файле (header file) одновременно как `extern`, так и `inline` — почти то же самое, что и определение макроса. Другая копия той же функции (без квалификаторов `extern` и `inline`) может быть скомпилирована и сохранена в объектной библиотеке так, что будут разрешимыми все ссылки на нее, не расширяемые подстановкой.

Имена функций (Function Names)

Идентификатор `__FUNCTION__` содержит имя той функции, в которой он применяется. Имя представляется в форме строки буквенных символов и может соединяться с другими строками точно так же, как значения `__FILE__` или `__LINE__`. Следующий пример оператора собирает в одной строке текст с информацией о своем расположении в исходном коде, используя все три этих макроса:

```
char *here = "Line " __LINE__ " of " __FILE__ " in " __FUNCTION__;
```

Опубликованный в стандарте ISO C99 идентификатор `__func__` также содержит имя текущей функции, но только в виде массива символов (array of char), а не строки буквенных символов.

Замечание

Семантика `__FUNCTION__` считается более не актуальной, планируется изменение этого встроенного идентификатора — приведение его в соответствие семантике `__func__`.

В языке C идентификатор `__PRETTY_FUNCTION__` содержит то же значение, что и `__FUNCTION__`, но следует учитывать, что в языке C++ их содержимое отличается по форме представления.

Использование вложенных функций (Function Nesting)

Функции могут вкладываться друг в друга. Внутренняя функция может быть вызвана только в пределах родительской функции. Следующий пример функции содержит вложенную функцию с именем `randint()`, возвращающую псевдослучайное целое число в заданных пределах:

```
void rangers()
{
    int randint(int low,int high) {
        return((int)random() % (high-low+1)) + low;
    }

    printf("0 to 100: %d\n",randint(0,100));
    printf("5 to 10: %d\n",randint(5,10));
    printf("-1 to 1: %d\n",randint(-1,1));
}
```

При применении вложенных функций действуют следующие правила:

- Вложенная функция создает свой стэк примерно таким же образом, как и переменные. Поэтому она может быть объявлена в блоке (перед первым исполняемым оператором блока) — в том же месте, где объявляются локальные переменные.
- Адрес вложенной функции не может быть передан вызывающей родительскую функцию структуре. Как и любая другая локальная переменная, вложенная функция исчезает после того, как родительская функция возвращает управление.
- Встроенная функция не может быть вызвана извне.
- Внутри родительской функции возможна передача адреса встроенной функции другой функции и возможен вызов ее оттуда. Так же, как в пределах функции возможно обращение по адресу к ее любой локальной переменной.
- Встроенная функция имеет прямой доступ к тем же переменным, что и родительская.

- Локальные переменные доступны встроенной функции лишь если они объявлены перед ней.
- Встроенная функция может использовать переход по оператору `goto` на внешние метки, но только в пределах родительской функции.
- Прототип встроенной функции может объявляться с ключевым словом `auto`, см. пример:

```
void right()
{
    auto double hypotenuse();
    double a = 3.0;
    double b = 4.0;

    double hypotenuse(double x,double y) {
        return(sqrt(x * x + y * y));
    }

    printf("Long side of %lf and %lf is %lf\n",
           a,b,hypotenuse(a,b));
}
```

Прототипы функций (Function Prototypes)

Новое определение прототипа функции замещает действующее определение любой функции только при сохранении ранее объявленного списка аргументов. Вот пример корректного замещения, аргумент короткого целочисленного типа автоматически расширяется до полного целочисленного при вызове функции:

```
int trigzed(int zvalue);
. . .
int trigzed(zvalue)
short zvalue;
{
    return(zvalue == 0);
}
```

В случае объявления с новым синтаксисом `int trigzed(short zvalue)`, компилятор будет выдавать сообщение об ошибке из-за конфликта между типами аргументов `int` и `short`.

Адреса возврата из функций и кадры стэка (Function Return Addresses and Stack Frames)

Существует встроенная функция, используемая для получения адреса, который используется функцией для выполнения возвращения в вызывающую ее конструкцию:

```
void *__builtin_return_address (unsigned int level);
```

Указывая значение `level` равное нулю, мы получаем адрес возврата, используемый текущей функцией. При значении `level` равном 1 — адрес возвращения, действующий в той функции, из которой была вызвана текущая функция; значении 2 —

адрес возвращения из внешней функции следующего уровня (второго по отношению к текущей). И так далее — до тех пор, пока весь стэк вызова не будет исчерпан. Функция `__builtin_return_address()` может применяться для расчета глубины стэка вызова. Учтите, что `level` должен передаваться только как константа, не как переменная.

В некоторых системах нет возможности получения адресов возврата никаких функций, кроме текущей. В зависимости от особенностей платформы на таких системах получаемое значение адресов (для `level = 1` и выше) равно нулю либо имеет случайное значение.

Следующая встроенная функция используется для получения адреса кадра стэка функции (stack frame address):

```
void * __builtin_frame_address (unsigned int level);
```

Указывая `level` равный нулю, мы получаем адрес кадра стэка текущей функции. `level` равный 1 — адрес кадра стэка функции, из которой была вызвана текущая. При значении 2 — адрес кадра стэка следующей внешней функции, и так далее, пока не будет исчерпан весь стэк вызова. Встроенная функция `__builtin_frame_address()` может быть применена для нахождения глубины стека вызова.

Кадр стэка — это блок памяти, содержащий значения регистров, сохраненные вызываемой функцией, и значения аргументов, передаваемых функции. Точный формат *кадра* зависит от применяемых для вызова функций соглашений и от платформы.

На некоторых системах не существует возможности получения адресов кадров стэка функций выше текущего уровня. В таких случаях при указании `level` больше нуля рассматриваемая встроенная функция возвращает нулевое значение адреса. Также возвращается нулевой адрес и в случаях, когда `level` превышает глубину стэка вызова.

Идентификаторы (Identifiers)

Идентификаторы могут содержать знак доллара ("\$"). Это необходимо для совместимости со многими традиционными компиляторами C и также связано с большим объемом используемых программ на языке C, имеющих имена переменных и функций со знаком доллара. Использование знака доллара допустимо не для всех систем, некоторые ассемблеры его не воспринимают.

Целые числа (Integers)

Стандартом "ISO C99" определены целочисленные типы длиной более 64 бит. Компилятор GCC поддерживает их, начиная с более ранних версий языков C и C++. Вот примеры объявления таких типов:

```
long long int a;           // Целое 64 бит со знаком
unsigned long long int b;   // Целое 64 бит без знака
```

Могут также объявляться константы любого из этих типов:

```
a = 855LL;    // Целочисленная константа 64 бит со знаком
b = 855ULL;   // Целочисленная константа 64 бит без знака
```

Арифметические операции сложения, вычитания и Булевые битовые операции могут производиться над этими типами на любых машинах. А умножение, деление и битовые сдвиги поддерживаются не на всяком оборудовании и могут требовать использования особых библиотечных подпрограмм.

Важно использовать прототипы с определениями соответствующих типов аргументов, если вы намерены использовать их в аргументах вызова функций. Без прототипа размер и положение переменных в стэке вызова может оказаться неправильным.

Альтернативные формы ключевых слов (Keyword Alternates)

Опции командной строки `-std` и `-ansi` исключают использование ключевых слов `asm`, `typeof` и `inline`, однако возможно использование их альтернативных форм `__asm__`, `__typeof__` и `__inline__`.

Адреса меток (Label Addresses)

Существует возможность получения адреса метки, сохранения его в указателе и последующего перехода на нее по оператору `goto` с этим указателем. Адрес может быть получен оператором `&&` и сохранен в указателе на элемент данных (void pointer).

Вот вам пример, показывающий как это можно сделать:

```
/* gotoaddr.c */
#include <stdio.h>
#include <time.h>
int main(int argc,char *argv[])
{
    void *target;
    time_t now;

    now = time((time_t *)NULL);
    if(now & 0x0001)
        target = &&oddtag;
    else
        target = &&eventag;
    goto *target;

eventag:
    printf("The time value %ld is even\n",now);
    return(0);
oddtag:
    printf("The time value %ld is odd\n",now);
    return(0);
}
```

Псевдослучайное число использовано для принятия решения, адрес которой из меток будет сохранен в `target`. Оператор затем воспринимает адрес указателя типа `void` в качестве допустимого адреса для перехода (`jmp`). Указатель не может быть корректно использован для перехода на адрес метки внутри другой функции.

Благодаря тому, что в операторе `goto` может быть использовано любое выражение, возвращающее `void`-указатель, можно создать массив адресов меток и перейти на них по индексу:

```
void *loc[] = { &&label1, &&label2, &&label3, &&label4 };
. . .
goto *loc[i];
```

Локально объявляемые метки

Есть возможность такого объявления метки, что она будет определена лишь в границах определенной области действия. Ключевое слово `__label__` используется в начале области действия (например, за открывающей операционной скобкой "`{`") для объявления локальной метки, действующей только в пределах этой области. Вот пример программы, демонстрирующей объявление и использование двух локальных меток:

```
/* loclabel.c */
#include <stdio.h>
int main(int argc,char *argv[])
{
    int count = 0;
    {
        __label__ restart;
        __label__ finished;
        restart:
        printf("count=%d\n",count);
        if(count > 5)
            goto finished;
        count++;
        goto restart;
        finished:
        count += 10;
    }
    return(0);
}
```

Можно объявлять несколько меток (через запятую):

```
__label__ restart, finished;
```

Использование такого рода меток может быть полезным внутри кода, расширяемого по макроопределению. Добавляя операционные скобки для отделения области действия можно избежать пересечения областей в случае применения разных меток с одинаковыми именами внутри одной функции.

Составные выражения в левой части оператора присваивания, Lvalue (Lvalue Expressions)

Составные выражения могут использоваться с левой стороны оператора присваивания (это и есть "*lvalue*", то есть "значение левой части"). Это справедливо для того случая, когда доступен адрес результата составного выражения. Обычной формой

lvalue является имя переменной. В следующем примере переменная *a* — имя расположения в памяти, куда будет записано числовое значение 5:

```
a = 5;
```

В другой форме своего применения *lvalue* в действительности оказывается на правой стороне оператора, но остается самим собой, потому что правая сторона присваивания в этом случае является адресом памяти, а не значением. В примере переменная *a* используется как *lvalue*:

```
ptr = &a;
```

При известных особых обстоятельствах составные выражения также могут использоваться как *lvalue*.

Вот список свойств и правил для составления выражений *lvalue*:

- Составное выражение, может использоваться как *lvalue* если может быть получен адрес последнего члена этого выражения. В примере показано два равносильных оператора:

```
(fn(), b) = 10;  
fn(), (b = 10);
```

- Имеется возможность получения адреса составного выражения. Его адресом будет адрес последнего члена выражения. В этом примере в *ptr* будет записан адрес переменной *b* (иначе говоря, *lvalue b*):

```
ptr = &(fn(), b);
```

- Можно применять *lvalue* как условные выражения в случае, если оба варианта выбора являются корректными выражениями *lvalue*. В этом примере оператор присваивает *b* значение 100, и оно больше чем константа 5. При этом *c* присваивается значение 100:

```
((a > 5) ? b : c) = 100;
```

- *lvalue* поддерживает приведение к другому типу. В предлагаемом примере указатель *chptr* типа *char* приводится к *int* типа указателю, в который записывается абсолютный адрес константы "894":

```
char *chptr;  
(int) chptr = 894;
```

Макроопределения с переменным количеством аргументов (Macros with Variable Arguments)

Существует два способа для создания макросов с переменным количеством аргументов, потому что один из них — собственное расширение GCC, а другой, несколько отличающийся способ, соответствует стандарту "ISO C99".

Далее — пример, использующий метод стандарта ISO:

```
#define errout(fmt, ...) fprintf(stderr,fmt,__VA_ARGS__)
```

Любой список аргументов, указываемый после *fmt*, будет подставляться вместо *__VA_ARGS__* везде, где бы он не встречался в теле макроопределения. При ис-

пользовании синтакса GNU C тот же макрос будет определяться следующим образом:

```
#define errout(fmt, args ...) fprintf(stderr, fmt, args)
```

Более подробно о макроопределениях и расширениях макросов смотрите в главе 3.

Строки (Strings)

Переход на новую строку может вставляться в строку литер без использования *escape-последовательности* "\n". Он может передаваться прямо из исходного кода. Два предлагаемых примера строковых литералов **str1** и **str2** идентичны:

```
char *str1 = "A string on\ntwo lines"
char *str2 = "A string on
two lines"
```

Escape-код "\e" соответствует специальной лите "ESC" кодировки ASCII. Последовательность "\e" также может использоваться в строковых литералах.

Как обычно, обратная наклонная черта в конце строки исходного кода объединяет эту строку с последующей:

```
char *str3 = "This string will \
be joined into one line.";
```

Это вполне соответствует стандарту языка C. Расширение GNU C упрощает неудобно строгое правило языка C о том, что при объединении строк символ перехода строки должен следовать сразу же за обратной наклонной чертой. GCC разрешает оставлять любое количество пробелов после обратной наклонной черты. Лишние пробелы убираются компилятором и строки объединяются, но при этом препроцессор выдает предупредительное сообщение.

Арифметические действия над указателями (Pointer Arithmetics)

Для указателей на данные (void pointers) и указателей на функции (function pointers) поддерживаются операции сложения и вычитания. Указатель инкрементируется или декрементируется с учетом размера того типа, на который он указывает. В GCC указатели на функции и данные инкрементируются или декрементируются на 1. Это является результатом того, что оператор **sizeof** для указателя всегда возвращает 1.

Опция **-Wpointer-arith** специально используется для вывода предупредительных сообщений об использовании этого расширения.

Операторы Switch и Case

Диапазон значений может назначаться с помощью троеточия. Вот пример стандартного ("ANSI C") способа выбора из четырех значений для обычного оператора **case**:

```
case 8:  
case 9:  
case 10:  
case 11:
```

То же самое можно написать с использованием троеточия:

```
case 8 ... 11:
```

Важно отметить, что троеточие должно быть окружено пробелами. Это необходимо, чтобы предотвратить ошибку работы синтаксического разделителя (parserегр), состоящую в том, что точки могут быть восприняты в качестве десятичного разделителя константы. Также и пересечение диапазонов в операторах **case** (как и дублирование простых констант в них) приведет к выдаче компилятором сообщения об ошибке:

```
case 8 ... 15:  
...  
case 12 ... 32: // ОШИВКА!
```

Этот способ особенно удобен при указании оператору **case** диапазона буквенных констант:

```
case 'a' ... 'm':
```

Создание имени определяемого типа (Typedef Name Creation)

Ключевое слово **typedef** может использоваться для создания имени для типа данных результата выражения. Определенное таким образом имя может использоваться для объявления или приведения типа переменных, они будут иметь тот же тип, что и результат выражения. Новое имя типа определяется так:

```
typedef name = expression;
```

Например, следующие операторы определяют **smallreal** как числовой тип с плавающей точкой и **largereal** как вещественное число с двойной точностью **double**:

```
typedef smallreal = 0.0f ;  
typedef largereal = 0.0 ;
```

Эти новые имена могут в дальнейшем использоваться для объявления переменных представляемого ими типа. В следующем примере операторов (стоящих дальше в коде той же программы) **real1** объявляется как **float** и **real2** как **double**:

```
smallreal real1;  
largereal real2;
```

Одним из удобных мест использования этого расширения являются макроопределения, которые должны применяться к нескольким типам данных. Следующий макрос не получает предварительных сведений о типах аргументов, переменные в макросе получают тип аргументов и могут поэтому обмениваться с ними своим значением:

```
#define swap (a, b)          \
({ typedef _tp = a; \
  _tp temp = a; \
  a = b; \
  b = temp; })
```

Для определения имени локального типа `_tp` используется тип первого аргумента. Локальная переменная `temp` объявляется как локальное временное расположение данных типа `_tp`, что дает возможность обмена значениями двум переменным независимо от их типов.

Ссылки на типы переменных (Typeof References)

Ключевое слово `typeof` возвращает тип выражения. Оно используется подобно оператору `sizeof`, но в отличие от него возвращает тип вместо размера. Вот — пример использования:

```
char *chptr;                                // Указатель на char
typeof (*chptr) ch;                          // Тип char
typeof (ch) *chptr2;                         // Указатель на char
typeof (chptr) charray[10];                  // 10 указателей на char
typeof (*chptr) chararray[10];                // 10 переменных типа char
typeof (ch) chararray2[10];                   // 10 переменных типа char
```

В этом примере `chptr` объявляется как указатель (pointer) на переменную типа `char`. Используя `typeof` для определения типа, на который указывает `chptr`, `ch` объявляется соответственно как переменная типа `char`. И наоборот, используя тип данных переменной `ch`, объявляется `chptr2` как указатель (pointer) на переменную типа `char`. Переменная `charray` объявляется как массив из десяти указателей на данные типа `char`. Объявление массива `chararray` основывается на типе указателя на переменную `chptr`, таким образом, являясь объявлением массива десяти переменных `char`. Другой массив десяти переменных типа `char` основывается на типе переменной с именем `ch`.

В следующем примере объявляется переменная того же типа, что и результат, возвращаемый функцией:

```
char func();
typeof (func) retval;
```

Функция `func()` возвращает результат типа `char`, поэтому выражение `typeof` объявляет переменную с именем `retval` типа `char`.

В выражениях `typeof` вы можете использовать непосредственно имена типов. Следующий пример содержит два равносильных оператора:

```
char *charptr;
typeof (char *) charptr;
```

Применение `typeof` дает возможность создавать макроопределения, используемые для объявления переменных. В следующем примере определяется макрос, используемый затем для создания двух массивов — массива десяти переменных `double`, и массива десяти переменных `float`:

```
#define array(type, size) typeof(type[size])
array(double, 10) dblarray;
array(float, 10) fltarray;
```

Приведение типов объединения (Union Casting)

Тип элемента данных объединения (union) может быть применен для приведения к нему всего объединения в целом. Например, в следующей программе объединение, содержащее элемент типа `double`, приводится к типу `double`, затем программа обращается к каждому байту через ссылку на объединение:

```
/* unioncast.c */
#include <stdio.h>
union dparts {
    unsigned char byte[8];
    double dbl;
};

int main(int argc,char *argv[])
{
    int i;
    double value = 3.14159;
    for(i=0; i<8; i++) {
        printf("%02X ",((union dparts)value).byte[i]);
    }
    printf("\n");
    return(0);
}
```

Приведение объединения также может быть использовано в качестве аргумента вызова функции:

```
void procun(union dparts);
. . .
procun((union dparts) value);
```

Приведение объединения несколько отличается от других приведений типов. В действительности оно является конструктором и поэтому не может иметь значения *lvalue*. Это делает ошибочными операторы подобные следующему:

```
(union dpart)value.dbl = 1.2; // ОШИБКА!
```



Глава 5

Компиляция программ на языке C++

Компилятор GNU C++ является полнофункциональным компилятором, вырабатывающим готовый к выполнению объектный код на своем выходе. Первоначальный компилятор C++, выпущенный компанией AT&T, назывался `cfront` и в действительности был транслятором (прекомпилятором) кода на языке C++ в код на C. До сих пор некоторые компиляторы C++ действуют именно так. GCC был изначально компилятором C, поэтому язык C++ был в него добавлен без особых проблем в качестве выбираемого режима компиляции.

Базовая компиляция

В таблице 5.1 приводится список суффиксов имен файлов, задействованных в компиляции и компоновке (linking) программ на языке C++. Полный список распознаваемых GCC суффиксов приведен в приложении Г.

Таблица 5.1. Суффиксы имен файлов, связанных с компиляцией C++

Суффикс	Содержимое файла
.a	Статическая объектная библиотека (архив).
.c, .c++, .cc, .cp, .cpp, .cxx	Исходный код на языке C++, подлежащий обработке препроцессором.
.h	Включаемый в программы заголовочный файл (header file) на языке C или C++.
.ii	Исходный код программы на языке C++, не подлежащий предобработке. Файлы этого типарабатываются на промежуточном этапе компиляции.
.o	Объектный файл в формате, поддерживаемом компоновщиком (linker). Файлы этого типарабатываются на промежуточном этапе компиляции.

Суффикс	Содержимое файла
.в	Исходный код на ассемблере. Файлы этого типа вырабатываются на промежуточном этапе компиляции.
<поле>	Разделяемая (динамическая) объектная библиотека. (Shared object library.) Стандартные динамические библиотеки основного набора функций языка C++ имеют имена без суффикса.

Компиляция отдельного исходного файла в готовую к запуску программу

В качестве примера вам предлагается исходный код простейшей программы на языке C++, сохраните его в файле `helloworld.cpp`:

```
/* helloworld.cpp */
#include <iostream>
int main(int argc,char *argv[])
{
    std::cout << "hello, world\n";
    return(0);
}
```

Для выдачи отдельной строки на стандартное устройство вывода эта программа использует функцию `cout`, определенную в стандартном заголовочном файле `iostream`. В готовую для запуска форму эта программа может быть скомпилирована следующей командой:

```
$ g++ helloworld.cpp
```

Компилятор `g++` распознает файл по суффиксу его имени как исходный файл на языке C++. В соответствии с действующим предопределением он должен скомпилировать указанный исходный файл в объектный, далее скомпоновать объектный файл в готовую к запуску машинную программу, и затем удалить объектный файл. В связи с тем, что в командной строке не определено имя для выходного файла, по умолчанию будет создан файл с именем `a.out`. Команда для запуска готовой программы и ее вывод выглядят так:

```
$ a.out
hello, world
```

Имя для выполнимого системой файла, в который должен быть скомпилирован исходник, передается компилятору опцией `-o`. По этой команде будет выработан файл с именем `helloworld`:

```
$ g++ helloworld.cpp -o helloworld
```

Ввод этого имени в командной строке приведет к выполнению программы:

```
$ helloworld
hello, world
```

Программа `g++` является особым вариантом утилиты `gcc`, она устанавливает C++ в качестве основного языка компиляции, что приводит к автоматическому исполь-

зованию стандартных библиотек языка *C++* вместо применяемых по умолчанию библиотек языка *C*. При использовании соглашений об именах библиотек и указании соответствующей библиотеки в командной строке возможна компиляция и компоновка программы на языке *C++* и командой *gcc*. Например, такой:

```
$ gcc helloworld.cpp -lstdc++ -o helloworld
```

Опция *-l* ("эль") заменяет следующее за ней имя, подставляя к нему префикс *lib* и суффикс *.a*, что превращает имя указанной библиотеки в *libstdc++.a*. Затем поиск библиотеки с таким именем проводится в каталогах для стандартных библиотек. Дальнейший процесс компиляции и выходной файл точно соответствуют действию команды *g++*.

На многих системах при инсталляции GCC создается программа с именем *c++*. Ее использование вполне равносильно действию *g++*. Вот пример такой команды:

```
$ c++ helloworld.cpp -o helloworld
```

Преобразование нескольких исходных файлов в готовую к запуску программу

Если в командной строке указывается несколько исходных файлов, то все они компилируются и компонуются вместе в один исполняемый файл. Далее — пример заголовочного файла (header file) с именем *speak.h*, этот файл содержит определение класса, состоящего из одной единственной функции:

```
/* speak.h */
#include <iostream>
class Speak
{
public:
    void sayHello(const char *);
};
```

Далее — листинг файла *speak.cpp*, содержащего код реализации функции *sayHello()*:

```
/* speak.cpp */
#include "speak.h"
void Speak::sayHello(const char *str)
{
    std::cout << "Hello " << str << "\n";
}
```

Файл *hellospeak.cpp* содержит программу, использующую класс *Speak*:

```
/* hellospeak.cpp */
#include "speak.h"
int main(int argc,char *argv[])
{
    Speak speak;
    speak.sayHello("world");
    return(0);
}
```

Для компиляции и компоновки этих трех файлов в единственный исполняемый файл используется одна команда:

```
$ g++ hellospeak.cpp speak.cpp -o hellospeak
```

Компиляция исходного кода в объектный

Для компиляции исходного кода с подавлением компоновки и, соответственно, вывода объектного файла вместо исполняемого применяется опция **-c**. По умолчанию имя вырабатываемого файла совпадает с именем исходного, только имеет суффикс **.o**. Например, по следующей команде исходный файл **hellospeak.cpp** будет скомпилирован в объектный файл **hellospeak.o**:

```
$ g++ -c hellospeak.cpp
```

Программа **g++** также распознает файлы с суффиксом **.o** в качестве входных и передает их компоновщику (**linker**):

```
$ g++ -c hellospeak.cpp  
$ g++ -c speak.cpp  
$ g++ hellospeak.o speak.o -o hellospeak
```

Следует заметить, что опция **-o** используется не только для присвоения имен исполняемым файлам. Ее также можно использовать для назначения имен и другим выходным файлам компилятора. Например, в результате следующей последовательности команд будет создан тот же файл с исполняемым кодом, что и в предыдущем примере. С той лишь разницей, что имена промежуточных объектных файлов будут другими:

```
$ g++ -c hellospeak.cpp -o hspk1.o  
$ g++ -c speak.cpp -o hspk2.o  
$ g++ hspk1.o hspk2.o -o hellospeak
```

Предобработка

Назначение опции **-E** указывает драйверу **g++** пропустить исходный код через препроцессор CPP и не производить никаких дальнейших действий. По следующей команде будет выполнена последовательная предобработка исходного кода из файла **helloworld.cpp** и выдача результата предобработки на стандартное устройство вывода:

```
$ g++ -E helloworld.cpp
```

Исходный код программы в файле **helloworld.cpp** состоит всего из шести строк и эта программа только и делает, что выводит строку текста. Но на выходе препроцессора будет более 1200 строк. Так много потому, что будет включен заголовочный файл **iostream.h**, а он, в свою очередь, включит в код еще несколько заголовочных файлов. В них определяется несколько больших классов, отвечающих за ввод и вывод.

Суффикс имени файла, принятый в GCC для результатов предобработки — `.ii`. Такой файл может быть выработан при использовании опции `-o`, как в следующем примере:

```
$ gcc -E helloworld.cpp -o helloworld.ii
```

Выработка компилятором ассемблерного кода

Опция `-S` указывает компилятору сгенерировать код на языке ассемблера и на этом остановиться. Следующая команда создает файл с именем `helloworld.s` с ассемблерным кодом из исходного файла на языке C++:

```
$ g++ -S helloworld.cpp
```

Выработанный ассемблерный код зависит от типа целевой платформы. Если вы будете просматривать полученный в результате такой компиляции код, то увидите в нем не только выполняемые инструкции и определения данных, но также и адресные таблицы, необходимые для обеспечения наследования и компоновки программ на языке C++.

Создание статической библиотеки

Статическая библиотека (static library) — это архивный файл, содержащий набор выработанных компилятором объектных файлов. Составляющие библиотеку части могут содержать применяемые функции, определения классов и объекты, являющиеся вхождениями экземпляров (instances) определений классов. В действительности все, что может содержаться в объектном файле с суффиксом `.o`, может быть помещено в библиотеку.

В следующем примере создаются два объектных модуля, используемые затем для составления статической библиотеки. Заголовочный файл (header file) содержит необходимые программе функции, определения классов и объекты.

Заголовочный файл `say.h` содержит прототип функции `sayHello()` и определение класса с именем `Say`. Далее приводится содержимое этого файла:

```
/* say.h */
#include <iostream>
void sayHello(void);
class Say {
private:
    char *string;
public:
    Say(char *str)
    {
        string = str;
    }
    void sayThis(const char *str)
    {
        std::cout << str << " from a static library\n";
    }
    void sayString(void);
};
```

Следующий файл с именем **say.cpp** — исходник первого из двух объектных модулей, которые должны быть помещены в библиотеку. В нем содержится тело определения функции **sayString()** класса **Say**. Также он содержит объявление объекта **librarysay**, который является экземпляром класса **Say**:

```
/* say.cpp */
#include "say.h"
void Say::sayString()
{
    std::cout << string << "\n";
}
Say librarysay("Library instance of Say");
```

Файл **sayhello.cpp** содержит исходный код второго модуля, который также должен быть помещен в библиотеку. В нем находится определение функции **sayHello()**:

```
/* sayhello.cpp */
#include "say.h"
void sayHello()
{
    std::cout << "hello from a static library\n";
}
```

Приведенная далее последовательность команд выполняет компиляцию двух исходных файлов в объектные и затем сохраняет их с помощью утилиты **ar** в статической библиотеке:

```
$ g++ -c sayhello.cpp
$ g++ -c say.cpp
$ ar -r libsay.a sayhello.o say.o
```

Утилита **ar**, использованная с опцией **-r**, создает новую библиотеку с именем **libsay.a** и помещает в нее перечисленные в команде объектные модули. При таком способе применения **ar** создает новую библиотеку с указанным именем, если таковой еще не существует. При наличии указанной библиотеки утилиты **ar** заменяет в ней существующие объектные модули их новыми версиями.

Далее — содержимое главного исходного файла программы **saymain.cpp**, использующей статическую библиотеку **libsay.a**:

```
/* saymain.cpp */
#include "say.h"
int main(int argc,char *argv[])
{
    extern Say librarysay;
    Say localsay = Say("Local instance of Say");
    sayHello();
    librarysay.sayThis("howdy");
    librarysay.sayString();
    localsay.sayString();
    return(0);
}
```

Эта программа компилируется и компонуется приведенной далее командой. В команде определяется, что драйвер верхнего уровня компилятора `g++` разрешает все внешние ссылки, имеющиеся в `saymain.cpp`, через поиск определений для требуемых символических имен в библиотеке `libsay.a`:

```
$ g++ saymain.cpp libsay.a -o saymain
```

Внешнее отношение к `librarysay` ссылается на объект, объявленный в `libsay.cpp` и хранящийся в библиотеке `libsay.a`. Как `librarysay.sayThis()`, так и `librarysay.sayString()` являются вызовами методов объекта библиотеки. Также и `sayHello()` является вызовом функции из объектного файла `sayHello.o`, который также находится в этой библиотеке. При запуске программы происходит вывод следующих строк:

```
hello from a static library
howdy from a static library
Library instance of Say
Local instance of Say
```

Создание разделяемой библиотеки

Разделяемая библиотека (shared library) также содержит набор объектных файлов. Только ее объектные модули должны использовать относительную адресацию, чтобы их код мог быть динамически загружен в любую область памяти и запущен оттуда без его избыточного перемещения. Это позволяет загружать исполняемый код из разделяемой библиотеки во время выполнения программы, вместо его статической компоновки в исполняемый файл.

Приведенный далее заголовочный файл с именем `average.h` определяет класс, который должен быть сохранен в разделяемой библиотеке:

```
/* average.h */
class Average {
private:
    int count;
    double total;
public:
    Average(void) {
        count = 0;
        total = 0.0;
    }
    void insertValue(double value);
    int getCount(void);
    double getTotal(void);
    double getAverage(void);
};
```

Исходный файл `average.cpp`, который также должен быть скомпилирован и сохранен в разделяемой библиотеке, содержит код функций, определенных в классе `Average`:

```
/* average.cpp */
#include "average.h"
```

```
void Average::insertValue(double value)
{
    count++;
    total += value;
}
int Average::getCount()
{
    return(count);
}
double Average::getTotal()
{
    return(total);
}
double Average::getAverage()
{
    return(total / (double)count);
}
```

Исходные файлы компилируются в объектные, которые затем используются для построения разделяемой библиотеки. Это происходит при выполнении следующих двух команд:

```
$ g++ -c -fPIC average.cpp
$ gcc -shared average.o -o average.so
```

Первая команда использует опцию `-c`, поэтому компилятор вырабатывает объектный файл `average.o` и не предпринимает попытки его компоновки в исполнимый машинный код. Опция `-fPIC` ("pic" — сокращение от англ. "Position Independent Code" — "независимый от положения код") указывает компилятору вырабатывать пригодный для разделяемой библиотеки код. Т.е. такой код, в котором внутренняя адресация вычисляется относительно начального адреса загрузки модуля в память. Вторая команда использует опцию `-shared`, что приводит к созданию разделяемой библиотеки с именем `average.so` (указанным опцией `-o`). Для второй команды можно применить как команду `g++`, так и `gcc`. Создание разделяемой библиотеки из объектных файлов при компиляции C++ не имеет никаких особенных отличий от общей процедуры. Для сборки библиотеки из нескольких файлов достаточно их просто перечислить в командной строке.

Две предыдущие команды можно объединить в одну. По ней исходные файлы компилируются в объектные, из которых затем создается разделяемая библиотека:

```
$ g++ -fPIC -shared average.cpp -o average.so
```

Следующая программа использует определение класса, находящееся в разделяемой библиотеке, для создания экземпляра объекта, который применяется для помещения в него 4-х значений и расчета их общей суммы:

```
/* showaverage.cpp */
#include <iostream>
#include "average.h"
int main(int argc,char *argv[])
{
    Average avg;
```

```
avg.insertValue(30.2);
avg.insertValue(88.8);
avg.insertValue(3.002);
avg.insertValue(11.0);
std::cout << "Average=" << avg.getAverage() << "\n";
return(0);
}
```

Далее приведена команда для компиляции и компоновки программы с разделяемой библиотекой. По этой команде вырабатывается файл `showaverage` с исполняемым машинным кодом:

```
$ g++ showaverage.cpp average.so -o showaverage
```

Для запуска программы библиотека `average.so` должна быть установлена в каталоге, разрешимом для поиска разделяемой библиотеки во время выполнения программы. Подробнее об этом — в главе 12.

Расширения языка C++

Этот раздел разъясняет применяемые в GNU расширения стандартных правил языка C++. Компилятор C++ весьма сложен, и документ, определяющий его стандарт, имеет довольно большой объем. Здесь, конечно, описаны не все расширения языка и не все расхождения со стандартом. Тут также опущены многие расширения, унаследованные от компилятора GNU C и применяемые при компиляции в GCC программ на языке C++ (они перечислены в главе 4).

Атрибуты (Attributes)

В главе 4 приведен список атрибутов, используемых в программах на языке C. Эти атрибуты применимы также и для программ на языке C++. Кроме них имеются особые атрибуты, используемые только в C++. Атрибут применяется указанием ключевого слова `__attribute__` со следующим за ним заключенным в скобки именем атрибута. В таблице 5.2 содержится описание только особых специально разработанных для языка C++ атрибутов, поддерживаемых в GCC.

Таблица 5.2. Список особых атрибутов, используемых в программах на языке C++

Атрибут	Описание
<code>init_priority</code>	Стандарт C++ определяет, что инициализация объектов происходит в том порядке, в котором они расположены в компилируемом модуле (<code>unit</code>). В стандарте нет особой спецификации о последовательности их инициализации между модулями. Атрибут <code>init_priority</code> предоставляет возможность указания, в какой последовательности должна происходить инициализация объектов в пределах выделяемой области памяти. Объектам при их объявлении назначаются приоритеты в соответствии с указанными числами. Приоритет тем выше, чем меньшее число указано в параметре атрибута. В следующем примере три объекта будут инициализированы в таком порядке — сначала <code>B</code> , затем <code>C</code> и только потом <code>A</code> независимо от того, в каком модуле они находятся:

Атрибут	Описание
<code>SpoClass A __attribute__ ((init_priority(680)));</code> <code>SpoClass B __attribute__ ((init_priority(220)));</code> <code>SpoClass C __attribute__ ((init_priority(400)));</code>	Числа, указанные в параметре атрибута, не имеют никакого особыго значения, кроме их величины относительно друг друга.
<code>java_interface</code>	Атрибут указывает, что класс должен быть определен как интерфейс Java. Этот атрибут применим только к классам, имеющим внутри себя определение блока <code>extern "Java"</code> . Вызов методов класса, определенного таким способом, использует интерфейс GCJ таблицы виртуальных функций C++.

Включаемые заголовочные файлы (Header Files)

Все системные заголовочные файлы включаются по умолчанию, если они заключены в блок `extern "C" { ... }`. Это вызывает проблемы, когда заголовочный файл содержит код на языке C++. Однако эти проблемы легко разрешить, если применить директивой препроцессора следующую прагму:

```
#pragma cplusplus
```

При применении такой прагма-директивы в заголовочном файле дальнейший код компилируется так, как если бы он был заключен в блок `extern "C++" { ... }`. Прямое использование этой прагмы непосредственно в блоке `extern "C" { ... }` является ошибкой.

Имена функций

Идентификатор `__FUNCTION__` содержит имя текущей функции, как в языке C, так и в C++. В языке C++ `__PRETTY_FUNCTION__` также содержит имя функции, но только в форме, предоставляющей больше информации. В примере показано применение этих идентификаторов, а также идентификатора `__func__`, определенного стандартом языка C:

```
/* showfuncname.cpp */
#include <iostream>
class Xyz
{
public:
    void NameShow(int i,double d)
    {
        std::cout << "__FUNCTION__\n      "
                  << __FUNCTION__ << "\n";
        std::cout << "__PRETTY_FUNCTION__\n      "
                  << __PRETTY_FUNCTION__ << "\n";
        std::cout << "__func__\n      "
                  << __func__ << "\n";
    }
};

int main(int argc,char *argv[])
{
```

```
xyz xyz;
xyz.Nameshow(5,5.0);
return(0);
}
```

При выполнении эта программа выводит такие строки:

```
__FUNCTION__
NameShow
__PRETTY_FUNCTION__
void Xyz::NameShow (int, double)
__func__
NameShow
```

Идентификаторы `__FUNCTION__` и `__func__` определены как строки, содержащие только само имя текущей функции. Идентификатор `__PRETTY_FUNCTION__` содержит полное имя текущей функции, включая тип возвращаемого результата, имя класса и список типов аргументов.

Объявление класса и код его реализации (Interface и Implementation)

Объявление класса и код его реализации можно объединить в одно целое. Это означает, что нет необходимости в отдельном объявлении прототипа класса, потому что код, полностью реализующий класс, может применяться в его интерфейсном определении.

Это достигается выбором директивы `#pragma interface` для указания определения класса только в качестве его интерфейса (т.е. объявления прототипа) или `#pragma implementation` для указания GCC компилировать определения функций и данных класса в объектный код.

Замечание

Это очень удобная "фича" и ее используют немало программистов, но в скором времени планируется ее замена. Похоже, что в будущей версии GCC от этой пары прагм откажутся, и для достижения того же результата будет применяться другой механизм.

Для использования этой пары прагм можно применять следующую последовательность действий:

1. Создайте заголовочный файл, содержащий полную реализацию класса. Например, файл для класса `MaxHolder` может быть назван `maxholder.h`.
2. В начале заголовочного файла и после определения класса поставьте строку:
`#pragma interface`
3. В любом исходном файле, ссылающемся на класс `MaxHolder`, включайте заголовочный файл обычным способом.
4. В одном из исходных файлов (обычно это делается в файле, содержащем главную процедуру) перед директивой `#include` вставьте следующую прагма-директиву:

```
#pragma implementation "maxholder.h"
#include "maxholder.h"
```

Файлы, включающие заголовочный файл (header file) обычным способом, будут подключать только интерфейсное определение класса, в то время как один файл с директивой `#pragma implementation` подключит полный исходный код класса, тот код, который будет скомпилирован в объектный формат. Это значит, что будет создан только один набор расширяемых подстановкой `inline`-функций, один набор отладочной информации и одна внутренняя таблица виртуальных функций *visible*.

- Если имя заголовочного файла совпадает с именем файла, содержащем реализацию, то нет необходимости указывать имя прагмы. Например, если файл `maxholder.cpp` подключает заголовок `maxholder.h`, то прagma-директиву можно написать просто `#pragma implementation`.
- Если один заголовочный файл подключает другой заголовочный файл из другого каталога, то в прagma-директиве интерфейса указывается путь: `#pragma implementation "subdirectory/filename.h"`. Если это делается так, то же имя файла должно быть и в прагме реализации.
- Директива `#include` должна применяться обязательно, включение заголовочных файлов прagma-директивами не выполняется.
- В результате подключения по `#pragma interface` функции класса действуют, как объявленные с ключом `extern`, тело функции используется только в том случае, если функция расширяется подстановкой (`inline`).
- В результате применения `#pragma implementation` подстановливаемые кодом `inline` функции имеют не расширяемые подстановкой скомпилированные версии на случай их вызова из модулей, не содержащих кода для их подстановки. Это правило может быть отменено применением в командной строке опции `-fno-implement-inlines`.

Операторы <? и >?

Для выбора наименьшего и наибольшего значения из двух аргументов доступны специальные операторы.

Приведенный пример выражения возвращает наименшее из двух значений `a` и `b`:

```
minvalue = a <? b;
```

А выражение в следующем примере соответственно возвращает наибольшее из двух значений `a` и `b`:

```
minvalue = a >? b;
```

- Эти операторы являются примитивами языка и могут применяться без каких-либо побочных эффектов. Следующий оператор возвращает наименьшее из значений `x` и `y`, и затем только однажды их инкрементирует:

```
int minvalue = x++ <? y++;
```

- Операторы <? и >? могут быть перегружены (overload) для действий над классами. Это показано на примере, где определяется класс `Iholder` с оператором >?, который возвращает копию объекта с наибольшим значением целочисленного типа. Также класс `Iholder` использует первичный оператор >? как внутренний для сравнения двух целочисленных (int) значений:

```
/* minmax.cpp */
#include <iostream>
class Iholder
{
    friend Iholder operator>?(Iholder&, Iholder&),
protected:
    int value;
public:
    Iholder(int v)
    {
        value = v;
    }
    int getValue(void)
    {
        return(value);
    }
};

Iholder operator>?(Iholder& ih1, Iholder& ih2)
{
    return(Iholder(ih1.getValue() >? ih2.getValue()));
}

int main(int argc, char *argv[])
{
    Iholder ih1 = Iholder(44);
    Iholder ih2 = Iholder(34);
    Iholder imax = ih1 >? ih2;
    std::cout << "The maximum is " << imax.getValue() << "\n";
    return(0);
}
```

Ограничение указателей (Restrict)

Ключевое слово `restrict` из стандарта ISO C99 сейчас отменено решением комитета стандартов языка C++, но GCC все же поддерживает его использование в виде ключевого слова `__restrict__`. Объявление указателя с ключом `__restrict__` гарантирует исключительный доступ к расположению памяти, на которое он указывает. На деле гарантия отсутствия псевдонимов у указателя позволяет вырабатывать более эффективный код. Ключевое слово `__restrict__` может использоваться как квалификатор (qualifier), подобно `const` или `volatile`, что показано на следующем примере:

```
double *__restrict__ avg;
```

- Ключ `__restrict__` применим только к указателям и ссылкам. В отличие от `const` или `volatile`, этот квалификатор не может применяться к адресуемым данным.
- Аргументы функций указатели также могут квалифицироваться как ограниченные. В следующем примере функции гарантировано, что указатели `bp1` и `bp2` не пересекаются:

```
void copy(char *__restrict__ bp1,
          char *__restrict__ bp2, int size) {
    for (int i=0; i < size; i++)
        bp1[i] = bp2[i];
}
```

- Аргументы функций ссылки могут быть ограничены применением того же синтаксиса, который применяется к указателям, например:

```
void icopy(int &__restrict__ ip1,
           int &__restrict__ ip2) {
    ip1 = ip2;
}
```

- Ключ `__restrict__` игнорируется для различных экземпляров функции, поэтому нет необходимости его применения в прототипах.
- Указатель `this` может быть ограничен использованием `__restrict__` в его объявлении как части определения функции, например:

```
void T::fnctn() __restrict__ { ... }
```

Действия компилятора (Compiler Operation)

Этот раздел описывает некоторые внутренние действия компилятора, о которых вам необходимо знать на случай особых обстоятельств, иногда возникающих при компиляции программ. Обычно для компиляции и компоновки ваших программ на языке C++ ничего более не нужно, кроме использования команды `g++`. Но в некоторых сложных ситуациях вам очень пригодятся знания о кое-каких внутренних подробностях механизма компиляции программ.

Библиотеки (Libraries)

Стандартная библиотека, содержащая набор стандартных подпрограмм языка C++, называется `liststdc++.a. Эта библиотека довольно велика, и, хотя это обычно и не имеет значения, но все-таки следует знать, что при статической компоновке в программу могут включаться очень много избыточных определений, в действительности не используемых программой. Это — следствие того, что когда вашей программе нужна отдельная процедура, являющаяся частью объектного модуля библиотеки, то к программе в качестве ее части статически компонуется весь модуль.`

Когда программе требуется статическая компоновка, и вы не используете стандартные библиотечные подпрограммы, есть возможность компоновать программу с библиотекой `libvsrc++`.a. При этом подключаются только те подпрограммы, которые являются частью основных определений языка. Для того чтобы изменить

стандартную ситуацию, нужно только указать в команде `g++` имя библиотеки `-lsupc++`.

Представление символических имен в объектном коде (Mangling Names)

Как функции на языке *C++*, так и методы языка *Java* могут быть перегружены определением функций с тем же именем, но другим набором параметров. Например, следующие три строки содержат прототипы для разных функций:

```
int *cold(long);
int *cold(struct schold *);
int *cold(long, char *);
```

Компилятор без труда определит, какая из них вызывается, — по разнице в типах аргументов. Единственная проблема возникает со стороны компоновщика (linker). Он слепо по именам внешних ссылок подбирает в модулях соответствующие этим символическим именам определения, не обращая внимания на типы аргументов. Для решения этой проблемы применяется принудительная замена имен компилятором. Такая замена, при которой не теряется информация об аргументах, и компоновщик может находить точные соответствия. Процесс этой замены имен называется в документации "*mangling*".

Замененное имя строится из следующих участков расположения информации и именно в таком порядке:

1. Основное имя функции
2. Пара знаков подчеркивания — "___".
3. Список кодов квалификаторов (qualifiers) функции — таких как `const`. Этот список может иметь нулевую длину.
4. Количество букв в имени класса, членом которого является эта функция.
5. Имя этого класса.
6. Список кодов, показывающих типы аргументов функции.

Например, имя функции `void cname::fname(void)` будет преобразовано в `fname__5cname`. Имя функция `int cname::stname(long frim) const` — в `stname__C5cname1`, где буква "С" означает, что функция имеет квалификатор `const`, и "1" ("эль") в конце показывает, что она может принимать единственный аргумент типа `long`. При кодировании имени процедуры-конструктора пропускается основное имя функции.

Еще пример: конструктор `cname::cname(signed char)` кодируется как `_C5cnameSc`, где "Sc" — пара, которая показывает типы параметров.

Буквенные коды для различных типов и квалификаторов перечислены в таблице 5.3. Расшифровка кодирующих букв в таблице и некоторая практика научат вас свободно преобразовывать применяемые в объектных файлах имена так, чтобы находить их соответствия в исходном коде.

Таблица 5.3. Буквенные коды, применяемые при замене имен

Буква кода	Значение
<code><число></code>	Число букв в имени пользовательского типа данных. Например, имя функции <code>Mph::pdq(char, drip, double)</code> будет закодировано как <code>pdq_3Mphc4dripd</code> . Числу 4 не обязательно должна предшествовать буква G, т.е. <code>pdq_3Mphc4dripd</code> эквивалентно <code>pdq_3MphcG4dripd</code> .
A	Массивы. Массивы в языке C++ всегда раскладываются на указатели, так что этот тип в действительности нигде не встречается. В языке Java массив кодируется как указатель на тип <code>JArray</code> .
B	Тип данных <code>bool</code> C++, или тип данных <code>boolean</code> языка Java.
C	Для языка C++ тип данных <code>char</code> , или для Java тип данных <code>byte</code> .
C	Модификатор, показывающий тип аргумента <code>const</code> или включаемую функцию.
D	Тип данных <code>double</code> .
E	Дополнительные аргументы неизвестных типов, например, имя функции <code>Mph::pdq(int, ...)</code> кодируется <code>pdq_3Mphie</code> .
F	Тип данных <code>float</code> .
G	См. <code><number></code> .
H	Шаблон кода функции.
I	Тип данных <code>int</code> .
I	Особый целочисленный тип данных, содержащий нестандартное количество бит. Например, имя функции <code>Mph::pdq(int, int60_t, char)</code> с 60-битным целочисленным типом в качестве ее второго аргумента будет закодировано как <code>pdq_3MphiI_3C_c</code> . Для указания количества бит в таком целом числе используется шестнадцатиричное число, окруженное символами подчеркивания. Шестнадцатиричное число может не быть окружено символами подчеркивания, если соседние буквы не создают двусмысленной ситуации.
J	Тип данных языка C++ <code>complex</code> .
L ("эль")	Тип данных языка C++ <code>long</code> .
L	Имя локального класса.
P	Указатель. За ним всегда следует индикатор типа указателя. То же, что и "р".
P	Указатель. За ним всегда следует индикатор типа указателя. То же, что и "р".
Q	Квалифицируемое имя, такое, как возникающее из вложенного класса.
R	Тип данных языка C++ <code>long double</code> .
R	Ссылка C++. Всегда сопровождается индикатором типа, на который указывает ссылка. Например, имя функции <code>Mph::pdq(ofstream&)</code> кодируется как <code>pdq_3MphR7ostream</code> .
S	Тип данных <code>short</code> .
S	Предшествует имени класса и означает <code>"static"</code> . Например, <code>Mph::pdq(void) static</code> кодируется <code>pdq_s3Mph</code> . Если "s" предшествует индикатору типа данных <code>char</code> , то имеет значение <code>psigned</code> . Например, имя функции <code>Mph::pdq(signed char)</code> кодируется как <code>pdq_SMphSc</code> .
T	Экземпляр шаблона кода на языке C++.
T	Повторение предыдущего типа аргумента.
U	Квалификатор типа ограниченного указателя.
U	Модификатор, означающий беззнаковый целочисленный тип данных. Он также используется как модификатор для имени класса или имени именованного пространства, указывающий на символьную кодировку Unicode.

Буква кода	Значение
v	Тип данных <code>void</code> .
v	Модификатор, который показывает тип данных <code>volatile</code> .
w	Для языка C++ тип данных <code>wchar_t</code> или для Java тип данных <code>char</code> .
x	Для C++ тип данных <code>long long</code> или для Java тип данных <code>long</code> .
X	Указывает на использование типа шаблона (<code>template</code>) в аргументе.
Y	Указывает на использование в аргументе типа константного шаблона (<code>constant template</code>).

В состав пакета утилит `binutils` входит *деманглер* (от "demangle") — программа `c++filt`. Вы можете ввести замененное имя функции в качестве параметра команды запуска этой программы и получить имя функции таким, каким оно было до замены. Вот пример применения этой программы:

```
$ c++filt pdq__3MphiUsJde
Mph::pdq(int, unsigned short, __complex double, ...)
```

Утилита `c++filt` способна обрабатывать имена, замененные в соответствии с целим рядом применяющихся схем замены. Применяемая схема выбирается параметром опции командной строки `-S`. Доступны варианты выбора платформ (`lucid`, `arm`, `hp` и `edg`) и опции выбора языков (`java`, `gnat`).

Замечание

Схемы замены имен файлов (*tangling schemes*), используемые для C++ и Java, вообще говоря, совместимы между собой, но с другими компиляторами они не дружат. Каждый компилятор использует собственную схему замены имен, что, конечно, оправдано. Также отличаются схемы расположения классов, обеспечивающие выполнимость множественного наследования и применяющиеся в технологии поддержки виртуальных функций. Когда применяются совместимые схемы, то существует возможность компоновки модулей объектного кода GCC с библиотеками, вырабатываемыми другими компиляторами. Но при этом нельзя твердо рассчитывать на то, что программы будут нормально запускаться и правильно работать.

Компоновка программ (Linkage)

В объектном файле находится не только исполняемый код, но и некоторая информация, имеющая значение для ряда действий оптимизации и для нахождения расположения кода при разрешении ссылок. Некоторая часть этой информации относится к категории "неопределенной компоновки" (*vague linkage*), потому что она представляет собой нечто иное, чем результат обычного процесса простого сопоставления имен с соответствующими им адресами. Здесь приводится неполное описание элементов "неопределенной компоновки" реализации языка C++ в компиляторе GNU.

Таблица виртуальных функций (Virtual Function Table)

Это список адресов виртуальных функций класса. Если класс **A** содержит виртуальную функцию и эта функция замещена подклассом **B**, то в этом случае адрес

первоначальной функции заменяется адресом новой функции в таблице виртуальных функций — *vtable*. Это сделано для того, чтобы соблюсти требования полиморфизма. В нашем случае, когда объект класса **B** замещает объект класса **A**, тогда вызов виртуальной функции класса **A** использует эту таблицу и в соответствии с ней осуществляется вызов функции из класса **B**, а не **A**.

Идентификация типа объекта во время выполнения программы (Runtime Type Identification)

В языке C++ каждый объект содержит идентификационную информацию для обеспечения действий таких операторов как `dynamic_cast` и `typeid`, и для обработки исключений. Для классов, содержащих виртуальные функции, вместе с таблицей *vtable* включается блок информации, по которому `dynamic_cast` во время выполнения программы может определить тип интересующего объекта. При отсутствии таблицы *vtable* (это означает что класс неполиморфичен), этот блок информации помещается только в ту область объектного кода, где используется этот класс.

COMDAT

При использовании объявления `comdat` в заголовочном файле можно добиться того, что при каждой компиляции копия вырабатываемого кода будет включаться во все модули, использующие этот заголовочный файл. Это касается таких вещей, как глобальные переменные и функций, тело которых объявляется как часть определения класса. На поддерживающих COMDAT системах (компоновщик GNU на ELF-системах, таких как Linux, Solaris, Microsoft Windows и др.) компоновщик самостоятельно распознает и удаляет все лишние копии такого кода, кроме единственной, которая должна быть помещена в результирующей выполнимой программе.

В документации по компоновщику (linker) вы можете найти описание этого по ключевым фразам `folding`, `comdat folding`, `identical comdat folding`, `comdat discarding` или даже `transitive comdat elimination`.

Функции, расширяемые подстановкой кода (Inline Functions)

Расширяемая подстановкой кода (`inline`) функция, как правило, объявляется в заголовочном файле (`header file`), подключаемом любым модулем, нуждающимся в вызовах этой функции. Даже когда функция объявляется как `inline`, все же создается ее объектный код на тот случай, когда она не может быть расширена подстановкой, например при обращении к ней по адресу.

Экземпляры компилируемого шаблона (Compiling Template Instantiations)

При помещении определения шаблона (`template`) в заголовочном файле и включении этого файла во множество модулей создается соответствующее множество копий скомпилированного шаблона. Этот подход будет работать, однако в большой программе с большим количеством шаблонов в каждый объектный файл будут помещены скомпилированные копии каждого шаблона. Это может значительно уве-

личить время компиляции и размер объектных файлов. Для решения этой проблемы существует несколько альтернатив:

- Возможно использование директив `#pragma interface` и `#pragma implementation` в исходных файлах (это описано выше в этой главе), это приводит к созданию единственной скомпилированной копии шаблона.
- Способ, подобный использованию двух прагма-директив, состоит в применении при компиляции всего исходника опции `-falt-external-templates`. Она указывает компилятору включать экземпляр скомпилированного шаблона только в тех модулях, которые действительно его используют.
- Компиляция исходного кода с применением опции командной строки `-frepo`. Это приводит к созданию файлов с суффиксом `.gro`, каждый из которых содержит список включений экземпляров шаблона, которые можно найти в соответствующем объектном файле. Затем следует задействовать оболочку компоновщика — утилиту `collect2` для дополнения файлов `.gro` инструкциями компоновщику относительно расположения шаблонов в результатеющей программе. Единственная трудность применения этого подхода касается библиотек. Трудность состоит в том, что до тех пор, пока не будут удалены соответствующие файлы `.gro`, компоновка находящихся в библиотеке шаблонов происходит не будет.
- Использование при компиляции кода опции `-fno-implicit-templates`, которая отменяет подразумеваемое применение шаблона и выполняет только указанную вами подстановку. Этот подход предусматривает, что вы точно знаете, какие экземпляры шаблона следует использовать. Этот подход приводит к выработке наиболее точного и чистого кода.

Глава 6



Компиляция программ на языке Objective-C

Алгоритмический язык *Objective-C* вполне можно считать расширением языка программирования *C*. В *Objective-C* добавлена поддержка классов, иногда его называют "языком *C* с объектами". Если применить другую распространенную точку зрения, *Objective-C* — результат объединения языков *C* и *Smalltalk*. *Objective-C* намного проще, чем язык *C++*. Реализация *Objective-C* в GCC представляет собой ничего более чем язык *C* с добавлением в него синтаксиса определения и применения классов, создания экземпляров объектов и передачи объектам сообщений (т.е. вызова методов). При передаче объектам сообщений используются правила, весьма близкие к применяемым в языке *Smalltalk*.

В отличие от других компилируемых в GCC языков *Objective-C* не имеет стандартного определения. Реализация в GCC языка *Objective-C* довольно сходна с версией этого языка, разработанной для системы NeXTStep.

Базовая компиляция

В таблице 6.1 приводится список типовых суффиксов имен файлов, применяемых при компиляции и компоновке (linking) программ на языке *Objective-C*. Полный список распознаваемых GCC суффиксов приведен в приложении Г.

Таблица 6.1. Суффиксы имен файлов, связанных с компиляцией *Objective-C*

Суффикс	Содержимое файла
.a	Библиотека (архивный файл), содержащий объектные файлы для статической компоновки.
.h	Включаемый в программы заголовочный файл (header file).
.m	Исходный файл на языке <i>Objective-C</i> , подлежащий обработке препроцессором.
.mi	Исходный файл на языке <i>Objective-C</i> , не подлежащий предобработке.

Суффикс	Содержимое файла
.o	Объектный файл в поддерживаемом компоновщиком формате.
.so	Разделяемая (динамическая) библиотека, которая содержит объектные файлы, пригодные для динамической компоновки (<i>dynamic linking</i>) во время выполнения программы.

Компиляция отдельного исходного файла в готовую к запуску программу

Программа на языке *Objective-C* может быть написана в соответствии с правилами синтаксиса в формате программы на языке *C*. Это означает, что листинг программы на *Objective-C*, не использующей объектов, мало чем отличается от листинга такой же программы на языке *C*. Далее вам предлагается пример простейшей программы, которая может быть скомпилирована как программа на языке *Objective-C* и запущена на выполнение.

```
/* helloworld.m */
#import <stdio.h>
int main(int argc,char *argv[])
{
    printf("hello, world\n");
    return(0);
}
```

Программа почти во всем соответствует такой же программе на языке *C*, кроме использования директивы препроцессора `#import` вместо директивы `#include`. Обе директивы применяются для одной и той же цели, только `#import` имеет дополнительное удобство. Удобство состоит в том, что заголовочный файл по этой директиве подключается только однажды в пределах отдельного компилируемого модуля. Тот же эффект при использовании `#include` достигается применением директив условной компиляции, как это описано в главе 3. Вы можете выбрать наиболее удобный для вас способ.

Приведенная программа может быть скомпилирована следующей командой:

```
$ gcc -Wno-import helloworld.m -lobjc -o helloworld
```

Опция `-Wno-import` нужна здесь для подавления предупредительных сообщений об использовании `#import` вместо `#include` для включения в программу кода из заголовочных файлов (header files). Имея исходники GCC, вы в состоянии изменить соответствующую установку по умолчанию в файле `cppinit.c`, закомментировав в этом файле следующую строку кода:

```
CPP_OPTION (pfile, warn_import) = 1;
```

Опция `-lobjc` указывает, что при компиляции должна быть использована библиотека `libobjc.a` (основная библиотека языка *Objective-C*). Но именно для нашего примера нет необходимости в ее применении, потому что в коде этой простейшей программы поддержка объектов не используется. Компилятор определяет, что передаваемый ему файл содержит исходный код на языке *Objective-C*, по суффиксу име-

ни файла `.m`. Опция `-o` назначает имя выходному файлу. По умолчанию выходному файлу выполнимого формата было бы присвоено имя `a.out`.

Компиляция программ, использующих объекты

Определения классов обычно содержатся в двух исходных файлах. Язык *Objective-C* предусматривает, что в заголовочном файле с суффиксом `.h` содержится интерфейсное определение (*interface definition*) класса и в другом — исходном файле с суффиксом `.m` — содержится код реализации (*implementation*) методов этого класса. В предлагаемом примере в файле `Speak.h` определяется интерфейс класса `Speak`, способного хранить символьную строку и по запросу выдавать ее на стандартное устройство вывода:

```
/* Speak.h */
#import <objc/Object.h>
@interface Speak : Object
{
    char *string;
}
- setString: (char *) str;
- say;
- free;
@end
```

Директива `#import` применена для включения кода из заголовочного файла `Object.h`, который содержит интерфейсное определение базового класса `Object`. Класс `Object` является верхним (родительским) классом для всех пользовательских классов в программах на языке *Objective-C*. Определение класса `Speak` окружено директивами компилятору `@interface` и `@end`. Внутри этого определения находится выделенный фигурными скобками блок, где находятся определения данных класса `Speak`. В этом примере определяется единственный элемент данных — указатель на символьную строку. За блоком данных следует список методов этого класса. Определение каждого метода имеет следующий синтаксис: символ "минус" ("−"), имя метода и затем — список имен передаваемых ему аргументов (если есть таковые).

Код реализации методов класса `Speak` содержится в файле `Speak.m`:

```
/* Speak.m */
#import "Speak.h"
@implementation Speak

+ new
{
    self = [super new];
    [self setString: ""];
    return self;
}
- setString: (char *)str
{
    string = str;
    return self;
}
- say
```

```
{  
    printf ("%s\n", string);  
    return self;  
}  
- free  
{  
    return [super free];  
}
```

Код заголовочного файла `Speak.h` включается в программу препроцессором, поэтому все определения данных и методов будут доступны программе. Директива компилятору `@implementation` указывает, что этот файл, `Speak.m`, содержит код реализации (`implementation`) методов класса `Speak`, объявленного в файле `Speak.h`. Тело кода реализации того метода, перед именем которого стоит знак "минус", является реализацией соответствующих методов экземпляров объекта. Обращение к ним возможно только тогда, когда объект уже существует. А те методы, код реализации которых стоит после знака "плюс" ("+"), являются методами класса и могут быть вызваны в любом месте программы.

Сигнатура метода в коде его реализации должна соответствовать интерфейсному определению этого метода в заголовочном файле. Она дополняется кодом тела метода, заключенным в фигурные скобки. Несмотря на то, что метод может возвращать определенные в его объявлении типы данных, его результат кроме того включает дополнительный тип `id` (тип данных, представляющий в языке *Objective-C* принадлежность к объекту). Благодаря тому, что методы объекта по большей части возвращают указатель `self`, объект таким способом ссылается сам на себя.

Следующая программа `helloobject.m` использует объект `Speak` для вывода на стандартное устройство вывода строки "hello, world":

```
/* helloobject.m */  
#import <objc/Object.h>  
#import "Speak.h"  
  
main()  
{  
    id speak;  
  
    speak = [Speak new];  
    [speak setString: "hello, world"];  
    [speak say];  
    [speak free];  
}
```

Эта программа может быть скомпилирована последовательной компиляцией каждого из исходных файлов в объектный код и последующей компоновкой полученных объектных файлов. То есть таким набором команд:

```
$ gcc -Wno-import -c helloobject.m -o helloobject.o  
$ gcc -Wno-import -c Speak.m -o Speak.o  
$ gcc helloobject.o Speak.o -lobjc -o helloobject
```

Или же можно применить одну команду, выполняющую все три действия:

```
$ gcc -Wno-import helloobject.m Speak.m -lobjc -o helloobject
```

Создание и использование статической библиотеки

Набор объектных файлов с суффиксом `.o`, вырабатываемых при компиляции исходного кода на языке *Objective-C* может сохраняться в библиотеке (архиве) объектных файлов. В следующем примере создается библиотека с именем `libcat.a`, которая содержит код реализации (implementation code) класса с именем `Cat`. Класс имеет методы, которые принимают последовательность символьных строк и объединяют их в одну строку.

Файл `Cat.h` является заголовочным файлом на языке *Objective-C*, содержащим интерфейсное определение класса `Cat`:

```
/* Cat.h */
#import <objc/Object.h>
@interface Cat : Object
{
    char *string;
}
- add: (char *) str;
- (char *) get;
- init;
- free;
@end
```

Файл `Cat.m` содержит код реализации (implementation) класса `Cat`. Метод `add` используется для добавления символьных строк в конец строки и метод `get` — для получения текущей объединенной строки. Метод-конструктор `init` вызывается для создания нового объекта — экземпляра класса `Cat`:

```
/* Cat.m */
#import "Cat.h"
@implementation Cat
+ new
{
    self = [super new];
    [self init];
    return self;
}
- init
{
    string = NULL;
    return self;
}
- add: (char *)str
{
    int length;
    char *newstring;
    if(string == NULL) {
        length = strlen(str) + 1;
        string = (char *)malloc(length);
        strcpy(string,str);
    }
    else {
        length = strlen(string) + strlen(str) + 1;
        newstring = (char *)malloc(length);
        strcpy(newstring,string);
        strcat(newstring,str);
        free(string);
        string = newstring;
    }
}
```

```
    } else {
        length = strlen(str) + strlen(string) + 1;
        newstring = (char *)malloc(length);
        strcpy(newstring, string);
        strcat(newstring, str);
        free(string);
        string = newstring;
    }
    return self;
}
- (char *) get
{
    return string;
}
- free
{
    if(string != NULL)
        free(string);
    return [super free];
}
```

Исходный файл `Cat.m` компилируется в объектный `Cat.o` следующей командой:

```
$ gcc -c -Wno-import Cat.m -o Cat.o
```

Затем объектный файл используется для построения библиотеки по следующей команде:

```
$ ar -r libcat.a Cat.o
```

По опции `-r` утилита `ar` выполняет замену указанных объектных файлов в существующей библиотеке их новыми версиями. В случае отсутствия библиотеки, указанной первым параметром опции `-r`, создается новая библиотека с таким именем.

Далее следует пример программы. Программа `docat.m` использует класс `Cat` для объединения двух строк и извлечения результирующей строки, и затем выводит эту строку на стандартное устройство вывода.

```
/* docat.m */
#import <objc/Object.h>
#import "Cat.h"

main()
{
    id cat;
    char *line;

    cat = [Cat new];
    [cat add: "Part one"];
    [cat add: " and part two"];
    line = [cat get];
    printf("%s\n", line);
}
```

Эта программа компилируется в исполняемый файл `docat` следующей командой:

```
$ gcc -Wno-import docat.m libcat.a -lobjc -o docat
```

Создание разделяемой библиотеки

Вырабатываемые при компиляции программы на языке *Objective-C* объектные файлы могут сохраняться в разделяемой (динамической) библиотеке. Для включения в состав динамической библиотеки объектные файлы должны компилироваться таким образом, чтобы была обеспечена возможность их выполнения при загрузке в любую область памяти. Это возможно только при применении в объектных модулях относительной внутренней адресации. Для этого при компиляции необходимо указывать опцию командной строки **-fpic** (*pic* — "position independent code"). Такой объектный файл из класса, код которого находится в файле **Cat.m**, будет создан по следующей команде:

```
$ gcc -fpic -Wno-import -c Cat.m -o Cat.o
```

Далее, этот объектный файл используется для создания разделяемой библиотеки:

```
$ gcc -shared Cat.o -o cat.so
```

Две эти команды можно объединить в одну, по которой разделяемая библиотека будет создана непосредственно из исходного файла:

```
$ gcc -Wno-import -fpic -shared Cat.m -o cat.so
```

Вот пример программы, которая использует экземпляр класса **Cat** для объединения трех строк текста в одну и затем выводит результирующую строку:

```
/* showcat.m */
#import <objc/Object.h>
#import "Cat.h"
main()
{
    id cat;
    char *line;

    cat = [Cat new];
    [cat add: "The beginning"];
    [cat add: ", the middle"];
    [cat add: ", and the end."];
    line = [cat get];
    printf("%s\n", line);
}
```

Следующая команда скомпилирует программу **showcat.m** и скомпонует ее так, что во время выполнения она будет использовать разделяемую библиотеку **cat.so**:

```
$ gcc -Wno-import Showcat.m cat.so -lobjc -o showcat
```

Для успешного выполнения полученной при этом выполнимой программы **showcat**, использующей библиотеку **cat.so**, необходимо поместить библиотеку в такое расположение, где она может быть найдена во время выполнения программы. Это подробно описано в главе 12.

Общие замечания, касающиеся языка Objective-C

Язык *Objective-C* не имеет стандарта, ограничивающего его содержание. Когда вы пишете код на *Objective-C*, вы не должны ожидать его однозначной переносимости на другой компилятор. Описываемые в этом разделе свойства своеобразны для GCC-версии языка *Objective-C* и могут не соответствовать правилам, действующим в других реализациях языка. Наш компилятор *Objective-C* построен на полном и вполне стандартном компиляторе языка *C*. Поэтому вы можете применять все доступные возможности препроцессора и компилятора *C*.

Предопределенные типы

В таблице 6.2 перечисляются типы данных, определяемых в заголовочном файле `Object.h`. Те же типы применяются во многих компиляторах *Objective-C*, но имена типов могут отличаться.

Таблица 6.2. Предопределенные типы языка *Objective-C*

Тип	Описание
<code>BOOL</code>	Булевский тип данных, который может принимать только значение <code>YES</code> или <code>NO</code> . Основной для этого типа данных зависит от платформы, однако <code>NO</code> имеет нулевое значение, в то время как <code>YES</code> — ненулевое. Поэтому условные операторы <i>C</i> будут работать с типом <code>BOOL</code> так, как это от них и ожидается.
<code>id</code>	Указатель на объект <i>Objective-C</i> любого типа.
<code>IMP</code>	Адресная ссылка на метод какого-либо объекта.
<code>nil, Nil</code>	Нулевой указатель на объект <i>Objective-C</i> .
<code>SEL</code>	Ссылка на метод объекта по его имени.
<code>STR</code>	Определение <code>typedef</code> типа, соответствующего <code>char *</code> .

Создание интерфейсного объявления

Возможно использование в командной строке `gcc` опции `-gen-decls` для принудительного обновления интерфейса классов, находящихся в исходном файле. Это может быть удобным в случаях, когда необходимо гарантировать актуальность и соответствие заголовочного файла, содержащего интерфейсное определение, и исходного файла реализации класса (*implementation*). При добавлении нового метода в код реализации класса или изменении последовательности вызовов существующего метода драйвер `gcc` может запускаться с этой опцией для замены существующих объявлений методов в интерфейсе класса.

Например, интерфейсное определение ранее рассмотренного класса `Speak` может быть сгенерировано с помощью следующей команды:

```
$ gcc -Wno-import -gen-decls -c Speak.m
```

Опция `-gen-decls` сама по себе не предотвращает попытки компиляции и компоновки, поэтому применяется опция `-c` для удержания компилятора от компоновки

создаваемого определения класса `Speak.h`. В результате будет выведен файл `w.decl` с таким содержимым:

```
@interface Speak : Object
- free;
- say;
- setString:(char *)str;
+ new;

@end
```

Присвоение символических имен и их представление в объектном коде

В языке *Objective-C* методы классов обозначаются знаком плюс "+" либо минус "-", чтобы указать является ли этот метод методом класса, или методом его экземпляра соответственно. Например, следующее интерфейсное определение класса содержит общие методы класса `new` и `copy` вместе с методами экземпляров класса `reset` и `sort`:

```
@interface TinyList : Object
+ new;
+ copy;
- reset;
- sort;
@end
```

В целях отладки вам может понадобиться распознать их замененные (mangled) имена в объектном коде. Метод класса в объектном коде имеет имя, в начале которого стоит выделенная символами подчеркивания буква "c", затем — имя класса, и после символа подчеркивания — имя метода. Метод экземпляра (instance method) будет иметь тот же формат, только в начале будет стоять выделенная символами подчеркивания буква "i". Четыре метода из предыдущего листинга будут иметь следующие имена:

```
_c_TinyList_new
_c_TinyList_copy
_i_TinyList_reset
_i_TinyList_sort
```

Метод, принимающий более одного аргумента, может иметь более одного имени. Например, следующий пример метода принимает два указателя типа `char`: один с именем `string`, и другой — с именем `desc`. И этот метод имеет два имени: `accept` и `as`.

```
- accept: (char *) string as: (char *) desc;
```

Далее — пример вызова этого метода, принадлежащего классу `Lister` в экземпляре класса с именем `lister`:

```
[lister accept: "Herbert" as: "name"]
```

В объектном коде замененное имя этого метода такого экземпляра будет следующим:

```
_i_Lister_accept_as
```



Глава 7

Компиляция программ на языке Fortran

Алгоритмический язык *Fortran* широко известен своими возможностями выполнения сложных математических расчетов. Именно поэтому он по-прежнему сохраняет свое значение в научном сообществе. В некоторых научных кругах, например, среди физиков, язык *Fortran* занимает главенствующее положение среди других языков программирования.

Компилятор *GNU Fortran* изначально основан на стандарте ANSI Fortran 77, но он не ограничен одним этим стандартом. Он включает в себя многие, хотя и не все, свойства и особенности определенные в стандартах Fortran 90 и Fortran 95. Язык программирования *Fortran* — в большей степени традиция, чем стандарт. Сама документация стандартов предоставляет разработчикам компиляторов большую свободу действий, многое оставляя на их усмотрение. Все компиляторы *Fortran* действуют в основном одинаково, но каждый из них поддерживает свой собственный диалект.

Базовая компиляция

В таблице 7.1 приводится список типовых суффиксов имен файлов, которые так или иначе связаны с компиляцией и компоновкой (linking) программ на языке *Fortran*. Полный список распознаваемых GCC суффиксов приведен в приложении Г.

Таблица 7.1. Суффиксы имен файлов, связанные с языком *Fortran*

Суффиксы	Содержание файла
.a	Статическая объектная библиотека (архив).
.f, .for, .FOR	Исходный код на языке <i>Fortran</i> , не подлежащий предобработке.
.E, .fpp, .FPP	Исходный код на языке <i>Fortran</i> , подлежащий обработке препроцессором.
.o	Объектный файл в формате, пригодном для передачи его компоновщику (linker).

Суффиксы	Содержание файла
.f	Исходный код на языке <i>Fortran</i> , предназначенный для предобработки препроцессором RATFOR.
.so	Разделяемая (динамическая) объектная библиотека.

Преобразование отдельного исходного файла в готовую машинную программу

При написании программы на *традиционном Fortran* используются только заглавные буквы. Первые шесть позиций символов каждой строки зарезервированы для специальных целей. Первый столбец отводится для символа "C", указывающего строку комментария. Столбцы со второго по шестой зарезервированы для меток. Код операторов начинается с седьмой позиции. Вот пример программы на языке *Fortran*, написанной с применением *фиксированного формата* (также называемого *традиционным*):

```
C helloworld.f
C
PROGRAM HELLOWORLD
WRITE(*,10)
10 FORMAT('hello, world')
END PROGRAM HELLOWORLD
```

Компилятор GCC не требует строго использования только заглавных букв, однако соблюдение правил форматирования обязательно, если они прямым образом не отменены указанием соответствующей опции. По следующей команде будет выполнена компиляция программы в готовый к выполнению машинный код:

```
$ g77 helloworld.f -o helloworld
```

Команда `g77` запускает драйвер верхнего уровня `gcc`, который создает среду окружения, необходимую для компиляции программ на языке *Fortran*. Тот же результат может быть получен при использовании следующей команды:

```
$ gcc helloworld.f -lfrtbegin -lg2c -lm -shared-libgcc -o helloworld
```

Библиотека `libfrtbegin.a` (задействуемая опцией `lfrtbegin`) содержит код инициализации и завершения, необходимый для правильного выполнения в операционной среде программы на языке *Fortran*. Библиотека `libg2c.a` содержит основные подпрограммы, необходимые во время выполнения, такие как ввод и вывод. Библиотека `libm.a` является системной математической библиотекой. Опция `shared-libgcc` указывает, что используется разделяемая (`shared`) версия стандартной библиотеки `libgcc`.

GCC также позволяет компилировать *Fortran*-код *свободного формата*. Комментарии отмечаются восклицательным знаком ("!") в начале строки и продолжаются до конца строки. Версия предыдущей программы, написанная в *свободной форме*, может иметь операторы и метки, начинающиеся с произвольной позиции:

```
! helloworldff.f
!
Program Helloworld
write(*,10)
10 format('hello, world')
end Program Helloworld
```

Эта программа может компилироваться так же, как и программа предыдущего примера, только в командную строку следует добавить опцию **-free-form**:

```
$ g77 -free-form helloworldff.f -o helloworldff
```

Из-за принципиальных различий между двумя этими синтаксическими формами следует писать программы либо в свободной форме, либо в фиксированном формате. Трудно написать программу, которая будет компилироваться в любом варианте формата *Fortran*, потому что различия в синтаксисе комментариев и правилах общего расположения весьма существенны.

Преобразование нескольких исходных файлов в исполняемый файл

Команда **g77** способна компилировать и компоновать несколько исходных файлов на языке *Fortran* в отдельную машинную программу. В следующем примере программа, исходный текст которой сохранен в файле с именем **caller.f**, выполняет один вызов математической функции и выводит ее результат:

```
C caller.f
C
PROGRAM CALLER
I = Iaverageof(10,20,83)
WRITE(*,10) 'Average=' , I
10 FORMAT('A,I5')
END PROGRAM CALLER
```

Реализация функции **Iaverage** находится в отдельном исходном файле с именем **called.f**:

```
C called.f
C
INTEGER FUNCTION Iaverageof(i,j,k)
Iaverageof = (i + j + k) / 3
RETURN
END FUNCTION Iaverageof
```

Эти два исходных файла компилируются и компонуются в исполняемый файл **caller** следующей командой:

```
$ g77 caller.f called.f -o caller
```

Того же результата можно достичь за три действия. Сначала создать объектные файлы для каждого исходного, и затем скомпоновать их в готовую к запуску программу:

```
$ g77 -c caller.f -o caller.o
$ g77 -c called.f -o called.o
$ g77 caller.o called.o -o caller
```

Генерирование ассемблерного кода

Опция `-S` указывает компилятору `g77` выработать код на языке ассемблера и на этом остановиться. Для выработки ассемблерного кода из исходного файла `helloworld.f` — ранее использованного в этой главе примера, нужно ввести следующую команду:

```
$ g77 -S helloworld.f
```

Выходной ассемблерный файл будет иметь имя `helloworld.s`. Разновидность ассемблерного языка зависит от текущего или назначенного типа целевой платформы компиляции.

Предобработка

При компиляции Fortran-программы из файла с суффиксом имени `.F`, `.fpp`, или `.FPP`, она будет вначале подвергнута предобработке и только затем скомпилирована. Применяемый при этом препроцессор (preprocessor), описанный в главе 3, первоначально предназначался для работы с программами на языке `C`. В следующем примере программа, написанная в *свободной форме Fortran*, использует препроцессор для включения кода функции в главную программу.

```
! evenup.F
!
#define ROUNDUP
#include "iruefunc.h"
!
program evenup
do 300 i=11,22
    j = irue(i)
    write(*,10) i,j
300 continue
10 format(I5,I5)
end program evenup
```

Исходный код функции `irue()` находится в файле с именем `iruefunc.h` и будет компилироваться по-разному, в зависимости от того, определен ли макрос `ROUNDUP`. Эта функция округляет любое нечетное число до четного. По умолчанию функция округляет к ближайшему четному числу в сторону уменьшения, но если макрос с именем `ROUNDUP` определен, то функция будет увеличивать передаваемое в аргументе число до ближайшего четного. Ниже приводится код тела функции `irue()`:

```
integer function irue(i)
k = i / 2
k = k * 2
if (i .EQ. k) then
    irue = i
```

```
else
#define ROUNDUP
    irue = i + 1
#else
    irue = i - 1
#endif
end if
end function irue
```

Следующая командная строка скомпилирует эту программу в исполняемый файл evenup:

```
$ g77 -ffree-form evenup.F -o.evenup
```

Нет необходимости в программах свободной формы специально применять такое форматирование, которое учитывает использование препроцессора. Поскольку препроцессор убирает директивы и передает на компиляцию только результирующий код, следующая программа также имеет вполне приемлемый код:

```
C adder.F
C
#define SEVEN 7
#define NINE 9
C
program adder
    isum = SEVEN + NINE
    write(*,10) isum
10 format(I5)
end program adder
```

Создание и использование статической библиотеки

Статическая библиотека объектных модулей создается компиляцией исходного кода на языке *Fortran* в файлы .o и последующим применением утилиты *ar* для сохранения этих файлов в архиве. *Архив* — это другое название статической библиотеки.

В следующем примере демонстрируется построение объектной библиотеки. В ее состав входят две отдельные функции, вызываемые одной основной программой. Первая функция *imaximum()* находится в файле *imaximum.f*, она возвращает наибольшее из трех передаваемых ей целых чисел:

```
C imaximum.f
C
INTEGER FUNCTION imaximum(i,j,k)
iret = i
IF (j .gt. iret) iret = j
IF (j .gt. iret) iret = k
imaximum = iret
RETURN
END FUNCTION imaximum
```

Вторая функция `iminimum()` находится в файле `iminimum.f`, она довольно похожа на первую, но только возвращает наименьшее из передаваемых ей трех целых чисел:

```
C iminimum.f
C
    INTEGER FUNCTION iminimum(i,j,k)
    iret = i
    IF (j .lt. iret) iret = j
    IF (j .lt. iret) iret = k
    iminimum = iret
    RETURN
END FUNCTION iminimum
```

Приведенные ниже три команды компилируют эти две функции и сохраняют полученные из них объектные модули в библиотеке с именем `libmima.a`:

```
$ g77 -c iminimum.f -o iminimum.o
$ g77 -c imaximum.f -o imaximum.o
$ ar -r libmima.a imaximum.o iminimum.o
```

Опция `-c` команды `g77` указывает компилятору выполнить компиляцию исходного файла в объектный, не задействуя затем компоновщик. Использование утилиты `ar` с опцией `-r` создаст новую библиотеку с именем `libmima.a`, если такой библиотеки еще не существует. Если библиотека уже есть, то входящие в ее состав объектные файлы будут заменены новыми версиями файлов, указанных в командной строке.

Далее приведенная программа вызывает две функции, сохраненные в библиотеке, и выводит их результат:

```
C minmax.f
C
    PROGRAM MINMAX
    WRITE(*,10) 'Maximum= ', imaximum(10,20,30)
    WRITE(*,10) 'Minimum= ', iminimum(10,20,30)
10 FORMAT(A,I5)
    END PROGRAM MINMAX
```

Программа может быть откомпилирована и скомпонована статически с функциями хранящимися в библиотеке `libmima.a` следующей командой:

```
$ g77 minmax.f libmima.a -o minmax
```

Компилятор распознает `minmax.f` как исходный файл на языке *Fortran*. Этот файл компилируется в объектный код. Затем компилятор компонует программу, сообщая компоновщику имя библиотеки `libmima.a`.

Создание разделяемой библиотеки

Создание разделяемой библиотеки очень похоже на созданию статической. Только объектные файлы, сохраняемые в разделяемой библиотеке, должны компилироваться с опцией `-fpic` или `-FPIc`. Необходимо, чтобы код объектных модулей мог ди-

намически загружаться в память во время выполнения программы (PIC – сокращение от "position independent code" — "независимый от положения код").

Используя тот же исходный код, что в предыдущем примере применялся при создании статической библиотеки, два объектных файла с относительной адресацией и разделяемая библиотека из них создаются с помощью следующей последовательности команд:

```
$ g77 -c -fpic iminimum.f -o iminimum.o  
$ g77 -c -fpic imaximum.f -o imaximum.o  
$ g77 -shared iminimum.o imaximum.o -o libmima.so
```

Опция **-c** указывает компилятору вырабатывать объектные файлы без задействования компоновщика. Опция **-fpic** необходима для выработки объектных файлов в формате, пригодном для их динамической загрузки из разделяемой библиотеки во время выполнения программы. Опция **-shared** компонует все объектные файлы, перечисленные в командной строке, в разделяемую библиотеку **libmima.so**. Для того чтобы библиотека была доступна для использования приложением, необходимо поместить ее в соответствующее расположение. Этот вопрос подробно рассматривается в главе 12.

Для компиляции и компоновки программы, использующей разделяемую библиотеку, нужно только включить имя этой библиотеки в команду компилятору:

```
$ g77 minmax.f -lmima -o minmax
```

Опция **-l** определяет, что указанное за ней имя библиотеки **mima**, компилятор должен дополнить до **libmima.so** и искать библиотеку с таким именем в каталогах, перечисленных в списке путей для поиска разделяемых библиотек. Список путей возможного расположения разделяемых библиотек указан в системной конфигурации.

Ratfor

Ratfor – сокращение от "Rational Fortran". Это общедоступный метод предобратки, который позволяет при написании *Fortran*-программ использовать синтаксис, подобный языку *C*, и затем преобразовывать такой исходный код в *Fortran*-код стандартного формата для дальнейшей его компиляции.

Оригинальный транслятор *Ratfor* был создан Керниганом (Kernighan) и Плаугером (Plauger) в 1976 году. После того, как он был принят компанией AT&T, появилось множество версий *Ratfor*. Две последние версии можно бесплатно загрузить из ряда источников в Интернете, среди них: <http://sepwww.stanford.edu/software/ratfor.html> для версии Ratfor77 и <http://sepwww.stanford.edu/software/ratfor90.html> для версии Ratfor90. Загружаемые файлы очень невелики и программы из них устанавливаются довольно просто. Предоставляемая в комплекте с ними процедура инсталляции устанавливает по вашему выбору либо **ratfor77** (готовый к компиляции исходный код на языке *C*), либо **ratfor90** (сценарий на языке *Perl*). Любая из этих программ может вырабатывать *Fortran*-код, пригодный для ввода в компилятор *GCC*.

Для дальнейшей работы пользователь должен выбрать между предлагаемыми версиями *Ratfor*, поскольку они существенно между собой отличаются. Ratfor90 не яв-

ляется прямым расширением Ratfor77. Так что не большого ума дело написать простую программу, которая будет правильно компилироваться в одном из этих стандартов, и не будет в другом.

Входным языком для транслятора *Ratfor* является *Fortran*. Обрабатываемая программа может быть написана на чистом языке *Fortran*, однако в ней допускается применение многих конструкторов, доступных в языке *C*. Следующий пример демонстрирует формат и расположение кода допустимой исходной программы для транслятора *Ratfor*.

```
# ratdemo.r
program ratdemo {
    integer i;
    integer counter;

    counter = 10;
    for(i=0; i<10; i=i+1) {
        counter = counter + 5;
        write(*,10) i, counter;
    }
10 format(I5,I5);
}
end program ratdemo
```

Этот код может быть обработан транслятором *ratfor77* и скомпилирован в готовую для запуска программу следующими двумя командами:

```
$ ratfor77 <ratdemo.r >ratdemo.f
$ g77 ratdemo.f -o ratdemo
```

Файл *ratdemo.f* выводится транслятором *Ratfor*, он является исходным файлом для компилятора *Fortran* и выглядит следующим образом:

```
C output from Public domain Ratfor, version 1.0
program ratdemo
integer i
integer counter
counter = 10
i=0
23000 if(.not.(i.lt.10))goto 23002
       counter = counter + 5
       write(*,10) i, counter
23001 i=i+1
       goto 23000
23002 continue
10 format(i5,i5)
end program ratdemo
```

Особенности и расширения GNU Fortran

Компилятор GCC поддерживает стандарт ANSI Fortran 77 с некоторыми специфичными для GNU расширениями. Он также включает в себя многие, хотя и не все, свойства, определенные в стандарте Fortran 90.

Встроенные функции (Intrinsics)

Компилятор GCC языка *Fortran* содержит сотни *встроенных функций* (*intrinsics*). Полная документация по ним находится на web-сайте GNU и включает в себя описания не только огромного набора встроенных функций, специфичных для GNU Fortran, но также и действующих в GNU Fortran встроенных функций, определяемых другими источниками.

Спецификация языка ANSI Fortran 77 определяет как набор встроенных функций, наследуемых от предыдущего стандарта, так и набор специфических встроенных функций. *Специфической* встроенной функцией называется встроенная функция, возвращающая своеобразный (специфический) тип данных, определяемый специально для нее. Тип, возвращаемый *наследуемой* встроенной функцией может изменяться, в зависимости от способа ее применения, и обычно он определяется типом одного из ее аргументов.

Компилятор GCC Fortran строже некоторых других *Fortran*-компиляторов в требованиях к типам аргументов встроенных функций, поэтому есть вероятность, что g77 откажется компилировать написанную для другого компилятора программу. Например, если переменная **X** объявлена как **INTEGER*8**, то встроенная функция **ABS()** не может принять этот аргумент, потому что она определена для типа **INTEGER*4** и не допускает снижения точности передаваемого ей числа. Необходимо внести исправления в исходный код программы, может быть просто преобразовать передаваемое значение к допустимому типу.

GCC Fortran поддерживает определяемые стандартом MIL-STD 1753 встроенные функции **BTEST**, **IAND**, **IBCLR**, **IBITS**, **IEOR**, **IOR**, **ISHIFT**, **ISBFTC**, **MVBITS** и **NOT**.

Все встроенные функции, которые можно найти в стандартах Fortran 77 и f2c, доступны в g77. Они включают в себя встроенные функции битовых операций **AND**, **LSHIFT**, **OR**, **RSHIFT** и **XOR**. Среди других поддерживаемых встроенных функций — **CDABS**, **CDCOS**, **CDEXP**, **CDLOG**, **CDSIN**, **CDSQRT**, **DCMPLX**, **DCONJG**, **DFLOAT**, **DIMAG**, **DREAL**, **IMAG**, **ZABS**, **ZCOS**, **ZEXP**, **ZLOG**, **ZSIN** и **ZSQRT**.

Всего GCC поддерживает 402 документированные встроенные функции языка *Fortran*.

Формат исходного кода (Source Code Form)

Как уже было показано на ранее рассмотренных в этой главе примерах, GNU Fortran воспринимает как исходный код в формате стандарта ANSI Fortran 77, так и в *свободном формате*. Свободный формат очень схож с форматом стандарта Fortran 90, но при этом GNU Fortran несколько мягче относится к таким вещам, как, например, знаки табуляции.

Ниже приводится список особых для GCC правил, применяемых к исходным текстам программ как свободного, так и фиксированного формата:

- **Символы возврата каретки (Carriage returns).** Любые символы возврата каретки в исходных файлах игнорируются.
- **Символы табуляции (Tabs).** Каждый символ табуляции заменяется нужным количеством пробелов, выравнивая следующий за ней текст к номеру позиции, кратному восьми.

- **Амперсант ("&").** Символ "амперсант" в первом столбце строки исходного текста фиксированного формата указывает, что эта строка является продолжением предыдущей.
- **Короткие строки.** При применении свободного формата длина строки не имеет значения, а при фиксированном формате ее длина составляет 72 символа. Более короткие строки автоматически добавляются справа пробелами до требуемой длины в 72 символа. Это правило действует только для непродолжаемых строк и констант типа `Hollerith`. Это применяемое для фиксированного формата требование к длине строки можно отменить или изменить при использовании опции командной строки `-ffixed-line-length`.
- **Длинные строки.** Стока, длиннее предполагаемой длины, урезается без предупреждения. Это применяется, главным образом, для поддержки следующего кода *Fortran*, где в столбцах с 73 по 80 может содержаться сопутствующая информация (обычно — последовательная нумерация строк). Требование к длине строки исходника фиксированного формата можно отменить или изменить использованием опции командной строки `-ffixed-line-length`.

Комментарии (Comments)

Для создания комментариев можно применять сочетания `/*` и `*/`, но только в случае, когда код предназначается к обработке препроцессором `cpr`. В этом случае препроцессор удаляет блок комментария, выделяемый этими символами, поэтому компилятор его не видит. Форма обозначения комментария сочетанием `//` не используется, это сочетание используется в языке *Fortran* в качестве оператора объединения строк. В языке *Fortran* проекта GNU, как в предобрабатываемых, так и в не подлежащих предобработке файлах, восклицательный знак `("!")` используется для указания того, что в остальной части этой строки находится комментарий. И, конечно, для фиксированного формата буквы `"c"` и `"C"` в первой позиции означают, что строка является строкой комментария.

Знак доллара (Dollar Signs)

Вы можете использовать знак доллара `"$"` в именах с условием, что они не начинаются с этого знака. Если вы применяете такие имена в исходном файле, то при его компиляции следует устанавливать опцию командной строки `-fdollar-ok`.

Заглавные и строчные буквы (Case Sensitivity)

Существует большое число комбинируемых опций компиляции, применяемых для назначения тех или иных правил применения заглавных и строчных букв. По умолчанию ограничения на применение во входном исходном коде заглавных или прописных букв отсутствуют. Буквы в любом регистре воспринимаются одинаково. Установление любой из опций ограничения или уточнения использования регистра букв не действует на комментарии, символьные константы или поля типа `Hollerith`.

В таблице 7.2 перечислены опции, используемые для применения тех или иных требований к регистру букв исходного кода программы. Существуют отдельные установки для ключевых слов, встроенных функций и для имен, определяемых в программе (символических имен). Таблица 7.3 отдельно описывает особенности каждой из четырех разновидностей опций (`any`, `upper`, `lower` и `initcap`), показанных в таблице 7.2.

В таблице 7.4 показаны три установки, определяющие регистр передаваемых из исходного кода ассемблерных инструкций. Необходимо применять эти опции с осторожностью, поскольку внешние ссылки должны в точности соответствовать именам подпрограмм, используемых при компоновке библиотек.

Таблица 7.2. Опции для назначения требований к регистру букв исходного кода

Ключевые слова	Встроенные функции	Символические имена
<code>-fmath-case-any</code>	<code>-fintrin-case-any</code>	<code>-fsymbol-case-any</code>
<code>-fmath-case-upper</code>	<code>-fintrin-case-upper</code>	<code>-fsymbol-case-upper</code>
<code>-fmath-case-lower</code>	<code>-fintrin-case-lower</code>	<code>-fsymbol-case-lower</code>
<code>-fmath-case-initcap</code>	<code>-fintrin-case-initcap</code>	<code>-fsymbol-case-initcap</code>

Таблица 7.3. Четыре возможные требования к регистру букв

Опция	Описание
<code>-...-any</code>	Ограничения на применение того или иного регистра отсутствуют, все комбинации равнозначны. Это значит, что: <code>Function</code> , <code>FUNCTION</code> , <code>function</code> , <code>Function</code> имеют одно значение.
<code>-...-upper</code>	Все буквы должны быть заглавными, т.е. набраны в верхнем регистре клавиатуры.
<code>-...-lower</code>	Все буквы должны быть строчными, т.е. набраны в нижнем регистре клавиатуры.
<code>-...-initcap</code>	Начальная буква должна быть в верхнем регистре, а все остальные — в нижнем. Например: <code>Maximum</code> , <code>Function</code> , <code>Do</code> , <code>Return</code> .

Таблица 7.4. Управление регистром буквенных знаков ассемблерных инструкций, передаваемых из исходного кода

Опция	Описание
<code>-fsource-case-preserve</code>	Выводимый ассемблерный код передается из исходника на входе компилятора без изменений.
<code>-fsource-case-upper</code>	Все ассемблерные инструкции из исходника преобразуются в заглавные буквы.
<code>-fsource-case-lower</code>	Все ассемблерные инструкции выводятся строчными буквами.

Определенные сочетания опций из таблиц 7.2 и 7.4 являются общеупотребительными, вместо них может назначаться одна из опций перечисленных в таблице 7.5.

Таблица 7.5. Отдельные опции, определяющие регистр букв ввода и вывода

Опция	Описание
-fcase-initcap	Эта опция устанавливает требования, чтобы все начиналось с заглавных букв, кроме комментариев и символьных констант. Те же требования устанавливаются определением всех трех опций -initcap из таблицы 7.2 и опции -fsource-case-preserve .
-fcase-lower	Каноническая модель UNIX, где все исходные файлы должны быть набраны строчными буквами. Таким же образом действует применение всех трех опций -lower из таблицы 7.2 и опции -fsource-case-lower .
-fcase-preserve	Эта опция сохраняет регистр букв исходного текста программы, в том числе код на ассемблере. То же, что и определение всех трех опций -any из таблицы 7.2 и опции -fsource-case-preserve .
-fcase-strict-upper	Эта опция применяет требования строгого соответствия стандарту ANSI Fortran 77, где весь исходный код должен быть в верхнем регистре, кроме комментариев и символьных констант. То же, что и определение всех трех опций -upper из таблицы 7.2 и опции -fsource-case-preserve .
-fcase-strict-lower	Эта опция указывает, что все должно быть в нижнем регистре, кроме комментариев и символьных констант. Таким же образом действует применение всех трех опций -lower опций из таблицы 7.2 и опции -fsource-case-preserve .
-fcase-upper	Классическая модель стандарта ANSI Fortran 77, где весь исходный код должен быть в верхнем регистре. То же, что и определение всех трех опций -upper из таблицы 7.2 и опции и -fsource-case-upper .

Особенности Fortran 90

Этот раздел содержит в себе краткое описание некоторых наиболее часто употребимых особенностей стандарта Fortran 90, поддерживаемых g77. Конечно, перечислены они не все, потому что спецификаций этого языка очень много и они довольно сложны. Здесь описаны только те из них, которые поддерживаются g77 без каких-либо дополнительных установок и флагов.

Символьные строки (Character Strings)

Константы типа символьных строк (character strings) могут быть ограничены как двойными кавычками, так и одинарными. Например строка "hello world" имеет то же значение, что и 'hello world'. В строке, записанной с двойными кавычками, отдельный содержащийся в строке символ двойной кавычки обозначается парой двойных кавычек. Например "say ""Hi"" — литерал, содержащий строку say "Hi".

Символьные константы могут иметь нулевую длину (т.е. не содержать символов). Также возможно объявление подстроки, пример такого объявления имеет форму 'hello world' (7:4), оно равнозначно 'worl'.

Имя выделенной конструкции кода (Construct Name)

Имя конструкции может использоваться для выделения исполняемого блока управляемого операторами IF, DO или SELECT CASE. В следующем примере используется имя конструкции cname как определитель начала и конца блока IF:

```
C connname.f
C
PROGRAM connname
key = 12
cname: IF(key .gt. 10) THEN
    key = key - 1
    WRITE(*,10) key
END IF cname
10 FORMAT('Key=',I5)
END PROGRAM connname
```

Операторы CYCLE и EXIT

Оператор **EXIT** используется для немедленного выхода из цикла и для перехода к следующему за этим циклом оператору. То есть выполнение **EXIT** равнозначно применению внутри цикла оператора **GOTO** для перехода на первый оператор, следующий за циклом. (Если вы знакомы с языком *C*, то **exit** в языке *Fortran* имеет то же значение, что и **break** в *C*).

Оператор **CYCLE** используется для пропуска оставшейся части тела цикла и перехода к следующему его повторению (итерации). Поэтому выполнение **CYCLE** в цикле равнозначно применению внутри цикла оператора **GOTO** для перехода на оператор **CONTINUE**, последний оператор цикла. (Если вы знакомы с синтаксисом языка *C*, то **CYCLE** в *Fortran* имеет то же значение, что и **continue** в языке *C*).

В следующем примере показано использование обоих операторов **CYCLE** и **EXIT**:

```
C cycle.f
PROGRAM cycle
DO 10 i=1,3
    IF (i .EQ. 2) CYCLE
    WRITE(*,30) i
10 CONTINUE
DO 20 i=1,3
    IF (i .EQ. 2) EXIT
    WRITE(*,30) i
20 CONTINUE
30 FORMAT('i=',I5)
END PROGRAM cycle
```

Вот что мы получим на выходе этой программы:

```
i=      1
i=      3
i=      1
```

Первый цикл выводит число 1 при первом его повторении, при втором повторении пропускает оператор **WRITE** (при переходе на конец цикла), при третьем повторении выводит число 3. Второй цикл выводит число 1 при первом повторении, на втором повторении происходит выход из цикла.

Оператор DO WHILE

Оператор **DO WHILE** образует цикл, он используется с логическим выражением, блок цикла закрывается оператором **END DO**. Например:

```
C dowhile.f
PROGRAM dowhile
k = 5
DO WHILE ( k .gt. 0 )
    WRITE(*,20) k
    k = k - 1
END DO
20 FORMAT('k=',I5)
END PROGRAM dowhile
```

Бесконечный цикл DO

Если в строке нет ничего, кроме оператора DO, то это означает, что стоящие за ним до END DO операторы будут циклически повторяться либо до принудительной остановки программы, либо до специально предусмотренного выхода. В следующем примере программы цикл повторяется до тех пор, пока значение счетчика не достигнет 8-ми, затем происходит выход из цикла по GOTO:

```
C doforever.f
PROGRAM doforever
k = 0
DO
    WRITE(*,20) k
    if ( k .ge. 8 ) GOTO 100
    k = k + 1
END DO
20 FORMAT('k=',I5)
100 CONTINUE
END PROGRAM doforever
```

Оператор IMPLICIT NONE

Применение оператора IMPLICIT NONE запрещает автоматическое объявление переменных, и требует, чтобы каждая из них объявлялась явно с точным указанием своего типа. Например, следующая программа автоматически определяет тип и объявляет переменную счетчика цикла:

```
PROGRAM imp
do 10 k=1,5
    PRINT *,k
10 CONTINUE
END PROGRAM imp
```

Добавление в начало программы оператора IMPLICIT NONE требует, чтобы все переменные, которые используются после него, были явным образом объявлены до их использования, включая также и счетчик цикла:

```
PROGRAM imp
IMPLICIT NONE
INTEGER k
do 10 k=1,5
    PRINT *,k
```

```
10  CONTINUE  
END PROGRAM imp
```

Директива INCLUDE

Директива **INCLUDE** в соответствии с определением стандарта имеет следующий синтаксис:

```
INCLUDE filename
```

Точный смысл *filename* зависит от конкретной реализации. Компилятор GNU распознает *filename*, как имя файла из текущего расположения или любого другого пути, указанного параметром опции командной строки **-I**. То есть директива **INCLUDE** работает так же, как и директива препроцессора CPP **#include**, с той лишь разницей, что директива **INCLUDE** выполняется компилятором и потому ее применение не требует использования препроцессора.

Целочисленные константы (Integer Constants)

Значения целочисленных констант выражаются в системах счисления на основании 2, 8, 10 и 16. В следующем примере показано, как записывается одно и то же значение в каждой из перечисленных систем счисления. Двоичная система обозначается буквой "B", восьмеричная — "O", шестнадцатиричная — "X" или "Z". Цифры шестнадцатиричной системы счисления могут записываться как с использованием заглавных, так и строчных букв.

```
C bases.f  
C  
PROGRAM bases  
M = 18987  
PRINT *,M  
M = X'4A2B'  
PRINT *,M  
M = Z'4A2B'  
PRINT *,M  
M = O'45053'  
PRINT *,M  
M = B'0100101000101011'  
PRINT *,M  
END PROGRAM bases
```

Операторы сравнения

В таблице 7.6 показаны последовательности символов, которые используются для обозначения традиционных операций сравнения величин.

Таблица 7.6 Альтернативные символы для операторов сравнения

Оригинальный оператор	Альтернативный оператор	Значение
.GT.	>	Больше
.LT.	<	Меньше
.GE.	>=	Больше или равно

Оригинальный оператор	Альтернативный оператор	Значение
.LE.	<=	Меньше или равно
.NE.	/=	Не равно
.EQ.	==	Равно

Разновидности основных типов данных. Селектор DATA

Предусмотрено применение специальных отметок-селекторов **DATA** для внесения изменений в основные типы переменных. Например, синтаксис для определения разновидности **KIND** 3 типа **INTEGER** выглядит так:

```
INTEGER(KIND=3)
```

Возможными значениями для **KIND** являются 0, 1, 3, 5, 7. Этот синтаксис пригоден для всех родительских типов (**INTEGER**, **REAL**, **COMPLEX**, **LOGICAL**, **CHARACTER**), хотя не для всех типов задействованы все значения. В таблице 7.7 отдельно описываются все значения селектора **KIND** и их применение в **GCC** для каждого из типов данных. Точное значение **KIND** меняется от одной платформы к другой, потому что разрядность и диапазон значений основных типов определяются возможностями оборудования.

Таблица 7.7. Числа, определяемые для селектора **KIND**

Вид значения	Описание
0	Значение сейчас не задействовано, но зарезервировано для использования в будущем.
1	Установка по умолчанию. Результат будет тем же, как и без применения KIND . Обычно соответствует INTEGER*4 , REAL*4 , COMPLEX*8 , LOGICAL*4 .
2	Эти типы занимают в два раза больше памяти, чем применяемые по умолчанию. В GNU , изменение этого KIND соответствует стандарту Fortran 90 для двойной точности. Поэтому REAL(KIND=2) соответствует DOUBLE PRECISION , который обычно соответствует REAL*8 . Также COMPLEX(KIND=2) соответствует DOUBLE COMPLEX , который обычно соответствует COMPLEX*16 . INTEGER(KIND=2) и LOGICAL(KIND=2) поддерживаются не каждой реализацией GNU Fortran .
3	Эти типы занимают столько же пространства, как тип CHARACTER(KIND=1) . Это обычно соответствует INTEGER*1 и LOGICAL*1 . Это значение KIND может быть предусмотрено не для всех типов в некоторых реализациях GNU .
5	Эти типы занимают вдвое меньшее пространство, чем предусмотрено по умолчанию (или при применении KIND=1). Это обычно соответствует INTEGER*2 и LOGICAL*2 . Это значение KIND может быть предусмотрено не для всех типов в некоторых реализациях GNU .
7	Действует только для INTEGER(KIND=7) , этот тип имеет размер наименьшего возможного указателя, который может содержать адрес переменной единственной CHARACTER*1 . На 32-битных системах это соответствует INTEGER*4 , а на 64-битных — INTEGER*8 .



Глава 8

Компиляция программ на языке Java

Несмотря на отсутствие стандарта языка *Java* в таком виде, как опубликованные официальными органами стандартизации документы для языков *C*, *C++* и *Ada*, все же существует вполне ясное и простое определение этого языка. Язык *Java* находится под полным контролем корпорации Sun Microsystems, и эта организация несет всю ответственность его за поддержку и развитие. Синтаксис и основные структуры самого языка очень мало изменились со времени выпуска. А вот API (системные классы) регулярно пополняются, и их количество по сравнению с первоначальным возросло уже в несколько раз.

Что касается самого компилятора языка, то компилятор *Java* принципиально отличается от других компиляторов. Он вырабатывает объектный код в двух различных формах. Во-первых, он подобно *C*, *C++*, или любому другому компилятору, может использоваться для создания *системно-ориентированных* (*native*) двоичных объектных файлов, содержащих машинный код, непосредственно исполняемый целевой машиной. Во-вторых, он может вырабатывать объектные файлы в формате *байт-кода Java* (*bytecode*), которые выполняются особым интерпретатором — любой *виртуальной машиной Java* (*Java Virtual Machine, JVM*). Компилятор GNU Java также способен принимать на входе файлы с байт-кодом интерпретатора *JVM* и вырабатывать из них системно-ориентированные объектные файлы, применимые в компоновке машинных программ.

Базовая компиляция

В таблице 8.1 представлен список суффиксов (расширений) имен файлов, имеющих отношение к компиляции и компоновке программ на языке *Java*. Полный перечень распознаваемых GCC суффиксов файловых имен находится в приложении Г.

Таблица 8.1. Расширения файлов, применяемые в программировании на Java

Суффиксы	Содержание файла
.a	Файл архива, библиотека объектных файлов для статической компоновки.
.class	Объектный файл, содержащий байтовый код Java в формате интерпретатора виртуальной машины Java (байт-код <i>JVM</i>).
.java	Исходный код на языке Java.
.o	Двоичный объектный файл в формате, поддерживаемом компоновщиком.
.s	Исходный код на ассемблере.
.so	Разделяемая библиотека для динамической компоновки во время выполнения программ.

Преобразование отдельного исходного файла в машинную программу

Для того, чтобы Java-класс был выполняемым, он должен быть объявлен как `public` и иметь метод с сигнатурой `public main()`, как в следующем примере:

```
/* HelloWorld.java */
public class HelloWorld {
    public static void main(String arg[]) {
        System.out.println("hello, world");
    }
}
```

Для компиляции программы на языке *Java* следует использовать команду `gcj`. Она является драйвером верхнего уровня (front end) для языка *Java*, т.е. интерфейсной программой компилятора `gcc`. В *Java* любой класс, имеющий метод `main()` является выполнимым. Это прекрасно работает с интерпретатором *JVM*, когда вы указываете в командной строке при запуске программы имя класса. Но когда требуется компиляция программы в выполнимый машинной двоичный формат, то нужно указывать начальную входную точку вырабатываемой программы. Следующая команда скомпилирует и скомпонует в машинную программу исходный файл `HelloWorld.java`. Опция `--main` используется как раз для того, чтобы указать метод `main()` класса `HelloWorld` в качестве входной точки программы:

```
$ gcj --main=HelloWorld -Wall HelloWorld.java -o HelloWorld
```

Опция `-o` назначает выполнимому файлу имя `HelloWorld`, который по умолчанию был бы назван `a.out`. Для запуска на выполнение готовой программы нужно просто ввести ее имя в командной строке:

```
$ HelloWorld
```

Исполняемые машиной двоичные файлы свободны от ограничения, налагаемого интерпретатором *Java* на имена выполнимых классов. Поэтому исполняемый файл готовой программы в машинном коде может быть назван как угодно. В следующем примере тот же `HelloWorld.java` компилируется в машинную программу `howdy`:

```
$ gcj --main=HelloWorld -Wall HelloWorld.java -o howdy
```

Но это послабление относится только к исполняемым файлам, содержащим машинный код. При компиляции в байт-код для выполнения в интерпретаторе *JVM* исходный файл public-класса должен иметь то же имя, что и определяемый в нем класс. То есть файл, содержащий определение public-класса `HelloWorld` должен называться `HelloWorld.java`.

Компиляция отдельного исходного файла в байт-код класса виртуальной машины Java

Возможно использовать компилятор GNU для создания файла с суффиксом `.class`, выполнимого в *виртуальной машине Java (JVM)*. Следующая команда `gcj` использует опцию `-C` для создания файла `HelloWorld.class` из исходного файла `HelloWorld.java`:

```
$ gcj -C -Wall HelloWorld.java
```

В сочетании с `-C` опция `-o` недоступна, так что выходной файл `.class` всегда будет иметь то же имя, что и входной файл `.java`. Класс `HelloWorld` содержит необходимый метод `public static void main()`, это значит, что он может быть запущен на выполнение в среде *виртуальной машины Java* проекта GNU `gij` из командной строки:

```
$ gij HelloWorld
```

Вырабатываемый командой `gcj` Java-класс совместим с другими интерпретаторами *Java*. Ту же программу можно запустить в *виртуальной машине Java*, поставляемой корпорацией Sun Microsystems:

```
$ java HelloWorld
```

Двоичный объектный файл из отдельного исходного файла на языке Java

Предлагаемый далее пример команды использует опцию `-c` для подавления компоновки и выработки объектного файла. Такой объектный файл не только можно скомпоновать в исполняемую программу, но также и сохранить в составе статической библиотеки для последующей компоновки.

```
$ gcj -c HelloWorld.java
```

По этой команде будет создан объектный файл `HelloWorld.o`. При необходимости можно назначить другое имя для выходного объектного файла, используя опцию `-o`:

```
$ gcj -c HelloWorld.java -o hello.o
```

Для компоновки объектного файла в машинную программу можно использовать все ту же команду `gcj`. Файл `hello.o` в объектном коде содержит определение класса

`HelloWorld` со статическим методом `main()`. Поэтому имя класса должно быть указано в команде в качестве точки входа в программу:

```
$ gcj --main=HelloWorld hello.o -o hello
```

Необходимость подобного изменения имен объектных файлов возникает крайне редко. Мы построили этот пример так, чтобы нагляднее показать, что в опции `--main` должно указываться имя класса, а не файла.

Преобразование байт-кода интерпретатора Java в машинную программу

Можно использовать `gcj` для компиляции *байт-кода JVM* в самостоятельную выполняемую машинную программу. Указываемый в командной строке файл с суффиксом имени `.class` при компиляции в машинный код воспринимается `gcj` так же, как и исходный файл с расширением `.java`. В следующем примере первая команда компилирует исходный файл в файл Java-класса, а вторая команда компилирует файл класса в самостоятельный исполняемый файл:

```
$ gcj -C HelloWorld.java  
$ gcj HelloWorld.class -o HelloWorld
```

Компиляция нескольких исходных файлов Java в запускаемую программу

Для того, чтобы создать самостоятельный файл готовой к выполнению машинной программы из набора исходных файлов, следует скомпилировать каждый исходный файл и затем скомпоновать полученные объектные модули, указав компоновщику тот из них, который содержит метод `main()`. Следующий простой пример содержит главную процедуру, использующую другой класс для построения строки и еще один класс для вывода этой строки. Вот класс `SayHello`, содержащий главную процедуру:

```
/* SayHello.java */  
public class SayHello {  
    public static void main(String arg[]) {  
        WordCat cat = new WordCat();  
        cat.add("Hello");  
        cat.add("cruel");  
        cat.add("world");  
        Say say = new Say(cat.toString());  
        say.speak();  
    }  
}
```

Метод `add()` класса `WordCat` принимает слово и добавляет его к концу строки, которая содержится во внутренней переменной. Метод `toString()` класса `WordCat` возвращает строку-результат. Эта строка передается методу `speak()` объекта клас-

са **Say**, который и выводит ее на дисплей. Далее приводится класс **WordCat**, который добавляет к внутренней строке по одному слову при каждом вызове его метода **add()**:

```
/* WordCat */
public class Wordcat {
    private String string = "";
    public void add(String newWord) {
        if(string.length() > 0)
            string += " ";
        string += newWord;
    }
    public String toString() {
        return(string);
    }
}
```

Класс **Say** содержит внутреннюю строку буквенных символов и метод **speak()**, используемый для вывода этой строки:

```
/* Say.java */
public class Say {
    private String string;
    Say(String str) {
        string = str;
    }
    public void speak() {
        System.out.println(string);
    }
}
```

Все три класса могут быть скомпилированы в выполнимый файл несколькими путями. Наиболее прямой способ состоит в том, чтобы сделать все одной командой:

```
$ gcj --main=SayHello Say.java SayHello.java WordCat.java -o SayHello
```

По этой команде все три исходных файла будут скомпилированы в объектные, которые затем будут скомпонованы в один двоичный файл с именем **SayHello**. Причем метод **main()** класса **SayHello** будет установлен в качестве точки входа в программу. Того же результата можно достичь выполнением последовательности команд для компиляции каждого исходного файла в объектный и последующей их компоновки с указанием точки входа в программу:

```
$ gcj -c SayHello.java
$ gcj -c Say.java
$ gcj -c WordCat.java
$ gcj --main=SayHello Say.o SayHello.o WordCat.o -o SayHello
```

Также возможно вначале скомпилировать исходный код программы в классы интерпретатора *JVM*, и уж потом скомпилировать те же объектные модули из файлов классов. Затем скомпоновать двоичный файл из полученных объектных модулей с указанием головной процедуры в качестве точки входа в программу.

Компиляция нескольких входных файлов интерпретатора JVM в машинный код

Примеры исходных файлов предыдущего раздела могут быть скомпилированы в три файла классов интерпретатора *Java*:

```
$ gcj -C sayHello.java Say.java WordCat.java
```

Результатом выполнения этой команды будет набор файлов классов, который может быть запущен на выполнение в *виртуальной машине Java*. Вот вариант команды такого запуска нашей программы:

```
$ java SayHello
```

Скомпилировать все исходные файлы на языке *Java* текущего каталога в классы интерпретатора можно и такой командой:

```
$ gcj -C *.java
```

При компиляции и компоновке в двоичный код файлы классов на байт-коде *JVM* могут восприниматься компилятором в качестве исходных. Далее приводятся две команды: первая транслирует исходные файлы в классы интерпретатора, а вторая компилирует эти классы в двоичную исполняемую программу с именем *SayHello*:

```
$ gcj -C SayHello.java Say.java WordCat.java
$ gcj --main=SayHello Say.class WordCat.class SayHello.class -o SayHello
```

Команда **gcj** определяет тип входного файла по суффиксу его имени. Благодаря этому существует возможность смешивать входные файлы различных типов. Программа **gcj** способна компилировать и компоновать программы из комбинации исходных файлов, классов *JVM* и файлов в объектном коде, что демонстрирует следующий набор команд:

```
$ gcj -c SayHello.java -o SayHello.o
$ gcj -C WordCat.java
$ gcj --main=SayHello SayHello.o Say.java WordCat.class -o SayHello
```

Выработка ассемблерного кода

Приведенный далее класс при выполнении создает экземпляр самого себя и, используя метод **speak()**, выдает строки на стандартное устройство вывода:

```
/* Jasm.java */
public class Jasm {
    public static void main(String arg[]) {
        Jasm jsm = new Jasm();
        jsm.speak();
    }
    public void speak() {
        System.out.println("Jasm speaks");
    }
}
```

Этот класс представляет собой вполне законченное приложение. Он может быть скомпилирован в ассемблерный язык соответствующей целевой платформы следующей командой:

```
$ gcj -S Jasm.java
```

На выходе мы получим файл с ассемблерным кодом **Jasm.s**, который можно использовать для создания исполнимого двоичного файла.

Для той же цели в качестве входного может быть использован и файл класса *Java* с байт-кодом интерпретатора *JVM*. Первая из приведенных далее команд создает файл класса **Jasm.class** из исходного **Jasm.java**, вторая — использует этот файл **.class** для генерирования программы на ассемблере **Jasm.s**:

```
$ gcj -C Jasm.java  
$ gcj -S Jasm.class
```

Создание статической библиотеки

Статической библиотекой называется набор объектных файлов (с суффиксом **.o**), сохраняемых внутри одного файла, называемого *статической библиотекой* или *архивом*. Компоновка программы с модулем библиотеки ничем не отличается от ее компоновки с отдельным объектным файлом.

Используя ранее приведенные в этой главе примеры программ на языке *Java*, создадим объектные файлы **WordCat.o** и **Say.o** для последующего их сохранения в статической библиотеке:

```
$ gcj -c WordCat.java Say.java
```

Для построения и обслуживания статических библиотек применяется утилита **ar**. Для создания библиотеки из перечисленных в команде объектных файлов используется опция **-r**. В случае, когда указана уже существующая библиотека, новые модули добавляются в нее, а существующие заменяются при совпадении имен более свежими их версиями. Следующая команда создает библиотеку **libsay.a** из двух объектных файлов:

```
$ ar -r libsay.a WordCat.o Say.o
```

Для использования объектных модулей библиотеки достаточно в командной строке **gcj** указать имя библиотеки. Следующая команда использует библиотеку **libsay.a** при создании готовой к запуску машинной программы **libhello**:

```
$ gcj --main=SayHello SayHello.java libsay.a -o libhello
```

Такое указание в командной строке имени библиотеки предполагает ее наличие в текущем каталоге. Если библиотека находится в расположении, разрешимом для поиска библиотек командой **gcj**, то для указания ее имени можно использовать опцию **-l**, как это показывает следующий пример команды:

```
$ gcj --main=SayHello SayHello.java -lsay -o libhello
```

Подробнее о расположении библиотек читайте в главе 12.

Создание разделяемой (динамической) библиотеки

Разделяемая библиотека — это коллекция объектных файлов, хранимая внутри отдельного файла. Это примерно то же самое, что и статическая библиотека, только она имеет два существенных отличиями. Во-первых, объектные файлы в составе разделяемой (иначе называемой *динамической*) библиотеки подгружаются и компонуются к программе во время ее выполнения. Во-вторых, эти объектные файлы должны быть скомпилированы особым образом — так, чтобы их код мог выполняться без модификации независимо от места его загрузки. Для создания пригодных для размещения в динамической библиотеке объектных файлов их следует компилировать с опцией `-fpic` для выработки *позиционно-независимого кода* ("Position Independent Code"). При этом для разрешения всех внутренних ссылок и вызовов используется относительная адресация, учитывающая возможность многократной выгрузки и загрузки кода во время выполнения программы.

В разделе будет рассматриваться пример, использующий ранее приведенные в этой главе исходные файлы. Следующая команда создает объектные файлы перемещаемого формата:

```
$ gcj -fpic -c WordCat.java Say.java
```

Для компоновки объектных файлов в новую разделяемую библиотеку `libsay.so` используется команда `gcj` с опцией `-shared`:

```
$ gcj -shared WordCat.o Say.o -o libsay.so
```

Теперь исходный файл `SayHello.java` можно скомпилировать в машинную программу `shlibhello`, которая при выполнении использует объектные модули из разделяемой библиотеки `libsay.so`:

```
$ gcj --main=SayHello SayHello.java libsay.so -o shlibhello
```

Содержимое библиотеки не компонуется в исполнимую машинную программу `shlibhello`. В программу только помещаются инструкции, необходимые для загрузки во время выполнения требуемых объектных модулей из разделяемой библиотеки `libsay.so`. Также необходимо, чтобы используемая разделяемая библиотека находилась в соответствующем расположении, разрешимом для ее поиска запущенной на выполнение программой. Подробности о размещении динамических библиотек читайте в главе 12.

Создание Java-архива .jar

Язык программирования *Java* имеет особый род архивов для хранения классов в формате байтового кода интерпретатора. Эти архивы кода известны как *jar-файлы*. Они имеют тот же формат, что и *zip*-архивы, но в них есть особый раздел, называемый *манифестом* (*manifest*). Манифест содержит описание классов, содержащихся в архиве `.jar`. Все внутренние определения в языке *Java* основаны на именах классов. Поэтому для программы на языке *Java* достаточно найти нужный архив `.jar`.

(или несколько архивов) в разрешимом для поиска расположении и просмотреть манифестные разделы на наличие требуемых классов.

Здесь мы продолжаем использовать те же примеры исходного кода, что и в предыдущих разделах. Для создания jar-файла вначале следует скомпилировать исходные файлы в классы на байт-коде интерпретатора *JVM*:

```
$ gcj -C WordCat.java Say.java
```

Для сборки архива .jar применяется утилита *jar* с опцией *c*. Флаг *f* указывает, что следующий за опцией аргумент определяет имя для создаваемого jar-файла. Остальная часть команды составлена из имен файлов классов, предназначенных для помещения в создаваемый архив. Следующая команда создает jar-файл *libsay.jar*, содержащий два класса и манифест:

```
$ jar cf libsay.jar WordCat.class Say.class
```

Классы, сохраненные в jar-файле, не только могут загружаться *виртуальной машиной Java* при выполнении в ней программы. Их также можно компилировать в объектные файлы, которые затем могут использоваться компоновщиком при создании исполняемых программ в двоичном машинном коде. Следующий пример команды создает исполняемый файл *jarlibhello*, при этом происходит компиляция и компоновка классов из jar-файла, который находится в текущем каталоге:

```
$ gcj --main=SayHello libsay.jar SayHello.java -o jarlibhello
```

УТИЛИТЫ КОМПИЛЯТОРА Java

Кроме компилятора *gcj* дистрибутив GCC содержит несколько утилит для обработки исходных и объектных файлов *Java*.

gij

Утилита *gij* является *виртуальной машиной Java* (Java Virtual Machine, JVM). Это – программа-интерпретатор, которая выполняет байтовый код, находящийся в файлах классов *Java*. Командная строка для запуска интерпретатора содержит имя назначаемого к выполнению класса (файла с суффиксом *.class* или класса из jar-файла). Например, следующая программа на языке *Java* выводит на стандартное устройство вывода все передаваемые ей параметры командной строки:

```
/* ListOptions.java */
public class ListOptions {
    public static void main(String arg[]) {
        for(int i=0; i<arg.length; i++) {
            System.out.println(arg[i]);
        }
    }
}
```

Эту программу можно скомпилировать в байтовый код и запустить на выполнение в интерпретаторе двумя командами:

```
$ gcj -C ListOptions.java
$ gj ListOptions
```

Любые аргументы командной строки, следующие сразу после его имени, передаются запускаемому классу. Класс `ListOptions` последовательно выводит на стандартное устройство вывода все переданные ему аргументы. То есть результат выполнения этого класса из командной строки выглядит примерно так:

```
$ gj ListOptions apple butter --help
apple
butter
--help
```

В таблице 8.2 приводится список опций, которые могут использоваться в командной строке `gij`.

Таблица 8.2. Опции, поддерживаемые утилитой `gij`

Опция	Описание
<code>-Dname[=value]</code>	Устанавливает системную переменную <code>name</code> с указанным значением <code>value</code> . Если значение не указано, то переменная с таким именем будет содержать строку нулевой длины.
<code>--help</code>	Выводит этот список опций и на этом завершает программу.
<code>-jar</code>	Имя, указанное в командной строке воспринимается как имя jar-файла, вместо имени файла класса.
<code>-ms=number</code>	Назначает начальный размер (в байтах) выделяемой динамической памяти (иначе говоря, "кучи").
<code>-mx=number</code>	Ограничивает наибольший размер выделяемой динамической памяти ("кучи").
<code>--version</code>	Выводит версию утилиты <code>gij</code> и завершает ее работу.

Опция `jar` позволяет выполнить класс, находящийся в библиотеке `.jar`. При этом назначаемый к выполнению jar-файл должен содержать в своем *манифесте* атрибут `Main-Class`, имеющий значением имя выполнимого главного класса. Указываемый этим атрибутом класс в свою очередь должен содержать `public`-метод `main()`. К примеру, у нас есть библиотека `sayhello.jar`, содержащая класс `SayHello.class` с методом `main()`, и в манифесте этого jar-файла содержится строка:

```
Main-Class: SayHello
```

В этом случае мы можем запустить программу из файла `sayhello.jar`:

```
$ gij -jar sayhello.jar
```

jar

Файл типа `.jar` (сокращение от "Java archive") содержит набор файлов-классов Java в таком формате, в котором они могут непосредственно считываться и выполняться *виртуальной машиной Java*. Для создания, просмотра и модификации таких

файлов и служит утилита `jar`. Опции командной строки, поддерживаемые этой утилитой, приведены в таблице 8.3.

Таблица 8.3 Опции командной строки, поддерживаемые утилитой `jar`

Опция	Описание
<code>-@</code>	Считывает список файлов, указанный со стандартного устройства ввода.
<code>-c</code>	Создает новый файл <code>.jar</code> .
<code>-C dir file</code>	Добавляет в архив указанный файл <code>file</code> из каталога <code>dir</code> .
<code>-E dir</code>	Указывает пропускать файлы из каталога <code>dir</code> .
<code>-f file</code>	Имя <code>file</code> , которое следует за опцией <code>-f</code> , считается именем jar-файла.
<code>--help</code>	Выводит этот список опций и краткую информацию.
<code>-m file</code>	Указывает файл, содержащий информацию для манифеста, включаемого в файл <code>.jar</code> .
<code>-M</code>	Отказ от создания манифеста.
<code>-O</code>	Создает архив <code>.jar</code> без использования сжатия.
<code>-t</code>	Выводит список содержимого архива <code>.jar</code> .
<code>-u</code>	Добавляет файлы в существующий архив <code>.jar</code> .
<code>-v</code>	Выводит на стандартное устройство вывода подробное описание выполняемых действий.
<code>-V</code>	То же, что и <code>--version</code> .
<code>--version</code>	Выводит номер версии утилиты <code>jar</code> .
<code>-x</code>	Извлекает файлы из архива.

Опции командной строки утилиты `jar` весьма сходны с опциями UNIX-архиватора `tar`. Буквы опций могут указываться вместе в начале командной строки без предварительного дефиса. Например, следующая команда создает jar-архив `sayhello.jar`, который содержит все файлы-классы Java из текущего каталога:

```
$ jar cvf sayhello.jar *.class
```

Для создания того же архива Java-классов, но содержащего в качестве *манифеста* информацию из текстового файла `hello.manifest`, используется следующая команда:

```
$ jar cvfm sayhello.jar hello.manifest *.class
```

Имя для jar-файла и имя файла-манифеста должны следовать в том же порядке, что и опции `f` и `m`. Следующая команда отличается от предыдущей только тем, что имена этих файлов поставлены в обратном порядке:

```
$ jar cvmf hello.manifest sayhello.jar *.class
```

При раздельном формате указания опций (с начальными дефисами) будет получен тот же результат:

```
$ jar -c -v -f sayhello.jar -m hello.manifest
```

Следующая команда выведет список содержимого архива `sayhello.jar`:

```
$ jar tvf sayhello.jar
```

Содержимым `.jar` файла может быть просто набор файлов, а может быть и целое дерево каталогов с файлами. *Манифест* архива всегда помещается в файле со стандартным именем `MANIFEST.MF`, который находится в каталоге архива `META-INF`.

`gcjh`

В GNU системно-ориентированные (native) методы для языка *Java* могут разрабатываться с использованием интерфейсов *CNI* ("C++ Native Interface") или *JNI* ("Cygnus Native Interface"). Утилита `gcjh` считывает файлы-классы в байт-коде *Java* и генерирует заголовочные файлы (header files) интерфейса CNI или JNI и заготовки файлов реализации системно-ориентированных методов (*stub-файлы*, англ. "stub" — "обрубок"). Заголовочные файлы стандарта CNI предназначены для их включения в программы на языке *C++*, интерфейс JNI используется программами на *C*. Опция `-stubs` применяется для генерации "болванок" — заготовок исходных файлов, *stub-файлов* на языках *C* и *C++*, используемых для реализаций системных методов с использованием JNI или CNI. Таблица 8.4 представляет список поддерживаемых программой `gcjh` опций командной строки.

Таблица 8.4. Опции командной строки, поддерживаемые утилитой `gcjh`

Опция	Описание
<code>-add text</code>	Вставляет комментарий <code>text</code> в реализацию класса на языке <i>C++</i> . Эта опция игнорируется при указании опции <code>-jni</code> .
<code>-append text</code>	Вставляет комментарий в заголовок определения класса на языке <i>C++</i> . Игнорируется при указании опции <code>-jni</code> .
<code>--bootclasspath=path</code>	Замещает системную переменную <code>CLASSPATH</code> .
<code>--classpath=path</code>	Назначает путь <code>path</code> к каталогу для поиска файлов-классов <i>Java</i> .
<code>--CLASSPATH=path</code>	Назначает путь <code>path</code> к каталогу для поиска файлов-классов <i>Java</i> .
<code>-d directory</code>	Указывает имя каталога <code>directory</code> для выходных файлов.
<code>-friend text</code>	Вставляет указанный <code>text</code> в заголовок класса <i>C++</i> в качестве определения для атрибута объявления <code>friend</code> . Игнорируется при указании опции <code>-jni</code> .
<code>--help</code>	Выводит на стандартное устройство вывода этот список опций.
<code>-Idirectory</code>	Добавляет указанное значение <code>directory</code> к списку <code>CLASSPATH</code> .
<code>-M</code>	Подавляет обычный вывод программы и распечатывает на стандартное устройство вывода все зависимости.
<code>-MD</code>	Выводит все зависимости.
<code>-MM</code>	Подавляет обычный вывод программы и распечатывает на стандартное устройство вывода только те зависимости, которые не назначены стандартными определениями применяемой системы.
<code>-MMD</code>	Направляет на стандартное устройство вывода только те зависимости, которые не назначены стандартными определениями применяемой системы.
<code>-o file</code>	Назначает имя для выходного файла. Если команда вырабатывает несколько файлов, то эта опция вызывает сообщение об ошибке.
<code>-prepend text</code>	Вставляет комментарий <code>text</code> перед объявлением класса <i>C++</i> . Эта опция игнорируется при указании опции <code>-jni</code> .

Опция	Описание
-stubs	По этой опции вместо заголовочных файлов вырабатываются stub -файлы, заготовки для кода реализации. Создаваемые файлы имеют то же имя, что и обрабатываемый Java-класс, но с суффиксом .cc . При указании опции -jni они будут иметь суффикс .c .
-td directory	Указывает имя каталога для хранения временных файлов.
-v	Выводит во время обработки дополнительную описательную информацию. То же, что --verbose .
--verbose	Выводит во время обработки дополнительную описательную информацию. То же, что -v .
--version	По этой опции программа gcjh только выводит номер своей версии и на этом завершает работу.

Входной информацией для **gcjh** является один файл или более в формате классов интерпретатора *JVM*. К примеру, следующая команда считывает файл-класс *Java* с именем **Spangler.class** и создает файл-заголовок **Spangler.h**, пригодный для определения системно-ориентированных методов класса *Java* на языке *C++*:

```
$ gcjh Spangler
```

Другая команда считывает файл **Spangler.class** и вырабатывает начальный исходный файл **Spangler.cc**, содержащий интерфейс с классом *Java*. Этот файл-заготовка далее может редактироваться и использоваться как исходный код на языке *C++*.

```
$ gcjh -stubs Spangler
```

Следующая команда генерирует интерфейсный файл-заголовок **Spangler.h** для дальнейшей реализации системных методов класса **Spangler.class** на языке *C*:

```
$ gcjh -jni Spangler
```

Дальнейшая команда вырабатывает заготовку исходного файла **Spangler.c**, содержащий интерфейс с классом *Java*. Этот файл-заготовка далее может редактироваться и использоваться как исходный код на языке *C*.

```
$ gcjh -jni -stabs Spangler
```

Примеры использования утилиты **gcjh** для комбинирования в одной программе кода на языках *C* и *C++* с кодом на языке *Java* содержатся в главе 10.

jcf-dump

Утилита **jcf-dump** выводит информацию о содержимом файла класса интерпретатора *JVM*. В эту информацию также включается полный список значений используемых констант, список родительских классов, интерфейсов, полей и методов. Таблица 8.5 представляет список опций этой утилиты.

Таблица 8.5. Список опций утилиты `jcf-dump`

Опция	Описание
<code>--bootclasspath=path</code>	Замещает значение системной переменной <code>CLASSPATH</code> .
<code>-c</code>	Транслирует байт-код методов <i>Java</i> на язык <i>C</i> .
<code>--classpath=path</code>	Назначает путь <i>path</i> к каталогу для поиска файлов классов <i>Java</i> .
<code>--CLASSPATH=path</code>	Назначает путь к каталогу для поиска файлов классов <i>Java</i> .
<code>--help</code>	Выводит этот список опций и завершает работу <code>jcf-dump</code> .
<code>-Idirectory</code>	Добавляет указанное имя каталога <i>directory</i> к переменной <code>CLASSPATH</code> .
<code>--javap</code>	Вырабатывает вывод в том же формате, что и утилита <code>javap</code> . Инструмент <code>javap</code> поставляется корпорацией Sun Microsystems в составе стандартного дистрибутива <i>Java</i> .
<code>-o file</code>	Направляет выход вместо стандартного устройства вывода в файл с указанным именем.
<code>-v</code>	Выводит дополнительную описательную информацию. То же, что <code>-verbose</code> .
<code>--verbose</code>	Выводит дополнительную описательную информацию. То же, что опция <code>-v</code> .
<code>--version</code>	По этой опции программа <code>jcf-dump</code> выводит номер своей версии и завершает работу.

Вот пример. Следующая команда выведет дамп внутренней информации класса `SwmpMilin.class` в файл с именем `sm.dump`:

```
$ jcf-dump SwmpMilin.class -o sm.dump
```

jv-scan

Утилита `jv-scan` считывает и анализирует содержимое одного или более исходных файлов на языке *Java* и выводит информацию об исходном коде. Таблица 8.6 содержит список поддерживаемых этой программой опций.

Таблица 8.6. Опции утилиты `jv-scan`

Опция	Описание
<code>--complexity</code>	Выводит показатель сложности (index of cyclomatic complexity) для каждого класса. Он представляет собой число, которое высчитывается особым образом. При этом управляющий поток анализируется для построения графа, далее происходит подсчет узлов, связей и вызовов внешних компонентов.
<code>--encoding=name</code>	Указывает стандарт кодировки символов, который следует применять при считывании исходных файлов. По умолчанию применяется набор UTF-8.
<code>--help</code>	Выводит этот список опций и завершает работу <code>jv-scan</code> .
<code>--list-class</code>	Выводит имена классов, определения которых содержатся в файлах, перечисленных в команде.
<code>--list-filename</code>	В сочетании с опцией <code>--list-class</code> дополнительно выводит для каждого класса имя содержащего его файла.

Опция	Описание
<code>-o file</code>	Направляет выход программы в указанный файл вместо стандартного устройства вывода.
<code>--print-main</code>	Выводит имена классов, имеющих методы, объявленные как <code>public static void main()</code> .
<code>--version</code>	Выводит номер версии программы <code>jv-vasn</code> и завершает ее работу.

jv-convert

Утилита `jv-convert` служит для перекодировки строчных символов из одного стандарта в другой. По умолчанию программа получает информацию со стандартного устройства ввода, но может и обрабатывать файл, если он указан в командной строке первым или как параметр опции `-i`. Выход программы направляется на стандартное устройство вывода, либо в файл, если имя для него указано вторым в командной строке или в параметре опции `-o`. Список опций утилиты `jv-convert` предоставляет таблица 8.7.

Таблица 8.7. Список опций утилиты `jv-convert`

Опция	Описание
<code>--encoding=имя</code>	Имя кодовой таблицы для входных данных. По умолчанию используется кодировка, установленная в системе в качестве основной. То же, что и опция <code>--from</code> .
<code>--from имя</code>	Имя кодировки для входных данных. По умолчанию используется установленная в системе основная кодировка. То же, что и опция <code>--from</code> .
<code>--help</code>	Вводит этот список опций.
<code>-i файл</code>	Имя входного файла.
<code>-o файл</code>	Назначает имя для выходного файла.
<code>--reverse</code>	Обращает установки опций <code>--from</code> и <code>--to</code> .
<code>--to имя</code>	Имя кодовой таблицы для выходных данных. По умолчанию применяется <code>JavaSrc</code> , то есть стандартный набор ASCII с шестнадцатиричным представлением расширенных символов Unicode с помощью escape-последовательности в форме <code>\uxxxx</code> .
<code>--version</code>	Выводит номер версии утилиты <code>jv-convert</code> .

Следующий пример команды преобразовывает содержимое файла `PierNum.uni` из Unicode в 8-битную кодировку и записывает его в файл `PierNum.java` в формате исходного кода на языке `Java`, применяя escape-последовательность "`\u`" для представления расширенных символов Unicode:

```
$ jv-convert --from UTF8 --to JavaSrc PierNum.uni PierNum.java
```

Доступные форматы кодировки перечислены в таблице 8.8.

К сожалению, рассматриваемая утилита не имеет справочной опции для вывода списка поддерживаемых ею кодировок, а он может (и должен) со временем расти. Для того, чтобы точнее узнать о возможностях вашей версии утилиты `jv-convert`, загляните в исходный код. В исходном дереве GCC в каталоге `gcc/libjava-gnu/gcj/convert` находятся файлы с форматом имен `Input_*.c` и `Output_*.c`. После

символа подчеркивания в именах таких файлов стоит имя формата кодировки, поддерживаемой для ввода и вывода соответственно. В процессе преобразования в качестве внутренней промежуточной кодировки используется Unicode. Так что для конвертирования используются соответствующие направлению преобразования пары модулей `Input_*` и `Output_*`. Учтите, что некоторые из сочетаний могут быть доступны не на всех платформах.

Таблица 8.8. Форматы для перекодировки файлов утилитой `jv-convert`

Имя формата	Описание
8859-1	Набор символов ISO-Latin-1 (8851-1).
ASCII	Стандартный набор ASCII.
EUCJIS	Расширенный набор Unicode для представления символов японского языка.
JavaSrc	Стандартный набор ASCII со встроенным в Java шестнадцатиричным представлением расширенных символов Unicode в форме \uxxxx.
SJIS	Набор Shift JIS, применяемый в японской версии операционной системы Microsoft Windows.
UTF8	Форма кодировки символов Unicode, сохраняющая 8-битное включение основного набора символов ASCII.

grepjar

Утилита `grepjar` выполняет поиск строк в файлах `.jar` на соответствие заданному регулярному выражению. Она выводит имена файлов архива `Java` и те строки этих файлов, в которых обнаружены соответствия регулярному выражению. Поиск ведется во всех файлах определенной в команде `jar`-библиотеки, включая и ее манифест. К примеру, такая команда выводит имена всех классов библиотеки `sayhello.jar`, которые имеют метод `main()`:

```
$ grepjar main sayhello.jar
```

Следующая команда выведет из манифеста архива строку, содержащую имя класса, определяемого как `Main-Class`:

```
$ grepjar Main-Class sayhello.jar
```

Опции утилиты `grepjar` приведены в таблице 8.9.

Таблица 8.9. Опции утилиты `grepjar`

Опция	Описание
-b	Выводит смещение найденного соответствия в байтах от начала находящегося в архиве файла.
-c	Выводит количество найденных соответствий вместо подробного вывода каждого включения.
-e	Этой опцией может быть указан шаблон для поиска в том случае, когда его положение в строке может вызвать разночтение.
--help	Распечатывает этот список опций.
-i	Игнорирует при поиске регистр символов (т.е. не различает прописные и строчные буквы).

Опция	Описание
<code>-n</code>	Выводит номер строки в файле для каждого соответствия.
<code>-e</code>	Эта опция подавляет вывод сообщений об ошибках.
<code>--version</code>	Выводит номер версии утилиты <code>grepjava</code> .
<code>-w</code>	Опция назначает условие, при котором требуется соответствие целого отдельного слова заданному шаблону регулярного выражения.

RMI

Набор средств *RMI* (Remote Method Invocation) для вызова удаленных методов предоставляет возможность методу *Java*, выполняемому в одной виртуальной машине, вызывать метод объекта другой виртуальной машины. Две эти виртуальные машины *Java* могут при этом как параллельно выполняться в одной многозадачной системе, так и быть запущенными на разных компьютерах, т.е. в распределенной системе (*distributed system*).

Вся эта кухня выглядит примерно следующим образом. Аргументы вызова последовательно упорядочиваются таким образом, чтобы они могли быть переданы от вызывающего метода к вызываемому. ("serialize"; чаще в описании распределенных систем применяется особый термин "marshalling" — *маршалинг* — т.е. преобразование Маршала.) Через такое же преобразование проходят также и возвращаемые значения.

Имеется центральный реестр, который содержит имя и расположение доступных для вызова, то есть *актуальных* методов. Для объекта, выполняющего вызов нет необходимости знать, вызывает ли он локальный или удаленный метод. Вызывающий метод обращается к вызываемому удаленному методу по его имени и этот вызов принимает особый локальный метод-заглушка, *stub-метод*, передающий запросы. ("stub" можно перевести как "обрубок".)

Далее *stub-метод* находит актуальный метод в центральном реестре виртуальной машины, выполняет маршалинг аргументов и передает их вместе с адресом возврата *скелетному* методу. При использовании транспортного протокола TCP/IP удаленная виртуальная машина может быть найдена где угодно. На удаленной машине *скелетный* метод производит обратное преобразование Маршала над аргументами и вызывает требуемый актуальный метод.

Метод отдает свой результат вызвавшему его *скелетному* методу. Тот в свою очередь подвергает маршалингу этот результат и передает его назад *stub-методу* побес-покоившей его виртуальной машины. И теперь, конечно, уже *stub-метод* выполняет обратное преобразование Маршала результата и передает его первоначальному вызывающему методу.

Машинка,зывающая удаленный метод называется клиентом. А вызываемая ма-шина — сервером.

Следует знать, что выполнение удаленных вызовов связано с некоторыми слож-ными моментами:

- Во-первых, из-за того, что в процессе удаленного вызова объекты могут со-здаваться, разрушаться, проходить прямое и обратное преобразования Мар-

шала, следует как-то управлять распределением хранения данных ("garbage collection" — буквально "сборкой мусора") в распределенной системе. Система задействования удаленных методов RMI использует счетчик, который увеличивается при каждом входящем *отношении* (reference) и уменьшается при его сбросе. Это усложняется тем, что возвращаемые удаленному вызывающему методу объекты могут содержать *отношения* с другими удаленными объектами.

- Клиентская виртуальная машина содержит свой локальный счетчик активных *отношений* к удаленным объектам. Сообщения с *отношениями* посылаются удаленной виртуальной машине. Их счет инкрементируется и декрементируется по мере того, как *отношения* уходят и приходят, и о каждом изменении счета сообщается серверу. Когда этот счетчик обнуляется, то объект может быть "собран в мусор" (garbage collected) сервером.
- Серверная виртуальная машина хранит список всех клиентских виртуальных машин и активных объектов, к которым каждый из клиентов имеет действующие *отношения*. Объект может быть удален только тогда, когда больше нет ни одного связанного с ним удаленного отношения. Частные серверные счетчики *отношений* машины-клиента к объектам сервера также могут быть обнулены при превышении допустимого времени отсутствия обращений.

rmic

Утилита **rmic** — это компилятор *скелетных* (skeleton) методов и *stub-методов* интерфейса RMI. На входе этот компилятор принимает скомпилированный в байт-код файл класса *Java*, реализующий интерфейс `java.rmi.Remote`. Компилятор **rmic** вырабатывает на выходе исходные файлы *stub-метода* `*_Stub.java` и *скелетного* метода `*_Skel.java` и их готовые классы — файлы `.class` с байт-кодом JVM.

Для примера приведем простой класс, реализующий интерфейс удаленного вызова через объявление `implements java.rmi.Remote`.

```
/* HelloRemote.java */
public class HelloRemote implements java.rmi.Remote {
    public void speak() {
        System.out.println("hello from remote");
    }
}
```

Следующая последовательность команд создаст исходные файлы и готовые к выполнению в виртуальной машине Java RMI-методы *stub* и *skeleton*:

```
$ gcj -C HelloRemote.java
$ rmic HelloRemote
```

Первая команда выведет файл `HelloRemote.class`. Вторая команда создаст пару исходных файлов `HelloRemote_Stub.java` и `HelloRemote_Skel.java`, а также соответствующие им файлы-классы `HelloRemote_Stub.class` и `HelloRemote_Skel.class`. Компилятор **rmic** задействует `gcj` для компиляции *stub*- и *скелетного* методов.

В таблице 8.10 приводятся опции командной строки, поддерживаемые утилитой `rmic`. Все опции могут применяться не только в форме с одним предварительным дефисом, как они показаны в таблице, но и с двумя предварительными дефисами. Т.е. команда `rmic -help` равнозначна команде `rmic --help`.

Таблица 8.10. Опции командной строки утилиты `rmic`

Опция	Описание
<code>-classpath path</code>	Путь к системному каталогу для разрешения ссылок на включаемые классы.
<code>-d directory</code>	Имя каталога для выходных файлов.
<code>-depend</code>	Включает проверку зависимостей с перекомпиляцией всех неактуальных файлов, на которые имеются ссылки в компилируемой программе.
<code>-g</code>	Помещает отладочную информацию в вырабатываемые файлы.
<code>-help</code>	Выводит на стандартное устройство вывода этот список опций.
<code>-J flag</code>	Передает указанный флаг компилятору Java для компиляции <code>*_Stub</code> и <code>*_Skel</code> методов.
<code>-keep</code>	Сохраняет временные файлы, удаляемые по умолчанию. То же, что и <code>-keepgenerated</code> .
<code>-keepgenerated</code>	Сохраняет удаляемые по умолчанию временные файлы. То же, что и <code>-keep</code> .
<code>-nocompile</code>	При этой опции вырабатываемые исходные <code>*_Stub.java</code> и <code>*_Skel.java</code> файлы не компилируются далее в файлы-классы <code>_Stub.class</code> и <code>_Skel.class</code> .
<code>-nowarn</code>	Подавляет вывод предупреждений и сообщений об ошибках.
<code>-v1.1</code>	Вырабатывает stub-файлы в формате Java версии 1.1.
<code>-v1.2</code>	Вырабатывает stub-файлы в формате Java версии 1.2.
<code>-vcompar</code>	Вырабатывает stub-файлы, совместимые с обоими версиями языка Java — 1.1 и 1.2.
<code>-verbose</code>	Выводит описания выполняемых действий.
<code>-version</code>	Выводит номер версии компилятора <code>rmic</code> .

rmiregistry

Программа `rmiregistry` — это "демон", который поддерживает список актуальных для удаленного вызова методов в виртуальной машине Java. Он принимает входящие сообщения через TCP/IP порт (по умолчанию — порт с номером 1099). Если для этого применяется другой номер порта, то он может быть указан в командной строке при запуске `rmiregistry`. Остальные доступные опции команды приведены в таблице 8.11, их совсем немного.

Таблица 8.11. Опции командной строки `rmiregistry`

Опция	Описание
<code>--help</code>	Выводит список допустимых опций и завершает работу программы.
<code>--version</code>	Выводит номер версии программы <code>rmiregistry</code> и завершает ее работу.

Свойства (System Properties)

Java имеет набор "свойств системы" (*properties*) — предустановленных переменных, доступных из выполняемой программы. Каждая из таких переменных состоит из имени ключа и его значения, оба поля имеют тип строки символов — **String**. Например, такой вызов метода может быть использован для определения имени пользователя, запустившего программу:

```
String username = System.getProperty("user.name");
```

Следующая программа выводит список всех предустановленных свойств системы:

```
/* AllProps.java */
import java.util.Properties;
public class AllProps{
    public static void main(String arg[]) {
        Properties properties = System.getProperties();
        properties.list(System.out);
    }
}
```

Более 30-ти свойств системы являются стандартно предустановленными. Они включают в себя название операционной системы, версию компилятора Java, версию операционной системы, имя пользователя, символ-разделитель имен каталогов в путях, символ-разделитель строк, и т.п. В дополнение к ним вы можете назначать свои свойства хоть в самой программе, хоть из командной строки.

Следующая программа показывает значения трех стандартных свойств системы с именами **java.vm.version**, **java.vm.vendor** и **java.vm.name**. Она также выводит значение свойства **magic**, если оно определено.

```
/* ShowProps.java */
public class ShowProps {
    public static void main(String arg[]) {
        System.out.println(
            "vm.version=" + System.getProperty("java.vm.version"));
        System.out.println(
            "vm.vendor=" + System.getProperty("java.vm.vendor"));
        System.out.println(
            "vm.name=" + System.getProperty("java.vm.name"));
        String magic = System.getProperty("magic");
        if(magic == null)
            System.out.println("There is no magic");
        else
            System.out.println("magic=" + magic);
    }
}
```

Свойство **magic** можно определить в командной строке опцией **-D** при компиляции программы в двоичный код:

```
$ gcj --main=ShowProps -Dmagic=xyzzy ShowProps.java -o showprops
```

Команда на запуск программы и ее вывод будут выглядеть примерно так:

```
$ showprops  
vm.version=3.2 20020412 (experimental)  
vm.vendor=Free Software Foundation, Inc.  
vm.name=GNU libgcj  
magic=xyzzy
```

Ситуация несколько меняется при компиляции программы в байт-код и ее запуске на выполнение в среде *JVM*. Тогда свойство должно назначаться не при компиляции программы, а при ее запуске. Для компиляции в файл-класс, выполнимый интерпретатором *Java* следует применить следующую команду:

```
$ gcj -C ShowProps.java
```

Для запуска скомпилированного класса и назначения проверяемого из программы свойства *magic* применяется следующая команда:

```
$ gcj -Dmagic=xyzzy ShowProps
```

Эта команда должна вывести на стандартное устройство вывода ту же информацию, что и при запуске двоичной версии программы.

Глава 9



Компиляция программ на языке Ada

Компилятор *GNAT* (сокращение от "GNU NYU Ada95 Translator", или от более простого названия "GNU Ada Translator") является компилятором алгоритмического языка *Ada*, интегрированным в сборный компилятор GNU. На сегодняшний день он входит в состав GCC.

Ada 95 — новейший стандарт языка программирования *Ada*. Этот стандарт поддерживается GCC в полном объеме. Этот стандарт вместе со строгой типизацией, унаследованной от стандарта Ada 83, включает в себя объектно-ориентированное программирование, наследование, полиморфизм и динамическое перенаправление вызовов. Стандарт Ada 95 также включает в себя внутренние интерфейсные определения для прямого взаимодействия с программами на языках *C* и *Fortran*.

И как язык программирования, и как компилятор, *Ada* имеет свои специфические требования. Наиболее заметной особенностью является возможность обратной трассировки и проверки кода на этапе преобразования исходных файлов в объектные. Эти проверки могут выполняться не только как обычная часть процесса компиляции. Для этих целей также существует набор запускаемых из командной строки утилитных программ, выполняющих действия по сравнению и проверке работоспособности кода. В отличие от прочих входящих в GCC языков компилятор *Ada* написан на самом языке *Ada*. Поэтому для его инсталляции в системе требуются некоторые дополнительные действия.

Инсталляция

Верхний уровень (front end) языка *Ada* является новейшим дополнением GCC. Начиная с выпуска GCC 3.1, он интегрирован в GCC достаточно хорошо, чтобы вырабатывать исполняемый код для основных платформ, но пока еще есть затруд-

нения с его переносимостью на некоторые более новые системы. Верхний уровень компилятора *Ada* написан на самом языке *Ada*. Вообще такой подход вполне оправдан и считается наилучшим, в GCC верхний уровень компилятора *C* тоже написан на языке *C*. Но именно это и создает своеобразную ситуацию с переносом компилятора *Ada*, отличающуюся от переноса компиляторов других языков семейства GCC. Можно надеяться, что со временем процедура переноса компилятора языка *Ada* станет столь же простой, как и компиляторов других языков семейства GCC. Но пока остается необходимость в наличии инсталлированного в вашей системе начального (bootstrap) компилятора *Ada* для полной установки этого языка в GCC.

Чтобы установить в вашей системе последнюю редакцию компилятора *Ada*, вам нужно преобразовать его исходные файлы в машинный код. Для этого нужно установить начальный (bootstrap) *Ada*-компилятор, соответствующий вашей системе. Со временем проблема с установкой *Ada* из исходных файлов GCC будет решена, но пока для этого требуется выполнение особой последовательности действий:

1. Загрузите копию готового компилятора *Ada* в машинном коде для использования его в качестве начального компилятора. Есть несколько источников, где можно найти подходящую версию для вашего компьютера:
 - <http://www.gnuada.org>
 - <http://cs.nyu.edu/pub/gnat>
 - <http://www.gnat.com>

Если же у вас есть уже установленный компилятор *Ada*, то нужно только установить переменную окружения **ADAC** значением, равным его имени и удостовериться, что путь к этой программе указан в установках переменной **PATH**.

2. При инсталляции загруженной версии компилятора следуйте инструкциям по ее установке. Подробности установочной процедуры меняются в зависимости от платформы. Инсталляция состоит из двух этапов. Сначала запускается конфигурационный сценарий **doconfig**, который разъясняет инсталляционную процедуру и запрашивает тип и пути инсталляции. Он создает уже настоящий установочный сценарий **doinstall**. Затем запускается сценарий **doinstall**, который и выполняет всю назначеннную установку.
3. Измените глобальную переменную **PATH** так, чтобы из командной строки могла быть выполнена команда к вновь устанавливаемому **gcc**. Если у вас уже установлена версия компилятора **gcc**, то важно, чтобы новый каталог с компилятором *Ada* стоял в списке путей **PATH** раньше, чем путь к установленной версии **gcc**. Теперь у вас есть полностью функциональный компилятор *Ada*, на котором уже можно компилировать программы. Так что вы можете на этом остановиться и приступить к написанию программ на этом языке. Однако если вы желаете построить собственный компилятор *Ada* из исходников GCC, то переходите к следующему этапу установки.
4. Выполните сценарий конфигурации **configure**, как это описывалось в главе 2. Обязательно должна быть при этом определена опцией **--enable-languages** компиляция языков *Ada* и *C*. Даже если вы собираетесь включать остальные языки позже, все равно лучше начать с этих двух. Далее приводится пример

последовательности команд, запускаемых из родительского каталога исходных файлов `gcc` и конфигурирующих установку в каталоге `mybuild`. По значению опции `--prefix` части компилятора будут установлены в каталогах `/usr/gnat/bin`, `/usr/include`, `/usr/info`, `/usr/gnat/lib`, `/usr/man` и `/usr/share`:

```
$ DIR='pwd'
$ mkdir $DIR/mybuild
$ cd $DIR/mybuild
$ $DIR/gcc/configure --prefix=/usr --enable-languages=c,ada
```

Возможно, вы сочтете удобным поместить эти команды в сценарий. Учитите, что следует обязательно подключать компиляцию языка *C*, без этого новую версию компилятора *Ada* получить не удастся.

5. Для гарантии успешной компиляции начальных программ, устанавливающих язык *Ada*, следует изменить дату некоторых файлов из исходного каталога. После выполнения сценария конфигурации следует применить команду `touch`, чтобы атрибуты даты и времени изменения файла показывали, что устанавливаемые файлы "свежее" тех, с которыми они будут сравниваться. Для удобства этот набор команд можно поместить в отдельный сценарий.

```
$ cd $DIR/mybuild/gcc/ada
$ touch treeps.ad
$ touch einfo.h
$ touch sinfo.h
$ touch nmake.adb
$ touch nmake.ads
```

6. Скомпилируйте начальные программы, необходимые для установки компилятора *Ada*:

```
$ cd $DIR/mybuild/gcc
$ make bootstrap
```

7. Возможно, потребуется отдельная компиляция `gnatlib`. Она может и не понадобиться, но лишний раз выполнить команду для компиляции `gnatlib` ни чему не повредит:

```
$ cd $DIR/mybuild/gcc
$ make gnatlib
```

8. Если до сих пор все прошло хорошо, то теперь можно подать команду `make`, завершающую установку компилятора *Ada*. Эта установка вносит изменения в некоторые системные каталоги, поэтому вам могут понадобиться полномочия администратора локальной машины (super user):

```
$ su
Password: *****
$ cd $DIR/mybuild
$ make install
$ exit
```

Если вы просмотрите каталоги установки, то обнаружите, что инсталляция языка *Ada* дублирует некоторые файлы `gcc`. Это нормально. В следующих вы-

пусках эти излишества наверняка будут устраниены, но пока что должно быть так.

9. Восстановите глобальную переменную **PATH**. Сначала уберите временные установки для компиляции начальных программ и других компонентов *Ada*, затем добавьте новый каталог **bin**:

```
$ PATH= $PATH:/usr/gnat/bin
```

Базовая компиляция

В таблице 9.1 перечислены суффиксы имен файлов, с которыми приходится иметь дело при компиляции и компоновке программ на языке *Ada*. Полный список распознаваемых GCC суффиксов файловых имен находится в приложении Г.

Таблица 9.1. Суффиксы имен файлов, задействованных в программировании на языке Ada

Суффикс	Содержание файла
.a	Библиотека объектных модулей для статической компоновки (файл архива).
.adb	Файл с телом реализации исходного кода модуля библиотеки на языке <i>Ada</i> .
.adc	Конфигурационный файл GNAT для предотвращения использования неактуального кода (<i>dead code elimination</i>).
.ads	Спецификационный файл (<i>spec-file</i>) на языке <i>Ada</i> , содержащий объявления библиотечных модулей или объявления изменений их имен.
.adt	Файл структуры дерева используемых исходных файлов. Используется в GNAT для предотвращения использования неактуального устаревшего кода.
.ali	Вырабатываемый компилятором временный файл, содержащий необходимую информацию для проверки целостности пакетов и их компоновки.
.atb	Файл, представляющий компилятору внутреннее дерево содержания файла .adb.
.ats	Файл, представляющий компилятору дерево внутреннего содержания файла .ads.
.o	Объектный файл в формате, поддерживаемом компоновщиком.
.v	Файл в ассемблерном коде. Такие файлы вырабатываются на промежуточном этапе создания объектных файлов.
.wo	Библиотека объектных файлов для динамической компоновки. (Разделяемая библиотека.)

Преобразование отдельного исходного файла в исполняемый код

Для создания из исходного файла на языке *Ada* готовой к запуску программы в машинном коде необходимо выполнить следующие действия:

1. Компиляция файла, содержащего исходный текст программы на языке *Ada*, в объектный файл.
2. Обработка объектного файла (или нескольких) *биндером* компилятора *Ada*.

3. Компоновка объектного файла (или нескольких) с соответствующими библиотеками для создания готовой машинной программы.

Первый и третий этапы в этой последовательности применяются при компиляции программ и на других языках. Второй же выполняется только для компиляции программ на языке *Ada*.

Программа *биндер* (*binder*) создает подшивку объектных файлов. При этом он проверяет объектные файлы и выполняет следующие действия:

- Выполняет проверки содержимого объектных файлов на соблюдение таких условий, как совместимость с установленными опциями и применяемой версией компилятора.
- Проверяет допустимость применяемого порядка построения программы.
- Создает верхнюю процедурную программку, основанную на указанном порядке построения вырабатываемой программы. Это — небольшая программка на языке *C*, вызывающая в необходимом порядке функции для построения всей программы, и затем выполняющая вызов главной подпрограммы.
- Определяет полный набор действуемых в компоновке программы объектных файлов и помещает эти сведения в генерируемую на языке *C* программку, что делает эту информацию доступной для утилиты *gnatlink*, которая используется для компоновки готовой к запуску программы.

Далее следует исходный код простой программы, которая выводит строку текста на дисплей:

```
with Text_IO; use Text_IO;
procedure HelloWorld is
begin
    Put_Line("hello world");
end HelloWorld;
```

Эта программа сохранена в файле *helloworld.adb*, она компилируется в объектный файл следующей командой:

```
$ gcc -c helloworld.adb
```

Опция *-c* указывает, что программа должна быть скомпилирована в объектный файл, но не должна после этого компоноваться в готовый к запуску исполняемый файл. Это опция необходима при компиляции с языка *Ada*, так как процесс компоновки программ на этом языке отличается от компоновки программ на других языках. Далее используется утилита *gnatbind* для создания подшивки:

```
$ gnatbind helloworld.ali
```

Результатом выполнения этой команды будут два временных рабочих файла *b-helloworld.adb* и *b-helloworld.ads*. Файл *helloworld.ali* не изменяется, как и первоначальный исходный файл *helloworld.adb*. Поэтому теперь мы имеем на диске четыре файла.

Окончательным шагом будет запуск утилиты *gnatlink*:

```
$ gnatlink helloworld.ali
```

В результате всех действий мы получаем выполнимый файл `helloworld`. На диске также остаются исходный файл `helloworld.adb`, файл `helloworld.ali` и объектный файл `helloworld.o`.

Программы на языке *Ada* могут компилироваться и компоноваться другим способом. Утилита `gnatmake` использует критерии, сходные с применяемыми утилитой `make`, для определения файлов, участвующих в компиляции. Затем `gnatmake` задействует компилятор и утилиты `gnatbind` и `gnatlink`. Результат будет таким же, как и при использовании трех отдельных команд. После следующей единственной команды мы получим те же четыре файла, которые создавались рассмотренной последовательностью из трех команд:

```
$ gnatmake helloworld.adb
```

Эту команду еще можно упростить. Если суффикс имени файла в командной строке не указан, то утилита `gnatmake` автоматически добавляет суффикс `.adb`. То есть предыдущую команду можно написать так:

```
$ gnatmake helloworld
```

Выработка готовой программы из нескольких исходных файлов

Набор процедур может быть определен как пакет. Файл `howdy.ads` содержит спецификацию пакета с именем `Howdy` как содержащего процедуры `Hello` и `Goodbye`:

```
package Howdy is
    procedure Hello;
    procedure Goodbye;
end Howdy;
```

Тела самих процедур определены в файле `howdy.adb` следующим образом:

```
with Text_IO; use Text_IO;
package body Howdy is
    procedure Hello is
        begin
            Put_Line("Howdy from package");
        end Hello;
    procedure Goodbye is
        begin
            Put_Line("Goodbye from package");
        end Goodbye;
end Howdy;
```

Программа, которая использует процедуры из пакета `Howdy` для вывода текста на дисплей, находится в файле `howdymain.adb`:

```
with Howdy;
procedure HowdyMain is
begin
    Howdy.hello;
    Howdy.goodbye;
end HowdyMain;
```

Утилита **gnatmake** в состоянии распознать такую организацию программы и скомпилировать необходимые исходные файлы в законченную выполнимую программу. Для этого можно применить следующую команду:

```
$ gnatmake howdymain
```

Это равносильно применению последовательности команд:

```
$ gcc -c howdymain.adb
$ gcc -c howdy.adb
$ gnatbind -x howdymain.ali
$ gnatlink howdymain.ali
```

В результате будут получены файлы с именами `howdy.ali`, `howdymain.ali`, `howdy.o`, `howdymain.o` и готовая к запуску программа `howdymain`. При выполнении эта программа будет выдавать следующее:

```
Howdy from package
Goodbye from package
```

Преобразование исходного файла на языке Ada в ассемблерный код

Опция **-S** указывает **gcc** переработать исходный код в набор ассемблерных инструкций и на этом остановиться. По следующей команде будет создан содержащий ассемблерный код файл `helloworld.s` из исходного файла программы на языке *Ada* `helloworld.adb`:

```
$ gcc -S helloworld.adb
```

Содержимое файла ассемблерного кода зависит от назначенному компилятору целевой платформы. При указании в командной строке нескольких исходных файлов для каждого из них будет создан отдельный файл с ассемблерным кодом.

ОПЦИИ КОМПИЛЯЦИИ

Полный набор поддерживаемых компилятором GNU опций командной строки находится в приложении Г. Некоторые из них имеют особое значение для языка программирования *Ada*. В таблице 9.2 перечислены те опции, которые вообще применяются для всех языков, но при компиляции программ на языке *Ada* имеют особое значение.

Таблица 9.2. Общие опции командной строки, имеющие специальное значение для компилятора *Ada*

Опция	Описание
-c	Назначает компиляцию до объектного файла без компоновки в исполняемый код. Эта опция необходима при компиляции <i>Ada</i> , потому что gcc не задействует gnatbind и gnatlink .
-fno-linkline	Подавляет любые подстановки исходного кода независимо от уровня оптимизации.

Опции	Описание
-g	Включает в объектный файл отладочную информацию, добавляемую при компоновке в исполняемый файл, и делает ее доступной для программы-отладчика.
-Idirectory	Добавляет имя каталога к списку путей, в которых должен выполняться поиск необходимых для компиляции исходных файлов.
-I-	Указывает при компиляции назначаемого командой файла не проводить поиск других исходных файлов далее текущего каталога.
-o [n]	уровни оптимизации для программ Ada применяются те же, что и для других языков. Они описаны в приложении Г. В том числе возможно применение значения n равного 3, что включает автоматическое расширение подстановкой кода.
-s	Назначает вывод ассемблерного кода.
-v	Показывает номер текущей версии GCC и список всех команд, вырабатываемых драйвером gcc .
-Vversion	Применяет указанную версию компилятора gcc .
-Wuninitialized	Вырабатывает предупредительное сообщение по каждой неинициализированной переменной.

В дополнение к общим опциям компиляции, перечисленным в таблице 9.2, и большому количеству других опций, приведенных в приложении Г, таблица 9.3 содержит набор опций, которые применяются только для языка Ada. Все эти специфические опции начинаются с последовательности из пяти букв **"-gnat"**.

Таблица 9.3. Особые для языка Ada опции командной строки

Опция	Описание
-gnat83	Указывает, что программу следует компилировать в соответствии со стандартом Ada 83. В основном эта опция применяется для проверки переносимости программного кода на компиляторы стандарта Ada 85. По умолчанию применяется -gnat95 .
-gnat95	Компилирует исходный код в соответствии со стандартом Ada 95. Применяется по умолчанию.
-gnata	Вводит в действие директивы pragma Assert и pragma Debug . Без указания этой опции такие pragma -установки в исходных файлах игнорируются.
-gnatb	Все сообщения об ошибках будут выводиться на стандартное устройство вывода в сокращенной форме и включаться в развернутом виде в листинг.
-gnatc	Компилятор будет выполнять подробные семантические проверки и не будет вырабатывать никаких выходных файлов, кроме содержащих возможные предупреждения и сообщения об ошибках.
-gnatdxx	Эта опция может быть использована для получения информации о процессе компиляции для отладки самого компилятора. Значение жж — одна буква или цифра либо их сочетание, определяет тип извлекаемой отладочной информации. Существует 65 допустимых символов (большие и малые латинские буквы и цифры от 1 до 9). Применяется редко. Описание для этой опции находится в комментариях исходного файла debug.adb , файл входит в комплект исходников компилятора.
-gnate	Сообщения об ошибках не накапливаются до завершения обработки исходника, а выдаются по ходу компиляции. При этом сообщения могут возникать непоследовательно, но они все-таки будут выведены в случае аварийного завершения компиляции.

Опция	Описание
-gnate	Включает режим проверки динамического доступа перед генерированием вызовов подпрограмм и выработки последовательных реализаций методов.
-gnatf	При применении этой опции выдаваемые компилятором сообщения об ошибках могут быть избыточными. Например, если обычно по необъявленной переменной сообщение выдается только один раз, то при этой опции сообщение об ошибке выскакивает каждый раз, как встречается ссылка на эту переменную.
-gnatg	Применяет стили, определенные подпрограммами в исходном файле styles.adb (часть компилятора). Элементы стилей документированы в комментариях этого исходного файла. Обычно эта опция используется для компиляции модулей самого компилятора <i>Ada</i> .
-gnatich	Значение ch — одна литера, указывающая распознаваемую компилятором кодировку буквенных знаков. Все символы указанного набора кодировки могут быть использованы в текстовых константах и именах идентификаторов. Возможны следующие значения ch :
	1: Набор знаков <i>Latin-1</i> . Значения 0–127 используются для кодировки стандартного набора знаков ASCII. Значения от 128 до 255 представляют дополнительные символы европейских алфавитов, такие как германские гласные с верхними точками и шведское долгое "A".
	2: Набор знаков <i>Latin-2</i> .
	3: Набор знаков <i>Latin-3</i> .
	4: Набор знаков <i>Latin-4</i> .
	P: Набор знаков IBM PC кодовой страницы 437. Сходен с набором <i>Latin-1</i> , но отличается кодировкой значений 128–255.
	8: Набор знаков IBM PC кодовой страницы 850. Это модификация набора кодовой страницы 437, она включает все символы набора <i>Latin-1</i> , но их кодировка отличается.
	F: Все коды символов от 128 до 255 доступны и имеют собственное определенное значение. Это делает возможным применение пользовательских символьных наборов (обычно используется для представления китайских знаков).
	H: Диапазон значений 128–255 не используется, этот формат совместим со стандартом <i>Ada 83</i> .
-gnatjch	Значение ch — одна литера, указывающая формат применяемых в литерных константах и идентификаторах расширенных буквенных знаков. ch может иметь следующие значения:
	N: Формат расширенных знаков не указывается. Это значение применяется по умолчанию.
	H: Шестнадцатиричная кодировка. Каждый расширенный буквенный символ представлен последовательностью из пяти знаков ASCII. Первая из них — символ \# . Следующие четыре — шестнадцатиричные цифры в верхнем регистре, представляющие 16-битный код буквенного символа.
	U: Перекодировка "верхней" половины алфавита. Первый бит начального байта последовательности установлен в единицу. Таким образом, диапазон применяемых значений составляет от #16#8000# до #16#FFF#. Учтите, что при этом исключается использование верхнего регистра латинских букв знакового набора <i>Latin-1</i> .
	S: Сдвигнутая кодировка <i>JIS</i> . Она похожа на перекодировку "верхней" половины алфавита. Отличие состоит в том, что представление расширенного символа состоит из двух последовательных знаков. Первый из них также имеет установленный верхний бит, поэтому доступный диапазон кодировки составляет от #16#80# до #16#FF#. Учтите, что при этом также исключается использование верхнего регистра набора <i>Latin-1</i> .

Опция	Описание
	E: Кодировка <i>EUC</i> . Сходна с перекодировкой "верхней" половины алфавита за исключением того, что каждый расширенный буквенный символ записывается двумя последовательными знаками ASCII, причем каждый из них имеет установленный верхний бит. Первое и последнее значения диапазона соответственно — #16#80# и #16#FF#. Использование букв набора символов <i>Latin-1</i> в верхнем регистре также исключается.
-gnatkl	Значение <i>л</i> — число в пределах от 1 до 999, назначает наибольшую допустимую длину имен обрабатываемых файлов, это не влияет на суффиксы .adb и .ads.
-gnat1	Выводится весь исходный файл с номерами строк и всеми сообщениями об ошибках в указанном опцией -gnatv формате.
-gnatml	Указывает предельное число выводимых компилятором сообщений об ошибках. Диапазон применимых для <i>л</i> значений — от 1 до 999. Например, -gnatm3 разрешает вывод трех сообщений об ошибках, после чего дальнейшая компиляция отменяется. По умолчанию число сообщений не ограничено.
-gnatn	Включает расширение подстановкой кода в пределах одного модуля и, при использовании pragma inline , между компилируемыми модулями. Эта опция действует только совместно с флагом оптимизации -O .
-gnatN	То же, что и -gnatn , но только pragma inline автоматически применяется ко всем исходным файлам.
-gnato	Задействует проверку на переполнение арифметических операций при выполнении программы. При этом вырабатывается более "тяжелый" и объемный код из-за добавления к арифметическим операциям проверок условий переполнения и деления на ноль.
-gnatp	Подавляет создание проверок, действующих во время выполнения программы, точно так же, как при включении в исходный код директивы pragma Supress (all_checks) . Повышает производительность за счет снижения защиты от неправильной обработки данных.
-gnatq	Эта опция вынуждает компилятор продолжать работу независимо от присутствия в исходнике синтаксических ошибок. Это может приводить к появлению большого количества дальнейших сообщений об ошибках, к аварийному завершению компиляции или генерации кода с непредсказуемым поведением.
-gnatr	Опция проверяет построение исходного кода на соответствие соглашениям, перечисленным в официальном руководстве по программированию на языке Ada. Нарушения таких соглашений считаются синтаксическими ошибками.
-gnatv	Выполняет синтаксические проверки исходника и на этом останавливает компиляцию. Выходные файлы при этом нерабатываются. При использовании этой опции допускается указание в команде нескольких исходных файлов (при этом необходимо также указывать флаг -c).
-gnatt	Внутреннее дерево содержания будет записываться компилятором в отдельный файл. Этот файл будет нести то же имя, что и исходный, только его расширение для исходных файлов тела кода будет .adb, а для файлов спецификаций — .adt.
-gnatu	Компилятор будет выводить на стандартное устройство вывода список всех модулей, прямо или косвенно задействованных в текущей компиляции.
-gnatv	Ко всем сообщениям об ошибках применяется формат, предоставляющий дополнительную информацию. В применяемом по умолчанию формате содержатся поля имени файла, номеров строки и колонки и описание сообщения, например:
	hloword.adb:2:01: incorrect spelling of the keyword "procedure"
	С опцией -gnatv то же самое сообщение будет примерно таким:

Опция	Описание
	Compiling helloworld.adb (source file time stamp 2002-05-13 20:00:29) 2. procedure HelloWorld is >>> incorrect spelling of the keyword "procedure"
-gnatwe	Все предупреждения воспринимаются как сообщения об ошибках. Сами сообщения при этом не изменяются, но любое предупреждение при этой опции отменяет выработку объектного файла.
-gnatwl	Вырабатывает предупредительные сообщения, относящиеся к порядку выработки программы.
-gnatws	Подавляет вывод всех предупредительных сообщений.
-gnatwu	Выдает предупреждения по объявленным, но нигде не использованным включениям, а также при отсутствии ссылок на какие-либо участвующие в пакете процедуры. Предупреждения выдаются и при неиспользовании любого содержимого, включаемого по оператору <code>with</code> .
-gnatx	Подавляет информацию о перекрестных ссылках, обычно включаемую в файлы <code>.ali</code> . Это экономит некоторое пространство на диске, но делает невозможным применение средств, которые используют такую информацию, — <code>gnatfind</code> и <code>gnatxref</code> .

Все опции из таблицы 9.3 имеют одну или две определяющие буквы и могут назначаться отдельно или объединяться в одну опцию. Следующий пример команды назначает компиляцию с выводом подробных сообщений и включенным режимом динамических проверок:

```
$ gcc -gnatv -gnatE -c helloworld.adb
```

Та же пара опций может быть скомбинирована следующим образом:

```
$ gcc -gnatvE -c helloworld.adb
```

Утилиты, связанные с компиляцией программ на языке Ada

Вместе с компилятором *Ada* поставляется набор утилит. Некоторые из них, такие как `gnatbind` и `gnatlink`, требуются при разработке программ, другие применяются в различных особых обстоятельствах. Эти утилиты предоставляют достаточно широкий выбор способов исследования исходного кода на языке *Ada*. Такие средства особенно важны при работе над большими проектами или при необходимости исследования чужих исходников.

gnatbind

Утилита `gnatbind` выполняет действия по созданию подшивок пакетов *Ada* в таком порядке:

1. Проверяет целостность программы и выдает сообщения при обнаружении какой-либо несогласованности или любого несоответствия между различными модулями программы.

2. Определяет возможность соблюдения стандартного порядка согласования при выработке программы и выдает сообщение об ошибке в случае невозможности его применения.
3. Генерирует небольшую программку на языке *C*, которая будет использована как главная процедура при окончательной компоновке готовой к выполнению программы. Эта программка вначале вызывает инициализирующие подпрограммы, которые подготавливают пакеты, и затем передает управление главной процедуре на языке *Ada*.
4. Определяет список предназначенных для компоновки объектных файлов. Этот список вставляется в упомянутую уже программку на языке *C* таким образом, чтобы он был доступен утилите **gnatlink**.

Утилита **gnatbind** требует на входе вырабатываемый компилятором файл **.ali**. Утилита сканирует другие **.ali** и исходные файлы, тщательно проверяя целостность программного пакета. Если какие-либо используемые программой исходные файлы изменились и не были скомпилированы, то **gnatbind** сразу определяет такую ситуацию и сообщает о ней.

В результате подшивки участвующих в пакете модулей выводится весь исходный код программы. По умолчанию программе присваивается имя входного файла **.ali**. Дополнительно создаются два файла, их имена начинаются с **b~** и они имеют суффиксы **.ads** и **.adb**. При указании опции **-C** генерируется исходный файл на языке *C*, имя этого файла имеет суффикс **.c**.

В таблице 9.4 перечислены опции командной строки для запуска **gnatbind**.

Таблица 9.4. Опции командной строки, поддерживаемые утилитой **gnatbind**

Опция	Описание
-ai directory	Указывает расположение для поиска исходных файлов.
-ao directory	Указывает расположение для поиска файлов .ali .
-b	Записывает краткие сообщения об ошибках в отдельный файл, даже если указана опция -v .
-c	По этой опции выходной файл не вырабатывается, выполняется обработка всех входных файлов и выдаются все сообщения об ошибках.
-C	На выходе вырабатывается исходный файл программы на <i>C</i> вместо файла на языке <i>Ada</i> .
-e	Печатает на стандартное устройство вывода полный список зависимостей последовательности выработки программы, включая описание вида и причин зависимостей.
-E	Сохраняет информацию обратной трассировки для объектов Exception .
-h	Выводит краткое описание списка зависимостей.
-I directory	Указывает расположение для поиска любых файлов (как исходных, так и .ali).
-I-	Отменяет поиск файлов в текущем каталоге и в расположении назначаемого в командной строке файла .ali .
-K	Печатает на стандартное устройство вывода список опций, передаваемых компоновщику (<i>linker</i>). Этот список соответствует списку в генерируемом файле .adb .

Опция	Описание
-1	Выводит избранный порядок выработки программы.
-Lxxx	Для построения библиотек (программы Ada без главной процедуры) имена программ <code>adainit</code> и <code>adafinal</code> изменяются на <code>xxxinit</code> и <code>xxxfinal</code> .
-lwarning	Ограничивает число выводимых сообщений об ошибках до значения <code>lwarning</code> . Это число в диапазоне от 1 до 999. При его достижении обработка прерывается.
-Mxxx	Заменяет имя вырабатываемой главной программы с <code>main</code> на <code>xxx</code> .
-n	Отсутствие главной программы (это означает, что главная программа написана не на Ada).
-nostdtinc	Указывает не проводить поиск исходных файлов в доступном по умолчанию каталоге системы.
-nostdlib	Указывает не проводить поиск библиотечных файлов в доступном по умолчанию каталоге системы.
-o filename	Назначает имя выходного файла <code>filename</code> вместо применения по умолчанию правила присвоения такому файлу имени <code>b_name.c</code> , где <code>name</code> — имя входного файла.
-o	Распечатывает список необходимых для выполнения компоновки объектных файлов.
-p	Указывает применить наихудший ("pessimistic") порядок построения.
-r	Распечатывает на стандартный вывод список дополнительных pragma-ограничений.
--RTS=dir	Назначает использующийся по умолчанию каталог <code>dir</code> для поиска исходных и объектных файлов.
-s	Все исходные файлы должны присутствовать и быть проверенными на соответствие. Обычно <code>gnatbind</code> игнорирует любые пропуски исходных файлов, но при этой опции требуется наличие всех файлов исходного кода, от которых зависит компиляция главной подпрограммы.
-Sxx	Определяет способ инициализации скалярных величин. При указании в поле <code>xx</code> значения <code>1p</code> они инициализируются недопустимой для их типа величиной. При указании в этом поле <code>lo</code> эти величины инициализируются наименьшим для своего типа значением, при указании <code>hi</code> — наибольшим значением. Любая другая пара буквенных символов интерпретируется как шестнадцатиричные цифры, побайтно назначающие начальное значение.
-shared	Назначает компоновку с использованием разделяемых динамически загружаемых библиотек.
-static	Назначает компоновку с использованием статических версий библиотек.
-t	Сообщения об ошибочных отметках времени считаются предупреждениями. В результате применения этой опции отключаются проверки целостности и соответствия.
-Tплл	Устанавливают значение интервала нарезки времени в <code>плл</code> микросекунд, <code>плл</code> — целое число больше нуля.
-v	Вырабатывает сообщения с описанием ошибок и направляет их на стандартное устройство вывода вместо записи в файл, куда эти сообщения выводятся по умолчанию.
-we	Воспринимает любое предупредительное сообщение как критическую ошибку.

Опция	Описание
-wb	Подавляет вывод предупреждительных сообщений.
-x	Отключает проверку исходных файлов. Проверяются только файлы .ali на соответствие между собой. Эта опция ускоряет работу, но при этом не будут определены и поэтому пропущены изменения исходных файлов. Эту опцию целесообразно использовать в компоновочном файле (makefile), потому что между компиляцией и применением утилиты gnatbind изменения исходных файлов не ожидаются. Утилита gnatmake использует эту опцию для задействования gnatbind .
-z	Указывает на отсутствие главной подпрограммы.

Для выполнения проверок соответствия утилита **gnatbind** должна находить сопровождающие программу исходные и файлы .ali. Поиск каждого из файлов выполняется в следующем порядке:

- В каталоге, в котором расположен указанный в команде файл .ali (не обязательно текущий каталог). Поиск в этом каталоге пропускается при указании в команде опции -I-.
- Во всех каталогах, указанных в команде опциями -I-.
- Только для исходных файлов (не относится к файлам .ali) — в каждом из каталогов, перечисленных в переменной окружения **ADA_INCLUDE_PATH**. Эта переменная содержит список путей, разделенных точкой с запятой (тот же формат, что и для переменной окружения **PATH**).
- Только для файлов .ali (не относится к исходным файлам) — в каждом из каталогов, перечисленных в переменной окружения **ADA_OBJECTS_PATH**. Эта переменная содержит список путей, разделенных точкой с запятой (формат переменной окружения **PATH**).
- Каталог инсталляции компилятора Ada. Он назначается во время инсталляции этого компилятора.

gnatlink

Утилита **gnatlink** служит для компоновки объектных файлов Ada в готовую исполняемую программу. Утилита является верхним уровнем для задействования компоновщика через программу **gcc**, она обеспечивает компоновщика точным списком объектных файлов и библиотек. Сама она использует выходные файлы программы **gnatbind** для определения порядка выполнения компоновки.

Большая часть требуемой **gnatlink** информации хранится в файлах, вырабатываемых утилитой **gnatlink**. Поэтому применяется совсем немного опций командной строки. Все они перечислены в таблице 9.5. Последовательность различных элементов в командной строке может иметь большое значение. Общее их расположение должно быть таким:

```
$ gnatlink [options] mainprog.ali [non-ada object] [linker options]
```

В начале идут опции (**options**) самой утилиты **gnatlink**, затем — имя файла .ali головной процедуры. За ним (**non-ada object**) — список включаемых в исполняемый код объектных файлов, выработанных компиляцией с других, кроме Ada,

языков. Дальнейшие опции (*linker options*) передаются компоновщику (*linker*) на заключительном этапе компоновки.

Таблица 9.5. Опции командной строки, поддерживаемые утилитой *gnatlink*

Опция	Описание
-A	Промежуточный исходный файл, вырабатываемый утилитой <i>gnatbind</i> , ожидается в формате программы на языке Ada. Применяется по умолчанию.
-b target	Исходный файл, вырабатываемый утилитой <i>gnatbind</i> , должен компилироваться для указанной целевой платформы.
-B directory	Загружает исполняемые программы для компиляции и компоновки из указанного каталога.
-C	Ожидаемый на выходе утилиты <i>gnatbind</i> промежуточный исходный файл, должен иметь формат программы на языке C.
-f	Печатает список задействованных в компоновке программы объектных файлов.
-g	При этой опции в компонуемую программу включается отладочная информация и не удаляются временные рабочие файлы утилиты <i>gnatbind</i> .
--GCC=паме	Указывается имя программы, выполняющей верхний уровень (front end) компиляции. По умолчанию применяется <i>gcc</i> .
--LINK=паме	Указывается имя программы, выполняющей верхний уровень компоновки. По умолчанию — <i>gcc</i> .
-n	При этой опции файлы, вырабатываемые утилитой <i>gnatbind</i> , не компилируются.
-o	Имя исполняемой программы, вырабатываемой <i>gnatlink</i> .
-v	Вывод программой подробных описаний событий и ошибок. Опция может быть указана дважды для вывода еще более подробных описаний.

gnatmake

Утилита *gnatmake* — это программа, действующая подобно стандартной утилите *make*, только она более адаптирована для языка Ada и учитывает его особые требования. Вы можете ввести простую команду *gnatmake* с указанием имени файла главной процедуры. По ней вся программа будет скомпилирована и скомпонована в исполняемый код. Для каждого исходного файла определяется набор всех необходимых исходных и объектных файлов. Каждый объектный файл при этом сравнивается с исходным файлом на соответствие, при необходимости компилируется исходный файл.

Утилита *gnatmake* имеет большое количество опций, они перечислены в таблице 9.6. Некоторые из опций также используются и *gnatmake*, но большая их часть передается через *gcc* утилитам *gnatbind* или *gnatlink*. Учтите, что опции **-P**, **-vRx** и **-Xpm** относятся к *файлу проекта*, который использует особые свойства редактора *Emacs* (версии 20.2 или новее). Этот редактор позволяет редактировать и поддерживать такие файлы в соответствии с требованиями конфигурирования и управления процессом компиляции.

Таблица 9.6. Опции командной строки утилиты `gnatmake`

Опция	Описание
<code>-a</code>	Проверяет все входные файлы, включая файлы <code>.ali</code> с атрибутом "только для чтения". По умолчанию проверка таких файлов пропускается.
<code>-aidirectory</code>	Указываемый этой опцией каталог включается в список путей для поиска исходных файлов.
<code>-alddirectory</code>	Файлы <code>.ali</code> в указанном каталоге считаются принадлежащими другому исходному пакету. Утилита <code>gnatmake</code> не будет делать попыток их проверять или компилировать. Это действует так же, как если бы эти файлы имели атрибут "только для чтения" (имеется в виду применение утилиты без опции <code>-a</code>).
<code>-aodirectory</code>	Указываемый каталог включается в список путей для поиска объектных файлов и библиотек.
<code>-Adirectory</code>	Эта опция действует так же, как и одновременное применение опций <code>-aidirectory</code> и <code>-alddirectory</code> с одинаковым каталогом <code>directory</code> .
<code>-bargs list</code>	Следующий за <code>-bargs</code> список опций <code>list</code> передается утилите <code>gnatbind</code> . Это могут быть любые опции, представленные в таблице 9.4.
<code>-c</code>	Указывает выполнять только компиляцию. При этом <code>gnatbind</code> и <code>gnatlink</code> не вызываются. Эта опция действует по умолчанию, если назначаемый в командной строке исходный файл не является главной процедурой.
<code>-cargs list</code>	Следующий за <code>-cargs</code> список опций <code>list</code> передается компилятору. Это могут быть как любые специфические для компилятора Ada опции, перечисленные в таблице 9.4, так и любые опции общего назначения из приложения Г.
<code>-f</code>	Включает такой режим компиляции, при котором заново компилируются измененные исходные файлы в зависимости от отметок времени в вырабатываемых из них объектных файлах.
<code>--GCC=name</code>	Использует в качестве верхнего уровня (front end) компилятора программу с указанным именем <code>name</code> . По умолчанию применяется <code>gcc</code> .
<code>--GNATBIND=name</code>	Использует в качестве программы подшивки пакетов — биндера — программу с указанным именем <code>name</code> . По умолчанию применяется <code>gnatbind</code> .
<code>--GNATLINK=name</code>	Использует <code>name</code> как команду к компоновщику. По умолчанию — <code>gnatlink</code> .
<code>-i</code>	Указывает выполнять все действия по компиляции файлов на месте их расположения с заменой существующих <code>.ali</code> файлов. При отсутствии таких они будут создаваться в каталогах расположения исходных файлов. По умолчанию все файлы <code>.ali</code> создаются в текущем каталоге.
<code>-Idirectory</code>	Эта опция действует так же, как и одновременное применение опций <code>-aidirectory</code> и <code>-aodirectory</code> с одинаковым именем каталога <code>directory</code> .
<code>-I-</code>	Указывает не искать другие исходные файлы в каталоге расположения того исходного файла, который указан в команде.
<code>-j number</code>	Разрешает использовать количество параллельных процессов до величины <code>number</code> для выполнения компиляций и перекомпиляций. При этом сообщения от разных компилирующих процессов могут перемешиваться между собой.
<code>-k</code>	Указывает продолжать компиляцию после возникновения критической ошибки. При этом будет сделана попытка компиляции всех исходных файлов, список неудачно скомпилированных файлов будет выведен до завершения работы <code>gnatmake</code> .

Опция	Описание
-largs list	Следующий за -largs список опций <i>list</i> передается компоновщику gnatlink . Это могут быть любые опции, перечисленные в таблице 9.5.
-Idirectory	Добавляет каталог с именем <i>directory</i> к списку каталогов для поиска библиотек.
-m	Выполняет минимальное количество рекомпиляций. При этой опции игнорируются отметки времени в объектных файлах, если сделанные в исходных файлах изменения относятся к комментариям или форматированию текста.
-M	Распечатывает на стандартное устройство вывода зависимости в формате, пригодном для их помещения в компоновочный файл. Каждый файл помещается в список в соответствии с абсолютным или относительным путем его расположения в зависимости от применения опции -q . При использовании опции -a не выводятся зависимости от системных файлов. В любом случае зависимости от внешних библиотек не показываются.
-n	Подавляет выполнение шагов компиляции, подшивки пакета и компоновки. По этой опции выполняются только проверки на соответствие версий всех объектных файлов. При обнаружении несоответствий выводится имя первого просроченного файла.
-nostdinc	Отменяет поиск исходных файлов в системном каталоге, установленном по умолчанию.
-nostdlib	Отменяет поиск библиотек в действующем по умолчанию системном каталоге.
-o name	Назначает имя <i>name</i> вырабатываемому в исполняемом коде файлу.
-P name	Использует проектный файл с указанным именем <i>name</i> .
-q	Включает режим минимального вывода. При этом не выводятся команды, подаваемые утилитой gnatmake .
-r	Заново компилирует все файлы, которые ранее компилировались с другими опциями.
-u	Компилирует только указанный в команде файл, при этом игнорируются несоответствия любых зависимостей.
-v	Вывод сообщений с их описаниями. Описания поясняют также причины, по которым необходимы все компиляции или перекомпиляции.
-vRx	Выводит описания сообщений при использовании проектного файла для управления компиляцией.
-Xpm=value	Назначает <i>value</i> в качестве внутренней ссылки для использования проектным файлом.
-z	Указывает на отсутствие головной процедуры, то есть на невозможность компоновки в исполняемый код.

Из-за того, что за опциями **-cargs**, **-bargs** и **-largs** может следовать неограниченное количество передаваемых опций, их нужно ставить в командной строке последними. Общий синтаксис командной строки утилиты **gnatmake** такой:

```
$ gnatmake [options] filename [-cargs ...] [-bargs ...] [-largs ...]
```

После имени **gnatmake** следует список собственных опций *options*, обрабатываемых этой утилитой. Имя файла *filename* может быть указано как с суффиксом *.ali*, так и без него. Список опций, который следует за **-cargs**, передается ком-

пилиятору. Передаваемый список ограничивается одной из опций **-bargs** или **-largs** либо концом строки. Три этих опции могут ставиться в любом порядке. Список опций за **-bargs** передается биндеру, а список после **-largs** — компоновщику.

gnatchop

Утилита **gnatchop** считывает исходный файл и переписывает его в один или более новых исходных файлов, строго соответствующих соглашению об именах стандарта GNAT Ada. Компилятор требует, чтобы каждый отдельный файл содержал только один компиляционный модуль, и чтобы имя файла соответствовало имени модуля. Утилита позволяет сконвертировать весь набор исходных файлов за один проход. Или же вы можете создать список компиляционных команд для конвертирования имен файлов при каждой компиляции программы. Этот список имеет такой же формат, как и компоновочный файл (make file).

Командная строка для запуска **gnatchop** имеет следующий формат:

```
$ gnatchop [options] file [file ...] [directory]
```

По команде выполняется нарезка указанного файла **file** (или нескольких перечисленных файлов). Вырабатываемый при этом файл (или файлы) помещаются в указанный каталог **directory**. Если каталог не указан, то вырабатываемые файлы сохраняются в текущем каталоге. Допустимые опции (**options**) команды представлены в таблице 9.7.

Таблица 9.7. Опции командной строки утилиты **gnatchop**

Опция	Описание
-c	Задействует режим компиляции. Прагмы конфигурации в нарезаемом файле устанавливаются в соответствии с правилами стандарта Ada 95.
-gnatxx	Любая указываемая опция -gnat... передается процессу синтаксического разделения (парсеру).
-k [number]	Длина имен вырабатываемых файлов ограничивается до количества букв, равного number . Если number не указывается, то по умолчанию количество знаков равно 8.
-q	"Тихий" режим работы программы. При этой опции подавляется вывод списка имен входных и выходных файлов.
-r	В выходные файлы включаются прагмы Source_Reference . Это можно использовать в случае, когда выходные файлы являются временными рабочими файлами. Компилятор при выводе предупреждений и сообщений об ошибках будет использовать информацию этих прагма-директив. Сообщения об ошибках и предупреждения будут ссылаться на оригиналный файл вместо компилируемого нарезанного файла. Отладочная информация, вставляемая по опции -g , также будет ссылаться на оригиналный исходный файл.
-v	Вывод включает в себя описание сообщений. При этом на стандартный выход выводятся все генерируемые утилитой команды.
-w	Переписывает существующие файлы при совпадении их имен с именами выходных файлов.
-x	Прекращает работу программы при любом сообщении об ошибке.

gnatxref

Утилита **gnatxref** считывает и показывает информацию, сохраняемую компилятором в файле типа **.ali**. Применяется следующий формат команды:

```
$ gnatxref [options] file [file ...]
```

Каждый файл из передаваемого утилите списка является файлом **.ali**. По команде последовательно выводится алфавитный список процедур для каждого пакета. Для каждой процедуры выводится информация о месте ее объявления, расположения кода реализации, перечисляются все обращения к ней. Список доступных опций команды представлен в таблице 9.8.

Таблица 9.8. Опции командной строки утилиты gnatxref

Опция	Описание
-a	Включаются все файлы. Обычно содержимое .ali файла, имеющего атрибут "только для чтения", не выводится.
-aidirectory	Добавляет указанный каталог в список путей для поиска входных исходных файлов.
-aodirectory	Добавляет указанный каталог в список путей для поиска библиотек и объектных файлов.
-d	Добавляет к выводу перекрестных ссылок информацию о наследовании типов.
-f	Имена файлов в списках перекрестных ссылок перечисляются с полными именами путей. По умолчанию выводятся только собственные имена файлов.
-g	Ограничивает представление программных символов в перекрестных ссылках только включениями уровня библиотек. Локальные включения пропускаются.
-Idirectory	То же, что одновременное указание опций -aidirectory и -aodirectory с одним именем каталога.
-filename	Названный файл используется как файл проекта. По умолчанию gnatxref будет искать этот файл в текущем каталоге.
-u	Включает в вывод только неиспользуемые программные символы.
-v	Вместо листинга перекрестных ссылок выводится текст в форме файла тегов, который может использоваться редактором vi .

gnatfind

Утилита **gnatfind** считывает информацию из файлов **.ali** и находит указанный в командной строке предмет поиска. На выходе выводится список всех расположений, где обнаружены включения искомого предмета. Вот синтаксис командной строки для этой утилиты:

```
$ gnatfind [options] pattern[:filename[:line[:column]]] [file ...]
```

Указываемый шаблон **pattern** является подмножеством поддерживаемых утилитой **grep** регулярных выражений. Он может включать в себя знак "звездочка" ('*') для представления группы любых буквенных знаков, знак вопроса (?) для представления любого отдельного знака и конструкцию с квадратными скобками ('[...]')

для указания соответствия особому набору или диапазону буквенных символов. Как видно из представленного синтаксиса команды, возможно ограничение поиска пределами одного файла и даже указанием отдельной строки и номера колонки. Если в команде указывается файл или список файлов, то поиск будет выполняться только в них.

Допустимые опции командной строки перечислены в таблице 9.9.

Таблица 9.9. Поддерживаемые опции командной строки для утилиты gnatfind

Опция	Описание
-a	Обрабатываются все файлы. По умолчанию файлы .ali, имеющие атрибут "только для чтения", пропускаются.
-aidirectory	Включает указанный каталог в список путей для поиска входных исходных файлов.
-aodirectory	Включает указанный каталог в список путей для поиска библиотек и объектных файлов.
-d	Добавляет к выводу информацию о наследовании типов.
-e	Воспринимает полный синтаксис регулярных выражений, кроме поддерживаемых по умолчанию "звездочки", вопросительного знака и пары квадратных скобок. Полное множество специальных символов регулярных выражений для обозначения операторов включает в себя следующий набор буквенных знаков:
	[] . * + ? ^
-f	В выводимом листинге имена файлов включают в себя полный путь к ним, по умолчанию выводятся только собственно имена файлов.
-g	Ограничивает представление выводимых программных символов только включениями уровня библиотек. Локальные включения пропускаются.
-Idirectory	То же, что одновременное указание опций -aidirectory и -aodirectory с одним именем каталога.
-pfilename	Названный файл используется как файл проекта. По умолчанию gnatxref ожидает, что этот файл находится в текущем каталоге.
-r	Находит и выводит все ссылки. По умолчанию выводятся только объявления.
-s	Распечатывает на стандартный выход полную строку исходного файла кроме вывода информации о ее расположении.
-t	Выводит иерархию типов каждого найденного предмета поиска.

gnatkr

Утилита **gnatkr** вырабатывает сокращенные имена в соответствующем компилятору Ada стандарте из передаваемых ей длинных имен. Несмотря на применение особого набора правил для сокращения длины уникальность вырабатываемых названий не гарантируется. По умолчанию имя файла сокращается до 8-ми символов, но возможно указание и другой длины, как это видно из формата команды:

```
$ gnatkr name [length]
```

При сокращении вначале выполняется разделение указанного в команде имени на части по позициям дефисов или знаков подчеркивания. Затем — последовательное сокращение каждого члена в обратном порядке до достижения указанной длины. Вот несколько примеров:

```
$ gnatkr longer-names-can-be-crunched
lncabecr
$ gnatkr The_Ada_Names_Are_Long
tanaarlo
$ gnatkr The_Ada_Names_Are_Long 5
tanal
```

gnatprep

Утилита **gnatprep** может использоваться как простейший препроцессор исходного кода на языке *Ada*. В командной строке требуется указание имен как входного, так и выходного файлов. Все определения для предобработки должны находиться в третьем указываемом файле либо определяться в командной строке. Синтаксис командной строки следующий:

```
$ gnatprep inputfile outputfile [definitionsfile] [options]
```

Указание имен файлов входного (*inputfile*) и выходного (*outputfile*) обязательно, включая их суффиксы. Вследствие того, что обычно выходной файл предназначается для компиляции, его имя обычно имеет суффикс **.adb** или **.ads**. Допустимые опции команды **gnatprep** перечислены в таблице 9.10.

Таблица 9.10. Опции командной строки для утилиты **gnatprep**

Опция	Описание
-b	Заменяет каждую предобработанную строку пустой строкой. По умолчанию строка удаляется.
-c	Оставляет предобработанные строки в выходном исходном файле как комментарии. Каждая такая строка маркируется последовательностью " - ! ".
-Dsymbol=value	Определяет символ указанным значением точно так же, как если бы он был включен в файл определений такой строкой: symbol := value
-r	Генерирует pragma-директиву source_Reference так, чтобы все сообщения об ошибках и отладочная информация ссылались на первичный непредобработанный файл. Несмотря на определение опции -c , эта опция применяет также и опцию -b для сохранения порядка нумерации строк первичного файла.
-s	Выводит отсортированный список символов, определенных для предобработки, вместе с их значением.
-u	При этой опции директивы #if воспринимают все неопределенные символы как имеющие значение false .

Необязательный файл определений (*definitionsfile*) может содержать определения символов для предобработки (одно или более) в следующем формате:

```
symbol := value
```

Значение (*value*) определяемого символа (*symbol*) может быть пустым, может быть ограниченной одинарными кавычками строкой литер или любым набором буквенных знаков допустимого диапазона и в доступной для компилятора *Ada* кодиров-

ке. В отличие от препроцессора C (CPP) **gnatprep** не подставливает каждое вхождение, совпадающее с именем символа. Предназначенный к подстановке символ должен быть специально помечен в программе знаком доллара ('\$'). К примеру, предположим, что файл определений содержит такую строку:

```
bracklin := thermolimit
```

В этом случае предобработка приведет к подстановке значением **thermolimit** каждого вхождения строки **\$bracklin**, найденного во входном файле. Также во входном файле могут использоваться директивы **#if**, **#elseif** и **#endif if** для управления условной компиляцией в зависимости от того, каким значением определен проверяемый символ — **true** или **false**. Вот пример применения условных директив предобработки:

```
#if condrep then
    Put_Line("condrep is defined as true");
#elsif
    Put_Line("condrep is defined as false");
#end if;
```

Логика приведенного выражения может быть обращена применением оператора **not**:

```
#if condrep then
    Put_Line("condrep is defined as false");
#else
    Put_Line("condrep is defined as true");
#endif if;
```

gnatls

Утилита **gnatls** является просмотрщиком содержимого библиотек. Она может быть использована для извлечения и вывода информации о скомпилированных модулях. Она показывает взаимоотношения между объектами, именами модулей и исходными файлами. Утилита также может быть использована для определения зависимостей между компиляционным модулем и исходными файлами. Входными файлами для утилиты **gnatls** могут быть вырабатываемые компилятором файлы как типа **.ali**, так и объектные файлы **.o**.

Формат вывода по умолчанию состоит из четырех колонок. В первой колонке выводится имя анализируемого объектного файла, во второй — имя головного модуля этого объектного файла, в третьей — статус соответствующего модулю исходного файла, в четвертой — имя этого исходного файла. Возможные коды статуса (состояния) исходных файлов приведены в таблице 9.11.

Таблица 9.11. Коды статуса исходных файлов, выводимые утилитой **gnatls**

Код состояния	Описание
???	Исходный файл не найден.
DIF	Найдено не менее одного файла исходного кода, но ни один из найденных исходных файлов не соответствует версии объектного модуля.

Код состояния	Описание
НД	Найден исходный код, точно соответствующий объектному модулю, но, по крайней мере, один из исходных файлов, найденных раньше этого соответствующего файла, объектному модулю не соответствует. Иначе говоря, точно соответствующий исходный код скрыт от компилятора другими файлами.
ИСК	После последней выработки объектного файла в исходном коде делались незначительные изменения, не требующие повторной компиляции. Такие изменения могут относиться к форматированию исходного текста программы или к комментариям.
OK	Объектный файл вполне соответствует своему исходному коду.

Опции командной строки, поддерживаемые утилитой **gnatls**, перечислены в таблице 9.12. Они позволяют управлять содержанием и формой представления информации, выводимой **gnatls**, а также определять пути для поиска файлов.

Таблица 9.12. Опции командной строки для gnatls

Опция	Описание
-a	Добавляет к выводимой информации сведения об использовании предопределенных модулей, имеющих отношение к указанному в команде файлу. При этом выводится весь список таких модулей, включая и те, которые предопределены в стандартных библиотеках языка Ada.
-aidirectory	Указанный в этой опции каталог добавляется в список расположений для поиска исходных файлов.
-aodirectory	Указанный в этой опции каталог добавляется в список расположений для поиска объектных файлов.
-d	Включает в выводимый список файлов имена таких исходных файлов, с которыми файл, указанный в команде, имеет компиляционные зависимости.
-h	Распечатывает на стандартный выход этот список опций.
-ldirectory	То же, что и одновременное применение опций -aidirectory и -aodirectory с одним именем каталога.
-I-	Отменяет поиск исходных и объектных файлов в системном каталоге, применяемом по умолчанию.
-nostdinc	Отменяет поиск исходных файлов в системном каталоге, применяемом по умолчанию.
-o	Ограничивает выводимую информацию только сведениями об объектных файлах.
-Pname	Использует указанный файл проекта.
-v	Ограничивает выводимую информацию только сведениями об исходных файлах.
-u	Ограничивает выводимую информацию только сведениями о компиляционных модулях.
-v	Генерирует вывод описаний, включающих полные пути к исходным и объектным файлам. Выводятся также и разъяснения (на английском языке) терминов, выводимых с именами файлов.
-v#number	Устанавливает уровень подробности выводимых описаний. Возможные значения для number — 0, 1 или 2.
-Xsymbol=value	Определяет значение внешнего параметра.

gnatsys и gnatsta

Утилита **gnatravus** выводит исходный код пакета на языке *Ada*, содержащий все системные размерности и характеристики системы, на которой она запущена. Выводимый код включает в себя системные определения таких величин как наибольшие и наименьшие значения целых чисел, порядок точности чисел с плавающей точкой, начальное аппаратное значение целого числа по умолчанию, наибольший размер адресуемой памяти и аппаратный порядок представления байтов (обратный или прямой).

Вывод утилиты **gnatsta** — исходный код пакета *Ada*, содержащий присвоенные значения определениям, зависимым от реализации. Это включает в себя наибольшие и наименьшие значения чисел с плавающей точкой, весь распознаваемый компилятором набор знаковой кодировки и используемый метод представления расширенных буквенных знаков.

Эти утилиты не имеют опций командной строки. При запуске они динамически определяют все выводимые значения.

Глава 10



Совмещение языков

Иногда обстоятельства требуют объединения в одной программе исходных частей, написанных на разных алгоритмических языках. Обычно это происходит когда требуется совместимость существующего тела кода программы с частями другой программы, написанной на ином языке. Так бывает при объединении проектов, отделов или даже компаний. Другой распространенной причиной комбинирования языков является потребность в использовании программой, написанной на одном языке, возможностей другого языка. Чаще всего при написании программы на языке высокого уровня оказывается весьма полезным применение возможностей языка системного уровня (например, языка C). Также приходится применять два языка в одном программном проекте для поддержки уже написанных частей при изменении корпоративной политики.

В этой главе обсуждается смешивание языков внутри семейства GCC. Возможно смешивать языки компоновкой получаемых от различных компиляторов объектных модулей, хоть это и намного сложнее. Сложность заключается в своеобразии задействуемых при этом компиляторов, и порой это может приводить к непредсказуемым затруднениям и даже неразрешимым ситуациям. GCC использует один нижний уровень (back end) для производства единообразного объектного кода при компиляции любых языков. Причем даже обновления компилятора не влияют на правильность работы ранее выполненных модулей и программ. Благодаря этому обстоятельству в рамках GCC возможна компоновка объектных модулей, написанных на разных языках и даже скомпилированных в различных версиях компилятора. Конечно, никаких гарантий в GCC не существует, и возможны некоторые нестыковки, но они, как правило, разрешимы. Широко распространенная в GCC практика объединения различных языков в одном проекте ведет к быстрому устранению любых возникающих затруднений.

При совмещении языков могут возникать ситуации, требующие применения некоторых особых приемов. Дело не только в различиях базовых структур языков. Программист должен быть готов иметь дело с такими вещами, как стандартные соглашения о применении глобальных имен, правила замещения имен, различия в формате передаваемых аргументов, обработка исключений, смешивание стандартных динамических библиотек разных языков.

Совмещение C++ и C

Язык C++ был разработан как расширение языка C, поэтому части исходного кода на этих алгоритмических языках объединяются вполне естественно. Применяются одинаковые соглашения об именах, и базовые типы данных, в основном, те же. Единственное отличие заключается в именах функций. При компиляции с языка C используются простые имена функций без указания количества и типов аргументов. А в C++ имя функции всегда включает в себя список типов аргументов. Однако в языке C++ специально предусмотрена возможность объявления функций на языке C, что, конечно, означает, что программа на языке C++ может непосредственно включать в себя прямые вызовы функций C.

Вызовы функций C из кода на языке C++

В следующем примере программа на языке C++ вызывает функцию C с именем `sayhello()`. Этот вызов является прямым вызовом, что возможно благодаря тому, что функция объявлена в программе C++ как `extern "C"`:

```
/* cpp2c.cpp */
#include <iostream>
extern "C" void csayhello(char *str);
int main(int argc,char *argv[])
{
    csayhello("Hello from cpp to c");
    return(0);
}
```

Функция на языке C не требует особого объявления, она находится в файле `csayhello.c` и представляет собой следующее:

```
/* csayhello.c */
#include <stdio.h>
void csayhello(char *str)
{
    printf("%s\n",str)
}
```

Последовательность следующих трех команд скомпилирует две эти программы и скомпонует их в готовый исполняемый файл. Гибкость применения `g++` и `gcc` позволяет сделать это различными способами, но такой набор команд, вероятно, является наиболее прямым путем:

```
$ g++ -c cpp2c.cpp -o cpp2c.o
$ gcc -c csayhello.c -o csayhello.o
$ gcc cpp2c.o csayhello.o -lstdc++ -o cpp2c
```

Учтите, что при окончательной компоновке в этом примере необходимо указывать стандартную библиотеку языка *C++*, потому что компоновщик (*linker*) задействован здесь командой **gcc**, а не командой **g++**. При использовании команды **g++** по умолчанию применяется стандартная библиотека *C++*.

Распространенным способом является размещение объявлений функций в заголовочном файле (header file) и применение объявления **extern "C"** ко всему содержимому этого файла. При этом применяется стандартный для языка *C++* синтаксис, как показано в следующем примере:

```
extern "C" {
    int mlimitav(int lowend, int highend);
    void updatedesc(char *newdesc);
    double getpct(char *name);
};
```

Вызовы функций *C++* из кода на языке *C*

Чтобы программа на языке *C* могла вызывать функцию из программы на *C++* необходимо, чтобы вызываемая функция *C++* поддерживала последовательность вызова, соответствующую языку *C*. Следующий пример демонстрирует синтаксис создания вызываемой из программы на языке *C* функции программы на *C++*:

```
/* cppsayhello.cpp */
#include <iostream> .

extern "C" void cppsayhello(char *str);
void cppsayhello(char *str)
{
    std::cout << str << "\n";
}
```

Несмотря на то, что функция **cppsayhello()** объявлена как **extern "C"**, она является частью программы на языке *C++*. Это означает, что код самой функции в действительности является кодом на языке *C++*. Вы вполне можете создавать и освобождать объекты внутри этой функции. Кроме того, при необходимости обратного вызова из функции **cppsayhello()** функции из программы на *C* вам придется объявить вызываемую функцию как **extern "C"**. Иначе компилятор воспримет вызываемую функцию как функцию *C++* и соответственно преобразует ее имя.

Далее — программа на языке *C*,зывающая рассмотренную выше функцию *C++*:

```
/* c2cpp.c */
int main(int argc,char *argv[])
{
    cppsayhello("Hello from C to C++");
    return(0);
}
```

Следующая последовательность команд выполнит компиляцию и компоновку программы `c2cpp`:

```
$ g++ -c cppsayhello.cpp -o cppsayhello.o
$ gcc -c c2cpp.c -o c2cpp.o
$ gcc cppsayhello.o c2cpp.o -lstdc++ -o c2cpp
```

Совмещение языков Objective-C и C

Благодаря тому, что язык *Objective-C* представляет собой не что иное, как расширение языка *C* добавлением поддержки классов, довольно просто комбинировать исходные модули, написанные на этих языках. Последовательности вызова в этих языках совпадают, поэтому остается только просто вызвать функцию.

Вызов C-функции из программы на языке Objective-C

Следующая программа *Objective-C* передает адрес переменной строкового типа в функцию на языке *C* `csayhello()`:

```
/* objc2c.m */
#import <stdio.h>
int main(int argc,char *argv[])
{
    csayhello("Hello from Objective-C to C");
    return(0);
}
```

Функция `csayhello()` передает эту строку на стандартное устройство вывода:

```
/* csayhello.c */
#include <stdio.h>
void csayhello(char *str)
{
    printf("%s\n",str);
}
```

Следующая последовательность из трех команд компилирует программу `objc2c` и компонует полученный объектный код в готовый к запуску файл. При компоновке следует указывать опцию `-lobjc` для подключения стандартной разделяемой библиотеки языка *Objective-C*.

```
$ gcc -Wno-import -c objc2c.m -o objc2c.o
$ gcc -c csayhello.c -o csayhello.o
$ gcc objc2c.o csayhello.o -lobjc -o objc2c
```

Вызов функции Objective-C из программы на языке C

Программа в следующем примере вызывает написанную в синтаксисе *Objective-C* функцию `objcSayHello()`:

```
/* c2objc.c */
int main(int argc, char *argv[])
{
    objcsayhello("Hello from C to Objective-C");
    return(0);
}
```

Далее следует код вызываемой функции на языке *Objective-C*:

```
/* objcsayhello.m */
#import <objc/Object.h>
#import "SpeakLine.h"
void objcsayhello(char *str)
{
    id speak;
    speak = [SpeakLine new];
    [speak setString: str];
    [speak say];
    [speak free];
}
```

Как видите, функция `objcsayhello()` создает экземпляр объекта `SpeakLine`, записывает в него символьную строку и затем использует метод `Say` этого объекта для вывода строки. Далее вам предлагаются листинги заголовочного файла (header file) объекта `SpeakLine` и файла с кодом его реализации (implementation file):

```
/* SpeakLine.h */
#import <objc/Object.h>
@interface SpeakLine : Object
{
    char *string;
}
- setString: (char *) str;
- say;
- free;
@end

/* SpeakLine.m */
#import "SpeakLine.h"
@implementation SpeakLine

+ new
{
    self = [super new];
    return self;
}
- setString: (char *)str
{
    string = str;
    return self;
}
- say
{
    printf("%s\n", string);
```

```

        return self;
    }
- free
{
    return [super free];
}

```

Следующая последовательность из четырех команд компилирует весь набор исходных файлов в объектный код и затем компонует два полученных объектных файла в готовую программу:

```

$ gcc -Wno-import -c objcsayhello.m -o objcsayhello.o
$ gcc -Wno-import -c SpeakLine.m -o SpeakLine.o
$ gcc -c c2objc.c -o c2objc.o
$ gcc c2objc.o objcsayhello.o SpeakLine.o -lobjc -o c2objc

```

Сочетание языков Java и C++

Существует возможность использования интерфейса **CNI** (сокращение от "Cygnus Native Interface") для доступа к классам *Java* из программ на языке *C++*. Два этих языка довольно сильно различаются между собой, но все же они имеют определенные фундаментальные сходства:

- Классы объявляются по имени и наследуют свойства других классов.
- Классы содержат функции, они могут замещаться (перегружаться) по соответствию параметров.
- Применяемые типы данных имеют прямые соответствия между языками.
- Правила построения выражений в языке *Java* наследуют синтаксис выражений языка *C*.

GCC компилирует классы и языка *C++* и языка *Java* сходным образом. Благодаря этому необходимо только избегать ситуаций, которые могут возникать по причине наиболее принципиальных различий этих языков. Иногда требуется специально вносить в код поправки, делающие возможным использование классов, написанных на языке *Java*.

Создание Java-объектов и вызов статических методов

Следующий пример программы создает объект *Java*-класса `java.lang.String` и передает его методу `java.lang.System.out()` для вывода:

```

/* cnistrout.cpp */
#include <gcj/cni.h>
#include <java/lang/System.h>
#include <java/io/PrintStream.h>

int main(int argc, char *argv)
{
    java::lang::String *str;

```

```

JvCreateJavaVM(NULL);
JvAttachCurrentThread(NULL,NULL);

str = JvNewStringLatin1("Hello from C++ to Java");
java::lang::System::out->println(str);

JvDetachCurrentThread();
}

```

Эта программа может быть скомпилирована и скомпонована следующей командой:

```
$ g++ cnistrout.cpp -lgcj -o cniexception
```

Заголовочный файл `cni.h` содержит прототипы функций, необходимые для воздействования интерфейса CNI. Также в программе имеются директивы `#include`, которые подключают классы `java.lang.System` и `java.io.PrintStream`. Несложно было бы также подключить и `java.lang.String`, но это уже было бы излишним, поскольку этот класс вместе с некоторыми другими заголовочными файлами (header files) системного уровня всегда подключается в `cni.h`.

В языке *Java* для обработки классов используются ссылки на них (references), поэтому для хранения адреса объекта `java.lang.String` объявляется собственный указатель. Полное имя, применяемое для описания *Java*-класса в синтаксисе языка C++, включает в себя пару двоеточий ("::"). Следует соблюдать это соглашение об именах при каждом обращении к *Java*-классу по имени, если для него не определено именное пространство. Например, классы `String` и `System` могут быть объявлены и использованы следующим образом:

```

using namespace Java::lang;
String *str;
...
System.out->println(str);

```

Вызов функции `JvCreateJavaVM()` инициализирует среду выполнения *Java*-программ. Это включает в себя установку *Java*-интерфейса управления потоками, организацию динамического распределения памяти ("garbage collection" — буквально "сборка мусора") и обработки исключений. Эта функция должна быть вызвана в программе один раз до создания классов *Java* и использования их методов.

Вызов функции `JvAttachCurrentThreadJavaVM()` регистрирует процесс, порождаемый вызывающей программой, в предварительно инициализированной среде *виртуальной машины Java*. ("Java Virtual Machine" или *JVM* — интерпретатор, выполняющий код программ на языке *Java*.) Эта функция также должна быть вызвана в программе один раз до создания *Java*-классов и использования методов *Java*, но только после вызова `JvCreateJavaVM()`.

В конце программы вызов функции `JvDetachCurrentThread()` снимает процесс, ранее зарегистрированный вызовом функций `JvCreateJavaVM()` и `JvAttachCurrentThreadJavaVM()`. При завершении это гарантирует "чистый выход" — полное освобождение ресурсов, занимаемых приложением.

В интерфейсе CNI, *Java*-объекты `Strings` всегда создаются вызовом одной из следующих функций:

- **JvNewString(const char*chars, jsize length)**
Возвращаемый объект **String** указанной длины **length** содержит буквенные символы, содержащиеся в строке типа **chars**.
- **JvNewStringLatin1(const char*bytes, jsize length)**
Возвращаемый объект **String** указанной длины **length** содержит значения из массива переменных типа **byte**.
- **JvNewStringLatin1(const char*bytes)**
Возвращаемый объект **String** содержит значения из массива переменных типа **byte**, но без начального байта с нулевым значением.
- **JvNewStringUTF(const char*bytes)**
Возвращаемый объект **String** содержит значения из массива переменных типа **byte** в кодировке UTF, без первого байта с нулевым значением.

Загрузка Java-класса и создание его экземпляров

Использование интерфейса CNI дает возможность свободно смешивать в одной программе классы *C++* и *Java*. Следующий пример программы состоит из одной основной процедуры *C++* и одного класса *Java*, экземпляр которого создается и используется для записи и вывода строк текста.

Вот класс на языке *Java* с именем **Speak**, предназначенный для хранения и вывода одной строки:

```
/* Speak.java */
public class Speak {
    String string;
    Speak() {
        string = "Uninitialized";
    }
    public void setString(String str) {
        string = str;
    }
    public void showString() {
        System.out.println(string);
    }
}
```

Процедура-конструктор класса **Speak** инициализирует внутреннюю строку со свойствами, применяемыми по умолчанию. Однако это может быть замещено вызовом функции **setString()**. Метод **showString()** может быть вызван для вывода текущей строки на стандартное устройство вывода. Теперь этот класс должен быть скомпилирован в файл типа **.class**. Такие файлы содержат код для выполнения в *виртуальной машине Java*, т.е. *байт-код* интерпретатора *JVM*. Это можно сделать в любом стандартном компиляторе *Java*, в *GCC* можно использовать следующую команду:

```
$ gcj -C Speak.Java
```

Следующий шаг состоит в применении команды `gcjh` для переработки файла `Speak.class` в заголовочный файл интерфейса CNI с именем `Speak.h`. Вот пример команды:

```
$ gcjh Speak
```

Команда `gcjh` способна вырабатывать файлы как в стандарте CNI, так и в стандарте JNI ("Java Native Interface"). По умолчанию применяется CNI, поэтому никаких дополнительных опций в команде применять не требуется. Выводимый по этой команде заголовочный файл имеет имя `Speak.h` и содержит следующее:

```
// DO NOT EDIT THIS FILE - it is machine generated -*- c++ -*-

#ifndef __Speak__
#define __Speak__

#pragma interface

#include <java/lang/Object.h>

extern "Java"
{
    class Speak;
};

class ::Speak : public ::java::lang::Object
{
public: // actually package-private
    Speak ();
public:
    virtual void setString (::java::lang::String *);
    virtual void showString ();
public: // actually package-private
    ::java::lang::String *string;
public:

    static ::java::lang::Class class$;
};

#endif /* __Speak__ */
```

Как видите, заголовочный файл `Speak.h` содержит определение класса `Speak` на языке C++. Поэтому он может непосредственно быть включен в программу на C++ директивой `#include`. Вот пример такой программы:

```
/* cnispeak.cpp */
#include <gcj/cni.h>
#include "Speak.h"

int main(int argc, char *argv)
{
    java::lang::String *str;

    JvCreateJavaVM(NULL);
    JvAttachCurrentThread(NULL,NULL);

    Speak *speak = new Speak();
    speak->setString(JvNewStringLatin1("Hello from CNI to Java"));
```

```
speak->showString();  
JvDetachCurrentThread();  
}
```

Программа в основном такая же, что и рассмотренная ранее `cnistrtout.cpp`. Включаемый заголовочный файл `CNI gcj/cni.h` в свою очередь включает файлы с определениями всех необходимых *Java*-классов. Эти классы могут загружаться и их методы могут выполняться после инициализации среды *виртуальной машины Java* и присоединения к ней текущего процесса. Ключевое слово `new` применено для вызова конструктора класса `Speak` и получения адреса нового объекта `Speak`. Далее вызывается метод `setString()` для сохранения нового объекта `String` в объекте `Speak`. Затем вызывается метод `showString()` для вывода содержащейся в объекте `Speak` строки.

Следующая команда выполнит компиляцию и компоновку программы:

```
$ g++ cnispeak.cpp Speak.class -lgcj -o cnispeak
```

Обработка исключений

Исключения, порождаемые *Java*-объектами, могут перехватываться и обрабатываться программой на языке *C++*. Давайте рассмотрим следующий пример:

```
/* cniexception.cpp */  
#include <gcj/cni.h>  
#include <java/lang/System.h>  
#include <java/io/PrintStream.h>  
#include <java/lang/Exception.h>  
  
using namespace java::lang;  
  
int main(int argc, char *argv)  
{  
    JvCreateJavaVM(NULL);  
    JvAttachCurrentThread(NULL, NULL);  
    try {  
        String *message = JvNewStringLatin1("Hello from CNI");  
        System::out->println(message);  
    } catch(Exception *e) {  
        e->printStackTrace();  
    }  
    JvDetachCurrentThread();  
}
```

Как и другие примеры использования интерфейса *CNI* эта программа также начинается с инициализации среды выполнения *виртуальной машины Java* и заканчивается освобождением в ней процесса, созданного этой программой. Оператор `using namespace` используется для определения пространства имен `java::lang`, что обеспечивает автоматическое разрешение таких имен классов как `String`, `System` и `Exception` без указания их полного имени.

Блоки `try` и `catch` написаны в тех же правилах, которые применяются к классам *Java*. При создании объекта `Exception` в блоке `try` он будет перехвачен опе-

ратором `catch`. Затем будут распечатаны данные трассировки стэка (stack trace), описывающие расположение вызвавших исключение инструкций.

Типы данных интерфейса CNI

В языках *C++* и *Java* применяются в основном сходные типы данных, но все же отличия существуют. Благодаря очень точному определению применяемых в языке *Java* типов возможно использование в коде на языке *C++* команды `typedef` для объявления данных, предназначенных к обработке в *Java*-интерфейсе. Применяемые в CNI базовые типы *Java* перечислены ниже в таблице 10.1.

Таблица 10.1. Основные типы данных, определенные в CNI-интерфейсе

Тип Java	Тип C++	Описание
char	Jchar	16-битное представление буквенного символа в формате Unicode
boolean	Jboolean	Логическая переменная, которая может иметь значение <code>true</code> или <code>false</code>
byte	Jbyte	8-битное целое число со знаком
short	Jshort	16-битное целое число со знаком
int	Jint	32-битное целое число со знаком
long	Jlong	64-битное целое число со знаком
float	Jfloat	32-битное IEEE представление числа с плавающей точкой
double	Jdouble	64-битное IEEE представление числа с плавающей точкой
void	Void	указатель на экземпляр любого типа

Совмещение языков Java и C

Интерфейс **JNI** ("Java Native Interface") применяется для взаимодействия между классами, выполняющимися в *виртуальной машине Java (JVM)*, и исполнимыми системными модулями, написанными на языках *C*, *C++* или ассемблере. Интерфейс разрабатывался для использования в программах на языке *Java* некоторых специфических для тех или иных платформ возможностей, которые, конечно, не могут быть включены в *Java* из-за требований переносимости написанных на этом языке программ. Именно для этих целей его и следует применять. Использование JNI сохраняет переносимость кода *Java*-программ, но может требовать применения такого системно-ориентированного (*native*) кода, который распознает и учитывает особенности конкретных платформ.

Применение системно-ориентированного метода в классе Java

Распространенный способ объединения программ на языках *Java* и *C* заключается в создании *Java*-классов, содержащих методы, реализованные на языке *C*. Впрочем, не только *C*, для этой цели годятся и *C++*, и ассемблер. В нашем примере создается простой *Java*-класс, содержащий всего один метод, реализованный на языке *C*.

Приведенный далее класс с именем `HelloNative` содержит метод `main()`, в свою очередь использующий метод на языке системного уровня для вывода строки буквенных символов. Системно-ориентированный метод объявлен как составляющая часть *Java*-класса, но тело его определения написано на другом языке и потому не приводится в исходном коде программы. Также класс включает в себя статический инициализатор, использующий системный метод `LoadLibrary()` для загрузки динамической (разделяемой) библиотеки. Это и есть та библиотека, которая содержит тело метода, реализованного на системно-ориентированном языке. Дальше — пример программы на языке *Java*:

```
/* HelloNative.java */
public class HelloNative {
    static {
        System.loadLibrary("libspeak.so");
    }
    public static void main(String arg[]) {
        HelloNative hn = new HelloNative();
        hn.sayHello();
    }
    public native void sayHello();
}
```

Для компиляции `HelloNative.java` в файл класса `HelloNative.class` можно использовать следующую команду:

```
$ gcj -C HelloNative.Java
```

Заголовочный файл прототипа функции *C*, которая используется *Java*-классом `HelloNative`, создается из файла `HelloNative.class` командой `gcjh` с опцией `-jni`. Вот пример такой команды:

```
$ gcjh -jni HelloNative
```

Результатом выполнения этой команды будет файл `HelloNative.h` со следующим содержанием:

```
/* DO NOT EDIT THIS FILE – it is machine generated */

#ifndef __HelloNative__
#define __HelloNative__

#include <jni.h>

#ifndef __cplusplus
extern "C"
{
#endif

extern void Java_HelloNative_sayHello (JNIEnv *env, jobject);
#endif /* __HelloNative__ */
```

Имя прототипа функции, объявленного в этом файле, составлено из имени класса `Java` и имени входящего в него метода, который собственно использует эту функцию на языке С. Имена функций в таких генерируемых файлах, как наш `HelloNative.h` всегда начинаются с `Java_`, далее следует полное имя класса, затем символ подчёркивания `_`, и заканчиваются они именем метода. Таким образом, имя объявляемой функции записывается как `Java_HelloNative_sayHello()`.

Прототип этой новой функции объявляется с двумя параметрами, даже если у метода, объявленного на языке `Java`, не было параметров. Два этих параметра необходимы в любой функции, вызываемой через интерфейс JNI. Первый параметр — указатель на интерфейс, используемый для доступа к списку обрабатываемых в теле функции действительных передаваемых методу аргументов. Второй параметр — указатель на вызывающий функцию объект (здесь — на переменную `this` объекта `HelloNative`).

Далее приводится реализация на языке С функции, соответствующей прототипу, объявленному в заголовочном файле `HelloNative.h`:

```
/* HelloNative.c */
#include <jni.h>
#include "HelloNative.h"

void Java_HelloNative_sayHello(JNIEnv *env, jobject this);
{
    printf("A native JNI hello\n");
}
```

В этот код директивами `#include` включаются заголовочные файлы `jni.h` и `HelloNative.h` (последний содержит прототип объявляемой функции). Код реализации функции содержит в точности те же параметры вызова, что и прототип.

По двум следующим командам `gcc` скомпилирует файл `HelloNative.c` в объектный модуль перемещаемого формата и поместит его в разделяемую библиотеку.

```
$ gcc -fPIC -c HelloNative.c -o HelloNative.o
$ gcc -shared HelloNative.o -o libsspeak.so
```

На заключительном этапе следует скопировать библиотеку `libsspeak.so` в расположение, разрешимое для поиска во время выполнения программ используемых ими разделяемых библиотек. Затем можно запустить на выполнение находящуюся в файле `HelloNative.class` главную `Java`-программу нашего примера командой `gij`:

```
$ gij HelloNative
```

Передача аргументов методу, реализованному на языке системного уровня

Так же как и для любого другого `Java`-метода, в программах на языке `Java` возможна также передача аргументов и системно-ориентированному методу. Также возможно и получение возвращаемого им значения. Применяемые для этого типы в интерфейсе `JNI` для языков `C` и `C++` — те же, что определены для интерфейса `CNI`. Они приведены в таблице 10.1.

Следующий пример — *Java*-класс с системно-ориентированным методом, написанным на языке *C*, который получает четыре целочисленных аргумента типа *int* и возвращает их сумму того же типа.

```
/* AddFour.java */
public class AddFour {
    static {
        System.loadLibrary("libaddfour.so");
    }
    public static void main (String arg[]) {
        AddFour af = new AddFour();
        int value = af.sum(1,2,3,4);
        System.out.println("The sum of four is " + value);
    }
    public native int sum(int a,int b,int c,int d);
}
```

Далее следует код реализации на языке *C* системно-ориентированного метода, используемого в приведенной выше *Java*-программе:

```
/* AddFour.c */
#include <jni.h>
#include "AddFour.h"

jint Java_AddFour_sum(JNIEnv *env, jobject this,
                      jint a,jint b,jint c,jint d)
{
    jint total = a + b + c + d;
    return(total);
}
```

Четыре новых параметра добавляются в конец списка аргументов после пары обязательных аргументов. Тип данных *jint* определен во включаемом заголовочном файле *jni.h*. Он соответствует типу *int* языка *C* и в этом примере используется для объявления всех аргументов и возвращаемого значения функции.

Этот пример будет скомпилирован и скомпонован из двух приведенных файлов применением следующей последовательности из четырех команд. Первая создаст класс *AddFour.class*, содержащий главную программу на языке *Java*. Вторая — заголовочный файл *AddFour.h* на языке *C*, этот файл содержит прототип вызываемого программой системно-ориентированного метода. Третья команда скомпилирует содержащуюся в исходнике *AddFour.c* реализацию этого метода, причем при этом будет выработан объектный файл, пригодный для использования в разделяемой (динамической) библиотеке. Последняя команда создает из полученного объектного файла динамическую библиотеку *libaddfour.so*.

```
$ gcj -C AddFour.java
$ gcjh -jni AddFour
$ gcc -fPIC -c AddFour.c -o AddFour.o
$ gcc -shared AddFour.o -o libaddfour.so
```

Теперь остается только скопировать динамическую библиотеку в такое расположение, где она может быть обнаружена загрузчиком, и запустить программу на выполнение следующей командой:

```
$ gij AddFour
```

Обратный вызов метода Java-класса из системно-ориентированного метода на языке С

Также предусмотрена возможность и обратного вызова метода *Java*-класса из кода системно-ориентированного метода. В следующем примере класса *EchoKeystrokes* имеется один системно-ориентированный метод и один *Java*-метод, используемый для обратного вызова. Системно-ориентированный метод *getKeystrokes()* считывает вводимые с клавиатуры символы и передает каждый из них на вывод, выполняя обратный вызов *Java*-метода *characterCallback()*.

```
/* EchoKeystrokes.java */
public class EchoKeystrokes {
    static {
        System.loadLibrary("libgetkeys.so");
    }
    public static void main(String arg[]) {
        EchoKeystrokes ek = new EchoKeystrokes();
        ek.getKeystrokes();
    }
    public native void getKeystrokes();
    public void characterCallback(char character) {
        System.out.println(character);
    }
}
```

Системно-ориентированный метод использует два аргумента, которые автоматически передаются каждому методу JNI для получения информации, необходимой для выполнения обратного вызова. Для получения значения применяющегося в JNI типа *jclass* используется функция *GetObjectClass()*. Этот тип представляет класс объекта, содержащего предназначаемый для обратного вызова метод. Функция *GetMethodID()* возвращает уникальный идентификатор этого метода в классе. Затем необходимый метод может быть неоднократно вызван использованием функции JNI-интерфейса *CallVoidMethod()*.

```
/* getkeystrokes.c */
#include <jni.h>
#include <stdio.h>
#include "EchoKeystrokes.h"

void Java_EchoKeystrokes_getKeystrokes(JNIEnv *env, jobject obj)
{
    jchar character = ' ';
    jclass class_ = (*env)->GetObjectClass(env,obj);
    jmethodID id = (*env)->GetMethodID(env,class_,
        "characterCallback","(C)V";

    if(id != 0) {
        while(character != '.') {
            character = getchar();
            (*env)->CallVoidMethod(env,obj,id,character);
        }
    }
}
```

Вызов функции `GetMethodID()` для однозначного определения метода требует указания его имени, типа возвращаемого значения и списка типов аргументов. Тип возвращаемого значения и типы аргументов задаются строкой буквенных символов следующего формата:

`"(список типов аргументов) тип возвращаемого значения"`

Индикаторы типов, включаемые в эту строку перечислены в таблице 10.2.

Таблица 10.2. Символы, применяемые в JNI для кодировки типов возвращаемых значений и аргументов методов для их обратного вызова

Индикатор	Тип данных языка Java
<code>Z</code>	<code>Boolean</code>
<code>B</code>	<code>Byte</code>
<code>C</code>	<code>Char</code>
<code>S</code>	<code>Short</code>
<code>I</code>	<code>Int</code>
<code>J</code>	<code>long</code>
<code>F</code>	<code>float</code>
<code>D</code>	<code>double</code>
<code>V</code>	<code>void</code>
<code>Lclassname;</code>	Объект класса, указанного именем <code>classname</code>
<code>[type</code>	Массив значений указанного типа <code>type</code>
<code>(arg type list) return type</code>	Метод с указанным в <code>arg type list</code> списком типов аргументов и возвращаемым значением типа <code>return type</code>

Например, если методу передается один аргумент типа `int` и один типа `double` и этот метод возвращает значение типа `double`, то определяющая строка будет иметь такой вид:

`"(ID)D"`

Еще один пример: первый аргумент — массив байтов, второй аргумент — строка, тип возвращаемого методом значения — `void`:

`"([BLjava/lang/String;)V"`

Приведенная далее последовательность команд выполнит следующие действия. Вначале команда `gcj` скомпилирует исходный файл `EchoKeystrokes.java` в файл `EchoKeystrokes.class`, содержащий Java-код главной программы примера. Затем утилита `gcjh` считает полученный файл и создаст файл `EchoKeystrokes.h`, содержащий прототип (на языке C) системно-ориентированного метода. Далее будет выполнена компиляция исходного файла `getkeystrokes.c`, содержащего реализацию этого метода на языке C, в объектный файл с перемещаемой адресацией кода (position independent code) `getkeystrokes.o`. И в заключительной команде из объектного файла будет создана разделяемая (динамическая) библиотека `libgetkeys.so`.

```
$ gcj -C EchoKeystrokes.java
$ gcjh -jni EchoKeystrokes
$ gcc -fpic -c getkeystrokes.c -o getkeystrokes.o
$ gcc -shared getkeystrokes.o -o libgetkeys.so
```

Совмещение языков Fortran и C

В GNU языки *Fortran* и *C* могут использоваться вместе довольно естественно благодаря тому, что между этими языками возможны прямые вызовы функций. Нужно только тщательно учитывать совместимость типов передаваемых аргументов. И тогда написанные на этих языках функции могут использоваться в прямых и в обратных вызовах в точности так же, как если бы они были написаны на одном языке.

В таблице 10.3 перечислены типы *C* и их соответствия в языке *Fortran*. Эта таблица применима на многих платформах, но возможны и некоторые исключения. Было бы благоразумно создать небольшую тестовую программу для проверки правильности передачи функциям тех типов, которые вас интересуют. Для этого вполне можно использовать примеры этого раздела.

Таблица 10.3. Совместимые типы данных языков C и Fortran

Тип языка C	Тип языка Fortran	Описание
signed char	INTEGER*1	8-битное целое число со знаком
short	INTEGER*2	16-битное целое число со знаком
int	INTEGER	32-битное целое число со знаком
float	REAL	32-битное число с плавающей точкой
double	DOUBLE PRECISION	64-битное число с плавающей точкой
SUBROUTINE SUB()	void sub_()	Функция C, не имеющая возвращаемого значения, (void function), эквивалент подпрограммы в языке Fortran
REAL FUNCTION FUN()	float fun_()	Возвращающая значение функция C эквивалентна функции языка Fortran

В языке *Fortran* аргументы всегда передаются ссылками на них, в языке *C* массивы всегда передаются через их адрес. Это делает возможной прямую передачу массивов между частями программы, написанными на разных языках. Но следует учесть, что для этих языков отличается порядок индексации многомерных массивов. В *Fortran* массивы организованы по столбцам, в то время как в *C* элементы массивов располагаются построчно. Поэтому при передаче массивов требуется перестановка индексов.

Вызов функции C из кода на языке Fortran

Предлагаемый пример программы на языке *Fortran* вызывает функцию *C* и передает ей строку буквенных знаков и число с плавающей точкой:

```
C f772c.f
C
PROGRAM F772C
```

```
C
CHARACTER*32 HELLO
REAL PI
C
HELLO = "Hello C from Fortran"
HELLO(21:21) = CHAR(0)
PI = 3.14159
CALL SHOWHIPI(HELLO,PI)
END PROGRAM F772C
```

Переменная `HELLO` типа `CHARACTER` имеет фиксированную длину в 32 буквенных символа. В нее записывается строка, содержащая только 21 символ, после чего оставшаяся часть будет заполнена пробелами. Для преобразования этой строки к такой форме, которая может быть использована в языке `C`, следует вставить нулевой байт (символ конца строки) после последнего байта, занимаемого записанным в `HELLO` значением. Переменная `PI` типа `REAL` имеет тот же формат, что и тип данных `float` языка `C`, она может прямо передаваться функции `C`.

Важно заметить, что аргументы передаются из кода на языке *Fortran* по ссылке, а не своим значением. Далее приведена функция на языке `C`, которая выводит на стандартное устройство вывода строку и число, переданные ей из программы на языке *Fortran*:

```
/* showhipi.c */
#include <stdio.h>
void showhipi_(char *string, float *pi)
{
    printf("%s\nPI=%f\n", string, *pi);
}
```

От одной платформы к другой могут несколько изменяться соглашения об именах и совместимость типов. Детальнее об этом можно прочитать в документации GCC. Как видно из примера, в рассматриваемой автором реализации компилятора к имени вызываемой из *Fortran* функции `C` требуется добавлять символ подчеркивания.

Следующие команды скомпилируют оба исходных файла и создадут из них готовую к запуску программу:

```
$ g77 -c f772c.f -o f772c.o
$ gcc -c showhipi.c -o showhipi.o
$ g77 c2f77.o showhipi.o -o f772c
```

Вызов из программы на языке C подпрограммы на языке Fortran

При вызове подпрограммы на языке *Fortran* из программы на `C` следует форматировать передаваемые строки в соответствии с правилами языка *Fortran*. Следующий пример программы передает строку буквенных символов и число с плавающей точкой:

```

/* c2f77.c */
int main(int argc,char *argv[])
{
    int i;
    float e = 2.71828;
    char hello[32];
    int length = sizeof(hello);
    strcpy(hello,"Hello Fortran from C");
    for(i=strlen(hello); i<length; i++)
        hello[i] = ' ';
    showhie_(hello,&length,&e);
    return(0);
}

```

В программах на языке *C* длина строк из буквенных символов определяется положением нулевого байта — признака конца строки, а в языке *Fortran* строки имеют фиксированную длину. Поэтому необходимо включать в список аргументов число, указывающее действительную длину передаваемой строки. В этом примере строка записывается в массив `char`, который первоначально заполнен нулевыми значениями. Поэтому остальная часть массива `hello`, состоящего из 32 элементов, заполняется пробелами. Размер массива передается во втором аргументе. Учтите, что в *Fortran* все передаваемые переменные ожидаются как ссылки на их значения, поэтому все аргументы в этом примере передаются указателями. При вызове подпрограммы на языке *Fortran* обычно требуется добавлять к ее имени символ подчеркивания.

Далее приводится исходный код вызываемой подпрограммы (на языке *Fortran*):

```

C showhie.f
C
SUBROUTINE SHOWHIE(HELLO,LENGTH,E)
CHARACTER*(*) HELLO
INTEGER LENGTH
REAL E
C
WRITE(*,100) HELLO(1:LENGTH),LENGTH,E
100 FORMAT(3X,A,2X,I3,4X,F6.4)
RETURN
END SUBROUTINE SHOWHIE

```

Следующая далее последовательность команд скомпилирует исходники в объектные файлы и затем скомпонует из них готовую к запуску программу `c2f77`:

```

$ g77 -c showhie.f -o showhie.o
$ gcc -c c2f77.c -o c2f77.o
$ gcc c2f77.o showhie.o -lfprtbegin -lg2c -lm -o c2f77

```

Третья команда `gcc` требует наличия библиотек языка *Fortran*. При применении команды `g77` необходимые библиотеки подключаются автоматически. Так что последнюю команду можно записать короче:

```
$ g77 c2f77.o showhie.o -o c2f77
```

Совмещение языков Ada и C

Язык *Ada* включает в себя встроенные возможности вызова функций на языках *C* и *Fortran*. Это делается применением при объявлении тела процедуры оператора **pragma Import**, указывающего внешний язык и имя исходного файла функции.

Типы данных, используемые в языках *Ada* и *C* вполне совместимы друг с другом. На это вполне можно рассчитывать, особенно когда объектный код для всех частей программы генерируется компилятором *GCC*. В таблице 10.4 перечислены идентичные типы для обоих языков.

Таблица 10.4. Соответствия типов языков *Ada* и *C*

Тип данных языка Ada	Тип данных языка C
Float	float
Integer	int
Long_Float	double
Long_Integer	long
Long_Long_Integer	long long
Short_Float	float
Short_Integer	short
Short_Short_Integer	signed char

Вызов кода C из программы на языке Ada

Приведенный далее пример показывает, как тело процедуры в пакете *Ada* может быть реализовано на языке *C*. Следующий листинг представляет главную программу,зывающую процедуры *hello* и *goodbye* из пакета *Ada Howdy*:

```
-- ada2c.adb
with Howdy;
procedure Ada2C is
begin
    Howdy.hello;
    Howdy.goodbye;
end Ada2C;
```

Обе процедуры *hello* и *goodbye* выполняют вывод строки текста, их отличие состоит в том, что *goodbye* написана на *Ada*, а *hello* — на языке *C*. Участники пакета *Howdy* определены в файле *howdy.ads* следующим образом:

```
-- howdy.ads
package Howdy is
    procedure Hello;
    procedure Goodbye;
end Howdy;
```

Далее следует код реализации процедур, включенных в пакет. Файл *howdy.adb* содержит действительный *Ada*-код процедуры *goodbye* и объявляет внешнюю реализацию процедуры *hello*.

```
-- howdy.adb
with Text_IO; use Text_IO;
with Interfaces.C;
package body Howdy is
    procedure Hello is
        procedure sayhello;
        pragma Import(C,sayhello);
    begin
        sayhello;
    end Hello;
    procedure Goodbye is
    begin
        Put_Line("Goodbye");
    end Goodbye;
end Howdy;
```

Оператор `with Interfaces.C` используется для включения совместимости применяемых типов данных с языком *C*. Он не является здесь строго необходимым, поскольку вызов функции *C* не использует аргументов и не возвращает результат. Процедура `Hello` вызывает `sayhello` — функцию на языке *C*. Для этого требуется использование операторов `procedure` и `pragma Import`. Они указывают, что `sayhello` является внешней функцией на языке *C*.

Первым аргументом оператора `pragma Import` является название языка, на котором реализован код тела процедуры. Стандарт *Ada* определяет в качестве известных компилятору языков *C* ("C"), *C++* ("C++"), *Fortran* ("Fortran") и *COBOL* ("COBOL"). Второй аргумент определяет локальное имя, под которым внешняя функция используется в текущей программе. В случае, когда действительное имя внешней функции неприемлемо для использования в программе на языке *Ada*, то это внешнее имя указывается в третьем аргументе. Например, вам нужно удаленно вызвать функцию `_stprob()`, но в программах на языке *Ada* имена не должны начинаться знаком подчеркивания. Тогда вы определяете прагму `Import` так, чтобы внутреннее имя `stprob` указывало на внешнюю функцию `_stprob`:

```
pragma Import (C,stprob,"_stprob")
```

В примере нашей программы функция на языке *C* вызывается очень просто, и ее реализация выглядит так:

```
/* sayhello.c */
#include <stdio.h>
void sayhello()
{
    printf("Hello C from Ada\n");
}
```

Дальше приводится последовательность команд для компиляции всех исходных файлов на языках *Ada* и *C* и для компоновки готовой программы:

```
$ gcc -c sayhello.c -o sayhello.o
$ gcc -c howdy.adb
$ gcc -c ada2c.adb
$ gnatbind ada2c.ali
$ gnatlink ada2c.ali sayhello.o
```

Вызов с аргументами функции C из программы на языке Ada

Этот пример программы очень похож на предыдущий, только функции, реализованные на языке C, получают аргументы и возвращают значения. В примере используются системные вызовы среды UNIX для порождения и остановки процесса, выполняемого в фоновом режиме, (background process). Вот главная программа на языке Ada, записанная в файле `adaspawn.adb`:

```
-- adaspawn.adb
with Spawn;
procedure AdaSpawn is
pid : Integer;
status : Integer;
begin
    pid := Spawn.startProcess("flex");
    status := Spawn.stopProcess(pid);
end AdaSpawn;
```

В главной процедуре выполняется вызов функции `startProcess()`, ей передается имя программы для выполнения. Функция возвращает идентификатор запущенной фоновой задачи, который затем используется для остановки этой же задачи функцией `stopProcess()`. Обе функции определены как участники пакета `Spawn` в файле `spawn.ads`:

```
-- spawn.ads
package Spawn is
    function startProcess(name : String) return Integer;
    function stopProcess(pid : Integer) return Integer;
end Spawn;
```

Все передаваемые функциям параметры имеют типы языка Ada. Внутри этих Ada-функций выполняются вызовы функций на языке C. В связи с этим нужно некоторое преобразование данных для гарантии совместимости аргументов. Тела функций `startProcess()` и `stopProcess()` определены в файле `spawn.adb`:

```
-- spawn.adb
with Interfaces.C;
package body Spawn is
    function startProcess(name : String) return Integer is
        function start (name : String) return Interfaces.C.int;
        pragma Import(C,start);
    begin
        return Integer(start(name));
    end startProcess;
    function stopProcess(pid : Integer) return Integer is
        function stop(pid : Integer) return Interfaces.C.int;
        pragma Import(C,stop);
    begin
        return Integer(stop(pid));
    end stopProcess;
end Spawn;
```

Функции `startProcess()` и `stopProcess()` действуют как оболочки написанных на языке C функций `start()` и `stop()`. При этом имеет место некоторое преобразование данных. Обе функции C возвращают значения типа `Interface.C.int`, которые затем приводятся к типу `Integer` языка Ada для их последующего возвращения функциями `startProcess()` и `stopProcess()`.

Вот код C-функций `start()` и `stop()`, который находится в файле `startstop.c`:

```
#include <unistd.h>
#include <signal.h>
#include <errno.h>

int start(char *name)
{
    int pid;
    char *argv[4];
    pid = fork();
    if(pid == -1)
        return(-1);
    if(pid == 0) {
        argv[0] = "sh";
        argv[1] = "-c";
        argv[2] = name;
        argv[3] = 0;
        execve("/bin/sh", argv, 0);
        exit(-1);
    } else {
        return(pid);
    }
}
int stop(int pid)
{
    if(kill(pid,SIGTERM) < 0)
        return(errno);
    return(0);
}
```

Функция `start()` выполняет системный вызов `fork()`, клонирующий текущий процесс. Возвращаемое им значение сообщает, является ли текущий процесс оригинальным процессом или клоном. Клонированный процесс далее преобразует себя в отдельный процесс с помощью системного вызова `execve()`. Вызов `execve()` не возвращает управление, потому что он немедленно заменяет текущий процесс новым, что принуждает системную оболочку выполнять код сначала. Так что возвратить идентификатор нового запущенного процесса (PID) может только оригинальная программа.

Такое построение обеспечивает оболочку из функций на языке Ada над функциями C. При таких условиях последние вполне чисто выполняют системные вызовы. Впрочем, в нашем примере это построение использовано только для наглядности представления. Абсолютной необходимости в применении такого подхода здесь нет. На деле ничего не мешает вам выполнять из Ada прямые обращения к `execve()`, `kill()` или любые другие системные вызовы тем же способом, какой был применен для `startProcess()` и `stopProcess()`.



Глава 11

Интернационализация (Internationalization)

Любая программа, будучи правильно написанной, (и даже сам компилятор GCC) может запускаться так, что ее интерфейс приспосабливается к местным условиям и языку.

Интернационализацией (internationalization) называется внутренняя способность программы или набора программ, составляющих отдельный пакет, поддерживать несколько языков.

Локализацией (localization) называются действия по использованию встроенных возможностей программы или набора программ, предпринимаемые для преобразования выводимых пользователю текстов к другому национальному языку или стандарту. Такие действия также называются изменениями локальных установок в системных файлах, из которых оничитываются и выполняются программами при их запуске.

NLS ("Native Language Support", поддержка родного языка) — термин, используемый для описания всех операций, необходимых для интернационализации и локализации программ.

Часто можно видеть, что в документации вместо термина "интернационализация" ("internationalization") применяется сочетание `i18n`. Это весьма употребительный и традиционно остроумный способ сокращения английского термина. Это сокращение означает, что обозначаемое им длинное английское слово начинается с литеры "`i`", состоит из 18-ти букв и заканчивается литерой "`n`". Точно таким же способом и термин "локализация" часто заменяется сокращением `l10n`.

Можно заметить, что программисты чаще употребляют термин `i18n`, в то время как переводчики и пользователи обычно используют `l10n`.

Для примеров и разъяснений в этой главе используется язык программирования C. Те же действия могут быть использованы и для программ на других алгоритмических языках — C++, Objective-C, Python, Lisp, EmacsLisp, Jawa и awk.

Пример программы для применения перевода

Следующий пример программы содержит нуждающиеся в переводе строки:

```
/* starter.c */
#include <locale.h>
#include <libintl.h>

#define PACKAGE "starter"
#define LOCALEDIR "/usr/share/locale"

int main(int argc,char *argv[])
{
    setlocale(LC_ALL,"");
    bindtextdomain(PACKAGE,LOCALEDIR);
    textdomain(PACKAGE);

    printf("%s\n",gettext("This string will translate."));
}
```

Заголовочный файл `locale.h` содержит некоторые из основных макроопределений, которые используются для указания подвергаемых локализации типов и задействованных в конвертировании денег структур данных. Другой заголовочный файл `libintl.h` содержит прототипы функций, необходимых для настройки и применения действий интернационализации.

В функции `main()` выполняется вызов `setlocale()` для указания элементов, к которым применяется интернационализация. Указание `LC_ALL` означает применение интернационализации ко всем данным. Однако может понадобиться применения ее только к некоторым элементам программы. Вместо одного вызова `setlocale()` с указанием `LC_ALL` возможно выполнение нескольких вызовов этой функции с указанием отдельных элементов, перечисленных в таблице 11.1. Функция `setlocale()` возвращает строку, соответствующую текущей установке локализации.

Таблица 11.1. Категории установок, применяемые в функции `setlocale()`

Установка	Описание
<code>LC_ADDRESS</code>	Расположение стандартных частей адреса, включая название фирмы, здания, отдела, адрес головной конторы, номер дома, почтовый индекс, страну назначения и т.д.
<code>LC_ALL</code>	То же, что применение всех остальных установок.
<code>LC_COLLATE</code>	Соответствие регулярных выражений. Определяет значение и диапазон буквенных знаков, применяемых в выражениях.
<code>LC_CTYPE</code>	Соответствие регулярных выражений. Определяет применяемую классификацию буквенных знаков, правила преобразования, учет их регистра при сравнении и широкий диапазон функций обработки буквенных знаков.

Установка	Описание
LC_IDENTIFICATION	Форматирование такой информации, как имя, адрес, телефон, адрес электронной почты, номер факса и т.д.
LC_MEASUREMENT	Применяет локализацию в отношении единиц измерения (метрическая либо английская системы измерений).
LC_MESSAGES	Применяет локализацию в отношении формализованных текстовых сообщений.
LC_MONETARY	Формат представления денежных единиц.
LC_NAME	Применяет тот или иной формат представления личных имен, включающих инициалы, обращение, сокращенное обращение и положение имени и фамилии.
LC_NUMERIC	Формат представления чисел, включающих десятичный разделитель и выделение порядков (тысяч).
LC_PAPER	Стандартный размер бумаги для печати.
LC_TELEPHONE	Формат представления телефонных номеров, включающих префиксы и коды направления.
LC_TIME	Формат представления в строках времени и даты.

В нашем примере две директивы `#define` внутри самой программы указывают имя пакета, к которому относится программа, и полное имя каталога, содержащего локальные установки. Хотя более обычным для этого способом является применение настроек в файле `config.h` или использование опции `-D` в командах, генерируемых компоновочным файлом (`makefile`).

Для перевода текстовой строки с одного языка на другой, выполняется вызов функции `gettext()`. На самом деле существует целое семейство функций `gettext()`, это поясняется в следующем разделе. Эти функции запускают утилиту `xgettext` для извлечения строки. Страна-оригинал (такая строка показана в примере листинга программы) используется как ключ для поиска соответствующего ей перевода с учетом действующих установок локализации. Если соответствий ей не найдено, то используется сама строка-оригинал. Значение, возвращаемое функцией `gettext()`, имеет тип символьной строки. Таким образом, в нашем примере оператор `printf()` просто выводит строку, возвращаемую функцией `gettext()` независимо от того, была она переведена или нет.

При программировании и для преобразования существующих программ удобно использовать короткое макроопределение, заменяющее употребление имени `gettext()`. Например, вызов функции можно сократить до знака подчеркивания, применив следующее макроопределение:

```
#define _(a) gettext(a)
```

При использовании этого макроопределения оператор `printf()` в нашем примере можно записать следующим образом:

```
printf("%s\n",_("This string will translate."));
```

Такой способ сокращает количество символов, которые следует добавить для вызова выполняющей перевод функции, до трех (одного символа подчеркивания и двух скобок).

Создание нового файла ".po"

Если все нуждающиеся в переводе строки в тексте программы включены в вызовы функции `gettext()`, то тогда следует построить файл, содержащий эти строки и ключевые слова, а также все необходимые им соответствия. Такой файл должен иметь формат, поддерживающий перевод необходимых строк в соответствии с каждой предназначаемой локальной установкой. Это дело начинается с применения утилиты `xgettext` для извлечения из исходных файлов текстовых строк в новый файл `.po` и последующего их оформления. По следующей команде из указанного программе `xgettext` файла `starter.c` будут выбраны все предназначенные для передачи функции `gettext()` строки и помещены файл с именем `messages.po`:

```
$ xgettext starter.c
```

В файле `messages.po` после некоторого заголовка будут находиться такие строки:

```
msgid "This string will translate."
msgstr ""
```

Для завершения оформления этого файла необходимо, используя текстовый редактор, вместо пустой строки справа от ярлыка (тэга) `msgstr` ввести нужный перевод сообщения. Если исходник программы содержит несколько таких строк, то они все будут находиться в том же файле.

Утилита `xgettext` может использоваться с несколькими языками программирования. Она способна собирать строки из всех передаваемых ей файлов в один файл `.po`, который может использоваться целым программным пакетом. Поддерживаемые программой `xgettext` опции командной строки перечислены в таблице 11.2.

Таблица 11.2. Опции командной строки, распознаваемые программой `xgettext`

Опция	Описание
-	Вместо считывания исходного кода из файла программа получает его со стандартного устройства ввода.
-a	По этой опции из исходников на языках C или C++ извлекаются все строковые константы.
--add-comments=tag	То же, что -c.
--add-location	То же, что -n.
-C	Сокращенная форма опции --language=C.
-c tag	Используется для размещения в выводимом файле блока комментария с указанным ярлыком (<code>tag</code>).
--c++	Сокращенная форма опции --language=C++.
--copyright-holder=str	В <code>str</code> передается строка для сообщения об охране авторских прав программного пакета. Эти права соответственно распространяются и на строки сообщений, извлекаемые из исходных текстов программ, входящих в этот пакет. Если эта опция не назначается, то по умолчанию строка имеет значение "Free Software Foundation" ("Ассоциация свободно распространяемых программ").
-d name	Выводимый файл будут иметь имя <code>name.po</code> (вместо применяемого по умолчанию <code>messages.po</code>). См. также -o.

Опция	Описание
-D <i>directory</i>	Добавляет указанный в <i>directory</i> каталог к списку каталогов, которые должны просматриваться при поиске указанных в команде исходных файлов.
--default-domain= <i>name</i>	То же, что -d .
--directory= <i>directory</i>	То же, что -D .
--exclude-file= <i>file</i>	То же, что -x .
--extract-all	То же, что -a .
-F	Упорядочивает выводимые строки в соответствии с расположением исходных файлов.
-f <i>file</i>	При указании этой опции имена исходных файловчитываются из указанного файла <i>file</i> , а не из командной строки.
--files-from= <i>file</i>	То же, что -f .
--force-po	По этой опции выводимый файл создается даже в том случае, когда не найдено ни одной такой строки, к которой может быть применен перевод.
--foreign-user	Отменяет вывод, определяемый опцией --copyright-holder , в том числе и действующий по умолчанию.
-h	По этой опции xgettext показывает этот список опций и затем завершает работу.
--help	То же, что -h .
-i	При записи файла .po используются абзацы.
--indent	То же, что -i .
-j	Опция указывает объединять сообщения с уже существующими в перезаписываемом выходном файле.
--join-existing	То же, что -j .
-k <i>keywordspec</i>	При обработке входных файлов на языках C и C++ <i>keywordspec</i> определяет дополнительные ключевые слова для извлечения строк. Поле <i>keywordspec</i> имеет формат <i>named</i> : <i>num</i> , где <i>num</i> — порядковый номер аргумента функции, указанной в <i>named</i> . Применимыми по умолчанию сочетаниями являются gettext , gettext:2 , dgettext:2 , ngettext:1 , dngettext:2,3 , dnggettext и gettext_noop . При указании <i>keywordspec</i> умолчания не действуют.
-keyword= <i>keywordspec</i>	То же, что -k .
-L <i>name</i>	Название языка программирования входных файлов. Параметр <i>name</i> может иметь такие значения: C, C++, ObjectiveC, PO, Python, Lisp, EmacsLisp, librep, Java, awk, YCP, Tcl, RST или Glade.
--language= <i>name</i>	То же, что -L .
-m [<i>string</i>]	Использует указанную строку <i>string</i> как приставку (префикс) для всех вхождений msgstr в выводимом файле. См. также -M .
-M [<i>string</i>]	Использует указанную строку как окончание (суффикс) для всех вхождений msgstr в выводимом файле. См. также -m .
--msgstr-prefix[= <i>string</i>]	То же, что -m .
--msgstr-suffix[= <i>string</i>]	То же, что -M .
-n	Включает в выводимый файл комментарии, показывающие имена исходных файлов, из которых взяты строки. Применяется по умолчанию.

Опция	Описание
--no-location	Опция указывает не включать в выводимый файл комментарии, показывающие имена исходных файлов, из которых взяты строки.
--no-wrap	Опция указывает не разбивать длинные строки в выводимом файле.
-o <i>file</i>	Имя для выходного файла (вместо применяемого по умолчанию <i>messages.po</i>). См. также -d.
--omit-header	Отменяет включение в файл заголовка, обычно имеющего формат <i>msgid " "</i> .
--output-dir= <i>directory</i>	То же, что -p.
--output-file= <i>file</i>	То же, что -o.
-p <i>directory</i>	Каталог, в котором будет помещен выводимый файл.
-s	Эта опция указывает применять сортировку строк вместо их обычного расположения, когда строки помещаются в файл в том порядке, в каком они находятся в исходных файлах.
--sort-by-file	То же, что -F.
--sort-output	То же, что -s.
--strict	Указывает, что файл <i>.po</i> должен записываться в строгом с форматом Uniforum. Такой формат не поддерживается расширениями GNU.
-T	Включить распознавание многоточий в исходниках на языке C.
--trigraphs	То же, что -T.
-v	По этой опции <i>xgettext</i> выводит сообщение о своей версии и завершает работу.
--version	То же, что -v.
-w <i>number</i>	Назначает ширину страницы выводимого файла. Строки, превышающие допустимую длину, будут разбиваться.
--width= <i>number</i>	То же, что -w.
-x <i>file</i>	Указывает не извлекать вхождения из указанного <i>.po</i> или <i>.pot</i> файла.

Одной из наиболее важных является опция **-j**. При ее применении из старой версии файла *.po* в новую переносятся строки и их уже выполненный перевод. Это позволяет обновлять файлы сообщений без потери уже сделанной работы. Например, по следующей команде будет считан файл *starter.po*, и к существующей в нем информации будут дописаны только новые, не встречавшиеся ранее строки подлежащих переводу сообщений:

```
$ xgettext -j -d starter starter.c
```

Использование функций gettext()

Помещение подлежащих переводу строк в аргумент вызова функции **gettext()** — простейший способ разметки исходного текста программы для ее интернационализации. Бывают и такие ситуации, когда приходится применять несколько иной подход. Для решения ряда определенных проблем возможно применение еще нескольких функций.

Статические строки

Далее предлагается пример, который показывает возможность динамического перевода во время выполнения программы строки, объявленной как начальное значение глобальной переменной:

```
/* statictrans.c */
#include <locale.h>
#include <libintl.h>

#define PACKAGE "starter"
#define LOCALEDIR "/usr/share/locale"

#define gettext_noop(a) (a)

char *lbl = gettext_noop("This is a global static string.");

int main(int argc,char *argv[])
{
    setlocale(LC_ALL,"");
    bindtextdomain(PACKAGE,LOCALEDIR);
    textdomain(PACKAGE);

    printf("%s\n",gettext(lbl));
}
```

Функция `gettext_noop()` — макроопределение, передающее строку без каких-либо действий над ней. При определении строки для перевода, `xgettext` увидела бы только имя этой функции-боловки и пропустила бы такую строку. Вследствии вызова `gettext()` передается адрес нужной строки и при этом перевод происходит в том месте программы, где эта строка используется. То есть перевод происходит так же, как происходит перевод строковой константы — аргумента функции `xgettext()`. Если глобальная строка используется в нескольких местах программы, то она будет переводиться при каждой ссылке на нее.

Перевод строки, имеющейся в другом домене

Если вам нужно применить перевод какой-нибудь строки из другого домена, то следует применить функцию `dgettext()` и при вызове указать ей название другого домена. Например, в домене с именем `hrdomain` уже есть перевод для ключа `"Daily average catch"`. Тогда вам нужно применить функцию `dgettext()` так:

```
dgettext ("hrdomain","Daily average catch ");
```

Простое выполнение `xgettext()` здесь не годится, потому что для перевода требуется указать другое расположение.

Перевод из другого домена в указанной категории

Функция `dcgettext()`, как и `dgettext()`, тоже делает возможным получение перевода строки из другого домена, кроме того, она позволяет выбрать категорию для перевода.

В качестве категории указывается одна из констант локализационных установок, перечисленных в таблице 11.1. Например, можно применить такой вызов для перевода календарной даты в соответствии с установками интернационализации, действующими в домене, имеющем название `hrdomain`:

```
dcgettext("hrdomain", "12/04/03", LC_TIME)
```

Множественное число

Метод `ngettext()` применяет формы множественного числа при выборе строки для перевода. Функции передаются формы единственного и множественного числа слова вместе со степенью множественности. В некоторых языках слова имеют форму единственного числа для обозначения одного предмета и отдельные формы множественного числа для двух предметов и для их количества, равного или превышающего три. Следующий пример может быть применен для перевода слова "image" по отношению к количеству обозначаемых предметов, равном двум:

```
ngettext("picture", "pictures", 2L);
```

В этом примере процесс автоматического перевода требует правильного выбора точной формы множественного числа пред назначаемого языка для обозначения двух предметов.

Формы множественного числа из другого домена

Функция `dnggettext()` действует так же, как и `ngettext()`, только она выполняет поиск перевода в другом домене. Пример вызова функции будет искать в домене `hrdomain` подходящую форму множественного числа перевода подписи для пары изображений:

```
dnggettext("hrdomain", "picture", "pictures", 2L);
```

Формы множественного числа из другого домена в указанной категории

Функция `dcngettext()` работает также как описанная выше `dnggettext()`, за исключением того, что она ищет перевод, соответствующий определению назначенной категории локализации. Категории те же, что перечислены в таблице 11.1. Следующий пример выполняет поиск в домене `hrdomain` перевода формы множественного числа для обозначения двух изображений:

```
dcngettext("hrdomain", "Mr. Garcia", "Messrs. Garcia", 2L, LC_NAME);
```

В этом примере подходящий перевод для обозначения двух человек по фамилии "Garcia" будет выбран в соответствии с правилами, применяемыми для форматирования имён.

Объединение двух файлов

Как вам уже известно, существует возможность использования утилиты `xgettext` для генерирования новых таблиц для перевода, которые автоматически объединяются с уже существующими файлами `.po`. Однако возможны ситуации, когда предпочтительнее использование двух отдельных файлов `.po` — один файл для предыдущей версии программы, и другой файл, который содержит выбранные вхождения переводимых строк из новой версии. Именно для таких случаев предусмотрена утилита объединения файлов `msmerge`:

```
$ msmerge oldfile.po newfile.po
```

В этом примере из файла `oldfile.po` будут выбраны и перенесены в новый файл `newfile.po` переводы для одинаковых ключевых строк. Остальные строки, существующие в `newfile.po`, то есть такие, каких нет в `oldfile.po`, будут перенесены без изменений. Кроме того, все новые данные, записываемые в выходной файл, выводятся также и на стандартное устройство вывода. Опции утилиты `msmerge` перечислены в таблице 11.3.

Таблица 11.3. Опции командной строки утилиты `msmerge`

Опция	Описание
<code>--add-location</code>	Включает в выводимый файл строки комментария, указывающие на положение каждой ключевой строки в исходном файле. Включено по умолчанию. См. также <code>--no-location</code> .
<code>-D directory</code>	Названный в <code>directory</code> каталог добавляется к списку расположений для поиска указанных в команде файлов.
<code>--directory=directory</code>	То же, что <code>-D</code> .
<code>-e</code>	Указывает не использовать escape-последовательности языка C в выводимых текстовых строках.
<code>-E</code>	Включает использование escape-последовательностей языка C в выводимых текстовых строках.
<code>--escape</code>	То же, что <code>-E</code> .
<code>--force-po</code>	Создает выходной файл, даже если он пуст.
<code>-h</code>	Выводит этот список опций и завершает работу программы.
<code>--help</code>	То же, что <code>-h</code> .
<code>-i</code>	Применяет абзацы для форматирования вывода.
<code>--indent</code>	То же, что <code>-i</code> .
<code>--no-location</code>	Подавляет комментарии, указывающие на положение каждой ключевой строки в исходном файле. См. также <code>--add-location</code> .
<code>-o file</code>	Параметр <code>file</code> назначает имя для выходного файла. По умолчанию вывод идет только на стандартное устройство вывода.
<code>--output-file=file</code>	То же, что <code>-o</code> .
<code>--strict</code>	Вырабатывает вывод в строгом соответствии с форматом Uniforum. Этот стиль расширениями GNU не поддерживается.
<code>-v</code>	Вырабатывает более информативный вывод, описывающий действия программы.

Опция	Описание
<code>-v</code>	Показывает номер версии и завершает работу программы.
<code>--verbose</code>	То же, что <code>-v</code> .
<code>--version</code>	То же, что <code>-v</code> .
<code>-w number</code>	В поле <code>number</code> назначается наибольшая ширина страницы. Строки длиннее, чем <code>number</code> разбиваются на несколько строк.
<code>-width=number</code>	То же, что <code>-w</code> .

Создание двоичного файла ".mo" из файла ".po"

Когда перевод выводимых программой сообщений добавлен в файл `.po`, то следующим шагом будет создание двоичного файла типа `.mo`. Этот тип файлов используется программами для выполнения перевода. Двоичный файл вырабатывается из файла `.po` утилитой `msgfmt`, например, такой командой:

```
$ msgfmt starter.po
```

Эта команда создает двоичный файл с именем `starter`. Давайте вернемся к примеру программы `starter.c`, приведенной в начале главы. Она начинается тремя вызовами функций:

```
setlocale(LC_ALL, "");  
bindtextdomain(PACKAGE, LOCALEDIR);  
textdomain(PACKAGE);
```

Макрос `PACKAGE` определен как `"starter"`, и `LOCALEDIR` — как `"usr/share/locale"`. Чтобы программа могла найти таблицы перевода для, скажем, канадского диалекта английского языка, необходимо только скопировать двоичный файл в каталог `usr/share/locale/en_CA/starter`. Если текущий язык установлен в `en_CA`, то программа обнаружит там необходимые ей таблицы перевода сообщений. Для создания переводов сообщений на другие языки, нужно будет отредактировать файл `.po`, сделать из него другой двоичный файл `.mo`, и затем скопировать новый файл `.mo` в соответствующий каталог.

Утилита `msgfmt` поддерживает опции командной строки, перечисленные в таблице 11.4.

Таблица 11.4. Опции командной строки утилиты `msgfmt`

Опция	Описание
<code>-a number</code>	Выравнивает строки к границе указанного в поле <code>number</code> количества байт. По умолчанию равно 1.
<code>--alignment=number</code>	То же, что <code>-a</code> .
<code>-c</code>	Выполняет проверки строк, зависимых от исходного текста программы. Это включает в себя проверку соответствия форматирующих последовательностей с символом "%", применяемых в языке C, а также проверку корректности заголовка файла.

Опция	Описание
<code>--check</code>	То же, что <code>-c</code> .
<code>-D directory</code>	Названный в <code>directory</code> каталог добавляется к списку путей для поиска указанных в команде файлов.
<code>--directory=directory</code>	То же, что <code>-D</code> .
<code>-f</code>	Использует неупорядоченные записи из входного файла.
<code>-h</code>	Выводит этот список опций и завершает работу программы.
<code>--help</code>	То же, что <code>-h</code> .
<code>--no-hash</code>	Указывает не включать в выходной двоичный файл таблиц хэширования ключей.
<code>-o file</code>	Поле <code>file</code> назначает имя для выходного файла. По умолчанию имя выходного файла определяется именем указанного в команде входного файла.
<code>--output-file=file</code>	То же, что <code>-o</code> .
<code>--statistics</code>	Выводит статистическую информацию о таблицах перевода.
<code>--strict</code>	Включает режим строгого соответствия формату Uniforum.
<code>--use-fuzzy</code>	То же, что <code>-f</code> .
<code>-v</code>	Выводит сообщения обо всех ошибках во входном файле.
<code>-V</code>	Выводит номер версии утилиты <code>msgfmt</code> и завершает ее работу.
<code>--verbose</code>	То же, что <code>-v</code> .
<code>--version</code>	То же, что <code>-V</code> .

*Полное
руководство*



Часть III

**Внутренняя структура и
окружение**



Глава 12

Использование библиотек и способы компоновки

Компилятор вырабатывает объектные файлы, которые содержат исполняемый машиной двоичный код. В действительности почти всегда объектные файлы, которые выводит компилятор, еще неполные и для преобразования в готовую программу их необходимо скомбинировать с другими объектными модулями. Даже простейшая программа "hello world" использует функцию другого объектного файла для воспроизведения на экране текстовой строки.

Эта глава описывает процесс компоновки и применение утилит, которые используются для проверки и обработки объектных файлов. Объектный файл вырабатывается компилятором и имеет имя с суффиксом `.o`. Многие из описанных в этой главе утилит работают с набором объектных модулей, которые могут находиться как в отдельных файлах, так и в *статической* библиотеке (иначе называемой *архивом*), или в *разделяемой* (*динамической*) библиотеке. Имеются также утилиты, которые обрабатывают полностью скомпонованные и готовые к запуску исполняемые файлы.

Объектные файлы и библиотеки

При компоновке объектных модулей в готовую к запуску программу компоновщик может использовать как модули из отдельных файлов в каталоге, так и модули, хранящиеся в статической или динамической (разделяемой) библиотеке. Отдельная компоновочная операция может использовать объектные модули из всех трех типов расположения, как это часто и происходит.

Объектные файлы в каталоге диска

Простейший способ компоновки программы — ее компиляция в объектные файлы, и затем — указание этих файлов компоновщику в командной строке. Это прекрасно работает, когда объектные модули используются при компоновке одной или двух программ.

Например, программа на языке С состоит из исходных файлов `main.c`, `inlet.c`, `outlet.c` и `genspru.c`. С помощью следующего набора команд исходные файлы компилируются в объектные, которые затем компонуются в готовую к запуску программу с именем `spinout`:

```
$ gcc -c main.c -o main.o
$ gcc -c inlet.c -o inlet.o
$ gcc -c outlet.c -o outlet.o
$ gcc -c genspru.c -o genspru.o
$ gcc main.o inlet.o outlet.o genspru.o -o spinout
```

После успешного выполнения этой серии команд на диске остается четыре объектных файла и один исполняемый. Пуще предоставить компилятору полное управление процессом, и подать одну команду:

```
$ gcc main.c inlet.c outlet.c genspru.c -o spinout
```

В любом случае полученный в результате выполнимый файл будет содержать весь код объектных файлов вместе с системным кодом, который компоновщик сочтет необходимым добавить.

Объектные файлы в статической библиотеке

Объектные модули компонуются из статической библиотеки почти так же, как если бы они находились в отдельных объектных файлах. Компоновщик автоматически просматривает содержимое библиотеки и помещает в программу только необходимые объектные модули. Если библиотека не содержит ни одного объекта, используемого вырабатываемой программой, то ничего из содержимого этой библиотеки не включается в исполняемый файл.

Статическая библиотека содержит объектные файлы. Она также имеет другое название — *архив*. Библиотека создается и обслуживается утилитой `ar`. Имена архивов обычно имеют префикс `lib` и суффикс (расширение) `.a`. Следующий набор команд компилирует три исходных файла в объектные и сохраняет копии полученных объектных файлов в библиотеке с именем `libspin.a`. Затем программа-компоновщик, используя объектный файл программы `main.o` и содержимое библиотеки, собирает готовую к запуску программу `spinner`:

```
$ gcc -c inlet.c outlet.c genspru.c
$ ar -r libspin.a inlet.o outlet.o genspru.o
$ gcc main.c libspin.a -o spinner
```

Первая команда `gcc` вырабатывает три объектных файла, которые помещаются в статическую библиотеку командой `ar`. Последняя команда компилирует `main.c` в объектный файл `main.o` и затем задействует компоновщик. Компоновщик считывает содержимое библиотеки `libspin.a` и пытается разрешить имеющиеся в `main.o` внешние вызовы функций и обращения к данным. Содержащийся в библиотеке `libspin.a` модуль включается в готовую программу только в том случае, если в нем находится функция или данные, адресуемые из уже используемого в компоновке модуля. При компоновке программы с библиотекой вырабатывается исполняемый файл меньшего размера, чем при компоновке с отдельными объектными

файлами, когда в готовую программу включаются все перечисленные в команде объектные файлы.

В статической библиотеке вместе с объектными модулями находится список (*index*), в котором содержатся имена всех глобально определенных функций и данных, которые находятся в библиотеке. Программа-компоновщик использует этот список для того, чтобы определить, какие из включаемых модулей находятся в библиотеке. Как правило, подобные списки создаются утилитой **ar** при обновлении уже существующей библиотеки или создании новой. Некоторые опции утилиты **ar** предотвращают построение списка. Эти опции применяют для ускорения работы при поддержке больших библиотек, когда в них вносятся многочисленные изменения. Список не обновляется до тех пор, пока не будут сделаны все модификации. Для создания новых или обновления уже имеющихся списков используется утилита **ranlib**. Например, в следующей паре команд вначале применяется утилита **ar** с опцией **-q**, чтобы быстро присоединить файлы к архиву **libspin.a** без обновления индексного списка библиотеки. Затем утилита **ranlib** обновляет список объектов этой библиотеки, она приводит список в соответствие с текущим содержанием архива:

```
$ ar -q libspin.a mongul.o strop.o klbrgr.o
$ ranlib libspin.a
```

Расположение модулей в библиотеке может иметь определенное значение. Если один и тот же программный символ определен в нескольких модулях, то компоновщик сможет обнаружить и включить в программу только первый из них. Кроме того, в архиве может находиться несколько версий одного модуля. И, опять же, компоновщик включит в готовую программу тот модуль, который будет найден первым. Чтобы изменить порядок расположения уже присутствующих в архиве модулей и чтобы добавить в него новые используются опции утилиты **ar**. Команда для запуска утилиты **ar** имеет следующий формат:

```
ar [options] [positionname] [count] archive objectfile [objectfile ...]
```

Команда **ar** относится к старейшим утилитам Unix и ее синтаксис схож с синтаксисом других традиционных утилит, таких как **tar**. В начале списка параметров команды ставятся флаги опций, при этом буквы опций группируются вместе, без пробелов между ними, опции могут быть с начальным дефисом или без него. Необязательные поля **positionname** и **count** присутствуют в команде только тогда, когда установлены использующие их опции. В команде **ar** опции подразделяются на две категории: опции, которые назначают действие (в командной строке должна быть только одна такая опция), и модифицирующие опции, которые определяют способ выполнения действий. В таблице 12.1 представлен список командных опций утилиты **ar**. Таблица 12.2 содержит описание модифицирующих опций.

Таблица 12.1. Опции утилиты **ar, назначающие действия для выполнения**

Опция	Описание
a	Удаляет из архива модули, указанные как <i>objectfile</i> . При использовании модификатора v выводится имя каждого удаляемого модуля.

Опция	Описание
m	Перемещает модули внутри архива. По умолчанию все модули, имена которых указаны как <i>objectfile</i> , переносятся в конец архива. Для перемещения указанных модулей в другие расположения используются модификаторы <i>a</i> , <i>b</i> и <i>i</i> .
p	Выводит на стандартный выход содержимое модулей <i>objectfile</i> в двоичном представлении. Если не указано ни одного <i>objectfile</i> , выводятся все модули архива.
q	Быстрое присоединение модулей, указанных как <i>objectfile</i> , в конец библиотеки, без проверки возможности их перемещения. До использования библиотеки необходимо ее обработать утилитой <i>ranlib</i> .
r	Размещает в архиве все модули с именами <i>objectfile</i> . В случае присутствия в архиве модуля с тем же именем, он заменяется новым. Если указанный в команде архив не существует, то он создается. Новые модули по умолчанию помещаются в конец архива. Модификаторами <i>a</i> , <i>b</i> или <i>i</i> можно назначить точное положение для новых модулей.
t	Выводит список модулей, содержащихся в файле архива. При использовании модификатора <i>v</i> выводятся атрибуты: время последнего изменения, имя владельца, имя группы и размер каждого модуля. Если не указаны имена модулей <i>objectfile</i> , то выводится полный список всего содержимого архива.
x	Извлекает из архива модули с именами <i>objectfile</i> в отдельные файлы. Если ни один модуль <i>objectfile</i> не указан, то извлекаются все объектные файлы.

Таблица 12.2. Опции утилиты *ar*, модифицирующие выполняемые действия

Опция	Описание
a	При добавлении в архив новых объектных файлов, располагает их сразу после указанного в поле <i>positionname</i> объектного модуля библиотеки.
b	При добавлении в архив новых объектных файлов, располагает их непосредственно перед указанным в <i>positionname</i> объектным модулем библиотеки. То же, что и <i>i</i> .
c	При необходимости создает новый архив. Новый архив, когда он нужен, создается всегда, но использование этой опции подавляет вывод предупреждения.
f	Сокращает имена объектных файлов в архиве. Обычно длина имен объектных файлов не ограничена. Сокращение имен бывает необходимо для обеспечения совместимости с некоторыми системами.
i	При добавлении в архив новых объектных файлов, располагает их непосредственно перед указанным в <i>positionname</i> объектным модулем библиотеки. То же, что и <i>b</i> .
n	Этот модификатор использует параметр <i>count</i> как переключатель между файлами с именем <i>objectfile</i> , когда в архиве их больше одного.
o	При извлечении файлов из архива, сохраняет существующие атрибуты даты и времени.
s	Создает новый список содержимого архива, даже если в архиве не происходит никаких изменений. Эта опция может использоваться отдельно в команде <i>ar -s</i> . Тот же результат достигается использованием утилиты <i>ranlib</i> .
u	Используется при добавлении файлов в архив. По опции <i>u</i> в библиотеку заносятся только новые версии объектных модулей. Применяется совместно с опцией <i>r</i> .
v	Режим вывода описаний, выводится дополнительная информация о ходе выполнения программы <i>ar</i> .
v	Выводит номер версии утилиты и завершает работу.

Объектные файлы в динамических библиотеках

Динамическая библиотека содержит объектные файлы, которые загружаются в память и компонуются с программой уже во время запуска и выполнения программы. В этом есть два преимущества: во-первых, малый размер исполняемых файлов; во-вторых, две программы и более могут использовать объектные модули из одной динамической библиотеки одновременно (именно поэтому такие библиотеки и называются *разделяемыми*).

Объектные файлы для динамической библиотеки несколько отличаются по формату от файлов, вырабатываемых для статической компоновки. Эти два типа объектных файлов принципиально отличаются тем, что в генерируемом компилятором коде применяются различные способы внутренней адресации.

Оболочка компоновщика

При выполнении программы, написанной на объектно-ориентированном языке, таком как C++, до запуска главной подпрограммы требуется выполнение статических процедур-конструкторов. Не все программы-компоновщики способны выполнять необходимые для этого действия. Поэтому к процессу компоновки добавлена оболочка, верхний уровень (front end) компоновщика, он имеет имя `collect2`.

Почти на всех системах компилятор `gcc` задействует утилиту `collect2`, которая принимает на себя всю ответственность за компоновку программы. Процесс `collect2` определяет набор статических конструкторов, выполнение которых требуется перед запуском главной процедуры вырабатываемой программы. Для надежного выполнения этих процедур-конструкторов `collect2` генерирует особую таблицу конструкторов и сохраняет ее во временном исходном файле с расширением `.c`. Затем `collect2` компилирует этот временный файл, и включает полученный из него объектный код в компонуемую программу. В начале основной процедуры, которая имеет имя `main()`, помещается вызов функции `__main()`, выполняющей вызовы всех необходимых статических конструкторов.

Программа `collect2` запускается точно так же, как утилита `ld`. Она принимает весь набор аргументов, предназначенных компоновщику, и передает их утилите `ld`, которая и выполняет основную компоновку. Может оказаться, что программу необходимо компоновать дважды. Первый раз для определения имен необходимых статических конструкторов (их список будет выдан на выходе компоновщика). И, затем, повторно — уже для получения скомпонованной машинной программы.

Размещение библиотек

Для правильной компоновки, программа-компоновщик должна быть способной находить библиотеки, необходимые для разрешения внешних ссылок.

Если программа скомпонована со статической библиотекой, то все объектные модули собраны вместе и находятся в одном выполнимом файле. Такая программа полностью самостоятельна и переносима, она может выполняться в любой совмес-

тимой системе. Даже в том случае, если исходная статическая библиотека потеряна. Разделяемые же библиотеки, напротив, должны быть доступными и во время компоновки программы и всякий раз при запуске такой программы на выполнение.

Поиск библиотек во время компоновки

Когда при компоновке требуется найти библиотеку, программа-компоновщик проводит поиск в каталогах, определенных соответствующим списком путей расположения. Пути для поиска библиотек назначаются: во-первых, при конфигурировании утилиты `ld` во время ее компиляции; во-вторых, они зависят от режима эмуляции; и, в-третьих, могут быть заданы в командной строке. Чаще всего системные библиотеки находятся в каталогах `/lib` и `/usr/lib`, поэтому два этих каталога просматриваются автоматически. Другие каталоги задаются одной или несколькими опциями `-L` в командной строке. Например, по следующей команде компоновщик будет просматривать текущий каталог и каталог с именем `/home/fred/lib` для поиска любой библиотеки, не обнаруженной в пути поиска по умолчанию:

```
$ gcc -L. -L/home/fred/lib prog.o
```

Компоновщик сначала проводит поиск разделяемых библиотек, а затем уже статических. По следующей команде будет происходить поиск в каждом каталоге из действующего списка путей сначала библиотеки с именем `libmilt.so`, а затем `libmilt.a`:

```
$ gcc -lmilt prog.o
```

Поиск библиотек может быть отменен указанием в командной строке точных имен библиотек, включающих их путь расположения. В следующем примере используется библиотека `libjj.a` из текущего каталога и библиотека `libmilt.so`, которая находится в каталоге `/home/fred/lib/`:

```
$ gcc libjj.a /home/fred/lib/libmilt.so prog.o
```

Поиск библиотек во время выполнения программ

Программа, скомпонованная с использованием разделяемых библиотек, должна находить необходимые ей разделяемые библиотеки во время своего запуска и выполнения. Поиск библиотек происходит по их именам и почти всегда без учета пути расположения. Поэтому существует возможность, что программа, скомпонованная с одной копией библиотеки, при запуске будет находить и использовать другую ее копию. Это может вызвать проблемы при переходе от одной версии библиотеки к другой ее версии без обновления программы. По этой причине имена многих библиотек содержат номер версии (например, `libm.so.6`, или `libutil-2.2.4.so`).

Во время загрузки программы и ее подготовки к запуску, а также во время выполнения программы, все необходимые разделяемые библиотеки могут быть обнаружены последовательным поиском в следующих местах:

- Каждый из каталогов списка переменной окружения `LD_LIBRARY_PATH`. (В системах UNIX имена каталогов в этом списке разделяются двоеточием ":".)
- Список библиотек в файле `/etc/ld.so.cache`, этот список поддерживается с помощью утилиты `ldconfig`.
- Каталог `/lib`.
- Каталог `/usr/lib`.

Если нужно узнать, какие библиотеки загружены и используются определенным приложением, то можно использовать утилиту `ldd`. Эта утилита будет подробно рассмотрена в этой главе.

Другая переменная среды окружения, `LD_PRELOAD` может содержать список имен предварительно загружаемых разделяемых библиотек (имена библиотек в этом списке отделяются друг от друга пробелами, символами табуляции или символами перевода строки). Предварительно загружаемые библиотеки — это библиотеки, которые загружаются в память до поиска любых других библиотек. При этом имеется возможность замещения тех функций, которые обычно используются программами из предварительно загруженных библиотек. Из соображений безопасности этой методики существует ряд ограничений, которые применяются по отношению к программам, использующим `setuid`.

Загрузка функций из разделяемой библиотеки

Для загрузки и выполнения программой функции из разделяемой библиотеки не требуется компоновка тела этой функции в саму программу. Все что нужно сделать для выполнения такой функции, это — загрузить в память разделяемую библиотеку, и затем вызывать из нее нужную функцию по имени.

В следующем примере рассматриваются две простые функции разделяемой библиотеки, программа динамически загружает библиотеку и выполняет каждую из функций.

Обе функции разделяемой библиотеки выводят строки на стандартное устройство вывода. Вывод этих строк показывает выполнение функций. Первая функция `sayhello()` объявляет и выводит внутреннюю строку "Hello from a loaded function" и выводит ее на стандартное устройство вывода:

```
/* sayhello.c */
#include <stdio.h>
void sayhello()
{
    printf("Hello from a loaded function\n");
}
```

Вторая функция выводит на стандартное устройство вывода строку, передаваемую в качестве аргумента:

```
/* saysomething.c */
#include <stdio.h>
```

```
void saysomething(char *string)
{
    printf ("%s\n", string);
}
```

Эти две функции компилируются как позиционно независимый код (т.е. с динамически перемещаемой внутренней адресацией) и используются для создания разделяемой библиотеки `libsayfn.so`. Все это делается одной командой:

```
$ gcc -fpic -shared sayhello.c saysomething.c -o libsayfn.so
```

Пример программы, которая будет использовать созданную динамическую библиотеку, использует четыре основные функции работы с разделяемыми библиотеками. Вызов `dlopen()` загружает в память разделяемую библиотеку (если она еще не загружена) и возвращает идентификатор, используемый для адресации к функциям библиотеки. Вызов `dlsym()` возвращает адреса функций. Обращение к функции `dlclose()` отсоединяет (detaches) текущую программу от загруженной разделяемой библиотеки. Если к динамической библиотеке не присоединено больше ни одной программы, то она выгружается из памяти. Функция `dlerror()` возвращает строку описания ошибки, произошедшей при последнем вызове одной из функций `dlopen()`, `dlclose()`, `dlsym()`. При отсутствии ошибок `dlerror()` возвращает значение `NULL`.

Следующая программа будет при выполнении загружать разделяемую библиотеку `libsayfn.so` и вызывать из нее две функции:

```
/* say.c */
#include <dlfcn.h>
#include <stdio.h>

int main(int argc,char *argv[])
{
    void *handle;
    char *error;
    void (*sayhello)(void);
    void (*saysomething)(char *);

    handle = dlopen("libsayfn.so",RTLD_LAZY);
    if(error = dlerror()) {
        printf("%s\n",error);
        exit(1);
    }

    sayhello = dlsym(handle,"sayhello");
    if(error = dlerror()) {
        printf("%s\n",error);
        exit(1);
    }

    saysomething = dlsym(handle,"saysomething");
    if(error = dlerror()) {
        printf("%s\n",error);
        exit(1);
    }
}
```

```
sayhello();  
saysomething("This is something");  
dlclose(handle);  
}
```

Директивой `#include` в программу включается заголовочный файл `d1fcn.h`, который содержит прототипы используемых функций и некоторые другие определения. В начале функции `main()` находятся следующие объявления: указателя идентификатора для обращения к разделяемой библиотеке `*handle`; указателя на строку сообщения об ошибке `*error`; и указателей для каждой вызываемой из библиотеки функции.

В командной строке для компиляции программы нашего примера необходимо указать разделяемую библиотеку, которая содержит используемые программой основные функции для использования динамических библиотек:

```
$ gcc say.c -ldl -o say
```

Загрузка в память разделяемой библиотеки выполняется вызовом функции `dlopen()`. При этом в аргументах необходимо указать имя библиотеки и значение флага, который определяет способ загрузки функций при их вызове из библиотеки. При вызове `dlopen()` выполняется последовательный поиск требуемой разделяемой библиотеки по ее имени в следующих вариантах расположения:

- Если имя библиотеки начинается с символа наклонной черты "/", то считается, что оно содержит полный или относительный путь к файлу. При этом имя и расположение библиотеки должны быть указаны точно, в других местах поиск не проводится. В противном случае, когда имя начинается с любого другого символа, выполняется поиск библиотеки с указанным именем последовательно в вариантах расположения, описанных далее в этом списке.
- В каждом из каталогов списка переменной окружения `LD_LIBRARY_PATH`. (В системах UNIX имена каталогов в таких списках разделяются двоеточием ":".)
- Среди библиотек из списка разделяемых библиотек, который содержится в файле `/etc/ld.so.cache`. Для поддержки этого списка применяется утилита `ldconfig`.
- В каталоге `/usr/lib`.
- В каталоге `/lib`.
- В текущем каталоге.

Второй аргумент вызова функции `dlopen()` — флаг способа загрузки библиотеки. Он может иметь следующие значения. При значении `RTLD_NOW` все функции библиотеки сразу загружаются в память и после этого становятся доступными для вызова. При значении флага `RTLD_LAZY` загрузка каждой функции задерживается до тех пор, пока ее имя не будет передано функции `dlsym()`. Каждое из двух этих значений флага может быть соединено с помощью ключевого слова `OR` со значением `RTLD_GLOBAL`. При этом все внешние вызовы загружаемой динамической библиотеки разрешаются вызовом функций из других динамических библиотек. Последние при этом также загружаются в память машины.

В нашем примере вызов `dlsym()` с идентификатором библиотеки (уже полученным из функции `dlopen()`) и именем требуемой функции библиотеки возвращает адрес памяти для вызова названной функции.

После обращений к `dlopen()` и `dlsym()` вызывается функции `dlerror()`. Благодаря этому программа может определять любые ошибки и выдавать по ним сообщения.

Утилиты для работы с объектными файлами и библиотеками

Обслуживание библиотек и расположенных в них объектных файлов состоит из достаточно устоявшихся действий. Они зависят от применяемых соглашений об именах файлов и уровня организации вашей системы. Несмотря на широкие возможности для работы с библиотеками, предоставляемые программами `gcc` и `ar`, в некоторых обстоятельствах возникает необходимость применения других утилит. Например, может понадобиться проверить содержимое какого-либо двоичного файла и выполнить некоторые его преобразования, или нужно перестроить отдельную библиотеку, или изменить порядок поиска библиотек.

Конфигурирование поиска разделяемых библиотек

Утилита `ldconfig` выполняет две основные функции работы с разделяемыми библиотеками. Во-первых, она создает и обновляет динамические компоновочные связи (links) между разделяемыми библиотеками. Эти связи служат для того, чтобы программы использовали последние и наиболее актуальные версии библиотек. Во-вторых, утилита `ldconfig` создает полный список доступных разделяемых библиотек и записывает его в файл `/etc/ld.so.cache`.

Утилита `ldconfig` считывает список каталогов для поиска разделяемых библиотек из файла `/etc/ld.so.conf`. Затем она выполняет в этих каталогах (вместе с каталогами `/lib` и `/usr/lib`) поиск любых разделяемых библиотек, при необходимости обновляя их динамические связи, и заносит все найденные и обработанные библиотеки вместе с их связями в список, который сохраняет в файле `/etc/ld.so.cache`. Имена каталогов в списке, который находится в файле `/etc/ld.so.conf`, могут отделяться друг от друга символами перевода строки, символами табуляции (отступами), двоеточиями или пробелами. Файл `/etc/ld.so.cache` имеет не текстовый формат, ручное редактирование этого файла не предусмотрено.

Перед построением файла `/etc/ld.so.cache` утилита `ldconfig` анализирует имя и содержимое каждой библиотеки и создает динамические связи библиотек таким образом, чтобы при выполнении программ использовались новейшие версии библиотек. Например, при загрузке программой библиотеки `libd1.so.2` через установленную для этой библиотеки динамическую компоновочную связь (link) может в действительности загружаться библиотека с именем `libd1-2.2.4.so`. При выпуске новой версии этой библиотеки, содержащей исправления допущенных ошибок (например, `libd1-2.2.5.so` или `libd1-2.3.0.so`), утилита `ldconfig`

обновит существующую динамическую связь `libdl.so.2`, установив ее на новую версию. Однако при выпуске принципиально новой версии, использование которой может повлиять на выполнение старых программ (скажем, `libdl-3.0.0.so`), существующая динамическая связь с именем `libdl.so.2` сохраняется и создается новая связь с именем `libdl.so.3`.

Запуск утилиты `ldconfig` должен выполняться от имени привилегированного пользователя. Поэтому перед применением этой утилиты следует зарегистрироваться в системе как суперпользователь (root). Следующая команда создаст все необходимые новые динамические связи и сгенерирует новую версию файла `/etc/ld.so.cache`:

```
% ldconfig -v
```

По опции `-v` программа `ldconfig` выводит список всех создаваемых динамических связей и описания всех выполняемых действий. Полный список опций приводится в таблице 12.3.

Таблица 12.3. Опции командной строки для утилиты `ldconfig`

Опция	Описание
<code>-?</code>	Выводит список опций программы и завершает на этом ее работу.
<code>-c filename</code>	Использует указанный файл <code>filename</code> в качестве выходного файла для хранения кэша вместо применяемого по умолчанию <code>/etc/ld.so.cache</code> .
<code>-c fmt</code>	То же, что и <code>--format</code> .
<code>-f filename</code>	Использует файл с именем <code>filename</code> в качестве входного конфигурационного файла вместо применяемого по умолчанию <code>/etc/ld.so.conf</code> .
<code>--format=fmt</code>	Указывает формат выходного файла кэша <code>/etc/ld.so.cache</code> . Поле <code>fmt</code> может содержать следующие доступные значения: <code>old</code> , <code>new</code> или <code>compat</code> . По умолчанию применяется <code>compat</code> .
<code>--help</code>	Выводит список опций программы и на этом завершает ее работу.
<code>-n</code>	По этой опции утилиты <code>ldconfig</code> обновляет динамические связи библиотек в каталогах, указанных в командной строке. При этом не создается выходной кеш-файл.
<code>-N</code>	Опция указывает не обновлять существующий кеш-файл.
<code>-p</code>	То же, что и опция <code>--print-cache</code> .
<code>--print-cache</code>	Выводит в алфавитном порядке список всех библиотек в кеш-файле. При этом выводятся имена (включающие путь расположения) всех библиотек, которые имеют компоновочные связи с указываемой в листинге текущей библиотекой.
<code>-r directory</code>	Устанавливает указанный каталог в качестве текущего и использует его как корневой каталог для поиска библиотек.
<code>--usage</code>	Выводит информацию о синтаксисе команды утилиты <code>ldconfig</code> и на этом завершает ее работу.
<code>-v</code>	Выводит описания действий, выполняемых программой.
<code>-V</code>	Выводит информацию о версии программы.
<code>--verbose</code>	Выводит описания выполняемых программой действий. Равносильна опции <code>-v</code> .
<code>--version</code>	Выводит информацию о версии программы. Равносильна опции <code>-V</code> .
<code>-x</code>	Опция указывает не создавать динамические компоновочные связи (<code>links</code>).

Вывод из объектных файлов имен программных символов

Утилита `nm` может использоваться для вывода списка всех символьических имен, определенных в объектном модуле, а также имеющиеся в объектном модуле ссылки на внешние символы. Эта утилита применяется как к отдельным объектным файлам, так и к модулям статических или разделяемых объектных библиотек. Если в командной строке не указано имя выходного файла, то по умолчанию ему присваивается имя `a.out`. Используя различные опции командной строки, можно применять разные варианты сортировки выходного списка символов — по адресам, по размеру адресуемых объектов или по именам символов. Также можно выбрать один из возможных форматов выводимого списка. Выводимые имена символов могут быть преобразованы из сокращенной формы в полную (demangled symbol names) и представлены в выходном списке в том же виде, в котором они были определены в исходном коде.

Следующий пример команды выводит список имен объектных модулей статической библиотеки `libc.a` вместе со всеми программными символами, определенными в каждом модуле, и всеми внешними символьическими ссылками, к которым имеются обращения из каждого модуля:

```
$ nm libc.a
```

В таблице 12.4 содержится список опций командной строки, поддерживаемых утилитой `nm`.

Таблица 12.4. Опции утилиты `nm`

Опция	Описание
<code>-A</code>	То же, что и <code>--print-file-name</code> .
<code>-a</code>	То же, что и <code>--debug-syms</code> .
<code>-B</code>	То же, что и <code>--format=bsd</code> . Действует по умолчанию.
<code>-C [type]</code>	То же, что и <code>--demangle</code> .
<code>-D</code>	То же, что и <code>--dynamic</code> .
<code>--debug-syms</code>	Выводит программные символы, предназначенные для использования при отладке. Обычно они не выводятся.
<code>--demangle [=type]</code>	Имена символов преобразуются из сокращенной формы в полную (demangled symbol names). То есть применяется деманглирование имен объектного уровня в их представление пользовательского уровня. При применении этой опции в выходном списке имена символов выводятся в том же виде, в котором они были определены в исходном коде. При указании типа деманглера поле <code>type</code> может содержать одно из допустимых значений: <code>auto</code> , <code>gnu</code> , <code>lucid</code> , <code>arm</code> , <code>hp</code> , <code>edg</code> , <code>gnu-v3</code> , <code>java</code> , <code>gnat</code> или <code>cortexq</code> .
<code>--dynamic</code>	Для динамических объектов, таких как разделяемая библиотека, эта опция выводит адреса динамически перемещаемой адресации вместо обычного смещения.
<code>--extern-only</code>	Выводит только те символы, которые определены как внешние (external).
<code>-f fmt</code>	То же, что и <code>--format</code> .

Опция	Описание
--format= <i>fmt</i>	При выводе использует указанный формат представления буквенных символов. Допустимыми значениями поля <i>fmt</i> могут быть bcd , ayv и posix . По умолчанию применяется --format=bcd.
-g	То же, что и --extern-only.
-h	Выводит список опций утилиты nm и завершает на этом ее работу. То же, что и опция --help.
--help	Выводит список опций утилиты nm и завершает на этом ее работу. То же, что и опция -h.
-l	То же, что и опция --line-numbers.
--line-numbers	Использует отладочную информацию, содержащуюся в объектном файле, для определения имени файла и номера строки, где содержится объявление программного символа в исходном коде.
-n	То же, что и --numeric-sort.
--no-sort	Отключает сортировку. В выходном листинге символы выводятся в том же порядке, в котором программа их находит в объектном файле.
--numeric-sort	Включает числовую сортировку символов по их адресам.
-o	То же, что и --print-file-name.
-p	То же, что и --no-sort.
-P	То же, что и --format=posix.
--portability	То же, что и --format=posix.
--print-armap	При выводе списка символов из модуля статической библиотеки в вывод включается информация из списка (индекса) модулей библиотеки вместе с любой другой имеющейся информацией об обрабатываемом модуле.
--print-file-name	Выводит имя исходного файла для каждого символа, несмотря на то, что имя исходного файла уже может стоять в заголовке списка программных символов.
-r	То же, что и --reverse-sort.
--radix= <i>base</i>	Указывает систему счисления для представления числовых полей при выводе значений символов. Поле <i>base</i> может иметь значение d для применения десятичной системы, o — для восьмеричной, или x — для шестнадцатиричной.
--reverse-sort	Применяет обратный порядок сортировки списка. Как при алфавитной, так и при числовой сортировке.
-s	То же, что и --print-armap.
--size-sort	Упорядочивает выводимый список символьических имен в соответствии с размером адресуемых ими объектов.
-t <i>base</i>	То же, что и --radix.
--target= <i>bfdname</i>	Поле <i>bfdname</i> указывает формат объектного файла в случае, когда он отличается от формата, применяемого в текущей системе. Чтобы получить список доступных форматов, следует выполнить команду: objdump -i
-u	То же, что и --undefined-only.
--undefined-only	Выводит только те символы, на которые имеются ссылки, но они не определены в текущем файле.
-V	То же, что и --version.
--version	Выводит информацию о версии утилиты nm и завершает на этом ее работу.

Удаление неиспользуемой информации из объектных файлов

Утилита **strip** убирает отладочную информацию из объектных файлов, перечисленных в командной строке. Она способна обрабатывать как отдельные вырабатываемые компилятором объектные файлы (с расширением **.o**), так и модули статической или разделяемой библиотеки. В зависимости от количества отладочной информации, обработка объектного файла утилитой **strip** может существенно уменьшить его размер. Следующий пример команды удаляет всю отладочную информацию из объектного файла **main.o** и из всех объектных модулей библиотеки **libglom.a**:

```
$ strip main.o libglom.a
```

Эта утилита после обработки заменяет имеющиеся файлы их новыми версиями с удаленной отладочной информацией. Вам еще может понадобиться необработанная версия, содержащая необходимую для отладки информацию. В таком случае необходимо или сохранить файлы перед их обработкой в другом каталоге, или использовать опцию **-o** для вывода обрезанного объектного кода в файл с другим именем.

В таблице 12.5 перечислены опции командной строки утилиты **strip**. Несколько опций имеют поле **bfdname**, которое указывает формат объектного кода. Его необходимо указывать тогда, когда предназначаемый для обработки файл имеет формат объектного кода другой машины, т.е. отличается от применяемого в текущей системе формата. Для того, чтобы получить список доступных значений **bfdname**, следует выполнить команду **objdump -i**.

Таблица 12.5. Опции утилиты **strip**

Опция	Описание
--discard-all	Удаляет все символы, не объявленные как глобальные.
--discard-locales	Удаляет сгенерированные компилятором локальные символы. Их имена обычно начинаются с буквы "L" или точки.
-F bfdname	То же, что и опция --target .
-g	То же, что и --strip-debug .
-h	Выводит список опций утилиты strip и завершает на этом ее работу.
--help	Выводит список опций утилиты strip и на этом завершает ее работу.
-I bfdname	То же, что и опция --input-target .
--input-target=bfdname	Воспринимает входные файлы в указанном формате объектного кода. Поле bfdname содержит имя формата. См. также --output-target и --target .
-K name	То же, что и опция --keep-symbol .
--keep-symbol=name	Копирует в выходной файл только символы с именами, указанными в поле name . Для сохранения более одного имени эта опция может использоваться в одной команде несколько раз.
-N name	То же, что и опция --strip-symbol .
-o bfdname	То же, что и --output-target .

Опция	Описание
<code>-o filename</code>	По этой опции, утилиты <code>strip</code> выводит объектный код в новый файл с именем <code>filename</code> вместо перезаписи обрабатываемого входного файла. Команда с этой опцией может обработать только один файл.
<code>--output-target=bfname</code>	Заменяет назначенный для обработки объектный файл его обрезанной версией (т.е. без отладочной информации), имеющей формат объектного кода с именем <code>bfname</code> . См. также <code>--input-target</code> и <code>--target</code> .
<code>-p</code>	То же, что и опция <code>--preserve-dates</code> .
<code>--preserve-dates</code>	Выходной файл после обработки утилитой будет иметь те же атрибуты времени, что и начальный объектный файл до обработки.
<code>-R name</code>	То же, что и опция <code>--remove-section</code> .
<code>--remove-section=name</code>	Удаляет из объектных файлов раздел с указанным именем <code>name</code> . Для удаления более одного раздела эта опция может использоваться в одной команде несколько раз.
<code>-s</code>	То же, что и опция <code>--strip-all</code> .
<code>-S</code>	То же, что и <code>--strip-debug</code> .
<code>--strip-all</code>	Удаляет все символы, включая также и необходимую для компоновки информацию о перемещении адресов.
<code>--strip-debug</code>	Удаляет только те символы, которые вставлены компилятором для обеспечения отладки кода.
<code>--strip-symbol=name</code>	Удаляет символ с именем, указанным в поле <code>name</code> . Как и прочие опции обрезки, эта опция может использоваться в одной команде несколько раз.
<code>--strip-unneeded</code>	Удаляет все символы, которые не используются для перемещения кода.
<code>--target=bfname</code>	Применяет указанный формат объектного кода как для считывания входных файлов, так и для вывода. Поле <code>bfname</code> содержит имя формата. См. также опции <code>--input-target</code> и <code>--output-target</code> .
<code>-v</code>	То же, что и опция <code>--verbose</code> .
<code>--verbose</code>	Выводит информацию о версии утилиты <code>strip</code> и завершает на этом ее работу.
<code>-x</code>	То же, что и опция <code>--discard-all</code> .
<code>-X</code>	То же, что и <code>--discard-locales</code> .

Вывод списка зависимостей, связанных с разделяемыми библиотеками

Утилита `ldd` выводит список всех зависимостей, связанных с разделяемыми библиотеками. Она способна считывать объектные модули как в двоичных выполняемых файлах, так и в разделяемых объектных библиотеках. Имена файлов для обработки передаются утилите в командной строке. Например, следующая команда выдаст список разделяемых библиотек, которые используются консольной программой командной оболочки `bash`. (Она, в частности, применяется в системах Linux.)

```
$ ldd /bin/bash
    libtermcap.so.2 => /lib/libtermcap.so.2 (0x40027000)
    libdl.so.2 => /lib/libdl.so.2 (0x4002b000)
    libc.so.6 => /lib/libc.so.6 (0x4002f000)
    /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

Каждая строка вывода начинается с имени разделяемой библиотеки. Имя библиотеки представлено в том же виде, в котором оно присутствует в программе. Второе имя — имя библиотеки, загружаемой в действительности. Оно включает в себя путь расположения на диске файла, из которого загружается используемая библиотека. В конце строки выводится начальный адрес, с которого библиотека загружена в память машины. Теперь вернемся к примеру. Программа оболочки `bash` при запуске использует функции библиотеки `libtermcap.so` для вывода текста на экран. Программа также вызывает функции разделяемой библиотеки `libdl.so`. Библиотека `libc.so` используется многими программами — это библиотека стандартных функций программ, написанных на языке *C*. Файл с именем `ld-linux.so` является вспомогательной программой (*helper program*) `ld.so`, которая выполняет основную работу по загрузке разделяемых библиотек и выполнению их функций.

Довольно удобно использовать `ldd` для точного определения версии разделяемой библиотеки, используемой программой во время выполнения. Другая важная причина применения утилиты `ldd` — определение любых неразрешимых ссылок на разделяемые библиотеки. Например, если программа `stwohellos` из четвёртой главы была скомпилирована правильно, но разделяемая библиотека, с которой компилировалась программа, не была должным образом установлена в системе, то вывод утилиты `ldd` будет выглядеть следующим образом:

```
$ ldd stwohellos
shello.so => not found
libc.so.6 => /lib/libc.so.6 (0x40027000)
libgcc_s.so.1 => /usr/lib/libgcc_s.so.1 (0x4015d000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

Вывод внутренней информации объектного файла

Для вывода из объектных файлов, статических и разделяемых библиотек содержащейся в них внутренней информации может использоваться программа `objdump`. Эта утилита находит в файлах разного рода дополнительную информацию и представляет ее в читаемом виде. Она может использоваться для сбора информации из файлов различного объектного формата. Для получения перечня доступных форматов объектного кода, распознаваемых утилитой `objdump`, нужно ввести следующую команду:

```
$ objdump -i
```

При запуске утилиты `objdump` для выделения информации из файлов строка команды должна содержать опции, которые определяют способ и порядок извлечения информации. Они перечислены в таблице 12.6. В таблице 12.7 представлены до-

полнительные опции, которые могут использоваться для модификации обработки входных данных или для форматирования вывода. Например, у нас есть файл `helloworld.o`, в котором содержится объектный код для машины с обратным порядком представления байт (формат "big endian"). Для вывода информационного заголовка этого файла (file header) и заголовков всех разделов объектного кода (section headers) следует ввести следующую команду:

```
$ objdump -f -h -EB helloworld.o
```

Таблица 12.6. Короткие и длинные формы основных опций утилиты `objdump`

Короткая	Длинная	Выводимая информация
-a	--archive-headers	Поля заголовка архива.
-d	--disassemble	Представление исполняемого кода на языке ассемблера.
-D	--disassemble-all	Ассемблерное представление исполняемого кода и данных.
-f	--file-headers	Все содержимое заголовка объектного файла.
-g	--debugging	Отладочная информация.
-G	--stabs	Любая информация формата STABS в необработанном виде.
-h	--section-headers	Содержимое заголовков разделов.
-H	--help	Список опций утилиты <code>objdump</code> .
-i	--info	Список поддерживаемых форматов объектного кода и машинных архитектур.
-p	--private-headers	Специфическое содержимое полей заголовка файла особого объектного формата.
-r	--reloc	Информация о перемещаемом коде.
-R	--dynamic-reloc	Информация о динамически перемещаемом коде.
-s	--source	Ассемблерное представление исполняемого объектного кода, смешанное с исходным кодом на языке высокого уровня.
-v	--full-contents	Ассемблерное представление всего объектного кода, смешанное с исходным кодом на языке высокого уровня.
-t	--symbs	Содержимое таблицы программных символов.
-T	--dynamic-symbs	Содержимое таблицы динамически перемещаемых символов.
-V	--version	Версия утилиты <code>objdump</code> .
-x	--all-headers	Содержимое заголовков всех файлов.

Таблица 12.7. Модифицирующие опции утилиты `objdump`

Опция	Описание
--adjust-vma=offset	Добавляет заданное смещение <code>offset</code> к значениям всех выводимых адресов раздела.
--architecture=machine	Задает формат объектного кода входного файла. Для определения доступных вариантов машинной архитектуры следует выполнить команду: <code>objdump -i</code>
-b bfdname	То же, что и опция <code>--target</code> .

Опция	Описание
-C type	То же, что и --demangle .
--demangle=type	Имена символов в объектном коде воспринимаются как сокращенные (mangled) имена исходного кода. Тип сокращения указывает значение поля <i>type</i> . Допустимыми могут быть следующие значения: <i>auto</i> , <i>gnu</i> , <i>lucid</i> , <i>arm</i> , <i>hp</i> , <i>edg</i> , <i>gnu-v3</i> , <i>java</i> , <i>gnat</i> и <i>comraq</i> .
--disassembler-options=op	Список одной или нескольких опций в поле <i>op</i> передается программе, которая выполняет обратное преобразование объектного кода в ассемблерные инструкции (дизассемблеру).
--disassemble-zeroes	Указывает не пропускать блоки нулевых байтов при обратном преобразовании объектного кода в ассемблерные инструкции (дизассемблировании).
-EB	То же, что и опция --endian=big .
-EL	То же, что и --endian=little .
--endian=which	Указывает прямой или обратный порядок байт машинного формата представления данных во входном объектном коде. Поле <i>which</i> может иметь значение <i>little</i> или <i>big</i> . Слово <i>little</i> соответствует прямому порядку байт (little endian), а слово <i>big</i> — обратному порядку байт (big endian).
--file-start-context	При использовании опции -s , этот модификатор будет включать в вывод контекстную информацию (<i>context</i>) из начальной области файла.
-j name	То же, что и опция --section .
-l	То же, что и --line-numbers .
--line-numbers	Включает в вывод имена файлов и номера строк.
-M	То же, что и --disassembler-options .
-m machine	То же, что и --architecture .
--prefix-address	Выводит полную адресную информацию, связанную с каждой дизассемблированной инструкцией.
--section=name	Выводит внутреннюю информацию только из разделов объектного кода с именами, указанными в поле <i>name</i> .
--show-raw-insn	Выводит шестнадцатиричные коды машинных операций вместе с мнемоническими инструкциями ассемблера.
--start-address=address	Обрабатывает данные, расположенные после указанного в поле <i>address</i> адреса.
--stop-address=address	Обрабатывает данные, расположенные до указанного в поле <i>address</i> адреса.
--target bfdname	Задает формат кода входного объектного файла. Для определения доступных форматов следует выполнить команду: <i>objdump -i</i>
-w	То же, что и опция --wide .
--wide	Снимает ограничение ширины страницы выходного листинга. По умолчанию применяется ширина в 80 столбцов.
-z	То же, что и опция --disassemble-zeroes .



Глава 13

Использование отладчика GNU

Утилита `gdb` является основным отладчиком (debugger) проекта GNU. Этот отладчик запускается из командной строки, он может использоваться для полного управления выполнением любого процесса и для исследования его работы.

На команды утилиты `gdb` будут реагировать все программы. Но только те из них, которые компилировались и компоновались с опциями включения информации, касающейся исходного кода, могут предоставить необходимые данные для трассировки процесса выполнения. Самый простой способ запуска интерактивного сеанса отладки — при запуске утилиты `gdb` указать имя программы в командной строке. Тот же результат дает запуск отладчика с последующей загрузкой программы. Кроме того, отладчику можно указать подключиться к запущенной программе, что позволяет проанализировать ход выполнения программы, которая начинает странно себя вести после некоторого времени работы. Третье назначение утилиты `gdb` заключается в "посмертном" (postmortem) анализе аварийно завершившейся программы для определения причин ее аварийного завершения.

Форматы отладочной информации

Для отладки программы необходимо, чтобы в объектный файл была включена информация о программе. С помощью такой информации отладчик имеет возможность сопоставить двоичный код его исходному тексту и предоставить в читаемом виде данные, на основе которых можно точно определить выполняемые программой действия. Без этой включаемой информации все, что известно отладчику о программе, — это только абсолютные двоичные адреса и машинные коды выполняемых операций. Эти данные очень сложно сопоставить с исходным кодом программы.

Существует несколько форматов для хранения информации в объектных файлах. Для выполнения своих задач отладчик должен корректно воспринимать формат отладочной информации. К счастью, отладчик `gdb` распознает целый ряд существующих форматов и дополнительно понимает некоторые особые расширения отладочной информации, помещаемые в код компилятором `gcc`.

Формат STABS

Формат отладочной информации *STABS* (Symbol TABle directiveS) был первоначально разработан для использования в качестве отладчика языка программирования Pascal. Он оказался довольно удобным и получил широкое распространение.

Компилятор `gcc` добавляет отладочную информацию в формате STABS (таблицы перекодировки символов) в генерируемый им ассемблерный код, и в дальнейшем эта информация включается в создаваемый ассемблером объектный код. Ассемблер добавляет информацию STABS в таблицу перекодировки символов и таблицу строк, которые включаются в конец каждого файла с расширением `.o`. Компоновщик объединяет объектные файлы в выполнимый файл, собирая все таблицы в единую таблицу перекодировки символов, которая в дальнейшем используется отладчиком для идентификации разделов выполнимого кода.

Три директивы ассемблера, используемые для создания таблиц перекодировки символов, имеют следующий вид:

```
.stabs "name:symdecs=typeinfo",type,other,description,value
.stabn type,other,description,value
.stabd type,other,description
```

Каждая из приведенных директив содержит поле `type`, которое предоставляет базовую информацию (например, является ли директива новым определением или ссылкой на существующее определение). Поле `type`, кроме того, определяет содержимое полей `other` и `description`. Поле `value` используется для указания значения, присвоенного определению.

Директива `.stabs` определяет строку, которая помещается в таблицу перекодировки программных символов. В кавычках указывается параметр `name` — имя, под которым символ вставляется в таблицу. Поле `symdecs` — это один символ (например, 'F' для глобальной функции, 'G' для глобальной переменной или 't' для имени типа) и номер типа (который фактически может состоять из двух цифр), определяющий символ как новый тип или ссылающийся на ранее определенный тип по номеру его записи. Поле `typeinfo` предоставляет дополнительную информацию о типе, например, числовой диапазон значений или размер.

Директива `.stabn` определяет числовое значение.

Директива `.stabd` устанавливает метку для текущего адреса (адреса расположения директивы). Для метки не используется параметр `value`, поскольку ее адрес определяется собственным адресом расположения директивы.

Документ, полностью описывающий формат STABS, можно найти по адресу <http://sources.redhat.com/gdb/onlinedocs/stabs.html>.

Формат DWARF

Формат отладочной информации *DWARF* на сегодняшний день уже находится на этапе эксплуатации его второго поколения, *DWARF2*. Стандарт *DWARF3* продолжает разрабатываться. Между разными версиями формата существуют определенные различия, вследствие чего они несовместимы друг с другом. Тем не менее, отладчик *gdb* распознает и считывает отладочную информацию как в формате *DWARF*, так и в формате *DWARF2*.

Отладочная информация генерируется на языке ассемблера в особые разделы кода с такими именами, как *.debug_pubnames*, *.debug_ranges*, *.debug_info* или просто *.debug*. Эти специальные разделы содержат данные и выполнимый код, который может использоваться для идентификации и извлечения информации из программы по время ее выполнения. Компоновщик группирует разделы с одинаковыми именами в один блок объектного кода, позволяющий указывать расположение различных элементов и устанавливать взаимосвязь между адресами объектного кода и строками исходных файлов.

Полную спецификацию формата *DWARF2* можно найти по адресу <http://services.worldnet.fr/~stcarrez/dwarf2.pdf>.

Формат COFF

Формат *COFF* (Common Object File Format), который иногда называют форматом "a.out", представляет собой стандартный формат объектных файлов для *UNIX System V* и многих производных от нее систем. Этот же формат принят корпорацией Microsoft для DOS и Windows. Вариант формата *COFF* для *Linux* носит название *ELF* (Executable Linux Format).

Формат *COFF* не содержит информации, специально предназначеннной для отладки. В основном, данные этого формата предназначены для компоновки. Тем не менее, в нем содержится достаточно много информации, используемой отладчиком. В таблице перекодировки символов содержатся все перемещаемые символы, а таблица перемещений содержит ссылки на элементы таблицы перекодировки и информацию о типах данных. Кроме того, он также может содержать информацию о номерах строк, которую можно использовать для сопоставления двоичного кода с исходными файлами. Таблица перекодировки символов включает полное описание каждого символа, а также его размер и дополнительные сведения.

Формат *COFF* делит объектный код на разделы. Раздел *.text* содержит выполнимый код, раздел *.data* — переменные с начальными значениями, а раздел *.bss* — неинициализированные данные. Основной причиной такого разделения является то, что при одновременном выполнении нескольких экземпляров программ они могут использовать один и тот же раздел *.text* в памяти машины. Раздел *.data* может загружаться в память в виде одного блока для установки всех начальных значений, а раздел *.bss* может находиться в файле в виде одного числа (его размера) и при загрузке программы расширяться до величины требуемой области памяти.

Информация, содержащаяся в формате *COFF*, не столь обширна, как для формата *STABS* или *DWARF*. Поэтому зачастую в объектный *COFF*-файл вставляется ин-

формация форматов STABS или DWARF, обеспечивающая выполнение расширенной отладки.

Формат XCOFF

Формат объектных файлов *XCOFF* представляет собой расширение формата COFF. Он предоставляет таблицы и ссылки для динамической компоновки. Кроме того, формат XCOFF может содержать объектный код как для 32-битной, так и 64-битной модели.

Основа формата XCOFF та же, что и для формата COFF, но включает строки STABS, содержащиеся в разделе `.debug`, и не использует применяемый форматом COFF способ хранения строк STABS в ячейках строковых таблиц. Таким образом, формат XCOFF представляет собой сочетание форматов COFF и STABS. При этом некоторые части стандарта COFF исключены во избежание дублирования данных при включении полей STABS в формат COFF.

Компиляция программы для отладки

Обязательным требованием для вывода отладчиком данных в удобной для восприятия человеком форме является обеспечение возможности поставить в соответствие выполняемому коду исходный текст программы. Это значит, что компилятор должен размещать в объектном коде некоторую дополнительную информацию. Для задания объема и типа включаемой информации можно воспользоваться опциями командной строки компилятора.

Объем включаемой информации устанавливается с помощью номеров уровней, описанных в таблице 13.1. Номер уровня устанавливается совместно с флагами, приведенными в таблице 13.2.

Таблица 13.1. Три уровня отладочной информации в объектном файле

Уровень	Описание
1	Для первого уровня в объектный код вставляется минимальный объем отладочной информации. Ее вполне достаточно для трассировки вызовов функций и исследования глобальных переменных, тем не менее, отсутствует информация для сопоставления выполняемого кода со строками исходного кода и информация для отслеживания локальных переменных.
2	Этот уровень используется по умолчанию. Помимо всей отладочной информации уровня 1 он дополнительно включает данные, необходимые для сопоставления строк исходного кода с выполняемым кодом, а также имена и расположение локальных переменных.
3	Помимо всей отладочной информации уровней 1 и 2 этот уровень включает дополнительную информацию, в частности определения макросов препроцессора.

Формат отладочной информации, включаемой в объектный файл, изменяется в зависимости от "родного" формата объектного кода платформы. Отладчик `gdb` распознает и может работать с несколькими различными форматами. Системы, использующие STABS, как правило, используют дополнительную отладочную информацию, распознаваемую отладчиком `gdb`.

Таблица 13.2. Опции компилятора `gcc`, используемые для вставки отладочной информации

Опция	Описание
<code>-g [level]</code>	Генерирует отладочную информацию в "родном" формате данной системы. Отладчик GNU, как и другие отладчики, может работать с такой информацией. На системах, использующих формат STABS, эта опция приводит к формированию дополнительной информации, которая может использоваться только утилитой <code>gdb</code> и может приводить к неправильной работе других отладчиков. По умолчанию в качестве уровня отладочной информации (необязательный параметр <code>level</code>) используется значение 2.
<code>-ggdb [level]</code>	Генерирует отладочную информацию в формате по умолчанию и при наличии возможности включает расширенную информацию для <code>gdb</code> . Информация генерируется в наилучшем доступном формате. Если ни формат STABS, ни DWARF2 не доступны, используется "родной" формат платформы.
<code>-gstabs [level]</code>	Генерирует отладочную информацию в формате STABS (если он доступен).
<code>-gstabs+</code>	Генерирует отладочную информацию в формате STABS (если он доступен) и включает расширенную информацию для отладчика <code>gdb</code> . Эта расширенная информация может приводить к неправильной работе других отладчиков.
<code>-gcoff [level]</code>	Генерирует объектный код и отладочную информацию в формате COFF (если он доступен). Этот формат чаще всего используется в системах System V до 4-го выпуска.
<code>-gxcoff [level]</code>	Генерирует объектный код и отладочную информацию в формате XCOFF (если он доступен).
<code>-gxcoff+</code>	Генерирует объектный код и отладочную информацию в формате XCOFF (если он доступен) и включает расширенную информацию для отладчика <code>gdb</code> . Эта расширенная информация может приводить к неправильной работе других отладчиков и компоновщиков.
<code>-gdwarf</code>	Генерирует объектный код и отладочную информацию в формате DWARF версии 1 (если он доступен). Этот формат используется в большинстве систем System V 4-го выпуска.
<code>-gdwarf+</code>	Генерирует объектный код и отладочную информацию в формате DWARF версии 1 (если он доступен) и включает расширенную информацию для отладчика <code>gdb</code> . Эта расширенная информация может приводить к неправильной работе других отладчиков и компоновщиков.
<code>-gdwarf-2</code>	Генерирует объектный код и отладочную информацию в формате DWARF версии 2 (если он доступен).
<code>-gvms [level]</code>	Генерирует объектный код и отладочную информацию в формате VMS (если он доступен). Этот формат используется в системах DEC VMS.

В таблице 13.2 перечислены опции командной строки компилятора `gcc`, которые могут использоваться для передачи компилятору указаний по вставке в объектный код отладочной информации. Совместно с опцией отладки можно использовать опцию оптимизации `-O`, но при этом следует иметь в виду, что оптимизация может привести к перестановке (или даже удалению) кода, что усложняет процесс слежения за ходом выполнения программы. Тем не менее, необходимость совместного использования опций отладки и оптимизации может возникать при отладке оптимизированной версии программы.

Некоторые опции из таблицы 13.2 позволяют указывать номер уровня, а другие этого не позволяют. Тем не менее, для опций, не требующих указания номера уров-

ня, всегда можно задать уровень с помощью отдельной опции `-g`. Например, для указания с опцией `-gstabs+` третьего уровня отладочной информации в командной строке используется следующая последовательность опций:

```
$ gcc -g3 -gstabs+ . . .
```

Загрузка программы в отладчик

Для загрузки программы в память и подготовки ее к отладке достаточно в командной строке `gdb` указать имя программы. По этой команде программа загружается, но для запуска ее на выполнение необходимо подать дополнительную команду. Эта пауза между загрузкой и запуском дает возможность установить точки останова (на которых выполнение программы будет приостанавливаться) и выполнить другие подготовительные операции, например, указать переменные, значения которых должны выводиться при лошаговом выполнении программы.

Ниже приведен пример выполнения программы в отладчике. Пример демонстрирует работу интерфейса, а также использование основных команд для управления выполняемой программой. Программа, написанная на языке C и названная `fibonacci.c`, выводит первые 20 членов числовой последовательности Фибоначчи:

```
/* fibonacci.c */

int current;
int next;
int nextnext;

void calcnext();
void setstart();

int main(int argc,char *argv[])
{
    int i;

    setstart();
    for(i=0; i<20; i++) {
        printf("%2d: %d\n",i+1,current);
        calcnext();
    }
    return(0);
}
void setstart()
{
    current = 0;
    next = 1;
}
void calcnext()
{
    nextnext = current + next;
    current = next;
    next = nextnext;
}
```

Для компиляции программы с включением отладочной информации достаточно будет воспользоваться опцией `-g`:

```
$ gcc -g Fibonacci.c -o Fibonacci
```

Для генерации диагностической информации отладчику `gdb` не обязательно иметь доступ к исходному коду, поскольку вся необходимая ему информация находится в объектном файле. Тем не менее, если при запуске отладчик находит исходный код, то выполняются проверки на соответствие исходного и объектного кодов. В случае обнаружения расхождений выводится предупреждение.

Следующий простой сеанс отладки загружает программу, устанавливает точку останова на входе в функцию `main()` и требует постоянного контроля значений двух переменных. После этого программа запускается и выполняется до достижения точки останова, после чего для выполнения программы в пошаговом режиме используются команды `step` и `next`:

```
$ gdb fibonacci
(gdb) break main
Breakpoint 1 at 0x80483a0: file fibonacci.c, line 14.
(gdb) display current
(gdb) display next
(gdb) run
Starting program: /home/fred/progs/fibonacci

Breakpoint 1, main (argc=1, argv=0xbfffffa9c) at fibonacci.c:14
14      setstart();
2: next = 0
1: current = 0
(gdb) step
setstart () at fibonacci.c:23
23      current = 0;
2: next = 0
1: current = 0
(gdb) step
24      next = 1;
2: next = 0
1: current = 0
(gdb) step
25 }
2: next = 1
1: current = 0
(gdb) step
main (argc=1, argv=0xbfffffa9c) at fibonacci.c:15
15      for(i=0; i<20; i++) {
2: next = 1
1: current = 0
(gdb) step
16          printf("%2d: %d\n",i+1,current);
2: next = 1
1: current = 0
(gdb) next
17      calcnext();
```

```

2: next = 1
1: current = 0
(gdb) step
calcnext () at fibonacci.c:28
28     nextnext = current + next;
2: next = 1
1: current = 0
(gdb) step
29     current = next;
2: next = 1
1: current = 0
(gdb) step
30     next = nextnext;
2: next = 1
1: current = 1
(gdb) step
31 }
2: next = 1
1: current = 1
(gdb) bt
#0 calcnext () at fibonacci.c:31
#1 0x080483d4 in main (argc=1, argv=0xbfffffa9c) at fibonacci.c:17
#2 0x40042316 in __libc_start_main (main=0x8048390 <main>, argc=1,
    ubp_av=0xbfffffa9c, init=0x8048230 <_init>, fini=0x8048460
    <fini>,
    rtd_fini=0x4000d2fc <_d1_fini>, stack_end=0xbfffffa8c)
    at ../../sysdeps/generic/libc-start.c:129
(gdb) quit
The program is running. Exit anyway? (y or n) y
$
```

Первое, что делает утилита **gdb**, — загружает выполняемую программу. После этого отладчик останавливается и ожидает ввода команды. Загруженная программа не запускается. При ее загрузке отладчик извлекает отладочную информацию и строит собственный набор внутренних таблиц. Это означает, что к моменту запуска программы отладчику известны все необходимые имена и адреса (предполагается, что программа компилировалась и компоновалась с опциями **-g**), и он готов к анализу.

Перед запуском программы для указания того, что отладчик должен выводить значения переменных **current** и **next**, используется команда **display**. Значения этих переменных будут выводиться на экран при каждой остановке выполнения программы.

Если бы программа была запущена на данном этапе, она была бы выполнена до самого конца, без остановок и какой-либо возможности вмешаться в ход выполнения. Назначение отладчика заключается в анализе хода выполнения программы, что предполагает наличие возможности выбора точки, в которой выполнение программы можно остановить, проанализировать текущее состояние и далее выполнять команды в пошаговом режиме. В приведенном примере команда **break main** устанавливает точку останова на входе в функцию **main()**. Отладчик подтверждает получение команды путем вывода адреса и номера строки, в которой установлена

точка останова. Выводится исходный код строки 14, содержащий вызов функции `setstart()`, устанавливающей два начальных значения для вычисления последовательности Фибоначчи.

Команда `run` запускает выполнение программы. Выполнение начинается с кода инициализации, присутствующего в любой программе, сгенерированной компилятором `gcc`. Код инициализации выполняется до тех пор, пока он не запустит функцию `main()`. Как только выполнение программы дойдет до строки 14 исходного кода, программа будет остановлена, и на экран будут выведены значения двух выбранных переменных. После этого отладчик `gdb` будет ожидать ввода новой команды.

Команда `step` используется для выполнения одной строки исходного кода. На уровне языка ассемблера одна строка исходного кода, как правило, соответствует нескольким инструкциям. Отладчик `gdb` выполнит требуемое количество инструкций, соответствующее одной строке исходного файла программы. В приведенном примере команда `step` выполняет вызов функции `setstart()` и останавливает выполнение на первой строке функции, т.е. на строке 23 исходного файла. Следующие две команды `step` выполняют два оператора в функции `setstart()`, которые устанавливают начальные значения для переменных `current` и `next`, равные 1 и 0 соответственно. Следующая команда `step` выходит из функции и останавливает выполнение программы на начале цикла `for`.

После входа по команде `step` в цикл программа останавливается на вызове функции `printf()`. Если на данном этапе выполнить еще одну команду `step`, то выполнение перейдет к функции `printf()`, что может быть нежелательным. Если для компиляции функции `printf()` не применялись специально опции `-g`, то отладочная информация для нее включена не будет. Несмотря на то, что отладчик может выполнить код функции `printf()` в пошаговом режиме, он не сможет выводить значения или исходный код. Поэтому вместо команды `step` у нас используется команда `next`, выполняющая вызов функции как одну строку кода и останавливает выполнение программы на строке, следующей непосредственно после вызова функции.

Серия команд `step` используется для выполнения операторов цикла. Такую процедуру можно применять для исследования выполняемых программой действий и определения строки, содержащей ошибку. С ее помощью можно в интерактивном режиме проверить, что все операции выполняются так, как это предполагалось при написании программы.

В приведенном примере команда `bt` (backtrace) используется для генерации информации обратной трассировки вызовов функций. Эта информация описывает путь выполнения программы, по которому она достигла данной точки. Информация обратной трассировки содержит не только имена функций, но также имена и значения аргументов, переданных каждой функции, и имена исходных файлов этих функций.

"Посмертный" анализ (Postmortem)

В системе UNIX при аварийном завершении программы обычно вызывается функция операционной системы, которая "сливает" образ программы из памяти в файл

с именем **core**. Если программа компилировалась с опцией **-g**, определить строку кода, вызвавшего аварийное завершение, достаточно просто.

Следующая программа обязательно будет аварийно завершаться при каждом запуске, поскольку она пытается записать в память данные по нулевому адресу, который запрещен для использования программами:

```
/* falldown.c */
char **nowhere;
void setbad();

int main(int argc,char *argv[])
{
    setbad();
    printf("%s\n",*nowhere);
}
void setbad()
{
    nowhere = 0;
    *nowhere = "This is a string\n";
}
```

Приведенную программу можно откомпилировать и запустить на выполнение с помощью следующих команд, программа будет "слетать" и при этом будет создаваться файл **core** с образом программы:

```
$ gcc -g falldown.c -o falldown
$ falldown
Segmentation fault (core dumped)
```

Для загрузки в отладчик как программы, так и файла с ее "посмертным" дампом **core** необходимо выполнить следующую команду:

```
$ gdb falldown core
```

В большинстве случаев приведенная команда представит всю необходимую информацию, поскольку она выводит строку кода, выполнявшуюся в момент аварийного завершения работы программы. Следующий сеанс отладки демонстрирует информацию, которую выводит отладчик **gdb**, а также содержит пример использования других команд, позволяющих получать дополнительную информацию:

```
$ gdb falldown core
Core was generated by 'falldown'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0 0x080483d0 in setbad () at falldown.c:14
14      *nowhere = "This is a string\n";
(gdb) print nowhere
$1 = (char **) 0x0
(gdb) bt
#0 0x080483d0 in setbad () at falldown.c:14
```

```
#1 0x080483a5 in main (argc=1, argv=0xbfffffa8c) at falldown.c:8
#2 0x40042316 in __libc_start_main (main=0x8048390 <main>, argc=1,
    ubp_av=0xbfffffa8c, init=0x8048230 <_init>, fini=0x8048410
<_fini>,
    rtld_fini=0x4000d2fc <_dl_fini>, stack_end=0xbfffffa7c)
    at ../sysdeps/generic/libc-start.c:129
(gdb) quit
$
```

В приведенном примере на экран выводится строка кода с ошибкой (та строка, в которой предпринимается попытка записи по абсолютному адресу "0"). Выводимая информация включает имя функции и имя файла, показывая положение этой строчки исходного кода. Команда `print` используется для проверки того, что указатель `nowhere` указывает на ошибочный адрес, а команда `bt` используется для обратной трассировки кода, которая показывает, каким путем программа пришла к аварийному завершению. Кроме того, имеется целый ряд других команд, которые позволяют получить достаточно информации для немедленного устранения ошибки в программе.

Подключение отладчика к выполняемой программе

Возможность присоединения отладчика к выполняемому процессу может быть весьма полезной. Если, например, программа после некоторого времени входит в бесконечный цикл, то можно подключить отладчик к программе и точно определить место, в котором она зациклилась. Еще одна ситуация, в которой желательно подключение отладчика к выполняемой программе, может возникнуть при работе с интерактивной программой, когда она начинает делать что-то не то. В этом случае отладчик можно подключить к программе и отследить причину выполнения некорректных действий.

Для подключения отладчика к выполняемому процессу необходимо соблюдение двух условий. Во-первых, выполняемый процесс опять-таки должен быть откомпилирован с одной из форм опции `-g`. А во-вторых, необходимо как-то определить идентификатор процесса (*PID* — Process ID). Если идентификатор PID процесса не известен, то для его определения можно воспользоваться командой `ps`. Аргументы командной строки для команды `ps` в разных системах могут отличаться, поскольку разные операционные системы предоставляют информацию о выполняемых процессах в различных форматах. Типичную форму команды `ps` показывает следующий пример, в котором она определяет, что идентификатор процесса `looper` равен `29627`:

```
$ ps ax | grep looper
29627 pts/4 R 1.58 looper
32298 pts/4 S 0:00 grep looper
```

Результаты выполнения команды `ps` указывают на то, что процесс `looper` является активным (`R` означает "running", то есть в стадии выполнения). Фактически програм-

ма `looper.c` была специально написана с бесконечным циклом — для демонстрации подключения отладчика `gdb` к выполняемому процессу:

```
/* looper.c */
void goaround(int);
int main(int argc,char *argv[])
{
    printf("started\n");
    goaround(20);
    printf("done\n");
}
void goaround(int counter)
{
    int i = 0;
    while(i < counter) {
        if(i++ == 17)
            i = 10;
    }
}
```

Основная функция программы `looper` вызывает функцию `goaround()`, выполнение которой не прекращается, поскольку значение переменной цикла `i` никогда не достигает значения `counter`. Приведенную программу можно откомпилировать с генерацией отладочной информации с помощью следующей команды:

```
$ gcc -g looper.c -o looper
```

Для запуска программы в фоновом режиме введите команду:

```
$ looper &
```

Используемая для запуска программная оболочка, как правило, выведет идентификатор нового процесса. Если же идентификатор не выводится, для его определения можно воспользоваться командой `ps`. Приведенная ниже последовательность показывает порядок подключения отладчика к процессу и его использование для обнаружения ошибки. В командной строке на запуск отладчика указывается имя находящегося на диске двоичного файла программы и идентификатор процесса:

```
$ gdb looper 29627
Attaching to program: /home/fred/looper, process 29627
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
0x080483ea in goaround (counter=20) at looper.c:14
14          if(i++ == 17)
(gdb) display i
1: i = 14
(gdb) step
13      while(i < counter) {
1: i = 15
(gdb) step
14          if(i++ == 17)
```

```
1: i = 15
(gdb) step
13      while(i < counter) {
1: i = 16
(gdb) step
14          if(i++ == 17)
1: i = 16
(gdb) step
13      while(i < counter) {
1: i = 17
(gdb) step
14          if(i++ == 17)
1: i = 17
(gdb) step
15          i = 10;
1: i = 18
(gdb) step
13      while(i < counter) {
1: i = 10
(gdb) step
14          if(i++ == 17)
1: i = 10
(gdb) step
13      while(i < counter) {
1: i = 11
(gdb) quit
The program is running. Quit anyway (and detach it)? (y or n) y
Detaching from program: /home/fred/looper, process 29627
$
```

Как видно из первых строк листинга, вначале отладчик считывает бинарный файл с диска и загружает из программы и используемых библиотек таблицы перекодировки символов. Если библиотеки компилировались с включением отладочной информации, то при необходимости можно выполнять трассировку кода и в библиотеках, но в приведенном простом примере отладчик используется только для трассировки цикла в функции `goaround()`.

Далее отладчик `gdb` подключается к процессу, останавливает его выполнение на текущей инструкции и выводит соответствующую ей строку исходного кода с ее номером в исходном файле. На данном этапе можно установить точки останова, просмотреть значения переменных и выполнить любые другие действия, обычно предпринимаемые в процессе отладки. В приведенном примере команда `display` используется для вывода значения переменной `i` при каждом останове программы. Для отслеживания значения этой переменной при выполнении нескольких строк цикла используется серия команд `step`. Они показывают, что значение переменной `i` изменяется таким образом, что цикл не может прийти к завершению.

В конце сеанса отладки с помощью команды `quit` отладчик отсоединяется от процесса и позволяет продолжить его выполнение. В приведенном примере после отключения отладчика `gdb` программа продолжит выполнение бесконечного цикла, пока она не будет остановлена каким-нибудь другим способом. Процесс снова нор-

мально выполняется, и в любой момент к нему снова можно подключить отладчик. Для остановки процесса `looper` перед командой `quit` можно было бы выполнить команду `kill`.

Сводная таблица команд отладчика

Отладчик `gdb` в интерактивном режиме поддерживает огромное количество команд. Для их просмотра в командной строке `gdb` можно ввести команду `help`. На экране появится длинный список категорий команд. В некоторых категориях приведены описания команд, в то время как в других находятся подкатегории, содержащие списки команд.

В таблице 13.3 приведены только некоторые наиболее часто используемые команды, которые нужны практически в любом сеансе отладки.

Таблица 13.3. Часто используемые команды отладчика `gdb`

Команда	Описание
<code>awatch</code>	Устанавливает слежение за значением. При каждой операции считывания или записи в области памяти, соответствующей указанному имени символа, выполнение программы останавливается. Также см. команды <code>rwatch</code> и <code>watch</code> .
<code>backtrace</code>	Выводит обратный след вызовов всех стековых фреймов, показывающий вызовы функций и значения их аргументов, через которые выполнение дошло до данной точки. Эта команда имеет короткую форму <code>bt</code> .
<code>break</code>	Устанавливает точку останова, которая останавливает выполнение программы на указанной строке кода или в указанной функции.
<code>clear</code>	Снимает точку останова в ранее указанной с помощью команды <code>break</code> строке кода или функции.
<code>continue</code>	Продолжает выполнение программы, остановленной отладчиком.
<code>Ctrl-C</code>	Прерывает выполнение программы так, как если бы в выполняемой строке была определена точка останова.
<code>disable</code>	Отключает определенные точки останова по их номерам.
<code>display</code>	При каждом останове программы выводит на экран значение указанного выражения.
<code>enable</code>	Включает определенные точки останова по их номерам.
<code>finish</code>	Продолжает выполнение программы, остановленной отладчиком, до выхода из текущей функции.
<code>ignore</code>	Устанавливает счетчик игнорирования точки останова. Так, например, команда <code>ignore 4 23</code> указывает, что до остановки на точке останова с номером 4 она должна быть пройдена 23 раза.
<code>info breakpoints</code>	Выводит состояние и описание для всех точек останова с их номерами.
<code>info display</code>	Выводит состояние и описание для всех ранее введенных команд <code>display</code> , включая их номера.
<code>kill</code>	Завершает выполнение текущего процесса.
<code>list</code>	Выводит на экран десять строк кода. Если в командной строке нет других аргументов, выводимые строки начинаются с текущей строки. Если в команде указано имя функции, выводимые строки начинаются от начала функции. Если указан номер строки, то эта строка будет выведена первой.

Команда	Описание
<code>load</code>	Динамически загружает указанный выполняемый файл в <code>gdb</code> и подготавливает его к отладке.
<code>next</code>	Продолжает выполнение остановленной программы и выполняет все инструкции языка ассемблера, соответствующие одной строке исходного кода. Данная команда воспринимает вызов функции как одну строку и не останавливает выполнение до выхода из этой функции.
<code>nexti</code>	Продолжает выполнение остановленной программы и выполняет одну инструкцию ассемблера. Эта команда воспринимает вызов функции как одну строку и не останавливает выполнение до выхода из данной функции.
<code>print</code>	Выводит на экран значение указанного выражения.
<code>ptype</code>	Выводит на экран тип указанного элемента.
<code>return</code>	Приводит к немедленному выходу из текущей функции.
<code>run</code>	Запускает программу на выполнение с начала.
<code>rwatch</code>	Устанавливает слежение за значением. При этом выполнение программы останавливается при каждой операции считывания из области памяти с указанным именем. См. также <code>watch</code> и <code>awatch</code> .
<code>set</code>	Устанавливает значение указанной переменной, равным результату выражения. Так, например, команда <code>set nval=54</code> запишет значение 54 в область памяти с именем <code>nval</code> .
<code>step</code>	Продолжает выполнение остановленной программы и выполняет все инструкции ассемблера, соответствующие одной строке исходного кода. По этой команде отладчик входит в вызываемые функции.
<code>stepi</code>	Продолжает выполнение остановленной программы и выполняет одну инструкцию языка ассемблера. По этой команде отладчик входит в вызываемые функции.
<code>txbreak</code>	Устанавливает временную точку останова (которая срабатывает только один раз) на выходе текущей функции. Также см. команду <code>xbreak</code> .
<code>undisplay</code>	Удаляет ранее заданное выражение <code>display</code> по его номеру.
<code>watch</code>	Устанавливает слежение за значением. При этом выполнение программы останавливается при каждой операции записи в область памяти с указанным именем. См. также <code>rwatch</code> и <code>awatch</code> .
<code>whatis</code>	Выводит тип данных и значение указанного выражения.
<code>xbreak</code>	Устанавливает точку останова на выходе из текущей функции. Также см. команду <code>txbreak</code> .

Глава 14



Утилиты `make` и `Autoconf`

Эта глава является введением в работу с утилитой `make`, которая может использоваться для управления проектами разработки программного обеспечения, и с утилитой `Autoconf`, используемой для настройки конфигурации и упаковки программного обеспечения с открытым кодом для его выпуска и распространения. Глава не является полным руководством по всем возможностям этих утилит, но она содержит достаточно большой объем материала для помощи программисту в выборе средств разработки программного обеспечения. В ней кратко рассмотрены назначение и общие принципы работы каждой из рассматриваемых утилит.

Утилита `make`

Утилита `make` является наиболее часто используемым средством в процессе разработки программного обеспечения. Основная идея утилиты `make` очень проста — исследовать файлы исходного и объектного кода для определения того, какие из исходных файлов следует перекомпилировать для создания новых объектных файлов. Утилита `make` предполагает, что во все более новые исходные файлы, чем соответствующие им объектные, были внесены изменения, и поэтому их следует скомпилировать заново. Вся работа `make` основана именно на этом принципе. Взаимосвязь между объектным файлом и файлом исходного кода, на основе которого был создан данный объектный файл, называется *зависимостью* (*dependency*). Объектный файл, полученный после выполнения команд, определяемый зависимостью, называется *целевым файлом* или *целью* (*target*).

Для получения зависимостей утилита `make` считывает определяющий их сценарий. Этот сценарий, как правило, имеет имя `makefile` или `Makefile`. Он содержит список зависимостей, а также команды, которые преобразуют исходный код в объектные файлы. Например, следующие строки сценария `makefile` указывают, что

программа **frammis** имеет зависимость от исходного файла **frammis.c**, сценарий также содержит команду компилятора **gcc**, необходимую для создания целевого файла:

```
frammis: frammis.c
gcc frammis.c -o frammis
```

Следует сказать, что утилита **make** была создана еще в первые годы существования UNIX и до сих пор сохранила странное требование к команде, которая находится под строкой зависимости — команде ДОЛЖЕН предшествовать символ табуляции. Символ табуляции, хотя он и не виден на экране и при печати, является обязательным требованием синтаксиса сценария **makefile**. Если табуляцию в начале строки команды пропустить (или вместо табуляции воспользоваться пробелами), то будет выдано сообщение об отсутствии оператора, которое, к счастью, содержит и номер той строки, в которой отсутствует табуляция.

Очень часто файл зависит от другого файла, полученного на основании другой зависимости. Например, следующие зависимости приводят к компиляции программы **frammis**:

```
frammis: frammis.o
gcc frammis.o -o frammis

frammis.o: frammis.c
gcc -c frammis.c -o frammis.o
```

В приведенном примере выполняемая программа **frammis** зависит от файла **frammis.o**, который, в свою очередь, как видно из листинга, зависит от файла **frammis.c**. При запуске утилиты **make** она начинает считывать содержимое всего компоновочного файла (его другое название *make-файл*) и на основе цепочки зависимостей строить внутреннее дерево, при этом первая зависимость в файле становится корнем дерева. В приведенном примере корневой является зависимость **frammis**, под ней в дереве находится зависимость **frammis.o**. После построения дерева зависимостей программа начинает с корневого элемента и опускается до низшего уровня, а затем выполняет проход в обратном направлении, выполняя те команды, которые должны быть выполнены для достижения корня дерева и соблюдения всех зависимостей.

Следует отметить, что утилита **make** строит и выполняет только одно внутреннее дерево, поэтому возможна ситуация, когда в файле определены зависимости, которые не будут выполняться, поскольку они не связаны через другие зависимости с первой зависимостью в файле. Это вовсе не ограничение, как может показаться на первый взгляд, поскольку всегда можно вставить специальные зависимости, включающие все остальные зависимости, как показано в следующем примере, где слово **all** используется в качестве фиктивной цели:

```
all: frammis cooker

frammis: frammis.c frammis.h
gcc frammis.c -o frammis

cooker: cooker.c cooker.h
gcc cooker.c -o cooker
```

Цель `all` связана с зависимостями `frammis` и `cooker`. Несмотря на то, что цель `all` имеет две зависимости, с ней не связаны команды. Тем не менее, это не ошибка, поскольку единственным ее назначением является принудительное соблюдение зависимостей. В качестве корневого элемента внутреннего дерева компоновки будет использоваться элемент `all`, а `frammis` и `cooker` будут узлами дерева.

В компоновочный файл могут включаться элементы, не являющиеся зависимостями, и связанные с ними команды, но они также используются с единственной целью — определить зависимости и команды.

Make-файл вызывается следующей простой командой:

```
$ make
```

По умолчанию утилита `make` сначала в текущем каталоге осуществляет поиск файла с именем `makefile`. Затем, если такой файл не найден, осуществляется поиск файла `Makefile`. Если и такой файл не найден, никаких дальнейших действий `make` не предпринимается. В командной строке можно указать имя make-файла, как показано в следующем примере:

```
$ make -f mymakefile.text
```

Внутренние определения

Для удобства построения правил на основе зависимостей и требований к целевым файлам можно использовать предопределенные макросы и устанавливать неявные правила, которые могут использоваться утилитой `make` для преобразования файла из одного типа в другой.

Макросы

Макрос можно определить тремя различными способами. Следующее определение цели в компоновочном сценарии демонстрирует назначение и использование макросов:

```
showmacros:
    echo HOME is $(HOME)          #определенена как переменная среды
    echo compile.f is $(COMPILE.f)  #определенена по умолчанию для
                                    #make-файла
    echo HERBERT is $(HERBERT)     #определяется локально в make-файле
```

При считывании make-файла, когда утилита `make` встречает символ `#`, оставшаяся часть строки считается комментарием и игнорируется.

Цель `showmacros` будет всегда выполнять связанные с ней команды, поскольку для нее не приведены зависимости, а по умолчанию предполагается, что цель должна быть создана. Содержимое переменной может извлекаться и использоваться в выражениях. Для этого имя переменной указывается с предшествующим знаком доллара и заключается в скобки `$(...)`. В приведенном примере значение переменной `HOME` считывается из переменной окружения, `COMPILE.f` представляет собой имя, определенное в самой утилите `make`, а переменная `HERBERT` определена в самом make-файле с помощью примерно такой строки:

```
HERBERT=Herbivore
```

Результат обработки make-файла будет выглядеть следующим образом:

```
echo HOME is /home/arthur
HOME is /home/arthur
echo COMPILE.f is f77 -c
COMPILE.f is f77 -c
echo HERBERT is Herbivore
HERBERT is Herbivore
```

Каждый раз команда `echo` выводится в начале выводимой ею строки, поскольку по умолчанию для утилиты `make` установлено, что команда перед выполнением отображается на экране.

Правила суффиксов

В make-файле можно назначать правила, которые будут распознавать типы файлов по их суффиксам (расширениям) и автоматически транслировать один тип в другой. В следующем примере приведен make-файл, который распознает суффиксы и определяет пару команд, которые транслируют файл с одним суффиксом в новый файл с назначенным ему другим суффиксом:

```
all: hello.o hello.s

hello.o: hello.c
hello.s: hello.c

.SUFFIXES: .o .c .s

.c.o:
    gcc -c $<

.c.s:
    gcc -S $<
```

Приведенный компоновочный сценарий предназначен для создания двух целей: `hello.o` и `hello.s`. Поскольку правила выработки этих целевых файлов не устанавливают команды, то распознаются три суффикса: `.c`, `.o` и `.s`. Правило для суффиксов `.c` и `.o` преобразовывает файл с суффиксом `.c` в файл с суффиксом `.o`. А правило для `.c` и `.s` — файл с суффиксом `.c` в файл с суффиксом `.s`. Специальный макрос `$<` представляет собой ссылку на имя файла, используемого для создания цели. Результат выполнения рассмотренного make-файла будет тем же, что и результат выполнения следующего компоновочного сценария:

```
hello.o: hello.c
    gcc -c hello.c
hello.s: hello.c
    gcc -S hello.c
```

Правила суффиксов, как правило, действуют сложнее, чем показано в этом примере, тем не менее, их не приходится писать вручную. В утилиту `make` встроено большое количество правил суффиксов, которых достаточно для того, чтобы от разработчика требовалось только указать команду.

Просмотр определений

Существуют опции командной строки, которые позволяют просмотреть полный список макросов и определений правил суффиксов, поддерживаемых утилитой `make`. Опция `-p` не мешает нормальному считыванию и выполнению `make`-файла, но при этом будут выводиться все назначенные макросы и действующие правила суффиксов. Для просмотра всего списка введите следующую команду:

```
$ make -p | more
```

Для просмотра того же списка, но с запрещением на выполнение команд сценария выполните следующую команду:

```
$ make -p -q | more
```

При необходимости просмотра только встроенных определений утилиты `make` без вывода определений локального `make`-файла можно указать утилите `make` пустой файл для считывания:

```
$ make -p -f /dev/null | more
```

Как написать `make`-файл

Если вы новичок в написании `make`-файлов, то лучше всего скопировать существующий файл и изменить его в соответствии со своими намерениями. После нескольких правок вы поймете общий принцип их построения. Если необходимо изучить утилиту `make` в объеме, достаточном для написания `make`-файлов с нуля, придется потратить некоторое время на исследования и эксперименты. На самом деле это не так сложно, как может показаться. Просто имеются некоторые особенности, которые могут сбивать с толку поначалу, пока вы не научитесь все делать правильно.

Можно попробовать написать шаблон `make`-файла, который будет использоваться в качестве заготовки при написании каждого нового `make`-файла. Но, к сожалению, для сценариев компоновки нет универсальной формы, которая бы подошла для любого случая. Ниже приводится пример обобщенного `make`-файла, который компилирует две программы, написанные на языке `C`, и компонует их в готовый к запуску выполнимый файл:

```
CC=gcc
PROGS=howdy hello
CFLAGS=-Wall

all: $(PROGS)

howdy: howdy.c

hello: hello.c
    $(CC) $(CFLAGS) hello.c -o hello

clean:
    rm -f *.o
    rm -f *.so
    rm -f *.a
    rm -f $(PROGS)
```

Переменной `CC` присваивается значение `gcc`, а переменной `CFLAGS` — значение `-Wall`. Список имен целей хранится в переменной `PROGS`. Представленный макет-файл компилирует две программы одним и тем же способом. Но для одной из них используется встроенная команда, а для другой команда указывается явно:

```
gcc -Wall howdy.c -o howdy
gcc -Wall hello.c -o hello
```

Существует предопределенная "чистящая" цель `clean`, которую можно вызвать в любой момент для удаления всех промежуточных файлов, созданных при выполнении компоновочного сценария. Текущий набор команд не оставляет на диске файлы `.o`, `.so` или `.a`, поэтому здесь чистка не нужна. Однако make-файлы растут по мере развития проекта и генерируют все возможные типы промежуточных файлов. В таком случае без удаления временных файлов не обойтись. По умолчанию утилита `make` пытается построить первую цель, найденную в make-файле, но ее можно попросить построить любую цель, явно указав ее имя в командной строке, что показывает следующий пример:

```
$ make clean
```

Некоторую помощь можно получить и от компилятора. В главе 18 приведены примеры использования компилятора для формирования списка зависимостей, по-мещаемого в make-файл.

Опции утилиты `make`

Версий утилиты `make` столько же, сколько и версий UNIX. Все они, по своей сути, одинаковы, но могут отличаться отдельными свойствами и характеристиками. Версия GNU утилиты `make` привлекает тем, что она свободно распространяется в форме исходного кода. Она, несмотря на то, что содержит собственные расширения, пожалуй, лучше всего будет работать с компилятором GCC в любой системе. В частности, если компилятор GCC планируется создавать из исходного кода, то имеет смысл установить пакет `binutils`, включающий `make` и несколько других утилит. Утилиты пакета `binutils` гарантированно совместимы с GCC. Многие опции командной строки общие для всех версий утилиты `make`. Опции, поддерживаемые утилитой `make` версии GNU, перечислены в таблице 14.1.

Таблица 14.1. Опции командной строки утилиты `make`

Опция	Описание
<code>--assume-old=filename</code>	Указывает не перекомпоновывать файл независимо от его возраста и не перекомпоновывать другие файлы, основанные на зависимостях от этого файла.
<code>--assume-new=filename</code>	Предполагает, что указанный файл является новым. Также и все файлы, зависящие от указанного файла, должны быть перекомпонованы.
<code>-C directory</code>	Выполняет переход в указанный каталог перед поиском файлов для определения зависимостей.
<code>--directory=directory</code>	То же, что и <code>-C</code> .
<code>-d</code>	То же, что и <code>-debug=a</code> .

Опция	Описание
<code>--debug [=flags]</code>	Выводит информацию об обработке в форме, которую можно использовать для отладки ошибок make-файла. Если флаги не указаны, то выводится базовая отладочная информация. Значение параметра <code>flag</code> может быть любой комбинацией следующих букв: <code>a</code> — выводит все типы отладочной информации. Эта опция выводит большие описания результатов. <code>b</code> — выводит базовую отладочную информацию, включая список устаревших целей и данные об успешности выполнения команд. <code>i</code> — выводит информацию о поиске неявных правил для каждой цели и также информацию, выводимую при использовании опции <code>b</code> . <code>j</code> — выводит информацию о вызове подкоманд. <code>m</code> — другие опции при построении компоновочных файлов с помощью текущего целевого make-файла блокируются, но этот флаг включает другие флаги, используемые при генерации самого целевого make-файла. <code>v</code> — выводит ту же информацию, что для опции <code>b</code> , и дополнительно информацию о целях, которые не требуют выполнения команд.
<code>--dry-run</code>	Холостой запуск. Указывает не выполнять никаких команд. Эта опция выводит все команды, которые были бы выполнены в случае, если бы это был не холостой запуск.
<code>-e</code>	То же, что и <code>--environment-overrides</code> .
<code>--environment-overrides</code>	Переменные среды замещают переменные, определенные в make-файле.
<code>-f filename</code>	То же, что и <code>--file</code> .
<code>--file=filename</code>	Использует указанный файл в качестве make-файла и отменяет поиск файла с именем <code>Makefile</code> или <code>makefile</code> .
<code>-h</code>	Выводит этот перечень опций.
<code>--help</code>	Выводит этот перечень опций.
<code>-i</code>	То же, что и <code>--ignore-errors</code> .
<code>-I directory</code>	То же, что и <code>--include-dir</code> .
<code>--ignore-errors</code>	В нормальном режиме обработка завершается при первой неудаче компоновки цели, но при этой опции обработка не останавливается и переходит к созданию следующей цели.
<code>--include-dir=directory</code>	Поиск включаемых компоновщиком make-файлов будет производиться в указанном каталоге.
<code>-j [number]</code>	То же, что и <code>--jobs</code> .
<code>--jobs [=number]</code>	Указывает количество команд, которые могут выполняться одновременно. Если число не указано, утилита <code>make</code> будет запускать максимально возможное количество команд.
<code>--just-print</code>	То же, что и <code>--dry-run</code> .
<code>-k</code>	То же, что и <code>--keep-going</code> .
<code>--keep-going</code>	Указывает после возникновения ошибки обработать столько целей, сколько возможно. Файлы, имеющие зависимости от цели, которая не была создана, также создаваться не будут, но ошибка в одной зависимости не исключает возможность обработки остальных.

Опция	Описание
<code>-l [number]</code>	То же, что и <code>--max-load</code> .
<code>--load-average[=number]</code>	То же, что и <code>--max-load</code> .
<code>--makefile=filename</code>	То же, что и <code>--file</code> .
<code>--max-load[=number]</code>	Новые команды не будут запускаться на выполнение, если уже запущена по крайней мере одна команда, и средняя загрузка системы выше, чем указанное значение (число с плавающей точкой). Если число не указано, то максимальная загрузка не ограничена.
<code>-n</code>	То же, что и <code>--dry-run</code> .
<code>--new-file=filename</code>	То же, что и <code>--assume-new</code> .
<code>--no-builtin-rules</code>	Отменяет встроенные правила и определения суффиксов, возможность назначения пользователем собственных правил и определений при этом остается.
<code>--no-builtin-variables</code>	Исключает использование встроенных переменных, возможность назначения пользователем собственных переменных при этом остается. Эта опция также автоматически устанавливает опцию <code>--no-builtin-rules</code> .
<code>--no-keep-going</code>	То же, что и <code>--stop</code> .
<code>--no-print-directory</code>	Отменяет действие опции <code>--print-directory</code> .
<code>-o filename</code>	То же, что и <code>--assume-old</code> .
<code>--old-file=filename</code>	То же, что и <code>--assume-old</code> .
<code>-p</code>	То же, что и <code>--print-data-base</code> .
<code>--print-data-base</code>	Выводит действующие правила и значения переменных. Эти данные представляют собой комбинацию предопределенных значений и содержимого make-файла.
<code>--print-directory</code>	Выводит сообщение, содержащее имя рабочего каталога до и после выполнения make-файла. Эта опция имеет смысл только в том случае, когда make-файлы вызывают друг друга.
<code>-q</code>	Указывает не выполнять команды и не выводить никаких результатов кроме кода завершения. Код завершения 0 говорит о том, что все цели новые и при нормальном запуске make-файла компиляция не проводилась бы. Код завершения 1 указывает, что одну или несколько целей нужно перекомпилировать. Код завершения 2 говорит о наличии критической ошибки.
<code>-quiet</code>	То же, что и <code>--silent</code> .
<code>-r</code>	То же, что и <code>--no-builtin-rules</code> .
<code>-R</code>	То же, что и <code>--no-builtin-variables</code> .
<code>--recon</code>	То же, что и <code>--dry-run</code> .
<code>-s</code>	То же, что и <code>--silent</code> .
<code>-S</code>	То же, что и <code>--stop</code> .
<code>--silent</code>	Подавляет обычный вывод строки команды при ее выполнении.
<code>--stop</code>	Отменяет действие опции <code>--keep-going</code> .
<code>-t</code>	То же, что и <code>--touch</code> .
<code>--touch</code>	Обновляет атрибуты времени целевых файлов для их соответствия последней версии, но при этом не выполняет команд для создания их новых версий.

Опция	Описание
-v	Выводит информацию о версии утилиты <code>make</code> и завершает выполнение утилиты.
--version	Выводит информацию о версии и завершает выполнение утилиты.
-w	То же, что --print-directory .
-W filename	То же, что --assume-new .
--warn-undefined-variable	Выводит предупреждение для каждой ссылки на переменную, которая не была определена.
--what-if=filename	То же, что --assume-new .

В качестве своей команды процесс `make` может запускать другой такой же процесс. В таком случае опции, установленные для родительского экземпляра процесса, передаются и дочернему экземпляру. Ввиду этого имеются опции, позволяющие восстанавливать используемые по умолчанию настройки, которые могут быть включены в команду запуска дочернего процесса `make`. Другая причина применения опций восстановления настроек по умолчанию состоит в том, что настройки могут быть изменены с помощью переменной окружения `MAKEFLAGS`.

Утилита Autoconf

`Autoconf` — это утилита, которая создает сценарии установки для включения их в распространяемый исходный код. По умолчанию вырабатываемый сценарий командной оболочки называется `configure`. Он выполняется независимо, поэтому для настройки и установки программного обеспечения устанавливать в целевой системе утилиту `Autoconf` не требуется.

Использование утилиты `Autoconf` для упаковки и организации распространения программного обеспечения дает несколько преимуществ. Сценарий `configure` проверит наличие или присутствие некоторых системных возможностей и сформирует компоновочные `make`-файлы, учитывающие свойства текущей среды. Это означает, что приложение может быть перенесено практически на любую версию UNIX. Метод установки программного обеспечения с помощью используемого для настройки компиляции сценария `configure` получил широкое распространение и хорошо знаком многим пользователям программ с открытым исходным кодом. Для установки программного обеспечения, пакетированного с помощью утилиты `Autoconf`, как правило, выполняются следующие команды:

```
$ ./configure
$ make
$ make install
```

Утилита `Autoconf` в действительности является набором инструментов, перечисленных в таблице 14.2.

В зависимости от сложности приложения и требуемой степени его переносимости процесс создания установочных сценариев может изменяться от достаточно простой процедуры до очень сложной. В любом случае в качестве общего руководства может использоваться приведенная ниже последовательность действий.

Таблица 14.2. Инструменты утилиты Autoconf

Инструмент	Описание
<code>autoconf</code>	На основе файла шаблона, используемого в качестве исходных данных, этот инструмент генерирует сценарий настройки конфигурации. Тот, в свою очередь, будет генерировать <code>make</code> -файлы и конфигурационные сценарии для текущей (или указанной) платформы.
<code>autoheader</code>	Эта программа создает файл шаблона, содержащий операторы <code>#include</code> , который будет использоваться сценарием <code>configure</code> , созданным с помощью инструмента <code>autoconf</code> .
<code>autoreconf</code>	Программа, обновляющая сценарии настройки конфигурации. Она запускает программы <code>autoconf</code> только в тех каталогах, где атрибут даты/времени файлов указывает на необходимость обновления.
<code>autoscan</code>	Эта программа сканирует файлы исходного кода в дереве исходных каталогов и генерирует предварительную версию файла шаблона, который будет использоваться в качестве входного файла для <code>autoconf</code> .
<code>autoupdate</code>	Данная программа обновляет существующий файл шаблона для приведения его в соответствие с текущей версией <code>autoconf</code> .
<code>ifnames</code>	Эта программа сканирует все файлы исходного кода на языке C и имена, содержащиеся в директивах препроцессора <code>#if</code> , <code>#elif</code> , <code>#ifdef</code> и <code>#ifndef</code> . Список выводится в упорядоченном виде, для каждого имени приводится список файлов, где это имя было обнаружено.

Перейдите в каталог, где хранятся файлы исходного кода, и выполните следующие действия:

1. Установите параметры условной компиляции. Для обеспечения переносимости программного обеспечения часто в заголовочных файлах используются директивы препроцессора. Для сбора информации по условной компиляции запустите программу `ifnames` для исходных файлов, которые будут обрабатываться препроцессором. Например, следующая команда обработает все исходные файлы на языке C и все включаемые в них заголовочные файлы:

```
$ ifnames *.c, *.h
```

Результат выполнения приведенной команды представляет собой список имен макросов, определенных с условиями, и файлов, в которых эти макросы определены.

2. Создайте файл `configure.in`. В исходном каталоге из командной строки запустите утилиту `autoscan` без аргументов:

```
$ autoscan
```

Результатом выполнения `autoscan` будет файл `configure.scan`, который будет в дальнейшем использован в качестве шаблона для создания окончательной версии сценария `configure`. Скопируйте (или переместите) файлы `configure.scan` и `configure.in`, это потребует внести в них соответствующие поправки.

3. Отредактируйте файл `configure.in`. Это основная часть задачи. Файл `configure.in` состоит из макро-директив `m4`, предназначенных для обработки утилитой `Autoconf` при генерации окончательной версии скрипта `configure`. Если процесс установки становится сложнее, и только макросы

уже не могут им управлять, в сценарий можно включить фрагменты сценария оболочки. Эти фрагменты будут скопированы в окончательную версию сценария `configure`.

Исходный скрипт `configure.in` содержит большое количество макросов, необходимых для окончательной версии, а также целый ряд описательных комментариев (которые начинаются с символа "#"). При внесении в файл изменений старайтесь добавлять свои комментарии. В таблице 14.3 приведено описание информации, необходимой для определения различных макросов.

Таблица 14.3. Макросы m4, используемые в сценарии `configure.in`

Макрос	Описание
<code>AC_C_CHAR_UNSIGNED</code>	Этот макрос проверяет тип <code>char</code> по умолчанию и определяет макрос <code>_CHAR_UNSIGNED_</code> , если тип не содержит знака.
<code>AC_C_CONST</code>	Макрос проверяет метод обработки компилятором С ключевого слова <code>const</code> и при необходимости переопределяет его.
<code>AC_CHECK_FUNCS</code>	Этот макрос проверяет наличие указанных в списке функций. Имена функций разделяются запятыми.
<code>AC_CHECK_HEADERS</code>	Проверяет наличие одного или большего количества заголовочных файлов, содержащихся в списке и разделенных запятыми.
<code>AC_CHECK_LIB</code>	Этот макрос проверяет наличие указанных в списке библиотек. Имена библиотек указываются в короткой форме. Функции, входящие в состав библиотеки, должны также быть указаны для проверки. Например, библиотека <code>libcfont</code> должна содержать функцию <code>bdf</code> , если указано: <code>AC_CHECK_LIB(cfond, bdf)</code>
<code>AC_CONFIG_AUX_DIR</code>	Макрос указывает имя каталога, содержащего <code>install-sh</code> , <code>config.sub</code> и <code>config.guess</code> . Как правило, каталоги по умолчанию обычно корректны. Но для предотвращения ошибок может применяться этот макрос для указания абсолютного или относительного пути к этим файлам.
<code>AC_CONFIG_HEADER</code>	Макрос составляет заголовочные файлы, содержащие директивы <code>#define</code> . После имени создаваемого файла указывается двоеточие и имя входного файла, содержащего директивы. Например, заголовочный файл <code>config.h</code> создается на основе содержимого файла <code>config.in</code> с помощью макроса: <code>AC_CONFIG_HEADER(config.h: config.in)</code>
<code>AC_CONFIG_SUBDIR</code>	Указывает список каталогов, которые могут содержать сценарии <code>configure</code> , запускаемые генерируемым сценарием. Имена каталогов разделяются пробелами.
<code>AC_FUNC_MEMCMP</code>	Макрос проверяет правильность работы функции <code>memcmp()</code> с 8-битным выравниванием.
<code>AC_FUNC_STRFTIME</code>	Этот макрос проверяет правильность работы функции <code>strftime()</code> .
<code>AC_FUNC_VPRINTF</code>	Макрос проверяет наличие функции <code>vprintf()</code> .
<code>AC_HEADER_STDC</code>	Проверяет наличие в системе стандартных заголовочных файлов языка C.
<code>AC_HEADER_SYS_WAIT</code>	Этот макрос проверяет наличие заголовочного файла <code>sys/wait.h</code> , совместимого с POSIX.
<code>AC_HEADER_TIME</code>	Этот макрос подтверждает, что заголовочные файлы <code>time.h</code> и <code>sys/time.h</code> могут быть включены в один и тот же блок компиляции.

Макрос	Описание
<code>AC_INIT</code>	Этот макрос должен быть первым. Он содержит имя уникально названного файла в качестве проверки того, что пользователь запустил сценарий в нужном каталоге, например: <code>AC_INIT(hello.c)</code>
<code>AC_OUTPUT</code>	Вторым обязательным макросом является <code>AC_OUTPUT</code> . Этот макрос обязателен. Он именует и создает make-файл, иногда и некоторые другие выходные файлы. Если в него включить дополнительные аргументы, то это будут команды, которые добавляются в сценарий <code>config.status</code> для выполнения их после выполнения всех остальных команд. Как правило, макрос используется в таком виде: <code>AC_OUTPUT(Makefile)</code>
<code>AC_OUTPUT_COMMANDS</code>	Вторым обязательным макросом является <code>AC_INIT</code> . Указывает дополнительные команды, которые будут выполняться в конце сценария <code>config.status</code> . Он может использоваться повторно, например: <code>AC_OUTPUT_COMMANDS(echo An extra command)</code>
<code>AC_PREFIX_DEFAULT</code>	Этот макрос указывает префикс установки вместо используемого по умолчанию <code>/usr/local</code> , например: <code>AC_PREFIX_DEFAULT(/home/fred/sets)</code>
<code>AC_PREFIX_PROGRAM</code>	Если пользователь не установил префикс с помощью опции <code>-prefix</code> , то этот макрос будет искать указанную программу с помощью переменной <code>PATH</code> и установит значение префикса равным каталогу, содержащему программу.
<code>AC_PREREQ</code>	Этот макрос гарантирует, что используется достаточно новая версия <code>Autoconf</code> . Например, следующий макрос проверит, что используется версия 1.8 или выше: <code>AC_PREREQ(1.8)</code>
<code>AC_PROG_MAKE_SET</code>	Предопределяет переменную <code>MAKE</code> так, как если бы была установлена переменная окружения <code>MAKE=make</code> .
<code>AC_REVISION</code>	Копирует указанную информацию в сценарий <code>configure</code> .
<code>AC_TYPE_OFF_T</code>	Проверяет наличие указанных операторов <code>typedef</code> и определяет пропущенные операторы.
<code>AC_TYPE_SIZE_T</code>	Этот макрос также проверяет наличие указанных операторов <code>typedef</code> и определяет пропущенные типы.

Каждый макрос содержит список элементов, разделенных запятыми, в следующем формате:

`AC_CHECK_LIB(dl, dlopen, socket)`

Между именем макроса и открывающей скобкой не должно быть пробелов. Аргументы могут быть заключены в квадратные скобки. В случае если список аргументов занимает более одной строки, то он обязательно должен быть заключен в квадратные скобки.

4. Создайте `makefile.in`. Чтобы воспользоваться преимуществами конфигурирования утилитой `Autoconf`, необходимо изменить содержимое компоновочного файла. Его следует переименовать в `makefile.in` и включить в него сценарии, генерированные средством `Autoconf`. Некоторые часто встречающиеся определения приведены в таблице 14.4.

Таблица 14.4. Ключевые слова, определяемые утилитой Autoconf для make-файлов

Ключевое слово	Описание
<code>@CC@</code>	Компилятор C.
<code>@CFLAGS@</code>	Набор флагов, передаваемых компилятору C.
<code>@CPP@</code>	Препроцессор C.
<code>@CPPFLAGS@</code>	Набор флагов, передаваемых препроцессору C.
<code>@CXX@</code>	Компилятор C++.
<code>@CXXFLAGS@</code>	Набор флагов, передаваемых компилятору C++.
<code>@DEFINES@</code>	Как правило, при использовании макроса <code>AC_CONFIG_HEADER</code> определяется как <code>-DHAVE_CONFIG_H</code> .
<code>@INSTALL@</code>	Утилита <code>install</code> или сценарий <code>install-sh</code> .
<code>@LDFLAGS@</code>	Флаги, передаваемые компоновщику.
<code>@LIBOBJS@</code>	Объектные файлы, включаемые при компоновке программ.
<code>@LIBS@</code>	Библиотеки, используемые при компоновке программ.
<code>@RANLIB@</code>	Утилита <code>ranlib</code> .
<code>@SET_MAKE@</code>	Как правило, имеет значение "MAKE=make".
<code>@srcdir@</code>	Имя каталога, содержащего исходные файлы.

5. **Создайте config.h.in.** Самый простой способ создания конфигурационного заголовочного файла — запуск утилиты `autoheader` и создание с ее помощью файла `config.h.in`. Последний используется в качестве входного файла при построении файла `config.h`. Для этого введите команду без аргументов:

```
$ autoheader
```

6. **Обновите исходный код.** Во все файлы исходного кода, для которых перенос программы может иметь значение, необходимо директивой `#include` включить заголовочный файл `config.h`. Это обеспечивает выполнение условной компиляции в соответствии с параметрами среды. Например, при отсутствии стандартных заголовочных файлов языка C вам, возможно, придется изменить порядок установки:

```
#ifdef STDC_HEADERS
    /* Компилируется только в случае отсутствия
       стандартных заголовочных файлов С      */
#endif
```

7. **Создайте сценарий установки.** С помощью следующей команды утилита `autoconf` считывает файл `configure.in` и вырабатывает сценарий `configure`:

```
$ autoconf
```

8. **Скопируйте сценарии Autoconf.** Следующие три сценария должны быть включены в пакет установки. Они входят в состав `Autoconf` и, как правило, находятся в каталоге `/usr/lib/autoconf` или `/usr/share/automake`:

```
config.guess
config.sub
install-sh
```



Глава 15

Ассемблер GNU

Ассемблер GNU составлен из довольно большого набора ассемблеров. Он поддерживает множество разнообразных платформ. Несмотря на то, что каждый отдельный ассемблер имеет свои особенности, основной набор директив — общий для всех. Переносу программ способствует также и то, что формат мнемонических инструкций машинных операционных кодов (opcodes) для каждого семейства платформ мало изменяется от одной версии платформы к следующей ее версии.

Ассемблер проекта GNU разработан для ассемблирования выходного кода компилятора в объектный код для дальнейшей передачи компоновщику. Поэтому он интегрирован с компилятором GNU и обычно действует на заднем плане. Но иногда могут возникать и такие обстоятельства, когда нужно поработать непосредственно с ассемблерным кодом.

Управление ассемблированием из командной строки

Когда программа пишется на языке высокого уровня, GCC обычно сам задействует ассемблер соответствующей платформы. При этом редко приходится применять опции команд `gcc`, связанные с ассемблированием. Однако если понадобится написать отдельный модуль на языке ассемблера для особых целей, то придется применить некоторые специальные опции из тех, что приведены в таблице 15.1.

Если требуется написать на ассемблере отдельный модуль, то для начала лучше написать на языке C простую программу, которая содержит все необходимые структурные элементы, и затем применить `gcc` с опцией `-S`, чтобы сгенерировать начальный исходный модуль на ассемблере. Написание программ на ассемблере — трудоемкий процесс, и при этом трудно избежать ошибок. Поэтому лучше начинать с применения такого испытанного генератора ассемблерного кода, как GCC.

Таблица 15.1. Опции командной строки `gcc` для управления ассемблером GNU

Опция	Описание
<code>-a [opts] [=file]</code>	Включает вывод листинга в файл с именем <code>file</code> . С этой опцией в поле <code>opts</code> можно использовать одну из приведенных ниже букв или их сочетание для указания формата и содержания вывода. По умолчанию эта опция применяется как сочетание <code>-ahl</code> . Листинг обычно направляется на стандартный выход, но его можно перенаправить в файл, если указать имя файла в параметре опции, например, так: <code>-ahl=asembly.list</code> . Значение буквенных кодов флагов этой опции:
	с — не выводит код, пропускаемый по условию;
	д — пропускает все отладочные директивы;
	h — включает в вывод исходный код языка высокого уровня;
	l — включает в вывод дамп асsembлированного кода в шестнадцатиричном формате;
	ль — выводит статистику построчной отладки;
	м — включает в вывод макро-расширения;
	п — не выводит результаты обработки форм;
	в — выводит таблицу перекрестных ссылок программных символов.
<code>--defsym symbol=value</code>	Определяет символ с именем <code>symbol</code> и назначает ему значение <code>value</code> .
<code>-f</code>	Пропускает предобработку пустых строк и комментариев.
<code>--fatal-warnings</code>	Воспринимает предупредительные сообщения как ошибки.
<code>--gdwarf2</code>	Генерирует отладочную информацию в формате DWARF2 и помещает ее в объектном файле.
<code>--glibs</code>	Генерирует отладочную информацию в формате STABS и помещает ее в объектном файле.
<code>--help</code>	Выводит список опций и на этом завершает работу программы.
<code>-I directory</code>	Добавляет указанное имя каталога к списку расположений для поиска файлов, включаемых по директиве <code>.include</code> .
<code>-J</code>	Указывает не выдавать предупреждений при переполнении знаковых целочисленных переменных (<code>signed overflow</code>).
<code>-K</code>	Выдает предупредительные сообщения при изменениях в таблице смещений (differences table). Эта таблица содержит абсолютные величины, получаемые из пары перемещаемых адресов с помощью вычитания. Она должна обновляться каждый раз при установке новых значений адресов.
<code>--keep-locals</code>	Сохраняет записи таблицы символов (symbol table entries) для локально определенных символов, в исходном листинге они начинаются с последовательности <code>".L"</code> .
<code>-L</code>	То же, что и <code>--keep-locals</code> .
<code>-M</code>	То же, что и <code>--mri</code> .
<code>-MD filename</code>	Информация зависимостей в формате, допускающем ее помещение в компоновочный сценарий (makefile). Выводится в файл с именем, указанным в поле <code>filename</code> .
<code>--mri</code>	Компилирует в режиме совместимости с MRI. Это означает, что процесс асsembлирования воспринимает синтаксис стандартного ассемблера, поставляемого компанией Microtec Research.

Опция	Описание
--no-warn	Подавляет вывод любых предупредительных сообщений. То же, что и --w.
-o <i>filename</i>	Назначает имя выходного файла.
-R	Помещает код из раздела DATA в разделе ТЕХТ.
--statistics	Показывает общее время ассемблирования и объем памяти, задействованной процессом ассемблера.
--strip-local-absolute	Любые символы, локальные для данной сессии ассемблирования и имеющие постоянное значение, удаляются. Ссылки на них заменяются литералами с их значением.
--traditional-format	Назначает вывод в формате ассемблера, комплектного применяемой системе.
--target-help	Выводит список опций, специфичных для целевой платформы, и на этом завершает работу программы.
--version	Выводит номер версии и завершает работу программы.
-w	Подавляет вывод всех предупредительных сообщений. То же, что и --no-warn.

Если нет необходимости делать большой кусок работы на ассемблере, то лучше сделать вставку ассемблерного кода в программу на языке более высокого уровня. Этот вопрос будет подробно рассмотрен далее в этой главе.

Абсолютная и относительная адресация, выравнивание адресов

Многие действия в языке ассемблера непосредственно связаны с адресами и с вычислением адресов. Адреса ссылок, указываемые в ассемблерном коде как символические имена расположений памяти, вычисляются и подставляются ассемблером. Например, следующая инструкция `jle` (локальный переход к указанному адресу) передает управление на строку, которая следует за меткой `.L3`:

```
add $16,%esp
jle .L3
call function
.L3
movl $0,%eax
```

Расположение кода, отмеченное как `.L3`, не является *абсолютной* числовой величиной. Компоновщик будет изменять его значение при каждой компоновке исполняемой программы. Так что адрес метки `.L3` — относительная величина. Она может быть определена только как смещение относительно начала модуля. Компоновщик заменяет значения всех ссылок их *относительными* адресами, так же как и аргументную ссылку оператора `jle` в рассмотренном выше примере.

Абсолютное выражение имеет постоянное, неизменяемое при компоновке значение. Оно может представлять собой постоянную величину, либо вычисляться как результат выражения. Возможно вычисление абсолютного значения и из относительных величин при выполнении действий над относительными адресами. Например,

следующее выражение дает в результате абсолютную величину, потому что смещение между двумя указанными расположениями кода будет постоянным:

`.L6 - .L3`

Компоновщик будет перемещать расположение как `.L6`, так и `.L3`, но при этом смещение между ними будет постоянным. В следующем примере результат выражения будет относительным к положению метки `.L44`, потому что в выражении вычисляется абсолютная величина и складывается с относительным адресом метки `.L44`:

`.L44 + .L6 - .L3`

Некоторые выражения с адресной арифметикой не могут быть однозначно определены (ill defined expressions). Например, результатом следующего выражения будет число, не имеющее смысла. Оно является функцией начального адреса размещения модуля в памяти, которое выбирается компоновщиком.

`.L6 + .L3`

Следующим важным положением применения ассемблера является *выравнивание адреса* (address boundary). Если адрес является числом, кратным 16 (т.е. деление этого числа на 16 не дает остатка), то мы говорим, что этот адрес имеет выравнивание по границе 16-бит, или 16-битное выравнивание. Это может иметь важное значение для некоторых структур данных и для инструкций. В некоторых случаях от этого зависит эффективность программы, но чаще всего применение определенного выравнивания диктуется требованиями аппаратуры. Такие директивы ассемблера, как `.org` и `.align`, используются для заполнения указанным в них значением нужного количества байт для обеспечения требуемого выравнивания адреса следующей инструкции. Разумеется, при этом необходимо, чтобы при компоновке программы выравнивание применялось и к начальному адресу модуля. Только тогда внутреннее выравнивание кода и элементов данных будет корректно соответствовать применяемым требованиям.

Вставка ассемблерного кода

Необходимость непосредственного включения в программу ассемблерного кода может возникнуть по целому ряду причин. Но при этом не обязательно писать на ассемблере всю программу и даже отдельный модуль. Если необходимо сделать что-то особенное на машинном уровне, то обычно самым простым выходом является включение в программу на языке более высокого уровня фрагмента ассемблерного кода. Для этого GCC предоставляет возможность непосредственного включения команд ассемблера в функции на языке C.

По самой своей природе код на языке ассемблера не может быть переносимым. Код, который написан для одной платформы, практически всегда для любой другой платформы будет неправильным. В этом разделе при рассмотрении процедуры вставки ассемблерного кода в программу мы будем использовать синтаксис инструкций, совместимый с ассемблерами семейства процессоров Intel.

Конструкция `asm`

В следующем примере программы на языке *C* для вставки фрагмента ассемблерного кода используется конструкция `asm`. В этом примере значение переменной, объявленной в коде на языке *C*, загружается в регистр, сдвигается на один бит вправо для его деления на 2, и результат затем записывается в другую переменную *C*.

```
/* half.c */
#include <stdio.h>
int main(int argc,char *argv[])
{
    int a = 40;
    int b;

    asm("movl %1,%eax; \
        shr %eax; \
        movl %eax,%0;" \
        : "=r"(b) \
        : "r"(a) \
        : "%eax");

    printf("a=%d b=%d\n",a,b);
    return(0);
}
```

Конструкция `asm` — нечто большее, чем просто удобный способ вставки ассемблерного кода. Она позволяет использовать синтаксис языка *C* для обращения к переменным. Кроме того, она дает возможность передавать управляющую информацию на этапах генерации кода и его дальнейшей оптимизации. Таким образом, вы можете генерировать эффективный код, который учитывает ваши особые требования к оптимизации программы.

Синтаксис конструкции `asm` имеет следующий формат:

```
asm( "шаблон кода на ассемблере"
      : список выходных operandов
      : список входных operandов
      : список используемых регистров);
```

При желании предотвратить попытки компилятора оптимизировать ассемблерный код следует воспользоваться ключевым словом `volatile`. Примерно так:

```
asm volatile ( . . .
```

Для обеспечения совместимости с POSIX можно использовать `__asm__` и `__volatile__` вместо ключевых слов `asm` и `volatile`.

Шаблоны ассемблерного кода

Шаблон ассемблерного кода состоит из одного или более операторов на языке ассемблера и представляет собой действующий код, непосредственно подставляемый компилятором в объектный модуль. Инструкции могут адресоваться: к непосредственно разрешаемым значениям (константам); к содержимому регистров; к расположениям в памяти. Далее следуют синтаксические правила, которые применяются к адресуемым величинам:

- Перед именем регистра ставятся два знака процента — "%%". Например, `%%eax` или `%%esi`. Имена регистров процессоров Intel начинаются со знака "%", а применение второго знака "%" требуют правила конструкции `asm`, так что в нашем случае их должно быть два.
- Адресуемым расположением данных в памяти является один из входных или выходных операндов. Каждый из них указывается порядковым индексом, соответствующим его положению в строке объявления. Первый входной операнд адресуется в шаблоне кода сочетанием `%0`, второй входной операнд — `%1` и т.д. Индексы выходных операндов продолжают эту нумерацию. Например, при наличии двух входных операндов первый выходной операнд в шаблоне будет адресован как `%2`.
- Расположение в памяти может быть указано его адресом, записанным в регистре процессора. В этом случае имя регистра заключается в круглые скобки. Например, нужно загрузить в регистр `%%al` один байт из ячейки памяти, адресом которой является число, записанное в регистре `%%esi`. Инструкцию для этого действия можно записать так:

```
movb (%%esi),%%al
```

- Непосредственное значение (константа) обозначается знаком доллара "\$", сразу за которым следует само число. Например: `$86` или `$0xF12A`.
- Весь фрагмент ассемблерного кода является одной буквенной строкой (character string). Поэтому каждая строка вставляемого кода должна заканчиваться терминатором — символом перехода на следующую строку. В качестве терминаатора можно использовать наклонную черту "\\" (semicolon) или escape-последовательность перехода на следующую строку — "\n". Для улучшения читаемости кода допускается вставка символов табуляции (tabs).

Входные и выходные операнды

Входные и выходные операнды состоят из списка переменных, которые должны быть доступными для адресации из вашего ассемблерного кода. Для указания адреса памяти можно использовать любые выражения языка C, определяемые как адрес расположения. Следующий пример программы — вариация на тему предыдущего. Эта программа удваивает числа, сдвигая влево их двоичное представление, и использует массив для хранения входных и выходных величин.

```
/* double.c */
#include <stdio.h>
int main(int argc,char *argv[])
{
    int array[2];
    array[0] = 150;
    int i = 0;

    asm("movl %1,%eax; \
        shl %%eax; \
        movl %%eax,%0;" \
        :"=r"(array[i+1])
```

```
: "r"(array[i])
: "%eax";
printf("array[0]=%d array[1]=%d\n", array[0], array[1]);
return(0);
}
```

Далее приводятся правила синтаксиса, которые применяются к входным и выходным операндам:

- Выражение языка C, указывающее на адрес данных программы, должно быть заключено в круглые скобки.
- Если перед указываемыми расположениями ставится признак ограничения (constraint) "r", то он означает, что данные должны быть загружены из входных переменных в свободные регистры до выполнения ассемблерного кода. Применяется во входных операндах. В выходных операндах соответственно применяется признак ограничения "=r" — он указывает, что значения в выходные переменные должны быть записаны после выполнения ассемблерного кода.
- Признак ограничения может указывать назначение для переменных C определенных регистров:

```
"a"    %%eax
"b"    %%ebx
"c"    %%ecx
"d"    %%edx
"s"    %%esi
"t"    %%edi
```

- Переменные могут быть адресованы из ассемблерного кода своим адресом памяти без их загрузки в регистры процессора, для этого ставится признак ограничения (constraint) "m".
- Одна и та же переменная может использоваться и для ввода входного значения, и для вывода результата. При этом для этой переменной в выходном операнде применяется признак ограничения (constraint) "=a", а во входном операнде в признаке ограничения ставится его порядковый индекс. В следующем примере переменная из кода на языке C `counter` используется как для ввода, так и для вывода:

```
asm("incw %0;"  
    : "=a"(counter)  
    : "0"(counter));
```

- Возможно использование любого количества входных и выходных операндов. В списке они разделяются запятыми.
- Все выходные и, затем, входные операнды нумеруются последовательно от \$0 до \$n-1, где n — общее число всех входных и выходных операндов. Например, если операндов в конструкции всего 6, то последний будет иметь адресуемое из ассемблерного кода имя \$5.

Список используемых регистров

Список регистров, задействованных в ассемблерном коде, (*clobbered registers*) — это просто список имен регистров. Между именами ставится запятая. Например:

```
asm( . . .
     . . .
     : "%eax", "%esi");
```

Информация о регистрах передается компилятору. Благодаря этому исключаются ошибки, связанные с использованием этих регистров.

Директивы ассемблера GNU

Основным назначением ассемблера является перевод мнемонических инструкций ассемблерного языка (*mnemonic opcodes*) в двоичные машинные команды (*binary opcodes*), которые уже могут непосредственно выполняться аппаратной частью или использоваться для хранения данных. В дополнение к этому ассемблер способен воспринимать и обрабатывать *директивы ассемблера* (*assembler directives*). Они могут использоваться для применения выравнивания к адресам инструкций, определения макрорасширений, разбиения кода на именованные разделы, объявления именованных констант, обеспечения условного ассемблирования или как упрощенный способ определения данных.

Дальше приводится список директив ассемблера GNU. Каждая директива начинается с точки. Некоторые директивы применяются сами по себе. Другие имеют параметры, которые должны стоять в одной строке с именем директивы. И некоторые из директив могут содержать блок ассемблерного кода до соответствующей им закрывающей этот блок директивы. Некоторые из директив, в частности, те, которые используются для размещения отладочной информации в объектном коде, применяются только к одному или к нескольким форматам объектных файлов.

Некоторые из директив распознаются ассемблером, но не вызывают никаких действий. В эту категорию скоро должна войти директива `.abort`. Она пока еще служит для отмены ассемблирования, но в следующей версии она уже не будет поддерживаться. Такие директивы, как `.file`, `.app-file`, `.extern`, `.ident` и `.lflags`, распознаются ассемблером, но никаких действий по ним не предпринимается.

Некоторые директивы могут иногда странно себя вести. Применение таких директив связано с историей. Некоторые устаревшие решения продолжают поддерживаться многими ассемблерами в целях обеспечения переносимости существующих программ. О таких директивах нельзя сказать, что они бесполезны, — просто при некоторых условиях на более новых платформах они могут вызывать непредсказуемые явления. Примером такой директивы в предлагаемом списке может служить `.fill`.

Многие директивы из списка используют или объявляют программные *символы*. *Символом* в этом контексте называется имя, имеющее связанные с ним атрибуты `value` (значение адреса) и `type` (тип). Значением атрибута `value` может быть как абсолютная, так и относительная адресная величина. Атрибут `type` определяет размер и способ использования адресуемых данных.

.align *boundary* [,*filler*] [,*maximum*]

Заполняет нужное количество байт значением *filler* для применения заданного в поле *boundary* выравнивания адреса инструкции, следующей за этой директивой. Все три аргумента директивы являются абсолютными выражениями. Если значение *filler* не указано, то по умолчанию происходит заполнение нулевыми байтами для разделов данных и "пустыми" операционными кодами *поор* для разделов исполняемого кода. Если количество заполняющих байтов, требуемое для выравнивания следующей инструкции, превышает *maximum*, то по этой директиве ассемблер не предпринимает никаких действий.

Аргументы *filler* и *maximum* являются необязательными. Для применения аргумента *maximum* без указания *filler* следует ставить две запятые.

Точный формат этой директивы непостоянный. Из-за того, что ассемблер GNU поддерживает эмуляцию "родных" ассемблеров на целом ряде довольно разнообразных платформ, поле *boundary* может интерпретироваться весьма своеобразно. Например, на одних системах 8-битное выравнивание может указываться как *.align 8* (т.е. адрес должен быть кратным 8-ми), в то время как на других — *.align 3* (число здесь указывает 3 нулевых младших бита в двоичном представлении адреса). Для использования более переносимого синтаксиса лучше использовать директиву *.balign* или *.p2align*.

.ascii [*string*] [,*string* ...]

Ассемблирует ноль или большее количество указанных текстовых строк в текстовый раздел ASCII. Размещаемые по этой директиве строки не имеют завершающего нулевого байта.

.asciz [*string*] [,*string* ...]

Ассемблирует ноль или более текстовых строк в текстовый раздел ASCII. К размещаемым по этой директиве строкам добавляется нулевой байт.

.balign *boundary* [,*filler*] [,*maximum*]

Вставляет байты *filler* для применения указанного аргументом *boundary* выравнивания адреса следующей инструкции. Все три аргумента — абсолютные выражения. Если *filler* не указан, то по умолчанию для разделов данных применяется нулевой байт, а для разделов выполняемых инструкций — "пустой" оператор *поор*. Если нужное количество заполняющих байтов превышает *maximum*, то никаких действий не предпринимается.

Аргументы *filler* и *maximum* являются необязательными. Для применения аргумента *maximum* без указания *filler* следует ставить две запятые.

.balignl *boundary* [,*filler*] [,*maximum*]

Такая же директива, как и *.balign*, за исключением того, что *filler* имеет фиксированную длину 32 бит (тип *long*).

.balignw *boundary* [,*filler*] [,*maximum*]

Действует так же, как директива *.balign*, кроме того, что *filler* имеет длину 16 бит.

.byte expression [,expression ...]

Для каждого выражения *expression* размещается один байт, и в него записывается результат выражения.

.comm symbol, length

По этой директиве резервируется блок памяти, имеющий длину *length* байт. Этому блоку присваивается имя *symbol*. В случае, когда несколько модулей имеют объявления одинаковых имен символов, то их блоки при компоновке объединяются. Если длина адресуемых такими символами блоков отличается, то при их объединении выбирается наибольшее значение.

На системах ELF поддерживается третий необязательный аргумент, указывающий выравнивание. На системах HPPA эта директива имеет следующий синтаксис: *symbol .comm length*.

.data subsection

Операторы, которые следуют за этой директивой, ассемблируются в подраздел с номером, задаваемым аргументом *subsection*. Он должен быть абсолютным выражением. По умолчанию *subsection* имеет нулевое значение.

.def name

Эта директива открывает блок, содержащий отладочную информацию для размещения в объектном коде формата COFF. Ему присваивается имя *name*. Блок продолжается до закрывающей его директивы *.endef*. См. также *.dim*, *.scl*, *.tag*, *.type*, *.val* и *.size*.

.desc symbol, value

Определяет программный символ с именем *symbol* и присваивает ему значение *value*. Аргумент *value* должен быть абсолютным выражением. Эта директива не генерирует выходной информации для размещения в объектных файлах формата COFF.

.dim

Эта директива может использоваться только в блоках между парой директив *.def* и *.endef*. Она используется для включения дополнительной информации в символьные таблицы объектных файлов формата COFF.

.double value [,value ...]

Для каждого указанного значения *value* ассемблируется и записывается в память число с плавающей точкой. Внутреннее представление чисел с плавающей точкой (включая их размер и диапазон значений) зависит от платформы. Смотри также *.float*.

.eject

Включает в выходной листинг ассемблера символы конца страницы.

.else

См. **.if**.

.endifef

См. **.def**.

.endif

См. **.if**.

.equ symbol, value

Определяет программный символ *symbol* со значением *value*. Аргумент *value* может быть как абсолютным, так и относительным выражением. Директива **.equ** может применяться к одному символу сколько угодно, каждый раз изменяя его значение. На системах HPPA имеет синтаксис *symbol .equ value*. Эта директива равносильна **.set**. См. также **.equiv**.

.equiv symbol, value

Действует так же, как директивы **.equ** и **.set**. Отличается тем, что при применении этой директивы генерируется сообщение об ошибке в том случае, если символ ранее уже был определен.

.err

Директива **.err** генерирует сообщение об ошибке, даже если в командной строке применена опция **-z**. Кроме того, она отменяет выработку объектного файла. Обычно используется внутри блока условного ассемблирования для указания ошибки. В следующем примере эта директива выдает ошибку, если не определен символ с именем **BLACKLINE**:

```
.ifndef BLACKLINE
,err
#endif
```

.fill repeat, size, value

Эта директива ассемблера создает нужное количество блоков данных, каждый из которых имеет длину до 8 байт. Значением **repeat** должно быть абсолютное выражение, оно задает количество создаваемых блоков. Значением аргумента **size** может быть любое абсолютное значение, но если оно превышает число 8, то воспринимается как равное восьми. Аргумент **size** определяет число байт в каждом блоке.

Аргумент этой директивы **value** используется для заполнения каждого 8-байтного блока создаваемого директивой массива. Заполнение младших 4-х байт каждого блока определяется из значения аргумента **value**, которое считывается последовательно по байтам. Каждые считанные 4 байта преобразовываются к аппаратному представлению 32-битного двоичного целого числа, которое и записывается в блок. Представление записываемого числа зависит от платформы. Блоки создаваемого массива заполняются последовательно требуемым количеством байт, считываемых из значения **value**.

Если размер массива ***size*** не указан, то по умолчанию ему присваивается значение 1. Если не указан аргумент ***value***, то считается, что он равен нулю. См. также **.org** и **.p2align**.

.float *value*[,*value* ...]

Для каждого указанного значения ***value*** ассемблируется число с плавающей точкой (тип **float**). Внутреннее представление таких чисел, включая их размер и диапазон значений, зависит от платформы. См. также **.double**.

.global *symbol*

Указанный этой директивой символ становится глобальным, т.е. известным компоновщику. Директива не определяет символ, он может быть объявлен в любом месте программы. Программный символ можно объявить в одном модуле программы, при компоновке программы во всех остальных модулях ссылки на него являются разрешимыми.

.globl

Синоним директивы **.global**.

.hword *value*[,*value* ...]

Для каждого указанного ***value*** резервируется 16 бит, и в это расположение записывается значение ***value***, преобразованное к 16-битному двоичному целому числу. В зависимости от платформы эта директива может быть равнозначна **.short** или **.word**.

.if *expression*

Строки, которые следуют за этой директивой, ассемблируются, только если результат абсолютного выражения ***expression*** не равен нулю. Конец раздела условно ассемблируемого кода отмечает директива **.endif**. Например, следующие две инструкции будут ассемблироваться только в том случае, когда значения **topside** и **current** равны:

```
.if topside - current
    pushl %ebp
    movl %esp, %bp
.endif
```

Необязательная директива **.else** выделяет блок альтернативного кода, который ассемблируется, только когда выражение ***expression*** директивы **.if** ложно (т.е. равно нулю). В этом примере выполняемая инструкция зависит от значения **ENTERING**:

```
.if ENTERING
    pushl %ebp
.else
    popl %ebp
.endif
```

Директива `.if` имеет формы `.ifdef`, `.ifndef` и `.ifnotdef`, используемые для проверки определенности символов.

`.ifdef symbol`

Условное ассемблирование выполняется, кода символ с именем `symbol` определен. См. `.if`.

`.ifndef symbol`

Условное ассемблирование выполняется, кода символ с именем `symbol` не определен. См. `.if`.

`.ifnotdef symbol`

То же, что и `.ifndef`.

`.include "filename"`

В текущий файл в том месте, где применена эта директива, вставляется файл с указанным именем, и его код ассемблируется. Для указания каталогов для поиска вставляемых по `.include` файлов может быть использована в строке команды опция `-I`.

`.int value[,value ...]`

Для каждого указанного значения `value` ассемблируется целое число. Размер, диапазон значений и порядок байтов зависят от платформы. См. также `.long`, `.short` и `.word`.

`.irp tag,str[,str]`

Весь код между директивами `.irp` и `.endr` ассемблируется столько раз, сколько параметров `str` указано. При этом каждая ссылка "\tag" в блоке кода заменяется текущей для данного прохода строкой `str`. Например, следующий код применяет одну и ту же инструкцию последовательно к трем регистрам:

```
.irp tag,esp,ebp,eax
subl $1,%\tag
.endr
```

Результирующий ассемблируемый код выглядит так:

```
subl $1,%esp
subl $1,%ebp
subl $1,%eax
```

См. также `.macro`, `.rept` и `.irpc`.

`.irpc tag,charlist`

Весь код между директивами `.irpc` и `.endr` ассемблируется столько раз, сколько имеется буквенных символов в строке `charlist`. При каждом следующем проходе ссылка \tag в блоке кода заменяется следующим буквенным символом из строки. В этом примере одна строка кода применяется трижды к разным operandам:

```
.irpc tag,123
addl $tag,%esp
.endr
```

Результирующий код выглядит так:

```
addl $1,%esp
addl $2,%esp
addl $3,%esp
```

См. также `.macro`, `.rept` и `.irp`.

.lcom *symbol*, *length*

Директива резервирует локальный блок данных в разделе `bss` длиной `length` байт. Аргумент `length` должен быть абсолютным выражением. Этот блок (как и все данные раздела `bss`) инициализируется нулевыми байтами при загрузке программы. Объявляемое имя блока символа `symbol` является локальным, то есть не может быть адресованным из других модулей. Для системы НРРА синтаксис этой директивы следующий: `symbol .lcom, length`.

.line *number*

Заменяет номер текущей строки результатом абсолютного выражения `number`. На некоторых системах вместо этой директивы должен применяться ее синоним `.ln`.

.linkonce [type]

Директива помечает текущий раздел (section) кода так, чтобы он компоновался в программу только один раз. Эта директива должна применяться в каждом отдельном экземпляре именованного раздела не более одного раза. Помеченный раздел может адресоваться только назначенным ему именем, поэтому его имя должно быть уникальным.

Необязательный аргумент `type` может иметь следующие значения: `discard` — для пропуска дубликатов без каких-либо сообщений; `one_only` — для выдачи предупредительных сообщений об обнаружении каждого дубликата; `same_size` — для выдачи предупредительного сообщения в случае несоответствия размера любого из дубликатов; `same_contents` — для выдачи предупреждения в случае несоответствия содержания любого из дубликатов.

.list

Директива увеличивает на единицу счетчик выдаваемых листингов. Ассемблер выдает листинг на стандартный выход, когда значение этого счетчика превышает ноль. Директива `.nolist` имеет противоположное действие — уменьшает счетчик на единицу. По умолчанию счетчик выдаваемых листингов имеет нулевое значение, оно также может быть установлено в командной строке опцией `-a`.

.ln [number]

Синоним директивы `.line`.

.long expression

Синоним директивы .int.

.macro name [tag[=value]][,tag[=value]]

Эта директива является рекурсивным макропроцессором. Используется для назначения имени блоку кода и указания значений **value** тэгов **tag** для подстановки в коде блока сочетаний "\tag". Назначенное имя блока расширяется его ассемблируемым кодом. Определение блока закрывает директиву .endm. Например, такой макрос с именем **saveregs** расширяется в любом месте кода двумя операторами **pushl**:

```
.macro saveregs
pushl %ebp
pushl %eax
.endm
```

Для расширения имени макроса используется в коде как обычный оператор. Например:

```
main:
    saveregs
    movl %esp, %ebp
```

Директива .macro может использоваться рекурсивно и принимать аргументы. Следующую макро-директиву можно применить для объявления блока, содержащего переменное количество констант выбиаемого типа:

```
.macro block type=.int count=1
.if \count
\type 0
block \type, \count-1
.endif
.endm
```

Без передачи аргументов макрос будет содержать только одно объявление .int. Для объявления пяти элементов данных типа .long с помощью макроса **block** можно применить такой оператор:

```
block .long 5
```

Для выхода из блока директивы .macro в любом месте его объявления можно применить директиву .exitm. Например, если применить в объявлении макроса такой оператор, то можно отменить расширение имени макроса в случае, когда **trigger** имеет значение 12:

```
.if trigger-12
.exitm
.endif
```

См. также .rept, .irp и .irpc.

.mri *expression*

Если выражение *expression* дает ненулевой результат, то процесс ассемблирования переходит в режим совместимости с ассемблером MRI. Это — тот режим, который может применяться указанием опции **-M** или **--mri**. Режим MRI действует до применения директивы **.mri** с нулевым значением результата выражения *expression*.

.nolist

См. **.list**.

.octa *bignum*[,*bignum* ...]

Для каждого аргумента *bignum* объявляется 16-битное целое число, и в него записывается значение этого аргумента. Директива имеет имя **".octa"**, потому что она может восприниматься как объявление восьми 16-битных значений. См. также **.quad**.

.org *address*[,*filler*]

Эта директива сдвигает текущий адрес вперед к расположению, адресуемому результатом выражения *address*. Выражение *address* должно быть относительным к начальному адресу текущего раздела. Допускается сдвиг адреса только в одном направлении — вперед. Пропускаемая при этом свободная область заполняется байтами, имеющими значение *filler*. По умолчанию *filler* имеет значение ноль.

См. также **.fill**, **.skip** и **.p2allign**.

.p2allign *zeroes*[,*filler*][,*maximum*]

Текущий адрес, если это необходимо, увеличивается до ближайшего значения, которое имеет заданное аргументом *zeroes* количество младших нулевых двоичных разрядов. Если, к примеру, *zeroes* имеет значение 3, то директива применяет 8-битное выравнивание текущего адреса. Пропускаемое пространство заполняется байтами, имеющими значение *filler*. По умолчанию *filler* имеет значение нулевого байта для разделов, содержащих данные, и код "пустой" инструкции **noop** для разделов выполняемого кода. Значение аргумента *maximum* указывает в байтах наибольшее допустимое расстояние, на которое переносится текущий адрес.

.p2allignl *zeroes*[,*filler*][,*maximum*]

Директива действует так же, как **.p2allign**, только *filler* имеет 16-битное значение.

.p2allignw *zeroes*[,*filler*][,*maximum*]

Директива действует так же, как **.p2allign**, только *filler* имеет 32-битное значение.

.psize *lines*[,*columns*]

Указывает размер страницы листинга. По умолчанию 60 строк (*lines*) и 200 колонок (*columns*). При указании числа строк, равного нулю, символы конца страницы (form feeds) в листинг не вставляются.

.quad *bignum*[,*bignum* ...]

Директива действует так же, как **.octa**, только ***bignum*** имеет 8-байтное значение.

.rept *count*

Код между директивами **.rept** и **.endr** повторяется заданное количество раз. Например, следующая последовательность объявляет 14 целых чисел (того же типа, который применяется инструкцией **.int**), каждое из этих чисел инициализируется значением 10:

```
.rept 14  
.int 10  
.endr
```

См. также **.macro**, **.irp** и **.irpc**.

.sbttl "subtitle"

Выводит на каждой странице листинга указанный подзаголовок.

.scl *class*

Эта директива может применяться внутри блока между парой директив **.def** и **.endif** для указания принадлежности программного символа к указанному классу.

.section *name*

Такой формат директивы **.section** применим для любого формата объектных файлов, который поддерживает "арбитражно" именованые разделы (arbitrarily named sections). Следующий за этой директивой код асsembлируется в раздел с указанным именем ***name***.

.section *name*[,"*flags*"]

Эта форма директивы **.section** применима для формата объектных файлов COFF. Следующий за этой директивой код асsembлируется в раздел с указанным именем ***name***. Каждая литера в передаваемой строке "***flags***" является флагом директивы и имеет особое значение:

- b** — раздел содержит не инициализированные данные (раздел **bss**);
- n** — при выполнении программы этот раздел не загружается;
- w** — допускается запись в раздел во время выполнения программы;
- d** — раздел данных, не содержит выполняемый код;
- r** — раздел "только для чтения";
- x** — раздел содержит выполняемый код, не предназначен для хранения данных.

В случае, когда флаги не указываются, применяемые по умолчанию установки зависят от имени раздела (**section name**). Если имя раздела не имеет предустановленного значения, то код асsembлируется в раздел по умолчанию.

.section name[, "flags"[, type]]

Такая форма директивы `.section` применима для формата объектных файлов ELF. Следующий за этой директивой код ассемблируется в раздел с указанным именем `name`. Каждая буква в передаваемой строке "`flags`" является флагом директивы и имеет особое значение:

- `a` — перемещаемый раздел (allocatable section);
- `w` — в раздел допускается запись;
- `x` — раздел содержит выполняемый код.

Если указывается аргумент `type`, то он может иметь следующее значение:

- `@progbits` — раздел содержит данные;
- `@nobits` — раздел не содержит данных (свободное пространство).

В случае, когда флаги не указываются, установки по умолчанию зависят от имени раздела (`section name`). Если имя раздела не имеет предустановленного значения, то по умолчанию считается, что: или раздел не перемещаемый и запись в него разрешена; или раздел выполнимый и может содержать данные.

.section "name"[, flag ...]

Такая форма директивы `.section` применима для ассемблера системы Solaris, вырабатывавшего объектный код формата ELF. Необязательный список флагов может содержать следующие значения `flag`:

- `#alloc` — перемещаемый раздел (allocatable section);
- `#write` — в раздел допускается запись;
- `#execinstr` — раздел состоит из выполнимых инструкций.

.set symbol, value

Директива определяет программный символ `symbol` со значением `value`. Аргумент `value` может быть как абсолютным, так и относительным выражением. Директива `.set` может применяться к одному символу столько раз, сколько это нужно, каждый раз изменяя его значение. На системах HPPPA она имеет синтаксис `symbol .set value`. Эта директива равносильна `.equ`. См. также `.equiv`.

.short value[, value ...]

Может быть синонимом одной из директив `.hword` или `.short` в зависимости от платформы. Также см. `.int`.

.single value[, value ...]

Синоним директивы `.float`.

.size

Директива может ставиться только в блоке между парой директив `.def` и `.endef`. Используется компиляторами для включения дополнительной информации в таб-

лицу программных символов (symbol table). Применяется только с форматом объектных файлов COFF.

.sleb128 *value[,value ...]*

Название директивы является сокращением от "signed little endian base-128". При этом имеется в виду компактное, переменной длины представление чисел, которое используется стандартом символьной отладки DWARF. См. также **.uleb128**.

.skip *size[,filler]*

Эта директива создает блок, имеющий длину в байтах, равную **size**. Этот блок заполняется значением аргумента **filler**. По умолчанию **filler** равен нулю. См. также **.fill**, **.org** и **.p2align**.

.stabd *type,other,description*

См. описание формата STABS в главе 13.

.stabs "name:symdesc=typeinfo",*type,other,description,value*

См. описание формата STABS в главе 13.

.stabn *type,other,description,value*

См. описание формата STABS в главе 13.

.string "characters"[,"characters" ...]

Директива записывает в память строку буквенных символов "**characters**" или несколько таких строк. После каждой строки добавляется закрывающий нулевой байт (terminator). В них допускается использование применяемых в языке C escape-последовательностей, начинающихся с символа "\".

.symver *name,name2@nodename*

При применении формата объектного кода ELF эта директива "подшивает" символ к узлу (node) с указанным именем **nodename**, это используется при ассемблировании кода с разделяемой библиотекой (shared library). Директива создает символ **name2@nodename** в качестве псевдонима (alias) символа с именем **name**, определенного в любом месте того же исходного файла. Часть псевдонима **name2** становится при этом действующим локальным именем, разрешимым для любых внутренних ссылок. Имя **nodename** является именем поддерживаемого при компоновке узла и может указываться компоновщику (linker) в командной строке.

.tag *structname*

Применяется только для формата объектных файлов COFF только внутри блока кода между парой директив **.def** и **.endef**. Эта директива используется компиляторами для включения информации о результатах отладки в таблицу символов.

.text [*subsection*]

Операторы, которые следуют после этой директивы, добавляются в конец текстового подраздела (text subsection) с именем, которое указывает значение абсолютного выражения *subsection*. По умолчанию аргумент *subsection* равен нулю.

.title "*heading*"

Указанная в этой директиве строка выводится в заголовках страниц листинга сразу после имени исходного файла и номера страницы.

.type *value*

Эта директива может применяться только для формата объектных файлов COFF внутри блока кода между парой директив .def и .endef. Аргумент *value* определяет тип целых чисел для использования в таблице программных символов (symbol table).

.val *address*

Может применяться только для формата COFF внутри блока кода между парой директив .def и .endef. Аргумент назначает адрес для текущей записи таблицы символов (symbol table).

.uleb128 *value*[,*value* ...]

Название директивы является сокращением от "unsigned little endian base-128". Сокращение обозначает компактное, переменной длины представление чисел, используемое стандартом отладки DWARF. См. также .sleb128.

.word *value*[,*value* ...]

Директива объявляет и присваивает числовые значения, при этом их размер и порядок следования байт зависят от платформы.



Глава 16

Перекрестная компиляция и перенос программ на систему MS Windows

По умолчанию компилятор GCC вырабатывает машинный код для той же вычислительной системы, на которой он используется. Но его можно установить и сконфигурировать таким образом, чтобы код генерировался для других машин. Существует возможность устанавливать модули компилятора, необходимые для выработки кода для различных целевых платформ и выбирать необходимый модуль из командной строки.

Целевые платформы

Свежий перечень доступных целевых платформ (targets) можно найти на web-сайте <http://gcc.gnu.org/install/specify.html>. На этом сайте приведен обновленный перечень поддерживаемых системно-аппаратных сочетаний и последние сведения о переносе программ на любую из них. Для каждой платформы есть краткое описание, и для многих приведены особые условия переносимости на них программ. Список доступных платформ уже довольно велик и разработка технологии переноса программного обеспечения на новые системы непрерывно продолжается. Ниже приведен перечень поддерживаемых компилятором систем, актуальный на момент написания этой книги:

#s390-*-linux*	m6811-elf
#s390x-*-linux*	m6812-elf
--freebsd*	m68k-att-sysv
--linux-uml	m68k-crds-unos
--solaris2*	m68k-hp-hpux

--sysv	m68k-ncr-*
-ibm-aix	m68k-sun
*-lynx-lynxos	m68k-sun-sunos4.1.1
alpha**-*	Microsoft Windows
alpha*-dec-osf*	mips**-
alphaev5-cray-unicosmk*	mips-sgi-irix5
arc*-elf	mips-sgi-irix6
arm**-linux-gnu	Older systems
arm**-aout	OS/2
arm**-elf	powerpc**-*powerpc**-sysv4
avr	powerpc**-darwin*
c4x	powerpc**-eabi
DOS	powerpc**-eabiaix
dsp16xx	powerpc**-eabisim
ELF (SVR4, Solaris 2, etc)	powerpc**-elf powerpc**-sysv4
h8300-hms	powerpc**-linux-gnu*
hppa*-hp-hpux*	powerpc**-netbsd*
hppa*-hp-hpux10	powerpcle**-eabi
hppa*-hp-hpux11	powerpcle**-eabisim
hppa*-hp-hpux9	powerpcle**-elf powerpcle**-sysv4
i370**-	powerpcle**-winnt powerpcle**-pe
i?86**-esix	sparc**-linux*
i?86**-linux*	sparc-sun-solaris2*
i?86**-linux*aout	sparc-sun-solaris2.7
i?86**-sco	sparc-sun-sunos4*
i?86**-sco3.2v4	sparc-unknown-linux-gnulibc1
i?86**-sco3.2v5*	sparc64**-
i?86**-udk	sparcv9**-solaris2*
ia64**-linux	vax-dec-ultrix
m32r*-elf	xtensa**-elf
m68000-hp-bsd	xtensa**-linux*

Построение кросс-компилятора

Применяемое в GCC соглашение об именах позволяет скомпилировать и установить на одной машине столько перекрестных компиляторов, сколько вам может понадобиться. Чтобы иметь возможность компилировать и компоновать программы для другого компьютера, нужны основные инструменты (ассемблер, компоновщик и т.д.), которые принимают объектные файлы и вырабатывают из них выполнимый код в требуемом для целевой машины формате. Кроме того, на компьютере следует установить копии всех необходимых библиотек, которые присутствуют в предназначаемой целевой системе.

Приведенная ниже последовательность шагов может служить общим руководством для установки среды *кросс-компилятора* (cross compiler). При этом следует иметь в виду, что нередко могут возникать ситуации, имеющие некоторые особенности.

Перед настройкой среды окружения кросс-компилятора следует еще раз просмотреть информацию об утилатах пакета *binutils* и сценарии *configure*, приведенную во второй главе. Также посетите web-сайт проекта GCC, где можно найти свежие дан-

ные об интересующей вас платформе переноса. И неплохо было бы подписатьсь на соответствующие почтовые рассылки, о которых говорилось в главе 1. Это позволит вам поддерживать связи с другими разработчиками, занимающимися теми же проблемами. Дискуссии на тему проблем с компиляцией кода с помощью компилятора GCC для различных платформ не прекращаются никогда.

Если у вас нет веской причины для применения текущей разрабатываемой версии компилятора, то лучше используйте устоявшиеся выпущенные версии кода компилятора, а не текущую версию *CVS*. Может оказаться, что компилятор версии *CVS* будет прекрасно работать на нескольких машинах, но при этом вы можете столкнуться с большим количеством неизвестного, чем хотелось бы. Если уж вы попадете в такое приключение, то сообщите о своих открытиях участникам разработки компилятора через соответствующую подписку. Тем самым вы тоже внесете вклад в разработку и усовершенствование GCC.

Установка начального компилятора

В случае, когда кросс-компилятор должен вырабатывать объектный код, который будет установлен на целевой машине, сам компилятор, а также все необходимые для его работы вспомогательные программы, в частности ассемблер и компоновщик, должны быть скомпилированы для работы на локальной ("домашней") машине. Для этого понадобится комплектный вашей локальной системе (иначе говоря, "родной") компилятор. Первая ваша задача заключается в установке "домашней" версии компилятора GCC, если она еще не установлена. И, конечно же, такой же версии набора утилит *binutils*.

Вполне возможно, что кросс-компилятор можно построить и без помощи GCC, но это может привести к неприятным проблемам. Как уже говорилось, не хотелось бы сталкиваться с большим количеством неизвестного, чем это необходимо.

Построение отдельного набора *binutils*, ориентированного на целевую платформу

Утилиты набора *binutils*, описанные в главе 2, должны компилироваться для целевой машины. Благодаря используемому в GCC соглашению об именах при компиляции и установке утилит *binutils* для другой машины не возникает конфликтов имен. Компиляция может быть основана на том же наборе файлов исходного кода, который применялся для создания собственного набора утилит *binutils* "домашней" машины кросс-компилятора.

В следующем примере предполагается, что исходный код пакета *binutils* находится в подкаталоге *sun* текущего каталога. Приведенные ниже четыре команды (их можно поместить в простой сценарий оболочки) создают новый каталог *sun* и конфигурируют его для компиляции исходников *binutils* в выполняемые на локальной машине программы, которые способны вырабатывать код, ориентированный на платформу *sparc-sun-solaris2.7*:

```
DIR='pwd'  
mkdir $DIR/sun  
cd $DIR/sun
```

```
$DIR/srs/configure --prefix=/usr/local \
--target=sparc-sun-solaris2.7
```

После завершения конфигурирования этот набор утилит binutils можно скомпилировать так: перейти в новый каталог `sun` и запустить команду `make`:

```
$ cd sun
$ make
```

И последний шаг — нужно войти с правами суперпользователя и установить новые программы с помощью следующей команды:

```
$ make install
```

Эта команда создает в каталоге `/usr/local/bin` следующий набор файлов:

```
$ ls /usr/local/bin/sparc-sun-solaris2.7-
sparc-sun-solaris2.7-addr2line      sparc-sun-solaris2.7-objdump
sparc-sun-solaris2.7-ar            sparc-sun-solaris2.7-ranlib
sparc-sun-solaris2.7-as            sparc-sun-solaris2.7-readelf
sparc-sun-solaris2.7-c++filt       sparc-sun-solaris2.7-size
sparc-sun-solaris2.7-ld            sparc-sun-solaris2.7-strings
sparc-sun-solaris2.7-nm            sparc-sun-solaris2.7-strip
sparc-sun-solaris2.7-objcopy
```

Также в `/usr/local` могут появиться новые каталоги:

```
$ ls /usr/local/sparc-sun-solaris2.7
bin    lib
$ ls /usr/local/sparc-sun-solaris2.7/bin
ar    as   ld   nm   ranlib   strip
```

Установка файлов из целевой системы

Для компиляции исходного кода на предназначаемую целевую машину необходимо иметь сконфигурированные для нее системные заголовочные файлы. Кроме того, выработка программ для выполнения на целевой машине требует их компоновки с соответствующими целевой системе библиотеками. Состав необходимых библиотек зависит от назначения кросс-компилятора. Если вам нужен кросс-компилятор общего назначения для компиляции полных приложений, то вам понадобятся все стандартные заголовочные файлы и все стандартные библиотеки целевой системы. Если же вам нужен кросс-компилятор для *встраиваемых систем* (*embedded systems*), не использующий стандартные библиотеки и заголовочные файлы, то, может быть, копировать файлы вообще не придется.

Вам могут понадобиться копии некоторых библиотек, находящихся в каталогах `/lib` и `/usr/lib` целевой системы. Эти новые файлы должны быть записаны в структуре, предварительно созданной при установке утилит binutils. В данном случае все необходимые библиотеки целевой системы должны быть скопированы в локальный каталог `/usr/local/sparc-sun-solaris2.7/lib`. Точный набор нужных библиотек зависит от целевой платформы и типа переносимой программы.

Помимо библиотек вам понадобятся объектные файлы целевой системы, которые компонуются в выполняемые файлы (например, файлы с такими именами, как `crt0.o` и `ctrn.o`), их можно скопировать в тот же каталог, что и библиотеки.

Заголовочные файлы целевой системы должны копироваться в каталог `/usr/local/sparc-sun-solaris2.7/include`. Очень важно, чтобы заголовочные файлы были скопированы на компьютер до компоновки кросс-компилятора, поскольку они используются для построения библиотеки `libgcc.a`.

Конфигурируемая библиотека `libgcc1.a`

Если на целевой машине имеется компилятор GCC и с нее возможно скопировать библиотеку `libgcc1.a`, то вам не нужно создавать ее заново. Если же нет, то библиотеку необходимо создать.

Эта библиотека содержит подпрограммы математических действий над числами с плавающей точкой, используемые на системах, не имеющих аппаратной поддержки таких операций. Если эмуляция операций с плавающей точкой не требуется, то можно использовать пустую библиотеку `libgcc1.a`.

Некоторые *встраиваемые системы* содержат всю арифметику для чисел с плавающей точкой, требующуюся библиотеке `libgcc1.a`.

Если на целевой системе имеется комплектный компилятор C, но нет компилятора GCC, то вы можете либо установить на ней GCC и с его помощью сгенерировать библиотеку, либо использовать комплектный этой системе компилятор C для создания только одной библиотеки `libgcc1.a`. Для этого установите дерево исходных каталогов компилятора GCC в целевой системе, создайте компоновочный каталог и выполните сценарий `configure`, указав ему текущую систему в качестве целевой платформы (`target`) и систему, на которой будет установлен кросс-компилятор в качестве локальной, или "домашней" платформы, (`host`). После этого скомпонуйте библиотеку. Вот все необходимые для этих действий команды:

```
$ ./configure --host=host --target=target  
$ make libgcc1.a
```

Полученную таким способом библиотеку нужно вместе с другими библиотеками скопировать на ту машину, на которой строится кросс-компилятор.

Компоновка кросс-компилятора

Если все подготовительные действия выполнены правильно, то единственное, что осталось сделать, — это скомпилировать новый компилятор. В приведенном ниже примере сценария предполагается, что исходный код GCC находится в подкаталоге `gcc`. Сценарий создает новый каталог `sun`, который будет содержать используемую при компиляции конфигурацию.

```
DIR='pwd'  
mkdir $DIR/sun  
cd $DIR/sun  
$DIR/srs/configure --prefix=/usr/local \  
--target=sparc-sun-solaris2.7
```

После завершения процедуры конфигурирования перейдите в новый каталог и скомпилируйте кросс-компилятор следующими командами:

```
$ cd sun  
$ make
```

При этом выполняется полная компиляция GCC, она потребует довольно много времени. Если все предыдущие операции были выполнены правильно, то при компиляции не должно возникать сообщений об ошибках. Если на компьютере содержится неправильная версия библиотеки `libgcc1.a` или она отсутствует, то компиляция аварийно завершится на обработке первого же модуля, использующего эту библиотеку. Также может оказаться, что отсутствует один или несколько заголовочных файлов.

Если компилятор скомпонован без ошибок, то следующая команда, запущенная с правами суперпользователя, установит компилятор и подготовит его к запуску:

```
$ make install
```

Запуск кросс-компилятора

Кросс-компилятор можно запустить из командной строки командой `gcc` с опцией `-b`. Например, для компиляции файла `helloworld.c` с помощью созданного в этой главе компилятора введите следующую команду:

```
$ gcc -b sun-sparc-solaris2.7 helloworld.c -o helloworld
```

Если текущая версия компилятора `gcc` — 3.2, то приведенная выше команда запустит компилятор `sun-sparc-solaris2.7-gcc-3.2`. Если по какой-то причине вы пользуетесь более новой версией компилятора, но хотите запустить кросс-компилятор версии 3.2, в командной строке можно также указать номер версии, как показано в следующем примере:

```
$ gcc -b sun-sparc-solaris2.7 -v 3.2 helloworld.c -o helloworld
```

С помощью опции `-v` вы выбираете одну из установленных у вас версий компилятора. Различные версии компилятора обычно находятся в каталогах, названных следующим образом:

```
/usr/local/lib/gcc-lib/machine/version
```

MinGW — компилятор для Windows

На операционных системах Microsoft Windows могут компилироваться два типа программ. Простейший из них — консольные программы, не использующие оконный интерфейс. Консольная программа Windows запускается из командной строки и может принимать аргументы командной строки. Выполнение консольной программы, написанной на языке C, начинается с функции `main()`. Она использует стандартные системные устройства ввода, вывода и сообщений об ошибках.

Консольная программа для Windows может компилироваться с помощью компилятора *MinGW* (Minimalist GNU for Windows). Этот компилятор можно скачать с сайта <http://www.mingw.org>. Компилятор MinGW представляет собой набор пакетов, их все можно загрузить в одном установочном файле, имя которого имеет следующий формат:

```
MinGW-<version>[-<stamp>].tar.gz  
MinGW-<version>[-<stamp>].zip
```

Поле `<version>` содержит номер версии, например 1.0 или 1.1. Необязательный параметр `<stamp>` — это дата в формате "YYYYMMDD", когда пакеты были собраны в один установочный файл.

Для установки компилятора MinGW загрузите установочный файл в рабочий каталог и создайте каталог, который будет использоваться для установки, например, `C:\mingw`. Распакуйте полученный файл в этот каталог. В архиве содержатся каталоги, поэтому убедитесь, что программа разархивации сохраняет структуру дерева каталогов, имеющуюся в архиве. В некоторых случаях для этого требуется указывать специальные опции командной строки.

После распаковки вам остается только добавить новый каталог `bin` в системную переменную окружения `PATH`. Способ назначения переменной среды окружения зависит от установленной версии Windows, но для большинства версий будет достаточно следующей команды:

```
PATH=%PATH%;c:\mingw\bin
```

Проверку правильности инсталляции можно произвести с помощью следующей команды, которая должна вывести на экран информацию о версии компилятора:

```
gcc -v
```

Программы `gcc` и `g++` пакета MinGW имеют практически те же опции командной строки, что и соответствующие программы в версиях для UNIX.

Компилятор проекта Cygwin

Cygwin представляет собой эмулятор среды UNIX, который устанавливается в системе Microsoft Windows. В него входит готовая версия пакета `binutils` и DLL-библиотека `cygwin1.dll`, которая представляет собой реализацию интерфейса прикладных программ (API) системы UNIX. Процесс установки Cygwin достаточно прост и сводится к выполнению следующих действий:

1. Создайте рабочий каталог, в котором будут храниться загруженные файлы. Это временный каталог, а не тот каталог, в который будет производиться окончательная установка.
2. С помощью web-браузера зайдите на сайт <http://cygwin.com>. Справа щелкните на пиктограмме с надписью "Install Cygwin now". После этого начнется загрузка файла `setup.exe`.
3. Из командной строки или из системного меню "Выполнить" запустите программу `setup.exe`. Она шаг за шагом проведет вас через весь процесс загрузки и установки системы Cygwin.

Компиляция в Cygwin консольной программы

Команды компиляции и компоновки программ очень похожи на соответствующие команды, использующиеся для компилятора GCC, тем не менее, имеются определенные отличия в соглашениях об именах файлов. Все выполняемые файлы имеют расширение `.exe`, а динамические библиотеки — расширение `.dll`. Приве-

денная ниже команда компилирует простую программу `helloworld.c` в выполнимый файл:

```
C:\> gcc helloworld.c -o helloworld.exe
```

Компиляция в Cygwin программы, использующей объекты Windows GUI

Исходный код приложений для Windows будет компилироваться практически без изменений, но есть несколько исключений.

Необходимо удалить атрибуты `__export`. В большинстве случаев атрибуты `__export` можно просто исключить из кода, но иногда их нужно заменить новыми объявлениями следующего вида:

```
int fn(int) __attribute__ ((__dllexport__));
int fn(int) ...
```

В исходный код можно включить следующий код условной компиляции:

```
#ifdef __CYGWIN__
WinMainCRTStartup() { mainCRTStartup(); }
#endif
```

При компиляции без включения приведенного выше кода в командной строке нужно указать опцию компоновки `-e mainCRTStartup`.

Приведенный далее код представляет собой фрагмент компоновочного сценария, который скомпилирует программу для Windows в готовый к выполнению файл, использующий графический интерфейс пользователя (*GUI*):

```
helloworld.exe: helloworld.o helloworld.res
    gcc -mwindows helloworld.o helloworld.res -o helloworld.exe

helloworld.res: helloworld.rc resource.h
    windres helloworld.rc -O coff helloworld.res
```

Утилита `windres` компилирует файл `helloworld.rc` в объектный формат COFF, который содержит пиктограммы (*icons*), точечные изображения (*bitmaps*) и другие необходимые для программы ресурсы. Если бы опция `-O coff` не была указана, то результатирующий файл `.res` имел бы формат системы Windows и не мог бы компоноваться в GCC.



Глава 17

Встраиваемые системы

Компиляция программного обеспечения для установки во встраиваемых системах (*embedded systems*) практически ничем не отличается от перекрестной компиляции для другой целевой системы. То есть программа компилируется на одной машине и на ней генерируется выполнимый файл, который будет выполняться на другой машине. Основное различие заключается в том, что операционная система предназначения разработана для выполнения определенной цели и не имеет возможностей для разработки программного обеспечения.

Как правило, встраиваемая система имеет жесткое ограничение по объему занимаемой памяти и в общем случае представляет собой гораздо более ограниченную среду, чем операционные системы общего назначения. Это означает, что компилятор встраиваемой системы должен не только генерировать код, который будет выполняться процессором встроенной системы, но и создавать выполнимые файлы минимально возможного размера и максимально возможной производительности.

Настройка компилятора и компоновщика

Основные моменты подготовки компилятора GCC к генерации кода для встраиваемой системы описаны в главе 16. Компилятор GCC достаточно хорошо приспособлен для настроек такого рода, поскольку его можно компилировать и устанавливать на целом ряде платформ для генерации кода, ориентированного на другие платформы. Одна из задач программиста заключается в настройке кросс-компилятора для целевого процессора с помощью библиотек и компонуемых объектных модулей, поставляемых с целевой операционной системой. Кроме того, вы должны загрузить и установить утилиты `binutils` (включающие ассемблер и компоновщик), которые позволяют создавать объектные модули для целевой системы.

После установки кросс-компилятора и компоновщика можно выбрать один из доступных в GCC языков программирования, или, при желании, можно использовать несколько языков. Более того, вы имеете возможность воспользоваться всеми имеющимися в GCC средствами оптимизации. Для случаев, когда необходимы средства для точной работы с аппаратной частью, есть возможность вставлять в программы ассемблерный код.

Поскольку доступны все опции компилятора GCC, вы можете настраивать содержимое объектного кода таким образом, что он будет соответствовать требованиям целевой (предназначенной) системы. Настраивая компоновочный сценарий (make-file), вы можете установить отдельные настройки для каждого модуля и, соответственно, оптимизировать полученный результат. Особое внимание следует уделить тем опциям командной строки компилятора, которые включают данные в объектные файлы, передаваемые через процесс компоновки в конечный выполнимый файл. Некоторые данные, в частности, отладочная информация, могут быть несовместимы с форматом объектных файлов предназначенной системы.

Для создания встраиваемой системы необходимо выполнить компоновку со специальным модулем запуска, соответствующим операционной системе. Как правило, это небольшой блок кода на языке ассемблера, который компонуется непосредственно в программу. Некоторые из этих процедур инициализации могут быть довольно большими, в то время как другие — очень просты. Общая последовательность инициализации фактически является стандартной и содержит все или некоторые из перечисленных ниже действий:

- блокирование всех аппаратных прерываний;
- обнуление области данных;
- копирование данных инициализации с носителя в ОЗУ;
- выделение памяти под стек и инициализация указателя на вершину стека;
- выделение динамически распределяемой памяти (heap);
- разблокирование соответствующих аппаратных прерываний;
- вызов программы или переход к ее главной процедуре.

Определенный код может вставляться также и после вызова основной подпрограммы. Он может использоваться для вывода диагностических данных для отладки, и затем снова возвращать управление основной подпрограмме. Кроме того, этот код может выдавать команду возврата в исходное состояние, которая перезапускает весь процесс с самого начала или, в зависимости от назначения встраиваемого программного обеспечения, просто останавливать процессор.

Встраиваемая система практически всегда представляет собой непрерывный цикл, вызывающий функции для выполнения основных задач программного обеспечения. В более сложных системах может инициализироваться несколько потоков, в каждом из которых запускается непрерывный цикл, выполняющий отдельную задачу. Во встраиваемых системах потоки, как правило, называются *задачами* (*tasks*).

Для использования вашего собственного кода запуска в некоторых случаях может потребоваться, чтобы компоновщик игнорировал полученные от GCC инструк-

ции. С этой целью компоновщик GNU имеет специальный язык сценариев — язык команд компоновщика (Linker Command Language), который предоставляет возможность явного и точного управления процессом компоновки. Этот язык сценариев очень гибок, он позволяет отказаться от компоновки с помощью GCC и организовать собственный компоновочный процесс — язык сценариев имеет достаточно подробные команды, с помощью которых можно указывать расположение разделов компонуемых объектов.

Выбор языка

Написание кода для встраиваемых систем отличается от написания кода при решении задач общего программирования. Здесь одним из основных критериев является размер кода, скорость его выполнения всегда имеет большое значение. Код, подходящий для системы общего назначения, в качестве встраиваемого программного обеспечения может стать источником многих проблем. Эта ситуация накладывает определенные ограничения на выбор языка и компилятора.

Как правило, возникает вопрос об использовании ассемблера или C. Конечно, можно воспользоваться и другими языками программирования, но ассемблер и C — наиболее часто используемые языки. Во всех случаях удобнее работать с языком более высокого уровня, поскольку он легче читается, на нем удобнее писать, и исходный код на языке высокого уровня более понятен. Процесс написания программы идет гораздо быстрее и с меньшим количеством ошибок, когда код легко читается. Компилятор C генерирует код, который хотя и не настолько сжат и эффективен, как написанный вручную ассемблерный код; однако достаточно чистый и вполне пригодный для использования. Очевидно, что язык ассемблера не исчезнет в ближайшее время, но его не следует использовать больше, чем это действительно необходимо.

Если вы ограничены временем, то компилятор GCC предоставит вам информацию, которую можно использовать для проверки корректности ассемблерного кода. Вы можете воспользоваться следующими функциями:

- *Оптимизация.* Для генерации компилятором кода языка ассемблера укажите опцию `-S`. Используйте ее на различных уровнях оптимизации и с установкой различных флагов оптимизации. Не исключено, что оптимизатор может сделать все, что вам необходимо.
- *Анализ инструкций.* Использование опции `-fP` совместно с опцией `-S` приводит к тому, что в листингах кода ассемблера в комментариях будет указываться длина каждой инструкции. Помимо длины инструкции приводится номер узла дерева и инструкция дерева, которая вызвала генерацию данной строки кода ассемблера.
- *Анализ дерева.* Использование опции `-fP` совместно с опцией `-S` дает тот же результат, что и опция `-fP` и, кроме того, вставляет строки комментариев, содержащие номера узлов дерева промежуточного языка. Для большинства разработчиков эта опция имеет ограниченное значение, поскольку предполагается, что разработчики знакомы с внутренней структурой gcc.

- *Проверка установленных опций.* Использование опции `-fverbose-asm` приводит к тому, что компилятор в начале листингов кода языка ассемблера предоставляет полный перечень всех опций, установленных при компиляции программы в ассемблерный код. Может оказаться, что для получения желаемого результата необходимо изменить одну или несколько опций.

Кроме того, есть и другие сведения о генерировании кода, подробно описанные в главе 18. Наиболее полезной информацией являются общий размер кода и данные о выделении памяти.

Если вы выбрали вариант языка ассемблера, для его реализации у вас есть несколько возможностей. Если необходимо внести несколько изменений в код анализируемой программы, то это можно сделать путем редактирования генерированного компилятором модуля ассемблерного кода. В таком коде уже присутствует необходимый интерфейсный блок, связывающий этот модуль с другими частями программы. Возможно, наиболее эффективный подход заключается в определении необходимых изменений и замене соответствующих частей кода программы на языке С вставками строк ассемблерного кода.

Средства GCC для разработки встраиваемого программного обеспечения

Компилятор GCC не предназначался специально для разработки встраиваемого программного обеспечения, но он достаточно развит и настолько гибок, что содержит практически все, что может понадобиться разработчику встраиваемых программ.

Опции командной строки

Некоторые опции командной строки компилятора GCC могут оказаться особенно полезными для разработки встраиваемых программ. Функция проверки ошибок может быть настроена для обнаружения всех возможных ошибок, касающихся одного определенного аспекта программирования. Изучите набор опций `-W`, приведенных в приложении Г, и установите нужные (или снимите ненужные) флаги, настроив компилятор на наилучшее соответствие вашей среде. Начать можно с установки опции `-Wall`, которая указывает компилятору выводить предупреждения даже о самых незначительных нарушениях. Если окажется, что после установки этой опции выводятся предупреждения, которые вы не хотели бы получать, то можно воспользоваться индивидуальными опциями и отключить вывод нежелательных предупреждений.

Для вырабатываемого кода может иметь большое значение оптимизация. У компилятора есть развитые возможности управления параметрами оптимизации (см. описание опции `-O` в приложении Г).

Для запрета использования компилятором определенного регистра можно использовать опцию командной строки `-ffixed-reg`. Например, если процессор предназначаемой платформы содержит регистр с именем `gr4` и этот регистр не должен

использоваться генерируемым кодом, то воспользуйтесь следующим набором опций:

```
$ gcc -c -Wall -ffixed-gr4 mainloop.c -o mainloop.o
```

Диагностика

Компилятор имеет возможность форматировать имена функций и файлов исходного кода в строку, которую затем можно использовать при выводе диагностических сообщений. Например, в приведенном ниже коде создается строка, содержащая имя текущей функции, имя ее файла исходного кода и дату компиляции:

```
sprintf(msg, "Function %s in file %s compiled %s\n",
__FUNCTION__,__FILE__,__DATE__);
```

В языке программирования C++ для определения имени функции используется макрос __PRETTY_FUNCTION__. В языке C применение макросов __FUNCTION__ и __PRETTY_FUNCTION__ дает одинаковый результат.

Ассемблерный код

Как было сказано в главе 15, задача компоновки модулей языка ассемблера с модулями, написанными на языках более высокого уровня, не представляет особых трудностей. Кроме того, ассемблерные блоки можно включать непосредственно в компилируемый код на языке высокого уровня.

Может оказаться, что модуль на ассемблере необходимо скомпоновать в выполнимый файл, но он не был написан для использования в программе на языке C и его имя не содержит обязательного начального символа подчеркивания. Приведенные ниже операторы определяют программные символы на языке C, которые можно локально использовать для адресации на глобальные символические имена, определенные в ассемблерном коде:

```
extern int musref asm("muslimit");
int rebar asm("rebclean");
extern int gribbit(void) asm("asmgribbit");
```

Символ `musref` в исходном файле на языке C будет компоноваться в качестве указателя типа `int` на глобально определенное имя `muslimit`. Переменная `rebar` типа `int` в коде на ассемблере определена как `rebclean`, в то время как нормальное объявление переменной `rebar` привело бы к тому, что в коде ассемблера она получила бы имя `_rebar`. Третья строка примера представляет собой прототип объявления функции, который приведет к тому, что функция объявлена или используемая как `gribbit` в коде ассемблера, будет доступна из кода на языке C по имени `asmgribbit`.

Ключевое слово `__attribute__` компилятора GCC можно использовать для указания имени раздела, в котором размещается функция или данные, объявленные на ассемблере. Например, в следующем примере использование ключевого слова `__attribute__` приведет к тому, что переменная `trigmax` будет помещена в раздел `convvals`:

```
const int trigmax __attribute__ ((section("convvals")));
```

Возможность указания имен разделов (секций кода) позволяет использовать компоновщик для указания точного расположения и порядка блоков объектного кода.

Библиотеки

Часто встраиваемое программное обеспечение включает в себя динамические библиотеки, подключаемые во время выполнения программы (*runtime libraries*). Хорошая библиотека времени выполнения (RTL) может содержать все необходимые вам функции, и от вас потребуется только написать код приложения. После компиляции и установки в вашей системе кросс-компилятора можно скомпилировать и скомпоновать библиотеку, и затем указывать ее расположение в командах компоновщику.

Если у вас нет динамической библиотеки, то, скорее всего, в качестве ее будет использоваться часть стандартной библиотеки GNU. К сожалению, при необходимости использования большой части стандартной библиотеки процесс вычисления нужных частей может оказаться достаточно утомительным из-за большого количества перекрестных ссылок. Но этих трудностей можно избежать при использовании библиотеки `newlib`.

Сокращение стандартной библиотеки

Использование полной стандартной библиотеки языка C может привести к тому, что размер сгенерированного выполняемого модуля окажется в несколько раз больше, чем нужно. Многие стандартные модули C предназначены для очень широкого использования и реализованы с учетом того, что они будут загружаться в память из разделяемой библиотеки несколькими процессами одновременно. Во встраиваемой системе статически связанные модули такого типа будут очень неэффективными виду их большого размера.

Одним из наиболее показательных примеров является функция `printf()`. Эта функция может иметь переменное количество аргументов. Первый аргумент является строкой символов, а остальные представляют собой список типов данных переменной длины. Поскольку информация о форматировании указывается строкой символов, в качестве части программы должны присутствовать функции приведения возможных типов данных к требуемому формату. Кроме того, многие функции форматирования используют другие библиотечные функции. В результате получаем эффект домино, требующий включения большого объема кода, который никогда не используется.

Если вы собираетесь использовать функции стандартной библиотеки GCC, то целесообразно сократить ее до используемых модулей, а остальные модули полностью удалить. В зависимости от используемого объема библиотеки процесс удаления ненужных модулей может оказаться достаточно трудоемким. Его можно начать с создания библиотеки, содержащей только вызовы необходимых функций, а затем по мере необходимости добавлять модули, необходимые для восстановления неразрешимых ссылок.

Библиотека, предназначенная для встраиваемых систем

Для компоновки программного обеспечения встраиваемых систем существует стандартная библиотека. Она свободно распространяется в соответствии с общественной лицензией и содержится на сайте <http://source.redhat.com/newlib/>.

Библиотека, названная **newlib**, представляет собой библиотеку языка C, предназначенную для использования во встраиваемых системах. Она состоит из функций и подпрограмм, собранных из разных источников, свободно распространяемых на основании общественной лицензии. Она распространяется в виде исходного кода. Код достаточно прост и компилируется без ошибок на многих процессорах. Одним из главных достоинств библиотеки **newlib** является то, что она разработана специально для встраиваемых систем.

Библиотека загружается и устанавливается практически так же, как GCC и набор binutils. После загрузки исходного кода его необходимо установить в рабочем каталоге **newlib**. Каталог **newlib** следует поместить на одном уровне с каталогами исходного кода GCC и binutils. Необходимо создать отдельный каталог для компоновки и выполнить сценарий **configure**, указав ему префикс каталога для установки **--prefix** и предназначаемую целевую платформу **--target**.

Кроме того, рекомендуется изучить и другие опции конфигурации. Возможно, что некоторые из них вам понадобятся. Для получения полного списка доступных опций используется опция **--help** сценария **configure**. В частности, опция **--newlib-hw-fp** компилирует функции библиотеки с использованием аппаратной поддержки арифметики с плавающей точкой. По умолчанию в функциях библиотеки предполагается, что аппаратная поддержка операций с плавающей точкой недоступна и используются только операции арифметических действий с целыми числами. При использовании системных операций с плавающей точкой программы, как правило, имеют меньший размер и выполняются быстрее.

Для создания библиотеки выполните команду **make**, и, затем, команду **make install**.

Язык сценариев компоновщика ld

Управление компоновщиком GNU **ld** осуществляется с помощью особых скриптов. Если сценарий отдельно не указан, то будет использоваться тот сценарий, который был встроен в **ld** при его установке. Существует возможность запускать собственный сценарий, это показано в следующем примере. Здесь применяется сценарий **spring.link** для компоновки выполнимого загрузочного модуля с именем **spring**:

```
$ ld -T spring.link start.o loop.o bfrspr.o -o spring
```

Опция **-T** указывает имя файла сценария. Опция **-c** — синоним опции **-T**.

Основной причиной использования специального сценария является схема адресации. Как правило, компоновщик (linker) вырабатывает выполнимый файл с настраиваемыми адресами, которые устанавливаются каждый раз при загрузке модуля

в память. Каждый раздел описывается двумя адресами (во многих случаях они имеют одинаковое значение): первый из них — виртуальный адрес памяти (VMA), предназначенный для внутреннего использования при запуске модуля; а второй представляет собой загружаемый адрес памяти (LMA), указывающий расположение памяти, куда должен загружаться раздел. В случае встраиваемого модуля все ссылки разрешаются компоновщиком в абсолютные адреса, поэтому все адресные ссылки являются полностью разрешимыми и неперемещаемыми. Описанный процесс определения адреса модуля называется его *размещением* (locating). Некоторые системы имеют специальную утилиту, преобразующую вырабатываемый компоновщиком перемещаемый код в модуль с абсолютной адресацией, но компоновщик GCC имеет собственную встроенную возможность такого преобразования.

Компоновщик считывает выработанные компилятором объектные файлы и объединяет их в новый объектный файл (который также называется выполнимым файлом). Этот объектный файл имеет разделы (секции). Каждый раздел имеет имя и размер. Компоновщик объединяет разделы с одинаковыми именами в один раздел. Некоторые из разделов содержат выполняемый код, в других находятся данные с заданными начальными значениями, в остальных содержатся неинициализированные данные. В разделе с неинициализированными данными, как правило, содержатся только имя раздела и его размер.

Пример сценария компоновщика № 1

Следующий пример можно использовать для получения скомпонованного объектного файла. Он берет разделы из различных входных объектных файлов, объединяет их и размещает по указанным адресам:

```
SECTIONS
{
    . = 0x0100000;
    .text : {
        *(.text)
    }
    . = 0x8000000;
    .data : {
        *(.data)
    }
    .bss : {
        *(.bss)
    }
}
```

Ключевое слово **SECTIONS** указывает, что ниже приводится карта распределения памяти для скомпонованного объектного модуля. Операторы между начальной и конечной скобками команды **SECTIONS** предназначены для назначения расположения в памяти генерируемого кода.

Точка представляет собой специальную переменную, которая содержит текущий адрес, также называемый счетчиком адреса (location counter), для помещения данных в выходной файл. Первый оператор в приведенном примере устанавливает те-

кущий адрес равным абсолютному адресу **0x0100000**. Если бы он не был установлен, по умолчанию использовалось бы значение 0. После установки текущего адреса он будет автоматически увеличиваться при добавлении элементов в выходной файл.

Оператор **.text{...}** помещает начало раздела **.text** выходного файла по текущему адресу. В скобках указываются элементы, которые должны включаться в раздел **.text** выходного файла. В приведенном примере раздел **.text** будет содержать все разделы **.text** входных файлов. Здесь можно указать конкретные имена входных файлов, звездочка "*" соответствует всем именам.

После раздела **.text** значение счетчика адресов устанавливается равным **0x08000000**. Это адрес раздела **.data** выходного файла. Объединение всех разделов **.data** входных файлов в один раздел **.data** приводит к перемещению счетчика адресов. После вставки содержимого раздела **.data** счетчик адресов будет указывать на следующий за разделом **.data** адрес, куда и помещаются данные раздела **.bss**.

Пример сценария компоновщика № 2

Следующий сценарий компоновщика указывает расположение разделов в форме, которая, возможно, будет вами использоваться для создания объектного файла для встраиваемых систем. В сценарии указываются адреса для ОЗУ (RAM) и ПЗУ (ROM):

```
MEMORY
{
    rom (rx) : ORIGIN = 0x00000000, LENGTH 1024K
    ram (rwx) : ORIGIN = 0x00100000, LENGTH 512K
}
SECTIONS
{
    .text rom : {
        *(.text)
    }
    .data ram : {
        _StartOfData = . ;
        *(.data)
        _EndOfData = . ;
    } >rom
    .bss : {
        *(.bss)
    }
    _HeapLocation = .
    _StackLocation = 0x80000000
}
```

Этот пример начинается с ключевого слова **MEMORY**, которое используется для присвоения имен блокам адресного пространства выходного кода. Этот метод может использоваться для того, чтобы разбивать адресное пространство выходного кода на отдельные блоки и с помощью дополнительных инструкций вставлять данные от-

дельных разделов в конкретные блоки памяти. В приведенном примере ключевое слово **MEMORY** применяется для задания месторасположения и размера ОЗУ и ПЗУ и присвоения им имен для дальнейшего использования.

Необязательные атрибуты памяти **rx** означают, что содержимое памяти может считываться и выполняться. Атрибуты **rwx** означают, что содержимое доступно для считывания и записи и может выполняться. Если опустить атрибуты, устанавливаются все разрешения.

Определенные в сценарии области памяти позволяют использовать для установки адресов имена вместо чисел. Раздел **.text** выходного файла помещается в ПЗУ, поскольку его имя и расположение определяется как **.text rom**, а раздел **.data** помещается в ОЗУ, поскольку он определяется как **.data ram**. Именованные разделы помещаются в том порядке, в котором они указаны, поэтому если адрес не указан, следующий раздел располагается после предыдущего. Например, раздел **.bss** будет находиться непосредственно после раздела **.data**.

Такие символы, как **_StartOfData** и **_EndOfData**, включенные в сценарий, в процессе компоновки становятся глобальными переменными. Эти имена могут использоваться в программе для получения непосредственного доступа из программы к тому разделу памяти, в котором они установлены. Символ **_HeapLocation** определен как адрес в ОЗУ, следующий непосредственно после раздела **.bss**, а для символа **_StackLocation** установлен абсолютный адрес **0x80000000**.

Некоторые другие команды сценариев компоновщика ld

Команда **OUTPUT_FORMAT** очень важна для получения выходного выполняемого модуля в такой форме, которая позволяет загружать его в вашей системе. Например, следующая команда сформирует выходной объект в шестнадцатеричном формате процессора Intel:

```
OUTPUT_FORMAT("ihex")
```

Для этой команды доступны и описатели двоичных файлов (BFD), в частности **"binary"**, **"ihex"** (для шестнадцатеричного формата Intel), **"srec"** (для S-записей), **"coff-sh"** (для SH-2) и **"coff-m68k"** (для CPU32) (и это не полный список). Использование в сценарии команды **OUTPUT_FORMAT** аналогично использованию команды **--oformat** командной строки компоновщика, которая имеет тот же набор возможных описателей двоичных файлов.

Команду **INPUT** можно использовать для указания списка библиотек и/или объектных файлов, которые вы хотите включить в каждую ссылку. Например, в следующем примере в ссылки будут включаться две библиотеки и один объектный файл:

```
INPUT(libc.a libg.a startmod.o)
```

С помощью команды **OUTPUT_FILENAME** можно указать имя выходного файла, как это показано в следующем примере:

```
OUTPUT_FILENAME("loadable.out");
```



Глава 18

Выход компилятора

Основное назначение компилятора заключается в выработке объектных файлов, библиотек, содержащих объектные файлы, и выполняемых программ. Кроме того, компилятор можно использовать для получения других выходных результатов. Хотя это требуется и очень редко, тем не менее, компилятор может оказаться полезным в ситуациях, когда бывает трудно найти выход из возникшей проблемы.

Имеются опции, позволяющие определить "мнение" компилятора о синтаксисе программы, узнать, где он осуществляет поиск подпроцессов и библиотек, и получить листинг промежуточного кода, выработанный в результате обработки вашей программы. У вас есть возможность просмотреть полный список всех заголовочных файлов, включаемых в программу, и автоматически на основании исходного кода сформировать зависимости для компоновочного файла.

Сведения о программе

Компилятор формирует подробные внутренние таблицы, содержащие данные о компилируемой программе. Существуют доступные опции командной строки, которые позволяют считать часть этих данных. У вас есть возможность не только изучить дерево синтаксического разделения, содержащее внутреннее представление вашего кода, но и получить полный список всех включаемых заголовочных файлов, узнать время компиляции и объем памяти, занимаемый каждым модулем программы. Для программ, написанных на языке C++, можно, кроме того, получить взаимоотношения определений классов.

Дерево синтаксического разбора

Компилятор выполняет синтаксический разбор программы (parsing), преобразовывая ее во внутреннюю древовидную структуру. Это дерево, представляющее струк-

туру входного исходного кода, можно записать в файл с расширением `.tu` с помощью опции `-fdump-translation-unit`. Для этого подается примерно такая команда:

```
$ gcc -fdump-translation-unit showdump.c -o showdump
```

Файл, созданный с помощью этой команды, `showdump.c.tu`, содержит текстовое представление дерева синтаксического разбора. Каждый узел имеет свой номер (`@1`, `@2` и т.д.), и структура программы представлена ссылками каждого узла на номера других узлов.

Можно в некоторых пределах управлять объемом данных, выводимых для каждого узла. Так, следующая команда формирует дерево синтаксического разбора, содержащее информацию о внутренней адресации компилятора. Эту информацию можно использовать для определения перекрестных ссылок в дереве синтаксического разбора и нахождения внутренних адресов, получаемых с помощью опции `-d` (описанной ниже в этой главе):

```
$ gcc -fdump-translation-unit-address showdump.c -o showdump
```

Приведенные ниже две команды выведут листинги с большим и с меньшим объемом информации соответственно:

```
$ gcc -fdump-translation-unit-all showdump.c -o showdump
$ gcc -fdump-translation-unit-slim showdump.c -o showdump
```

Дерево, полученное любой из приведенных выше команд, читается очень легко. Следующий фрагмент дерева синтаксического разбора показывает, что каждый его узел описывается уникальным идентификатором и несколько более описательным именем. Кроме того, дерево содержит список атрибутов:

<code>@1 function_decl</code>	<code>name: @2</code>	<code>type: @3</code>	<code>srcp: showdump.c:5</code>
	<code>chan: @4</code>	<code>args: @5</code>	<code>extern</code>
<code>@2 identifier_node</code>	<code>strg: main</code>	<code>lntg: 4</code>	
<code>@3 function_type</code>	<code>size: @6</code>	<code>algn: 64</code>	<code>retn: @7</code>
	<code>prms: @8</code>		
<code>@4 var_decl</code>	<code>name: @9</code>	<code>type: @7</code>	<code>srcp: showdump.c:3</code>
	<code>chan: @10</code>	<code>init: @11</code>	<code>size: @12</code>
	<code>algn: 32</code>	<code>used: 1</code>	
<code>@5 parm_decl</code>	<code>name: @13</code>	<code>type: @7</code>	<code>scpe: @1</code>
	<code>srcp: showdump.c:4</code>		<code>chan: @14</code>
	<code>argt: @7</code>	<code>size: @12</code>	<code>algn: 32</code>
	<code>used: 0</code>		

На этом уровне синтаксического дерева большинство атрибутов определены по отношению к другим узлам дерева. Например, атрибут `name` представляет собой номер узла, содержащего атрибуты `strg` (строка) и `lngth` (длина). Некоторые узлы, например, `function_type`, содержат атрибут `algn` (выравнивание). Переменные, в частности аргументы и объявления, содержат атрибуты `type` и `name` и, кроме того, атрибут `used`, представляющий собой количество использований переменной в программе. Многие узлы имеют атрибут `srcp` (положение источника), который указывает имя и номер строки файла исходного кода, откуда был получен данный узел.

Заголовочные файлы

Опция `-H`, которую также можно записать в форме `--trace-includes`, генерирует многоуровневый список всех включаемых файлов. Следующий пример представляет собой выходной код, сгенерированный на системе Linux для программы, написанной на языке C, включающей только один заголовочный файл `stdio.h`:

```
. /usr/lib/gcc-lib/i586-pc-linux-gnu/3.2/include/stdio.h
...
... /usr/include/features.h
... /usr/include/sys/cdefs.h
... /usr/include/gnu/stubs.h
... /usr/lib/gcc-lib/i586-pc-linux-gnu/3.2/include/stddef.h
... /usr/include/bits/types.h
... /usr/lib/gcc-lib/i586-pc-linux-gnu/3.2/include/stddef.h
... /usr/include/bits/pthreadtypes.h
.... /usr/include/bits/sched.h
... /usr/include/libio.h
.... /usr/include/_G_config.h
.... /usr/lib/gcc-lib/i586-pc-linux-gnu/3.2/include/stddef.h
.... /usr/include/wchar.h
..... /usr/lib/gcc-lib/i586-pc-linux-gnu/3.2/include/stddef.h
..... /usr/include/bits/wchar.h
.... /usr/include/gconv.h
.... /usr/include/wchar.h
..... /usr/lib/gcc-lib/i586-pc-linux-gnu/3.2/include/stddef.h
..... /usr/lib/gcc-lib/i586-pc-linux-gnu/3.2/include/stddef.h
... /usr/lib/gcc-lib/i586-pc-linux-gnu/3.2/include/stdare.h
.. /usr/include/bits/stdio_lim.h

Multiple include guards may be useful for:
/usr/include/bits/pthreadtypes.h
/usr/include/bits/sched.h
/usr/include/bits/stdio_lim.h
/usr/include/gnu/stubs.h
```

Каждый уровень включения указывается количеством точек перед именем файла. Кроме того, в нижней части листинга приводятся имена заголовочных файлов, которые, возможно, необходимо исправить, поскольку включение любого из них более одного раза может привести к возникновению проблем, связанных с многократными объявлениями.

Необходимый программе объем памяти

У компилятора можно запросить отчет об объеме памяти, требуемой для откомпилированной программы, а также подробные данные о том, каким образом эта память была выделена. Следующий листинг содержит пример отчета о требуемом объеме памяти:

RTX	Number	Bytes	% Total
address	7	56	0.664
const_int	129	1032	12.239
const_double	21	336	3.985
const_vector	19	152	1.803

pc	1	8	0.095
reg	14	224	2.657
mem	216	3456	40.987
symbol_ref	391	3128	37.097
cc0	1	8	0.095
plus	1	16	0.190
eq	1	16	0.190
Total	801	8432	
Size	Allocated	Used	Overhead
8	8192	6216	184
16	12 k	4192	180
32	8192	3392	88
64	32 k	28 k	288
512	28 k	24 k	196
1024	4096	1024	28
112	52 k	42 k	416
20	8192	2580	104
Total	152 k	112 k	1484
String pool			
entries	452		
identifiers	452 (100.00%)		
slots	16384		
bytes	4805 (3339 overhead)		
table size	64k		
coll/search	0.0168		
ins/search	0.7609		
avg. entry	10.63 bytes (+/- 5.78)		
longest entry	36		

Из приведенного листинга можно определить объемы памяти, выделенной для различных частей программы, и количество использований каждой из выделенных областей. Такие данные могут оказаться особенно полезными для анализа больших программ и объектных модулей, предназначенных для встраиваемых систем.

Время компиляции программы

Опция **-time** может использоваться при компиляции и компоновки для вывода компилятором **gcc** времени, затраченного на выполнение каждого отдельного процесса. Например, следующая команда компилирует три программы на языке *C* на ассемблерный язык, вызывает ассемблер для формирования отдельного объектного файла для каждой программы и использует **collect2** для их компоновки:

```
gcc -time getshow.c strmaker.c showstring.c -o getshow
# cc1 0.15 0.02
# as 0.01 0.00
# cc1 0.08 0.03
# as 0.01 0.01
# cc1 0.13 0.03
# as 0.01 0.00
# collect2 0.13 0.05
```

Первое из двух приведенных для каждого процесса значений представляет собой пользовательское время (время, затраченное на выполнение кода подпроцессов), а второе значение — системное время (время, затраченное процессом на выполнение системных вызовов). Действительные показания отсчетов времени в листинге не приводятся. Общее время, затраченное на выполнение всего процесса `gcc`, включая отсчеты времени, может быть получено при запуске `gcc` с помощью стандартной утилиты `time`, как показано в следующем примере:

```
$ time gcc -time getshow.c strmaker.c showstring.c -o getshow
```

Промежуточное дерево компиляции C++

Компилятору `g++` можно указать, что он должен выводить код промежуточного языка, который генерируется на верхнем уровне при предварительной трансляции. Вывод может выполняться в различных точках процесса компиляции. Следующая команда выведет промежуточный код после его генерации до внесения каких-либо изменений или оптимизации:

```
$ g++ -fdump-tree-original minmax.cpp -o minmax
```

Код промежуточного языка можно вывести также и после выполнения оптимизации:

```
$ g++ -fdump-tree-optimized minmax.cpp -o minmax
```

Процесс подстановки кода `inline`-функций выполняется в коде промежуточного языка, а его результаты можно вывести с помощью следующей команды:

```
$ g++ -fdump-tree-inlined minmax.cpp -o minmax
```

Формат выходных данных может задаваться указанием модификатора в конце имени каждой опции вывода кода. Добавление модификатора `-address` к концу опции приведет к включению адресной информации, соответствующей адресной информации, выводимой с помощью опции `-d` (описанной ниже в этой главе). Для уменьшения объема информации, включаемой в листинги, укажите модификатор `-slim`, а для его увеличения — модификатор `-all`. Например, следующая команда выведет код промежуточного языка после оптимизации в описательной форме:

```
$ g++ -fdump-tree-optimized-all minmax.cpp -o minmax
```

Иерархия классов в программах на языке C++

Компилятору `g++` можно указать, чтобы он выводил полную иерархию классов и таблицы виртуальных функций вашей программы. В выходные данные включается полная иерархия используемых программой системных классов, поэтому размер выходных данных может быть довольно большим. Следующая команда приведет к компиляции и выводу полной иерархии классов программы `minmax.cpp`:

```
$ g++ -fdump-class-hierarchy minmax.cpp -o minmax
```

В результате выполнения этой команды будет создан исполняемый файл `minmax` и текстовый файл `minmax.cpp.class`, содержащий иерархию классов. Следующая

команда дает тот же результат, но в иерархию классов также будет включена адресная информация, которая может пересекаться с информацией, выводимой по опции **-d**:

```
$ g++ -fdump-class-hierarchy-address -da minmax.cpp -o minmax
```

Опция **-d** выводит некоторую внутреннюю информацию компилятора, это описано далее в этой главе.

Объем выводимой информации можно уменьшить, используя следующую опцию:

```
$ g++ -fdump-class-hierarchy-slim minmax.cpp -o minmax
```

Файл большого размера можно получить с помощью следующей опции:

```
$ g++ -fdump-class-hierarchy-all minmax.cpp -o minmax
```

Информация для включения в сценарий Makefile

Имеется набор опций, которые можно использовать для указания компилятору сканировать файлы исходного кода и генерировать зависимости для вставки в make-файл. Например, следующая программа включает два заголовочных файла:

```
/* getshow.c */
#include "strmaker.h"
#include "showstring.h"
int main(int argc,char *argv[])
{
    char *string;
    string = strmaker();
    showstring(string);
}
```

Следующая команда компилятора считывает файл исходного кода и формирует строку зависимостей для make-файла. В этом примере заголовочный файл **strmaker.h** включает файл **motback.h**:

```
$ gcc -M getshow.c
getshow.o: getshow.c strmaker.h motback.h showstring.h
```

Опция **-M** устанавливает опцию **-E**, которая подавляет весь вывод, кроме строки зависимостей. Если вы желаете продолжить компиляцию после формирования строки зависимостей, то выполните следующую команду:

```
$ gcc -MD getshow.c -o getshow
```

Эта команда сгенерирует выполнимый файл **getshow** и запишет строку зависимостей в файл **getshow.d**. Для указания имени файла может использоваться опция **-MF**. Это показано в следующем примере, где строка зависимостей записывается в файл **depends.text**:

```
$ gcc -MD -MF depends.text getshow.c -o getshow
```

Кроме того, опцию **-MF** можно использовать совместно с опцией **-M** для подавления компиляции и записи строки зависимостей в файл:

```
$ gcc -M -MF depends.text getshow.c
```

Другим способом имя выходного файла можно указать с помощью переменной среды **DEPENDENCIES_OUTPUT**.

Опции **-M** и **-MM** обнаружат отсутствие заголовочного файла и выведут об этом отчет. При необходимости подавления такого рода сообщений можно указать опцию **-MP** вместе с опциями **-M** и **-MM**, которая генерирует фиктивные ссылки для каждого заголовочного файла.

Опция **-MT** может использоваться совместно с опцией **-M** или **-MM** для указания имени каждого заголовочного файла, как показано в примере:

```
$ gcc -M -MT spang.o getshow.c  
spang.o: getshow.c strmaker.h motback.h showstring.h
```

Информация о самом компиляторе

Существует несколько доступных опций, которые позволяют определить тип применяемого компилятора и параметры его конфигурации. Например, номер версии компилятора можно определить с помощью следующей команды:

```
$ gcc -dumpversion
```

Для определения целевой машины, т.е. типа того компьютера, для которого компилятор создает объектные файлы, выполните следующую команду:

```
$ gcc -dumpmachine
```

Отчет о времени компиляции

Для генерации листинга времени, затраченного на отдельных этапах компиляции, используется опция **-ftime-report**. Эта опция в основном предназначена для разработчиков компиляторов, тем не менее, ее можно использовать для определения относительной сложности различных программ. Результат компиляции при использовании этой опции выглядит следующим образом:

```
Execution times (seconds)  
garbage collection : 1.13 (23%) usr 0.00 ( 0%) sys 0.50 (10%) wall  
life analysis : 0.01 ( 0%) usr 0.00 ( 0%) sys 0.00 ( 0%) wall  
preprocessing : 0.43 ( 9%) usr 0.08 (24%) sys 1.00 (20%) wall  
lexical analysis : 0.38 ( 8%) usr 0.10 (29%) sys 0.00 ( 0%) wall  
parser : 2.72 (56%) usr 0.14 (41%) sys 3.00 (60%) wall  
expand : 0.02 ( 0%) usr 0.00 ( 0%) sys 0.00 ( 0%) wall  
varconst : 0.05 ( 1%) usr 0.00 ( 0%) sys 0.50 (10%) wall  
integration : 0.03 ( 1%) usr 0.01 ( 3%) sys 0.00 ( 0%) wall  
local alloc : 0.01 ( 0%) usr 0.00 ( 0%) sys 0.00 ( 0%) wall  
global alloc : 0.01 ( 0%) usr 0.00 ( 0%) sys 0.00 ( 0%) wall  
rest of compilation : 0.00 ( 0%) usr 0.01 ( 3%) sys 0.00 ( 0%) wall  
TOTAL : 4.84 0.34 5.00
```

В приведенном листинге значения показаны в секундах и в процентном отношении продолжительности каждого этапа по отношению к общей продолжительности компиляции. Время `cvt` — это время, потраченное на выполнение кода компилятора. Время `sys` соответствует времени, затраченному на выполнения системных вызовов (таких как ввод и вывод), а время `wall` представляет собой фактические затраты времени.

Ключи подпроцессов

Программа `gcc` является оболочкой (верхним уровнем) для других программ, таких как компилятор отдельного языка, ассемблер и компоновщик. Имена подпроцессов и передаваемые им опции настраиваются и устанавливаются во время конфигурирования и компиляции `gcc`. Для определения спецификаций, которые были использованы для создания аргументов командной строки подпроцессов, введите следующую команду:

```
$ gcc -dumprspcs | more
```

Спецификация для опций и аргументов, передаваемых подпроцессу, состоит из одной строки. Набор спецификаций по умолчанию для каждого основного подпроцесса встроен в `gcc` и автоматически становится частью компилятора, тем не менее, действующие по умолчанию строки спецификации можно подменять при конфигурировании компилятора.

Вот пример информации, находящейся в строке спецификации, это — спецификация, используемая для вызова препроцессора `C`:

```
*cpp:  
%{posix:-D_POSIX_SOURCE} %{pthread:-D_REENTRANT}
```

С помощью этой спецификации при вызове `cpp` опция `--posix` командной строки `gcc` приведет к использованию в командной строке `cpp` опции `-D_POSIX_SOURCE`, а опция `--pthread` — к использованию в командной строке `cpp` опции `-D_REENTRANT`.

Строка спецификации, устанавливающая условия для всех возможных опций, передаваемых подпроцессу, может быть довольно сложной. Примером более сложной строки спецификации (далеко не самой сложной) может служить строка вызова ассемблера:

```
*asm:  
%{v:-V} %{Qy:-Qy} %{!Qn:-Qy} %{n} %{t}
```

Давайте разберем этот пример. Если в командной строке `gcc` будет указана опция `-v`, то в командной строке ассемблера будет присутствовать опция `-V`. Если в командной строке `gcc` будет указана опция `-Qy`, она не передается ассемблеру, но если опция `-Qn` не установлена, то в командную строку ассемблера будет внесена опция `-Qy`. Если для `gcc` указаны опции `-n` или `-t`, то каждая из них будет передана ассемблеру. Остальные опции ассемблеру не передаются.

Расширенная отладочная информация компилятора

Опция `-d` может использоваться для указания системе GCC выводить внутреннюю информацию компилятора на различных этапах процесса компиляции. Полученная таким образом информация может быть полезной только для тех, кто работает над самим компилятором. Поэтому, несмотря на то, что информация достаточно подробная, она вряд ли пригодится вам в процессе отладки или анализа приложения.

Вывод информации можно организовать в одной или нескольких точках процесса компиляции. Полный набор возможностей приводится в описании опции `-d` в приложении Г. Выходные данные будут практически одними и теми же для всех точек. Они будут содержать информацию об удаленных ненужных инструкциях, выделении регистров, освобождении регистров (когда содержащееся в регистре значение удаляется) и генерируемые инструкции на внутреннем языке регистрового переноса (Register Transfer Language, RTL). Например, следующая простая программа сравнивает одно значение с другим и определяет должны ли выполняться операторы в ответвлении:

```
/* showdump.c */
int a = 44;
static int b = 22;
int main(int argc,char *argv[])
{
    if(a > b) {
        b = a;
    } else {
        a = b;
    }
}
```

Следующая команда компилирует программу и запрашивает отладочные данные непосредственно после генерации RTL-кода:

```
$ gcc -dr showdump.c -o showdump
```

Дамп отладочной информации записывается в файл `showdump.c.00 rtl` и выглядит примерно так:

```
; Function main
(note 2 0 5 NOTE_INSN_DELETED -1347440721)
(insn 5 2 6 (nil) (parallel[
  (set (reg/f:SI 7 esp)
    (and:SI (reg/f:SI 7 esp)
      (const_int -16 [0xfffffffff0])))
  (clobber (reg:CC 17 flags))
]) -1 (nil)
(nil))
(insn 6 5 7 (nil) (set (reg:SI 59)
  (const_int 0 [0x0])) -1 (nil))
```

```

(expr_list:REG_EQUAL (const_int 0 [0x0])
 (nil))

(Insn 7 6 8 (nil) (parallel[
    (set (reg/f:SI 7 esp)
        (minus:SI (reg/f:SI 7 esp)
            (reg:SI 59)))
    (clobber (reg:CC 17 flags))
  ] ) -1 (nil)
 (nil))

(Insn 8 7 3 (nil) (set (reg/f:SI 60)
    (reg/f:SI 55 virtual-stack-dynamic)) -1 (nil)
 (nil))

(note 3 8 4 NOTE_INSN_FUNCTION_BEG -1347440721)

(note 4 3 9 NOTE_INSN_DELETED -1347440721)

(note 9 4 10 NOTE_INSN_DELETED -1347440721)

(note 10 9 12 NOTE_INSN_DELETED -1347440721)

(Insn 12 10 13 (nil) (set (reg:SI 61)
    (mem/f:SI (symbol_ref:SI ("a")) [0 a+0 S4 A32])) -1 (nil)
 (nil))

(Insn 13 12 14 (nil) (set (reg:CCGC 17 flags)
    (compare:CCGC (reg:SI 61)
        (mem/f:SI (symbol_ref:SI ("b")) [0 b+0 S4 A32]))) -1 (nil)
 (nil))

(jump_insn 14 13 15 (nil) (set (pc)
    (if_then_else (le (reg:CCGC 17 flags)
        (const_int 0 [0x0]))
        (label_ref 22)
        (pc))) -1 (nil)
 (nil))

(note 15 14 16 NOTE_INSN_DELETED -1347440721)

(note 16 15 18 NOTE_INSN_DELETED -1347440721)

(Insn 18 16 19 (nil) (set (reg:SI 62)
    (mem/f:SI (symbol_ref:SI ("a")) [0 a+0 S4 A32])) -1 (nil)
 (nil))

(Insn 19 18 20 (nil) (set (mem/f:SI (symbol_ref:SI ("b")) [0 b+0 S4 A32])
    (reg:SI 62)) -1 (nil)
 (nil))

(jump_insn 20 19 21 (nil) (set (pc)
    (label_ref 28)) -1 (nil)
 (nil))

(barrier 21 20 22)

(code_label 22 21 23 2 "" "" [0 uses])

(note 23 22 24 NOTE_INSN_DELETED -1347440721)

```

```
(note 24 23 26 NOTE_INSN_DELETED -1347440721)
  (insn 26 24 27 (nil) (set (reg:SI 63)
    (mem/f:SI (symbol_ref:SI ("b")) [0 b+0 S4 A32])) -1 (nil)
    (nil))
  (insn 27 26 28 (nil) (set (mem/f:SI (symbol_ref:SI ("a")) [0 a+0 S4 A32])
    (reg:SI 63)) -1 (nil)
    (nil))
  (code_label 28 27 29 3 "" "" [0 uses$])
  (note 29 28 33 NOTE_INSN_FUNCTION_END -1347440721)
  (insn 33 29 34 (nil) (clobber (reg/i:SI 0 eax)) -1 (nil)
    (nil))
  (insn 34 33 31 (nil) (clobber (reg:SI 58)) -1 (nil)
    (nil))
  (code_label 31 34 32 1 "" "" [0 uses$])
  (insn 32 31 35 (nil) (set (reg/i:SI 0 eax)
    (reg:SI 58)) -1 (nil)
    (nil))
  (insn 35 32 0 (nil) (use (reg/i:SI 0 eax)) -1 (nil)
    (nil))
```

Информация о файлах и каталогах

Компилятор GCC имеет набор опций, при указании которых компилятор осуществляет поиск на диске требуемых файлов. Поскольку каталоги, в которых компилятор выполняет поиск библиотек, определяются конфигурацией системы, вам может понадобиться определить местоположение используемой библиотеки. Это можно сделать с помощью опции **-print-file-name**. Например, так можно определить местоположение библиотеки **libgcc.a**:

```
$ gcc -print-file-name=libgcc.a
/usr/lib/gcc-lib/i586-pc-linux-gnu/3.2/libgcc.a
```

Вообще опцию **-print-file-name** можно использовать для определения местоположения любой библиотеки, но для библиотеки **libgcc.a** есть собственная опция:

```
$ gcc -print-libgcc-file-name
/usr/lib/gcc-lib/i586-pc-linux-gnu/3.2/libgcc.a
```

Аналогично вы имеете возможность определить полный путь к запускаемым программам таких внутренних подпроцессов, как **cc1** и **cc1obj**. Например, для определения местоположения файла **f771**:

```
$ gcc -print-prog-name=f771
/usr/lib/gcc-lib/i586-pc-linux-gnu/3.2/f771
```

С помощью следующей команды можно определить каталог установки GCC и полный путь как к программам, так и к библиотекам:

```
$ gcc -print-search-dirs >path.txt
```

Результат выполнения этой команды может быть довольно большим, а пути выводятся в виде одной непрерывной строки, поэтому результат удобнее перенаправить в файл, который в дальнейшем можно проанализировать с помощью программы-редактора. Сначала в файле результатов указывается каталог установки компилятора, а затем пути к программам и библиотекам. Некоторые пути определяются с помощью алгоритма, который предоставляет их в более описательном виде, чем это необходимо, но если вы хотите знать порядок поиска, то его вполне можно определить на основе результата выполнения этой команды.



Глава 19

Реализация алгоритмического языка

Компилятор GCC состоит из двух уровней. *Верхний уровень* (front back) выполняет синтаксический разбор входного языка. *Нижний уровень* (back end) генерирует машинный код, который должен выполняться на пред назначенной целевой машине. GCC разработан для компиляции программ с неограниченного количества языков. Для каждого из поддерживаемых им языков имеется свой верхний уровень компилятора. В GCC предусмотрена открытая возможность его расширения для компиляции программ на любом дополнительном пользовательском языке. Допустим, вы придумали язык программирования. Если вы напишете свой верхний уровень для GCC, то к нему можно будет подключать любой из существующих нижних уровней (они также называются *портами* GCC). Так что ваш компилятор будет переносимым, и вы сможете компилировать программы на своем особом языке программирования в машинный код для целого ряда платформ.

Этот замысел понятен, красив и предельно прост. Но реализовать его совсем не так легко, как это может показаться на первый взгляд. Допустим, что у нас есть синтаксический разделитель (parser), способный распознавать элементы формализованного языка программирования. Для реализации компилятора требуется совместить этот верхний уровень с остальным уже существующим хозяйством. Это значит, что *парсер*, наш грамматический разделитель, должен вырабатывать выходную информацию в распознаваемом нижним уровнем формате. Верхний уровень GCC отделен от нижнего уровня не настолько, насколько хотелось бы. Поэтому над "сырым" выходом парсера придется еще поколдовать. И при этом надо помнить о постоянном развитии стандартных библиотек, используемых программами.

Этапы компиляции программы

От компилятора GCC, как от любого компилятора, требуется принимать на входе исходный код и вырабатывать из него готовые к запуску программы, которые могут непосредственно выполняться вычислительной машиной. Для выполнения этой цели компилятор предпринимает ряд последовательных действий:

- *Лексический анализ* (Lexical Scan). На этом этапе исходный код считывается и разделяется на элементы (*tokens*) лексическим сканером. Обычно исходный файл побайтно считывается в потоке. Буквенные символы последовательночитываются и собираются в цепочки, при этом выделяются элементы языковых конструкций — *лексемы*. Выделенные последовательности сразу же сортируются по категориям: имена, числа и пунктуация. Каждый формализованный язык имеет свой собственный набор правил, на этом этапе правила применяются для проверки соответствия элементов всех категорий критериям. Здесь принимается решение о возможности дальнейшей обработки кода.
- *Синтаксический разбор* (Parsing). Лексемы исходного кода имеют некоторые отношения между собой, определяемые, главным образом, их взаимным относительным положением в выходном потоке лексического сканера. Синтаксический разделитель (парсер) определяет тип каждого элемента (ключевое слово, имя символа, число и т.п.) и использует эту информацию для преобразования всего исходного файла в древовидную логическую структуру. Узлы этого дерева представляют объявления типов и данных, функции, отдельные операторы и т.п. Это дерево содержит все элементы программы и отображает все взаимодействия между элементами.
- *Обрезка* (Pruning). Далее с помощью анализа построенного дерева проводится некоторая оптимизация программы. Неиспользуемые и избыточные части структуры отсекаются. Некоторые части дерева могут быть перенесены в другое расположение для предотвращения излишне частого повторения некоторых групп операторов, лишних проходов, для сокращения накладных расходов на передачу управления и т.п.
- *Трансляция на промежуточный язык*. Содержание дерева разделения преобразуется в код RTL. RTL — язык регистрового переноса, "Register Transfer Language". Он представляет собой особый унифицированный псевдо-ассемблерный язык, содержащий обобщенные операционные коды инструкций гипотетической вычислительной машины. Дерево "раскатывается" в линейную последовательность RTL-инструкций. При этом проводится необходимая реорганизация, добавляются ветви, определяемые древовидной структурой, проверки условий. Выбираются варианты реализации операторов ветвления типа *case/switch*, циклов и т.п. Значительная часть трансляции, выполняемой на этом этапе, зависит от предназначаемой целевой платформы. То есть, RTL-код генерируется в терминах целевой машины и содержит такие вещи, как информация для размещения в выделяемых регистрах.
- *Оптимизация промежуточного кода*. Выполняется ряд оптимизаций полученного RTL-кода. Они включают в себя исключение излишней рекурсивности,

избыточного дробления подвыражений, оптимизацию переходов и ряд других ассемблерных оптимизаций. Этот этап идеально подходит для основной оптимизации кода. Она здесь одинаково применима как к верхнему уровню любого языка, так и к нижнему уровню, ориентированному на любую платформу.

- *Трансляция в ассемблерный код.* Окончательный вариант RTL-кода переводится на ассемблерный язык пред назначаемой целевой машины и записывается в файл.
- *Ассемблирование.* На этом этапе действует ассемблер для преобразования полученной программы на ассемблерном языке в объектный файл, содержащий машинный код. Этот файл еще не имеет формата выполняемого файла — он действительно содержит выполнимый машиной код, но еще не имеет процедур, необходимых для его загрузки. Кроме того, он наверняка содержит неразрешимые обращения к подпрограммам из других модулей.
- *Компоновка.* Компоновщик собирает выработанные ассемблером объектные файлы (некоторые из них могут находиться в объектных библиотеках) в машинную программу. Готовая программа имеет соответствующий системе формат, пригодный для ее загрузки на выполнение в пред назначенной машине.

Обратите внимание, что логическое разделение верхнего и нижнего уровня проходит между синтаксическим разделителем языка и генератором ассемблерного кода. Они взаимодействуют между собой через дерево синтаксического разбора. Любой синтаксический разделитель исходного языка (парсер), способный вырабатывать древовидную структуру, может быть подключен к нижнему уровню через генератор кода RTL. Следовательно, в GCC возможна компиляция исходного кода на любом языке, имеющем такой парсер. С другой стороны, перенос программ, написанных на любом из поддерживаемых верхним уровнем языков, возможен на любую машину, имеющую транслятор языка RTL в код своего ассемблера.

Это описание, конечно, очень упрощено и не отражает некоторые частные сложности. Но главное, что такой процесс возможен и он действительно работает в GCC.

Лексический анализ

Компилятор считывает код программы как поток буквенных знаков, группирует их в сочетания и выводит последовательность выделенных элементов исходного языка для их дальнейшей обработки в выходном потоке лексем. Каждый элемент может быть числом, именем или знаком пунктуации. Например, следующая строка содержит семь элементов:

```
if (grimle <= 43.1) {
```

Процесс разделения строки на лексические элементы (лексемы), называется *лексическим сканированием* (*lexical scan*). Программа лексического сканирования называется *lex-сканером* (по английски — просто "lex"). Механизм лексического сканирования универсален. Отличаются только правила, определяющие, какие буквенные знаки могут применяться в именах и какие являются знаками пунктуации. Эта про-

щедура настолько устоялась, что существуют стандартные утилиты — генераторы программ лексического сканирования. В системах UNIX для этого служит стандартная программа `lex`, ее эквивалент проекта GNU имеет название `flex`. На входе вы задаете свой набор правил и на выходе получаете программу, которая вырабатывает поток лексических элементов вашего языка.

Пример построения простого lex-сканера

В качестве простого определения правил для lex-сканера, который выделяет два ключевых слова `howdy` и `now`, можно привести такой сценарий (допустим, он записан в файле `howdy.lex`):

```
%%
howdy printf("(The word is 'howdy')");
now printf("(The time is %ld)",time(0L));
%%
```

Пары знаков процента “`%%`” применяются, чтобы отметить начало и конец списка ключевых слов. После каждого ключевого слова стоит соответствующая ей команда — оператор языка C. Команда должна быть помещена в программу, вырабатываемую в соответствии с этим сценарием. Выполнение следующих команд создаст программу `lex.yy.c` и скомпилирует ее в выполнимый файл `howdy`:

```
$ flex howdy.lex
$ gcc lex.yy.c -lfl -o howdy
```

Программа `lex.yy.c` в нашем случае содержит более 1500 строк кода на языке C, она использует функции библиотеки `libfl.a`. Одна из причин такого большого размера кода программы состоит в том, что она содержит подробные комментарии. Если вместо программы GNU `flex` вы используете стандартную утилиту UNIX `lex`, то команды будут немного отличаться:

```
$ lex howdy.lex
$ gcc lex.yy.c -lfl -o howdy
```

Полученную готовую программу можно запустить из командной строки. После запуска весь ввод с клавиатуры будет отображаться на выходе, и только два ключевых слова `howdy` и `now` каждый раз будут заменяться строками с вызовами функции `printf()`.

Построение lex-сканера с использованием регулярных выражений

Следующие определения для lex-сканера (назовем этот файл `kwords.lex`) распознают ключевые слова `switch` и `case`, пунктуацию, целые числа и скобки:

```
%%
switch printf("SWITCH ");
case printf("CASE ");
[a-zA-Z] [_a-zA-Z0-9]* printf("WORD(%s) ",yytext);
[0-9]+ printf("INTEGER(%s) ",yytext);
```

```
\{ printf("LEFTBRACE ");
\} printf("RIGHTBRACE ");
%%
```

Первые два правила определяют соответствия для ключевых слов `true` и `case`. Третье правило соответствует любому имени, которое начинается с большой или маленькой буквы и может содержать большие и маленькие буквы, цифры и символы подчеркивания. Заметьте, что выходная строка содержит поле `ytext` — это указатель на строку, содержащую считанный элемент программы. Четвертое правило относится к строкам, которые содержат последовательность цифр. Последние два правила — для фигурных скобок. Полученный из этих определений лексический анализатор сможет извлечь все элементы из следующего входного текста (запишем его в файле `kwtry.text`):

```
blatz {
    switch big_time_do
    case HamFram
    case 889
} dendl
```

Дальнейшая последовательность команд сначала компилирует сценарий генератора lex-сканера `kwords.lex` в программу `kwords`. Затем полученная программа используется для последовательного выделения лексем (tokens) из исходного файла `kwtry.text`:

```
$ flex kwords.lex
$ gcc lex.yy.c -lfl -o kwords
$ cat kwtry.text | kwords
WORD(blatz) LEFTBRACE
SWITCH WORD(big_time_do)
CASE WORD(HamFram)
CASE INTEGER(889)
RIGHTBRACE WORD(dendl)
```

Синтаксический разбор

В этом разделе рассматривается специально разработанный пример. Он демонстрирует использование lex-сканера для считывания лексем (tokens) исходной программы и применение синтаксического разделителя (parser) для логического построения полученных лексических элементов. Также будет рассмотрен порядок вызова функций C, которым передается выводимая парсером информация. Обычно в GCC на выходе парсера вырабатывается промежуточный код на языке RTL, но в нашем примере только будут выводиться текстовые строки, которые комментируют генерируемый код.

Программа, выполняющая синтаксическое разделение, может быть сгенерирована стандартной утилитой UNIX `yacc`. Ее имя — сокращение от "Yet Another Compiler Compiler", что можно перевести как "Компилятор Дополнительного Компилятора". Соответствующая ей утилита GNU называется `bison`. Обе программы почти идентичны по назначению и способу применения.

Наш пример основывается на очень простом языке `clang`, который воспринимает команды для прорисовки цветных кругов и прямоугольников в указанном расположении графического поля. Вот пример программы на языке `clang` (сохраним ее в файле `figures clang`):

```
set color blue;
set location (100,200);
draw circle 30;
set color red;
set location (250,200);
draw rectangle (10,10);
```

Операторы `set` используются для установки цвета и координат положения следующей фигуры, а операторы `draw` прорисовывают фигуру указанного типа и размера.

Далее следует содержимое файла лексических определений этого языка (файл называется `clang.lex`):

```
/* clang.lex */
%{
#include "clang.tab.h"
extern int yyval;
extern char *yytext;
%}
%%
set          { return(SETTOKEN); }
color         { return(COLORTOKEN); }
location      { return(LOCATIONTOKEN); }
draw          { return(DRAWTOKEN); }
circle         { return(CIRCLETOKEN); }
rectangle     { return(RECTANGLETOKEN); }
\;           { return(SEMICOLON); }
\,           { return(COMMA); }
\(
\)
[0-9]+       { yyval = atoi(yytext);
                return(NUMBER);
            }
[a-zA-Z][a-zA-Z0-9]* { yyval = strdup(yytext);
                return(NAME);
            }
\n           /* игнорировать конец строки */
[\t]+        /* игнорировать лишние пробелы */
%%
```

Этот листинг нуждается в небольшом комментарии. Включаемый по директиве `#include` заголовочный файл `clang.tab.h` вырабатывается утилитой `bison` из файла определений парсера, что будет описано позже. Каждое лексическое определение возвращает значение, которое указывает его тип (тип определен в заголовочном файле). Следует учитывать, что лексические определения используются для генерирования программы на языке *C*. Поэтому нужно использовать обратную наклонную черту "`\`" перед определяемыми как лексемы знаками пунктуации.

Каждый входящий элемент сохраняется в строке, на которую указывает значение переменной `yytext`. Когда это необходимо, их значения для дальнейшей передачи подпрограммам на языке C преобразовываются и сохраняются в переменной `yyval`. В этом примере значения лексем типа `NAME` сохраняются как строки, а значения лексем `NUMBER` преобразовываются в целые числа с помощью вызова функции `atoi()`.

Описания двух последних элементов не имеют соответствий. Но они необходимы для успешного считывания исходного текста после концов строк и за периодами из пробелов и отступов. Для создания парсера языка, ориентированного на построчное разделение операторов, может понадобиться специальное лексическое определение для символа перевода строки.

Далее следует файл `clang.y`, который содержит синтаксические определения рассматриваемого языка:

```
%start commands

%token SETTOKEN DRAWTOKEN COLORTOKEN
%token LOCATIONTOKEN CIRCLETOKEN RECTANGLETOKEN
%token SEMICOLON LEFTPAREN RIGHTPAREN COMMA
%token NUMBER NAME

%

commands:
/* nothing */
| commands command
;

command: SETTOKEN set SEMICOLON
| DRAWTOKEN draw SEMICOLON
;

set: COLORTOKEN NAME
{ setcolor($2); }
| LOCATIONTOKEN LEFTPAREN NUMBER COMMA NUMBER RIGHTPAREN
{ setlocation($3,$5); }
;

draw: CIRCLETOKEN NUMBER
{ drawcircle($2); }
| RECTANGLETOKEN LEFTPAREN NUMBER COMMA NUMBER RIGHTPAREN
{ drawrectangle($3,$5); }
;
%%
```

В первой строке файла указана начальная точка определения синтаксического дерева. Далее следуют определения лексических элементов (`% token ...`) как именованных констант, в генерируемом коде парсера они используются в качестве уникальных идентификаторов лексем входного потока.

Каждое вхождение определения для синтаксического разбора (`parse definition`) называется *выработкой* (`production`). Выработка имеет имя, связанное с одним или более определений синтаксического расположения (`syntax layout definitions`) в ее пра-

вой части. Синтаксические элементы правой части определения выработки разделяются между собой вертикальной чертой "|", за последним из них следует точка с запятой ";". Парсер после того как встречает известное ему имя выработки, просматривает входной поток лексем на их соответствие элементам правой части определения этой выработки.

Парсеры, генерируемые утилитами **bison** и **yacc**, относятся к типу LALR(1). Для поиска соответствий ожидаемому элементу они просматривают входной поток не далее, чем на одну лексему вперед. Семантические разделители типа LALR(1), упрощенно называемые *LR(1)-парсерами*, достаточно эффективны для поддержки современных языков программирования. Некоторые более древние языки с неоднозначным синтаксисом требуют применения более сложных и своеобразных процедур разделения кода. Все современные языки программирования разрабатываются с расчетом на использование парсеров LR(1).

Начальная выработка имеет имя **commands**. Она может быть пустой (в случае конца файла) или содержать список из одной или более выработок **commands**. Тот факт, что начальная выработка ссылается на себя прежде следующей выработки, обусловлен рекурсивными свойствами вырабатываемого парсером кода. В нашем примере в этом нет необходимости, но в общем случае делается именно так.

Выработка **command** может иметь соответствие одному из ключевых слов **set** или **draw**. От этого зависит, какая выработка будет применяться для поиска соответствий следующих лексем. Ключевое слово **set** указывает парсеру на выработку с именем **set**, соответственно ключевое слово **draw** указывает ему в качестве следующей выработку с именем **draw**. Имена выработок не должны обязательно соответствовать ключевым словам, просто листинг так лучше читается.

Внутри определения каждой конечной выработки имеется заключенный в фигурные скобки код на языке C. Это может быть любой блок кода, в нашем примере это простые вызовы функций, которые будут определены позже. Имена аргументов функций определяются положением соответствующих им лексем в правой части определения выработки: **\$1** — для первого параметра выработки, **\$2** — для второго и т.д. Обратите внимание, что в нашем примере функциям передаются значения лексем **NAME** и **NUMBER**, при этом соответствующие аргументы получают значения переменных **yytext** и **yyval**.

После того как утилита **bison** скомпилирует сценарий **clang.y** в заголовочный файл **clang.tab.h**, остается только написать программу на языке C, которая генерирует объектный код из исходного языка **clang**. В нашем примере вместо объектного кода программа только выводит описания работы предполагаемого объектного кода. Следующий исходный файл **clmain.c** содержит все функции, необходимые любому парсеру, и те функции, которые вызываются выработками определений нашего языка:

```
/* clmain.c */
#include "clang.tab.h"
#include <stdio.h>

char colorname[30] = "black";
int x = 0;
int y = 0;
```

```
main()
{
    yyparse();
}
int yywrap()
{
    return(1);
}
void yyerror(const char *str)
{
    fprintf(stderr,"Clang: %s\n",str);
}

int setcolor(char *name)
{
    strcpy(colorname,name);
    return(0);
}

/* Сохранение координат x и y положения следующей
изображаемой фигуры. */
int setlocation(int xloc,int yloc)
{
    x = xloc;
    y = yloc;
}

/* Прорисовка круга указанного цвета и размера
в текущем расположении. */
int drawcircle(int radius)
{
    printf("Draw %s circle at (%d,%d) radius=%d\n",
           colorname,x,y,radius);
}

/* Прорисовка прямоугольника указанного цвета,
высоты и ширины в текущем расположении. */
int drawrectangle(int height,int width)
{
    printf("Draw %s rectangle at (%d,%d) h=%d w=%d\n",
           colorname,x,y,height,width);
}
```

Заголовочный файл `clang.tab.h` вырабатывается утилитой `bison` из файла `clang.y`, он также содержит определения некоторых полезных констант, которые могут использоваться в компилирующей программе. Функция `main()` — головная процедура нашего компилятора. В примере она только выполняет вызов `yyparse()`, функции синтаксического разбора. В реальном компиляторе она отвечает за создание кода на промежуточном языке, управление преобразованием промежуточного языка в объектный код, за выполнение оптимизаций, обработку параметров командной строки, определение имен входных и выходных файлов и за прочие выполняемые компиляторами действия.

Функция `yywrap()` вызывается синтаксическим разделителем (парсером) в конце текущего входного файла. Она может использоваться для перехода к считыванию следующего исходного файла. Возвращаемое значение, равное 1, указывает на отсутствие дальнейшего ввода.

Функция `yyerror()` вызывается парсером при возникновении любой ошибки. Функции передается строка описания ошибки. В нашем примере она только выводит на стандартное устройство вывода сообщение об ошибке.

Функция `setcolor()` вызывается парсером каждый раз, когда в исходном файле ключевое слово `set` используется для назначения нового цвета. В зависимости от вырабатываемого объектного кода и возможностей управления графической системой эта функция может так или иначе устанавливать текущий цвет. Или, как в нашем примере, локально сохранять информацию о текущем цвете, чтобы она могла быть использована при прорисовке фигур.

Функция `setlocation()` действует сходно с рассмотренной функцией `setcolor()`, за исключением того, что она определяет положение для прорисовки следующей фигуры. В примере она локально сохраняет координаты в переменных, используемых при прорисовке фигур.

Функции `drawcircle()` и `drawrectangle()` вызываются парсером для выработки кода, выполняющего прорисовку. Генерируемые при этом инструкции могут использовать предварительно установленные значения цвета и координаты положения. В примере эти функции только лишь выводят информацию, которая могла бы использоваться для генерирования действующего объектного кода.

Приведенную далее последовательность команд можно использовать для компиляции программы,читывающей исходный код языка `clang` и вырабатывающей псевдоинструкции прорисовки фигур:

```
$ bison -d clang.y
$ flex clang.lex
$ gcc clmain.c lex.yy.c clang.tab.c -o clang
```

Команда `bison` считывает файл `clang.y` и вырабатывает файл `clang.tab.c`, который содержит код на языке `C`, разделяющий входной поток лексем. Поэтому он должен быть скомпилирован и скомпонован в вырабатываемый компилятор. Опция `-d` указывает, что также должен быть создан заголовочный файл `clang.tab.h`. Этот файл используется в исходном файле `clmain.c` и в файле `clang.lex`, где он предоставляет константы всех типов лексем.

Команда `flex` используется для выработки файла `lex.yy.c`, содержащего исходный код функций `C` для считывания входного потока и организации его в поток лексических элементов.

Команда `gcc` компилирует и компонует три исходных файла на языке `C` в готовую к запуску программу `clang`. Это — компилятор, принимающий исходный код в потоке стандартного ввода, поэтому для компиляции тестовой программы с именем `figures clang` следует применить такую команду:

```
$ cat figures clang | clang
```

Вывод этой команды будет выглядеть следующим образом:

```
Draw blue circle at (100,200) radius=30
Draw red rectangle at (250,200) h=10 w=10
```

Построение дерева синтаксического разбора

В результате процесса синтаксического разделения строится дерево синтаксического разбора (parse tree). Оно представляется в форме последовательности текстовых строк, каждая из которых описывает узел дерева. Каждый узел имеет идентификатор, используемый для разрешения ссылок на этот узел из любых других узлов дерева. Каждый узел также содержит символ, указывающий его тип. Типы узлов и соответствующие им обозначающие символы перечислены в таблице 19.1.

Таблица 19.1. Буквенные символы – коды типов узлов дерева синтаксического разбора

Символ	Описание типа узла
<	Выражение сравнения.
1	Унарное арифметическое выражение. (С одним операндом.)
2	Бинарное арифметическое выражение. (С двумя операндами.)
b	Лексический блок.
c	Константа.
d	Объявление переменной либо ссылка на переменную.
e	Выражение, которое не является сравнением, унарной или бинарной арифметической операцией, и его вычисление не имеет побочных эффектов.
r	Ссылка (reference) на элемент данных в памяти.
s	Выражение, вычисление которого имеет побочные эффекты.
t	Тип данных.
x	Особый узел, не соответствующий никакой категории.

Определения символов индикаторов типов узлов содержатся в файле исходного кода GCC `tree.def`. Для создания узлов дерева синтаксического разбора используется большое количество функций, их исходный код находится в файле `stmt.c`. Функций, предназначенных для построения узлов, настолько много, что создается впечатление, будто любой оператор языка верхнего уровня в каждом варианте применения имеет собственную процедуру-генератор кода RTL. Например, следующая функция генерирует код сравнения операндов `op1` и `op2` и передает управление коду программы, имеющему метку `label`, в случае совпадения их значений:

```
static void do_jump_if_equal(op1,op2,label,unsignedp);
```

В этом примере оба операнда `op1` и `op2` являются узлами выражений дерева синтаксического разбора. `label` является указателем расположения выполнимого кода в памяти. Последний аргумент определяет метод сравнения чисел: с учетом или без учета знака. Функция проверяет аргументы и только после этого определяет гене-

рируемый код. Например, в случае, когда `op1` и `op2` являются константами с равными значениями, вырабатывается простая инструкция передачи управления на адрес метки `label1`. Если на этом этапе форма инструкции определена, то вызывается подпрограмма, вырабатывающая соответствующую инструкцию.

Исходный код подпрограмм нижнего уровня, которые вырабатывают выходной RTL-код, находится в файле `emit-rtl.c`. Возможно, что функция `emit_note()` является наиболее простой из них. Она генерирует инструкции, не выполняющие никаких действий, которые только заполняют свободное место. Код, который вырабатывает действующие инструкции и помещает их в выходной RTL-код, выглядит подобно следующему:

```
note = rtx_alloc(NOTE);
INSN_UID(note) = cur_insn_uid++;
NOTE_SOURCE_FILE(note) = file;
BLOCK_FOR_INSN(note) = NULL;
add_insn(note);
```

В начале этой последовательности функция `rtx_alloc()` создает новый узел дерева RTL-кода. Узел дерева определен на языке C как составной оператор `rtx_def` в файле `rtl.h`. Он содержит набор идентифицирующих флагов, за ними следует массив переменной длины, содержащий операнды. Затем макрос `INSN_UID` вставляет уникальный номер (ID) узла. Макрос `NOTE_SOURCE_FILE` добавляет информацию об исходном файле.

Вызов `add_insn()` добавляет новый сконструированный узел в конец списка связанных выходного кода на языке RTL. Для вставки новой инструкции RTL *перед* уже записанной инструкцией применяется функция `add_insn_before()`. Для ее помещения сразу *после* существующей инструкции — функция `add_insn_after()`.

Остается одна проблема. В RTL-код не внесена информация символьных таблиц. На верхнем уровне присутствие таблиц программных символов в той или иной форме необходимо, они требуются для разрешения ссылок на имена элементов программы. А в RTL-коде все ссылки разрешаются через идентификаторы узлов дерева и только через них. Однако символьные таблицы должны существовать и быть доступными с нижнего уровня компилятора.

Подключение нижнего уровня компилятора к верхнему уровню

Нижний уровень (back end) компилятора не вполне отделен от его верхнего уровня (front end). Некоторые переменные и функции программы могут быть объявлены на верхнем уровне как глобальные. Следовательно, должна быть возможность прямой адресации к ним с нижнего уровня компилятора.

Код реализации языка верхнего уровня следует размещать в отдельном каталоге ниже корневого каталога GCC. Например, код для языка C++ содержится в каталоге `cp`, а для Ada — в каталоге `ada`. В таком каталоге имеется файл с именем `Make-lang.in` — компоновочный файл, включаемый в make-файл родительского каталога GCC. Также включается и файл `Makefile.in`. Он используется при создании компоновочного

сценария (`make`-файла) для языка. Файл `config-lang.in` используется конфигурационным сценарием `configure`.

Для подключения нового языка драйвер `gcc` должен быть модифицирован. Все изменения вносятся автоматически в процессе его сборки (`build process`).

Верхний уровень компилятора языка должен содержать глобальные переменные и функции, адресуемые с нижнего уровня. Это необходимо для обеспечения доступа к узлам синтаксического дерева и таблицам программных символов. Также они нужны и для процедур инициализации и завершения. Таблица 19.2 содержит краткое описание обязательных глобальных переменных, доступных с нижнего уровня GCC. Краткое описание адресуемых с нижнего уровня компилятора глобальных функций находится в таблице 19.3. Многие из этих функций используются и на верхнем уровне, но их имена должны быть объявлены глобально.

Таблица 19.2. Переменные верхнего уровня GCC, адресуемые с его нижнего уровня

Имя переменной	Описание
<code>error_mark_node</code>	Родительский узел дерева, содержащий условия вывода сообщений об ошибках.
<code>integer_type_node</code>	Узел основного целочисленного типа.
<code>char_type_node</code>	Узел основного типа представления буквенных знаков.
<code>void_type_node</code>	Узел основного типа вызова <code>void</code> .
<code>integer_zero_node</code>	Узел дерева, который содержит нулевое значение целого числа.
<code>integer_one_node</code>	Узел дерева, содержащий значение целого числа, равного единице.
<code>tree_current_function_decl</code>	Узел представляет текущую транслируемую функцию.
<code>language_string</code>	Адрес строки буквенных знаков, содержащей название языка.
<code>flag_traditional</code>	Обязательный узел, хотя он используется только в языке C.

Таблица 19.3. Функции верхнего уровня GCC,ываемые с его нижнего уровня

Имя функции	Описание
<code>lang_init()</code>	Выполняет все специфичные к языку инициализации.
<code>lang_finish()</code>	Выполняет все специфичные к языку процедуры завершения.
<code>lang_decode_option()</code>	Вызывается с опциями, обнаруженными в командной строке.
<code>init_lex()</code>	Выполняет все инициализации, необходимые для процесса лексического анализа. Применяется при запуске <code>lex</code> -сканера.
<code>init_parse()</code>	Выполняет все инициализации, необходимые для процесса синтаксического разделения. Применяется при запуске парсера.
<code>finish_parse()</code>	Выполняет все процедуры завершения работы парсера.
<code>type_for_mode()</code>	Возвращает узел дерева, определяющий машинное представление данных.
<code>type_for_size()</code>	Возвращает узел дерева, который содержит целое число, представляющее аппаратную точность (в байтах).
<code>type_for_unsigned()</code>	Возвращает целое число со знаком — значение узла дерева, определяющего формат представления таких чисел.

Имя функции	Описание
<code>signed_type()</code>	Возвращает целое число без знака — значение узла дерева, которое определяет формат представления таких чисел.
<code>signed_or_unsigned_type()</code>	Значение узла дерева, определяющего формат машинного представления числа и его знака.
<code>init_decl_processing()</code>	Инициализирует узлы переменных, которые представлены в таблице 19.2.
<code>global_bindings_p()</code>	Возвращает признак глобального объявления текущего участка кода.
<code>kept_level_p()</code>	Возвращает признак, показывающий на необходимость создания блока данных для текущего уровня дерева.
<code>getdecls()</code>	Возвращает листинг всех объявлений текущего уровня кода (current scope level).
<code>pushdecl()</code>	Помещает объявление в символьную таблицу и возвращает соответствующий ему узел дерева.
<code>pushlevel()</code>	Создает новый уровень кода дерева в символьной таблице.
<code>poplevel()</code>	Убирает из символьной таблицы текущий уровень кода дерева и восстанавливает предыдущее состояние.
<code>insert_block()</code>	Добавляет узел блока в конец списка блоков текущего уровня дерева.
<code>set_block()</code>	Устанавливает узел блока в текущем уровне дерева.
<code>maybe_build_cleanup()</code>	Функция может возвращать узел дерева, представляющий операцию освобождения ресурсов после предыдущих действий. (Например, освобождение объектов.)
<code>truthvalue_conversion()</code>	Возвращает выражение, равносильное указанному, но возвращающее логический результат <code>true</code> или <code>false</code> .
<code>mark_addressable()</code>	Отмечает выражение как указывающее адрес памяти.
<code>copy_lang_decl()</code>	Повторяет указанное объявление узла.
<code>incomplete_type_error()</code>	Выдает на стандартный выход сообщение об ошибке использования неполного типа.
<code>yyerror()</code>	Выдает сообщение об ошибке процесса синтаксического разделения (parse error message).
<code>print_lang_decl()</code>	Выводит объявление узла дерева в указанный файл.
<code>print_lang_type()</code>	Выводит информацию о типе узла в указанный файл.
<code>print_lang_identifier()</code>	Выводит информацию идентификатора узла дерева в указанный файл.
<code>set_yydebug()</code>	Включает или отключает отладку синтаксического разделения.



Глава 20

Язык регистрового переноса

Язык *регистрового переноса* (Register Transfer Language, *RTL*) — ключевой элемент процесса компиляции. Назначение верхнего уровня компилятора заключается в генерации кода на языке *RTL*, а нижнего уровня — в трансляции *RTL*-кода на язык ассемблера. Большая часть оптимизации программы выполняется именно в то время, когда она находится в формате *RTL*. Настоящая глава посвящена описанию языка *RTL*.

Инструкции языка *RTL*

Одиночный оператор языка *RTL* называется *инструкцией* (*insn*). Инструкции в программе объединены в двухсвязный список. Некоторые из них являются действительными инструкциями, в то время как другие содержат такие данные, как, например, таблицы ветвлений, используемые операторами ветвления. Кроме того, существуют инструкции, представляющие собой объявления и играющие роль меток для операторов ветвления. Каждой инструкции соответствует ее уникальный идентификатор (ID), с помощью которого одна инструкция может ссылаться на любую другую.

Шесть базовых кодов выражений

Существует много типов инструкций. Каждая инструкция имеет определенный код выражения, указывающий на ее тип. Код языка *RTL*, представляющий логическую последовательность выполнения программы, состоит всего из шести базовых типов, каждый из которых может содержать ссылки на другие типы. Например, код выражения *insn* служит указателем выполняемого оператора и в качестве операндов содержит другие инструкции. Так следующая инструкция считывает значение переменной *val* и записывает его в регистр. Для выполнения такой операции язык *RTL* содержит код выражения *insn* и инструкцию с кодом *set*, которая, в свою очередь,

использует инструкцию с кодами `reg` и `mem`. Инструкция с кодом `mem` включает инструкцию `symbol_ref`.

```
(insn 12 10 14 (nil) (set (reg:SI 61)
  (mem/f:SI (symbol_ref:SI("val")) [0 a+0 S4 A32])) -1 (nil)
  (nil))
```

Код выражения `insn` — это один из шести базовых типов кодов. Любая программа в RTL-коде состоит из комбинации кодов выражений, приведенных в таблице 20.1.

Таблица 20.1. Шесть базовых типов кодов выражений

Код выражения Описание

<code>insn</code>	Этот код выражения используется для инструкций, которые не выполняют перехода и не вызывают функций. Инструкции такого типа загружают данные в регистры, выполняют арифметические операции, сравнивают значения и т.д.
<code>jump_insn</code>	Применяется для инструкций, которые будут (или могут) выполнять переход, т.е. инструкция, как правило, содержит одну или несколько инструкций <code>label_ref</code> . Кроме того, данный код выражения используется для возврата из текущей выполняемой функции. Для простых условных и безусловных переходов указывается ссылка на <code>code_label</code> — метку для перехода. Для более сложных переходов при определении возможных меток перехода иногда приходится просмотреть все содержащиеся инструкции.
<code>call_insn</code>	Используется для инструкций, которые будут (или могут) вызывать функции. Такие инструкции должны обрабатываться отдельно, поскольку они могут непредсказуемым образом изменять содержимое регистров и ячеек памяти. Инструкции данного типа, как правило, содержат инструкции <code>clobber</code> и <code>mem</code> , которые указывают, содержимое каких регистров и ячеек памяти изменяется. Таким образом, в инструкции типа <code>call_insn</code> будет содержаться либо инструкция <code>mem</code> , указывающая на блок памяти, через который передаются параметры, либо инструкции <code>clobber</code> и <code>cve</code> , указывающие рабочие регистры и регистры с аргументами функции.
<code>code_label</code>	Применяется для обозначения метки, которая может использоваться в качестве цели перехода. Инструкции данного типа содержат инструкцию <code>code_label_limbeg</code> , указывающую уникальный идентификатор. Идентификатор метки уникален для всего компилируемого блока, а не только текущей функции. Инструкция <code>code_label</code> , как правило, используется в качестве ссылки в инструкции <code>label_ref</code> . Во время и после оптимизации ведется счетчик количества использования каждой метки в качестве цели перехода. Любая метка <code>code_label</code> может принадлежать к одному из следующих четырех видов: <code>NORMAL</code> — единственный вид меток, которые не могут быть альтернативной точкой входа в функцию; <code>STATIC_ENTRY</code> — метка точки входа в функцию, видимая только внутри блока компиляции; <code>GLOBAL_ENTRY</code> — является меткой точки входа в функцию и видима (с помощью компоновщика) во всех блоках компиляции; <code>WEAK_ENTRY</code> — метка точки входа в функцию, может быть переопределена другим символом с таким же именем.
<code>barrier</code>	Этот код выражения помещается внутри последовательности инструкций для указания места, которое не может быть достигнуто в процессе выполнения программы. Инструкция <code>barrier</code> вставляется после безусловного перехода и вызовов невозвращающих функций.

Код выражения Описание

note	Этот код выражения, который используется для указания определенной отладочной и декларативной информации. Любая инструкция типа note содержит номер и поле строки буквенных символов (character string). Номер, как правило, представляет собой номер строки исходного файла, указанного в строке символов. Инструкция note управляет данными о номерах строк, используемыми при отладке. Если число не является номером строки, то эта инструкция является указателем одного из следующих типов: NOTE_INSN_DELETED — указывает положение инструкции, которая была удалена; NOTE_INSN_DELETED_LABEL — заменяет удаленную инструкцию code_label , которая была удалена вследствие того, что не использовалась в качестве цели для перехода; NOTE_INSN_BLOCK_BEG — отмечает начало блока уровня кода; NOTE_INSN_BLOCK_END — отмечает конец блока уровня кода; NOTE_INSN_EH_REGION_BEG — отмечает начало блока уровня кода, используемого для обработки исключений; NOTE_INSN_EH_REGION_END — отмечает конец уровня, используемого для обработки исключений; NOTE_INSN_LOOP_BEG — отмечает начало цикла while или for ; NOTE_INSN_LOOP_END — отмечает конец цикла while или for ; NOTE_INSN_LOOP_CONT — отмечает в цикле место, на которое будет переходить оператор continue ; NOTE_INSN_LOOP_VTOP — отмечает в цикле место начала проверки выхода из цикла; NOTE_INSN_FUNCTION_END — отмечает место вблизи конца функции перед меткой, на которую переходит оператор return ; NOTE_INSN_SETJMP — отмечает код, непосредственно следующий за вызовом функции типа setjmp() .
-------------	--

Типы и содержимое инструкций

Каждый тип инструкций уникален и служит определенной цели. Поэтому каждый тип содержит набор данных, соответствующий его назначению. Очень часто данные типа могут быть представлены в виде другой инструкции. Тем не менее, такую цепочку связанных RTL-инструкций можно всегда проследить до базовых типов данных. Данные, содержащиеся в операторах языка RTL, могут принадлежать к одному из пяти типов инструкций, описанных в таблице 20.2.

Таблица 20.2. Базовые типы данных инструкций языка RTL

Тип инструкции	Описание
Выражение	Выполняемое выражение. Выражение языка RTL обозначается RTX.
Целое число	Целочисленный тип данных, соответствующий типу данных int языка C.
Целое число расширенной точности	Целочисленный тип данных, определяемый переменной HOST_WIDE_INT , которая, как правило, в качестве данного типа данных устанавливает 64-битное значение.
Строка	Последовательность символов, заканчивающаяся нулем, как и в языке C. Строки в большинстве случаев используются в качестве символьных ссылок, но иногда могут применяться и для представления описательных данных о машине, как будет указано в главе 21. Во внутреннем представлении строка нулевой длины соответствует нулевому указателю.

Тип инструкции Описание

Вектор	Произвольное количество указателей на выражения. Количество элементов вектора явным образом указывается в инструкции. Во внутреннем представлении векторы нулевой длины соответствуют нулевым указателям.
--------	---

Формат и содержимое выражений языка RTL может изменяться в широких пределах, но всегда существуют три поля: идентификатор, адрес предыдущей инструкции и адрес следующей инструкции. Эти три значения могут быть определены для любой инструкции с помощью следующих трех макросов:

```
INSN_UID(insn)
PREV_INSN(insn)
NEXT_INSN(insn)
```

Первую инструкцию можно извлечь с помощью функции `get_insns()`, а последнюю — с помощью функции `get_last_insn()`.

Большая часть кода компилятора *GCC*, работающего с инструкциями, написана для выполнения операций с выражениями. Выражение языка RTL обозначается RTX. Такие инструкции-выражения представляют собой операторы, содержащие выполнимый код программы. Внутри кода Компилятора они сортируются в структуру, доступ к которой осуществляется с помощью указателя с именем указателя типа `rtx`.

Каждое выражение имеет свой *код выражения* (*expression code*) (или RTX-код), указывающий тип выражения. Коды выражений перечислены в файле `rtl.def` в виде набора имен констант типа перечислений. Коды выражений не зависят от платформы машины, т.е. язык RTL является машинно-независимым. RTX-код можно установить в структуре `rtx`, а затем считать его с помощью следующих двух макросов, определенных в файле `rtl.h`:

```
PUT_CODE(rtx,code);
int code = GET_CODE(rtx);
```

Макрос `DEF_RTL_EXPR` используется для определения всех выражений языка RTL, содержащихся в файле `rtl.def`. Этот макрос, как показано в следующем примере, требует указания четырех аргументов:

```
DEF_RTL_EXPR(COND_EXEC, "cond_exec", "ee", 'x')
```

Первый передаваемый макросу аргумент представляет собой имя выражения языка RTL, представленное буквами верхнего регистра. Он используется в исходном коде на языке C в качестве параметра типа `enum` как уникальный идентификатор выражения. Второй аргумент — имя выражения языка RTL, состоящее из букв нижнего регистра в формате ASCII. Это имя выводится в диагностических сообщениях. Третий аргумент содержит список типов данных операндов, причем каждому типу соответствует один буквенный знак. Описание типов приведено в таблице 20.3. Четвертый аргумент — состоящий из одного буквенного символа указатель класса выражения RTL. Перечень классов приведен в таблице 20.4.

Таблица 20.3. Коды, используемые для указания типов operandов выражений языка RTL

Код операнда	Описание
*	Тип не указан. Попытка обработки этого типа сгенерирует предупреждение.
0	Неиспользуемое поле.
b	Указатель на заголовок битовой карты (bitmap).
v	Определение базового блока инструкций (с одним входом и одним выходом).
e	Указатель на выражение языка RTL.
E	Указатель на массив выражений языка RTL.
i	Целое число.
n	Целое число, указывающее следующее: 1 — инструкция удалена; 2 — начало блока; 3 — конец блока; 4 — начало цикла; 5 — конец цикла; 6 — продолжение цикла; 7 — инструкция в начале цикла; 8 — условие завершения цикла; 9 — конец функции; 10 — конец пролога функции; 11 — начало эпилога функции; 12 — удаленная метка; 13 — начало функции (точка входа); 14 — начало области обработки исключений; 15 — конец области обработки исключений; 16 — повторяющийся номер строки; 17 — базовый блок; 18 — ожидаемое значение; 19 — прогноз.
s	Строка символов (character string).
S	Необязательная строка символов.
u	Указатель на другую инструкцию.
t	Указатель на дерево.
T	Код, который должен подвергаться ассемблированию или компиляции и запускаться на выполнение.
v	Необязательный указатель на массив выражений языка RTL.

Таблица 20.4. Коды классов выражений языка RTL

Класс	Описание
<	Оператор сравнения, например, оператор "меньше или равно".
1	Унарный арифметический оператор, например, изменение знака числа или оператор дополнения до единицы.
2	Некоммутативная двоичная операция, например, вычитание или деление.
3	Код выражения языка RTL для не битовой операции с тремя аргументами, например, оператора if/then/else.

Класс	Описание
a	Код выражения языка RTL для режимов автоинкрементной адресации.
b	Битовая операция заполнения нулями или установки битов знака.
c	Коммутативная двоичная операция, например, сложение или умножение.
g	Код выражения языка RTL для группировки инструкций.
i	Машинная инструкция, например, передача управления или вызов.
m	Код выражения языка RTL для согласования инструкций.
o	Код выражения языка RTL, соответствующий физическому объекту, например, ячейке памяти или регистру.
x	Код выражения языка RTL, которое не попадает ни в один из выше перечисленных классов.

Количество и типы operandов изменяются в зависимости от кода инструкции. В таблице 20.3 приведены описания кодов, используемых для указания типа данных operandов. Эти коды не требуют строгого соблюдения. Так, например, несмотря на то, что код **t** используется для указания источника выполняемого кода, а тип **s** — для указания простой строки, некоторые определения языка RTL в качестве данных типа **s** содержат исходный код на языке C. Каждый код RTL обрабатывается отдельно, поэтому operandы могут содержать практически любые данные — коды типов используются только лишь для вывода значений operandов при отладке.

Следующий перечень содержит описания кодов языка RTL. Для каждого типа указано его имя, за которым следует его класс, и затем строка символов с типом и количеством operandов. Для всех типов дается описание и список используемых operandов, включающий их тип и назначение. Тип данных и порядок следования operandов должен соответствовать строке, взятой в кавычки. Нумерация operandов начинается с нуля. В коде компилятора GCC форма кодов языка RTL, задаваемая буквами верхнего регистра, используется для определения перечисляемого типа, который применяется в качестве числового идентификатора для RTL. Например, выражение **eq_attr** в качестве уникального идентификатора использует константу **EQ_ATTR**. Как указано в описаниях, некоторые коды RTL служат для четко определенной цели. В частности, некоторые из них применяются для создания списков выражений, в то время как другие используются лишь для удобства и никогда не указываются в инструкциях. Такие коды используются только как элементы описаний машин.

Список инструкций RTL

abs '1' "e"

Если результат выражения — отрицательное число, то он преобразуется к положительному значению.

absence_set 'x' "ss"

Эта инструкция используется только в описаниях машин для указания списка функциональных блоков процессора, которые не могут резервироваться, если оп-

ределенные функциональные блоки также не резервированы. Например, в процессоре VLIW slot0 не может резервироваться после slot1 и slot2. Также см. описание для `presence_set`. Операнд этого кода 0 представляет собой разделенный запятыми список функциональных блоков, которые не могут резервироваться, если зарезервирован хотя бы один функциональный блок из операнда 1. Операнд 1 представляет собой разделенный запятыми список функциональных блоков.

addr_diff_vec 'x' "eEee0"

Этот код содержит вектор разниц адресов между базовой операцией и целевой операцией, этот вектор будет использоваться при вычислении расстояний. Операнды 2, 3 и 4 имеют смысл только в случае, когда для компилятора установлен режим `CASE_VECTOR_SHORTEN_MODE`. Операнд 0 — это базовая операция, операнд 1 — разница адресов, соответствующая расстоянию каждого операнда до базовой операции. Операнд 2 представляет собой метку минимального адреса, а операнд 3 — метку максимального адреса. Операнд 4 содержит набор флагов, используемых для определения правил сокращения флагов. Флаг `"min_align"` устанавливает использование минимального выравнивания для ветвей. Флаг `"base_after_vec"` указывает, что базовый адрес задан после `base_after_vec`. Флаг `"min_after_vec"` указывает, что целевая метка минимального адреса задана после `addr_diff_vec`. Флаг `"max_after_vec"` указывает, что целевая метка максимального адреса задана после `addr_diff_vec`. Флаг `"offset_aligned"` устанавливает, что смещения должны считаться значениями без знака, а флаг `"scale"` говорит о необходимости выбора смещений в зависимости от режима.

addr_vec 'x' "E"

Вектор адресов. Каждый адрес включается в `code_label` в качестве метки `label_ref`.

address 'm' "e"

Ссылка на адрес аргумента. Операнд 0 представляет собой выражение, описывающее адрес.

addressof 'o' "eit"

Ссылка на адрес регистра, который был удален в компиляторе функцией `purse_addressof()` при удалении неиспользуемых адресов регистров. Операнд 0 — это регистр. Операнд 1 представляет собой исходный номер псевдорегистра, для которого была сгенерирована инструкция. Операнд 2 является объявлением хранящегося в регистре элемента для его использования функцией `put_reg_in_stack`.

and 'c' "ee"

Вычисляются значения операндов, а в качестве результата возвращается результат выполнения операции поразрядного сложения (AND) значений обоих операндов.

ashift '2' "ee"

Поразрядный арифметический сдвиг влево. Операнд 0 представляет собой выражение, дающее сдвигаемое значение, а операнд 1 — выражение, определяющее значение количества разрядов, на которое производится сдвиг.

ashiftrt '2' "ee"

Поразрядный арифметический сдвиг вправо (учитывающий бит знака числа). Операнд 0 представляет собой выражение, дающее сдвигаемое значение, а операнд 1 — выражение, определяющее значение количества разрядов, на которое производится сдвиг.

asm_input 'x' "s"

Строка, передаваемая ассемблеру в качестве инструкции. Она может использоваться в других инструкциях в роли части блока кода, а также для вставки комментариев в код ассемблера.

asm_operands 'x' "ssiEEsi"

Инструкция языка ассемблера со своими operandами. Операнд 0 представляет собой шаблон, описывающий инструкцию. Операнд 1 является ограничением для результата. Операнд 2 содержит значение-идентификатор, которое позволяет отличить данный оператор языка ассемблера от других операторов. Операнд 3 представляет собой набор значений, которые будут использоваться в качестве входных operandов. Операнд 4 содержит коллекцию режимов и ограничений для входных operandов. Каждый элемент массива принадлежит к типу **asm_input** и содержит строку, указывающую режим входного операнда. Операнд 5 используется для указания имени файла исходного кода, а операнд 6 — строки исходного кода.

attr 'x' "s"

Этот код используется только в описаниях машин для определения атрибутов инструкций. Это — маркер, который можно вставить для указания имени атрибута. Операнд 0 — это имя атрибута.

attr_flag 'x' "s"

Эта инструкция используется только в описаниях машин для установки атрибутов. Если значение условного выражения равно **true**, этот параметр указывает вероятность перехода на ветвь, а выполняемая инструкция задается флагом. Для флага допустимы следующие значения: **"forward"**, **"backward"**, **"very_likely"**, **"likely"**, **"veryunlikely"** и **"unlikely"**. Операнд 0 содержит значение флага.

automata_option 'x' "s"

Эта инструкция используется только в описаниях машин в качестве опции для генерации автоматов. Значение **"no-minimization"** для операнда 0 означает, что автомат не может минимизироваться. Это имеет смысл только в случае, когда в состоянии автоматизации будет запрашиваться резервирование блоков процессора. Опция **"time"** представляет собой запрос на вывод дополнительной статистики

расхода времени при генерации автомата. Опция "**v**" является запросом генерации файла с суффиксом **.dfa**, содержащего верификационную и отладочную информацию. Опция "**w**" приводит к тому, что при возникновении некритических ошибок генерируются сообщения об ошибках вместо предупреждений. Опция "**ndfa**" приводит к формированию недетерминированного конечного автомата.

barrier 'x' "iuu"

Маркер, который указывает, что через него не будет проходить поток выполнения программы. Операнд 0 содержит уникальный идентификатор для выражения языка RTL. Оператор 1 представляет собой указатель на предыдущую инструкцию в цепочке инструкций, а оператор 2 — на следующую инструкцию.

call 'x' "ee"

Вызывает подпрограмму. Операнд 0 представляет собой адрес вызываемой подпрограммы. Операнд 1 содержит количество передаваемых подпрограмме аргументов.

call_insn 'i' "iuuBteieee"

Инструкция, которая может вызывать подпрограмму, но не может изменять адрес, на который выполняется возврат из подпрограммы. Операнд 0 — это уникальный идентификатор выражения языка RTL. Операнд 1 представляет собой указатель на предыдущую инструкцию в цепочке, а операнд 2 — указатель на следующую инструкцию в цепочке. Операнд 3 содержит базовый блок инструкций. Операнд 4 является указателем на узел дерева. Операнд 9 — это инструкция **call_insn_function_usage**, выполняющая вызов функции.

call_placeholder 'x' "uiuu"

Заполнитель, который должен заменяться инструкцией **call_insn**, или кодом вызова подпрограммы этого же уровня, или кодом рекурсивного вызова. Операнд 0 — это уникальный идентификатор выражения языка RTL. Операнд 1 представляет собой указатель на предыдущую инструкцию в цепочке, а операнд 2 — указатель на следующую инструкцию в цепочке. Операнд 3 содержит инструкцию **code_label** для метки, используемой при рекурсивных вызовах. Если рекурсия не применяется, этот операнд содержит нулевой указатель.

clobber 'x' "e"

Указатель того, что что-то используется таким образом, который не требует объяснения. Например, вызов подпрограммы будет использовать регистр и перезаписывать ранее содержащееся в нем значение. Также см. **use**. Операнд 0 представляет собой выражение, указывающее на используемый элемент.

cc0 'o' ""

Представляет состояние регистра кода условия. Используемая в инструкции логика соответствует ситуации, когда регистр кода условия содержит значение, которое можно сравнить с нулем, но фактически оно равно **true** или **false** и является результатом предыдущей операции сравнения.

code_label 'x' "iuuB00iss"

Метка, за которой указаны инструкции. Операнд 0 — это уникальный идентификатор выражения языка RTL. Операнд 1 представляет собой указатель на предыдущую инструкцию в цепочке, а операнд 2 — указатель на следующую инструкцию в цепочке. Операнд 3 содержит базовый блок инструкций, следующий после метки. Операнд 4 — это счетчик переходов на данную метку. Операнд 5 содержит указатель на цепочку ссылок на метку для данной метки. Операнд 6 — это уникальный идентификатор. Операнд 7 содержит имя, которое пользователь присвоил этой метке, если такое имя существует. Операнд 8 представляет собой альтернативное имя метки для внутреннего использования.

compare '2' "ee"

Два операнда этой инструкции вычисляются и сравниваются между собой. Если значения операндов равны, инструкция в результате дает ноль. В противном случае результатом выполнения инструкции будет ненулевое значение.

concat 'o' "ee"

Производится объединение (конкатенация) двух выражений таким образом, что в результате формируется значение с количеством бит, равным сумме бит в исходных выражениях. Эта инструкция используется для комплексных чисел и, как правило, присутствует в коде языка RTL, но не в окончательной цепочке инструкций.

cond 'x' "Ee"

Обычный условный оператор. Операнд 0 представляет собой вектор пар выражений. Сначала вычисляются первые элементы каждой пары. Во втором элементе пары содержат условное выражение, результат вычисления которого равен нулю при выполнении условия и ненулевому значению в противном случае. Если ни одна из пар операнда 0 не дает `true`, то в качестве условного выражения используется операнд 1.

cond_exec 'x' "ee"

Условное выражение и блок кода, который выполняется, если условие соблюдается. Условное выражение не дает побочных эффектов. Операнд 0 содержит условное выражение, а операнд 1 — выполняемую инструкцию.

const 'o' "e"

Выражение, которое дает константу. Это приводит к тому, что выражение воспринимается компилятором не как выражение, транслируемое в окончательный код, а как константа.

const_double 'o' "ww"

Числовая константа с плавающей точкой. Операнды представляют собой цепочки значений, составляющих полное значение числа двойной точности. Синтаксис инструкции представлен в виде "`ww`", там не менее, он может варьироваться от "`ww`" до "`wwwww`" в зависимости от формата чисел с плавающей точкой на целевой платформе.

const_int 'o' "w"

Числовая целочисленная константа.

const_string 'o' "s"

Числовая строковая константа. В настоящее время она используется только в качестве атрибутов.

const_vector 'x' "E"

Константа, представляющая собой вектор (массив).

constant_p_rtx 'x' "e"

Выражение `__builtin_constant_p`. Это выражение создается при генерации инструкций языка RTL только при использовании оптимизации и удаляется при первом проходе CSE.

define_asm_attributes 'x' "V"

Устанавливает атрибуты для инструкций языка ассемблера. Операнд 0 представляет собой вектор, содержащий список атрибутов.

define_attr 'x' "sse"

Эта инструкция используется только в описаниях машин и используется для задания атрибутов инструкций. Операнд 0 содержит имя устанавливаемого атрибута. Операнд 1 представляет собой список возможных значений атрибута, разделенных запятыми. Операнд 2 — это выражение, которое будет использоваться для установки значения атрибута по умолчанию.

define_automation 'x' "s"

Эта инструкция применяется только в описаниях машин для задания имени автомата, используемого для конвейерного распознавания опасных ситуаций. Имя автомата используется в инструкциях `define_cpu_unit` и `define_query_cpu_unit`. Операнд 0 содержит список имен, разделенных запятыми.

define_bypass 'x' "issS"

Эта инструкция используется только в описаниях машин для установки задержки между различными наборами инструкций. Операнд 0 содержит значение задержки. Операнд 1 представляет собой список имен инструкций, разделенных запятыми, на которых начинается действие задержки. Операнд 2 — это список имен инструкций, разделенных запятыми, на которых заканчивается действие задержки. Операнд 3 содержит имя необязательной функции, которая в качестве аргументов принимает две инструкции и возвращает значение обхода. Если обход игнорируется, возвращается нулевое значение.

define_cond_exec 'x' "Ess"

Определение метаоперации условного выполнения, предназначенней для генерирования новых экземпляров `define_insn`. Операнд 0 содержит выражение, ко-

торое будет использоваться для сравнения. Операнд 1 — это условное выражение языка *C*, которое для выполнения сравнения должно давать ненулевое значение. Операнд 2 представляет собой блок кода на языке *C* или ассемблере, передаваемый в виде кода на языке ассемблера.

define_cpu_unit 'x' "sS"

Эта инструкция используется только в описаниях машин для определения имен функциональных блоков процессора. Операнд 0 представляет собой список имен функциональных блоков, разделенных запятыми. Операнд 1 — это имя автоматизации, соответствующее описанию кода `define_automation`.

define_delay 'x' "eE"

Устанавливает необходимость задержек. Операнд 0 представляет собой условное выражение, значение которого равно `true`, если инструкция требует указанное количество задержек. Операнд 1 содержит вектор (длина которого равна уточненному количеству задержек), приводящий в соответствие каждой задержке три условия. Значение `true` первого условия указывает, что инструкция должна занять место задержки. Второе условие равно `true` для тех инструкций, которые могут быть удалены, если выполнение кода пойдет по данной ветви. Третье условие равно `true` для тех инструкций, которые могут быть удалены, если выполнение кода пойдет не по данной ветви.

define_expand 'x' "sEss"

Устанавливает порядок генерирования блока инструкций для имени стандартной инструкции. Операнд 0 содержит имя стандартной инструкции. Операнд 2 — выражение языка *C*, которое для выполнения этой операции должно давать ненулевое значение. Операнд 3 представляет собой код на языке *C*, выполняемый перед генерированием инструкций. Это, например, может быть последовательность выражений языка RTL `sequence`, которая будет использоваться при генерации кода.

define_function_unit 'x' "siiieiIV"

Набор инструкций, требующих использования определенного функционального блока. То есть каждая из этих инструкций дает результат после некоторой задержки. Кроме того, на количество одновременно выполняемых инструкций данного типа может накладываться ограничение, что может быть вызвано наличием аналогичного ограничения для функционального блока процессора. Для одного процессора может быть объявлено несколько инструкций `define_function`, тем не менее, первые operandы во всех инструкциях для одного и того же функционального блока должны быть одинаковыми. Операнд 0 содержит имя функционального блока процессора. Операнд 1 представляет собой количество идентичных функциональных блоков процессора. Операнд 2 — максимальное количество одновременно работающих функциональных блоков процессора. Число 0 указывает на то, что ограничение на количество одновременно работающих блоков не установлено. Число 1 говорит о том, что в любой момент времени только одна инструкция может пользоваться

функциональным блоком. Операнд 3 представляет собой условное выражение, включающее атрибут функции. Если функция применима к данной инструкции, условное выражение должно давать ненулевое значение. Операнд 4 — это константное значение задержки, по истечении которой станет доступным результат инструкции, использующей функцию. Операнд 5 содержит константу — длительность задержки, по истечении которой другая инструкция может использовать этот же функциональный блок. Операнд 6, если он указан, представляет собой список выражений. Если одно из выражений дает ненулевое значение, значит, функциональный блок в настоящее время работает и необходимо вставить задержку с соответствующей длительностью. Если результаты вычисления всех выражений равны нулю, функциональный блок свободен и доступен для немедленного использования (при соблюдении условия, содержащегося в операнде 2).

define_insn 'x' "sEsTV"

Эта инструкция используется только в описаниях машин и представляет собой определение одного вида инструкции. В качестве операнда 0 указывается имя инструкции. Если имя является нулевой строкой, инструкция указана в описании машины только для ее использования в сравнениях и не будет использоваться в выражениях языка RTL. Операнд 1 содержит шаблон выражения. Операнд 2 — это выражение языка C, указывающее дополнительные условия для распознавания данного шаблона. Если операнд 2 является нулевой строкой, дополнительные условия отсутствуют. Операнд 3 представляет собой код на языке ассемблера, определяющий, какое действие должно быть выполнено, если сравнение будет успешным. Если выражение начинается с символа звездочки, используется код на языке C, а не ассемблера. Операнд 4 представляет собой необязательный вектор атрибутов для данной инструкции (см. `set_attr` и `set_attr_alternative`).

define_insn_and_split 'x' "sEsTsESV"

Определение инструкции и ее разделения. Эта инструкция получена с помощью конкатенации инструкций `define_insn` и `define_split`, использующих один и тот же шаблон. Операнд 0 содержит имя инструкции. Если имя представлено нулевой строкой, инструкция указывается в описаниях машин только для сравнения и не будет применяться в выражениях языка RTL. Операнд 1 представляет собой вектор выражений, используемых для сравнения. В качестве операнда 2 указывается исходный код на языке Си, который применяется для задания дополнительного выражения при сравнении. Этот операнд может содержать нулевую строку. Операнд 3 — это исходный код действия, выполняемого в случае, когда совпадение найдено. Операнд 4 содержит код выражения на языке C, результат вычисления которого должен давать ненулевое значение. Операнд 5 представляет собой вектор выражений, помещаемых в `sequence`. Операнд 6 содержит код на языке C, который будет выполняться перед генерированием инструкций. Это, например, может быть код создания выражения `sequence` языка RTL, используемого при генерации кода. Операнд 7 содержит вектор атрибутов для данной инструкции.

define_insn_reservation 'x' "seis"

Эта инструкция указывается только в описаниях машин и применяется для описания резервирования функциональных блоков процессора. Операнд 0 представляет собой строку, используемую в качестве сообщения при отладке и трассировке. Оператор 3 содержит регулярное выражение, используемое для выбора инструкций. Регулярное выражение основано на следующем синтаксисе: вертикальная черта ("|") используется в регулярном выражении в качестве оператора ИЛИ, а знак плюс ("+") — в качестве оператора И. Символ звездочки ("*") указывает повторение элемента заданное количество раз. Имя `cpu_function_unit_name` содержит имя функционального блока процессора. Имя `reservation_name` соответствует имени, определенному с помощью инструкции `define_reservation`. Синтаксис регулярного выражения выглядит следующим образом:

```

regexp = regexp "," oneof
| oneof

oneof = allof "+" repeat
| repeat

repeat = element "*" number
| element

element = cpu_function_unit_name
| reservation_name
| result_name
| "nothing"
| "(" regexp ")"

```

define_peephole 'x' "sEsTV"

Определение локальной оптимизации (peephole optimization). Операнд 0 содержит имя процедуры оптимизации. Операнд 1 представляет собой вектор, к которому может применяться данная оптимизация. Операнд 3 содержит выражение на языке C, которое должно давать ненулевое значение. Операнд 4 задает список необязательных атрибутов инструкции (см. `set_attr` и `set_attr_alternative`).

define_peephole2 'x' "EsES"

Определение локальной оптимизации языка RTL. Операнд 0 содержит вектор искомых инструкций. Операнд 1 содержит выражение на языке C, которое должно давать ненулевое значение. Операнд 2 — это вектор инструкций, помещаемых в `sequence`. Операнд 3 содержит код на языке C, который будет выполняться перед генерированием инструкций. Это, например, может быть код создания выражения `sequence` языка RTL, используемого при генерации кода.

define_query_cpu_unit 'x' "sS"

Эта инструкция указывается только в описаниях машин и применяется для описания функциональных блоков процессора, определенных инструкцией `define_cpu_unit`. Операнд 0 представляет собой список имен функциональных

блоков процессора, разделенных запятыми. Операнд содержит имя автоматизации, описанной в `define_automation`.

`define_reservation 'x' "ss"`

Эта инструкция используется только в описаниях машин для указания коллекции функциональных блоков процессора, которые зарезервированы как группа. Операнд 0 содержит имя, присвоенное набору функциональных блоков. Операнд 1 представляет собой список функциональных блоков, которым будут присвоено указанное имя.

`define_split 'x' "EsES"`

Определение операции разделения. Операнд 0 содержит вектор инструкций для сравнения. Операнд 1 — это выражение на языке C, которое должно давать ненулевое значение. Операнд 2 представляет собой вектор инструкций, помещаемых в `sequence`. Операнд 3 содержит код на языке C, который будет выполняться перед генерируением инструкций. Это, например, может быть код создания выражения `sequence` языка RTL, используемого при генерации кода.

`div '2' "ee"`

Вычисляются значения обоих выражений, и значение операнда 1 делится на значение операнда 0.

`eq '<' "ee"`

Генерируемый код вычисляет значения выражений и производит сравнение значений с учетом знака. Инструкция дает результат `true`, если первое значение равно второму.

`eq_attr 'x' "ss"`

Эта инструкция указывается только в описаниях машин и используется для назначения атрибутов. Для определения того, применим ли атрибут, используется имя атрибута результата сравнения. Операнд 0 содержит имя атрибута, а операнд 1 — значения для сравнения.

`exclusion_set 'x' "ss"`

Эта инструкция указывается только в описаниях машин и применяется для задания функциональных блоков процессора, которые не могут работать одновременно с другими функциональными блоками. Функциональные блоки, содержащиеся в первом операнде, не могут работать одновременно с блоками, указанными во втором операнде. Примером может служить невозможность одновременного выполнения операций с числами с плавающей десятичной точкой нормальной и двойной точности. Операнд 0 представляет собой список функциональных блоков, разделенных запятыми, которые не могут работать одновременно с функциональными блоками, приведенными в операнде 1.

expr_list 'x' "ee"

Связанный список выражений.

ffs '1' "e"

Вычисляет значение выражения и считает количество конечных нулевых битов в значении выражения.

fix '1' "e"

Преобразовывает значение выражения с плавающей десятичной точкой в значение с фиксированной десятичной точкой. Также см. `unsigned_fix`.

float '1' "e"

Преобразовывает значение выражения с фиксированной десятичной точкой в значение с плавающей десятичной точкой. Также см. `unsigned_float`.

float_extend '1' "e"

Генерируется код для вычисления значения выражения с плавающей десятичной точкой, и при необходимости расширяет его для сохранения в требуемом расширенном формате.

float_truncate '1' "e"

Генерируется код для вычисления значения выражения с плавающей десятичной точкой, и при необходимости усекает его для сохранения в требуемом усеченном формате.

ge '<' "ee"

Генерируемый код вычисляет значения выражений и производит сравнение значений с учетом знака. Инструкция дает результат `true`, если первое значение больше или равно второму.

geu '<' "ee"

Генерируемый код вычисляет значения выражений и производит сравнение значений без учета знака. Инструкция дает результат `true`, если первое значение больше или равно второму.

gt '<' "ee"

Генерируемый код вычисляет значения выражений и производит сравнение значений с учетом знака. Инструкция дает результат `true`, если первое значение больше второго.

gtu '<' "ee"

Генерируемый код вычисляет значения выражений и производит сравнение значений без учета знака. Инструкция дает результат `true`, если первое значение больше второго.

high 'o' "e"

Это значение старших битов в константном выражении на RISC-компьютере.

if_then_else '3' "eee"

Представление инструкции условного перехода. Операнд 0 представляет собой условное выражение. Операнд 1 — выражение, представляющее цель для перехода в случае, если значение условного выражения равно **false**.

include 'x' "s"

Операнд представляет имя включаемого файла.

insn 'i' "iuuBteiee"

Это инструкция, которая не может содержать ветвления. Операнд 0 представляет собой уникальный идентификатор выражения языка RTL. Операнд 1 — это указатель на предыдущую, а операнд 2 — на следующую инструкцию в цепочке. Операнд 3 содержит базовый блок инструкций, которые не могут ветвиться. Операнд 4 представляет собой указатель на узел дерева.

insn_list 'x' "ue"

Связный список инструкций.

ior 'c' "ee"

Вычисляются значения обоих операндов, и к полученным значениям применяется операция поразрядного включающего ИЛИ.

jump_insn 'i' "iuuBteiee0"

Это инструкция, которая может содержать ветвления. Операнд 0 представляет собой уникальный идентификатор выражения языка RTL. Операнд 1 — это указатель на предыдущую, а операнд 2 — на следующую инструкцию в цепочке. Операнд 3 содержит базовый блок инструкций, которые не могут ветвиться. Операнд 4 представляет собой указатель на узел дерева.

label_ref 'o' "u00"

Ссылка на метку в языке ассемблера. Операнд 0 представляет собой инструкцию **code_label**, которая содержится в цепочке инструкций. Также см. **symbol_ref**. Операнд 1 содержит объявление **LABEL_NEXTREF**, используемое в файле **flow.c**. Операнд 2 представляет собой объявление **CONTAINING_INSN**, используемое в файле **flow.c**.

le '<' "ee"

Генерируемый код вычисляет значения выражений и производит сравнение значений с учетом знака. Инструкция дает результат **true**, если первое значение меньше или равно второму.

leu '<' "ee"

Генерируемый код вычисляет значения выражений и производит сравнение значений без учета знака. Инструкция дает результат `true`, если первое значение меньше или равно второму.

lo_sum 'o' "ee"

Сумма регистра и младших битов константного выражения на RISC-компьютере.

lsshiftrt '2' "ee"

Поразрядный логический сдвиг вправо. Операнд 0 представляет собой выражение, представляющее сдвигаемое значение, а операнд 1 — выражение, определяющее количество разрядов, на которое производится сдвиг.

lt '<' "ee"

Генерируемый код вычисляет значения выражений и производит сравнение значений с учетом знака. Инструкция дает результат `true`, если первое значение меньше второго.

ltu '<' "ee"

Генерируемый код вычисляет выражения и выполняет сравнение с учетом знака. Возвращает `true` когда первое значение меньше второго.

ltu '<' "ee"

Генерируемый код вычисляет значения выражений и производит сравнение значений без учета знака. Инструкция дает результат `true`, если первое значение меньше второго.

match_insn 'm' "is"

Эта инструкция используется только в описаниях машин. Операнд 0 содержит индекс для таблицы operandов, а operand 1 — имя функции, которая будет производить сравнение.

match_dup 'm' "i"

Эта инструкция используется только в описаниях машин. Сравнение производится с содержимым указанной ячейки таблицы operandов. Операнд 0 представляет собой индекс для таблицы operandов.

match_op_dup 'm' "iE"

Эта инструкция используется только в описаниях машин. Сравнение производится с содержимым элемента таблицы operandов с указанным индексом. Операнд 0 представляет собой индекс для таблицы operandов, а operand 1 — вектор сравниваемых выражений.

match_operand 'm' "iss"

Эта инструкция используется только в описаниях машин. Она представляет собой сравнение функционального равенства двух operandов. Операнд 0 содержит индекс для таблицы operandов. Операнд 1 содержит имя функции, используемой для выполнения сравнения. Операнд 2 указывает имя первого операнда, а операнд 3 — второго операнда, участвующих в сравнении.

match_operator 'm' "isE"

Эта инструкция используется только в описаниях машин. Она рекурсивно сравнивает operandы выражений языка RTL. Операнд 0 содержит индекс для таблицы operandов. Операнд 1 содержит имя функции, используемой для выполнения сравнения. Операнд 2 представляет собой вектор сравниваемых operandов.

match_par_dup 'm' "iE"

Эта инструкция используется только в описаниях машин. Она применяется только для сравнения элемента, находящегося в таблице operandов в позиции с указанным индексом. Операнд 0 содержит индекс для таблицы operandов, а операнд 1 — вектор сравниваемых выражений.

match_parallel 'm' "isE"

Эта инструкция используется только в описаниях машин. Она применяется для сравнения вектора operandов — набора параллельных инструкций — путем вызова указанной функции. Операнд 0 содержит индекс для таблицы operandов, а операнд 1 — имя функции, которая будет производить сравнение. Операнд 2 представляет собой вектор параллельных инструкций, сравнение которых будет производиться.

match_scratch 'm' "is"

Эта инструкция используется только в описаниях машин. Описание формы использования в качестве выражения языка RTL приведено в описании инструкции **scratch**. Сравнение производится на наличие вспомогательного регистра. Операнд 0 содержит индекс для таблицы operandов, а операнд 1 — имя функции, которая будет производить сравнение.

mem 'o' "e0"

Адрес памяти. Операнд 0 содержит адрес памяти. Операнд 1 указывает набор, которому принадлежит данный адрес памяти.

minus '2' "ee"

Производится вычисление значений обоих operandов, и значение операнда 1 вычитается из значения операнда 0.

mod '2' "ee"

Производится вычисление значений обоих выражений, и значение операнда 1 делится на значение операнда 0. В качестве результата берется целочисленный остаток от деления.

mult '1' "e"

Производится вычисление обоих выражений, и окончательный результат инструкции вычисляется как произведение значений операндов.

ne '<' "ee"

Генерируемый код вычисляет значения выражений и выполняет сравнение с учетом знака. Результатом инструкции будет `true`, если первое значение не равно второму.

neg '1' "e"

Вычисляется значение выражения, и его знак изменяется на противоположный.

nil 'x' "*"

Нулевой указатель.

not '1' "e"

Вычисляется значение выражения, а затем выполняется операция поразрядного НЕ (с добавлением единицы для нахождения двоичного дополнения числа).

note 'x' "iuuB0ni"

Указывает место кода, где начинается строка исходного кода. Операнд 0 содержит уникальный идентификатор данного выражения языка RTL. Оператор 1 представляет собой указатель на предыдущую, а оператор 2 — на следующую инструкцию в цепочке. Оператор 3 содержит базовый блок инструкций, следующих после метки. Оператор 4 указывает имя файла, если номер строки больше нуля. В противном случае эти данные применяются только для данной инструкции. Оператор 5 представляет собой номер строки кода. Если номер строки равен нулю, это значение равно `enumnote_insn`. Если номер строки равен `note_insn_deleted_label`, то операнд 6 содержит уникальное значение.

ordered '<' "ee"

Сгенерированный код вычисляет значение выражений и выполняет упорядоченное сравнение чисел с плавающей запятой. Если хотя бы одно из значений равно "`Nan`", то упорядоченное сравнение вызывает исключение. Результат выполнения инструкции равен `true`, только если значения равны. Также см. `unordered`.

parallel 'x' "E"

Массив двух или большего количества операций, которые должны выполняться одновременно.

pc 'o' ""

Счетчик команд. Переходы указываются в виде операторов `set`, операнд 0 которых равен "`pc`".

phi 'x' "E"

SSA-оператор **phi**, который может находиться только в начале базового блока. Операнд представляет собой вектор, содержащий 2^n выражений языка RTL. В массиве элемент $2n+1$ является инструкцией **const_int** с идентификатором предшествующего блока, через который было передано управление при использовании элемента $2n$.

plus 'c' "ee"

Вычисляются значения обоих выражений, и полученные значения складываются.

post_dec 'a' "e"

Постдекремент адреса памяти, заданного выражением. Шаг декремента не указывается, поскольку тип элемента может быть определен из выражения.

post_inc 'a' "e"

Постинкремент адреса памяти, заданного выражением. Шаг инкремента не указывается, поскольку тип элемента может быть определен из выражения.

post_modify 'a' "ee"

Представляет побочный эффект для адреса (за исключением инкремента и декремента — для них используются другие операторы). Также см. **pre_modify**. Операнд 0 представляет собой инструкцию **reg**, используемую в качестве адреса. Операнд 1 содержит выражение, результат которого присваивается регистру. Этот операнд должен иметь форму **plus(reg) (reg)** или **plus(reg) (const_int)**, где первый operand инструкции **plus** соответствует первому operandу инструкции **post_modify**.

pre_dec 'a' "e"

Преддекремент адреса памяти, заданного выражением. Шаг декремента не указывается, поскольку тип может быть определен из выражения.

pre_inc 'a' "e"

Преинкремент адреса памяти, заданного выражением. Шаг инкремента не указывается, поскольку тип может быть определен из выражения.

pre_modify 'a' "ee"

Представляет побочный эффект для адреса (за исключением инкремента и декремента — для них используются другие операторы). Операнд 0 представляет собой инструкцию **reg**, которая используется в качестве адреса. Операнд 1 содержит выражение, которое присваивается регистру. Этот операнд должен иметь форму **plus(reg) (reg)** или **plus(reg) (const_int)**, где первый operand инструкции **plus** соответствует первому operandу инструкции **pre_modify**.

`prefetch 'x' "eee"`

Упреждающая выборка из памяти с атрибутами, поддерживаемыми на некоторых целевых машинах. Операнды 1 и 2 будут игнорироваться для платформ, которые их не поддерживают. Операнд 0 содержит адрес памяти, из которого будет производиться выборка. Для получения доступа на считывание значение операнда 1 устанавливается равным 0, а для получения доступа на запись — равным 1. Операнд 2 представляет собой число, указывающее уровень временного размещения, где 0 соответствует отсутствию, а 1, 2 и 3 — возрастающему уровню временного размещения.

`presence_set 'x' "ss"`

Эта инструкция встречается только в описаниях машин и используется для задания перечня функциональных блоков процессора, которые не могут быть резервированными, если не резервированы некоторые другие функциональные блоки. Также см. `absence_set`. Операнд 0 содержит перечень разделенных запятыми функциональных блоков, которые не могут быть резервированными, если не резервирован ни один функциональный блок из перечня операнда 2. Операнд 1 содержит перечень функциональных блоков.

`queued 'x' "eeeeee"`

Указатель на элемент очереди инструкций, генерируемых для последующего постинкремента или постдекремента. Таким образом инструкция из очереди никогда не попадает в генерируемый код. Поставленное в очередь выражение помещается в инструкцию. Таким образом, используется значение, получаемое перед инкрементом или декрементом. Операнд 0 содержит переменную (или регистр), для которой будет выполняться операция инкремента или декремента. Операнд 1 представляет собой инструкцию, выполняющую инкремент или декремент. Операнд 2 — это выражение `reg` языка RTL, содержащее исходное значение переменной. Операнд 3 представляет собой код, используемый в качестве инструкции инкремента или декремента. Операнд 4 указывает на следующее выражение `queued` в очереди.

`range_info 'x' "uuEiiiiibbbii"`

Заголовок для области данных. Операнд 0 представляет собой указатель на инструкцию `note`, обозначающую начало области, а оператор 1 — на инструкцию `note`, обозначающую конец области. Операнд 2 содержит вектор всех регистров, которые могут заменяться в пределах данной области. Операнд 3 — это количество вызовов внутри области. Операнд 4 указывает общее количество инструкций в области. Оператор 5 содержит уникальный идентификатор области. Операнд 6 представляет собой номер базового блока начала области, операнд 7 — номер базового блока конца области. Операнд 8 содержит глубину цикла. Операнд 9 является битовой картой, указывающей, какие регистры определены в начале области. Операнд 10 — это битовая карта (bitmap), указывающая, какие регистры определены в конце области. Операнд 11 представляет собой номер маркера начала области, а операнд 12 — номер маркера конца области.

range_live 'x' "bi"

Это информация о регистрах, которые определены в данной точке. Операнд 0 представляет собой битовую карту (bitmap), содержащую перечень доступных регистров. Оператор 1 указывает исходный номер данного блока.

range_reg 'x' "iiiiiiitt"

Эта инструкция определяет регистры, которые могут быть изменены внутри области. Операнд 0 содержит исходный номер псевдорегистра. Операнд 1 определяет значение псевдорегистра в пределах области. Операнд 2 указывает количество ссылок на регистр внутри области. Операнд 3 содержит количество изменений содержимого регистра в пределах области. Операнд 4 указывает количество отключений регистра в пределах области. Операнд 5 содержит флаги, которые указывают, нужно ли копировать данные из исходного регистра в новый регистр в начале области и нужно ли копировать данные из нового регистра в исходный регистр в конце области. Операнд 6 содержит время жизни регистра. Операнд 7 указывает количество вызовов за время жизни данных регистра. Операнд 8 представляет собой символьный узел переменной, если регистр содержит переменную. Операнд 9 — узел блока, в котором объявлена переменная, если регистр содержит переменную.

range_var 'x' "eti"

Эта инструкция содержит информацию об областях локальной переменной. Операнд 0 представляет собой список **expr_list**, содержащий области, в которых переменная копируется в псевдорегистр. Операнд 1 указывает блок, в котором объявлена переменная. Операнд 2 содержит количество областей, в которых переменная используется.

reg 'o' "i0"

Это аппаратный регистр или псевдорегистр. Также см. **scratch**. Операнд 0 содержит номер регистра. Если этот номер меньше чем **FIRST_PSEUDO_REGISTER**, то регистр аппаратный. Операнд 1 указывает исходный номер регистра, который будет отличаться от номера псевдорегистра, преобразованного в аппаратный регистр.

resx 'x' "i"

Это заполнитель для возможной вставки значения **_Unwind_Resume**, использование которого может понадобиться до определения необходимого действия: вызова функции или ветвления. Операнд 0 представляет собой область исключений, из которой передается управление.

return 'x' ""

Возврат из подпрограммы.

rotate '2' "ee"

Поразрядный сдвиг влево без учета знака. Разряды, выходящие слева за пределы числа, переносятся на его правую сторону. Операнд 0 содержит выражение, значение которого подвергается сдвигу, а операнд 1 — выражение, указывающее количество позиций сдвига.

`rotatert '2' "ee"`

Поразрядный сдвиг вправо без учета знака. Разряды, выходящие справа за пределы числа, переносятся на его левую сторону. Операнд 0 содержит выражение, значение которого подвергается сдвигу, а операнд 1 — выражение, указывающее количество позиций сдвига.

`scratch 'o' "0"`

Вспомогательный регистр. Это регистр, который используется только в пределах одной инструкции. При выделении регистров или их перегрузке эта инструкция будет преобразована в инструкцию `reg`. Операнд используется только для облегчения преобразования данной инструкции в `reg`.

`sequence 'x' "E"`

Эта форма последовательности инструкций является результатом генерирования кода, основанного на инструкции `define_expand`, которая формирует несколько инструкций. Функция `emit_insn()` разбивает последовательность `sequence` на отдельные инструкции. Операнд 0 содержит массив выражений.

`set 'x' "ee"`

Операция присваивания, предназначенная для записи значения в определенное место. Все операции присвоения должны использовать инструкцию `set`. Инструкции, требующие нескольких присвоений, должны использовать несколько инструкций `set`. Операнд 0 содержит значение левой части выражения присваивания (`lvalue`), указывающее на место записи присваиваемого значения (память, регистр, условный код и т.д.). Операнд 1 представляет правую часть (`rvalue`) выражения присваивания — значение или место расположения значения, записываемого по адресу, на который указывает левая часть.

`set_attr 'x' "ss"`

Эта инструкция может использоваться в качестве последнего операнда инструкций `define_insn`, `define_peephole` или `define_asm_insn` для задания атрибута, который назначается соответствующим заданному шаблону инструкциям. Операнд 0 содержит имя атрибута, а операнд 1 — значение атрибута.

`set_attr_alternative 'x' "sE"`

Эта инструкция может использоваться в качестве последнего операнда инструкции `define_insn` или `define_peephole` для указания набора альтернативных значений атрибута. То, какое значение будет присвоено, определяется на основе результатов сравнения. Операнд 0 содержит имя атрибута, а операнд 1 — массив возможных значений атрибута.

`sign_extend '1' "e"`

В результате вычисления выражения расширяются знаковые разряды. Количество знаковых разрядов определяется режимами машины и типом выражения. Также см. `zero_extend`.

sign_extract 'b' "eee"

Это указание размера и положения битового поля знака числа. Также см. **zero_extract**. Операнд 0 представляет собой блок памяти, содержащий первый бит знакового битового поля. Операнд 1 содержит количество битов в поле. Операнд 2 указывает смещение битового поля, т.е. количество бит блока памяти перед первым битом поля. Если установлена переменная **BITS_BIG_ENDIAN**, счет ведется от старшего бита блока памяти. В противном случае отсчет будет вестись от младшего бита.

smax 'c' "ee"

Сравнение чисел с учетом знака, результатом которого является большее значение из двух.

smin 'c' "ee"

Сравнение чисел с учетом знака, результатом которого является меньшее значение из двух.

sqrt '1' "e"

Извлекает квадратный корень из значения, полученного после вычисления выражения.

ss_minus '2' "ee"

Вычисляются значения обоих выражений, и значение операнда 1 вычитается из значения операнда 0 с учетом знака. Также см. **us_minus**.

ss_plus 'c' "ee"

Вычисляются значения обоих выражений, а затем складываются с учетом знака. Также см. **us_plus**.

ss_truncate '1' "e"

Вычисляется значение выражения, которое затем усекается с учетом знака. Также см. **us_truncate**.

strict_low_part 'x' "e"

Присваивание значения, изменяющего только младшие разряды указанного для записи результата расположения. Операнд 0 содержит выражение присваивания, на которое наложено ограничение **strict_low_part**.

subreg 'x' "ei"

Одно слово значения, состоящего из нескольких машинных слов. Операнд 0 содержит выражение, содержащее полное значение. Операнд 1 представляет собой селектор слова в значении.

symbol_ref 'o' "s"

Ссылка на именованную метку. Также см. **label_ref**. Операнд 0 содержит строку метки (с предшествующим символом подчеркивания). Если имя метки начинается

с символа звездочки, то звездочка удаляется, а символ подчеркивания не добавляется.

trap_if 'x' "ee"

Условная ловушка. Для безусловной ловушки значение условного выражения устанавливается равным "1". Операнд 0 содержит условное выражение. Операнд 1 представляет собой код, который должен выполняться, если значение условного выражения не равно нулю.

truncate '1' "e"

Результат вычисления выражения усекается для записи в заданное расположение.

udiv '2' "ee"

Вычисляются значения обоих выражений, и значение операнда 1 делится на значение операнда 0. Выполняется целочисленное деление без учета знака.

unordered '<' "ee"

Сгенерированный код вычисляет значение выражений и выполняет неупорядоченное сравнение чисел с плавающей точкой. Неупорядоченное сравнение не вызывает исключений, даже если значения выражений равны "NaN". Результат выполнения инструкции равен `true`, только если значения равны. Также см. `ordered`.

umax 'c' "ee"

Сравнение без учета знака, результатом которого является большее значение из двух.

umin 'c' "ee"

Сравнение без учета знака, результатом которого является меньшее значение из двух.

umod '2' "ee"

Производится вычисление значений обоих выражений, и значение операнда 1 делится на значение операнда 0. В качестве результата берется остаток от целочисленного деления без учета знака.

uneq '<' "ee"

Генерируемый код вычисляет значения выражений и производит неупорядоченное сравнение значений — чисел с плавающей точкой. Инструкция дает `true`, если первое значение равно второму.

unge '<' "ee"

Генерируемый код вычисляет значения выражений и производит неупорядоченное сравнение значений с плавающей точкой. Инструкция дает `true`, если первое значение больше или равно второму.

ungt '<' "ee"

Генерируемый код вычисляет значения выражений и производит неупорядоченное сравнение значений с плавающей точкой. Инструкция дает **true**, если первое значение больше второго.

unle '<' "ee"

Генерируемый код вычисляет значения выражений и производит неупорядоченное сравнение значений с плавающей точкой. Инструкция дает **true**, если первое значение меньше или равно второму.

unlt '<' "ee"

Генерируемый код вычисляет значения выражений и производит неупорядоченное сравнение значений с плавающей точкой. Инструкция дает **true**, если первое значение меньше второго.

UnKnown 'x' "*"

Выражение языка RTL, которое на текущий момент не имеет определенного типа.

unsigned_fix '1' "e"

Преобразовывает значение выражения с плавающей точкой в значение с фиксированной точкой без знака. Также см. **signed_fix**.

unsigned_float '1' "e"

Преобразовывает значение выражения с фиксированной точкой в значение с плавающей точкой без знака. Также см. **signed_float**.

unspec 'x' "Ei"

Машинно-зависимая операция. Операнд 0 представляет собой вектор operandов, используемых для операции. Операнд 1 содержит индекс элемента вектора operandов, указывающий на тот operand, который будет использоваться.

unspec_volatile 'x' "Ei"

Машинно-зависимая операция, в ней может содержаться ловушка. Операнд 0 представляет собой вектор operandов, используемых для операции. Операнд 1 содержит индекс вектора operandов, который указывает, какой operand будет использоваться.

us_minus '2' "ee"

Производится вычисление значений обоих operandов, и значение operand'a 1 вычитается из значения operand'a 0 без учета знака. Также см. **ss_minus**.

us_plus 'c' "ee"

Вычисляются значения обоих выражений, а затем складываются без учета знака. Также см. **ss_plus**.

us_truncate '1' "e"

Вычисляется значение выражения, которое затем усекается без учета знака. Также см. **ss_truncate**.

use 'x' "e"

Указатель того, что что-то используется таким способом, который не требует объяснения. Например, вызов подпрограммы в качестве своей вызывающей последовательности будет использовать регистр. Также см. **clobber**. Операнд 0 представляет собой выражение, указывающее на используемый элемент.

value '0' "0"

Используется функциями **cselib** для описания значения.

vec_concat 'x' "ee"

Описывает конкатенацию двух векторов. В результате выполнения конкатенации образуется вектор с длиной, равной сумме длин исходных векторов, причем элементы вектора из операнда 0 предшествуют элементам вектора из операнда 1. Операнд 0 содержит первый вектор, а операнд 1 — второй вектор.

vec_duplicate 'x' "e"

Описывает операцию, которая в целое количество раз увеличивает длину вектора, копируя все его члены. Длина результирующего вектора кратна длине исходного вектора.

vec_merge 'x' "eee"

Описывает операцию слияния двух векторов. Операнд 0 содержит первый вектор, а операнд 1 — второй вектор. Операнд 2 представляет собой битовую маску, указывающую, откуда должны извлекаться составляющие элементы результирующего вектора. Значение "0" первого бита говорит о том, что элемент должен браться из операнда 0, а значение "1" — из операнда 1.

vec_select 'x' "ee"

Описывает операцию, которая выбирает отдельные части вектора. Операнд 0 содержит исходный вектор, а операнд 1 — параллельную инструкцию, включающую константные значения, которые указывают, какие элементы исходного вектора должны копироваться в результирующий вектор.

xor 'c' "ee"

Вычисляются значения обоих operandов, и к полученным значениям применяется операция поразрядного исключающего ИЛИ.

zero_extend '1' "e"

В результате вычисления выражения знаковые разряды не расширяются, а производится заполнение нулями. Также см. **sign_extend**.

zero_extract 'b' "eee"

Это указание размера и положения битового поля без знака. Также см. **sign_extract**. Операнд 0 указывает блок памяти, содержащий первый бит битового поля без знака. Операнд 1 содержит количество битов в поле. Операнд 2 указывает смещение битового поля, т.е. количество бит блока памяти перед первым битом поля. Если установлена переменная **BITS_BIG_ENDIAN**, счет ведется от старшего бита блока памяти. В противном случае отсчет будет вестись от младшего бита.

Режимы и классы режимов

Для каждого выражения языка RTL определен режим, описывающий размер и тип данных, обрабатываемых выражением и получаемых в результате его вычисления. Два идентичных выражения могут приводить к генерации совершенно различного кода. Примерами могут служить варианты вычисления выражения, содержащего числа с плавающей точкой и содержащего целые числа. Описание режимов приведено в таблице 20.5. Режимы определены в файле **machmode.def** в виде набора перечисляемых типов.

Таблица 20.5. Режимы, применяемые к выражениям

Режим	Описание
BImode	Битовый режим. Определяет операцию над одним битом.
QImode	Целочисленный режим четвертной точности. Определяет операцию над одним байтом, который считается целым значением.
HImode	Целочисленный режим половинной точности. Определяет операцию над двумя байтами, которые считаются одним целым числом.
PSImode	Целочисленный режим частичной точности. Определяет операцию над целочисленным значением, занимающим четыре байта, но фактически не использующим все четыре байта.
SImode	Целочисленный режим нормальной точности. Определяет операцию над целочисленным значением длиной в четыре байта.
PDImode	Целочисленный режим частичной двойной точности. Определяет операцию над целочисленным значением, занимающим восемь байтов, но фактически не использующим все восемь байтов.
DImode	Целочисленный режим двойной точности. Определяет операцию над целочисленным значением длиной восемь байтов.
TImode	Целочисленный режим четырехкратной точности. Определяет операцию над целочисленным значением длиной шестнадцать байтов.
OImode	Целочисленный режим восьмикратной точности. Определяет операцию над целочисленным значением длиной в тридцать два байта.
QFmode	Режим значений с плавающей точкой четвертной точности. Определяет операцию над числом с плавающей точкой длиной один байт.
HFmode	Режим значений с плавающей точкой половинной точности. Определяет операцию над числом с плавающей точкой длиной в два байта.
TQFmode	Режим значений с плавающей точкой точности две третьих. Определяет операцию над числом с плавающей точкой длиной три байта.

Режим	Описание
SFmode	Режим значений с плавающей десятичной точкой нормальной точности. Определяет операцию над числом с плавающей десятичной точкой длиной четыре байта. В соответствии со стандартом IEEE это операция над числом с плавающей точкой нормальной точности, которая манипулирует 8-битными байтами, но может отличаться на машинах, работающих с байтами длиной 16 бит, и другом аппаратном обеспечении с собственной формой арифметики чисел с плавающей точкой.
DFmode	Режим значений с плавающей точкой двойной точности. Определяет операцию над числом с плавающей точкой длиной восемь байтов. В соответствии со стандартом IEEE это операция над числом плавающей десятичной точкой двойной точности, которая манипулирует 8-битными байтами. Она может отличаться на машинах, работающих с байтами длиной 16 бит, и другом аппаратном обеспечении со своей собственной формой арифметики чисел с плавающей точкой.
XFmode	Режим значений с плавающей точкой расширенной точности. Определяет операцию над числом с плавающей точкой длиной в двенадцать байт. В соответствии со стандартами IEEE это операция над числом с плавающей точкой расширенной точности, которая манипулирует 8-битными байтами. На некоторых системах будут использоваться не все 12 байт.
TFmode	Режим значений с плавающей точкой четырехкратной точности. Определяет операцию над числом с плавающей точкой длиной шестнадцать байтов. Этот режим используется как для расширенных типов значений с плавающей точкой длиной 96 бит в соответствии со стандартами IEEE, дополненных до 128 бит, так и со значениями длиной 128 бит.
CCmode	Код условия. Определяет операцию над значением условного кода. Условный код представляет собой машинно-зависимый набор аппаратных битов, используемых для хранения результатов операций сравнения. Этот режим не применяется на машинах, использующих инструкцию <code>cc0</code> .
BLKmode	Режим блоков. Определяет операцию над значениями составных типов, для которых не применим никакой другой режим. В языке RTL этот режим применяется только для ссылок на адреса памяти, используемых при выполнении аппаратных инструкций для работы с векторами. Он не применяется на машинах, на которых такие инструкции отсутствуют.
VOIDmode	Режим значений неизвестного типа. Этот режим используется при отсутствии определенного конкретного режима. Константные выражения могут использовать этот режим, поскольку их можно преобразовать в любой режим в зависимости от их значения.
QCmode	Режим комплексных чисел четвертной точности, в котором комплексное число составляется из двух чисел с плавающей точкой в режиме <code>SFmode</code> .
HCmode	Режим комплексных чисел половинной точности, в котором комплексное число составляется из двух чисел с плавающей точкой в режиме <code>XFmode</code> .
SCmode	Режим комплексных чисел нормальной точности, в котором комплексное число составляется из двух чисел с плавающей десятичной точкой в режиме <code>SFmode</code> .
DCmode	Режим комплексных чисел двойной точности, в котором комплексное число составляется из двух чисел с плавающей точкой в режиме <code>DFmode</code> .
XCmode	Режим комплексных чисел расширенной точности, в котором комплексное число составляется из двух чисел с плавающей точкой в режиме <code>XFmode</code> .
TCmode	Режим комплексных чисел четырехкратной точности, в котором комплексное число составляется из двух чисел с плавающей точкой в режиме <code>TFmode</code> .
CQImode	Режим целых комплексных чисел четвертной точности, в котором комплексное число составляется из двух чисел с плавающей точкой в режиме <code>QImode</code> .

Режим	Описание
<code>CHImode</code>	Режим целых комплексных чисел половинной точности, в котором комплексное число составляется из двух чисел с плавающей точкой в режиме <code>HImode</code> .
<code>CSImode</code>	Режим целых комплексных чисел нормальной точности, в котором комплексное число составляется из двух чисел с плавающей точкой в режиме <code>SImode</code> .
<code>CDImode</code>	Режим целых комплексных чисел двойной точности, в котором комплексное число составляется из двух чисел с плавающей точкой в режиме <code>DImode</code> .
<code>CTImode</code>	Режим комплексных чисел четырехкратной точности, в котором комплексное число составляется из двух чисел с плавающей точкой в режиме <code>TImode</code> .
<code>COImode</code>	Режим комплексных чисел восьмикратной точности, в котором комплексное число составляется из двух чисел с плавающей точкой в режиме <code>OImode</code> .

В блоках отладочной информации языка RTL и в описаниях машин имя режима операции указывается непосредственно за выражением после двоеточия. Например, регистровое выражение может быть записано следующим образом: (`reg:SI 7 esp`) или, если регистровый флаг установлен, (`reg/f:SI 7 esp`). Имя режима всегда указывается без слова `mode`. Если в данных не содержится наименования режима, то используется режим `VOIDmode`.

Не каждая машина позволяет поддерживать все режимы. Необходимым условием является поддержка режима `QImode` (однобайтное целое число), типов целых чисел, размер которых соответствует определенному константой `BITS_PER_WORD`, и типов чисел с плавающей точкой, размеры которых определены константами `FLOAT_TYPE_SIZE` и `DOUBLE_TYPE_SIZE`. Эти константы устанавливаются при настройке конфигурации компилятора и совпадают с соответствующими параметрами для целевой машины.

Несмотря на то, что в коде языка RTL режим устанавливается явным образом, сам компилятор часто использует ссылки на классы режимов. Классы режимов представлены в таблице 20.6 и определены в виде значений перечисления `emt` в файле `machmode.h`.

Таблица 20.6. Имена классов режимов и описание соответствующих им режимов

Классы режимов	Описание
<code>MODE_INT</code>	Класс режимов целых чисел, содержит режимы <code>BImode</code> , <code>QImode</code> , <code>HImode</code> , <code>SImode</code> , <code>DImode</code> , <code>TImode</code> и <code>OImode</code> .
<code>MODE_PARTIAL_INT</code>	Класс частичных целочисленных режимов, содержит режимы <code>PQImode</code> , <code>RHImode</code> , <code>TQImode</code> и <code>PDImode</code> .
<code>MODE_FLOAT</code>	Класс режимов для работы с числами с плавающей точкой, содержит режимы <code>QFmode</code> , <code>HFmode</code> , <code>TQFmode</code> , <code>SFmode</code> , <code>DFmode</code> , <code>XFmode</code> и <code>TFmode</code> .
<code>MODE_COMPLEX_INT</code>	Класс режимов для работы с комплексными числами, составленными из целых чисел, содержит режимы <code>CQImode</code> , <code>CHImode</code> , <code>CSImode</code> , <code>CDImode</code> , <code>CTImode</code> и <code>COImode</code> .
<code>MODE_COMPLEX_FLOAT</code>	Класс режимов для работы с комплексными числами, составленными из чисел с плавающей десятичной точкой, содержит режимы <code>QCmode</code> , <code>HCmode</code> , <code>SCmode</code> , <code>DCmode</code> , <code>XСmode</code> и <code>TCmode</code> .
<code>MODE_CC</code>	Класс кодов условий, содержит режим <code>CCmode</code> и любые другие режимы, которые могут быть определены макросом <code>EXTRA_CC_MODES</code> .

Классы режимов	Описание
MODE_RANDOM	Произвольный класс. Используется для представления любого режима, не входящего в состав других классов, например, <code>BLKmode</code> и <code>VOIDmode</code> .

Глобальная переменная `byte_mode` содержит режим, соответствующий значению константы `BITS_PER_UNIT` (на 32-битной машине она имеет значение `QImode`). Глобальная переменная `word_mode` содержит режим, соответствующий значению константы `BITS_PER_WORD` (на 32-битной машине оно равно `SImode`).

Флаги

В каждую инструкцию языка RTL может включаться целый ряд классов. На распечатке кода флаги представляют собой одиночные буквенные символы, перед которым указывается обратная косая черта, как показано в главе 18. Точное значение флага зависит от типа инструкции, использующей данный флаг. В таблице 20.7 приведен перечень инструкций, которые могут использоваться с флагами, а также флаг, выводимый в листинге кода на языке RTL.

Таблица 20.7. Значение флагов для различных инструкций

Инструкция	Флаг	Значения флага
<code>asm_input</code>	/v	Данные, находящиеся в памяти по указанному адресу, являются подвижными. Инструкция не может быть удалена, перемещена или сгруппирована.
<code>asm_operands</code>	/v	Данные, находящиеся в памяти по указанному адресу, являются подвижными. Инструкция не может быть удалена, перемещена или сгруппирована.
<code>call_insn</code>	/j	Вызов инструкции этого же уровня.
<code>call_insn</code>	/u	Вызов чистой функции или константы.
<code>code_label</code>	/s	Метка является целью нелокального оператора <code>goto</code> и не может быть удалена. Если такую метку удалить, ее нужно заменить инструкцией, указывающей, что метка была удалена.
<code>const</code>	/i	Этот код языка RTL был сгенерирован путем интеграции процедуры.
<code>expr_list</code>	/u	Вызов чистой функции или константы.
<code>insn</code>	/f	Это часть пролога функции, который устанавливает указатель стека. Он устанавливает указатель кадра стека, сохраняет значение регистра или создает временный регистр, который будет использоваться вместо указателя кадра.
<code>insn</code>	/i	Этот код языка RTL был сгенерирован путем интеграции процедур.
<code>insn</code>	/s	При удалении неиспользуемого кода этот флаг отмечает неиспользуемый код. При реорганизации в позиции задержки ветвления этот флаг указывает, что данная инструкция принадлежит ветвлению. При планировании этот флаг указывает, что данная инструкция должна планироваться совместно с предыдущей.
<code>insn</code>	/v	Эта инструкция удалена.
<code>insn_list</code>	/i	Этот код языка RTL был сгенерирован путем интеграции процедуры.
<code>jump_insn</code>	/s	При реорганизации в позиции задержки ветвления этот флаг указывает, что данная инструкция принадлежит к ветвлению.

Инструкция	Флаг	Значения флага
label_ref	/v	Используется ссылка на метку, которая находится вне внутреннего цикла, содержащего данную инструкцию.
label_ref	/v	Это ссылка на нелокальную метку.
mem	/f	Это ссылка на скалярное значение.
mem	/j	Псевдоним, установленный для этой ссылки на адрес памяти, не должен изменяться при получении доступа к компоненту памяти.
mem	/w	Используемая память является частью блока памяти (структура или массив). Этот флаг не устанавливается, если доступ к памяти осуществляется с помощью указателя C, указывающего на скалярное число или составной тип данных.
mem	/u	Значение данной ячейки памяти никогда не изменяется.
mem	/v	Данные, находящиеся в памяти по указанному адресу, являются подвижными. Инструкция не может быть удалена, перемещена или сгруппирована.
note	/u	Вызов чистой функции или константы.
reg	/f	Регистр содержит указатель.
reg	/i	Этот регистр содержит значение, которое должно возвращаться текущей функцией. На машинах, которые имеют специальный регистр для возвращаемого значения, этот флаг не устанавливается.
reg	/w	Содержимое регистра используется при вычислении условного выражения во все время выполнения цикла.
reg	/u	Значение регистра никогда не изменяется.
reg	/v	Этот флаг указывает переменную, определенную пользователем. Кроме того, это может быть временная переменная, созданная компилятором.
reg_label	/v	Ссылка на нелокальную метку.
set	/f	Это часть пролога функции, который устанавливает указатель стека. Он устанавливает указатель на кадр стека, сохраняет значение регистра или создает временный регистр, который будет использоваться вместо указателя на кадр стека.
set	/j	Устанавливается значение для возврата.
subreg	/w	Осуществляется доступ к объекту, режим которого получен на основании более широкого режима.
subreg	/u	Ссылка на значение без знака, режим которого получен на основании более широкого режима.
symbol_ref	/f	Ссылка указывает на пул строковых констант этой функции.
symbol_ref	/i	Ссылка на слабый (замещаемый) символ.
symbol_ref	/u	Ссылка на объект в пуле констант текущей функции.
symbol_ref	/v	Назначение этого флага зависит от машины.

Глава 21



Машинно-зависимые опции компилятора

В этой главе содержится информация об опциях компилятора GCC, которые можно применять при компиляции программ на специфические аппаратные платформы. По большей части опции и установки имеют отношение к настройке генерируемого кода для использования (или отказа от использования) характерных особенностей аппаратного обеспечения. Кроме того, имеются специальные опции для отладки программ и организации разделов в объектных файлах.

Перечень аппаратных платформ

Ниже приведен перечень файлов описания аппаратных платформ, поддержка которых входит в версию 3.1 компилятора GCC; все эти файлы находятся в каталоге `config` исходного дистрибутива GCC:

alpha/alpha.md	m88k/m88k.md
alpha/ev4.md	mcore/mcore.md
alpha/ev5.md	mips/mips.md
alpha/ev6.md	mmix/mmix.md
arc/arc.md	mn10200/mn10200.md
arm/arm.md	mn10300/mn10300.md
avr/avr.md	ns32k/ns32k.md
c4x/c4x.md	pa/pa.md
cris/cris.md	pdp11/pdp11.md
d30v/d30v.md	romp/romp.md

dsp16xx/dsp16xx.md	rs600/rs600.md
fr30/fr30.md	s390/s390.md
h8300/h8300.md	sh/sh.md
i370/i370.md	sparc/cypress.md
i386/i386.md	sparc/sparc.md
i386/athlon.md	sparc/hyperspark.md
i386/k6.md	sparc/sparclet.md
i386/pentium.md	sparc/supersparc.md
i386/ppro.md	sparc/ultra1_2.md
i960/i960.md	sparc/ultra3.md
ia64/ia64.md	stormy16/stormy16.md
m32r/m32r.md	v850/v850.md
m68hc11/m68hc11.md	vax/vax.md
m68k/m68k.md	

Каждый каталог соответствует отдельной платформе. В некоторых каталогах находится несколько файлов .md, поскольку директива `include` позволяет описывать полную конфигурацию с помощью набора файлов. Например, файл `alpha.md` из каталога `alpha` подключает файлы `ev4.md`, `ev5.md` и `ev6.md`, а файл `i386.md` из каталога `i386` — файлы `pentium.md`, `k6.md` и `athlon.md`.

ОПЦИИ КОМАНДНОЙ СТРОКИ ДЛЯ ЗАПУСКА КОМПИЛЯТОРА GCC

Для большого набора аппаратных платформ существуют особые наборы средств нижнего уровня GCC (они также называются *портами GCC*). Многие, хотя и не все существующие порты переноса программ имеют специальные опции командной строки. Они указывают компилятору генерировать код, в большей степени соответствующий режиму работы аппаратного обеспечения или его конфигурации во время выполнения программы. В последующих разделах будут приведены доступные опции `-m` для тех портов, для которых они определены.

Большинство опций начинается с пары символов "`-m`", но есть несколько исключений. Отдельные опции используются для указания генерирования кода для определенного процессора из семейства процессоров, в то время как другие — для генерации кода, учитывающего особенности аппаратного обеспечения. Это позволяет оптимальным образом сопоставить свойства разработанной программы с возможностями аппаратного обеспечения. Некоторые опции требуются для обеспечения соответствия вырабатываемого машинного кода своеобразной аппаратной конфигурации.

ОПЦИИ ДЛЯ ПЛАТФОРМ Alpha

Ниже перечислены опции, определенные для машин с процессорами DEC Alpha.

-malpha-as

Генерирует код, который далее будет асSEMBЛИРОВАТЬСЯ с использованием комплектного платформе асSEMBлера, предоставляемого поставщиком аппаратного обеспечения. Также см. опцию **-mgas**.

-mcix

Указывает, что компилятор должен вырабатывать код, использующий необязательный набор инструкций CIX. По умолчанию используется набор инструкций, задаваемый с помощью опции **-mcxi**. Использование набора инструкций CIX можно отключить опцией **-mno-cix**. Также см. опции **-mbwx**, **-mfix** и **-mmax**.

-mbuild-constants

Эта опция требует от компилятора формирования всех целочисленных констант, даже если для этого понадобится большее количество инструкций (обычно их максимальное количество равно шести).

В обычном режиме компилятор GCC оценивает 32- и 64-битные константы для определения возможности их формирования на основе констант меньшей разрядности с помощью двух или трех инструкций. Если такой возможности нет, то константы будут определены в виде литералов, а сгенерированный код будет загружать их из сегмента данных во время выполнения. Эта опция, как правило, используется для создания динамического загрузчика совместно используемой библиотеки. Со-вместно используемая библиотека для обнаружения переменных и констант в сегменте данных должна быть перемещаема в памяти.

-mbwx

Указывает, что компилятор должен генерировать код, использующий необязательный набор инструкций BWX. По умолчанию используется набор инструкций, задаваемый с помощью опции **-mcxi**. Использование набора инструкций BWX можно отключить опцией **-mno-bwx**. Также см. опции **-mcix**, **-mfix** и **-mmax**.

-mcxi=type

Устанавливает набор инструкций и параметры планирования инструкций, характерные для определенного типа процессора. Тип можно указать в формате EV или соответствующим номером чипа. Компилятор GCC поддерживает параметры планирования инструкций для семейств процессоров EV4, EV5 и EV6. При указании типа процессора будет применяться набор инструкций, используемый для указанного типа процессора. Если тип процессора не указан, то компилятор по умолчанию будет использовать тот тип, на котором компилятор был собран. Допустимые значения для параметра **type** приведены в таблице 21.1.

Таблица 21.1. Допустимые типы процессоров для платформ DEC Alpha

Тип	Описание
ev4, ev45, 21064	Параметры планировки выполнения соответствуют процессору ev4; расширенный набор инструкций не используется.

Тип	Описание
<code>ev5, 21164</code>	Параметры планировки соответствуют процессору <code>EV5</code> ; расширенные инструкции не используются.
<code>ev56, 21164a</code>	Параметры планировки соответствуют процессору <code>EV5</code> ; поддерживается расширенный набор инструкций <code>BWX</code> .
<code>pca56, 21164pc, 21164pc</code>	Параметры планировки выполнения соответствуют процессору <code>EV5</code> ; поддерживаются расширения <code>BWX</code> и <code>MAX</code> .
<code>ev6, 21264</code>	Параметры планировки выполнения соответствуют процессору <code>EV6</code> ; используются наборы инструкций <code>BWX</code> , <code>FIX</code> и <code>MAX</code> .
<code>ev67, 21264a</code>	Параметры планирования выполнения инструкций как для процессора <code>EV6</code> ; используются наборы инструкций <code>BWX</code> , <code>CIX</code> , <code>FIX</code> и <code>MAX</code> .

-mexplicit-relocs

Явным образом вырабатывает информацию о перемещении символов.

Прежние ассемблеры Alpha позволяли вырабатывать информацию о перемещении символов только с помощью макросов. Использование таких макросов делало невозможным оптимальную планировку выполнения инструкций. Пакет `binutils` GNU поддерживает новый синтаксис, дающий возможность явно указывать компилятору те перемещения, которые должны применяться к определенным инструкциям. Эта опцияенным образом используется для отладки, поскольку компилятор GCC определяет возможности ассемблера на этапе его сборки и установки в качестве применяемого по умолчанию. Генерацию таблиц перемещения символов можно подавить с помощью опции `-mno-explicit-relocs`.

Также см. опции `-msmall-data` и `-mlarge-data`.

-mfix

Указывает, что компилятор должен генерировать код, использующий необязательный набор инструкций `FIX`. По умолчанию используется набор инструкций, указываемый с помощью опции `-mcrci`. Использование набора инструкций `FIX` можно отключить опцией `-mno-fix`. Также см. опции `-mcix`, `-mbwx` и `-max`.

-mfloat-ieee

Генерирует код, который использует арифметические операции с плавающей точкой, соответствующие стандарту IEEE обычной и двойной точности, а не стандартов VAX F и G. Также см. опцию `-mfloat-vax`.

-mfloat-vax

Генерирует код, использующий арифметические операции с плавающей точкой, соответствующие стандарту VAX F и G, а не IEEE обычной и двойной точности. Так же см. опцию `-mfloat-ieee`.

-mfp-reg

Генерирует код, который использует набор особых регистров для работы с числами, имеющими формат представления с плавающей точкой. Эта опция установлена по умолчанию.

Опция **-mno-fp-reg**s отключает использование регистров для работы с числами с плавающей точкой и устанавливает опцию **-msoft-float**. Если набор регистров для работы с числами с плавающей точкой не используется, то операнды с плавающей точкой передаются в регистры, предназначенные для работы с целочисленными значениями. Так, как если бы они были целыми числами, а результаты вычислений возвращаются в формате \$0, а не \$f0. При этом применяется нестандартная последовательность вызова функций. Поэтому любая функция, которая принимает аргументы или возвращает значение с плавающей точкой, если она вызывается кодом, откомпилированным с установленной опцией **-mno-fp-reg**s, также должна компилироваться с установленной опцией **-mno-fp-reg**s.

Как правило, опция **-mno-fp-reg**s устанавливается при сборке ядра, которое не использует, а, следовательно, не сохраняет и не возвращает значения регистров с плавающей точкой.

-mfp-rounding-mode=mode

Устанавливает один из режимов округления в соответствии со стандартом IEEE. В других компиляторах для процессоров Alpha эта опция имеет вид **-fprm mode**. Допустимые значения параметра *mode* приведены в таблице 21.2.

Таблица 21.2. Режимы округления чисел с плавающей точкой для платформ DEC Alpha

Режим	Описание
p	Числа с плавающей точкой округляются до ближайшего машинного числа или до ближайшего четного числа, если округляемое число находится посередине между двумя машинными числами. Эта опция установлена по умолчанию.
m	Округление в сторону отрицательной бесконечности.
c	Округление отсечением. Числа с плавающей точкой округляются в сторону нуля.
d	Динамический режим округления. Применяемый режим округления определяется полем в регистре управления чисел с плавающей точкой (frcg). Библиотека языка C инициализирует этот регистр для округления чисел в сторону положительной бесконечности. Поэтому, если программа не изменяет содержимое регистра, то округление выполняется в сторону положительной бесконечности.

-mfp-trap-mode=mode

Эта опция управляет установкой прерываний, связанных с выполнением операций с числами с плавающей точкой. В других компиляторах для процессоров Alpha эта опция имеет вид **-fptm mode**. Допустимые значения параметра *mode* приведены в таблице 21.3.

Таблица 21.3. Параметры установки прерываний процессоров DEC Alpha

Режим	Описание
p	Допускаются только те прерывания, которые не могут быть отключены программно (например, аппаратное прерывание при делении на нуль). Эта опция установлена по умолчанию.
u	Помимо прерываний, разрешенных опцией p, допускаются прерывания при возникновении ошибки из-за потери значимости.

Режим	Описания
<code>vi</code>	Эти инструкции отмечены как безопасные для завершения выполнения программы.
<code>sui</code>	То же, что и <code>vi</code> , но допускаются неточные прерывания.

-mgas

Генерирует код, который в дальнейшем будет ассемблироваться ассемблером проекта GNU. Также см. опцию `-malpha-as`.

-mieee

Генерирует код, который полностью совместим с требованиями IEEE, за исключением поддержки флага `inexact-flag`. Архитектура аппаратного обеспечения для работы с числами с плавающей точкой процессора Alpha оптимизирована для максимальной производительности и, по большей части, соответствует стандартам IEEE для арифметических операций с плавающей точкой. Тем не менее, для обеспечения полного соответствия требуется некоторая программная поддержка.

При установленной опции `-mieee` при компиляции выполняется макрос препроцессора `_IEEE_FP`. В этом случае получаемый код будет менее эффективен, но будет поддерживать денормализованные числа и такие специальные значения стандарта IEEE, как `NaN` ("Not-A-Number" — "не число") и положительная и отрицательная бесконечность. В других компиляторах для процессоров Alpha эта опция имеет вид `-ieee_with_no_inexact`.

Также см. опцию `-mieee-with-inexact`.

-mieee-conformant

Генерирует код, полностью совместимый со стандартом IEEE.

Эта опция может использоваться только в случае, когда установлена опция `-mtrap-precision=i` и одна из опций `-mfp-trap-mode=su` или `-mfp-trap-mode=sui`.

Единственное назначение опции `-mieee-conformant` — вносить строку `.eflag48` в пролог (заголовок файла) генерируемого компилятором ассемблерного файла. В DEC UNIX это приводит к тому, что к программе будет подключена математическая библиотека, совместимая с IEEE.

-mieee-with-inexact

Действует аналогично опции `-mieee-conformant`, но за одним исключением: в генерируемом коде устанавливается флаг `inexact-flag`. Эта опция приводит к тому, что реализация математических операций в генерируемом коде полностью соответствует стандарту IEEE.

Кроме того, помимо макрона предпроцессора `_IEEE_FP`, выполняется и макрос `_IEEE_FP_EXACT`. В некоторых реализациях процессоров Alpha полученный при этом код может выполняться гораздо медленнее, чем код, генерируемый по умолчанию. Поскольку флаг `inexact-flag` используется лишь незначительной частью кода, опцию `-mieee-with-inexact` в большинстве случаев желательно не устанав-

ливать. В других компиляторах для процессоров Alpha эта опция имеет вид `-ieee_with_inexact`.

`-mlarge-data`

При установленной этой опции область данных ограничена объемом, немного меньшим 2-х Гбайт. Программы, которым при выполнении необходимо более 2 Гбайт памяти для размещения данных, должны использовать функцию `malloc()` или `mmap()` для выделения памяти под данные из "кучи" (heap) вместо использования сегмента данных программы. Эта опция установлена по умолчанию.

При генерации кода для совместно используемых библиотек опция `-fpic` устанавливает опцию `-msmall-data`, а опция `-fPIC` — `-mlarge-data`.

`-mmax`

Указывает, что компилятор должен генерировать код, использующий необязательный набор инструкций MAX. По умолчанию применяется набор инструкций, указываемый с помощью опции `-mcrci`. Использование набора инструкций FIX можно отключить опцией `-mno-fix`. Также см. опции `-mcix`, `-mbwx` и `-mfix`.

`-memory-latency=duration`

Устанавливает время ожидания, которое планировщик должен принимать при обращении к используемой приложением памяти. Время ожидания в значительной степени зависит от модели доступа к памяти, применяемой в приложении, и размежа внешней кэш-памяти компьютера.

Продолжительность может указываться в виде десятичного числа, которое соответствует количеству циклов процессора. Кроме того, ее можно задавать в формате L1, L2, L3 или main. Компилятор содержит оценки количества циклов процессора для стандартного аппаратного обеспечения типа EV4 и EV5 для кэшей уровней 1, 2 и 3 (которые также называют Dcache, Scache и Bcache соответственно), а также для основной памяти. Обратите внимание, что параметр L3 допустим только для типа EV5.

`-msmall-data`

При установленной опции `-mexplicit-relocs` доступ к статическим данным осуществляется с помощью перемещений `gp-relative`. Опция `-msmall-data` устанавливает, что объекты длиной 8 байт и менее должны помещаться в область данных малой длины (разделы `.sdata` и `.sbss`), а доступ к ним будет осуществляться с помощью 16-битных перемещений, основанных на регистре `$gp`. Это ограничивает объем области коротких данных 64 Кбайтами, но позволяет осуществлять прямой доступ к переменным одной инструкцией.

При генерации кода для совместно используемых библиотек опция `-fpic` устанавливает опцию `-msmall-data`, а опция `-fPIC` — `-mlarge-data`.

`-msoft-float`

Указывает, что компилятор не должен использовать аппаратные инструкции для выполнения операций над числами с плавающей точкой. Для выполнения таких действий будут применяться функции библиотеки `libgcc.a`.

При компиляции программы для выполнения на процессоре Alpha, не имеющем аппаратных инструкций операций с плавающей точкой, необходимо убедиться в наличии собранной библиотеки функций эмуляции таких операций. Следует учитывать, что современные реализации процессоров Alpha, не имеющие аппаратных инструкций операций с плавающей точкой, по-прежнему имеют регистры для хранения чисел с плавающей точкой.

По умолчанию действует опция **-mno-soft-float**.

-mtrap-precision=precision

Устанавливает точность, которая используется при определении положения кода, вызывающего прерывание для выполнения действий над числами с плавающей точкой.

В архитектуре процессоров Alpha прерывания для выполнения действий над числами с плавающей точкой не являются точными. Это означает, что без установленной программной поддержки прерывание обработать невозможно. Обычно вызов прерывания при отсутствии программной поддержки приводит к аварийному завершению программы. Компилятор генерирует код поддержки обработки прерываний операционной системы, определяющий точное место вызова прерывания в программе.

Параметры уровней точности вызова прерываний приведены в таблице 21.4.

Другие компиляторы для архитектуры Alpha имеют эквивалентные опции, **-scope_safe** и **-resumption-safe**.

Таблица 21.4. Параметры точности вызова прерываний для выполнения действий с плавающей точкой

Точность	Описание
p	Точность уровня программ. Эта опция означает, что обработчик прерываний может определять только программу, вызвавшую прерывание. Действует по умолчанию.
f	Точность уровня функций. Обработчик прерываний может определять функцию, вызвавшую прерывание.
i	Точность уровня инструкций. Обработчик прерываний может определять инструкцию, вызвавшую прерывание.

-mtune=type

Устанавливает параметры планирования инструкций, характерные только для данного типа аппаратного обеспечения. Допустимые значения параметра **type** приведены в таблице 21.1.

Опции для процессоров Alpha/VMS

Для процессоров DEC Alpha/VMS определена одна опция.

-mvms-return-codes

Устанавливает условие VMS, возвращаемое подпрограммой **main()**. По умолчанию возвращается условный код POSIX.

Опции для процессоров ARC

Для процессоров ARC определены следующие опции.

-mcpri=cpu

Генерирует код для указанного процессора ARC.

Поддержка различных вариантов процессоров зависит от конфигурации компилятора. Тем не менее, для всех процессоров ARC поддерживается опция **-mcpri=base**, которая применяется по умолчанию.

-mdata=section

Помещает данные в указанный раздел кода. Выбранный в этой опции раздел можно замещать в коде указанием значения атрибута **section**.

Также см. опции **-mtext** и **-mrodata**.

-mmangle-cpu

Присоединяет имя процессора ко всем открытым (public) именам программных символов.

В мультипроцессорных системах может существовать множество процессоров ARC с отличающимися наборами инструкций и регистров. Устанавливаемый этой опцией флаг предотвращает компоновку кодов, скомпилированных для разных процессоров. Никаких средств для одновременной поддержки различных, даже почти идентичных, процессоров не предусмотрено.

-mrodata=section

Помещает данные, доступные только для чтения, в указанный раздел кода. Выбранный в этой опции раздел можно изменить в коде значением атрибута **section**.

Также см. опции **-mtext** и **-mdata**.

-mtext=section

Помещает функции в указанный раздел кода. Выбранный в этой опции раздел можно изменить в коде присвоением другого значения атрибуту **section**.

Также см. опции **-mdata** и **-mrodata**.

-EL

Генерирует код для режима работы с прямым порядком байтов (little endian). Эта опция установлена по умолчанию.

-EB

Генерирует код для режима работы с обратным порядком байтов (big endian).

Опции для процессоров ARM

Ниже перечислены опции, определенные для архитектур ARM (Advanced RISC Machines).

-mabort-on-noreturn

В конце функции `noreturn` генерирует вызов функции `abort()`. Вызов располагается таким образом, что он будет выполнен при попытке выхода из функции.

-malignment-traps

Генерирует код, который не будет вызывать прерываний, даже если прерывания разрешены определением выравнивания MMU. Вырабатываемый код представляет собой последовательность обращений к байтам вместо прямого доступа к полусловам. Эта опция игнорируется при компиляции кода для архитектуры ARM версии 4 или выше, поскольку такие процессоры позволяют осуществлять прямой доступ к полусловам.

Установленная опция `-mno-alignment-traps` приводит к генерации кода, который предполагает, что блок управления памятью MMU не будет вызывать прерывания при обращениях с нарушением выравнивания. Опция формирует более эффективный код в случае, когда предназначаемый процессор не имеет набора операций с полусловами. Обратите внимание, что эту опцию нельзя использовать для доступа к объектам, не имеющим выравнивания к границе слов, поскольку процессор в этом случае считает из памяти только один объект длиной 32 бита, выровненный по границе слова.

По умолчанию применяется опция `-mno-alignment-traps`, которая приводит к генерации более эффективного кода в случае недоступности аппаратных инструкций для работы с полусловами.

В архитектурах ARM до версии ARMv4 не было предусмотрено инструкций для доступа объектам в памяти длиной в половину машинного слова. Тем не менее, архитектура ARM предусматривает возможность считывания из памяти слова без обязательного применения выравнивания адреса, и при этом ядро процессора способно обрабатывать такие данные при их загрузке. Опция `-malignment-traps` сообщает компилятору, что обращение с нарушением выравнивания вызовет прерывание блока управления памятью MMU, и что вместо этого он должен генерировать последовательность инструкций для доступа к байтам памяти. Компилятор способен использовать доступ по словам для загрузки данных длиной в полуслово, если ему известно, что применяется выравнивание адреса к границе машинного слова.

-mapcs

Идентична опции `-mapcs-frame`.

-mapcs-26

Генерирует код для процессора с 26-битным счетчиком команд, соответствующий стандарту вызова функций с 26-битной адресацией архитектуры ARM (ARM Procedure Call Standard).

Опция `-mapcs-26` заменила опции `-m2` и `-m3`, которые существовали в прежних версиях компилятора.

-mapcs-32

Генерирует код для процессора с 32-битным счетчиком команд, соответствующий стандарту вызова функций с 32-битной адресацией архитектуры ARM.

Опция **-mapcs-32** заменила опцию **-m6**, которая существовала в более ранних версиях компилятора.

-mapcs-frame

Генерирует кадр стека (frame stack), совместимый со стандартом вызова процедур архитектуры ARM, для всех функций, даже если соблюдение этого условия не обязательно для правильного выполнения кода. По умолчанию используется опция **-mno-apcs-frame**.

При одновременном указании опций **-mapcs-frame** и **-fomit-frame-pointer** кадры стека для вложенных функций формироваться не будут.

-march=name

Эта опция указывает имя конкретного процессора архитектуры ARM, для которого предназначена компилируемая программа. Компилятор использует имя для определения типа инструкций, которые можно генерировать при формировании откомпилированного кода. Параметр *name* может принимать следующие значения: **armv2**, **armv2a**, **armv3**, **>armv3m**, **armv4**, **armv4t**, **armv5**, **armv5t** и **armv5te**.

-mbig-endian

Генерирует код для процессора, работающего в режиме обратного порядка байтов (big endian). По умолчанию код генерируется для процессора, работающего с прямым порядком байтов. Также см. опцию **-mlittle-endian**.

-mcallee-super-interworking

Присваивает всем функциям из компилируемого файла, видимым извне, заголовок набора инструкций ARM, который переводит процессор в режим Thumb перед выполнением остальных функций. Такая конфигурация позволяет вызывать функции из кода, не поддерживающего обмен между программами, noninterworking code.

-mcaller-super-interworking

Позволяет корректно вызывать функции (включая виртуальные) посредством указателей на функции независимо от того, был ли код откомпилирован для работы в режиме межпрограммного обмена. При компиляции кода с установленной этой опцией код будет содержать дополнительные служебные данные небольшого размера.

-mbsd

Эта опция может использоваться только для процессоров RISC iX для включения эмуляции режима BSD, основного для компилятора. Опция установлена по умолчанию, если не указана опция **-ansi**. Также см. опцию **-mchorp**.

-mcpu=*name*

Эта опция указывает тип процессора ARM, для которого компилируется данная программа. Компилятор использует имя процессора для определения типа генерируемых инструкций при формировании им ассемблерного кода. Параметр *name* может иметь следующие значения: `arm2`, `arm250`, `arm3`, `arm6`, `arm60`, `arm600`, `arm610`, `arm620`, `arm7`, `arm7m`, `arm7d`, `arm7dm`, `arm7di`, `arm7dmi`, `arm70`, `arm700`, `arm700i`, `arm710`, `arm710c`, `arm7100`, `arm7500`, `arm7500fe`, `arm7tdmi`, `arm8`, `strongarm`, `strongarm110`, `strongarm1100`, `arm8`, `arm810`, `arm9`, `arm9e`, `arm920`, `arm920t`, `arm940t`, `arm9tdmi`, `arm190tdmi`, `arm1020t` и `xscale`.

Также см. опцию `-mtune`.

-mfpr=*number*

Идентична опции `-mfpe=number`.

-mfpe=*number*

Эта опция указывает версию эмуляции операций над числами с плавающей точкой, доступной на пред назначаемой платформе (target). Параметр *number* может иметь значения 2 и 3.

-mhard-float

Генерирует код, содержащий операции над числами с плавающей точкой. Эта опция установлена по умолчанию. Также см. опцию `-msoft-float`.

-mlittle-endian

Генерирует код для процессора, работающего в режиме прямого порядка байтов. Эта опция установлена по умолчанию для всех стандартных конфигураций. Также см. опцию `-mbig-endian`.

-mlong-calls

Эта опция приводит к генерации кода, который выполняет вызовы функций путем загрузки их адреса в регистр с последующей передачей управления функции на основании содержащегося в регистре адреса.

По умолчанию используется схема вызова `-mno-long-calls`, при которой вызовы функций помещаются в область действия директивы `#pragma long_calls_off`.

Опция `-mlong-calls` не влияет на способ выработки кода вызова функций с помощью указателей.

Эта опция необходима в случае, если адрес вызываемой функции лежит за пределами области непосредственной (короткой) адресации размером 64 Мбайт. Даже при установленной этой опции некоторые функции не будут вызываться с помощью алгоритма длинного вызова. В качестве примера можно привести статические функции, функции с атрибутом `short-call`, функции, находящиеся в пределах области видимости директивы `#pragma no_long_calls`, и функции, описания которых уже скомпилированы в текущем модуле компиляции. С другой стороны, функции с замещаемыми определениями (`weak definitions`), функции имеющие атрибут

long-calls, функции, находящиеся в разделе с атрибутом **long-calls**, и функции, находящиеся в пределах области действия директивы **#pragma long_calls**, всегда вызываются с помощью алгоритма длинного вызова.

-mno-pfun-dllimport

Исключает поддержку атрибута **dllimport**.

-mpic-register=reg

Указывает регистр, который будет использоваться для перемещаемой, позиционно-независимой (PIC) адресации. По умолчанию используется регистр R10, но если разрешена обратная трассировка стека (backtrace), то используется регистр R9. Также см. опцию **-msingle-pic-base**.

-mpoke-function-name

Записывает имена всех функций в текстовый раздел непосредственно перед кодом пролога функции. Генерируемый код будет выглядеть примерно следующим образом:

```
.ascii "arm_poke_function_name", 0
.align
t1
.word 0xff000000 + (t1-t0)
arm_poke_function_name
    mov ip, sp
    stmfd sp!, {fp, ip, lr, pc}
    sub fp, ip, #4
```

При обратной трассировке стека код может проверять счетчик команд по адресу **fp+0**. Далее, если функция трассировки обнаруживает, что старшие восемь бит по адресу **pc-12** установлены, то это означает, что имя функции находится непосредственно перед этим адресом. Длина имени равна **((pc[-3]) & 0xff000000)**.

-msched-prolog

Эта опция установлена по умолчанию. Установка опции **-mno-sched-prolog** предотвращает переупорядочение инструкций в прологе функции и объединение их с инструкциями, находящимися в теле функции.

В коде, генерируемом при установленной опции **-mno-sched-prolog**, все функции начинаются с более легко распознаваемого набора инструкций. Этот факт может использоваться для обнаружения начала функции внутри выполняемого кода.

-mshort-load-bytes

Устаревшая форма опции **-malignment-traps**.

-mshort-load-words

Это — устаревшая форма опции **-malignment-traps**. Она распознается, но не поддерживается.

-msingle-pic-base

Приводит к тому, что регистр, используемый для перемещаемой, позиционно-независимой (PIC) адресации, становится доступным только для чтения и не загружается в пролог функций. Инициализация этого регистра производится системой загрузки перед началом выполнения программы. Также см. опцию **-mpic-register**.

-msoft-float

Генерирует код, который для выполнения операций с плавающей точкой выполняет вызовы функций эмуляции из библиотеки. По умолчанию установлена опция **-mhard-float**.

Следует иметь в виду, что необходимые библиотеки имеются не для всех процессоров ARM. Как правило, для этого используются библиотеки языка C, но подключить их непосредственно в процессе кросс-компиляции невозможно. Если эта опция установлена, то необходимо подключение внешних библиотек.

Опция **-msoft-float** изменяет соглашение о вызовах функций, используемое в коде. Поэтому ее удобнее применять только при компиляции всей программы. Чтобы программа работала, с этой же опцией необходимо откомпилировать и библиотеку **libgcc.a** (эта библиотека входит в состав компилятора GCC).

-mstructure-size-boundary=number

Размер всех структур и объединений (unions) будет округляться до числа, кратного количеству бит, установленного с помощью этой опции. Для параметра *number* допускаются значения 8 и 32. Указание большего числа может привести к генерации более быстрого и эффективного кода, но и его размер будет больше.

Значение, установленное по умолчанию, зависит от формата объектных файлов. Для формата COFF по умолчанию используется значение 8.

Два допустимых значения опции потенциально несовместимы. Код, откомпилированный с одним значением, может не работать с кодом или библиотеками, откомпилированными с другим значением, если они передают или принимают данные с помощью структур или объединений.

-msymrename

Применяется по умолчанию. Как и опцию **-mno-symrename**, ее можно использовать только для процессоров с архитектурой RISC iX для подавления постпроцессора ассемблера **symrename**, обычно запускаемого после ассемблирования кода. Как правило, это бывает необходимо для изменения отдельных стандартных символов при подготовке к компоновке с библиотеками C для архитектуры RISC iX. Указание опции **-mno-symrename** также исключает применение постпроцессора.

Постпроцессор никогда не используется компилятором, настроенным для кросс-компиляции.

-mthumb

Генерирует код для 16-битного набора инструкций Thumb. По умолчанию используется 32-битный набор инструкций ARM.

-mthumb-interwork

Генерирует код, который позволяет осуществлять вызовы между наборами инструкций ARM и Thumb. Если эта опция не установлена, наборы ARM и Thumb нельзя с уверенностью использовать в одной программе.

Опция **-mthumb-interwork** приводит к генерации кода несколько большего объема, поэтому по умолчанию применяется опция **-mno-thumb-interwork**.

-mtpcs-frame

Формирует кадр стека (stack frame), соответствующий стандарту вызова процедур Thumb (Thumb Procedure Call Standard), для всех функций, имеющих вызовы вложенных функций (non-leaf functions). По умолчанию установлена опция **-mno-tpcs-frame**. Также см. опцию **-mtpcs-leaf-frame**.

-mtpcs-leaf-frame

Формирует стековый фрейм, соответствующий стандарту вызова процедур Thumb (Thumb Procedure Call Standard), для всех функций, имеющих вложенные вызовы (non-leaf functions). По умолчанию установлена опция **-mno-apcs-leaf-frame**. Также см. опцию **-mtpcs-frame**.

-mtune=name

Эта опция очень похожа на **-mcpu**. Только вместо указания конкретного типа процессора и ограничения набора используемых инструкций она указывает компилятору GCC оптимизировать производительность кода для указанного типа процессора. При этом генерируемые инструкции выбираются из набора инструкций процессора, указанного в опции **-mcpu**.

Допустимые значения для параметра **name** идентичны приведенным для опции **-mcpu**. Для некоторых реализаций машин ARM оптимальная производительность кода достигается только при использовании этой опции.

-mwords-little-endian

Генерирует код с прямым порядком машинных слов, но обратным порядком представления байтов.

Эта опция должна использоваться только при необходимости обеспечения совместимости скомпилированного кода с процессорами ARM, использующих обратный порядок байтов, когда этот код компилировался в GCC версии 2.8 или старше.

Опция **-mwords-little-endian** применяется только при генерации кода для процессоров ARM, работающих с обратным порядком байтов (имеется в виду порядок байтов типа "32107654").

-mxopen

Эта опция может использоваться только для процессоров с архитектурой RISC iX для эмуляции режима X/Open.

Опции для платформы AVR

Ниже перечислены опции, определенные для реализаций AVR.

-mcall-prologues

Уменьшает размер генерируемого кода за счет формирования прологов и эпилогов функций в виде соответствующих им подпрограмм.

-minit-stack=*address*

Указывает начальный адрес стека. Адрес может быть указан как символом, так и числовым значением. По умолчанию используется значение `__stack`.

-mmcu=*setting*

Значение *setting* устанавливает либо набор инструкций ATMEL AVR, либо тип MCU. Допустимые имена для параметра *setting* перечислены в таблице 21.5. В первом столбце приведены имена инструкций набора AVR, а во второй — соответствующие им типы MCU. В качестве значения *setting* могут использоваться имена как из первого, так и из второго столбца.

Таблица 21.5. Параметры AVR и MCU.

AVR	MCU	Описание
<code>avr1</code>	<code>at90s1200, attiny10, attiny11, attiny12, attiny15, attiny28</code>	Минимальный набор инструкций ядра AVR, не поддерживаемый компилятором C. Эти параметры используются только для программ на языке ассемблера.
<code>avr2</code>	<code>at90s2313, at90s2323, attiny22, at90s2333, at90s2343, at90s4414, at90s4433, at90s4434, at90s8515, at90s8534, at90s8535</code>	Набор инструкций <code>avr2</code> используется по умолчанию. Это классическое ядро AVR с размером используемой программной области памяти до 8 Кбайт.
<code>avr3</code>	<code>atmega103, atmega603, at43usb320, at76c711</code>	Классическое ядро AVR с размером используемой программной области памяти до 128 Кбайт.
<code>avr4</code>	<code>atmega8, atmega83, atmega85</code>	Расширенное ядро AVR с размером используемой программной области памяти до 8 Кбайт.
<code>avr5</code>	<code>atmega16, atmega161, atmega163, atmega32, atmega323, atmega64, atmega128, at43usb355, at94k</code>	Расширенное ядро AVR с размером используемой программной области памяти до 128 Кбайт.

-mno-interrupts

Уменьшает размер кода за счет генерации кода, несовместимого с аппаратными прерываниями.

-mno-tablejump

Уменьшает размер кода за счет генерации кода без инструкций табличных переходов, которые в некоторых случаях могут увеличивать размер кода.

-msize

Записывает информацию о размере инструкций в файл на языке ассемблера.

-mtiny-stack

Генерирует код, который изменяет только младшие восемь бит указателя стека.

Опции для платформ CRIS

Ниже перечислены опции, определенные для портов CRIS.

-m8-bit

Применяет 8-битное выравнивание адреса кадра стека (stack frame), записывающих данных и констант.

Также см. опции **-mdata-align**, **-mconst-align**, **-mstack-align**, **-m32-bit** и **-m16-bit**.

-m16-bit

Применяет 16-битное выравнивание к кадру стека (stack frame), записываемым данным и константам.

Также см. опции **-mdata-align**, **-mconst-align**, **-mstack-align**, **-m32-bit** и **-m8-bit**.

-m32-bit

Применяет 32-битное выравнивание кадра стека (stack frame), записываемых данных и констант.

Также см. опции **-mdata-align**, **-mconst-align**, **-mstack-align**, **-m16-bit** и **-m8-bit**.

-maout

Унаследованная от предыдущих версий компилятора форма опции **no-op**, поддерживается только для систем **cris-axis-aout**.

-march=architecture

Идентична опции **-mcphi**.

-mcc-init

Подавляет использование результатов вычисления условий, полученных предыдущими инструкциями, и всегда перед использованием кода, выполняемого по условию, генерирует инструкции проверки и сравнения.

-mconst-align

Выравнивает константы по границе максимального для выбранной модели процессора размера блока данных, выбираемого за одно обращение. По умолчанию применяется опция **-mno-const-align**, которая устанавливает 32-битное выравнива-

ние констант. Это не оказывает влияния на машинный интерфейс прикладных программ (ABI), в частности, на схему размещения структур данных.

Также см. опции **-mdata-align**, **-mstack-align**, **-m32-bit**, **-m16-bit** и **-m8-bit**.

-mcphi=architecture

Генерирует код для указанной архитектуры. Допускаются следующие значения параметра **architecture**: **v3** для ETRAX4, **v8** для ETRAX100 и **v10** для ETRAX100LX. По умолчанию используется значение **v0** (за исключением конфигураций **crix-axis-linux-gnu**, для них по умолчанию применяется значение **v10**).

-mdata-align

Выравнивает отдельные элементы данных по границе максимального для выбранной модели процессора размера блока данных, выбираемого за одно обращение к памяти. По умолчанию установлена опция **-mno-data-align**, которая применяет выравнивание данных по границе 32 бит. Это не оказывает влияния на машинный интерфейс прикладных программ (ABI), в частности, на схему размещения структур.

Также см. опции **-mstack-align**, **-mconst-align**, **-m32-bit**, **-m16-bit** и **-m8-bit**.

-melf

Унаследованная от предыдущих версий компилятора форма опции **по-ор**. Поддерживается только для конфигураций **cris-elf** и **crix-axis-linux-gnu**.

-melinux

Выбирает библиотеки, включаемые файлы GNU/Linux и набор инструкций **-mcphi=v8**. Эта опция действительна только для систем **cris-axis-aout**.

-melinux-stacksize=number

Устанавливает поля индикаторов программы таким образом, чтобы они сообщали загрузчику ядра о необходимости установки размера стека программы, равным **number** байт. Эта опция действительна только для конфигураций **cris-axis-aout**.

-metrax100

Идентична опции **-mcphi=v8**.

-metrax4

Идентична опции **-mcphi=v3**.

-mgotplt

Совместно с опцией **-fpic** или **-fPIC** приводит к генерации последовательностей инструкций, которые загружают адреса функций в процессор из раздела PLT аппаратной структуры GOT вместо вызова PLT. Это положение применяется по умолчанию, его можно отменить опцией **-mno-gotplt**.

-mlinux

Унаследованная от предыдущих версий компилятора форма опции `no-op`. Поддерживается только системами `crix-axis-linux-gnu`.

-mmax-stack-frame=*number*

Выдает предупреждение, когда размер кадра стека функции превышает указанное в поле *number* число байт.

-mno-side-effects

Подавляет генерацию инструкций, которые могут оказывать побочное действие при режимах адресации, отличных от метода постинкрементной адресации (postincrement).

-mpdebug

Разрешает включение в ассемблерный код специфической расширенной отладочной информации, характерной для систем CRIS. Кроме того, эта опция отключает индикатор форматированного кода `#NO_APP` в начале файла ассемблерного кода.

-mprologue-epilogue

Эта опция действует по умолчанию. Она используется для генерации компилятором кодов пролога и эпилога, которые формируют кадр стека вызова функций.

Опция `-mno-prologue-epilogue` подавляет генерацию нормальных прологов и эпилогов функций. В коде не генерируются ни инструкции, ни последовательности возврата. При использовании этой опции следует производить визуальный контроль скомпилированного кода. Поскольку при этом не выдаются предупреждения и сообщения об ошибках в случае необходимости сохранения регистров или выделения памяти для размещения локальных переменных.

-mstack-align

Выравнивает кадр стека вызова к размеру наибольшего блока данных, считываемых предназначаемым типом процессора за одно обращение. По умолчанию установлена опция `-mno-stack-align`, которая также применяет 32-битное выравнивание констант. Это не оказывает влияния на машинный интерфейс прикладных программ (ABI), в частности, на размещение структур.

Также см. опции `-mdata-align`, `-mconst-align`, `-m32-bit`, `-m16-bit` и `-m8-bit`.

-mtune=*architecture*

Оптимизирует все параметры генерируемого кода, за исключением машинного интерфейса прикладных программ (ABI) и используемого набора инструкций, для указанной в опции архитектуры. Допустимые значения для параметра *architecture* идентичны значениям параметра для опции `-mscri`.

-sim

Компонует программу с входными и выходными функциями библиотеки, моделирующей среду выполнения (*simulator library*). Память для кода, инициализированных данных и данных, инициализированных нулями, выделяется последовательно. Эта опция поддерживается только системами `cris-axis-aout` и `cris-axis-elf`. Также см. опцию `-sim2`.

-sim2

Аналогична опции `-sim`, но дополнительно передает компоновщику опции для размещения инициализированных данных, начиная с адреса `0x40000000`, а инициализированных нулями данных — с адреса `0x80000000`.

ОПЦИИ ДЛЯ ПЛАТФОРМЫ D30V

Ниже перечислены опции, пределенные для реализаций процессоров D30V.

-masm-optimize

Позволяет передавать ассемблеру опцию `-O` при оптимизации. Опция `-O` используется ассемблером для "запараллеливания" соседних инструкций, где это возможно. Опция `-masm-optimize` применяется по умолчанию, ее можно отключить опцией `-no-asm-optimize`.

-mbranch-cost=*number*

Увеличивает внутреннюю цену ветвлений, равную числу *number*. Большая цена означает, что компилятор во избежание ветвления будет генерировать большее количество инструкций. По умолчанию установлено значение 2.

Также см. опцию `-mcond-exec`.

-mcond-exec=*number*

Устанавливает максимальное количество условно выполняемых инструкций, генерируемых вместо ветвления. По умолчанию установлено значение 4.

Также см. опцию `-mbranch-cost`.

-mextmem

Компонует разделы `.text`, `.data`, `.bss`, `.strings`, `.rodata`, `.rodata1` и `.data1` во внешнюю память, которая начинается с адреса `0x80000000`.

-mextmemory

Идентична опции `-mextmem`.

-monchip

Компонует раздел `.text` в текстовый раздел внутренней области памяти (on-chip memory), начиная с адреса `0x40000000`. Кроме того, разделы `.data`, `.bss`, `.strings`, `.rodata`, `.rodata1` и `.data1` компонуются в раздел данных внутренней области памяти (on-chip memory), начиная с адреса `0x20000000`.

Опции для поддержки платформы Н8/300

Ниже перечислены опции, определенные для реализаций Н8/300.

-malign-300

При установке этой опции на платформах Н8/300Н и Н8/S будут использоваться такие же правила выравнивания, как и для платформы Н8/300.

По умолчанию выравнивание данных типа `long` и `float` на платформах Н8/300Н и Н8/S производится по границе 4 байтов. Установка опции `-malign-300` приводит к тому, что данные выравниваются по границе 2 байта.

Опция не влияет на параметры платформы Н8/300.

-mh

Генерирует код для платформы Н8/300Н.

-mint32

Генерирует данные типа `int` как 32-битные значения.

-mrelax

При наличии возможности во время компоновки сокращает отдельные адресные ссылки. Эта опция устанавливает опцию компоновщика `-relax`.

-ms

Генерирует код для платформы Н8/S.

-ms2600

Генерирует код для платформы Н8/S2600. Эта опция должна использоваться совместно с опцией `-ms`.

Опции для платформы НРРА

Ниже перечислены опции, определенные для реализаций Н8/300.

-march=architecture

Генерирует код для указанной архитектуры. Для параметра `architecture` допустимы следующие значения: `1.0` (для РА 1.0), `1.1` (для РА 1.1) и `2.0` (для РА 2.0).

Код, откомпилированный для более ранних архитектур, будет выполняться и на более поздних архитектурах, но не наоборот. Для определения требуемой архитектуры для конкретного компьютера изучите файл `/usr/lib/sched.models` в системе НРUX.

-mbig-switch

Генерирует код, совместимый с большими таблицами переключений. Эта опция используется только в случае, когда ассемблер или компоновщик выдает предупреждение о наличии в таблице переключений ветвлений, адреса которых выходят за пределы допустимого диапазона.

-mdisable-fpregs

Запрещает любое использование специальных регистров для хранения чисел с плавающей точкой.

Эта опция необходима для компиляции ядра, которое выполняет "ленивое" контекстное переключение регистров для чисел с плавающей точкой. Если установить опцию **-mdisable-fpregs** и попытаться выполнить операцию с числами с плавающей точкой, то компилятор выдаст ошибку.

-mdisable-indexing

Запрещает компилятору использовать режимы индексации адресов. Это позволяет избежать некоторых проблем, связанных с компиляцией в системе MACH скептирированного кода формата MIG.

-mfast-indirect-calls

Генерирует код, который предполагает, что вызовы не выходят за пределы допустимых областей.

В этом случае непрямые вызовы в коде, вырабатываемом компилятором, будут выполняться быстрее. Опция **-mfast-indirect-calls** не будет работать при использовании разделяемых библиотек или вложенных функций.

-mgas

Разрешает использовать только те директивы ассемблера, которые совместимы с ассемблером **gas**.

-mjump-in-delay

Заполняет отложенные вызовы функций инструкциями безусловного перехода. При этом изменяется указатель возврата вызова этой функции, он становится адресом назначения условного перехода.

-mlinker-opt

Разрешает оптимизационный проход для компоновщика HPUX. Эта опция исключает возможность использования отладки символов. Кроме того, она вызывает сбой в компоновщиках HPUX 8 и HPUX 9, в результате которого компоновщики выдают неправильные сообщения об ошибках при компоновке некоторых программ.

-mlong-load-store

Генерирует последовательности загрузки и записи, состоящие из трех инструкций. В некоторых случаях это требуется для компоновщика HPUX 10. Эта опция идентична опции **+k**, которая применяется в компиляторах HP.

-mno-space-reg

Генерирует код, в котором предполагается, что платформа, для которой выполняется компиляция, не имеет разделяющих регистров (space registers).

Это позволяет компилятору быстрее выполнять непрямые вызовы и использовать режимы не масштабированной индексной адресации. Такой код применяется для систем РА уровня "0" и системных ядер.

-mpa-risc-1-0

Идентична опции **-march=1.0**.

-mpa-risc-1-1

Идентична опции **-march=1.1**.

-mpa-risc-2-0

Идентична опции **-march=2.0**.

-importable-runtime

Использует соглашение о вызовах, предложенное НР для обеспечения переносимости кода на системы ELF.

-mschedule=type

Применяет планировку кода в соответствии с ограничениями указанного типа машины. Для параметра **type** допустимы следующие значения: 700, 7100, 7100LC, 7200, 7300 и 8000. Для определения опции планирования, требуемой для конкретного компьютера, изучите файл системы НРУХ **/usr/lib/sched.models**. По умолчанию используется значение 8000.

-msoft-float

Генерирует код, содержащий вызовы функций библиотеки эмуляции операций над числами с плавающей точкой.

Для некоторых платформ НРРА такие библиотеки недоступны. В обычных условиях используются средства компилятора C, но их невозможно подключить непосредственно при кросс-компиляции. Поэтому при выполнении кросс-компиляции необходимо точно указать соответствующие функции библиотек. Конфигурация компилятора **hppa1.1-*-pro** обеспечивает полную программную поддержку операций с плавающей точкой.

Опция **-msoft-float** изменяет соглашение о вызовах для объектных файлов, поэтому она должна использоваться только в случае компиляции всех модулей программы. Используемые программой библиотеки также следует скомпилировать с указанием этой опции. В этом случае с опцией **-msoft-float** должна быть скомпилирована библиотека **libgcc.a**, которая входит в состав компилятора GCC.

Опции компилятора для платформы IA-64

Ниже перечислены опции, определенные для архитектуры Intel IA-64.

-mauto-pic

Генерирует самоперемещающийся код, т.е. код с автоматически перемещаемой адресацией (self-relocatable code). Кроме того, применение этой опции устанавливает

ет опцию **-mconstant-gp**. Используется при компиляции кода для жестко прошиваемых программ (программно-аппаратных средств, firmware).

-mb-step

Генерирует код, который обходит известные ошибки аппаратуры Itanium B.

-mbig-endian

Генерирует код для архитектуры, работающей с обратным порядком байтов. Эта опция по умолчанию применяется для системы HPUX.

Также см. опцию **-mlittle-endian**.

-mconstant-gp

Генерирует код, который использует единственное постоянное значение регистра глобального указателя. Опция используется при компиляции ядра.

Также см. опцию **-mauto-pic**.

-mdwarf2-asn

Генерирует код ассемблера для построчной отладки в формате DWARF2. Эта опция может быть очень полезной, кода не используется ассемблер GNU. Отключить действие опции **-mdwarf2-asn** можно с помощью опции **-mno-dwarf2-asn**.

-mfixed-range=range

Генерирует код, который работает с указанным в поле *range* диапазоном регистров, как с фиксированными регистрами. Диапазон регистров задается адресами двух регистров, начального и конечного, с дефисом между ними. Кроме того, можно указывать одновременно несколько диапазонов регистров, разделяя их запятыми.

Фиксированными регистрами считаются те, которые не могут использоваться блоком распределения регистров. Опция **-mfixed-range** используется при компиляции кода ядра.

-mgnu-as

По этой опции компилятор вырабатывает код, предназначенный для ассемблера GNU. Опция **-mgnu-as** установлена по умолчанию. Ее действие можно отменить применением опции **-mno-gnu-as**.

-mgnu-ld

Генерирует код, предназначенный для компоновки программой-компоновщиком GNU. Опция **-mgnu-ld** установлена по умолчанию. Ее действие можно отменить опцией **-mno-gnu-ld**.

-minline-divide-max-throughput

По этой опции компилятор вырабатывает код для выполнения раздельной подстановки (inline divides) с использованием алгоритма максимальной пропускной способности. Также см. опцию **-minline-divide-min-latency**.

-minline-divide-min-latency

Генерирует код для выполнения раздельной подстановки (*inline divides*) с использованием алгоритма минимальной задержки. Также см. опцию **-minline-divide-max-throughput**.

-mlittle-endian

По этой опции компилятор генерирует код для архитектуры, работающей с прямым порядком представления байтов. Эта опция используется для систем AIX5 и Linux по умолчанию.

Также см. опцию **-mbig-endian**.

-mno-pic

Генерирует код, который для адресации не использует регистр глобального указателя (global pointer register). В результате этого вырабатывается код, который является зависимым от положения загрузки и не соответствует требованиям машинного интерфейса прикладных программ (ABI) для архитектуры IA-64.

-mregister-names

Генерирует имена **in**, **loc** и **out** для стековых регистров. Это позволяет сделать ассемблерный код более читаемым. Действие опции можно обратить применением опции **-mno-register-names**.

-msdata

Разрешает выполнение оптимизаций, которые используют раздел данных малой длины. Эта опция применяется по умолчанию. Ее действие можно отменить опцией **-mno-sdata**. Опция **-msdata** может быть удобной для обхода возможных ошибок оптимизатора.

-mvolatile-asn-stop

Генерирует стоповый бит непосредственно перед и сразу после изменяемых (*volatile*) операторов ассемблера. Действие этой опции можно отменить применением опции **-mno-volatile-asn-stop**.

Опции для платформ Intel 386 и AMD x86-64

Ниже перечислены опции, определенные для семейства компьютеров i386 и x86-64.

-m128bit-long-double

Устанавливает размер данных типа **long double** равным 128 бит (16 байт). Машинный интерфейс прикладных программ (Embedded Applications Binary Interface, EABI) для архитектур i386 устанавливает размер данных типа **long double** равным 12 байт, в то время как для новых архитектур (Pentium и выше) данные типа **long double** выравниваются по границе 8 или 16 байт. Очевидно, что соблюдение такого выравнивания невозможно при осуществлении доступа к 12-байтным данным, которые находятся в массиве.

Применение опции **-m128bit-long-double** изменяет размер структур и массивов, содержащих данные типа **long double**. Кроме того, будет изменено соглашение о вызове функций для тех функций, которые используют тип **long double**.

Также см. опцию **-m96bit-long-double**.

-m32

Для процессоров AMD x86-64 в 64-битной среде окружения эта опция устанавливает длину данных типа **int**, **long** и указателей равной 32 битам и генерирует код, совместимый с любой платформой i386.

-m386

Идентична опции **-mcpri=i386**. Эта форма опции считается устаревшей, она может не поддерживаться следующими версиями компилятора.

-m3dnow

Разрешает использование встроенных функций, позволяющих осуществлять прямой доступ к расширениям 3Dnow. Действие этой опции можно отключить применением опции **-mno-3dnow**.

-m486

Идентична опции **-mcpri=i486**. Этот формат опции считается устаревшим.

-m64

Для процессоров AMD x86-64 в 64-битной среде эта опция устанавливает длину данных типа **int** равной 32 битам, а данных типа **long** и указателей — 64 битам. По этой опции вырабатывается код специально для процессора AMD x86-64.

-m96bit-long-double

Указывает, что длина данных типа **long double** должна составлять 96 бит (12 байт), как того требует машинный интерфейс прикладных программ (Embedded Applications Binary Interface, EABI) процессоров семейства i386. Опция **-m96bit-long-double** применяется по умолчанию.

Также см. опцию **-m128bit-long-double**.

-maccumulate-outgoing-args

Указывает, что максимальный объем, занимаемый выходными аргументами, будет вычисляться в прологе функции. Такая схема работает быстрее на современных процессорах благодаря меньшей зависимости, улучшенному планированию и менее интенсивному использованию стека в случае, когда предпочтительное выравнивание границы стека не равно 2. Недостатком схемы является увеличение объема кода. Опция **-maccumulate-outgoing-args** автоматически устанавливает опцию **-mno-push-args**.

-malign-double

Указывает компилятору, что переменные типов `double`, `long double` и `long long` должны иметь выравнивание к границе двух машинных слов. А вот опция `-mno-align-double` применяет выравнивание переменных к границе одного машинного слова.

Выравнивание переменных типа `double` к границе двух машинных слов позволяет за счет увеличения размера программы получить код, выполняемый на процессорах Pentium несколько быстрее.

Опция `-malign-double` приводит к тому, что структуры с данными типов `double`, `long double` и `long long` выравниваются иначе, чем это предусмотрено машинным интерфейсом прикладных программ (Embedded Applications Binary Interface, EABI) процессоров семейства i386.

-march=architecture

Генерирует инструкции для указанной архитектуры. Допустимые значения для параметра `architecture` идентичны значениям, используемым для параметра `type` опции `-mcphi`. Установка опции `-march` аналогична применению опции `-mcphi` с указанием соответствующего типа процессора.

-masm=dialect

Генерирует инструкции ассемблера, используя указанный диалект. Параметр `dialect` может принимать следующие значения: `intel` и `att`. По умолчанию используется значение `att`.

-mcphi=type

Настраивает все параметры генерируемого кода, кроме машинного интерфейса прикладных программ (EABI), на полное соответствие указанному типу процессора и выбирает соответствующий набор инструкций. Параметр `type` может принимать следующие значения: `i386`, `i486`, `i586`, `i686`, `pentium`, `pentium-mmx`, `pentiumpro`, `pentium2`, `pentium3`, `pentium4`, `k6`, `k6-2`, `k6-3`, `athlon`, `athlon-tbird`, `athlon-4`, `athlon-xp` и `athlon-mp`.

Тип `i586` является синонимом типа `pentium`, а тип `i686` — типа `pentiumpro`. Типы `k6` и `athlon` соответствуют процессорам AMD.

Выбор типа процессора позволяет применять особенности планирования, соответствующие указанному процессору. Тем не менее, компилятор будет генерировать код, совместимый со всеми процессорами семейства i386, если не указана опция `-march`.

-mfpmath=unit

Генерирует инструкции для выполнения операций над числами с плавающей точкой для указанного типа аппаратного математического обеспечения.

Задание в качестве параметра `unit` значения `387` приведет к тому, что для выполнения операций с плавающей точкой будет использоваться стандартный математический сопроцессор `387`, интегрированный с большинством чипов или эмули-

руемый. Код, откомпилированный с этой опцией, будет выполняться практически на всех процессорах семейства. Промежуточные временные результаты вычисляются с точностью до 80 бит.

Также см. опцию **-ffloat-store**, описанную в приложении Г. Эта опция для архитектуры i386 применяется по умолчанию.

Задание значения **sse** в качестве значения параметра **unit** приведет к использованию скалярных инструкций для операций над числами с плавающей точкой. Такие инструкции имеются в наборе инструкций SSE (Streaming SIMD Extension). Набор SSE поддерживается процессорами Pentium 3 и более новыми чипами, а также процессорами AMD Athlon-4, Athlon-xp и Athlon-mp. Более ранние версии набора инструкций SSE поддерживают только операции обычной точности, поэтому операции двойной и повышенной точности выполняются математическим сопроцессором 387. Новые версии SSE, имеющиеся только в процессорах Pentium4 и AMD x86-64, поддерживают также и операции двойной точности.

При задании значения **i387** в качестве значения параметра **unit** необходимо также установить опцию **-march** и одну из опций **-msse** или **-msse2**, разрешающих использование расширений SSE, а, следовательно, и опцию **-mfpmath**. Для архитектуры x86-64 использование расширений SSE разрешено по умолчанию. Генерируемый код будет выполняться гораздо быстрее (в большинстве случаев) и не имеет проблем с нестабильностью числовой точности, характерной для сопроцессора 387. Тем не менее, использование набора инструкций SSE может негативно сказаться на выполнении уже существующего ранее скомпилированного кода, который предполагает, что длина временных значений составляет 80 бит.

При указании в качестве параметра **unit** значения **sse**, 387 компилятор будет пытаться одновременно использовать два набора инструкций. Фактически это эквивалентно увеличению количества используемых регистров чипа в два раза, при этом инструкции набора 387 и SSE выполняются на различном аппаратном обеспечении. В настоящее время опция **-mfpmath** считается экспериментальной, поскольку выделяющий регистры процесс компилятора GCC не всегда может достаточно точно моделировать отдельные функциональные узлы.

-mieee-fp

Указывает, что компилятор должен использовать соответствующие стандартам IEEE операции сравнения чисел с плавающей точкой. Это позволяет корректно обрабатывать случаи, когда операции сравнения дают неупорядоченный результат. Использование операций сравнения чисел с плавающей точкой в соответствии со стандартами IEEE можно отключить опцией **-mno-mieee-fp**.

-minline-all-stringops

При установке этой опции для всех операций со строками будет применяться подстановка кода (*inline*). По умолчанию подстановка кода применяется для операций со строками в тех случаях, когда известно, что результат операции выровнен по границе 4 байт. Опция **-minline-all-stringops** делает все операции со строками подставляемыми. Это увеличивает размер, но уменьшает время выполнения

кода, использующего функции `memcpy()`, `strlen()` и `memset()` для строк небольшой длины.

-mmmx

Разрешает использование встроенных функций, обеспечивающих прямой доступ к расширениям MMX.

Отменить действие опции `-mmmx` можно с помощью опции `-mno-mmx`.

-mno-aligne-stringops

Отменяет выравнивание результатов расширяемых подстановкой кода (`inline`) операций со строками. Опция `-mno-aligne-stringops` уменьшает размер кода и увеличивает его производительность в случае, если переменная результата операции уже имеет выравнивание.

-mno-fancy-math-387

Некоторые эмуляторы математического сопроцессора 387 не поддерживают следующие инструкции `sin`, `cos` и `sqrt`, имеющиеся в аппаратном сопроцессоре. Опция `-mno-fancy-math-387` гарантирует, что эти инструкции генерироваться не будут. Опция будет работать только при указании опции `-funsafe-math-optimizations`.

Опция `-mno-fancy-math-387` устанавливается по умолчанию для систем FreeBSD, OpenBSD и NetBSD. Она игнорируется, если опция `-march` указывает процессоры пред назначаемой платформы, которые содержат устройство для выполнения операций с плавающей точкой. При этом эмуляция аппаратных инструкций не требуется.

-mno-fp-ret-in-387

Указывает, что для возвращаемых результатов функций не должны использоваться регистры устройства для выполнения операций с плавающей точкой.

Стандартное соглашение о вызовах функций предполагает хранение возвращаемых значений функций типов `float` и `double` в регистре устройства для выполнения операций с плавающей точкой. Если такое устройство отсутствует, то система должна эмулировать устройство для выполнения операций с плавающей точкой. Применение опции `-mno-fp-ret-in-387` приводит к тому, что возвращаемые результаты функций записываются в обычные регистры центрального процессора.

-mno-red-zone

Для процессоров AMD x86-64 в 64-битной среде эта опция подавляет использование так называемой "красной зоны" (red zone) кода x86-64.

"Красная зона" предусмотрена машинным интерфейсом прикладных программ (Embedded Applications Binary Interface, EABI) процессора x86-64 и представляет собой 128-битную область за пределами указателя стека, содержимое которой не изменяется обработчиками сигналов и прерываний и благодаря этому может использоваться для хранения временных данных без изменения указателя стека. Опция `-mno-red-zone` запрещает использование "красной зоны" процессоров AMD x86-64.

-momit-leaf-frame-pointer

Не сохраняет в регистре указатель на кадр стека (frame stack) для функций, имеющих вложенные вызовы (leaf functions). Это запрещает использование инструкций записи, установки и восстановления указателя кадра стека и позволяет освободить один регистр для использования во вложенных функциях.

Для запрещения использования регистра указателя на кадр стека для всех функций может использоваться опция **-fomit-leaf-frame-pointer**. Учтите, что это может создавать сложности при отладке программы.

-mpentium

Идентична опции **-mcpu=pentium**. Этот формат опции считается устаревшим.

-mpentiumpro

Идентична опции **-mcpu=pentiumpro**. Этот формат опции считается устаревшим.

-mpreferred-stack-boundary=number

Пытается поддерживать выравнивание стека к границе 2 в степени *number* байт. По умолчанию используется значение 4 (т.е. выравнивание производится по границе 16 байт или 128 бит).

При оптимизации размера кода с помощью опции **-Os** выравнивание устанавливается по минимально возможному размеру (четыре байта для процессора x86, восемь байт для x86-64). Для процессоров Pentium и Pentium Pro значения типа **double** и **long double** во избежание снижения скорости выполнения кода должны выравниваться по границе 8 байт. Для процессоров Pentium III использование типа **_m128** из набора расширения SSE (Streaming SIMD Extension) также снижает скорость выполнения кода, если данные этого типа не выравнивать по границе 16 байт.

Для обеспечения правильного выравнивания данных в стеке граница самого стека также должна иметь выравнивание, соответствующее хранящимся в стеке значениям. Кроме того, все генерируемые функции должны сохранять стек с применением выравнивания. Это означает, что вызов функции, откомпилированной с большим пределом выравнивания, из функции с меньшим пределом выравнивания, скорее всего, приведет к нарушению выравнивания данных в стеке. Поэтому при компиляции библиотек, использующих обратные вызовы, рекомендуется применять установки по умолчанию.

Дополнительное выравнивание, устанавливаемое опцией **-mpreferred-stack-boundary** увеличивает размер стека и, как правило, также увеличивает размер кода. Для кода, чувствительного к объему стека, например, кода встраиваемых систем (embedded systems) и ядра, границу выравнивания можно понизить до **-mpreferred-stack-boundary=2**.

Также см. опцию **-malign-double**.

-mpush-args

Использует операции **push** для записи выходных параметров. Такой метод короче, и по характеристикам производительности, как правило, он не хуже, чем использо-

зование для этого операций `sub/mov`. Для семейства процессоров Intel 386 и AMD x86-64 используется по умолчанию. Его можно отменить опцией `-mno-push-args`. В некоторых случаях это позволяет несколько увеличить производительность благодаря улучшенному планированию и ослаблению зависимостей.

`-mregparm=number`

Устанавливает количество регистров, используемых для передачи целочисленных аргументов. По умолчанию для передачи аргументов регистры не используются. Максимальное значение, допустимое для параметра `number`, равно 3. Количество используемых регистров можно устанавливать отдельно для каждой функции с помощью атрибута `regparm`.

При использовании опции `-mregparm` с ненулевым значением параметра `number` все модули программы, включая используемые библиотеки, должны компилироваться с одним и тем же значением этого параметра.

`-mrtd`

Использует другое соглашение о вызове функций, в соответствии с которым возврат из функций с фиксированным количеством аргументов осуществляется с помощью инструкции `ret num`. Инструкция `ret num` при возврате из функции выталкивает аргументы из стека. Это позволяет сэкономить одну инструкцию в вызывающем модуле.

Такой тип соглашения о вызовах можно задать для отдельной функции, объявив ее с атрибутом `stdcall`. Действие опции `-mrtd` можно отменить с помощью атрибута `cdecl`.

Такое соглашение о вызовах функций несовместимо с системой UNIX, поэтому его нельзя применять при использовании библиотек, откомпилированных с помощью компиляторов UNIX. Кроме того, необходимо объявление прототипов всех функций с переменным количеством аргументов. Если этого не сделать, то для вызова таких функций будет сгенерирован неверный код.

Учтите, что каждый раз при вызове функции с лишними аргументами будет возникать ошибка. При стандартном соглашении о вызовах лишние аргументы игнорируются.

`-msoft-float`

Генерирует код, содержащий вызовы функций из библиотек для выполнения операций над числами с плавающей точкой. Библиотеки функций не входят в состав компилятора GCC. Как правило, используются библиотеки компилятора C пред назначаемой целевой платформы, но их невозможно непосредственно подключить в кросс-компиляции. Поэтому при выполнении кросс-компиляции необходимо точно указывать соответствующие функции библиотек.

При компиляции программ для компьютеров, на которых функции возвращают числовые результаты с плавающей точкой через стек регистров математического со-процессора 80387, могут генерироваться отдельные машинные коды операций, не-смотря на использование опции `-msoft-float`.

-msse

Разрешает использование встроенных функций, предоставляющих прямой доступ к инструкциям расширенного набора инструкций SSE (Streaming SIMD Extension). Действие опции **-msse** отключается применением опции **-mno-sse**.

-msse2

Разрешает использование встроенных функций прямого доступа к инструкциям расширенного набора инструкций SSE2 (Streaming SIMD Extension версии 2). Действие этой опции можно отключить указанием опции **-mno-sse2**.

-mthreads

Включает поддержку системы общей обработки исключений при использовании потоков, имеющейся в компиляторе Mingw32. Весь код, который использует обработку исключений в многопоточном режиме, должен компилироваться и компоноваться с установленной опцией **-mthreads**. Применение опции **-mthreads** также автоматически устанавливает опцию **-D_MT**. Во время компоновки с опцией **-lmingwthrd** подключается особая вспомогательная библиотека для обработки исключений в многопоточной среде. При этом также из программы убирается код раздельной обработки исключений для отдельных потоков.

-msvr3-shlib

Указывает, что компилятор должен помещать неинициализированные локальные переменные в сегмент данных **bss**. Чтобы компилятор размещал переменные в сегменте данных **data**, необходимо указать опцию **-mno-svr3-shlib**. Эти две опции доступны только для системы System V Release 3.

Опции для платформы Intel 960

Ниже перечислены опции, определенные для реализаций Intel 960.

-mtype

Устанавливает значения по умолчанию для указанного типа машины. Сюда входят параметры планирования инструкций, поддержка операций над числами с плавающей точкой и режимы адресации. Параметр **type** может принимать следующие значения: **ka**, **kb**, **mc**, **ca**, **cf**, **sa** и **sb**. По умолчанию используется значение **kb**.

-masm-compat

Включает режим совместимости с ассемблером iC960.

-mcode-align

Для обеспечения быстрого выполнения применяет выравнивание кода по границе 8 байт. В настоящее время эта опция установлена по умолчанию только для реализаций серий "С", для других серий по умолчанию используется опция **-mno-code-align**.

-mcomplex-addr

Указывает компилятору, что для этой реализации i960 желательно использовать режим комплексной адресации (complex addressing mode). Для серий "К" режимы комплексной адресации могут быть неэффективны, но для серий "С" их эффективность ощутима. По умолчанию для всех процессоров, кроме **c8** и **cc**, т.е. тех процессоров, для которых используется опция **-mcomplex-addr**, установлена опция **-mno-complex-addr**.

-mic-compat

Включает режим совместимости с iC960.

-mic2.0-compat

Включает режим совместимости с iC960 версии 2.

-mic3.0-compat

Включает режим совместимости с iC960 версии 3.

-mintel-asm

Идентична опции **-masm-compat**.

-mleaf-procedures

Эта опция указывает, что компилятор должен пытаться изменять процедуры, имеющие вложенные вызовы (leaf procedures), таким образом, чтобы они могли быть вызваны не только с помощью инструкции **call**, но и с помощью инструкции **bal**. Действие опции **-mleaf-procedures** можно отменить применением другой опции **-mno-leaf-procedures**.

При использовании опции **-mleaf-procedures** генерируется более эффективный код для явных вызовов, когда инструкция **bal** может подставляться ассемблером или компоновщиком. В других случаях код будет менее эффективным, например, при вызове функций с помощью указателей.

-mlong-double-64

Реализует тип данных **long double** как число с плавающей точкой длиной 64 бита. Без этой опции значения типа **long double** реализуются как 80-битные числа с плавающей точкой.

Единственной причиной наличия опции **-mlong-double-64** является отсутствие поддержки 128-битных данных типа **long double** в библиотеке **fp-bit.c**. Эта опция используется только при компиляции программ, предназначенных для работы на процессорах с программной поддержкой операций над числами с плавающей точкой.

-mnumerics

Эта опция указывает, что процессор имеет аппаратную поддержку инструкций для математических действий над числами с плавающей точкой. Также см. опцию **-msoft-float**.

-mold-align

Включает режим поддержки выравнивания структур данных для процессоров Intel компилятора `gcc` версии 1.3. (При этом в качестве основного выпуска рассматривается `gcc 1.37`.) Применение опции `-mold-align` автоматически устанавливает опцию `-mstrict-align`.

-msoft-float

Эта опция указывает, что процессор не имеет аппаратной поддержки инструкций для математических действий над числами с плавающей точкой. Также см. опцию `-mnumerics`.

-mstrict-align

Запрещает доступ к данным без применения выравнивания. Для разрешения не выровненного доступа используется опция `-mno-strict-align`.

-mtail-call

Указывает компилятору предпринимать дополнительные попытки оптимизации (помимо машинно-независимых оптимизаций) рекурсивных вызовов с преобразованием их в ветвления. Как правило, эта опция не используется, поскольку логика определения эффективности такой оптимизации полностью не разработана. По умолчанию применяется опция `-mno-tail-call`.

Опции для платформы M32R/D

Ниже перечислены опции, определенные для архитектур Mitsubishi M32R/D.

-m32r

Генерирует код для архитектуры M32R. Действует по умолчанию для архитектур Mitsubishi M32R/D.

-m32rx

Генерирует код для архитектуры M32R/X.

-mcode-model=*name*

Указание в параметре `name` значения `small` указывает, что компилятор должен считать, что все объекты находятся в нижней области памяти объемом 16 Мбайт и их адреса могут загружаться с помощью инструкции `ld24`. Кроме того, компилятор предполагает, что доступ ко всем подпрограммам можно получить с помощью инструкции `b1`. Такое поведение используется по умолчанию.

Задание значения `medium` в параметре `name` указывает, что компилятор должен считать, что объекты могут находиться в любом месте памяти, адресуемой 32-битными адресами (для загрузки адресов объектов компилятор будет использовать инструкции `seth/add3`). Кроме того, компилятор предполагает, что доступ ко всем подпрограммам можно получить с помощью инструкции `b1`.

Значение `large` параметра `name` указывает, что компилятор должен считать, что объекты могут находиться в 32-битном адресном пространстве памяти (для загрузки адресов объектов компилятор будет использовать инструкции `seth/add3`). Кроме того, компилятор предполагает, что доступ к подпрограммам нельзя получить с помощью инструкции `b1` (компилятор в этом случае генерирует намного более медленную последовательность инструкций `seth/add3/j1`).

-msdata=*setting*

Эта опция указывает, какие элементы будут храниться в области данных малой длины. Область данных малой длины состоит из разделов `sdata` и `sbsa`. Объекты могут помещаться в область коротких данных явным образом с помощью атрибута `section` с указанием одного из двух разделов. Также см. опцию `-G`.

Задание значения `none` параметра `setting` запрещает использование области коротких данных. Переменные будут помещаться в один из следующих разделов: `.data`, `.bss` или `.rodata` (если не установлен атрибут `section`). Такое поведение используется по умолчанию.

Задание значения `sdata` в качестве параметра `setting` помещает короткие глобальные и статические данные в область коротких данных, но не генерирует специального кода для доступа к ним.

Присвоение параметру `setting` значения `use` помещает короткие глобальные и статические данные в область коротких данных и для доступа к ним генерирует специальный код.

-G *number*

Помещает все глобальные и статические объекты длиной не более `number` байт в раздел коротких данных или раздел `.bss`, а не в стандартный раздел данных или раздел `.bsa`. По умолчанию для параметра `number` установлено значение 8.

Чтобы опция `-G` действовала, необходимо установить параметр `setting` опции `-msdata` равным `sdata` или `use`.

Все модули одной программы должны компилироваться с одними и теми же параметрами опций `-msdata` и `-G`. Компиляция модулей программы при различных значениях параметра `number` может вызывать ошибки. Если при компиляции возникает ошибка, то компоновщик обнаруживает ее и предотвращает генерацию неверного кода.

Опции для платформы M680x0

Ниже перечислены опции, определенные для серии 680x0. Значения по умолчанию для описываемых ниже опций могут отличаться в зависимости от выбранного типа процессора (серии 680x0) при конфигурировании компилятора.

-m5200

Код генерируется для семейства процессоров 520X "coldfire". Эта опция установлена по умолчанию при настройке компилятора для систем серии 520X. Опция `-m5200` используется для микроконтроллеров с ядром 5200, включая MCF5202, MCF5203, MCF5204 и MCF5202.

При применении этой опции автоматически устанавливается опция **-mno_bitfield**.

-m68000

Код генерируется для аппаратных систем серии 6800. Эта опция используется для микроконтроллеров с ядром 68000 или EC000, включая 68008, 68302, 68306, 68307, 68322, 68328 и 68356. Опция **-m68000** устанавливается по умолчанию при конфигурировании компилятора для систем серии 6800.

При применении этой опции автоматически устанавливается опция **-mno_bitfield**.

-m68020

Код генерируется для систем серии 68020. Опция **-m68020** устанавливается по умолчанию при конфигурировании компилятора для систем серии 68020.

При применении этой опции автоматически устанавливается опция **-m_bitfield**.

-m68020-40

Код генерируется для систем серии 68040 без использования новых инструкций. Получаемый код может достаточно эффективно выполняться на системах 68020/68881, 68030 или 68040. Генерируемый код использует инструкции 68881, которые эмулируются в серии 68040.

-m68020-60

Код генерируется для систем серии 68060 без использования новых инструкций. Получаемый код может достаточно эффективно выполняться на системах 68020/68881, 68030 или 68040. Генерируемый код использует инструкции 68881, которые эмулируются в серии 68060.

-m68030

Код генерируется для систем серии 68030. Опция **-m68030** устанавливается по умолчанию при конфигурировании компилятора для систем серии 68030.

-m68040

Код генерируется для систем серии 68040. Опция **-m68040** устанавливается по умолчанию при конфигурировании компилятора для систем серии 68040. Эта опция запрещает использование инструкций 68881/68882, которые в системах серии 68040 должны эмулироваться программными средствами. Опция **-m68040** применяется, если система 68040 не имеет специального кода для эмуляции необходимого набора инструкций.

-m68060

Код генерируется для систем серии 68060. Опция **-m68060** устанавливается по умолчанию при конфигурировании компилятора для систем серии 68060. Эта опция запрещает использование инструкций 68881/68882, которые в системах серии 68060 должны программно эмулироваться. Опция **-m68060** применяется, если сис-

тема 68060 не имеет специального кода для эмуляции необходимого набора инструкций.

-m68881

Генерируемый код будет содержать инструкции набора 68881 для выполнения операций с числами с плавающей точкой. Эта опция применяется по умолчанию для большинства систем серии 68020, если при конфигурировании компилятора не была установлена опция **--nfp**.

-malign-int

Эта опция устанавливает выравнивание переменных типа **int**, **long**, **long long**, **float**, **double** и **long double** по границе 32 бит. По умолчанию используется опция **-mno-align-int**, применяющая 16-битное выравнивание перечисленных типов переменных. Выравнивание данных по границе 32 бит приводит к генерации кода большего объема, который выполняется немного быстрее на процессорах с 32-битными шинами.

При использовании опции **-malign-int** компилятор GCC выравнивает структуры, содержащие данные перечисленных выше типов, иначе, чем это предусматривает большинство имеющихся спецификаций машинного интерфейса прикладных программ (Embedded Applications Binary Interface, EABI) для серий m680x0.

-mbitfield

Разрешает использование инструкций битовых полей (bit-field instructions). Опция **-m68020** предполагает установку данной опции. Опция **-mbitfield** используется по умолчанию, если компилятор сконфигурирован для работы с системами серии 68020.

-mc68000

Идентична опции **-m6800**.

-mc68020

Идентична опции **-m68020**.

-mcpri32

Код генерируется для ядра CPU32. Эта опция установлена по умолчанию, если компилятор сконфигурирован для систем, основанных на CPU32. Опция **-mcpri32** используется для микроконтроллеров с ядром CPU32 или CPU32+ (системы 68330, 68331, 68332, 68333, 68334, 68336, 68340, 68341, 68349 и 68360).

Эта опция автоматически устанавливает опцию **-mno_bitfiled**.

-mfpa

Генерируется код, содержащий машинные инструкции набора Sun FPA для операций с плавающей точкой.

-mno-strict-align

Компилятор будет считать, что система будет обрабатывать ссылки на ячейки памяти без обязательного применения выравнивания. Чтобы компилятор применял выравнивание к адресуемым данным, следует установить опцию **-mstrict-align**.

-mnobitfield

Запрещает использование инструкций битовых полей (bit-field instructions). Эта опция устанавливается автоматически при применении опций **-m68000**, **-mcpu32** и **-m5200**.

-mpcrel

Применяется для систем серии 68000 режим относительной адресации с помощью счетчика команд (PC-relative, program counter addressing mode) вместо глобальной таблицы смещений (global offset table).

Опция **-mpcrel** автоматически устанавливает опцию **-fpic** для разрешения использования максимального 16-битного смещения при относительной адресации по счетчику команд. Для опции **-mpcrel** в настоящее время не поддерживается опция **-fPIC**.

-mrtd

Использует другое соглашение о вызове функций. Возврат из функций с фиксированным количеством аргументов осуществляется с помощью инструкции **rtd**, выталкивающей аргументы из стека при возврате. Это позволяет сэкономить одну инструкцию в вызывающем модуле.

Инструкция **rtd** поддерживается процессорами 68010, 68020, 68030, 68040, 68060 и CPU32 и не поддерживается процессорами 68000 и 5200.

Такое соглашение о вызовах функций несовместимо с операционными системами UNIX, поэтому его нельзя применять при необходимости использования библиотек, откомпилированных с помощью компиляторов UNIX. Помимо этого, для всех функций с переменным количеством аргументов, например, функции **printf()**, необходимо объявлять их прототипы. Без этого для вызова функций будет генерирован неверный код.

При вызове функций с лишними аргументами будет возникать ошибка. При использовании стандартного соглашения о вызовах лишние аргументы игнорируются.

-mshort

Значения типа **int** считаются 16-битными числами, как и значения типа **short int**.

-msoft-float

Генерируемый с этой опцией код будет содержать вызовы функций библиотеки эмуляции операций с плавающей точкой.

Для использования опции **-msoft-float** необходимо наличие соответствующих библиотек для кросс-компиляции, которые отсутствуют в системах m680x0. В нормальных условиях используются средства комплектного компилятора *C*, но их невозможно подключить непосредственно в кросс-компиляции. Встраиваемые системы для платформ **m68k-*-aout** и **m68k-*-coff** обеспечены программной поддержкой операций с плавающей точкой.

Опции для платформы M68HC1x

Ниже перечислены опции, определенные для микроконтроллеров 68HC11 и 68HC12. Значения по умолчанию для описываемых ниже опций могут отличаться в зависимости от типа процессора, выбранного при конфигурировании компилятора.

-m6811

Генерируется код для системы 68HC11. Эта опция установлена по умолчанию, если компилятор сконфигурирован для систем 68HC11.

-m6812

Генерируется код для системы 68HC12. Эта опция установлена по умолчанию, если компилятор сконфигурирован для систем 68HC12.

-m68hc11

Идентична опции **-m6811**.

-m68hc12

Идентична опции **-m6812**.

-mauto-incdec

Разрешает использование в системе 68HC12 любого из следующих режимов адресации: пре- и пост-автоинкрементного, автодекрементного.

-mshort

Приводит к тому, что значения типа **int** считаются 16-битными числами, как и значения типа **short int**.

-msoft-reg-count=count

Указывает, что количество псевдо-программных регистров, которые используются при генерации кода, должно быть равно числу **count**. Максимально допустимое значение для параметра **count** равно 32. Использование большего количества псевдо-программных регистров в некоторых случаях может привести к генерации более эффективного кода, но, в конечном счете, это зависит от особенностей программы. По умолчанию для систем 68HC11 используется значение 4, а для систем 68HC12 — значение 2.

Опции для платформы M88K

Ниже перечислены опции, определенные для архитектур Motorola 88K.

-m88000

Генерирует код, который может выполняться как на процессорах m88100, так и на процессорах m88110.

-m88100

Генерирует код, который эффективно выполняется на процессорах m88100 и может работать на процессорах m88110.

-m88110

Генерирует код, который эффективно выполняется на процессорах m88110 и может не работать на процессорах m88100.

-mbig-pic

Устаревшая опция. Вместо нее рекомендуется использовать опцию **-fPIC**.

-mhandle-large-shift

Включает код определения больших сдвигов разряда (*big-shifts*) более чем на 31 бит. Такие сдвиги либо перехватываются вызываемыми прерываниями, либо генерируется код их корректной обработки. По умолчанию компилятор GCC не предусматривает специальных действий для обработки больших сдвигов разряда.

-midentify-revision

Включает в выходные данные на языке ассемблера директиву **ident**, которая позволяет записывать имя исходного файла, название и версию компилятора, временную метку и набор используемых при компиляции флагов.

-mno-check-zero-division

Генерирует код, который не определяет ситуацию целочисленного деления на нуль. По умолчанию используется опция **-mcheck-zero-division**, которая позволяет обнаруживать деление целых чисел на ноль.

Некоторые модели процессора MC88100 в определенных условиях не позволяют перехватывать целочисленное деление на ноль. По умолчанию при компиляции кода, который может выполняться на таких процессорах, компилятор GCC генерирует код, который явным образом производит проверку ситуации целочисленного деления на нуль и при ее обнаружении вызывает исключение с номером 503. Для запрещения проверки целочисленного деления на ноль в коде, предназначенном для процессоров MC88100, используется опция **-mno-check-zero-division**.

Компилятор GCC предполагает, что процессор MC88110 корректно обрабатывает все случаи целочисленного деления на ноль. При указании опции **-m88110** явной проверки целочисленного деления на нуль не производится и опции **-mcheck-zero-division** и **-mno-check-zero-division** игнорируются.

-mno-underscores

Указывает на то, что генерируемый компилятором ассемблерный код не содержит символа подчеркивания перед именами. Обычно все символические имена в ассемблерном коде начинаются с символа подчеркивания.

-mocs-debug-info

Включает в вырабатываемый код дополнительную отладочную информацию. Отладочная информация содержит данные о регистрах, используемых в каждом кадре стека (stack frame), и соответствует стандарту 88open OCS (88open Object Compatibility Standard). Эта дополнительная информация позволяет выполнять отладку кода при отсутствии указателя на кадр стека.

Для систем DG/UX, Svr4 и Delta 88 SVr3.2 дополнительная отладочная информация генерируется по умолчанию. Для отключения поведения по умолчанию используется опция **-mno-octs-debug-info**. Для других систем 88K по умолчанию дополнительная отладочная информация не вырабатывается.

-mocs-frame-position

При генерации отладочной информации формата COFF для переменных и параметров, автоматически записываемых в стек, эта опция приводит к использованию смещения от канонического адреса кадра (canonical frame address). Это тот адрес, который содержится в указателе стека (регистр 31) при входе в функцию.

Указание опции **-mno-octs-frame-position** приводит к тому, что при генерации кода для переменных и параметров, автоматически записываемых в стек, смещение будет отсчитываться от регистра указателя кадра стека (регистр 30). Под действием этой опции адрес указателя кадра не удаляется из отладочной информации, задаваемой с помощью ключа **-g**.

Для систем DG/UX, Svr4, Delta 88 SVr3.2 и BCS по умолчанию используется смещение, это можно отключить с помощью опции **-mno-octs-frame-position**. Для других систем 88K опция **-mno-octs-frame-position** применяется по умолчанию.

-moptimize-arg-area

Эта опция позволяет сэкономить место путем реорганизации кадра стека (stack frame). Она приводит к генерации кода, который не соответствует спецификациям 88open, но занимает меньше памяти.

Опция **-mno-optimize-arg-area** позволяет запретить реорганизацию кадра стека, она применяется по умолчанию. Вырабатываемый при этом код полностью соответствует спецификациям "88open", но требует большего объема памяти.

-mshort-data-number

Генерирует более короткие ссылки на данные за счет адресации относительно регистра **r0**. Это позволяет загружать значения с помощью одной инструкции, а не двух, как это требуется в обычном режиме. Значение параметра **number** должно быть больше нуля, значения больше 65535 игнорируются.

Значение параметра *number* устанавливает, какие ссылки на данные будут изменены. Так, например, при опции `-mshort-data-512` будут изменены только те ссылки на данные, для которых смещение составляет менее 512 байт.

-mserialize-volatile

Генерирует код, гарантирующий последовательную согласованность обращений к энергозависимой памяти. Действует по умолчанию, ее можно отменить применением опции `-mno-serialize-volatile`.

Порядок обращений к ячейкам памяти для процессора MC88110 не всегда соответствует порядку следования инструкций, запрашивающих эти обращения. В частности, инструкция загрузки может быть выполнена до выполнения стоящей перед ней инструкцией записи. Такой порядок инструкций при наличии нескольких процессоров нарушает последовательную по времени согласованность ссылок на ячейки памяти. В случае, когда необходимо обеспечить согласованность, компилятор GCC генерирует специальные инструкции, обеспечивающие правильный порядок выполнения инструкций.

Процессор MC88100 не изменяет порядка следования обращений к памяти, поэтому он всегда обеспечивает их согласование по времени. Тем не менее, по умолчанию компилятор GCC всегда генерирует инструкции согласования обращений к памяти даже при использовании опции `-m88100`. Это обеспечивает совместимость полученного кода с процессорами MC88110. Если же генерируемый код должен выполняться только на процессорах MC88100, то для отключения генерирования лишних инструкций можно применить опцию `-mno-serialize-volatile`. Дополнительный код, обеспечивающий последовательную согласованность обращений к памяти, может оказывать влияние на производительность приложения. Поэтому, если заранее известно, что код предназначается для процессоров, обеспечивающих правильно согласованную последовательность ссылок, то выработку дополнительного кода лучше отключить.

-msvr3

Отключает расширения компилятора, соответствующие системе System V Release 4 (SVr4) (см. опцию `-msvr4`).

-msvr4

Подключает расширения компилятора, соответствующие системе System V Release 4 (SVr4). Эта опция выбирает вариант синтаксиса генерируемого ассемблерного кода, она позволяет препроцессору C распознавать директиву `#pragma weak` и принуждает компилятор GCC генерировать дополнительные декларативные директивы, которые используются в системе SVr4.

Опция `-msvr4` используется по умолчанию для конфигураций `m88k-motorola-sysv4` и `m88k-dg-dgux` аппаратуры `m88k`. Для этих конфигураций умолчание можно отключить назначением опции `-msvr3`. Для других конфигураций по умолчанию действует опция `-msvr3`.

-mtrap-large-shift

Идентична опции **-mhandle-large-shift**.

-muse-div-instruction

Генерирует код, использующий инструкцию **div** на процессоре MC88100 для выполнения целочисленного деления со знаком. По умолчанию инструкция **div** не используется.

Для процессора MC88100 при выполнении инструкции **div** с отрицательными операндами вызывается прерывание операционной системы. Затем операционная система выполняет операцию деления, но это требует значительных расходов времени. По умолчанию при компиляции кода, который может выполняться на процессоре MC88100, компилятор GCC эмулирует операцию целочисленного деления со знаком с помощью инструкции целочисленного деления без знака **divu**. Это позволяет уменьшить время выполнения операции. Тем не менее, такая эмуляция все-таки требует некоторого дополнительного времени и памяти, хотя и не такого значительного, как при использовании инструкции **div**. Если известно, что основные операции целочисленного деления выполняются над неотрицательными операндами, то может оказаться эффективным использовать инструкцию **div**.

Для процессора MC88110 инструкция **div** (также известная как **divs**) обрабатывает случай отрицательных operandов без помощи операционной системы. Поэтому при указании опции **-m88110** опция **-muse-div-instruction** игнорируется, и для выполнения операций целочисленного деления со знаком используется инструкция **div**.

Результат деления значения **INT_MIN** на **-1** неопределен. В частности, результаты этого деления при установленной и не установленной опции **-muse-div-instruction** могут отличаться.

-mversion-03.00

Эта опция устарела и игнорируется последними выпусками компилятора.

-mwarn-passed-structs

Выдает предупреждение в случае, когда функция в качестве аргумента или возвращаемого значения передает структуру (struct). Соглашения о передаче структур менялись с развитием языка C. Как результат, во многих случаях это может вызывать проблемы с переносимостью программного обеспечения. По умолчанию компилятор GCC не выдает предупреждений при передаче структур.

Опции для платформы MCore

Ниже перечислены опции, определенные для процессоров Motorola MCore.

-m210

Генерирует код для процессора 210.

-m340

Генерирует код для процессора 340.

-m4byte-functions

Приводит к тому, что все функции выравниваются по границе 4 байт. Действие опции **-m4byte-functions** отключается применением опции **-mno-4byte-functions**.

-mbig-endian

Генерирует код для процессора, использующего обратный порядок представления байтов. Также см. опцию **-mlittle-endian**.

-mcallgraph-data

Вырабатывает информацию о вызовах в виде графа (callgraph). Действие этой опции отключается применением опции **-mno-callgraph-data**.

-mdiv

Использует аппаратную инструкцию целочисленного деления. Эта опция используется по умолчанию. Ее можно отключить применением опции **-mno-div**.

-mhardlit

Применяет подстановку констант в коде, если ее можно выполнить не более чем двумя инструкциями. Действие этой опции можно отменить опцией **-mno-hardlit**.

-mlittle-endian

Генерирует код для архитектуры, использующей прямой порядок следования байтов. Также см. опцию **-mbig-endian**.

-mrelax-immediate

Допускает использование в битовых операциях промежуточных значений произвольного размера. Действие опции **-mrelax-immediate** отменяется установкой опции **-mno-relax-immediate**.

-mslow-bytes

Использует доступ к машинному слову при считывании отдельных байтов. Действие опции **-mslow-bytes** отменяется установкой опции **-mno-slow-bytes**.

-mwide-bitfields

Битовые поля (bitfields) хранятся в виде данных типа **int**. Действие этой опции отменяется установкой опции **-mno-wide-bitfields**.

Опции для платформы MIPS

Ниже перечислены опции, определенные для семейства компьютеров MIPS.

-m4650

Устанавливает опции `-msingle-float`, `-mmad` и `-mcpr=r4650`.

-mabi=name

Генерирует код для указанного машинного интерфейса прикладных программ (ABI). Параметр `name` может принимать следующие значения: `32`, `o64`, `n32`, `64`, `eabi` и `meabi`.

Значение `eabi` выбирает встроенный машинный интерфейс прикладных программ (EABI), определенный для Cygnus. Значение `meabi` приводит к выбору встроенного машинного интерфейса прикладных программ (ABI) процессоров MIPS. Машинные интерфейсы прикладных программ имеют 32- и 64-битные версии. По умолчанию при выборе 64-битной архитектуры компилятор будет генерировать 64-битный код. Тем не менее, с помощью опции `-mgrp32` можно добиться получения 32-битного кода и в этом режиме.

-mabicalls

Генерирует псевдооперации `.abicalls`, `.cupload` и `.cprestore`, которые используется отдельными портами System V4 для перемещаемого кода (PIC). Действие опции `-mabicalls` можно отключить применением опции `-mno-abicalls`.

-march=architecture

Генерирует код, который будет работать на указанной архитектуре. При этом в качестве параметра `architecture` может задаваться как имя базовой архитектуры MIPS ISA, так и имя конкретного процессора.

Имена базовых архитектур выбираются из следующего перечня: `mips1`, `mips3`, `mips3`, `mips4`, `mips32` и `mips64`, а имена процессоров из перечня: `r2000`, `r3000`, `r3900`, `r4000`, `vr4100`, `vr4300`, `r4400`, `r4600`, `vr5000`, `r6000`, `r8000`, `4kc`, `4kp`, `5kc`, `20kc` и `orion`. В именах процессоров заключающие нули 000 могут быть сокращены до `k`, таким образом, имя `r2000` может быть представлено в виде `r2k`. Кроме того, префиксные символы являются необязательными, так, имя `vr5000` можно записать как `r5000`, `r5k` или `vr5k`.

Специальное имя архитектуры `from-abi` приводит к выбору наиболее совместимой архитектуры для указанного машинного интерфейса прикладных программ, ABI. (Т.е. `mips1` для 32-битных интерфейсов и `mips3` для 64-битных интерфейсов.)

Макрос `_MIPS_ARCH` содержит имя пред назначаемой архитектуры (для которой выполняется компиляция) в виде строки буквенных символов. Кроме того, определен макрос `_MIPS_ARCH_architecture`, который использует имя архитектуры, указанной в `_MIPS_ARCH` (все буквенные символы должны быть в верхнем регистре). Например, опция `-march=r2000` создает определение макроса `_MIPS_ARCH`, содержащего строку "`r2000`", и макроса `_MIPS_ARCH_R2000`. Макрос `MIPS_ARCH` содержит полное имя архитектуры с префиксными символами, т.е. нули 000 никогда не сокращаются до `k`. Указание имени архитектуры `from-abi` приводит к формированию макросов "`mips1`" или "`mips3`". Если опция `-march` не указана, используется имя архитектуры по умолчанию.

-mdouble-float

Разрешает процессору выполнять операции двойной точности над числами с плавающей точкой.

Также см. опцию **-msingle-float**.

-membedded-data

Приводит к тому, что сначала, когда это возможно, предпринимается попытка выделения памяти под переменные в разделе данных только для чтения, а затем — в разделе коротких данных. Когда и это невозможно, память под переменные выделяется в разделе данных. Эта опция приводит к генерации более медленного кода, но при этом уменьшается объем требуемой оперативной RAM-памяти. Это может быть предпочтительным для некоторых встраиваемых систем (embedded systems). Действие опции **-membedded-data** можно отменить с помощью другой опции **-mno-embedded-data**.

-membedded-pic

Генерируется код с перемещаемой адресацией (PIC-код), предназначенный для использования в некоторых встраиваемых системах (embedded systems). При вызове всех функций используется относительная адресация с помощью счетчика команд (PC-relative addressing), а доступ ко всем данным осуществляется с помощью регистра \$gp. Адресное пространство такого доступа составляет 65536 байт глобальных данных. Опция **-membedded-pic** требует использования ассемблера GNU **as** и компоновщика GNU **ld**, которые выполняют основную часть работы. Опция **-membedded-pic** в настоящее время доступна только для операционных платформ, использующих формат ECOFF, и не работает с форматом ELF.

-menrty

Использует псевдооперации **entry** и **exit**. Опция **-menrty** может использоваться только вместе с опцией **-mips16**.

-mfix 7000

Опция передается ассемблеру **gas**. Это приводит к тому, что между следующими подряд инструкциями считывания регистров **mfhi** и **mflo** вставляются опкоды пустых операций **noop**.

-mflush-func=*function*

Устанавливает имя функции, которая будет вызываться для сброса содержимого кэшей I и G в память. Опция **-mno-flush-func** указывает компилятору не вызывать функцию сброса кэш-памяти.

При вызове функция сброса должна принимать те же аргументы, что и стандартная функция **_flush_func()**. Т.е. адрес той области памяти, куда будет записываться содержимое кэшей, размер области памяти и параметр "3" ("сбрасывать содержимое обоих кэшей"). Поведение по умолчанию зависит от системы, для которой сконфигурирован компилятор GCC, но в большинстве случаев используются функции **_flush_func()** или **__cpu_flush()**.

-mfp32

Указывает, что все регистры для чисел с плавающей точкой имеют длину 32 бита.

-mfp64

Эта опция указывает компилятору, что все регистры для чисел с плавающей точкой имеют длину 64 бита.

-mfused-madd

Генерирует код, который использует аппаратные инструкции умножения и сложения, если такие инструкции имеются. Аппаратные инструкции умножения и суммирования при их наличии генерируются по умолчанию. Тем не менее, это может быть нежелательно в случае, когда использование повышенной точности вызывает проблемы. Это также может быть нежелательно для отдельных чипов в режимах, когда денормализованные значения округляются до нуля и когда денормализованные значения при выполнении операций умножения и суммирования могут вызывать исключения.

Использование аппаратных операций над числами с плавающей точкой можно отключить с помощью опции **-mno-fused-madd**.

-mgas

Генерирует код для ассемблера GNU. Эта опция установлена по умолчанию для базовой платформы OSF/1, использующей формат OSF/rose. Кроме того, опция **-mgas** применяется по умолчанию, когда установлена опция конфигурации **--with-gnu-as**.

Также см. опцию **-mmpir-as**.

-mgp32

Предполагает, что регистры общего назначения имеют длину 32 бита.

-mgp64

Предполагает, что регистры общего назначения имеют длину 64 бита.

-mgropt

Указывает компилятору записывать все объявления данных в текстовом разделе перед инструкциями, что позволяет ассемблеру MIPS генерировать обращения к памяти длиной в одно машинное слово вместо использования двух слов для коротких глобальных и статических данных. Опция **-mgropt** используется по умолчанию при включенной оптимизации, ее действие можно отключить с помощью опции **-mno-gropt**.

-mhalf-pic

Помещает указатели на внешние ссылки в раздел данных и загружает их из раздела данных, а не из текстового раздела. Действие опции **-mhalf-pic** можно отменить с помощью опции **-mno-half-pic**.

-mhard-float

Генерирует код, содержащий аппаратные инструкции для выполнения операций над числами с плавающей точкой. Опция **-mhard-float** используется по умолчанию.

Также см. опцию **-msoft-float**.

-mint64

Генерирует данные типов **int** и **long** длиной 64 бита.

Длина данных типов **int** и **long**, используемая по умолчанию, зависит от машинного интерфейса прикладных программ (ABI). Для всех поддерживаемых интерфейсов прикладных программ длина данных типа **int** составляет 32 бита. Интерфейс прикладных программ **n64** и 64-битный встраиваемый интерфейс Cygnus EABI используют данные типа **long** длиной 64 бита. Для всех остальных интерфейсов длина типа **long** составляет 32 бита. Указатель имеет ту же длину, что и тип **long** или целочисленные регистры в зависимости от того, какая длина меньше.

Также см. опции **-mlong64** и **-mlong32**.

-mips1

Идентична опции **-march=mips1**.

-mips2

Идентична опции **-march=mips2**.

-mips3

Идентична опции **-march=mips3**.

-mips4

Идентична опции **-march=mips4**.

-mips16

Разрешает применение 16-битных инструкций. Действие опции **-mips16** можно отменить с помощью опции **-mno-mips16**.

-mips32

Идентична опции **-march=mips32**.

-mips64

Идентична опции **-march=mips64**.

-mlong-calls

Генерирует код, в котором все функции вызываются с помощью инструкции **JALR**, требующей перед вызовом функции загрузки ее адреса в регистр. Эта опция необходима для вызовов функций, находящихся за пределами сегмента размером 512 Мбайт, доступ к которым невозможно получить непосредственно с помощью регистров-указателей. Действие опции **-mlong-calls** можно отменить с помощью опции **-mno-long-calls**.

-mlong32

Генерирует данные типов `long`, `int` и указателей длиной 32 бита.

Используемая по умолчанию длина указателей и данных типов `int`, `long` зависит от применяемого машинного интерфейса прикладных программ (ABI). Для всех поддерживаемых интерфейсов прикладных программ длина данных типа `int` составляет 32 бита. Интерфейс прикладных программ `n64` и 64-битный встраиваемый интерфейс прикладных программ Cygnus EABI используют данные типа `long` длиной 64 бита. Для всех остальных интерфейсов длина данных типа `long` составляет 32 бита. Указатель имеет ту же длину, что и тип `long` или целочисленные регистры в зависимости от того, какая длина меньше.

Также см. опции `-mint64` и `-mlong64`.

-mlong64

Генерирует данные типов `long` длиной 64 бита.

Используемая по умолчанию длина указателей и данных типов `int`, `long` зависит от применяемого машинного интерфейса прикладных программ (ABI). Для всех поддерживаемых интерфейсов прикладных программ длина данных типа `int` составляет 32 бита. Интерфейс прикладных программ `n64` и 64-битный встраиваемый интерфейс прикладных программ Cygnus EABI используют данные типа `long` длиной 64 бита. Для всех остальных интерфейсов длина данных типа `long` составляет 32 бита. Указатель имеет ту же длину, что и тип `long` или целочисленные регистры в зависимости от того, какая длина меньше.

Также см. опции `-mint64` и `-mlong32`.

-mmad

Разрешает использование инструкций `mad`, `madl` и `mul` для чипа `r4650`. Действие опции `-mmad` можно отменить с помощью опции `-mno-mad`.

-mtempcpry

Генерирует код, который при перемещении блоков будет вызывать соответствующую случаю строковую функцию, `tempcpry()` или `bscopy()`, а не генерировать подстановливаемый (inline) код. Вызовы этих функций можно запретить с помощью опции `-mno-tempcpry`.

-mmips-as

Генерирует код для ассемблера MIPS и вызывает программу `mips-tfile` для формирования стандартной отладочной информации. Опция `-mmips-as` установлена для всех платформ по умолчанию, за исключением базовой платформы OSF/1, использующей формат OSF/rose. Если указаны опции `-gstabs` или `-gstabs+`, программа `mips-tfile` будет инкапсулировать записи формата STABS в информацию формата MIPS ECOFF.

Также см. опцию `-mgas`.

-mmips-tfile

Выходной объектный файл ассемблера MIPS обрабатывается с помощью программы **mips-tfile** с целью включения в него отладочной информации. Действие опции **-mmips-tfile** можно отменить с помощью опции **-mno-mips-tfile**. Опция **-mno-mips-tfile** должна использоваться только в том случае, когда программа **mips-tfile** вносит ошибки, мешающие выполнить компиляцию.

Если программа **mips-tfile** не выполняется, отладчик не будет иметь доступа к локальным переменным. Кроме того, имена временных объектов **stage2** и **stage3** будут переданы ассемблеру в объектном файле, что значит, что объекты не будут считаться одинаковыми.

-mrnames

Указывает компилятору генерировать код с использованием программных имен регистров MIPS, а не аппаратных имен (т.е. **a0**, а не **\$4**). Единственным известным ассемблером, который поддерживает эту опцию, является ассемблер Algorithmics. Действие опции **-mrnames** можно отменить с помощью опции **-mno-rnames**.

-msingle-float

Предполагает, что математический сопроцессор, выполняющий операции с плавающей точкой, поддерживает только операции обычной точности. Также см. опцию **-mdouble-float**.

-msoft-float

Генерирует код, содержащий вызовы функций библиотек для выполнения операций с плавающей точкой.

Библиотеки функций не входят в состав компилятора GCC. Как правило, используются библиотеки языка C. Но их невозможно непосредственно подключить во время кросс-компиляции. Поэтому при выполнении кросс-компиляции следует указывать точные имена библиотечных функций.

Также см. опцию **-mhard-float**.

-msplit-addresses

Генерирует код, который отдельно загружает старшие и младшие части адресов. Это позволяет компилятору оптимизировать излишнюю нагрузку на старшие части адресов. Оптимизация требует использования ассемблера GNU **as** и компоновщика GNU **ld**. Оптимизация используется по умолчанию для систем, в которых **as** и **ld** считаются стандартными. Действие опции **-msplit-addresses** можно отменить установкой опции **-mno-split-addresses**.

-mstats

При этой опции для каждой функции, не расширяемой подстановкой кода, компилятор записывает одну строку в стандартный файл ошибок, содержащий статистические данные о программе (количество регистров, размер стека и т.п.). Действие опции **-mstats** можно отменить с помощью опции **-mno-stats**.

-mtune=*architecture*

Выполняет оптимизацию для указанной архитектуры. Помимо всего прочего эта опция управляет методом планирования инструкций и оценочной стоимостью арифметических операций. Допустимые значения для параметра *architecture* идентичны значениям параметра опции **-march**.

Если опция **-mtune** не указана, то компилятор оптимизирует код для процессора, указанного опцией **-march**. При совместном использовании опций **-march** и **-mtune** имеется возможность генерировать код, который будет совместим с семейством процессоров и при этом будет оптимизирован для конкретного процессора из этого семейства.

Опция **-mtune** приводит к созданию макросов **_MIPS_TUNE** и **_MIPS_TUNE_architecture**, которые генерируются по тем же правилам, что и для опции **-march**.

-muninit-const-in-rodata

При использовании совместно с опцией **-membedded-data** эта опция приводит к тому, что неинициализированные константы будут храниться в разделе данных только для чтения.

-EL

Генерирует код для режима работы с прямым порядком байтов. Также см. опцию **-EB**.

-EB

Генерирует код для режима работы с обратным порядком байтов. Также см. опцию **-EL**.

-G *number*

Помещает все глобальные и статические объекты размером не более *number* байт в раздел коротких данных или раздел **.sbss**, а не в стандартный раздел данных или **.bss**. Это позволяет ассемблеру на основании глобального указателя (**gp** или **\$28**) использовать инструкции доступа к памяти длиной в одно слово вместо инструкций стандартной длины в два слова. По умолчанию для параметра *number* установлено значение 8 при использовании ассемблера MIPS и значение 0 для ассемблера GNU.

Параметр этой опции передается ассемблеру и компоновщику, поэтому все модули программы обязательно должны компилироваться с одним и тем же значением параметра *number*.

-noscpp

Указывает ассемблеру MIPS при трансляции программы в машинный код не запускать свой предпроцессор для входных файлов с ассемблерным кодом (файлов с суффиксом **.s**).

-no-crt0

Исключает использование **crt0**.

Опции для платформы MMIX

Ниже перечислены опции, определенные для MMIX.

-mabi-*setting*

Задание значения `mmisware` в качестве параметра *setting* приводит к генерации кода, который передает функции параметры и возвращает значения результатов функции так, что внутри вызываемой функции они видимы как регистры `$0` и выше.

Задание значения `gnu` в качестве параметра *setting* приводит к генерации машинного интерфейса прикладных программ GNU, использующего глобальные регистры `$231` и выше.

-mbase-addresses

Генерирует код, использующий базовые адреса.

При использовании базовых адресов автоматически генерируется запрос (обрабатываемый ассемблером и компоновщиком) для константы, которая должна быть установлена в глобальном регистре. Регистр используется для одного или нескольких запросов базовых адресов из диапазона от 0 до 255 на основании значения, содержащегося в регистре. Как правило, это приводит к генерации более быстрого кода меньшего объема, но при этом ограничивается количество адресуемых элементов данных. Таким образом, программа, использующая большое количество статических данных, может потребовать установки опции `-mno-base-addresses` для отмены действия опции `-mbase-addresses`.

-mbranch-predict

Использует инструкции, допускающие возможность ветвления, когда прогнозирование (prediction) статического ветвления указывает на вероятность ветвления. Действие опции `-mbranch-predict` можно отменить опцией `-mno-nch-predict`.

-melf

Генерирует исполняемый файл в формате ELF, вместо используемого имитаторами MMIX по умолчанию формата `mmo`.

-mepsilon

Генерирует инструкции сравнения чисел с плавающей точкой с использованием ипсilon-регистра `rE`. Действие опции `-mepsilon` можно отменить с помощью опции `-mno-epsilon`.

-mknuthdiv

Устанавливает присвоение знака остатка от деления знака делителя. По умолчанию используется опция `-mno-knuthdiv`, при этом знак остатка от деления будет соответствовать знаку делимого. Оба метода с точки зрения арифметики считаются верными, хотя, как правило, используется только последний метод.

-mlibfuncs

Указывает, что внутренние библиотечные функции компилируются для передачи всех значений через регистры независимо от размера значений. Действие опции **-mlibfuncs** можно отменить с помощью опции **-mno-libfuncs**.

-msingle-exit

Генерирует код с одной точкой выхода для каждой функции. Действие опции **-msingle-exit** можно отменить с помощью опции **-mno-single-exit**.

-mtoplevel-symbols

Перед именем всех глобальных символов вставляет двоеточие, что позволяет использовать код ассемблера с директивой **PREFIX**. Действие опции **-mtoplevel-symbols** можно отменить с помощью опции **-mno-toplevel-symbols**.

-mzero-extend

При считывании из памяти данных длиной до 64 бит по умолчанию используются инструкции дополнения нулями, а не знаковыми разрядами. Действие опции **-mzero-extend** можно отменить с помощью опции **-mno-zero-extend**.

Опции для платформы MN10200

Ниже приведена опция, определенная для архитектуры Matsushita MN10200.

-mrelax

Указывает, что компоновщик должен выполнять релаксационную оптимизацию для уменьшения длины ветвлений, дистанций вызовов и абсолютных адресов памяти. Эта опция используется только в качестве опции командной строки при выполнении последнего этапа компоновки. Кроме того, опция **-mrelax** исключает возможность отладки символов.

Опции для платформы MN10300

Ниже перечислены опции, определенные для архитектуры Matsushita MN10300.

-mat33

Генерирует код, который использует особенности, характерные для процессора AM33. По умолчанию установлена опция **-mno-am33**, при которой особенности этого процессора не используются.

-mmult-bug

Генерирует код, который исключает ошибки, содержащиеся в инструкциях умножения процессоров MN10300. Эта опция используется по умолчанию. Ее действие можно отключить с помощью опции **-mno-mmult-bug**.

-mno-crt0

По умолчанию установлено, что в программу компонуются разделяемые библиотеки языка C, однако эта опция исключает их компоновку в программу.

-mrelax

Указывает, что компоновщик должен выполнять релаксационную оптимизацию для уменьшения длины ветвлений, вызовов и абсолютных адресов памяти. Эта опция используется только в качестве опции командной строки при выполнении последнего этапа компоновки. Кроме того, опция **-mrelax** исключает возможность отладки символов.

Опции для платформы NS32K

Ниже перечислены опции, определенные для систем серии 32000. Значения по умолчанию для описываемых ниже опций могут отличаться в зависимости от стиля серии 32000, выбранного при настройке компилятора.

-m32032

Генерирует код для процессора 32032. Эта опция установлена по умолчанию, если компилятор сконфигурирован для работы на процессорах 32032 и 32016.

-m32081

Генерирует код, содержащий инструкции процессора 32081 для выполнения операций с плавающей точкой. Эта опция установлена по умолчанию для всех процессоров.

Также см. опцию **-m32381**.

-m32332

Генерирует код для процессора 32332. Эта опция установлена по умолчанию, если компилятор сконфигурирован для работы на процессоре 32332.

-m32381

Генерирует код, содержащий инструкции для выполнения операций с плавающей точкой процессора 32381. Опция **-m32381** также автоматически устанавливает опцию **-m32081**. Процессоры серии 32381 совместимы только с процессорами 32332 и 32532. Опция **-m32381** установлена по умолчанию для конфигураций **pc532-netbsd**.

-m32532

Генерирует код для процессора 32332. Эта опция установлена по умолчанию, если компилятор сконфигурирован для работы на процессоре 32532.

-mbitfield

Генерирует код, использующий инструкции для работы с битовыми полями (bitfield). Эта опция установлена по умолчанию для всех архитектур, за исключением 32532.

-mhitem

Генерирует код, который может загружаться выше области 512 Мбайт.

Многие режимы адресации процессоров серии NS32000 используют смещение до 512 Мбайт. Если адрес находится за пределами 512 Мбайт, то смещения от нулевого адреса использовать не могут. Опция `-mhi.mem` может применяться для операционных систем и кодов ПЗУ (ROM).

Также см. опцию `-mpos.b`.

-mieee-compare

Указывает, что компилятор должен использовать операции сравнения чисел с плавающей точкой в соответствии со стандартами IEEE. Такие сравнения корректно выполняются в случае, когда результат сравнения неупорядочен. Зато обычно можно не обеспечивать требуемую для выполнения сравнений поддержку ядра. Действие опции `-mieee-compare` отменяет опция `-mno-ieee-compare`.

-mmulti-add

Пытается генерировать инструкции `polyF` и `dotF` для умножения и сложения чисел с плавающей точкой. Эта опция действует только в случае установленной опции `-m32381`.

Использование инструкций `polyF` и `dotF` влечет за собой изменение принципа выделения регистров, что, как правило, отрицательно оказывается на производительности программы. Опция `-mmulti-add` должна применяться при компиляции кода, от которого ожидается использование большого количества инструкций умножения и сложения.

Также см. опцию `-mnomulti-add`.

-mnobitfield

Указывает, что генерируемый код не должен использовать инструкции для работы с битовыми полями. На некоторых машинах быстрее выполняются операции сдвига и маскирования. Опция `-mnobitfield` установлена по умолчанию для систем PC532.

Также см. опцию `-mbitfield`.

-mnohimem

Предполагает, что код будет загружаться в начальные 512 Мбайт виртуального адресного пространства. Эта опция используется по умолчанию для всех платформ.

-mnomulti-add

Указывает, что компилятор не должен генерировать инструкции умножения и сложения `polyF` и `dotF` для чисел с плавающей точкой. Эта опция установлена по умолчанию.

Также см. опцию `-mmulti-add`.

-mnoregparam

Указывает, что аргументы функций не должны передаваться в регистрах. Эта опция используется по умолчанию для всех платформ.

Также см. опции `-mrtd` и `-mregparam`.

-mnosb

Опция сообщает компилятору, что регистр **sb** недоступен для использования или не инициализирован нулевым значением системой загрузки. Эта опция используется по умолчанию для всех платформ за исключением **pc532-netbsd**. Опция **-mnosb** устанавливается автоматически при установке опций **-mhimem** или **-fpic**.

-mregparam

Использует иное соглашение о вызовах функций, при котором первые два аргумента передаются в регистрах. Такое соглашение несовместимо с соглашением, применяемым в UNIX, поэтому его нельзя использовать с библиотеками, откомпилированными с помощью компилятора UNIX.

Также см. опции **-mrtd** и **-mnoregparam**.

-mrtd

Используется другое соглашение о вызове функций, в соответствии с которым при возврате из функций с фиксированным количеством аргументов используется инструкция **ret**, выталкивающая аргументы из стека при возврате.

Такое соглашение несовместимо со стандартным соглашением UNIX, поэтому его нельзя использовать с библиотеками, откомпилированными в UNIX. Кроме того, для всех функций с переменным количеством аргументов необходимо указать их прототипы. Если этого не сделать, то для вызова функций будет сгенерирован неверный код.

Также будет возникать ошибка при вызове функции с лишними аргументами. (При использовании стандартного соглашения о вызовах лишние аргументы игнорируются.)

Опция **-mrtd** получила свое название от инструкции **rtd** систем серии 680x0.

Также см. опции **-mregparam** и **-mnoregparam**.

-msb

Позволяет компилятору использовать регистр **sb** в качестве индексного регистра, который всегда инициализируется нулевым значением. Эта опция используется по умолчанию для конфигурации **pc532-netbsd**.

Также см. опцию **-mnosb**.

-msoft-float

Генерирует код, который для выполнения операций с плавающей точкой использует вызовы функций библиотек. Сами библиотеки, содержащие соответствующие функции, могут во время компиляции отсутствовать.

ОПЦИИ ДЛЯ ПЛАТФОРМЫ PDP-11

Ниже перечислены опции, определенные для PDP-11.

-mabshi

Использует схему **abshi2**. Опция используется по умолчанию. Ее действие можно отменить применением опции **-mno-abshi**.

-mbranch-cheap

Указывает при оптимизации не считать стоимость ветвлений высокой. Эта опция установлена по умолчанию.

Также см. опцию **-mbranch-expensive**.

-mbranch-expensive

Указывает при оптимизации считать стоимость ветвлений высокой. Эта опция используется только для экспериментов с генерируемым кодом.

Также см. опцию **-mbranch-cheap**.

-m10

Генерирует код для PDP-11/10.

-m40

Генерирует код для PDP-11/40.

-m45

Генерирует код для PDP-11/45. Эта опция действует по умолчанию.

-mac0

Возвращает результат с плавающей точкой в регистре ac0 (в регистре fr0 в синтаксисе ассемблера UNIX). По умолчанию установлена опция **-mno-ac0**, при этом результат с плавающей точкой возвращается через память.

-mbcopy

Указывает, что при копировании памяти не должна использоваться расширяемая подстановкой кода схема `movstrhi`.

Также см. опцию **-mbcopy-builtins**.

-mbcopy-builtins

При копировании памяти используется расширяемая подстановкой кода схема `movstrhi`. Эта опция действует по умолчанию.

Также см. опцию **-mbcopy**.

-mdec-asn

Использует синтаксис ассемблера DEC. Эта опция установлена по умолчанию для всех конфигураций платформ пред назначения PDP-11, за исключением `pdp11-*-bsd`.

-mfloat32

Устанавливает длину числовых данных с плавающей точкой равной 32 бит. Этую опцию можно задавать в виде **-mno-float64**.

-mfloat64

Устанавливает длину числовых данных с плавающей точкой равной 64 бит. Этую опцию можно задавать в виде **-mno-float32**.

-mfpu

Использует аппаратный формат чисел с плавающей точкой FPP. (Формат чисел с плавающей точкой FIS на платформе PDP-11/40 не поддерживается.)

-mint16

Устанавливает длину данных типа `int` равной 16 бит. Эта опция используется по умолчанию. Ее также можно задавать в виде `-mno-int32`.

-mint32

Устанавливает длину данных типа `int` равной 32 бита. Этую опцию также можно задавать в виде `-mno-int16`.

-msoft-float

Запрещает использование аппаратных операций с числами с плавающей точкой.

-msplit

Генерирует код для системы, включающий "split I&D". По умолчанию установлена опция `-mno-split`, которая генерирует ей код для системы без "split I&D".

-munix-asm

Использует синтаксис ассемблера UNIX. Эта опция установлена по умолчанию, когда компилятор сконфигурирован для `rdr11-*bsd`.

Опции для платформ RS/6000 и PowerPC

Ниже перечислены опции, определенные для IBM RS/6000 и PowerPC.

Компилятор GCC поддерживает два связанных набора инструкций для платформ RS/6000 и PowerPC. Набор инструкций "Power" содержит инструкции, поддерживающие набором микросхем RIOS, который используется в системах RS/6000. Набор инструкций "PowerPC" предназначен для архитектур микропроцессоров Motorola MPC5xx, MPC6xx, MPC8xx и микропроцессоров IBM 4xx.

Все архитектуры являются независимыми. Тем не менее, оба типа архитектур поддерживают достаточно большое подмножество общих инструкций. Все процессы, поддерживающие архитектуру "Power", имеют регистр MQ.

-mabi=altivec

Расширяет машинный интерфейс прикладных программ (ABI) включением в него расширений AltiVec. Это не изменяет используемый по умолчанию интерфейс прикладных программ, а лишь дополняет его расширениями интерфейса AltiVec. Для отключения расширений AltiVec применяется опция `-mabi=no-altivec`.

-mabi=spe

Расширяет машинный интерфейс прикладных программ (ABI) включением в него расширений SPE. Это не изменяет используемый по умолчанию интерфейс приклад-

ных программ, а лишь дополняет его расширениями интерфейса SPE. Для отключения использования расширений SPE применяется опция `-mabi=no-spe`.

-misel

Включает режим генерации инструкций набора ISEL. Эту опцию также можно указывать в виде `-misel=yes`. Для отключения генерации инструкций ISEL применяется опция `-misel=no`.

-mads

Во встраиваемых системах PowerPC эта опция предполагает, что модуль запуска называется `crt0.o`, а стандартные библиотеки C носят имена `libads.a` и `libc.a`.

-maix-struct-return

Возвращает структуры через память (как это предполагает машинный интерфейс прикладных программ систем AIX).

-maix32

Эта опция используется по умолчанию. Она отключает 64-битный машинный интерфейс прикладных программ (ABI) и устанавливает опцию `-mno-powerpc64`. Также см. опцию `-maix64`.

-maix64

Разрешает использование: 64-битного машинного интерфейса прикладных программ AIX; соглашения о вызовах, которое предполагает применение 64-битных указателей; 64-битных данных типа `long` и инфраструктуры их поддержки. Опция `-maix64` автоматически устанавливает опции `-mpowerpc64` и `-mpowerpc`.

Также см. опцию `-maix32`.

-maltivec

Эта опция разрешает использовать встроенные функции, которые открывают доступ к набору инструкций AltiVec. Кроме того, для настройки текущего машинного интерфейса прикладных программ на использование расширенного набора AltiVec необходимо также установить опцию `-mabi=altivec`. Для запрещения использования встроенных функций применяется опция `-mno-altivec`.

-mbig

В системе System V4 и встраиваемой системе PowerPC эта опция позволяет компилировать код для процессоров, работающих в режиме обратного порядка байтов.

Также см. опцию `-mlittle`.

-mbig-endian

Идентична опции `-mbig`.

-mbit-align

В системе System V4 и встраиваемой системе PowerPC эта опция выравнивает структуры и объединения, которые содержат битовые поля, по базовому типу битовых полей. Это поведение используется по умолчанию. Для его отмены применяется опция **-mno-bit-align**.

Например, по умолчанию структура, содержащая восемь битовых полей без знака длиной 1, будет выравниваться по границе 4 байт и будет иметь полный размер 4 байта. При указании опции **-mno-bit-align** такая структура будет выравниваться по границе 1 байт и будет иметь размер всего 1 байт.

-mcall-aix

В системе System V4 и встраиваемой системе PowerPC эта опция генерирует код с использованием соглашения о вызовах, аналогичного соглашению, используемому в системах AIX. Это поведение применяется по умолчанию, если компилятор сконфигурирован для **powerpc-*-eabiaix**.

-mcall-gnu

В системе System V4 и встраиваемой системе PowerPC эта опция генерирует код для системы GNU, основанной на Hurd.

-mcall-linux

В системе System V4 и встраиваемой системе PowerPC эта опция генерирует код для операционной системы Linux.

-mcall-netbsd

В системе System V4 и встраиваемой системе PowerPC эта опция генерирует код для операционной системы NetBSD.

-mcall-solaris

В системе System V4 и встраиваемой системе PowerPC эта опция генерирует код для операционной системы Solaris.

-mcall-sysv

В системе System V4 и встраиваемой системе PowerPC эта опция генерирует код, использующий соглашения о вызовах, соответствующие проекту марта месяца 1995 года машинного интерфейса прикладных программ (ABI) для System V, приложение для процессора PowerPC. Это поведение используется по умолчанию, если компилятор не настроен для работы с конфигурацией **powerpc-*-eabiaix**.

-mcall-sysv-eabi

Устанавливает опции **-mcall-sysv** и **-meabi**.

-mcall-sysv-noeabi

Устанавливает опции **-mcall-sysv** и **-mno-eabi**.

-mcpu=type

Устанавливает тип архитектуры, использование регистров, выбор мнемоники и параметры планирования инструкций в соответствии с указанным в поле *type* типом машины. Параметр *type* может принимать следующие значения: *rios*, *rios1*, *rsc*, *rios2*, *rs64a*, *601*, *602*, *603*, *603e*, *604*, *604e*, *620*, *630*, *740*, *7400*, *7450*, *750*, *power*, *power2*, *powerpc*, *403*, *505*, *801*, *821*, *823*, *860* и *common*.

Опция *-mcpu=common* позволяет генерировать код для базового процессора семейства, который может выполняться на любом процессоре Power или PowerPC. Компилятор GCC в этом случае будет использовать инструкции, общие для обеих архитектур, и не будет использовать регистр MQ. Для планирования инструкций также будут применяться правила базового процессора.

Опции *-mcpu=power*, *-mcpu=power2*, *-mcpu=powerpc* и *-mcpu=powerpc64* устанавливают соответственно машины Power, Power2, 32-битную версию машины PowerPC (т.е. не MPC601) и 64-битную версию машины PowerPC. При этом для планирования инструкций используются правила базового процессора.

Остальные опции предназначены для генерации кода на конкретный процессор. Код, полученный с помощью одной из приведенных опций, будет лучше всего работать на процессоре, соответствующем выбранной опции. При этом он может не работать на других процессорах.

Другие опции, которые устанавливаются автоматически при установке опции *-mcpu*, перечислены в таблице 21.6. В первом столбце приведены значения опции *-mcpu*, при выборе которых автоматически будут установлены опции, содержащиеся во втором столбце.

Таблица 21.6. Параметры опции *-mcpu*, автоматически устанавливающие другие машинные опции компилятора.

Параметры опции <i>-mcpu</i>	Автоматически устанавливаемые опции
<i>common</i>	<i>-mno-power</i> , <i>-mno-powerpc</i>
<i>power</i> , <i>power2</i> , <i>rios1</i> , <i>rios2</i> , <i>rsc</i>	<i>-mpower</i> , <i>-mno-powerpc</i> , <i>-mno-new-mnemonics</i>
<i>powerpc</i> , <i>rs64a</i> , <i>602</i> , <i>603</i> , <i>603e</i> , <i>604</i> , <i>620</i> , <i>630</i> , <i>740</i> , <i>7400</i> , <i>7450</i> , <i>750</i> , <i>505</i>	<i>-mno-power</i> , <i>-mpowerpc</i> , <i>-mnew-mnemonics</i>
<i>601</i>	<i>-mpower</i> , <i>-mpowerpc</i> , <i>-mnew-mnemonics</i>

-meabi

В системе System V4 и встраиваемой системе PowerPC эта опция генерирует код в соответствии с требованиями встраиваемого машинного интерфейса прикладных программ (Embedded Applications Binary Interface, EABI), который представляет собой набор изменений к спецификациям System V.4. Стек выравнивается по границе 8 байт. Для установки среды EABI в головной процедуре *main()* вызывается функция *__eabi()*. Если при этом применяется опция *-msdata*, то она использует регистры *r2* и *r13* для раздельной адресации отдельных областей данных малой длины.

При указании опции `-mno-eabi` стек выравнивается по границе 16 байт, из `main()` не вызывается функция инициализации среды, а опция `-msdata` использует только регистр `r13` для адресации единственной области коротких данных.

Опция `-meabi` действует по умолчанию, когда при конфигурировании компилятора GCC использовалась одна из опций `powerpc-* -eabi`.

-memb

Во встраиваемой системе PowerPC эта опция устанавливает бит `PPC_EMB` заголовка флагов ELF, что указывает на использование расширенного режима перемещения кода EABI.

-mfull-toc

Опция применяется по умолчанию. Компилятор будет выделять память, по крайней мере, для одного вхождения таблицы содержания ТОС ("table of contents") для каждой не автоматически разрешаемой ссылки на переменную программы. Кроме того, константы с плавающей точкой также будут включены в таблицу содержания (ТОС). Размер таблицы ТОС ограничен 16384 элементами.

Для уменьшения объема информации, хранящейся в ТОС, могут использоваться опции `-mno-fp-in-toc`, `-mno-sum-in-toc` и `-mmimal-toc`.

-mfused-madd

Генерирует код, использующий аппаратные инструкции умножения и сложения, если они имеются. Использование аппаратных инструкций умножения и сложения при их наличии включается по умолчанию. Действие опции `-mfused-madd` можно отменить применением опции `-mno-fused-madd`.

-mhard-float

Генерирует код, использующий набор инструкций математических операций над числами с плавающей точкой.

Также см. опцию `-msoft-float`.

-mmimal-toc

Эту опцию можно использовать при возникновении ошибок компоновщика, связанных с переполнением таблицы содержания ТОС ("table of contents"). Опция `-mmimal-toc` уменьшает размер ТОС за счет включения в нее только одного элемента содержания для каждого файла. Полученный код будет иметь немного больший размер и работать несколько медленнее, но объем таблицы содержания (ТОС) будет во много уменьшен.

Также см. опцию `-mfull-toc`.

-mlittle

В системе System V4 и встраиваемой системе PowerPC эта опция позволяет компилировать код для процессоров, работающих в режиме прямого порядка байтов (little endian).

Также см. опцию `-mbig`.

-mlittle-endian

Идентична опции **-mlittle**.

-mlongcall

Указывает, что при вызове всех функций должны использоваться указатели. Благодаря этому могут вызываться функции, находящиеся в памяти за пределами 64 Мбайт (67108864 байт). Действие этой опции можно отменять для отдельных функций путем указания атрибута **shortcall** или директивы **#pragma longcall(0)**. Использование длинных вызовов функций можно запретить опцией **-mno-longcall**.

Некоторые компоновщики способны определять вызовы функций за пределами 64 Мбайт и генерировать связующий код "на лету". В таких системах длинные вызовы обычно не назначаются, генерируемый при этом код работает несколько медленнее. На момент написания этой книги такая возможность была предусмотрена в компоновщиках AIX и GNU для систем PowerPC/64. В скором времени ожидается, что такая возможность будет добавлена в компоновщик GNU для систем PowerPC.

-mmultiple

Генерирует код, который использует инструкции "считывания нескольких слов" и "записи нескольких слов". Эти инструкции по умолчанию генерируются для систем Power и не генерируются для систем PowerPC.

Для отключения генерации этих инструкций используется опция **-mno-multiple**.

Не используйте опцию **-mmultiple** для систем PowerPC с прямым порядком байтов. Поскольку эти инструкции не работают с процессорами, использующими прямой порядок байтов. К исключениям можно отнести системы PPC740 и PPC750, для которых инструкции "загрузки нескольких слов" и "записи нескольких слов" могут применяться и в режиме прямого порядка байтов.

-mmvme

Во внедренной системе PowerPC эта опция предполагает, что модуль запуска называется **crt0.o**, а стандартные библиотеки C носят имена **libmvme.a** и **libc.a**.

-mnew-mnemonics

Выбирает мнемонику инструкций ассемблера, определенную для архитектуры PowerPC. Опция игнорируется в случае, если для выбранной архитектуры особая мнемоника не определена.

Также см. опцию **-mold-mnemonics**.

-mno-fp-in-toc

Эту опцию можно использовать при возникновении ошибок компоновщика, связанных с переполнением таблицы содержания ТОС ("table of contents"). Опция **-mno-fp-in-toc** уменьшает пространство, занимаемое таблицей содержания, за счет исключения из нее числовых констант с плавающей точкой.

Также см. опцию **-mfull-toc**.

-mno-sum-in-toc

Эту опцию можно использовать при возникновении ошибок компоновщика, связанных с переполнением таблицы содержания ТОС ("table of contents"). Опция **-mno-sum-in-toc** уменьшает пространство, занимаемое таблицей содержания, за счет генерации кода для вычисления адресов из смещений во время выполнения программы.

Также см. опцию **-mfull-toc**.

-mold-mnemonics

Выбирает мнемонику инструкций ассемблера, определенную для архитектуры Power. Опция игнорируется, если для выбранной архитектуры мнемоника не определена.

Также см. опцию **-mnew-mnemonics**.

-mpe

Генерирует код, поддерживающий среду IBM RS/6000 SP Parallel Environment (PE) за счет компоновки особого кода запуска приложения, использующего среду передачи сообщений.

При использовании этой опции в системе должна быть установлена среда PE либо в каталоге по умолчанию (`/usr/lpp/ppe.poe/`), либо следует применять опцию **-ppecs**, указывающую на другое расположение. Среда Parallel Environment не поддерживает потоки, поэтому опции **-mpe** и **-pthread** несовместимы.

-mpower

Генерирует инструкции, которые имеются только в архитектуре Power и использующие регистр MQ. Если при установке компилятора GCC использовалась эта опция, то она действует по умолчанию, ее можно отключить опцией **-mno-power**.

При одновременном указании опций **-mno-power** и **-mno-powerpc** компилятор будет использовать общие для обеих архитектур инструкции и отдельные общие вызовы AIX, регистр MQ используется не будет. Одновременная установка опций **-mpower** и **-mpowerpc** разрешает компилятору использовать любые инструкции обоих наборов и пользоваться регистром MQ. Эти опции должны указываться для процессора Motorola MPC601.

-mpower2

Генерирует инструкции, имеющиеся только в архитектуре Power2 и отсутствующие в архитектуре Power. При установке опции **-mpower2** автоматически применяется и опция **-mpower**.

Опция **-mpower2** действует по умолчанию, если она применялась при конфигурировании компилятора GCC, ее можно отключить опцией **-mno-power2**.

-mpowerpc

Генерирует инструкции, которые имеются только в 32-битной архитектуре PowerPC. При установке опции **-mpowerpc** автоматически устанавливается опция **-mpower**.

Опция **-mpowerpc** действует по умолчанию, если она применялась при конфигурировании компилятора GCC, ее можно отключить опцией **-mno-powerpc**.

Также см. опцию **-mpower**.

-mpowerpc-gropt

Генерирует код, который использует необязательные инструкции архитектуры PowerPC из набора инструкций общего назначения. Они включают в себя инструкции для вычисления квадратного корня чисел с плавающей точкой. При установке опции **-mpowerpc-gropt** автоматически устанавливается опция **-mpowerpc**.

Опция **-mpowerpc-gropt** действует по умолчанию, если она применялась при конфигурировании компилятора GCC, ее можно отключить опцией **-mno-powerpc-gropt**.

-mpowerpc-gfxopt

Генерирует код, использующий необязательные инструкции архитектуры PowerPC из набора графических инструкций, включая инструкции выбора (select) с плавающей точкой. При установке опции **-mpowerpc-gfxopt** автоматически устанавливается опция **-mpowerpc**.

Эта опция действует по умолчанию, если она применялась при конфигурировании компилятора GCC, ее можно отключить опцией **-mno-powerpc-gfxopt**.

-mpowerpc64

Генерирует дополнительные 64-битные инструкции, которые имеются только в архитектуре PowerPC64, регистры общего назначения считаются 64-битными. По умолчанию установлена опция **-mno-powerpc64**.

-mprototype

В системе System V.4 и встраиваемой системе PowerPC эта опция предполагает, что все вызовы функций с переменным количеством аргументов имеют соответствующие прототипы.

Без использования этой опции или при установке опции **-mno-prototype** компилятор должен перед каждой функцией без прототипа вставлять инструкцию установки или обнуления 6-го бита регистра условного кода (CR). Это необходимо для указания, передавались ли в регистрах с плавающей точкой числовые значения с плавающей точкой. При установке опции **-mprototype** значение 6-го бита будет устанавливаться процедурой вызова прототипизированных функций.

-mregnames

В системе System V.4 и встраиваемой системе PowerPC эта опция включает в выходной ассемблерный код имена регистров в символьной форме. Действие опции **-mregnames** можно отключить с помощью опции **-mno-regnames**.

-mrelocatable

Во встраиваемой системе PowerPC эта опция генерирует код, который позволяет перемещать программу в памяти во время ее выполнения. Если опция **-mrelocatable**

была установлена для любого модуля программы, то все компонуемые в программу объекты должны компилироваться с этой опцией или с опцией `-mrelocatable-lib`.

По умолчанию действует опция `-mno-relocatable`.

`-mrelocatable-lib`

Во встраиваемой системе PowerPC эта опция генерирует код, который позволяет перемещать программу в память во время ее выполнения. Модули программы, откомпилированные с установленной опцией `-mrelocatable-lib`, могут компоноваться с модулями, откомпилированными без установки опций `-mrelocatable` и `-mrelocatable-lib` и с модулями, откомпилированными с опцией `-mrelocatable`.

`-msdata=setting`

Установка опции `-msdata=eabi` в системе System V4 и встраиваемой системе PowerPC приводит к размещению коротких инициализированных неизменяемых глобальных и статических данных в раздел `.sdata2`, на который указывает регистр `r2`. При этом короткие инициализированные переменные глобальные и статические данные помещаются в раздел `.sdata`, на который указывает регистр `r13`. Короткие неинициализированные глобальные и статические данные помещаются в раздел `.sbss`, который находится рядом с разделом `.sdata`. Опция `-msdata=eabi` автоматически устанавливает опцию `-memb` и несовместима с опцией `-mrelocatable`.

Установка опции `-msdata=sysv` в системе System V4 и встраиваемой системе PowerPC приводит к размещению коротких глобальных и статических данных в раздел `.sdata`, на который указывает регистр `r13`. Короткие неинициализированные глобальные и статические данные помещаются в раздел `.sbss`, который находится рядом с разделом `.sdata`. Опция `-msdata=sysv` несовместима с опцией `-mrelocatable`.

Установка опции `-msdata=none` (эту опцию допускается использовать в формате `-mno-sdata`) во встраиваемой системе PowerPC приводит к размещению всех инициализированных глобальных и статических данных в раздел `.data`, а всех неинициализированных данных — в раздел `.bss`.

Установка опции `-msdata=default` (эту опцию допускается использовать в формате `-msdata`) в системе System V4 и встраиваемой системе PowerPC совместно с опцией `-meabi` аналогична установке опции `msdata=eabi`. Если опция `-meabi` не используется, код компилируется так, как если бы была установлена опция `-msdata=sysv`.

`-msdata-data`

В системе System V4 и встраиваемой системе PowerPC эта опция приводит к размещению коротких глобальных и статических данных в раздел `.sdata`. Кроме того, короткие неинициализированные глобальные и статические данные помещаются в раздел `.sbss`. При этом регистр `r13` для адресации коротких данных не используется.

Опция `-msdata-data` используется по умолчанию, если не установлены другие опции `-msdata`.

-msim

Во встраиваемой системе PowerPC эта опция предполагает, что модуль запуска называется **sim-crt0.o**, а стандартные библиотеки C носят имена **libsim.a** и **libc.a**.

-msoft-float

Генерирует код, который использует программную эмуляцию операций над числами с плавающей точкой.

Также см. опцию **-mhard-float**.

-mstrict-align

В системе System V4 и встраиваемой системе PowerPC эта опция генерирует код, который предполагает, что ссылки на не выровненные ячейки памяти будут обрабатываться системой. Опция **-mstrict-align** используется по умолчанию. Для отмены действия по умолчанию установите опцию **-mno-strict-align**.

-mstring

Генерирует код, который использует инструкции "загрузки строки" и "записи слова строки" для сохранения нескольких регистров и перемещения небольших блоков. Эти инструкции генерируются по умолчанию в системах Power, но не генерируются в системах PowerPC.

Для отмены выработки инструкций "загрузки строки" и "записи слова строки" необходимо указать опцию **-mno-string**.

Не используйте опцию **-mstring** в системах PowerPC, работающих в режиме прямого порядка байтов, поскольку инструкции "загрузки строки" и "записи слова строки" не работают при прямом порядке байтов. Исключение составляют системы PPC740 и PPC750, для которых эти инструкции могут использоваться также и в режиме прямого порядка байтов.

-msvr4-struct-return

Возвращает структуры длиной менее 8 байт в регистрах (как это предусматривается машинным интерфейсом прикладных программ SVR4).

-mtoc

В системе System V4 и встраиваемой системе PowerPC эта опция предполагает, что регистр 2 содержит указатель на глобальную область и указывает используемые программой адреса. Действие опции **-mtoc** можно отменить с помощью опции **-mno-mtoc**.

-mtune=type

Устанавливает параметры планирования инструкций для указанного в поле *type* типа машины, но не устанавливает тип архитектуры, параметры использования регистров и тип мнемоники инструкций. Допустимые значения параметра *type* соответствуют значениям параметра *type* для опции **-mcsrc**. Если опции **-mtune** и

-**mcri** указаны одновременно, то генерируемый код будет использовать архитектуру, регистры и мнемонику, установленные опцией -**mcri**, а параметры планирования инструкций, установленные опцией -**mtune**.

-mupdate

Генерирует код, использующий инструкции загрузки и записи, которые вычисляют и записывают в базовый регистр (base register) адрес области памяти. Опция -**mupdate** используется по умолчанию.

Для отмены генерации такого кода используется опция -**mno-update**, которая вводит небольшое время ожидания между обновлением содержимого указателя стека и обновлением адреса предыдущего кадра стека. Во время этого ожидания код текущего кадра стека при вызове прерываний или генерации сигналов может получить неверные данные.

-mvxworks

В системе System V4 и встраиваемой системе PowerPC эта опция указывает, что компиляция выполняется для системы **VxWorks**.

-mwindiss

Указывает, что компиляция выполняется для среды имитации окружения WindISS.

-mxl-call

Использует соглашение некоторых компиляторов систем AIX о передаче аргументов с плавающей точкой через стек. Для отключения действия опции -**mxl-call** используется опция -**mno-xl-call**.

В AIX аргументы с плавающей точкой передаются имеющим прототипы функциям не только с помощью регистров с плавающей точкой, но и через стек за пределами области сохранения регистров (register save area, RSA). Соглашение о вызовах AIX было без документирования расширено для поддержки своеобразной реализации языка "K&R C". Это расширение касается обработки случая вызова функции, принимающей адресуемые аргументы, с меньшим количеством аргументов, чем указано в объявлении функции. В компиляторе AIX XL доступ к числовым аргументам с плавающей точкой вне области RSA, в случае, когда подпрограмма компилируется без оптимизации, осуществляется с помощью стека. Поскольку постоянное хранение аргументов с плавающей точкой в стеке неэффективно и во многих случаях не требуется, опция -**mxl-call** по умолчанию не применяется. Она необходима только при вызове подпрограмм, откомпилированных в компиляторе AIX XL без оптимизации.

-myellowknife

Во встраиваемой системе PowerPC эта опция предполагает, что модуль запуска называется **crt0.o**, а стандартные библиотеки функций языка C носят имена **libyk** и **libc.a**.

-G *number*

Во встраиваемой системе PowerPC эта опция приводит к тому, что глобальные и статические данные длиной не более *number* байт помещаются в раздел коротких данных `.sdata` или `.sbss`, вместо стандартного раздела `.data` или `.bss`. По умолчанию для параметра *number* установлено значение 8.

Опция `-G` передается и компоновщику, поэтому все модули программы должны компилироваться с одним и тем же значением параметра *number*.

-pthread

Добавляет поддержку потоков за счет подключения библиотеки `pthread`. Опция `-pthread` одновременно устанавливает флаги для препроцессора и компоновщика.

Опции для платформы RT

Ниже перечислены опции, определенные для IBM RT PC.

-mcall-lib-mul

Генерирует инструкции `lmul$$` для выполнения операций умножения целых чисел.

Также см. опцию `-min-line-mul`.

-mfpxarg-in-fpregs

Использует вызывающую последовательность, несовместимую с соглашением о вызовах систем IBM, при котором числовые аргументы с плавающей точкой передаются в регистрах для чисел с плавающей точкой. Если установлена опция `-mfpxarg-in-fpregs`, то заголовочный файл `stdarg.h` не будет обрабатывать числовые операнды с плавающей точкой.

Также см. опцию `-mfpxarg-in-gregs`.

-mfpxarg-in-gregs

Для числовых аргументов с плавающей точкой используется обычная вызывающая последовательность. Эта опция установлена по умолчанию.

Также см. опцию `-mfpxarg-in-fpregs`.

-mfull-fp-blocks

Генерирует полноразмерные блоки данных с плавающей точкой, использующие минимальный объем дополнительной памяти, рекомендуемый IBM. Эта опция установлена по умолчанию.

Также см. опцию `-mmimum-fp-blocks`.

-mhc-struct-return

Для передачи структур длиной более одного слова используется память, а не регистры. Эта опция обеспечивает совместимость с компилятором "MetaWare HighC"

(hc). Для обеспечения совместимости с компилятором "Portable C Compiler" (pcc) используется опция **-fpcc-struct-return**.

-min-line-mul

Использует расширяемую подстановкой последовательность инструкций для выполнения умножения целых чисел. Эта опция действует по умолчанию.

Также см. опцию **-mcall-lib-mul**.

-mminimum-fp-blocks

Указывает не включать в блоки данных с плавающей точкой область дополнительной памяти. Это приводит к генерации кода меньшего размера, но выполняться он будет медленнее, поскольку выделение дополнительной памяти при этом будет осуществляться динамически.

Также см. опцию **-mfull-fp-blocks**.

-mnohc-struct-return

Возвращает некоторые структуры длиной более одного слова при необходимости в регистрах. Эта опция установлена по умолчанию. Для обеспечения совместимости с компиляторами IBM необходимо использовать опцию **-fpcc-struct-return** или **-mhcc-struct-return**.

Опции для платформ S/390 и zSeries

Ниже перечислены опции, определенные для архитектур S/390 и zSeries.

-m31

Генерирует код для машинного интерфейса прикладных программ (EABI) S/390, совместимый с операционной системой Linux. Для архитектур **s390** по умолчанию используется опция **-m31**, а для архитектур **s390x** — опция **-m64**.

-m64

Генерирует код для машинного интерфейса прикладных программ (EABI) zSeries, совместимый с операционной системой Linux. Это позволяет компилятору, в частности, генерировать 64-битные инструкции. Для архитектур **s390** по умолчанию используется опция **-m31**, а для архитектур **s390x** — опция **-m64**.

-mbackchain

Генерирует код, который поддерживает явным образом обратную связь с кадром стека вызывающей функции, что бывает необходимо для отладки. Эта опция используется по умолчанию. Ее действие можно отменить опцией **-mnobackchain**.

-mhard-float

Использует аппаратные инструкции и регистры для операций над числами с плавающей точкой. Компилятор генерирует инструкции, совместимые со стандартами IEEE. Опция **-mhard-float** используется по умолчанию.

Также см. опцию **-msoft-float**.

-mdebug

При компиляции генерирует дополнительную отладочную информацию. По умолчанию используется опция **-mno-debug**, отключающая выработку отладочной информации.

-mmvcl e

Генерирует код, использующий для выполнения перемещения блоков инструкцию **mvcl e**. По умолчанию установлена опция **-mno-mvcl e**, и для перемещения блоков используется цикл **mvc**.

-msmall-exec

Генерирует код, использующий для вызова подпрограмм инструкцию **bras**. Эта опция работает надежно, только если полный размер исполняемого файла не превышает 64 Кбайт. По умолчанию установлена опция **-mno-small-exec**, и для вызова подпрограмм используется цикл **basr**. В этом случае полный размер исполняемого файла не будет иметь значения.

-msoft-float

Указывает, что для выполнения операций над числами с плавающей точкой не должны использоваться аппаратные инструкции и регистры. Для выполнения таких операций будут использоваться функции библиотеки **libgcc.a**.

Также см. **-mhard-float**.

Опции для платформы SH

Ниже перечислены опции, определенные для реализаций архитектуры SH.

-m1

Генерирует код для реализации SH1.

-m2

Генерирует код для реализации SH2.

-m3

Генерирует код для реализации SH3.

-m3e

Генерирует код для реализации SH3e.

-m4-nofpu

Генерирует код для реализации SH4 без использования аппаратной поддержки математических операций с плавающей точкой.

-m4-single-only

Генерирует код для реализации SH4 с блоком операций с плавающей точкой, поддерживающим только арифметические операции обычной точности.

-m4-single

Генерирует код для реализации SH4, предполагающий, что устройство для выполнения операций с плавающей точкой по умолчанию работает в режиме арифметических операций обычной точности.

-m4

Генерирует код для реализации SH4.

-mb

Генерирует код для процессора, работающего в режиме обратного порядка байтов (big endian).

Также см. опцию **-ml**.

-mbigtable

Генерирует 32-битные смещения для таблиц **switch**. По умолчанию используются 16-битные смещения.

-mdalign

Применяет 64-битное выравнивание данных типа **double**. Это изменяет соглашения о вызовах функций, вследствие чего некоторые функции стандартной библиотеки языка C могут не работать, если их предварительно не перекомпилировать с опцией **-mdalign**.

-mfmovd

Разрешает использование инструкции **fmovd**.

-mhitachi

Компилирует программу в соответствии с соглашением о вызовах функций, определенным для Hitachi.

Также см. опцию **-mnomacsav**.

-mieee

Приводит код в соответствие с требованиями стандартов IEEE для операций с плавающей точкой.

-misize

В коде ассемблера приводит размеры и расположение инструкций.

-ml

Генерирует код для процессора, работающего в режиме прямого порядка байтов (little endian mode).

Также см. опцию **-mb**.

-mnomacsavе

Указывает, что данные регистра MAC при вызове стираются, даже несмотря на применение опции **-mhitachi**.

-mpadstruct

Это устаревшая опция. Она дополняла структуры до размера, кратного четырем байтам, что несовместимо с машинным интерфейсом прикладных программ SH.

-mprefergot

При генерации перемещаемого кода вырабатывает вызовы функций с использованием глобальной таблицы смещений (Global Offset Table) вместо таблицы связывания процедур (Procedure Linkage Table).

-mrelax

В процессе компоновки сокращает дистанцию некоторых адресов. Опция **-mrelax** автоматически устанавливает опцию компоновщика **-relax**.

-mspace

Оптимизирует размер кода за счет скорости его выполнения. Эта опция устанавливается автоматически при использовании опции **-Os**.

-musermode

Генерирует вызов библиотечной функции для указания на нарушение инструкций в кэше после исправления "трамплина" (trampoline). Опция **-musermode** используется по умолчанию для конфигурации **sh-*-linux***.

Вызов этой библиотечной функции не имеет возможности выполнения записи в любом месте адресуемого пространства памяти.

Опции для платформы SPARC

Ниже перечислены опции, определенные для процессора Sun Microsystems SPARC.

-m32

Эта опция для процессора SPARC V9 в 64-битной среде устанавливает длину данных типов **int** и **long**, а также длину указателей равной 32 бита.

-m64

Эта опция для процессора SPARC V9 в 64-битной среде устанавливает длину данных типа **int** равной 32 бита, а длину данных типа **long** и указателей равной 64 бита.

-mapp-reg>

Генерирует код, использующий регистры 2–4, которые в машинном интерфейсе прикладных программ SPARC SVR4 зарезервированы для приложений. Опция **-mapp-reg>** используется по умолчанию.

Чтобы код был полностью совместим с машинным интерфейсом прикладных программ SPARC SVR4, необходимо установить опцию `-mno-app-reg`. (Это несколько снижает скорость работы.) Библиотеки и системное программное обеспечение должны компилироваться с установленной опцией `-mno-app-reg`.

`-mcmodel=setting`

Эта опция для процессора SPARC V9 в 64-битной среде генерирует код указанной модели. Допустимые значения параметра *setting* приведены в таблице 21.7.

Таблица 21.7. Модели кода процессоров SPARC V9

Значение параметра <i>setting</i>	Модель кода
<code>medlow</code>	Модель кода Medium/Low означает, что программа должна компоноваться в младших 32 битах адресного пространства. Длина указателей составляет 64 бита. Программы могут компоноваться как статически, так и динамически.
<code>medmid</code>	Модель кода Medium/Middle означает, что программа должна компоноваться в младших 44 битах адресного пространства, размер текстового сегмента не должен превышать 2 Гбайт, а размер сегмента данных должен отличаться от размера текстового сегмента не более чем на 2 Гбайт. Длина указателей составляет 64 бита.
<code>medany</code>	Модель кода Medium/Anywhere означает, что программа может компоноваться в любом месте адресного пространства, размер текстового сегмента не должен превышать 2 Гбайт, а размер сегмента данных должен отличаться от размера текстового сегмента не более чем на 2 Гбайт. Длина указателей составляет 64 бита.
<code>embmedany</code>	Модель кода Medium/Anywhere для встраиваемых (embedded) систем предполагает наличие текстового сегмента с 32-битной адресацией и сегмента данных, которые могут начинаться в любом месте (определяется при компоновке). Регистр <code>%g4</code> содержит указатель на начало сегмента данных. Длина указателей составляет 64 бита. PIC не поддерживается.

`-mbroken-saverestore`

Эта опция генерирует код для процессора SPARCLET, не использующий нетривиальные формы инструкций `save` и `restore`.

Причина существования этой опции заключается в том, что ранние версии процессора SPARCLET не могли корректно обрабатывать инструкции `save` и `restore` с аргументами, хотя инструкции без аргументов обрабатывались без ошибок. Инструкция `save` при использовании ее без аргумента увеличивает на единицу значение указателя текущего окна, но не выделяет новый кадр стека, поскольку предполагается, что обработчик системного прерывания переполнения окна корректно обработает эту ситуацию.

`-mcrti=type`

Набор инструкций, набор регистров и параметры планирования инструкций устанавливаются в соответствии с выбранным типом *type*. Параметр *type* может

принимать следующие значения: `v7`, `cypress`, `v8`, `supersparc`, `sparclite`, `sparclite86x`, `f930`, `f934`, `sparclet`, `tsc701`, `v9` и `ultrasparc`.

Для значений, которые выбирают только базовую архитектуру, а не конкретную реализацию, используются установленные по умолчанию параметры планирования инструкций.

В таблице 21.8 приведены базовые архитектуры и поддерживающие их реализации.

Таблица 21.8. Поддержка базовых архитектур различными реализациями процессоров

Архитектура	Реализации
<code>v7</code>	<code>cypress</code>
<code>v8</code>	<code>supersparc</code> , <code>hypersparc</code>
<code>sparclite</code>	<code>f930</code> , <code>f934</code> , <code>sparclite86x</code>
<code>sparclet</code>	<code>tsc701</code>
<code>v9</code>	<code>ultrasparc</code>

-mcypress

Эта опция используется по умолчанию. Компилятор оптимизирует параметры для чипа Cypress CY7C602, используемого в сериях SparcStation/SparcServer 3xx. Эта опция также используется для более старых серий SparcStation 1, 2, IPX и т.д.

Опция `-mcypress` считается устаревшей и в следующей версии компилятора будет удалена.

-mfaster-structs

Опция `-mfaster-structs` предполагает, что структуры выравниваются по границе 8 байт. Это позволяет для копирования структур использовать пары инструкций `ldd` и `std`. В противном случае требуется вдвое большее количество пар инструкций `ld` и `st`.

Однако использование пар инструкций `ldd` и `std` нарушает требования машинного интерфейса прикладных программ процессоров SPARC. Поэтому опцию `-mfaster-structs` можно использовать только для тех систем, которые допускают отклонение от правил стандартного машинного интерфейса прикладных программ (ABI).

По умолчанию используется опция `-mno-faster-structs`.

-mflat

Эта опция указывает, что компилятор не должен генерировать инструкции `save` и `restore`, а должен вместо них использовать соглашения о вызовах "одноуровневые вызовы" и "одно окно регистров". По умолчанию используется опция `-mno-flat`.

Такая модель использует регистр `%i7` в качестве указателя кадра и совместима со стандартной моделью окна регистров. Коды, генерируемые с использованием этих

двух моделей, могут работать совместно. Локальные и входные регистры (0–5) при необходимости будут записываться в память.

При указании опции `-mno-flat` компилятор генерирует инструкции `save` и `restore` (кроме функций, имеющих вложенные вызовы), это является обычным режимом.

-mfpu

Опция используется по умолчанию. Она предполагает генерацию кода, содержащего аппаратные инструкции математических операций над числами с плавающей точкой.

Для генерации кода, использующего функции библиотек для выполнения операций с числами с плавающей точкой следует использовать опцию `-mno-fpu`. Библиотеки функций эмуляции операций с плавающей точкой для платформ SPARC не предусмотрены. Могут использоваться библиотеки компиляторов, комплектных предназначаемым машинам, но их нельзя использовать при кросс-компиляции. Программная поддержка операций с плавающей точкой поддерживается только для конфигураций `sparc-*-*-aout` и `sparclite-*-*-*`.

Опцию `-mfpu` допускается указывать в виде `-mhard-float`.

Также см. опцию `-msoft-float`.

-mhard-float

Идентична опции `-mfpu`.

-mhard-quad-float

Код, генерируемый при установленной опции `-mhard-quad-float`, содержит аппаратные инструкции операций с плавающей точкой длиной в четыре слова (тип `long double`).

-mlittle-endian

Для процессоров SPARCLET и SPARC V9 в 64-битной среде эта опция генерирует код, предназначенный для выполнения процессором в режиме прямого порядка байтов (little endian mode).

-mlive-g0

Для процессора SPARCLET эта опция будет считать регистр `%g0` обычным регистром. При необходимости компилятор GCC будет очищать этот регистр, но всегда будет предполагаться, что регистр содержит значение "ноль".

-msoft-float

Аналогична опции `-mno-fpu`, но в генерируемом коде будет применено другое соглашение о вызовах функций. Это означает, что опция `-msoft-float` должна использоваться при компиляции всех модулей программы. Кроме того, в этом случае для обеспечения совместимости кода с этой же опцией должна компилироваться и библиотека `libgcc.a`, которая входит в состав компилятора GCC.

-msoft-quad-float

Эта опция используется по умолчанию. Код, генерируемый при установленной опции **-msoft-quad-float**, будет содержать вызовы библиотечных функций для выполнения операций над числовыми операндами с плавающей точкой длиной в четыре слова (тип **long double**). Вызываемые функции определены в машинном интерфейсе прикладных программ SPARC.

Опция **-mhard-quad-float** отсутствует по той причине, что не существует реализаций SPARC, которые бы обеспечивали аппаратную поддержку операций с числами с плавающей точкой длиной в четыре слова. Все реализации для выполнения таких операций вызывают обработчик системных прерываний, который эмулирует выполнение инструкции. Вследствие такой схемы работы (с использованием обработчика прерываний) код будет выполняться гораздо медленнее, чем при вызове подпрограмм машинного интерфейса прикладных программ (ABI). Именно по этой причине опция **-msoft-quad-float** установлена по умолчанию.

-msparclite

Эта опция выбирает вариант архитектуры SPARC. Если компилятор при его установке не был сконфигурирован специально для работы с Fujitsu SPARClite, то он будет генерировать код для варианта v7 архитектуры SPARC. Вариант SPARClite поддерживает инструкции целочисленного умножения, деления и сканирования (**ffs**), которые не поддерживаются в SPARC V7.

Опция **-msparclite** считается устаревшей и будет удалена в следующей версии компилятора. Вместо нее должна использоваться опция **-mcpri=sparclite**.

-mstack-bias

Для процессора SPARC V9 в 64-битной среде эта опция предполагает, что указатель стека (и указатель кадра, если таковой имеется) смешен на -2047, причем это значение должно быть добавлено при выполнении ссылок на кадр стека. По умолчанию используется опция **-mno-stack-bias**, при которой смещение -2047 отсутствует.

-msupersparc

Компилятор оптимизирует код для процессора SuperSparc, используемого в сеиях SparcStation 10, 1000 и 2000. Кроме того, эта опция разрешает использование полного набора инструкций SPARC V8.

Опция **-msupersparc** считается устаревшей и будет удалена в следующей версии компилятора. Вместо нее должна использоваться опция **-mcpri=supersparc**.

-mtune=type

Устанавливает параметры планирования инструкций для указанного типа **type** машины, но не устанавливает наборы инструкций и регистров, которые были бы установлены при использовании опции **-mcpri**.

Допустимые значения параметра **type** соответствуют значениям параметра **type** опции **-mcpri**. Пригодными значениями будут те, которые указывают конкретную реализацию процессора (т.е. имена, приведенные во втором столбце таблицы 21.8):

`cypress, supersparc, hypersparc, f970, f934, sparclite86x, tsc701, ultrasparc.`

-munaligned-doubles

Эта опция предполагает, что данные типа `double` имеют 8-байтное выравнивание только в том случае, если они содержатся в другом типе или имеют абсолютный адрес. Все остальные данные типа `double` считаются выровненными по границе 4 байт. Использование опции `-munaligned-doubles` позволяет избежать отдельных редко встречающихся проблем совместимости с кодом, генерированным другими компиляторами. Опция `-munaligned-doubles` не установлена по умолчанию, поскольку она приводит к снижению производительности кода, особенно кода операций с плавающей точкой.

По умолчанию используется опция `-mno-unaligned-doubles`, которая предполагает, что все данные типа `double` имеют 8-байтное выравнивание.

-mv8

Эта опция выбирает один из вариантов архитектуры SPARC. При ее установке генерируется код для SPARC V8. Единственным его отличием от кода для SPARC V7 является наличие инструкций для выполнения целочисленного умножения и деления, отсутствующих в SPARC V7.

Опция `-mv8` считается устаревшей и будет удалена в следующей версии компилятора. Вместо нее должна использоваться опция `-mcrci=v8`.

Опции для платформы System V

Ниже приведены некоторые дополнительные опции, доступные для платформы System V Release 4.

-G

Создает разделяемый объект. Вместо этой опции рекомендуется использовать опции `-symbolic` или `-shared`.

-Qn

Запрещает вставку в выходной файл директив `.ident`. Используется по умолчанию.

-Qy

С помощью директивы `.ident` указывает в выходном файле версии всех средств, используемых компилятором.

-Ym, directory

Осуществляет поиск препроцессора M4 в каталоге `directory`.

-YP, directories

Осуществляет поиск библиотек, указанных опциями `-lname`, только в списке заданных каталогов `directories`.

Опции для платформы TMS320C3x/C4x

Ниже перечислены опции, определенные для реализаций TMS320C3x/C4x.

-mbig

Генерирует код для модели большой памяти. Модель большой памяти используется по умолчанию и требует перезагрузки регистра DP для каждого прямого обращения к памяти.

Также см. опцию **-msmall**.

-mbig-memory

Идентична опции **-mbig**.

-mbk

Разрешает запись целочисленных операндов в счетчик блоков BK. Действие опции **-mbk** можно отменить опцией **-mno-bk**.

-mcpri=type

Устанавливает набор инструкций, набор регистров и параметры планирования инструкций для указанного типа машины. Параметр **type** может принимать следующие значения: **c30**, **c31**, **c32**, **c40** и **c44**. По умолчанию используется **c40**, в соответствии с которым генерируется код для TMS320C40.

-mdb

Разрешает генерацию кода, использующего инструкции декремента и ветвления **dbcond()**. Эта опция установлена по умолчанию для реализаций C4x. Ее действие можно отменить с помощью опции **-mno-db**.

Опция **-mdb** не используется по умолчанию для реализаций C3x, поскольку максимальное количество итераций для C3x составляет $2^{23} + 1$. Обратите внимание, что компилятор GCC пытается выполнять цикл в направлении уменьшения переменной цикла, что позволяет использоваться инструкции декремента и ветвления, но изменения порядка выполнения цикла не происходит, если в цикле содержится больше одной ссылки на ячейки памяти. Как следствие, в случае, когда не может использоваться инструкция **RPTB**, цикл с уменьшением переменной цикла выполняется несколько быстрее.

-mdp-isr-reload

При входе в программу обработки прерываний содержимое регистра DP сохраняется и инициализируется начальным адресом раздела данных. Затем содержимое регистра DP восстанавливается при выходе из программы обработки прерываний. Как правило, такая схема работы требуется только тогда, когда модель малой памяти нарушена за счет изменения регистра DP.

-mfast-fix

Отключает генерацию дополнительного кода, необходимого для исправления результатов работы инструкции **FIX**. По умолчанию установлена опция **-mno-fast-fix**.

Инструкция **FIX** реализаций C3x/C4x, предназначенная для преобразования числа с плавающей точкой в целое число, выбирает ближайшее целое меньшее или равное числу с плавающей точкой, а не просто ближайшее целое число. Поэтому при работе с отрицательными числами с плавающей точкой результат инструкции будет ошибочным. Для обнаружения и устранения этой ошибки необходим дополнительный код, генерируемый при установленной опции **-mfast-fix**.

-mloop-unsigned

Разрешает использование счетчика итераций без знака. По умолчанию установлена опция **-mno-loop-unsigned**.

Максимальное значение счетчика итераций при использовании инструкций **RPTS** и **RPTB** (и **DB** для реализаций C40) составляет $2^{31}+1$, поскольку эти инструкции для прекращения выполнения цикла проверяют счетчик итераций на значение, меньшее нуля. Если счетчик итераций представляет собой значение без знака, то существует возможность того, что максимальное значение $2^{31}+1$ будет превышено.

Также см. опции **-mrptb** и **-mrpts**.

-mtemparm

Генерирует код, который для передачи аргументов функциям использует регистры. По умолчанию там, где это возможно, аргументы передаются в регистрах, а не записываются в стек.

Также см. опцию **-mregparm**.

-mmpy1

Реализации C3x для операций целочисленного умножения используют 24-битную инструкцию **MPY1** и не вызывают используемые по умолчанию библиотечные функции, возвращающие результат длиной 32 бита. Действие опции **-mmpy1** можно отменить с помощью опции **-mno-mpy1**.

Если один из operandов операции является константой, умножение будет выполняться с помощью инструкций сдвига и сложения. Если опция **-mmpy1** для реализаций C3x не установлена, то операции возведения в квадрат выполняются подстановкой кода без вызова функции из библиотеки.

-mparallel-insns

Разрешает выработку параллельных инструкций. Эта опция устанавливается по умолчанию при установке опции **-O2**, она может быть отключена другой опцией **-mno-parallel-insns**.

Также см. опцию **-mparallel-mpy**.

-mparallel-mpy

Разрешает генерацию параллельных инструкций **MPY || ADD** и **MPY || SUB** при условии, что также установлена и опция **-mparallel-insns**. Эти инструкции имеют более жесткие требования к регистрам, вследствие чего они могут негативно сказаться на эффективности кода. Действие опции **-mparallel-mpy** можно отменить применением опции **-mno-parallel-mpy**.

-mparanoid

Идентична опции **-mregparam**.

-mregparm

Генерирует код, который для передачи функциям всех аргументов использует стек. По умолчанию аргументы по возможности передаются через регистры, вместо их записи в стек.

Также см. опцию **-mtempparam**.

-mrptb

Разрешает генерацию повторяющихся блоков последовательностей, использующих инструкцию **RPTB** для организации циклов, в которых переменная цикла изменяется в сторону уменьшения. Эта опция используется по умолчанию при установке опции **-O2**, ее можно отменить опцией **-fno-rptb**.

Конструкция **RPTB** используется только для циклов самого нижнего уровня, которые не вызывают функций и не выходят за границы кода цикла. Использование вложенных циклов на основе инструкций **RPTB** неэффективно, поскольку они дополнительно требуют записи и восстановления содержимого регистров **RC**, **RS** и **RE**.

Также см. опции **-mrpts** и **-mloop-unsigned**.

-mrpts=number

Разрешает использование инструкции повторения одной инструкции **RPTS**. По умолчанию установлена опция **-fno-rpts**.

Если блок повторения содержит одну инструкцию и переменная цикла гарантированно меньше значения параметра **number**, то компилятор GCC будет генерировать инструкцию **RPTS**, а не **RPTB**. Если параметр **number** не указан или если значение переменной цикла во время компиляции не может быть определено, то будет генерироваться инструкция **RPTS**.

Повторяемая инструкция, которая следует за инструкцией **RPTS**, может не загружаться при каждой итерации. Это освобождает шины процессора для операндов. Инструкция **RPTS** не используется по умолчанию, поскольку она блокирует прерывания.

Также см. опции **-mrptb** и **-mloop-unsigned**.

-msmall

Генерирует код для модели малой памяти. Модель малой памяти предполагает, что все данные могут быть помещены на страницу памяти размером 64 Кбайт. Во время выполнения программы содержимое регистра **DP** должно указывать на страницу памяти размером 64 Кбайт, содержащую данные разделов программы **.bss** и **.data**.

Также см. опцию **-mbig**.

-msmall-memory

Идентична опции **-msmall**.

-mti

Пытается генерировать синтаксис ассемблера, соответствующий правилам ассемблера TI (`asm30`). Кроме того, эта опция генерирует код, совместимый с интерфейсом прикладного программирования (API), используемого компилятором C TI для реализаций C3x. Например, данные типа `long double` передаются в структуре, а не через регистры с плавающей точкой.

Опции для платформы V850

Ниже перечислены опции, определенные для реализаций V850.

-mbig-switch

Генерирует код, совместимый с большими таблицами переключений (switch tables). Эта опция должна использоваться только при выдаче ассемблером или компоновщиком ошибок, касающихся наличия в таблице переключения ветвлений, выходящих за пределы допустимого диапазона.

-mep

Оптимизирует базовые элементы, использующие один и тот же индексный указатель четыре и более раз, за счет копирования указателя в регистр `ep` и использования более коротких инструкций `sld` и `sst`. Опция `-mep` устанавливается по умолчанию при оптимизации. Ее можно отключить опцией `-mno-ep`.

-mlong-calls

Считает все вызовы длинными. При длинном вызове компилятор всегда будет загружать адрес функции в регистр и вызывать ее косвенно с помощью указателя.

Действие опции `-mlong-calls` можно отменить с помощью опции `-mno-long-calls`.

-mprolog-function

Использует внешние функции для записи и восстановления содержимого регистров в прологе и эпилоге функции. Внешние функции выполняются медленнее, но используют меньший объем кода, если нескольким функциям необходимо сохранять один и тот же набор регистров. Опция `-mprolog-function` устанавливается по умолчанию при выполнении оптимизации, но ее можно отключить с помощью опции `-mno-prolog-function`.

-msda=number

Помещает статические или глобальные данные с размером не более `number` байт в область коротких данных, на которую указывает регистр `gp`. Размер области коротких данных может составлять до 64 Кбайт.

Также см. опции `-mtda` и `-mzda`.

-mspace

Пытается генерировать код минимально возможного размера за счет установки опций `-mep` и `-mprolog-function`.

-mtda=number

Помещает статические или глобальные данные с размером не более *number* байт в область очень коротких данных, на которую указывает регистр *er*. Размер области очень коротких данных может составлять в сумме не более 256 байт (128 байт для ссылок на байты).

Также см. опции **-msda** и **-mzda**.

-mv850

Указывает, что код должен генерироваться для реализации V850.

-mzda=number

Помещает статические или глобальные данные размером не более *number* байт в первые 32 Кбайта памяти.

Также см. опции **-msda** и **-mtda**.

Опции для платформы VAX

Ниже перечислены опции, определенные для процессора DEC VAX.

-mg

Генерирует код для формата чисел с плавающей точкой типа *g-format*, вместо *d-format*.

-mgnu

Генерирует инструкции переходов в соответствии с требованиями ассемблера проекта GNU.

-munix

Указывает не генерировать определенные инструкции переходов (в частности, *aobleg*), которые не могут обрабатываться ассемблером UNIX для процессора VAX при больших дистанциях обращений.

Опции для платформы Xstormy16

Ниже приведена опция, определенная для Xstormy16.

-msim

Выбирает файлы запуска и сценарии компоновщика, соответствующие имитатору (*simulator*).

*Полное
руководство*



Часть IV

Приложения



Приложение А

Генеральная Общественная Лицензия GNU

Kомпилятор GCC лицензирован в соответствии с условиями *Генеральной Общественной Лицензии GNU (GNU General Public License)*, известной под акронимом *GNU GPL* или просто *GPL*.

Тип лицензионных прав, предоставляемых GPL, называют "копилэфт" ("Copyleft" — в противоречие устоявшемуся понятию "Copyright"). В двух словах это означает, что каждый может копировать себе и пользоваться таким программным обеспечением, однако если оно включается в состав другого продукта, то и он должен быть лицензирован для свободного распространения. Таким образом, нельзя взять программу, полученную по GPL, и приобрести на нее исключительные права собственности. Хотя и нет никакого ограничения на использование GCC для производства программ, лицензируемых, как угодно автору. Объектный код и части программ, вырабатываемые в GCC, не нуждаются для своего применения в использовании программ, лицензированных в соответствии с GPL.

Кроме GPL есть еще и *Сокращенная Генеральная Общественная Лицензия (Lesser General Public License, LGPL)*. Раньше она была известна как "GPL для библиотек", но это название признано неточным. Она действительно применима для некоторых библиотек, но не для всех. LGPL разрешает использовать подпрограммы из библиотек для создания патентованных программ при условии, что библиотеки не компонуются в программу статически. В качестве примера такой библиотеки можно назвать версию GNU стандартной библиотеки C.

Дальше приводится перевод текста GPL. Он дает достаточно ясное и точное представление о правовых положениях лицензии и передает дух этого документа. В конце приложения приводится оригинальный текст лицензии GNU GPL на английском языке.

Перевод на русский язык Генеральной Общественной Лицензии GNU

ГЕНЕРАЛЬНАЯ ОБЩЕСТВЕННАЯ ЛИЦЕНЗИЯ GNU (GNU GENERAL PUBLIC LICENSE)

Версия 2, июнь 1991 г. Copyright (C) 1989, 1991 Free Software Foundation, Inc. 59 Temple Place, Suite 330, Boston, MA 02111-1307, USA.

(Перевод издательства "Диа Софт". 2003 год.)

Каждый вправе копировать и распространять экземпляры настоящей Лицензии, но изменять ее текст не разрешается.

ПРЕАМБУЛА

Большинство лицензий на программное обеспечение лишает вас прав на его распространение и внесение в него изменений. Генеральная Общественная Лицензия GNU (GNU General Public License), напротив, разработана с целью гарантировать полную свободу совместного использования и изменения свободно распространяемого программного обеспечения, т.е. чтобы обеспечить свободный доступ к таким программам для всех пользователей. Условия настоящей Генеральной Общественной Лицензии применяются к большей части программного обеспечения Ассоциации Свободно Распространяемых Программ (Free Software Foundation), а также к любому другому программному обеспечению по желанию его автора. (К некоторому программному обеспечению Ассоциации Свободно Распространяемых Программ применяются условия Генеральной Общественной Лицензии GNU Для Библиотек.) Вы также можете применять Стандартную Общественную Лицензию к разработанному вами программному обеспечению.

Говоря о свободном программном обеспечении, мы имеем в виду свободу распространения, а не безвозмездность. Настоящая Стандартная Общественная Лицензия разработана с целью гарантировать вам право распространять свободное программное обеспечение (при желании, за вознаграждение), право и возможность получения исходного текста программного обеспечения, право вносить изменения в программное обеспечение или использовать его части в новом свободном программном обеспечении, а также право знать, что вы имеете все вышеперечисленные права.

Чтобы защитить ваши права, мы вынуждены ввести ограничения, которые запрещают кому бы то ни было лишать вас этих прав или обращаться к вам с предложением отказаться от этих прав. Данные ограничения налагаются на вас определенные обязательства в случае, если вы распространяете или модифицируете программное обеспечение.

Например, если вы распространяете экземпляры такого программного обеспечения за плату или бесплатно, вы обязаны передать новым обладателям все права в том же объеме, в каком они принадлежат вам. Вы обязаны передать новым обладателям исходный текст программы или обеспечить возможность его получения ими. Вы также обязаны сообщить им об их правах, ознакомив с условиями настоящей Лицензии.

Защищая ваши права, мы, во-первых, оставляем за собой авторское право на программное обеспечение и, во-вторых, предлагаем настоящую Лицензию, которая дает

вам законное право воспроизводить, распространять и модифицировать программное обеспечение.

Кроме того, как для своей защиты, так и для защиты других авторов мы уведомляем всех, что на свободно распространяемое в соответствии с настоящей Лицензией программное обеспечение не предоставляется никаких гарантий. Все пользователи, которые приобрели программное обеспечение с внесенными в него третьими лицами изменениями, должны знать, что они получают не оригинал, и поэтому ошибки в работе программного обеспечения не должны отражаться на репутации автора его оригинала.

Наконец, программное обеспечение перестает быть свободным в случае, если лицо приобретает на него исключительные права. Недопустимо, чтобы лица, распространяющие свободное программное обеспечение, могли приобрести и зарегистрировать исключительные права на его использование. Чтобы избежать этого, мы заявляем, что обладатель должен либо лицензировать программное обеспечение с правами на его дальнейшее свободное использование всеми, либо не заявлять на него никаких прав вообще.

Ниже приводятся подробные условия воспроизведения, распространения и модификации программного обеспечения.

УСЛОВИЯ ВОСПРОИЗВЕДЕНИЯ, РАСПРОСТРАНЕНИЯ И МОДИФИКАЦИИ

0. Настоящая Лицензия применяется ко всем видам программного обеспечения или иному произведению, которое содержит указание правообладателя на его распространение в соответствии с условиями Генеральной Общественной Лицензии. В качестве "Программы" далее понимается любое программное обеспечение или иное произведение, удовлетворяющее этому условию. Под термином "основывающееся на Программе произведение" понимается как Программа, так и любой производный от нее продукт, соответствующий законодательству об авторском праве, т.е. произведение, включающее в себя Программу или ее часть как с внесенными в нее изменениями, так и без них, возможно, переведенную на другой язык. (Здесь и далее понятие "модификация" включает в себя перевод в самом широком смысле.) Дальнейшее содержание Лицензии адресуется к вам как к "Лицензиату", то есть приобретателю экземпляра Программы и всех передаваемых прав.

Действие настоящей Лицензии не распространяется на осуществление иных прав, кроме воспроизведения, распространения и модификации программного обеспечения. Не устанавливается ограничений на запуск Программы. Условия Лицензии распространяются на результаты применения Программы только в том случае, если их содержимое составляет основывающееся на Программе произведение (независимо от того, было ли такое произведение создано в результате запуска Программы или выработано из ее содержимого). Это зависит от того, какие функции выполняет Программа.

1. Лицензиат вправе изготавливать и распространять экземпляры исходного текста Программы в том виде, в каком он его получил, на любом носителе, при соблюдении следующих условий: на каждом экземпляре заметным образом помещено соответствующее предупреждение об авторских правах и уведомление об отказе от

гарантийных обязательств; оставлены без изменений все уведомления, ссылающиеся на настоящую Лицензию, и уведомления об отсутствии гарантий; вместе с экземпляром Программы приобретателю передается копия настоящей Лицензии.

Лицензиат вправе взимать плату за передачу экземпляра Программы и также вправе, по своему усмотрению, оказывать услуги по гарантийной поддержке Программы за денежное или иное вознаграждение.

2. Лицензиат вправе модифицировать свой экземпляр или экземпляры Программы полностью или любую его часть. Данные действия Лицензиата влекут за собой создание им основывающегося на Программе произведения. Лицензиат вправе изготавливать и распространять экземпляры такого основывающегося на Программе произведения, или собственно экземпляры изменений в соответствии с пунктом 1 настоящей Лицензии при соблюдении следующих условий:

А) файлы, измененные Лицензиатом, должны приметным образом содержать заметки о том, что они были им изменены, а также дату внесения изменений;

Б) при распространении или публикации Лицензиатом любого произведения, которое содержит Программу или ее часть, или основывается на Программе или ее части, он обязан бесплатно передавать третьим лицам права на использование данного произведения в целом на условиях настоящей Лицензии;

В) в случае если модифицированная Программа при запуске читает команды в интерактивном режиме, Лицензиат обязан обеспечить в начале интерактивного режима вывод на экран дисплея или печатающее устройство сообщения, включающего в себя: знак охраны авторского права и уведомление об отсутствии гарантийных обязательств на Программу (или сообщение о таких обязательствах, если Лицензиат берет их на себя); указание на то, что пользователи вправе распространять экземпляры Программы в соответствии с условиями настоящей Лицензии, а также каким образом пользователь может ознакомиться с текстом настоящей Лицензии. (Иключение: если оригинальная Программа является интерактивной, но не выводит в своем обычном режиме работы сообщение такого рода, то вывод подобного сообщения произведением, основывающимся на этой Программе, в этом случае не обязателен.)

Перечисленные условия применяются к модифицированному произведению, основывающемуся на Программе, в целом. В случае если отдельные части данного произведения не являются производными от Программы, они являются результатом творческой деятельности и могут быть использованы как самостоятельное произведение; Лицензиат вправе распространять отдельно такое произведение на иных условиях. В случае если Лицензиат распространяет вышеуказанные части в составе основывающегося на Программе произведения, то условия настоящей Лицензии применяются к произведению в целом, при этом права, приобретаемые Лицензиатом в отношении Программы, передаются им получателям в отношении всего произведения, включая все его части, независимо от того, кто является их авторами.

Целью изложенных в пункте 2 правил не является заявление или оспаривание прав на произведение, созданное исключительно Лицензиатом. Целью этих правил является обеспечение как наших, так и ваших прав контролировать распространение отдельных и составных произведений, основывающихся на Программе.

Размещение произведения, которое не основывается на Программе, на одном устройстве для хранения информации или носителе вместе с Программой или ос-

новывающемся на Программе произведением не влечет за собой распространения условий настоящей Лицензии на него.

3. Лицензиат вправе воспроизводить и распространять экземпляры Программы или основывающееся на ней произведение, в соответствии с пунктом 2 настоящей Лицензии, в виде объектного кода или в исполняемой форме в соответствии с условиями п. 1 и 2 настоящей Лицензии при соблюдении одного из перечисленных ниже условий:

А) к экземпляру должен прилагаться соответствующий полный исходный текст в машиночитаемой форме, который должен распространяться в соответствии с условиями п.п. 1 и 2 настоящей Лицензии на носителе, обычно используемом для передачи программного обеспечения, либо

Б) к экземпляру должно прилагаться действительное в течение не менее трех лет предложение в письменной форме к любому третьему лицу передать за плату, не превышающую стоимость осуществления собственно передачи, экземпляр соответствующего полного исходного текста в машиночитаемой форме в соответствии с условиями п.п. 1 и 2 настоящей Лицензии на носителе, обычно используемом для передачи программного обеспечения, либо

В) к экземпляру должна прилагаться полученная Лицензиатом информация о предложении, в соответствии с которым можно получить соответствующий исходный текст. (Данное положение применяется исключительно в том случае, если Лицензиат осуществляет некоммерческое распространение программы, и при этом программа была получена самим Лицензиатом в виде объектного кода или в исполняемой форме и сопровождалась предложением, соответствующим условиям пп. Б п. 3 настоящей Лицензии.)

Под исходным текстом произведения понимается такая форма произведения, которая наиболее удобна для внесения изменений. Под полным исходным текстом исполняемого произведения понимается исходный текст всех составляющих произведение модулей, а также всех файлов, связанных с описанием интерфейса, и сценариев, предназначенных для управления Компиляцией и установкой исполняемого произведения. Однако, в качестве особого исключения, распространяемый исходный текст может не включать того, что обычно распространяется (в виде исходного текста или в двоичной форме) с основными компонентами (компилятор, ядро и т.д.) операционной системы, в которой работает исполняемое произведение, за исключением случаев, когда исполняемое произведение сопровождается таким компонентом.

В случае если произведение в виде объектного кода или в исполняемой форме распространяется путем предоставления доступа для копирования его из указанного места, то обеспечение такого же доступа для копирования исходного текста из этого же места считается предоставлением исходного текста, даже если третьи лица при этом не обязаны копировать исходный текст вместе с объектным кодом или исполняемой формой произведения.

4. Лицензиат вправе воспроизводить, модифицировать, распространять или передавать права на использование Программы только на условиях настоящей Лицензии. Любое воспроизведение, модификация, распространение или передача прав на

иных условиях являются недействительными и автоматически ведут к расторжению настоящей Лицензии и прекращению всех прав Лицензиата, предоставленных ему настоящей Лицензией. При этом права третьих лиц, которым Лицензиат в соответствии с настоящей Лицензией передал экземпляры Программы или права на нее, сохраняются в силе при условии полного соблюдения ими настоящей Лицензии.

5. Лицензиат не обязан принимать условия Лицензии, поскольку он ее не подписывает. Однако только настоящая Лицензия предоставляет право распространять или модифицировать Программу или произведение, производное от Программы. Подобные действия нарушают действующее законодательство, если они не осуществляются в соответствии с настоящей Лицензией. Таким образом, если Лицензиат внес изменения или осуществил распространение экземпляров Программы (или основанного на Программе произведения), он тем самым подтвердил свое присоединение к настоящей Лицензии в полном ее объеме, включая условия, определяющие порядок воспроизведения, распространения и модификации Программы или основанного на Программе произведения.

6. При распространении экземпляров Программы или основанного на Программе произведения первоначальный лицензиат автоматически передает приобретателю такого экземпляра право воспроизводить, распространять и модифицировать Программу в соответствии с условиями настоящей Лицензии. Лицензиат не вправе ограничивать каким-либо способом осуществление приобретателями полученных ими прав. Лицензиат не несет ответственности за несоблюдение условий настоящей Лицензии третьими лицами.

7. Лицензиат не освобождается от исполнения обязательств в соответствии с настоящей Лицензией в случае, если в результате решения суда или обвинения в нарушении авторских прав или в связи с наступлением иных обстоятельств на Лицензиата налагаются ограничения (приговор суда, взятые обязательства, либо какие-то иные), противоречащие настоящей Лицензии. В этом случае Лицензиат не вправе распространять экземпляры Программы, если он не способен одновременно выполнять условия настоящей Лицензии и возложенные на него выше упомянутые обязательства. Например, если по условиям лицензионного соглашения получателям не может быть предоставлено право бесплатного распространения экземпляров Программы, которые они приобрели напрямую или опосредованно у Лицензиата, то в этом случае Лицензиат обязан отказаться от распространения экземпляров Программы.

Если любое из положений этого пункта при наступлении конкретных обстоятельств может быть признано недействительным или неприменимым, настоящий пункт применяется без такого положения. Настоящий пункт применяется полностью при прекращении действия вышеуказанных обстоятельств или их отсутствии.

Целью данного пункта не является принуждение Лицензиата к нарушению патента или заявления на иные права собственности или к оспариванию действительности такого заявления. Единственной целью данного пункта является защита целостности системы распространения свободного программного обеспечения, существующей благодаря практике общественного лицензирования. Многие люди внесли свой щедрый вклад в создание большого количества программ, распростра-

няемых через эту систему в надежде на ее последовательное и долгосрочное применение. Лицензиат не вправе вынуждать автора распространять программное обеспечение через данную систему. Право выбора любой системы распространения программного обеспечения является исключительно делом добной воли авторов и получателей.

Настоящий раздел имеет целью ясно показать, на чем основываются последующие положения настоящей Лицензии.

8. В том случае, если распространение или использование Программы в отдельных государствах ограничено соглашениями в области патентных или авторских прав, первоначальный правообладатель, распространяющий Программу на условиях настоящей Лицензии, вправе устанавливать ограничение территории распространения Программы, указывая только те государства, на территории которых допускается распространение Программы без ограничений, обусловленных такими соглашениями. В этом случае такое указание в отношении распространения на территории определенных государств признается одним из условий настоящей Лицензии.

9. Ассоциация Свободно Распространяемых Программ (Free Software Foundation) может пересматривать и публиковать в данном случае исправленные или новые версии настоящей Генеральной Общественной Лицензии. Такие версии могут быть дополнены различными нормами, регулирующими правоотношения, которые возникли после опубликования предыдущих версий, однако в них будут сохранены и основные принципы и дух настоящей версий.

Каждой версии присваивается свой собственный номер. Если указано, что Программа распространяется в соответствии с определенной версией, т.е. указан ее номер, или любой более поздней версией настоящей Лицензии, Лицензиат вправе присоединиться к любой из последующих версий Лицензии, опубликованных Ассоциацией Свободно Распространяемых Программ. Если Программа не содержит такого указания на номер версии Лицензии, Лицензиат вправе присоединиться к любой из версий Лицензии, опубликованных когда-либо Ассоциацией Свободно Распространяемых Программ.

10. В случае если Лицензиат намерен включить часть Программы в другое свободно распространяемое программное обеспечение, которое распространяется на иных условиях, чем изложенных в настоящей Лицензии, ему следует испросить письменное разрешение на это у автора программного обеспечения. Разрешение в отношении Программ, права на которые принадлежат Ассоциации Свободно Распространяемых Программ (Free Software Foundation), следует испрашивать у Ассоциации Свободно Распространяемых Программ. В некоторых случаях Ассоциация Свободно Распространяемых Программ делает исключения. При принятии решения Ассоциация Свободно Распространяемых Программ будет руководствоваться двумя целями: сохранение статуса свободно распространяемого для любого произведения, основывающегося на свободно распространяемом программном обеспечении Ассоциации Свободно Распространяемых Программ, и обеспечение условий для наиболее широкого совместного использования программного обеспечения.

ОТСУТСТВИЕ ГАРАНТИЙНЫХ ОБЯЗАТЕЛЬСТВ

11. ПОСКОЛЬКУ ПРОГРАММА РАСПРОСТРАНЯЕТСЯ БЕСПЛАТНО, ГАРАНТИИ НА НЕЕ НЕ ПРЕДОСТАВЛЯЮТСЯ. ПРОГРАММА ПОСТАВЛЯЕТСЯ НА УСЛОВИЯХ "КАК ЕСТЬ". ЕСЛИ ИНОЕ НЕ УКАЗАНО ПИСЬМЕННО, АВТОР ЛИБО ИНОЙ ПРАВООБЛАДАТЕЛЬ НЕ ПРИНИМАЕТ НА СЕБЯ НИКАКИХ ГАРАНТИЙНЫХ ОБЯЗАТЕЛЬСТВ В ОТНОШЕНИИ ПРОГРАММЫ, КАК ЯВНО ВЫРАЖЕННЫХ, ТАК И ПОДРАЗУМЕВАЕМЫХ, В ТОМ ЧИСЛЕ ПОДРАЗУМЕВАЕМУЮ ОТВЕТСТВЕННОСТЬ ЗА ТОВАРНОЕ СОСТОЯНИЕ ПРИ ПРОДАЖЕ И ПРИГОДНОСТЬ ДЛЯ ИСПОЛЬЗОВАНИЯ В КОНКРЕТНЫХ ЦЕЛЯХ, А ТАКЖЕ ЛЮБЫЕ ИНЫЕ ГАРАНТИИ. ВСЕ ИЗДЕРЖКИ, СВЯЗАННЫЕ С КАЧЕСТВОМ И ПРОИЗВОДИТЕЛЬНОСТЬЮ ПРОГРАММЫ, НЕСЕТ ЛИЦЕНЗИАТ. В СЛУЧАЕ ЕСЛИ В ПРОГРАММЕ БУДУТ ОБНАРУЖЕНЫ НЕДОСТАТКИ, ТО ВСЕ РАСХОДЫ, СВЯЗАННЫЕ С ТЕХНИЧЕСКИМ ОБСЛУЖИВАНИЕМ, ВОССТАНОВЛЕНИЕМ ИЛИ ИСПРАВЛЕНИЕМ ПРОГРАММЫ, ТАКЖЕ НЕСЕТ ЛИЦЕНЗИАТ.

12. ЕСЛИ ИНОЕ НЕ ПРЕДУСМОТРЕНО ПРИМЕНЯЕМЫМИ ЗАКОНАМИ ИЛИ НЕ СОГЛАСОВАНО СТОРОНАМИ В ДОГОВОРЕ ПИСЬМЕННО, АВТОР ЛИБО ИНОЙ ПРАВООБЛАДАТЕЛЬ, КОТОРЫЙ МОДИФИЦИРУЕТ И/ИЛИ РАСПРОСТРАНЯЕТ ПРОГРАММУ НА УСЛОВИЯХ НАСТОЯЩЕЙ ЛИЦЕНЗИИ, НЕ НЕСЕТ ОТВЕТСТВЕННОСТИ ПЕРЕД ЛИЦЕНЗИАТОМ ЗА ЕГО УБЫТКИ, ВКЛЮЧАЯ ОБЩИЕ, ЧАСТИЧНЫЕ, СЛУЧАЙНЫЕ, ВОЗНИКШИЕ ВПОСЛЕДСТВИИ, ПРЕДВИДИМЫЕ, КОСВЕННЫЕ УБЫТКИ (В ТОМ ЧИСЛЕ И УТРАТЫ ИЛИ ИСКАЖЕНИЯ ИНФОРМАЦИИ, И ПОТЕРИ ЛИЦЕНЗИАТА ИЛИ ТРЕТЬИХ ЛИЦ, СВЯЗАННЫЕ С НЕСОВМЕСТИМОСТЬЮ ПРОГРАММЫ С ЛЮБОЙ ДРУГОЙ ПРОГРАММОЙ, И ИНОЙ УЩЕРБ). АВТОР ЛИБО ИНОЙ ПРАВООБЛАДАТЕЛЬ В СООТВЕТСТВИИ С НАСТОЯЩИМ ПОЛОЖЕНИЕМ НЕ НЕСУТ ОТВЕТСТВЕННОСТИ ДАЖЕ В ТОМ СЛУЧАЕ, ЕСЛИ ОНИ БЫЛИ ПРЕДУПРЕЖДЕНЫ О ВОЗМОЖНОСТИ ВОЗНИКНОВЕНИЯ ТАКОГО УЩЕРБА.

ПОРЯДОК ПРИМЕНЕНИЯ УСЛОВИЙ ГЕНЕРАЛЬНОЙ ОБЩЕСТВЕННОЙ ЛИЦЕНЗИИ К СОЗДАННОЙ ВАМИ ПРОГРАММЕ

Если вы создали новую программу и хотите, чтобы она имела наибольшую доступность для общественности, то лучший способ этого достичь — сделать вашу программу свободно распространяемой, которую каждый сможет распространять и вносить в нее изменения в соответствии с условиями настоящей Лицензии.

В этих целях Программа должна содержать приведенное ниже уведомление. Наиболее правильным будет поместить его в начале каждого файла с исходным текстом для наиболее ясного уведомления об отсутствии гарантийных обязательств. Каждый файл должен, по крайней мере, содержать знак охраны авторского права и указание, где можно ознакомиться с полным текстом уведомления.

*[одна строка с наименованием Программы и кратким описанием ее назначения]
Copyright (C) [год опубликования программы] [имя (наименование) автора]*

Данная программа является свободным программным обеспечением. Вы вправе распространять и модифицировать ее в соответствии с условиями версии 2 либо по вашему выбору более поздней версии Генеральной Общественной Лицензии GNU (GNU General Public License), опубликованной Ассоциацией Свободно Распространяемых Программ (Free Software Foundation).

Мы предоставляем эту программу в надежде на то, что она будет вам полезной, однако НЕ ДАЕМ НИКАКИХ ГАРАНТИЙНЫХ ОБЯЗАТЕЛЬСТВ, в том числе мы не несем ответственности за ТОВАРНОЕ СОСТОЯНИЕ и ПРИГОДНОСТЬ ДЛЯ ИСПОЛЬЗОВАНИЯ В КОНКРЕТНЫХ ЦЕЛЯХ. Для получения более подробной информации ознакомьтесь с Генеральной Общественной Лицензией GNU.

Вместе с данной программой вам предоставляется Генеральная Общественная Лицензия GNU. Если вы ее не получили, то сообщите об этом по адресу: Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307, USA.

Также укажите, как можно связаться с вами по электронной или обычной почте.

Сделайте так, чтобы при работе в интерактивном режиме ваша программа наиболее подходящим способом выводила короткое сообщение в соответствии с образцом:

Gnomovision версия 69, Copyright (C) [год опубликования программы] [имя (наименование) автора]

Gnomovision распространяется БЕЗ ВСЯКИХ ГАРАНТИЙ; чтобы ознакомиться с более подробной информацией, наберите "show w". Данная программа является свободно распространяемой, вы можете ее распространять при соблюдении требований общественной лицензии. Для получения более подробной информации, наберите "show c".

Здесь предполагается, что при вводе предлагаемых команд "show w" и "show c" на экран должны выводиться соответствующие пункты Стандартной Общественной Лицензии. Не обязательно использовать именно команды "show w" и "show c". В зависимости от функций программы, можно использовать другие команды, действия манипулятором или пункты меню программы.

Если вы создали программу в порядке выполнения служебных обязанностей или учебного задания, то вам следует получить письменный отказ от исключительных прав на использование созданной вами программы у своей организации, нанимателя или учебного заведения. Нижеприведенный текст можно использовать как образец:

Корпорация "Йойодайн" (Yoyodyne, Inc.) настоящим отказывается от исключительных прав на программу "Gnomovision" (делающую волшебные штуки с компьютерами) написанную Джеймсом Хакером (James Hacker), и передает все исключительные права на использование указанной программы ее автору, Джеймсу Хакеру (James Hacker).

[подпись господина Тай-Куна],

1 апреля 1989 г.

Тай-Кун (Ty Coon), Вице-президент.

Генеральная Общественная Лицензия GNU не позволяет получателям включать вашу программу в программы, использование которых ограничено имущественными правами их правообладателей. Если ваша программа является библиотекой подпрограмм, то, вероятно, более полезно будет разрешить компоновку программ, использование которых ограничено их правообладателями, с вашей библиотекой. В этом случае следует использовать Генеральную Общественную Лицензию GNU Для Библиотек вместо настоящей Лицензии.

Оригинальный текст Генеральной Общественной Лицензии GNU

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc.

59 Temple Place, Suite 330, Boston, MA 02111-1307, USA

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

PREAMBLE

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

HOW TO APPLY THESE TERMS TO YOUR NEW PROGRAMS

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.>

Copyright (C) <year> <name of author>

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

*Gnomovision version 69, Copyright (C) year name of author Gnomovision comes with
ABSOLUTELY NO WARRANTY; for details type 'show w'. This is free software, and you
are welcome to redistribute it under certain conditions; type 'show c' for details.*

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than 'show w' and 'show c'; they could even be mouse-clicks or menu items whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

*Yoyodyne, Inc., hereby disclaims all copyright interest in the program
'Gnomovision' (which makes passes at compilers) written by James Hacker.
<signature of Ty Coon>, 1 April 1989
Ty Coon, President of Vice*

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.



Приложение Б

Переменные окружения

Cуществует ряд переменных окружения, влияющих на способ компиляции программ в GCC. Эти переменные окружения действуют аналогично соответствующим им опциям командной строки, но их приоритет, как правило, ниже.

Несколько переменных окружения содержат список путей доступа к файлам. При этом применяется тот же формат списка, который используется в системной переменной окружения `PATH`. Каталоги должны быть представлены полным путем доступа к ним. Имена путей доступа к каталогам в их списке разделяются особым символом, определяемым при инсталляции компилятора как `NAME_SEPARATOR`. В системах UNIX в качестве такого символа-разделителя используется двоеточие ":" (colon), в системах Windows — точка с запятой ";" (semicolon).

`C_INCLUDE_PATH`

Эта переменная окружения применяется при компиляции программ на языке C, она действует аналогично опции командной строки `-isystem`. Переменная `C_INCLUDE_PATH` содержит список каталогов для поиска включаемых по директиве `#include` заголовочных файлов. При применении в командной строке опции `-isystem` указанные опцией `C_INCLUDE_PATH` каталоги будут просматриваться в первую очередь.

Также см. `CPATH`, `CPLUS_INCLUDE_PATH` и `OBJC_INCLUDE_PATH`.

`COMPILER_PATH`

Эта переменная содержит список одного или более путей к каталогам. В этих каталогах компилятор проводит поиск своих подпрограмм в случае, если они не были найдены с помощью установки переменной `GCC_EXEC_PREFIX`.

Также см. переменные окружения `LIBRARY_PATH`, `GCC_EXEC_PREFIX` и опцию командной строки `-B`.

СPATH

Эта переменная окружения влияет на компиляцию программ на языках *C*, *C++* и *Objective-C*. Она действует аналогично опции командной строки **-I**. Переменная **СPATH** содержит список одного или более путей для поиска включаемых заголовочных файлов. Каталоги, указанные в опции **-I**, просматриваются раньше каталогов из списка переменной **СPATH**.

Также см. **C_INCLUDE_PATH**, **CPLUS_INCLUDE_PATH** и **OBJC_INCLUDE_PATH**.

CPLUS_INCLUDE_PATH

Применяется при компиляции программ на языке *C++*. Она действует аналогично опции командной строки **-isystem**. Переменная **CPLUS_INCLUDE_PATH** содержит список каталогов для поиска включаемых по директиве `#include` заголовочных файлов. При применении в командной строке опции **-isystem** указанные опцией каталоги будут просматриваться в первую очередь.

Также см. **СPATH**, **C_INCLUDE_PATH** и **OBJC_INCLUDE_PATH**.

DEPENDENCIES_OUTPUT

При установке значения этой переменной окружения именем файла препроцессор записывает в этот файл правила зависимостей компоновки. При этом в данный файл не включаются зависимости, связанные с системными заголовочными файлами.

Если переменная окружения содержит только имя файла, то оно используется как имя выходного файла зависимостей, а для названия набора правил используется имя исходного файла. Когда же переменная **DEPENDENCIES_OUTPUT** содержит два имени, то второе применяется как имя цели, для которой создается отдельное правило.

Использование этой переменной окружения дает такой же результат, как и сочетание опций командной строки **-MM**, **-MF** и **-MT**. Также см. **SUNPRO_OUTPUT**.

GCC_EXEC_PREFIX

При назначении этой переменной окружения ее значение будет использоваться как префикс имен вызываемых компилятором подпрограмм. Например, если **GCC_EXEC_PREFIX** содержит строку **testver**, то при вызове асемблера вместо программы **as** вначале будет предпринята попытка поиска и запуска программы с именем **testveras**. В случае невозможности обнаружить такую программу компилятор выполнит вызов ассемблера по его обычному имени. Возможно использование символов наклонной черты для указания пути расположения.

По умолчанию переменная **GCC_EXEC_PREFIX** имеет значение **prefix/lib/gcc-lib**, где **prefix** — имя, которое было определено в скрипте **configure** при инсталляции компилятора. Значение переменной также используется при поиске стандартных компонуемых файлов, включаемых в вырабатываемую готовую программу.

При использовании в командной строке опции **-B** она замещает установку переменной окружения **GCC_EXEC_PREFIX**. См. также переменную окружения **COMPILER_PATH**.

LANG

Эта переменная окружения используется для указания компилятору набора кодировки, используемого для представления отдельных расширенных буквенных символов, текстовых строк и комментариев.

Рассмотрим применение нескольких вариантов кодировки текстовых знаков японского языка. Установка переменной `LANG` значением `C-JIS` указывает препроцессору, что мультибайтные коды буквенных символов следует интерпретировать как знаки набора JIS (Japanese Industrial Standard). Значение `C-SJIS` указывает на применение кодировки `SHIFT-JIS`, а значение `C-EUCJP` — кодировки Japanese EUC.

В случае если значение переменной `LANG` не установлено или при невозможности его распознать, для определения длины расширенных буквенных кодов применяется функция `mblen()`, а для конвертирования мультибайтных последовательностей в знаки расширенного набора — функция `mbtowc()`.

LC_ALL

Назначение этой переменной окружения замещает любые установки переменных `LC_MESSAGES` и `LC_CTYPE`.

LC_CTYPE

Переменная окружения `LC_CTYPE` указывает способ кодировки расширенных буквенных знаков, имеющих мультибайтное представление. Основное использование этой переменной состоит в определении границ строк. Это необходимо при декодировании буквенных символов расширенного набора в строках, выделенных кавычками или/и имеющих мультибайтные escape-последовательности. При неправильном определении длины представления расширенного символа возможна неправильная интерпретация последовательностей и кодов оставшейся части текстовой строки. Как пример: переменная `LC_CTYPE` может содержать значение `en_AU`, которое указывает на использование австралийского диалекта английского языка, или `es_MX` для мексиканского диалекта испанского языка. В случае если эта переменная окружения не установлена или ее значение не распознано, то по умолчанию применяется значение переменной `LANG`. Если переменная `LANG` также не установлена, то применяется стандартный английский набор компилятора языка `C`. См. также `LC_ALL`.

LC_MESSAGES

Эта переменная окружения указывает язык, используемый для выдачи компилятором диагностических сообщений. Например, значение переменной `en_AU` определяет использование для вывода сообщений австралийского диалекта английского языка, а значение `es_MX` — мексиканского диалекта испанского языка. Если значение этой переменной окружения не установлено или не распознано, то применяется установка переменной `LANG`. Если переменная `LANG` также не установлена, то используются стандартные сообщения компилятора `C`. См. также `LC_ALL`.

LD_LIBRARY_PATH

Переменная `LD_LIBRARY_PATH` не оказывает влияния на работу компилятора. Она применяется при выполнении программ. Эта переменная окружения предос-

тавляет список каталогов, в которых программы во время своего запуска и выполнения находят разделяемые библиотеки. Необходимость в установке этой переменной возникает при выполнении программ, использующих особые версии разделяемых библиотек, расположение которых отличается от расположения библиотек, обычно используемых при компиляции программ.

LD_RUN_PATH

Переменная **LD_RUN_PATH** не оказывает влияния на работу компилятора, она используется программами во время их выполнения. Запущенная программа использует значение этой переменной для поиска файла, содержащего имена символов и соответствующие им адреса. При этом применяется неперемещаемая адресация, что делает возможным разрешение символьских ссылок на адресуемые расположения в других файлах. Переменная окружения **LD_RUN_PATH** действует аналогично опции **-R** командной строки утилиты **ld**.

LIBRARY_PATH

Значение этой переменной окружения может содержать список одного или более имен каталогов. В этих каталогах программа-компоновщик ищет компоновочные сценарии и библиотеки, указываемые опцией командной строки **-l** ("эль").

Опция командной строки **-L** имеет более высокий приоритет, указываемые ею каталоги будут просматриваться в первую очередь.

OBJC_INCLUDE_PATH

Применяется при компиляции программ на языке *Objective-C*. Она действует аналогично опции командной строки **-isystem**. Эта переменная окружения содержит список каталогов для поиска включаемых по директиве **#include** заголовочных файлов. При применении в командной строке опции **-isystem** указанные опцией каталоги будут просматриваться в первую очередь.

Также см. **CPATH**, **CPLUS_INCLUDE_PATH** и **C_INCLUDE_PATH**.

SUNPRO_OUTPUT

При установке значения этой переменной окружения именем файла препроцессор записывает в указанный файл правила зависимостей компоновки. При этом не включаются зависимости, связанные с системными заголовочными файлами.

Если переменная окружения содержит только имя файла, то оно используется как имя выходного файла зависимостей, а для названия набора правил используется имя исходного файла. Если переменная **SUNPRO_OUTPUT** содержит два имени, то второе из них применяется как имя цели, для которой создается отдельное правило.

Использование этой переменной окружения дает такой же результат, как и сочетание опций командной строки **-MM**, **-MF** и **-MT**. Также см. **DEPENDENCIES_OUTPUT**.

TMPDIR

Содержит полный путь расположения каталога, используемого компилятором для хранения временных рабочих файлов,— тех файлов, что обычно удаляются по окончании компиляции. В качестве примера такого файла можно назвать используемый на входе компилятора выходной файл препроцессора.

Приложение В



Перекрестная совместимость опций компилятора GCC

Это приложение содержит сводку перекрестной совместимости опций командной строки компилятора GCC. Опции представлены в алфавитном порядке и сгруппированы по категориям и ключевым словам. Ключевые слова выделены из названий опций, а категории групп определяются общим назначением набора представленных в них опций. Некоторые категории совпадают с названием языка программирования. Они содержат опции, в основном имеющие отношение к компиляции программ на этом языке. Другие категории могут относиться к внутренним действиям компилятора, его процессам, настройкам и т.п. Например, если вам нужно узнать, какие опции имеют отношение к препроцессору, то смотрите категорию `preprocessor`. Названия категорий представлены в английском варианте, русские пояснения мы приводим в скобках.

Совместимость опций

Ada (язык программирования *Ada*)

`-gnat`, `-I`, `--include-directory`, `--no-standard-includes`, `-nostdinc`

alias (использование синонимов)

`-fargument-alias`, `-fargument-noalias`, `-fargument-noalias-global`,
`-fstrict-aliasing`

align (выравнивание)

`-falign-functions`, `-falign-jumps`, `-falign-labels`, `-falign-loops`,
`-Wcast-align`

argument (аргументы функций)

-fargument-alias, **-fargument-noalias**, **-fargument-noalias-global**,
-fugly-args, **-Wformat-extra-args**

asm (ассемблер)

--assemble, **-fasm**, **-fdata-sections**, **-ffunction-sections**,
-finhibit-size-directive, **--for-assembler**, **-fverbose-asm**, **-Wa**

assert (формализованные ответы, препроцессор)

-A, **-A-**, **-assert**

atexit (порядок запуска глобальных деструкторов)

-fuse-cxa-atexit

bitfields (битовые поля)

-fsigned-bitfields, **-funsigned-bitfields**

boehm (способ разметки битовых полей в СВОПЕ)

-fuse-boehm-gc

bounds (проверка границ массивов)

-fbounds-check, **-ffortran-bounds-check**

C (язык программирования C)

--ansi, **-ansi**, **-aux-info**, **-c**, **-C**, **-fallow-single-precision**,
-fasm, **-fbuiltin**, **-fcommon**, **-fcond-mismatch**,
-fdollars-in-identifiers, **-fdump-translation-unit**,
-ffreestanding, **-fhosted**, **-finline**, **-fshort-wchar**,
-fsigned-bitfields, **-fsigned-char**, **-funsigned-bitfields**,
-funsigned-char, **-fwritable-strings**, **-pedantic**,
-pedantic-errors, **-std**, **-traditional-cpp**, **-Waggregate-return**,
-Wbad-function-cast, **-Wcast-align**, **-Wcast-qual**, **-Wchar-subscripts**,
-Wcomment, **-Wconversion**, **-Wdeprecated-declarations**,
-Werror-implicit-function-declaration, **-Wformat**,
-Wformat-extra-args, **-Wformat-nonliteral**, **-Wformat-security**,
-Wformat-y2k, **-Wimplicit**, **-Wimplicit-function-declaration**,
-Wimplicit-int, **-Wimport**, **-Winline**, **-Wlarger-than-size**, **-Wlong-long**,
-Wmain, **-Wmissing-braces**, **-Wmissing-declarations**,
-Wmissing-format-attribute, **-Wmissing-noreturn**,
-Wmissing-prototypes, **-Wmultichar**, **-Wnested-externs**, **-Wpacked**,
-Wpadded, **-Wparentheses**, **-Wpointer-arith**, **-Wredundant-decls**,
-Wreturn-type, **-Wsequence-points**, **-Wshadow**, **-Wsign-compare**,
-Wstrict-prototypes, **-Wswitch**, **-Wsystem-headers**, **-Wtraditional**,
-Wtrigraphs, **-Wundef**, **-Wuninitialized**, **-Wwrite-strings**

C++ (язык программирования C++)

--ansi, **-ansi**, **-faccess-control**, **-falt-external-templates**,
-fasm, **-fcheck-new**, **-fconserve-space**, **-fconst-strings**,
-fdefault-inline, **-fdollars-in-identifiers**, **-fdump-class-hierarchy**,
-fdump-translation-unit, **-fdump-tree-switch**, **-felide-constructors**,
-fenforce-eh-specs, **-fexternal-templates**, **-ffor-scope**,

```

-fgnu-keywords, -fimplement-inlines, -fimplicit-inline-templates,
-fimplicit-templates, -finline, -fmemoize-lookups, -fms-extensions,
-fnonansi-builtins, -foperator-names, -foptional-diags,
-fpermissive, -frepo, -frtti, -fshort-wchar, -fstats,
-ftemplate-depth-number, -fuse-cxa-atexit, -fvtable-gc, -fweak,
-fwritable-strings, -nostdinc++, -pedantic, -pedantic-errors,
-Waggregate-return, -Wcast-align, -Wcast-qual, -Wchar-subscripts,
-Wcomment, -Wconversion, -Wctor-dtor-privacy, -Wdeprecated,
-Wdeprecated-declarations, -Weffc++, -Wextern-inline, -Wformat,
-Wformat-extra-args, -Wformat-nonliteral, -Wformat-security,
-Wformat-y2k, -Wimport, -Winline, -Wlarger-than-size, -Wlong-long,
-Wmain, -Wmissing-braces, -Wmissing-format-attribute,
-Wmissing-noreturn, -Wmultichar, -Wnon-template-friend,
-Wnon-virtual-dtor, -Wold-style-cast, -Woverloaded-virtual,
-Wpacked, -Wpadded, -Wparentheses, -Wpmf-conversions,
-Wpointer-arith, -Wredundant-decls, -Wreorder, -Wreturn-type,
-Wshadow, -Wsign-compare, -Wsign-promo, -Wswitch, -Wsynth,
-Wsystem-headers, -Wundef, -Wuninitialized, -Wwrite-strings

call (вызовы функций)
-fcall-saved-register, -fcall-used-register, -fcaller-saves,
-fnon-call-exceptions, -foptimize-sibling-calls

case (регистр буквенных знаков)
-fcase-initcap, -fcase-lower, -fcase-preserve,
-fcase-strict-lower, -fcase-strict-upper, -fcase-upper,
-fignore-case, -fintrin-case-spec, -fmatch-case-spec,
-fsource-case-spec, -fsymbol-case-spec

cast (приведение типов)
-Wbad-function-cast, -Wcast-align, -Wcast-qual,
-Wold-style-cast

char (тип char)
-fsigned-char, -funsigned-char, -Wchar-subscripts

check (проверки)
-fbounds-check, -fcheck-new, -fcheck-references,
-fdelete-null-pointer-checks, -fforce-classes-archive-check,
-ffortran-bounds-check, -fruntime-checking, -fstack-check,
-fstore-check

Chill (язык программирования Chill)
-fchill-grant-only, -fgrant-only, -fignore-case,
-flocal-loop-counter, -fold-string, -fruntime-checking, -I,
--include-directory, -lang-chill

class (классы)
--bootclasspath, -fconstant-string-class, -fdump-class-hierarchy,
-fforce-classes-archive-check,

```

```
-foptimize-static-class-initialization, -foutput-class-dir, --main,
--output-class-directory

comment (комментарии)
-C, -Wcomment

compile (компиляция)
-c, --compile, -E, -fassume-compiled, -fcommon, -fcompile-resource,
-fgnu-linker, -fgrant-only, -fhosted, -fmerge-all-constants,
-fmerge-constants, -fsyntax, -ftest-coverage, -ftime-report, --help,
-pass-exit-codes, -pipe, --preprocess, -Q, -s, -S, -save-temps,
-syntax-only, -time, -v

complex (арифметика комплексных чисел)
-femulate-complex, -fugly-complex

constant (константы)
-fconst-strings, -fconstant-string-class, -fkeep-static-consts,
-fmerge-all-constants, -fmerge-constants,
-fsingle-precision-constant

conversion (преобразование типов, неявное приведение)
-Wconversion, -Wpmf-conversions

cse (оптимизация Common Code Elimination)
-fcse-follow-jumps, -fcse-skip-blocks, -ffunction-cse,
-frerun-cse-after-loop

debug (отладка)
-a, -d, --debug, -fdump-class-hierarchy, -fdump-translation-unit,
-fdump-tree-switch, -fdump-unnumbered, -finstrument-functions,
-fmem-report, -foptional-diags, -fpermissive, -fsilent, -fstats,
-ftemplate-depth-number, -ftrapping-math, -ftrapv, -fverbose-asn,
-g, -gcoff, -gdwarf, -gdwarf-2, -ggdb, -gstabs, -gvms, -gxcoff,
-H, -v

declaration (объявления)
-gen-decls, -Wdeprecated-declarations,
-Werror-implicit-function-declaration,
-Wimplicit-function-declaration, -Wmissing-declarations,
-Wredundant-decls

dependencies (зависимости)
--dependencies, -M, -MD, -MF, -MG, -MM, -MMD, -MP, -MQ, -MT,
-pass-exit-codes, --print-missing-file-dependencies,
--user-dependencies, -Wout-of-date, --write-dependencies,
--write-user-dependencies

deprecated (прекращение поддержки)
-Wdeprecated, -Wdeprecated-declarations
```

directory (каталоги расположения)
`-B, --bootclasspath directory, -foutput-class-dir, -I, -I-,
 -idirafter, -include, --include-barrier, --include-directory,
 --include-directory-after, --include-prefix, --include-with-prefix,
 --include-with-prefix-after, -iprefix, -isystem, -iwithprefix,
 -iwithprefixbefore, -L, --library-directory,
 -output-class-directory, --prefix, -print-multi-directory,
 -print-prog-name, -print-search-dirs`

dollar (символ "\$")
`-fdollar-ok, -fdollars-in-identifiers`

dump (отладочный дамп)
`-d, --dump, -dumpbase, -dumpmachine, -dumpspecs, -dumpversion,
 -fdump-class-hierarchy, -fdump-translation-unit,
 -fdump-tree-switch, -fdump-unnumbered`

error (сообщения об ошибках)
`-fmessage-length, -pedantic-errors, -Werror,
 -Werror-implicit-function-declaration`

exception (исключения)
`-fasynchronous-unwind-tables, -fcheck-new, -fenforce-eh-specs,
 -fexceptions, -fnon-call-exceptions, -fnonansi-builtins`

extern (использование атрибута external)
`-falt-external-templates, -fexternal-templates, -Wextern-inline,
 -Wnested-externs`

file (файлы)
`-aux-info, -B, --bootclasspath, -include, --include-barrier,
 --include-directory-after, --include-prefix, --include-with-prefix,
 --include-with-prefix-after, --language, -llibrary, -MF, --output,
 -print-file-name, -print-libgcc-file-name,
 --print-missing-file-dependencies, -print-prog-name, -remap,
 -save-temps, -x`

float (числа с плавающей точкой)
`-ffloat-store, -fpretend-float, -Wfloat-equal`

form (форматы исходного кода)
`-ffixed-form, -ffree-form`

format (форматирование)
`-Wformat, -Wformat-extra-args, -Wformat-nonliteral,
 -Wformat-security, -Wformat-y2k, -Wmissing-format-attribute`

Fortran (язык программирования *Fortran*)
`-fautomatic, -fbackslash, -fbadu77-intrinsics-spec, -fbounds-check,
 -fcase-initcap, -fcase-lower, -fcase-preserve, -fcase-strict-lower,
 -fcase-strict-upper, -fcase-upper, -fdollar-ok, -femulate-complex,
 -ff2c, -ff2c-intrinsics-spec, -ff66, -ff77, -ff90,`

-ff90-intrinsics-spec, -ffixed-form, -ffixed-line-length-len,
-ffortran-bounds-check, -ffree-form, -fglobals,
-fgnu-intrinsics-spec, -finit-local-zero, -finline,
-fintrin-case-spec, -fmatch-case-spec, -fmil-intrinsics-spec,
-fonetrip, -fpedantic, -fsecond-underscore, -fsilent,
-fsource-case-spec, -fsymbol-case-spec, -fsyntax, -ftypeless-boz,
-fugly-args, -fugly-assign, -fugly-assumed, -fugly-comma,
-fugly-complex, -fugly-init, -fugly-logint, -funderscoring,
-funix-intrinsics, -fversion, -fvxt, -fvxt-intrinsics, -fzeros,
-maligned-data, -pedantic, -pedantic-errors, -Wglobals, -Wimplicit,
-Wsurprising, -Wunused-initialization

function (функции)

-falign-functions, -ffunction-cse, -ffunction-sections,
-finline-functions, -finstrument-functions, -fkeep-inline-functions,
-Wbad-function-cast, -Werror-implicit-function-declaration,
-Wimplicit-function-declaration, -Wunused-function

garbage (динамическая загрузка и выгрузка кода во время выполнения)

-fuse-boehm-gc, -fvtable-gc

gcse (оптимизация CSE исключения общих глобальных подвыражений)

-fgcse, -fgcse-lm, -fgcse-sm

global (глобальные переменные, функции и т.п.)

-fargument-noalias-global, -fglobals, -fvolatile-global, -Wglobals

gnu (GNU)

-fgnu-intrinsics-spec, -fgnu-keywords, -fgnu-linker, -fgnu-runtime

implicit (явные объявления)

-fimplicit-inline-templates, -fimplicit-templates,
-Werror-implicit-function-declaration, -Wimplicit,
-Wimplicit-function-declaration, -Wimplicit-int

include (включение в код программы заголовочных файлов)

-include, --include-barrier, --include-directory,
--include-directory-after, --include-prefix, --include-with-prefix,
--include-with-prefix-after, --include-with-prefix-before,
--no-standard-includes, --trace-includes

inline (подстановка кода)

-fasm, -fdefault-inline, -fimplement-inlines,
-fimplicit-inline-templates, -finline, -finline-functions,
-finline-limit, -fkeep-inline-functions, -Wwestern-inline, -Winline

intrinsics (встроенные функции)

-fbadu77-intrinsics-spec, -ff2c-intrinsics-spec,
-ff90-intrinsics-spec, -fgnu-intrinsics-spec, -fmil-intrinsics-spec,
-funix-intrinsics, -fvxt-intrinsics iprefix

Java (язык программирования Java)

```
--bootclasspath, -C, -D, --define-macro, --encoding,
-fassume-compiled, -fbebounds-check, -fcheck-references,
-fcompile-resource, -fencoding, -fforce-classes-archive-check,
-fhash-synchronization, -fjni,
-foptimize-static-class-initialization, -foutput-class-dir,
-fstore-check, -fuse-boehm-gc, -fuse-divide-subroutine, -I,
--include-directory, --main, --output-class-directory,
-Wextra-neous-seicolon, -Wlarger-than-size, -Wout-of-date,
-Wredundant-modifiers, -Wshadow
```

label (метки)

```
-falign-labels, -Wunused-label
```

length (длина текстовых полей)

```
-ffixed-line-length-len, -fmessage-length
```

lib (библиотеки)

```
-B, -L, -l, --library-directory, --no-standard-libraries,
-print-libgcc-file-name, -print-multi-lib, -shared, -shared-libgcc,
-static, -static-libgcc, -symbolic
```

link (компоновка)

```
-c, --compile, -fcommon, -fgnu-linker, -fhosted,
-fmerge-all-constants, -fmerge-constants, --for-linker,
--force-link, -fvtable-gc, -L, --library-directory, -llibrary,
--no-standard-libraries, -nodefaultlibs, -nostartfiles, -nostdlib,
-s, -shared, -shared-libgcc, -static, -static-libgcc, -symbolic,
-u, -Wl, -Xlinker
```

machine (аппаратные платформы)

```
-b, --target, --target-help
```

macro (макросы)

```
--ansi, -D, --define-macro, -ffast-math, -ffixed-register, -imacros,
-U, -undef, --undefined-macro
```

math (математика)

```
-fallow-single-precision, -femulate-complex, -ffast-math,
-ffloat-store, -fmath-errno, -fpretend-float, -fschedule-insns,
-fschedule-insns2, -fshort-double, -fsingle-precision-constant,
-ftrapping-math, -ftrapv, -ftypeless-boz, -fugly-complex,
-funsafe-math-optimizations, -funsigned-bitfields, -funsigned-char,
-fuse-divide-subroutine, -Wdiv-by-zero, -Wfloat-equal,
-Wsign-compare, -Wsign-promo, -Wsurprising
```

missing (пропущенные элементы)

```
--print-missing-file-dependencies, -Wmissing-braces,
-Wmissing-declarations, -Wmissing-format-attribute,
-Wmissing-noreturn, -Wmissing-prototypes
```

Objective-C (язык программирования Objective-C)

```
--ansi, -ansi, -fasm, -fbuiltin, -fconstant-string-class,
-fgnu-runtime, -finline, -gen-decls, -Waggregate-return,
-Wcast-align, -Wcast-qual, -Wchar-subscripts, -Wcomment,
-Wconversion, -Wdeprecated-declarations, -Wformat,
-Wformat-extra-args, -Wformat-nonliteral, -Wformat-security,
-Wformat-y2k, -Wimport, -Winline, -Wlarger-than-size, -Wlong-long,
-Wmissing-braces, -Wmissing-format-attribute, -Wmissing-noreturn,
-Wmultichar, -Wpacked, -Wpadded, -Wparentheses, -Wpointer-arith,
-Wprotocol, -Wredundant-decls, -Wselector, -Wshadow, -Wsign-compare,
-Wswitch, -Wsystem-headers, -Wundef, -Wuninitialized
```

optimization (оптимизация)

```
-fasynchronous-unwind-tables, -fbranch-probabilities,
-fcall-saved-register, -fcall-used-register, -fcaller-saves,
-fcommon, -fconserve-space, -fcprop-registers, -fcse-follow-jumps,
-fcse-skip-blocks, -fdata-sections, -fdefer-pop, -fdelayed-branch,
-fdelete-null-pointer-checks, -fdiagnostics-show-location,
-felide-constructors, -fexpensive-optimizations, -ffloat-store,
-ffunction-cse, -ffunction-sections, -fgcse, -fgcse-lm, -fglobals,
-fguess-branch-probability, -finit-local-zero, -fkeep-static-consts,
-fmemoize-lookups, -fmove-all-movables, -fomit-frame-pointer,
-foptimize-register-move, -foptimize-sibling-calls,
-foptimize-static-class-initialization, -fpack-struct, -fpeephole,
-fpeephole2, -fppc-struct-return, -fprefetch-loop-arrays,
-freduce-all-givs, -freg-struct-return, -fregmove,
-frename-registers, -frerun-cse-after-loop, -frerun-loop-opt,
-fruntime-checking, -fschedule-insns, -fschedule-insns2,
-fshort-double, -fshort-enums, -fssa, -fssa-cpp, -fssa-dce,
-fstack-check, -fstore-check, -fstrength-reduce, -fstrict-aliasing,
-fthread-jumps, -funroll-all-loops, -funroll-loops, -funwind-tables,
-fvtable-gc, -fzeros, -O, --optimize optimize, --optimize, --param,
-Wdisabled-optimization
```

preprocessor (препроцессор)

```
-A, -A-, --assert, -C, -D, --define-macro, --dependencies
directory, -E, -fident, -fpreprocessed, -H, -I, -I-, -idirafter,
-imacros, -include, --include-barrier, --include-directory,
--include-directory-after, --include-prefix, --include-with-prefix,
--include-with-prefix-after, --include-with-prefix-before, -iprefix,
-isystem, -iwithprefix, -iwithprefixbefore, -M, -MD, -MF, -MG, -MM,
-MMD, -MQ, -MT, --no-line-commands, --no-standard-includes,
-nostdinc, -nostdinc++, -P, --preprocess,
--print-missing-file-dependencies, -remap, --trace-includes,
-trigraphs, -U, -undef, --undefined-macro, --user-dependencies, -Wp,
--write-dependencies, --write-user-dependencies, -Wsystem-headers,
-Wundef, -Wunknown-pragmas
```

profile (профилирующий код)

-a, -fdata-sections, -fprofile-arcs, -ftest-coverage, -p, -pg,
--profile, --profile-blocks

prototypes (прототипы)

-Wmissing-prototypes, -Wstrict-prototypes

register (регистры)

-fcall-saved-register, -fcall-used-register, -fcprop-registers,
-ffixed-register, -fforce-addr, -fforce-mem,
-foptimize-register-move, -freg-struct-return, -fregmove,
-frename-registers, -fstack-limit-register, -remap

return (тип возвращаемого значения)

-fppc-struct-return, -freg-struct-return, -Waggregate-return,
-Wreturn-type

sign (знак числа)

-fsigned-bitfields, -fsigned-char, -Wsign-compare, -Wsign-promo

ssa (использование графа Static Single Assignment)

-fssa, -fssa-cpp, -fssa-dce

stack (использование стека)

-fstack-check, -fstack-limit-register, -fstack-limit-symbol

standard (стандарты)

--ansi, -ansi, -ff2c, -ff2c-intrinsics, -ff66, -ff77, -ff90,
-ff90-intrinsics, -ffixed-form, -ffixed-line-length-len,
-ffor-scope, -ffree-form, -fgnu-keywords, -fmil-intrinsics,
-fms-extensions, -fnext-runtime, -fnonansi-builtins,
-foperator-names, -fpedantic, -fpermissive, -fsigned-bitfields,
-fsigned-char, -ftrapping-math, -ftrapv, -fugly-args,
-fugly-assign, -fugly-assumed, -fugly-comma, -fugly-complex,
-fugly-init, -fugly-logint, -fvxt, -fvxt-intrinsics,
-fwriteable-strings, --no-standard-includes, --no-standard-libraries,
-pedantic, -std, -traditional

static (статические данные)

-fkeep-static-consts, -foptimize-static-class-initialization,
-fvolatile-static, -static, -static-libgcc

strings (строки)

-fconst-strings, -fconstant-string-class, -fold-string,
-fwriteable-strings, -Wwrite-strings

syntax (синтаксис)

-fsyntax, -syntax-only

template (шаблоны)
-falt-external-templates, -fexternal-templates,
-fimplicit-inline-templates, -fimplicit-templates,
-ftemplate-depth-number, -Wnon-template-friend

underscore (знаки подчеркивания)
-fleading-underscore, -fsecond-underscore, -funderscoring

version (версии)
-dumpversion, -fversion, --use-version, -v, -V

warn (предупредительные сообщения)
--all-warnings, --extra-warnings, -fmessage-length, --no-warnings,
-w, -W, -Waggregate-return, -Wall, --warn-, -Wbad-function-cast,
-Wcast-align, -Wcast-qual, -Wchar-subscripts, -Wcomment,
-Wconversion, -Wctor-dtor-privacy, -Wdeprecated,
-Wdeprecated-declarations, -Wdisabled-optimization, -Wdiv-by-zero,
-Weffc++, -Werror, -Werror-implicit-function-declaration,
-Wextern-inline, -Wextraaneous-seicolon, -Wfloat-equal, -Wformat,
-Wformat-extra-args, -Wformat-nonliteral, -Wformat-security,
-Wformat-y2k, -Wglobals, -Wimplicit,
-Wimplicit-function-declaration, -Wimplicit-int, -Wimport,
-Winline, -Wlarger-than-size, -Wlong-long, -Wmain,
-Wmissing-braces, -Wmissing-declarations,
-Wmissing-format-attribute, -Wmissing-noreturn,
-Wmissing-prototypes, -Wmultichar, -Wnested-externs,
-Wnon-template-friend, -Wnon-virtual-dtor, -Wold-style-cast,
-Wout-of-date, -Woverloaded-virtual, -Wpacked, -Wpadded,
-Wparentheses, -Wpmf-conversions, -Wpointer-arith, -Wprotocol,
-Wredundant-decls, -Wredundant-modifiers, -Wreorder,
-Wreturn-type, -Wselector, -Wsequence-points, -Wshadow,
-Wsign-compare, -Wsign-promo, -Wstrict-prototypes, -Wsurprising,
-Wswitch, -Wsynth, -Wsystem-headers, -Wtraditional, -Wtrigraphs,
-Wundef, -Wuninitialized, -Wunknown-pragmas, -Wunreachable-code,
-Wunused, -Wunused-function, -Wunused-label, -Wunused-parameter,
-Wunused-value, -Wunused-variable, -Wwrite-strings

Приложение Г



Опции командной строки компилятора GCC

Это приложение содержит алфавитный список опций командной строки компилятора GCC. Некоторые из опций применяются ко всем языкам программирования, в то время как другие — к одному или нескольким языкам. Также есть опции, избирательно действующие только на препроцессор, ассемблер или компоновщик. Каждая опция в списке этого Приложения имеет отметку, которая обозначает область ее применения. Опции, применимые ко всем языкам, не имеют такой отметки. Вот список всех отметок, которые применяются в алфавитном списке опций, и описание соответствующих им областей действия:

- **Ada** — Компиляция программ на языке *Ada*.
- **Asm** — Ассемблер.
- **Java** — Компиляция программ на языке *Java*.
- **Linker** — Компоновщик.
- **Pre** — Препроцессор.
- **ObjC** — Язык программирования *Objective-C*.
- **C** — Программы на языке *C*.
- **C++** — Программы на языке *C++*.
- **Fortran** — Программы на языке *Fortran*.

Командой `gcc` можно компилировать код на любом из поддерживаемых языков программирования. Но, вполне возможно, что при этом будут доступны не все специфические опции отдельных языков. Каждый язык имеет свой драйвер верхнего уровня компилятора. Для распознавания и использования отдельных специфических к языку опций может потребоваться запуск оболочки соответствующего языка.

Префиксы опций

Все опции командной строки начинаются с символа дефиса “-” (hyphen). Некоторые из опций в префиксах имеют два дефиса “--”. Есть также опции, которые имеют специальное значение, они начинаются с “-f” или “-W”.

Префикс “--”

Традиционный способ обозначения опции в командной строке состоит в том, что в начале опции стоит один дефис “-” и английская буква сразу после него. Более новая форма представления предполагает использование пары дефисов “--”. Многие опции компилятора в списке поддерживают как старый (с одним дефисом), так и новый (с двумя дефисами) форматы представления, имеющие одинаковое значение. Например, традиционная форма опции для включения в генерируемый код отладочной информации имеет такой вид:

```
-g
```

То же самое действие имеет и следующая, более длинная опция:

```
--debug
```

Префикс “-f”

Буква ‘f’ в начале опции определяет значение флага компиляции. Большая часть таких опций имеет два положения: установленное и отключенное. Например, следующая опция устанавливает флаг, применяющий локальную оптимизацию “peephole optimization”, связанную с заменой инструкций:

```
-fpeephole
```

Так как флаг может иметь включенное и выключенное положение, каждая устанавливающая флаг опция может быть обращена. При этом используется то же имя опции, но с дополнительным префиксом “no-”. В алфавитном списке такая форма опции, имеющая противоположное значение, будет называться *обратной*. Например:

```
-fno-peephole
```

Почти все такие опции устанавливают состояние соответствующих флагов, присваивая им значение “истина” или “ложь”. Причем многие флаги имеют значение по умолчанию. Но имеются и некоторые исключения. Например, любая из следующих опций может быть применена для указания области видимости переменных, объявляемых в разделе инициализации цикла **for**:

```
-ffor-scope  
-fno-for-scope
```

Однако по умолчанию область видимости переменных не задается. В общем случае она зависит от стандарта, но оба варианта установки области видимости имеют разное значение для различных стандартов.

Каждая опция с префиксом "**-f**" может указываться в форме представления с двумя дефисами. Например, две следующие опции имеют одинаковое значение:

```
-frtti  
--rtti
```

Префикс "**-W**"

Опции с префиксом "**-W**" используются для указаний компилятору, связанных с генерированием тех или иных предупредительных сообщений. Подобно флаговым опциям с префиксом "**-f**", эти опции могут включать или отключать вывод определенных предупреждений в зависимости от использования в начале опции дополнительного префикса "**по-**". Например, установка следующей опции (с префиксом "**-W**") определяет вывод предупреждений в случае превышения допустимого количества аргументов при вызове функций:

```
-Wformat-extra-args
```

Для подавления вывода таких сообщений следует применить обратную форму этой опции с префиксом "**-Wno-**":

```
-Wno-format-extra-args
```

Порядок следования опций

Порядок следования опций может иметь важное значение. Если в командной строке стоят две конфликтующие опции, то обычно действует вторая опция, заменившая установки, сделанные первой опцией. Параметры командной строкичитываются программой слева направо, и каждая опция последовательно устанавливает соответствующее значение или флаг (или набор значений и флагов). Поэтому любая установка, сделанная какой-либо опцией, может быть изменена последующими опциями, стоящими в командной строке после нее.

Этот порядок применения опций дает немало удобств. Например, известно, что флаг оптимизации **-O3** устанавливает опцию **-finline-functions**. Если вы хотите применить оптимизацию **-O3** и при этом сохранить отключенным расширение вызовов функций подстановкой кода, то для этого вы можете поставить опции в таком порядке:

```
-O3 -fno-inline-functions
```

Типы файлов

Компилятор определяет содержание файла по суффиксу его имени, в соответствии со списком, представленным в таблице Г.1. Любой файл с неизвестным суффиксом воспринимается как входной файл компоновщика предназначаемой целевой машины и передается ему на этапе компоновки программы. Для отмены действия предустановленного значения суффикса и указания типа отдельного входного файла используется опция **-x**.

Таблица Г.1. Распознаваемые GCC суффиксы имен файлов

Суффикс	Содержимое файла
.a	Статическая библиотека, которая содержит один или более используемых при компоновке программы объектных .o файлов. Она также имеет название архив.
.c	Исходный код на языке C, который подлежит предобработке.
.adb	Исходный <i>body</i> файл на языке Ada. Содержит "тело" (т.е., код реализации) библиотечного модуля Ada.
.adw	Файл спецификации (spec-файл) на языке Ada. Содержит объявления или переназначения объявлений модулей, как отдельных, так и в составе библиотек Ada.
.c .c++ .cc .cp .cpp .cxx	Исходный код на языке C++, подлежащий предобработке.
.class	Вырабатываемый при компиляции программ на языке Java файл класса, содержит выполнимый байтовый код (байт-код) Виртуальной Машины Java.
.f .for .FOR	Исходный код на языке Fortran, не требующий предобработки.
.F .fpp .FPP	Исходный код на языке Fortran, который подлежит предобработке.
.h	Заголовочный файл (header file) на языке C, C++ или Objective-C.
.i	Исходный код на языке C, не требующий предобработки.
.ii	Исходный код на языке C++, не требующий предобработки.
.java	Исходный файл на языке Java.
.m	Исходный код на языке Objective-C, который подлежит предобработке.
.mi	Исходный код на языке Objective-C, не требующий предобработки.
.mo	Двоичный файл с переводами строк для интернационализации программ.
.o	Объектный код соответствующего формата, поддерживаемого компоновщиком.
.po	Текстовый файл с переводами строк, которые применяются для интернационализации программ.
.r	Исходный код на языке Fortran, который подлежит предобработке препроцессором RATFOR.
.s	Предназначаемый для предобработки код на языке ассемблера.
.v	Код на ассемблере, не требующий предобработки.
.vo	Динамическая библиотека (также называемая разделяемой). Содержит один или более объектных .o файлов с перемещаемой внутренней адресацией, которые компонуются к программе во время ее загрузки и выполнения.
<другие суффиксы>	Файл с неизвестным суффиксом считается входным файлом компоновщика предназначаемой машины, он передается этому компоновщику на этапе компоновки объектного кода программы.

Алфавитный список опций

-####

Показывает номер версии компилятора и, затем, все поддерживаемые команды, которые могут запускаться на различных этапах компиляции и компоновки. При

в этом никакие команды не выполняются. Если используется только одна эта опция, то выводится номер версии компилятора. В сочетании с опцией `--help` выдает полный список поддерживаемых опций командной строки.

-a

Генерирует дополнительный профилирующий код в начале каждого основного блока (basic block) выполнимого кода. Профилирующая информация записывается каждый раз при запуске основного блока. Записываемая информация включает в себя начальный адрес и имя функции, содержащей основной блок. Если также применяется опция `-g`, то выводимая для каждого блока информация будет также включать имя исходного файла и номер начальной строки исходного кода каждого блока.

Если в описании предназначаемой машины не указано другое имя файла, то эта информация записывается в файл с именем `bb.out`.

См. также опции `-fprofile-arcs` и `-ftest-coverage`. Эта опция может применяться в другой форме: `--profile-blocks`.

-A *question(answer)*

Pre

Назначает ответ (утверждение, "assertion") на указанный вопрос. Действует аналогично следующей директиве:

```
#if #question(answer)
```

Опция может быть записана в форме `--assert`. См. также `-A-`.

-A-

Pre

Отключает стандартные ответы (утверждения, assertions) на вопросы, которые обычно применяются для описания предназначаемой (целевой) машины (target). См. также `-A`.

--all-warnings

То же, что и опция `-Wall`.

--ansi

C, C++, ObjC

То же, что и `-ansi`.

-ansi

C, C++, ObjC

Эта опция сообщает компилятору о соответствии исходного кода стандарту ANSI. При этом не применяется никаких ограничений по отношению к коду, не конфликтующему прямым образом со стандартом. Это значит, что допускается применение расширений GNU. При компиляции программ на языке C эта опция допускает применение Стандарта "ISO C89". При компиляции программ на языке C++ при этой опции отключается поддержка всех расширений GNU, конфликтующих со Стандартом "ISO C++".

Эта опция также устанавливает опции `-fno-asm`, `-fno-nonansi-builtins`, `-trigraphs` и `-fno-dollars-in-identifiers`. Для языка C++ также дополнительно устанавливаются опции `-fno-gnu-keywords` и `-fno-nonansi-builtins`.

Эта опция определяет макрос `__STRICT_ANSI__`, который предотвращает объявление в некоторых заголовочных файлах (header files) функций или макросов, имеющих конфликтовать с именами компилируемой программы.

Эта опция отключает ключевые слова расширения GNU языка C: `asm`, `typeof` и `inline`. При этом остаются доступными альтернативные формы этих ключевых слов: `__asm__`, `__typeof__` и `__inline__`.

Если требуется строгое ограничение на соответствие кода стандарту, то дополнительно к опции `-ansi` следует применять опцию `-pedantic`. См. также опцию `-std`.

Опция `-ansi` может быть записана в форме `--ansi`.

--assemble

То же, что `-S`.

-assert *question(answer)*

Pre

То же, что опция `-A`.

-aux-info *filename*

C

Выводит объявления прототипов для всех функций, объявленных в отдельном модуле компиляции (имеется в виду отдельный файл с исходным кодом на языке C и все заголовочные файлы, которые он подключает). Информация выводится в указанный файл *filename*.

-b *machine*

Эта опция указывает тип предназначаемой (или, иначе говоря, целевой) машины (target), для выполнения на которой компилируется программа. Если данная опция не применяется, то по умолчанию программа компилируется для текущей машины, т.е. для той, на которой запускается компилятор. Тип машины определяется указанием имени каталога, содержащего необходимую конфигурацию компилятора. Обычно имена таких каталогов соответствуют шаблону

`/user/local/bin/gcc-lib/machine/version.`

См. также опции `-B` и `-V`. Опция может быть записана в форме `--target`.

-B*prefix*

Имя префиксного каталога *prefix* указывает расположение библиотек, включаемых файлов, выполняемых программ и файлов с данными компилятора. При запуске таких подпрограмм как `cpp`, `as` или `ld` имя префиксного каталога используется для поиска запускаемой программы. В конце имени префиксного каталога можно поставить буквенный символ-разделитель имен каталогов для путей доступа, а можно, по вашему выбору, обойтись и без него.

Во всех случаях при поиске применяется одна и та же стандартная процедура. При этом для обнаружения предмета поиска последовательно предпринимаются попытки найти его в следующих местах:

1. Если опцией `-B` указано имя префиксного каталога, то оно используется для построения полного пути доступа к файлу.

2. Имя пути доступа строится с использованием стандартного префикса `/usr/lib/gcc`.
3. Для построения пути доступа к файлу используется префикс `/user/local/lib/gcc-lib/`.
4. Используются пути расположения, определенные в списке переменной окружения **PATH**. Они берутся последовательно в том же порядке, в котором представлены в значении переменной.

Опция **-B** применяется также и для поиска используемых компоновщиком библиотек. Компилятор транслирует значение указанного в этой опции параметра в опцию **-L**, которая передается компоновщику.

Имя префиксного каталога, указанного опцией **-B**, используется также для поиска включаемых заголовочных файлов (header files). Значение параметра опции **-B** компилятор передает препроцессору опцией **-isystem**. При поиске заголовочных файлов к передаваемому имени префиксного каталога препроцессор добавляет имена стандартных подкаталогов.

Для назначения префиксного каталога может использоваться переменная окружения **GCC_EXEC_PREFIX**. Она будет действовать так же, как и опция **-B**. При одновременном применении опция **-B** имеет приоритет выше, чем переменная окружения **GCC_EXEC_PREFIX**.

Для пускового (bootstrapping) компилятора предусмотрена особая установка. Если значение префиксного каталога имеет формат в диапазоне от `dirpath/stage0` до `dirpath/stage9`, то при поиске заголовочных файлов оно будет заменяться на `dirpath/include`.

Опция может применяться в форме **--prefix**.

-bootclass=pathname

Java

Значение **pathname** указывает расположение стандартных пакетов и классов **Java** (таких классов, как `java.lang.String`). Поле **pathname** может не только указывать каталог расположения, но и включать в себя имена **jar** или **zip** архивов.

См. также опции **--classpath** и **-I**.

-C

Опция указывает не задействовать компоновщик. Она позволяет выполнять компиляцию и ассемблирование исходного кода без запуска компоновщика для сборки готовой к запуску машинной программы. Вырабатываемые при этом объектные модули сохраняются в файлах с суффиксом **.o**, их имена соответствуют файлам. Все файлы с суффиксами, не имеющими соответствий в таблице Г.1 игнорируются при отсутствии опции **-x**, указывающей тип их содержимого.

Опция имеет альтернативную форму **--compile**.

-C

Java

Установка этой опции приводит к генерированию компилятором выхода в формате исполняемого байт-кода Виртуальной Машины Java вместо объектного кода, вырабатываемого по умолчанию.

См. также **-foutput-class-dir**.

-C

Pre

При использовании этой опции в сочетании с опцией **-E** все комментарии удаляются.

Опция может применяться в форме **--comments**.

--classpath=path

Java

Назначает расположение **path** в качестве имени верхнего каталога для поиска файлов-классов *Java*. Возможно указание не только каталога, но также имени **.jar** или **.zip** архива.

См. также опции **-bootclass** и **-I**.

-compile

То же, что опция **-c**.

-dletters

В поле **letters** может стоять одна или набор букв, определяющих содержание выводимого при отладке дампа информации (или нескольких дампов). Эта опция предназначена для отладки компилятора. Она дает возможность получения подробной информации о работе компилятора на различных этапах компиляции программы. Имя каждого выходного файла имеет суффикс, состоящий из номера прохода и некоторой последовательности идентифицирующих букв. Например, компилируемый исходный файл имеет имя **doline**. Тогда файл с дампом 21-го последовательного прохода, который содержит отладочную информацию, связанную с оптимизацией глобального распределения регистров, будет иметь имя **doline.21.greg**.

Также см. опции **-dumpbase**, **-fdump-unnumbered**, **-fdump-translation-unit**, **-fdump-class-hierarchy** и **-fdump-tree-switch**. Опция **-d** имеет альтернативную форму **--dump**.

В таблице Г.2 представлен список доступных для использования с опцией **-d** буквенных кодов. Они могут применяться в любом сочетании и в произвольном порядке. Реализация набора параметров для вывода дампа отладки строго соответствует потребностям отладки самого компилятора. Поэтому ряд буквенных кодов в некоторых выпусках компилятора могут не поддерживаться. Учтите, что коды **D**, **I**, **M** и **N** имеют особые значения, при использовании опции **-E** они применяются только по отношению к препроцессору.

Таблица Г.2. Буквенные коды содержания вывода отладки, применяемые с опцией **-d**

Буква	Вырабатываемый выход дампа отладки
A	Добавляет в выходной ассемблерный код разнообразную отладочную информацию.
a	Устанавливает флаг, в соответствии с которым дамп отладки создается для всех перечисленных в команде файлов за исключением файлов name.pass.vcd , указанных буквой v .

- A |
- Добавляет в выходной ассемблерный код разнообразную отладочную информацию. |

- a |
- Устанавливает флаг, в соответствии с которым дамп отладки создается для всех перечисленных в команде файлов за исключением файлов **name.pass.vcd**, указанных буквой **v**. |

Буква	Вырабатываемый выход дампа отладки
b	Выводит дамп в файл <code>name.14.bp</code> после расчета вероятностей ветвлений (branch probabilities).
B	Выводит дамп в файл <code>name.29.bbrc</code> после оптимизации переупорядочения блоков (block reordering).
c	Выводит дамп в файл <code>name.16.combine</code> после оптимизации объединения инструкций (instruction combining).
C	Выводит дамп в файл <code>name.17.ce</code> после первого преобразования условных переходов (if-conversion).
d	Выводит дамп в файл <code>name.31.dbr</code> после оптимизации планирования отложенного выполнения инструкций ветвления (delayed branch scheduling).
D	При использовании вместе с опцией <code>-E</code> добавляет к обычному выводу препроцессора все макроопределения.
e	Выводит дамп в файлы <code>name.04.eva</code> и <code>name.07.eva</code> после применения оптимизации отдельных статических переназначений (static single assignments).
E	Выводит дамп в файл <code>name.26.ce2</code> после второго преобразования условных переходов (if-conversion).
f	Выводит дамп в файл <code>name.13.cfg</code> после выполнения анализа потока данных (data flow analysis) и в файл <code>name.15.life</code> после выполнения анализа времени жизни данных (life analysis).
F	Выводит отладочный дамп в файл с именем <code>name.09.ARDESSOF</code> после очистки кодов ARDESSOF.
g	Выводит отладочный дамп в файл <code>name.21.greg</code> после глобального распределения регистров.
G	Выводит отладочный дамп в файл с именем <code>name.10.GCSE</code> после применения GCSE.
h	Выводит отладочный дамп в файл с именем <code>name.02.eh</code> после завершения оптимизации обработки исключений.
i	Выводит отладочный дамп в файл с именем <code>name.10.sibling</code> после оптимизации преобразования вложенных вызовов в циклы (sibling call optimisation).
I	Используется вместе с опцией <code>-E</code> . При этом кроме обычного выхода препроцессор выводит все директивы <code>#include</code> .
j	Выводит дамп в файл с именем <code>name.03.jump</code> после первой оптимизации дальних вызовов (jump optimisation).
k	Выводит дамп в файл <code>name.28.stack</code> после преобразования способа передачи параметров вызова, при котором вместо регистров для этого используется стек (register-to-stack conversion). При обратном преобразовании, когда передача аргументов переносится из стека в регистры (stack-to-register conversion), дамп выводится в файл <code>name.32.stack</code> .
l	Выводит дамп в файл <code>name.20.lreg</code> после оптимизации локального распределения регистров.
L	Выводит дамп в файл <code>name.11.loopt</code> оптимизации циклов (loop optimisation).
M	Выводит дамп в файл <code>name.30.mach</code> после прохода машинно-зависимой реорганизации. Вместе с опцией <code>-E</code> определяет в конце всей предобработки дополнительный вывод препроцессором списка всех выполненных макроопределений.
m	В конце компиляции выводит на стандартное устройство вывода сообщений об ошибках информацию об использовании памяти.
n	Выводит дамп в файл <code>name.25.rnreg</code> после изменения нумерации регистров (register renumbering).

Буква	Вырабатываемый выход дампа отладки
N	Выводит дамп в файл <code>name.25.register</code> после прохода оптимизации переноса регистров (register move pass). В сочетании с опцией <code>-E</code> включает в конце предобработки в обычный выход препроцессора список всех макросов в упрощенной форме <code>#define name</code> .
O	Выводит дамп в файл <code>name.22.reload</code> после оптимизации перезагрузок подпрограмм (post-reload optimisation).
P	Добавляет комментарии в выходной ассемблерный код, указывающие длину каждой инструкции и использованные методы оптимизации.
R	Добавляет в выходной ассемблерный код комментарии, представляющие RTL-код, использованный для выработки каждой инструкции ассемблера. См. также буквенный код <code>R</code> в этой таблице.
r	Выводит дамп в файл <code>name.00 rtl</code> после этапа генерирования кода в формате RTL. См. также буквенный код <code>x</code> в этой таблице.
R	Выводит дамп в файл с именем <code>name.27.shed</code> после второго прохода оптимизации планирования инструкций (sheduling).
s	Выводит отладочный дамп в файл <code>name.08.sve</code> после оптимизации исключения глобальных общих подвыражений CSE (Common Subexpression Elimination). Часто сразу после CSE следует оптимизация длинных переходов (jump optimisation), в таком случае дамп в файл <code>name.08.sve</code> записывается после него.
S	Выводит дамп в файл с именем <code>name.19.shed</code> после первого прохода оптимизации планирования инструкций (sheduling).
t	Выводит дамп в файл <code>name.12.sve2</code> после второго прохода CSE (Common Subexpression Elimination) и иногда следующей за ним оптимизации длинных переходов (jump optimisation).
u	Выводит дамп в файл с именем <code>name.06.null</code> после всех оптимизаций SSA (Static Single Assignment).
v	Выводит в файл <code>name.ravv.vcg</code> дамп после представления графа управляющего потока (control flow) для каждого из прочих файлов дампа, кроме <code>name.00.rtl</code> . Эти файлы имеют формат, пригодный для считывания и просмотра с помощью утилиты <code>vcd</code> .
w	Выводит в файл с именем <code>name.23.flow2</code> дамп после второго прохода оптимизации управляющего потока (flow).
W	Выводит дамп в файл с именем <code>name.05.vacccp</code> после прохода оптимизации SSA передачи кода, компилируемого по условию, (conditional code propagation).
X	Выводит дамп в файл с именем <code>name.06.vadce</code> после прохода оптимизации SSA устранения неиспользуемых участков кода (dead code elimination).
x	Вырабатывает RTL-код для функции, но дальше его не компилирует. Этот буквенный код часто используется в сочетании с <code>r</code> .
y	Определяет вывод отладочной информации синтаксическим разделителем (parser) на стандартное устройство вывода.
z	Выводит дамп в файл с именем <code>name.24.peephole2</code> после прохода локальной оптимизации замены инструкций (peephole optimization).

-Dproperty**[=*string*]****Java**

Эта опция может быть использована в командной строке совместно с опцией `--main`. Она определяет свойство с именем `property` и присваивает ему значение `string`. Значение может быть получено в программе с помощью вызова метода

`java.lang.System.getProperty()` с именем свойства в аргументе. Если не указано значение *string*, то значением свойства будет пустая строка.

Другая форма этой опции --**define-macro**.

-D*macro*[=*string*]

Pre

Когда указано значение *string*, то этим значением определяется макрос с указанным в поле *macro* именем. Точно так же, как если бы код программы содержал соответствующую директиву макроопределения. Например, опция **-Dbrunt=logger** генерирует следующее макроопределение:

```
#define brunt logger
```

Если же значение *string* не указано, то макрос определяется строкой "1". Например, по опции **-Dminke** генерируется следующее макроопределение:

```
#define minke 1
```

Все опции **-D** обрабатываются раньше любых опций **-U**. Так же, как и все опции **-U** обрабатываются прежде любых опций **-include** или **-imacros**.

--debug[/*level*]

То же, что опция **-g**.

-define-macro *macro*[=*string*]

Pre, Java

То же, что опция **-D**.

-dependencies

Pre

То же, что опция **-M**.

-dump *letters*

То же, что опция **-d**.

-dumpbase *base*

Поле *base* определяет основное имя файлов для вывода дампов отладки, которые вырабатываются по опции **-d**.

Опция может быть записана в форме **--dumpbase**.

-dumpmachine

Эта опция выводит название типа предназначаемой машины (target) текущей конфигурации компилятора. Больше никаких действий при этом не выполняется.

-dumpspecs

Выводит спецификации, использованные при сборке компилятора. Больше никаких действий при этом не выполняется. Выводится большой листинг, включающий все опции и установки (вместе с действующими по умолчанию), которые использовались при компиляции, ассемблировании и компоновке самого компилятора.

-dumpversion

Выводит номер версии компилятора. Никаких дальнейших действий не предпринимается.

-E**Pre**

Останавливает процесс компиляции после предобработки исходного кода и вывода ее результатов. Если не указана опция **-o**, то вывод направляется на стандартное устройство выхода. В противном случае информация записывается в указанный опцией **-o** файл. Препроцессор пропускает файлы, не требующие предобработки. Такие файлы определяются по суффиксу их имени (см. таблицу Г.1), если не существует переназначение типа файла опцией **-x**.

Эта опция устанавливает переменные окружения **__GNUC__**, **__GNUC_MINOR__** и **__GNUC_PATCHLEVEL__**.

Опции **-dD**, **-dT**, **-dM** и **-dN** при совместном использовании с опцией **-E** приобретают особые значения, которые предусмотрены для такого случая их использования.

--encoding=name**Java**

То же, что опция **-fencoding**.

--extra-warnings

То же, что **-w**.

-faccess-control**C++**

Данная опция действует по умолчанию. При использовании обратной опции **-fno-access-control** компилятор не будет выполнять проверки, связанные с разрешениями доступа. Единственное назначение этого флага состоит в обходе возможных ошибок обработки прав доступа компилятором.

-falign-functions[=number]

Опция применяет выравнивание начальных адресов кода функций по границе выравнивания второго типа (power 2) или по ближайшей границе выравнивания, не превышающей указанное в поле **number** число байт. Но применяется это выравнивание только тогда, когда не возникает необходимости пропускать более **number** байт. Например, **number** имеет значение **20**. Тогда в случае выравнивания к границе 32 байта, код будет выравниваться только при условии, что для этого не придется пропускать более 20-ти байт памяти.

Если значение поля **number** устанавливается равным границе выравнивания второго типа, то выравнивание будет применяться без исключения ко всем функциям. Если значение **number** не указано, то применяется установка по умолчанию, соответствующая типу машины. Для некоторых машин это число округляется до значения выравнивания второго типа (power 2). При этом, конечно, выравнивание будет применяться ко всем функциям. Указание в поле **number** значения **1** эквивалентно действию опции **-fno-align-functions**, при которой выравнивание функций не применяется.

-falign-jumps[=number]

Выравнивает целевые адреса переходов ветвления (branch targets) по границе выравнивания второго типа (power 2) или к ближайшей границе выравнивания, превышающей указанное число *number*, если при этом не возникает необходимости пропускать более *number* байт памяти. Например, если *number* равен 20 и применяется выравнивания к границе 32 байта, то целевой код переходов jump будет выравниваться лишь тогда, когда для этого перед адресуемым кодом не придется пропускать более 20-ти байт памяти. В отличие от сходной по действию опции **-falign-labels** рассматриваемая опция не требует заполнения пропускаемого пространства памяти пустыми операциями.

Если значение *number* не указано, то применяется машинная установка по умолчанию, обычно равная 1. Указание в поле *number* значения 1 эквивалентно действию опции **-fno-align-jumps**, при этом выравнивание ветвей кода не применяется.

-falign-labels[=number]

Выравнивает адрес целевых инструкций всех переходов по границе второго типа (power 2) или к ближайшей границе выравнивания, превышающей указанное число *number*. Это выравнивание применяется только тогда, когда при этом не возникает необходимости пропускать более *number* байт. Например, значение *number* равно 20. Тогда в случае 32-байтного выравнивания, адресуемые переходами ветви кода будут выравниваться к ближайшей границе 32-байтного выравнивания только если для этого придется пропускать не более 20-ти байт. Эта опция может увеличить размер вырабатываемого кода и время компиляции, потому что пропускаемые байты заполняются пустыми операциями. Более простая форма этой опции, не требующая дополнительных расходов на компиляцию, имеет вид **-falign-jumps**.

При одновременном использовании опций **-falign-jumps** и **-falign-labels** с разными значениями поля *number* для обеих опций используется наибольшее значение. Если значение *number* не указано, то применяется соответствующая машине установка по умолчанию, обычно равная 1. Указание в поле *number* значения 1 эквивалентно действию опции **-fno-align-labels**, при этом выравнивание переходов не применяется.

-falign-loops[=number]

Верхушки циклов выравниваются к границе второго типа (power 2) или к ближайшей границе выравнивания, превышающей указанное число *number*. Но только, если при этом пропускается не более *number* байт. Например, *number* имеет значение 20. Тогда в случае 32-байтного выравнивания, цикл будет выравниваться к ближайшей 32-байтной границе только при условии, что для этого придется пропустить не более 20-ти байт. Эта опция может увеличить размер вырабатываемого кода потому что пропускаемые байты заполняются пустыми операциями. Однако, в зависимости от типа машины, скорость выполнения циклов может увеличиться благодаря выравниванию адресации переходов в конце каждой итерации.

Если значение *number* не указано, то применяется машинная установка по умолчанию, обычно равная 1. Указание в поле *number* значения 1 эквивалентно действию опции **-fno-align-loops**, при этом циклы не выравниваются.

-fallow-single-precision**C**

Применяется по умолчанию. Не позволяет использование двойной точности при выполнении математических операций с плавающей точкой обычной точности. При установке опции **-traditional** все операции с плавающей точкой выполняются с двойной точностью, но данная опция оставляет возможность использования обычной точности.

-falt-external-templates**C++**

Данная опция распознается компилятором, но дальнейшая ее поддержка прекращена, (*deprecated option*). По этой опции дубликаты шаблонов могут генерироваться или не генерироваться в зависимости от расположения кода определения их оригиналов. Сейчас является предпочтительным использование директив. Подстановка шаблонов точно следует директивам `#pragma interface` и `#pragma implementation`. См. также **-fexternal-templates**.

-fargument-alias

Указывает на возможность использования в аргументах функций синонимов (*aliases*). Это означает, что два или более аргумента могут указывать на одно расположение памяти. Также возможно использование синонимов для глобально объявленных величин. Считается, что эта опция предназначена для внутреннего использования компилятором.

Применяется по умолчанию при компиляции программ на языках *C*, *C++* и *Objective-C*.

См. также опции **-fargument-noalias** и **-fargument-noalias-global**.

-fargument-noalias

Указывает, что аргументы, передаваемые функциям, не могут быть синонимами (*aliases*) друг друга. То есть, два или более аргумента вызова не могут указывать на одно расположение памяти. Однако, при этом допускается возможность использования синонимов глобально объявленных величин. Считается, что данная опция предназначена для внутреннего использования компилятором.

См. также опции **-fargument-alias** и **-fargument-noalias-global**.

-fargument-noalias-global

Указывает, что передаваемые функциям аргументы ни в коем случае не могут быть синонимами (*aliases*) друг друга. То есть, два или более аргумента вызова никогда не должны указывать на одно расположение памяти. При этом также исключается использование синонимов глобально объявленных величин. Считается, что данная опция предназначена для внутреннего использования компилятором.

Применяется по умолчанию при компиляции программ на языке *Fortran*.

См. также опции **-fargument-alias** и **-fargument-noalias**.

-fasm**C, ObjC**

Применяется по умолчанию. Эта опция разрешает использование в исходном коде ключевых слов `asm`, `inline` и `typeof`.

При компиляции программ на языке C опция `-fno-asm` отключает использование ключевых слов `asm`, `inline` и `typeof`.

При компиляции программ на языке C++ опция `-fno-asm` отключает использование только ключевого слова `typeof`. Она не оказывает действия на применение ключевых слов `asm` и `inline`, так как они являются частью языка.

На возможность использования этих ключевых слов также влияют флаги `-ansi`, `-gnu-keywords` и `-std`.

-fassume-compiled=classname

Java

Генерируемый компилятором код может зависеть от того, что определенные классы уже скомпилированы в системно-ориентированный (native) код. Опции `-fassume-compiled` (и `-fno-assume-compiled`) могут применяться последовательно для построения списка классов, которые будут считаться предварительно скомпилированными (или не скомпилированными).

-fasynchronous-unwind-tables

Генерирует неупорядоченную таблицу в формате DWARF2, если этот формат может поддерживаться предназначенной машиной. Полученная таблица затем может быть использована процессами, вызывающими не синхронизированные события. Например, отладчиком (debugger) или процессом динамического распределения памяти (garbage collector).

См. также опции `-fexceptions`, `-fnon-call-exceptions` и `-funwind-tables`.

-fautomatic

Fortran

Действует по умолчанию. Указание обратной опции `-fno-automatic` дает такой же результат, как если бы в программе при объявлении каждой локальной переменной и каждого массива использовался оператор `SAVE`. Опция не действует на групповые блоки (common blocks).

См. также `-finit-local-zero`.

-fbackslash

Fortran

Действует по умолчанию. Использование опции `-fno-backslash` отменяет возможность использования символа обратной наклонной черты "`\`" для указания escape-последовательностей в строках формата Hollerith. (Имеется в виду такое же использование этого буквенного символа, как и в языке C.) Escape-последовательности, или иначе escape-коды, в исходном коде программы заменяют специальные буквенные символы. Например, по умолчанию escape-код "`\n`" интерпретируется как символ перехода на новую строку, а escape-код "`\007`" — как символ звукового сигнала BEL (этот символ также называется "beep").

-fbadu77-intrinsics-specs

Fortran

Значение поля `specs` определяет возможность использования встроенных функций (intrinsics) дополнительного набора расширений для UNIX, которые имеют не-

корректную форму относительно определений применяемого стандарта языка *Fortran*. Возможны следующие значения *specs*:

- **enable** — Встроенные функции распознаются и их использование возможно. Это значение применяется по умолчанию.
- **hide** — Встроенные функции распознаются, но для их использования в первом вызове такой функции должен быть применен оператор **INTRINSICS**.
- **disable** — Встроенные функции распознаются, но их использование допускается только если перед именем каждой из них стоит оператор **INTRINSICS**.
- **delete** — Встроенные функции дополнительного набора для UNIX не распознаются.

-fbounds-check

Java, Fortran

При компиляции программ на языке *Java* применяется по умолчанию. Использование обратной опции **-fno-bounds-check** отключает проверку границ при обращениях к массивам. Это ускоряет индексацию массивов, но может приводить к непредсказуемому поведению программы в случае выхода за границы массива.

При компиляции с языка *Fortran* применение опции **-fbounds-check** вызывает генерирование кода, выполняющего во время выполнения программы проверки обращений к элементам массивов и к подстрокам данных типа **CHARACTER**. Этот код проверяет индексы на принадлежность к допустимому диапазону, который лежит между объявленными минимальным и максимальным значениями индексов.

См. также **-ffortran-bounds-check**.

-fbranch-probabilities

Применяет профилирующую опцию **-fprofile-arcs** для компиляции программы и затем запускает ее на выполнение. При этом создается файл, содержащий количество использования каждого блока кода. Затем программа может быть снова скомпилирована уже с опцией **-fbranch-probabilities**. И при этом информация из файла, уже записанного профилирующим кодом, может быть использована для оптимизации наиболее часто используемых ветвей программы. В случае отсутствия такого файла GCC для оптимизации может примерно оценить вероятный путь выполнения программы. Информация о проходах блоков записывается профилирующим кодом в файл, который имеет то же имя, что и файл исходного кода, только с добавлением суффикса **.da**.

См. также **-fguess-branch-probability**.

-fbuiltin

C, ObjC

Включает распознавание встроенных функций по их имени, применяется по умолчанию для *C*. Использование обратной опции **-fno-builtin** указывает, что встроенные функции языка должны распознаваться с помощью префикса **__builtin_**. Например, для обращения к встроенной версии функции **strcpy()** вы можете использовать в программе такой вызов: **__builtin_strcpy()**.

Вместо использования опции **-fno-builtin** для подавления вызова по имени всех встроенных функций, имеется возможность исключения таких вызовов для от-

дельной встроенной функции. В этом случае имя выбранной встроенной функции добавляется к опции через дефис и опция приобретает форму `-fno-built-in-function`. Например, чтобы исключить обращения по имени к встроенным функциям `bzero()` и `sqrt()` следует применить две опции:

`-fno-built-in-bzero() -fno-built-in-sqrt()`

Для языка C++ опция `-fno-built-in` действует всегда. Поэтому в C++ единственным способом непосредственного обращения к встроенной функции C является указание префикса `__builtin_`. В GNU C++ стандартная библиотека использует много встроенных функций.

См. также `-ffreestanding` и `-fnonansi-builtins`.

`-fcall-saved-register`

При указании в этой опции имени определенного регистра `register` этот регистр считается зарезервированным для хранения некоторого значения, сохраняемого в нем даже при вызове функций. Все функции, скомпилированные с этой опцией должны сохранять и восстанавливать содержимое этого регистра.

В этой опции не должны указываться регистры, имеющие фиксированное назначение. Например, регистры указателя стека (stack pointer) и указателя текущего кадра стека (frame pointer).

Имена регистров зависят от платформы, они перечислены в макросе описания машины `REGISTER_NAMES`.

См. также `-fcall-used-register` и `-ffixed-register`.

`-fcall-used-register`

Регистр машины `register`, указанный этой опцией считается свободным для размещения данных, не сохраняется при вызове функции. Регистр может использоваться для размещения данных, но его значение следует восстанавливать после вызова любой функции.

В этой опции не должны указываться регистры, имеющие фиксированное назначение. Например, не следует использовать регистры указателя стека (stack pointer) и указателя текущего кадра стека (frame pointer).

Имена регистров зависят от платформы, их список содержится в макросе описания машины `REGISTER_NAMES`.

См. также `-fcall-saved-register` и `-ffixed-register`.

`-fcaller-saves`

При этой опции в код включаются дополнительные инструкции для сохранения регистров перед вызовом функции и для восстановления их значений после вызова функции. Содержимое регистров может использоваться как при вызове функции, так и в самом коде функции. Сохраняются только те регистры, которые могут содержать полезные значения и только в тех случаях, когда сохранение и восстановление регистра выглядит предпочтительнее, чем более поздняя перезагрузка регистра непосредственно перед использованием его начального значения. Эта опция на некото-

рых машинах действует по умолчанию и всегда применяется при оптимизациях `-O2`, `-O3` и `-Og`. При необходимости она может быть отменена обратной опцией `-fno-caller-saves`.

-fcase-initcap**Fortran**

Требует, чтобы основная часть исходного кода была написана словами с заглавной буквы. (Кроме комментариев и символьных констант.) Устанавливает опции `-fintrin-case-initcap`, `-fmath-case-initcap`, `-fsource-case-preserve` и `-fsymbol-case-initcap`.

-fcase-lower**Fortran**

Требует использования в исходном тексте программы буквенных символов нижнего регистра клавиатуры (т.е. строчных букв). Эта опция устанавливает опции `-fintrin-case-any`, `-fmath-case-any`, `-fsource-case-lower` и `-fsymbol-case-any`.

-fcase-preserve**Fortran**

Сохраняет регистр всех буквенных знаков в определяемых пользователем программных символах и допускает применение как заглавных, так и строчных букв в ключевых словах и именах встроенных функций языка. Устанавливает опции `-fintrin-case-any`, `-fmath-case-any`, `-fsource-case-preserve` и `-fsymbol-case-any`.

-fcase-strict-lower**Fortran**

Требует, чтобы основная часть исходного кода была написана строчными буквами. (Кроме комментариев и символьных констант.) Устанавливает опции `-fintrin-case-lower`, `-fmath-case-lower`, `-fsource-case-preserve` и `-fsymbol-case-lower`.

-fcase-strict-upper**Fortran**

Требует, чтобы основная часть исходного кода была написана заглавными буквами. (Кроме комментариев и символьных констант.) Устанавливает опции `-fintrin-case-upper`, `-fmath-case-upper`, `-fsource-case-preserve` и `-fsymbol-case-upper`.

-fcase-upper**Fortran**

Требует использования в исходном тексте программы буквенных символов верхнего регистра клавиатуры (т.е. заглавных букв). Эта опция устанавливает опции `-fintrin-case-any`, `-fmath-case-any`, `-fsource-case-upper` и `-fsymbol-case-any`.

-fcheck-new**C++**

Вставляет код проверки указателя, возвращаемого оператором `new`, на его равенство значению `NUL`. Оператор `new` используется в C++ для выделения памяти. Обыч-

но такая проверка не требуется, версия функции `new` стандартной библиотеки C++ вызывает исключение при выходе за пределы доступной области памяти. Необходимость в проверке указателя возникает только когда функция `new` перегружена пользовательской версией, которая способна возвращать значение `NUL`.

-fcheck-references

Java

Вставляет расширяемый подстановкой (*inline*) код проверки нулевого значения указателей при обращении к объектам через ссылки на них. Обычно в этом нет необходимости, большая часть препроцессоров определяет ссылки с нулевыми указателями (*null pointer references*).

-fcommon

C

Применяется по умолчанию. Применение обратной опции `-fno-common` приводит в выделению компилятором в разделе данных (*data section*) избыточного пространства памяти для глобальных переменных. По умолчанию глобальные переменные размещаются в общем блоке (*common block*) раздела данных, при этом все ссылки на них разрешимы компоновщиком. Если одна глобальная переменная объявлена в программе более одного раза, то компоновщик все равно обеспечивает разрешение всех обращений к ней через один адрес.

Установка опции `-fno-common` может потребоваться для надежности компиляции и компоновки вашей программы при ее переносе на другую систему, не использующую GCC.

Опцию `-fno-common` не следует использовать для компиляции программ на языке *Fortran*.

См. также раздел главы 4 "Атрибуты".

-fcompile-resource=resourceName

Java

Поле `resourceName` указывает имя ресурсного файла, содержащего определения свойств и другие используемые программой ресурсы. Такой файл может компилироваться в объектный код. Обращения к нему во время выполнения программы разрешаются обработчиком ядра JVM (*core protocol handler*) с помощью запроса `core:/resourceName`.

-fcond-mismatch

C

Допускает несоответствия типов в выражениях условий.

-fconserve-space

C++

Размещает переменные, не инициализированные во время компиляции программы, в общем сегменте данных. Т.е. так, как это делается при компиляции программ на языке C. Это уменьшает размер выполняемого файла, потому что пространство для этих переменных не выделяется до загрузки программы. Этот флаг сейчас действует не для всех платформ, такое положение сложилось после того, как была добавлена поддержка размещения переменных в разделе BSS без открытия к ним общего доступа.

Внимание, предупреждение! Если скомпилированная с этой опцией программа завершается аварийно, то это может происходить из-за того, что разрушение объектов происходит дважды. Эта ситуация является следствием объединения объектов и присвоения им одного адреса.

-fconst-strings

C++

Действует по умолчанию. При определении обратной опции **-fno-const-strings** объявления литералов (literal declarations) определяются как `char *` вместо `const char *` по умолчанию. При этом для использования возможности выполнения действительной записи в объявленные с начальным значением строки буквенных символов следует также применить опцию **-fwritable-strings**.

-fconstant-strings-class=classname

ObjC

Указывает *classname* в качестве имени класса, экземпляры которого создаются для каждой строки буквенных символов, заданной в форме `@"..."`. По умолчанию *classname* имеет значение `NXConstantString`.

-fcrop-registers

По этой опции после распределения под данные всех регистров происходит отслеживание использования размещаемых в регистрах данных. При этом выполняется поиск таких мест, где можно обойтись без хранения данных в регистре, вместо этого повторяя загрузку в регистр только там, где это действительно необходимо.

Эта опция автоматически устанавливается при использовании опции **-O**, но может быть замещена обратной опцией **-fno-crop-registers**.

-fcse-follow-jumps

Действует при оптимизации CSE в случае, когда адрес назначения перехода (jump target) не может быть достигнут иначе, как действительным выполнением инструкции перехода. В этом случае при оптимизации устранении кода общих подвыражений (Common Code Elimination, CSE) выполняется упреждающее сканирование пути перехода. При этом считается, что все величины, присутствующие до выполнения перехода, остаются на своих местах и доступны из точки назначения перехода. Этот флаг устанавливается автоматически опциями **-O2**, **-O3** и **-Os**, но может быть отключен обратной опцией **-fno-cse-follow-jumps**. См. также опцию **-fcse-skip-blocks** и **--param**.

-fcse-skip-blocks

Действует при оптимизации CSE в случае, когда код тела условного оператора `if` (`if statement`) достаточно прост, и не изменяет предварительно рассчитанных величин. Процесс анализа потока общих подвыражений пропускает такой условный оператор и применяет значения предварительно рассчитанных величин к следующему оператору. Этот флаг устанавливается автоматически опциями **-O2**, **-O3** и **-Os**, но может быть отключен обратной опцией **-fno-cse-skip-blocks**. См. также опцию **-fcse-follow-jumps** и **--param**.

-fdata-sections

Каждый элемент данных в выходном ассемблерном коде размещается в собственном именованном разделе (section) данных. Имя каждого раздела наследует имя соответствующего элемента данных. Это дает эффект только на тех машинах, которые имеют компоновщик, использующий секционирование для оптимизации выделения памяти. Для применения той же оптимизации по отношению к выполняемому коду служит опция **-ffunction-sections**.

При установке опции **-fdata-sections** для машины, не поддерживающей секционирование выделения памяти, будет выдано предупредительное сообщение и опция будет игнорирована. Даже на тех машинах, которые поддерживают секционирование, применение этой опции может не дать никаких преимуществ, несмотря на оптимизацию, выполняемую компоновщиком. На деле такой подход может давать несбалансированный эффект из-за большого объема и медленной загрузки объектного кода.

Эта опция не действует при установке опции **-P** для выполнения профилирования. Также из-за реорганизации кода возможны проблемы с опцией **-g** и вообще при любой отладке.

-fdefault-inline

C++

Действует по умолчанию. Определяет автоматическое расширение подстановкой кода для функций — членов класса, код реализации которых определен внутри объявления того класса, к которому они принадлежат. По этой опции подстановка кода для таких функций применяется независимо от того, использовалось или нет ключевое слово **inline** в их объявлении. Для предотвращения автоматической подстановки должна использоваться обратная опция **-fno-default-inline**. Также см. опцию **--param**.

-fdefer-pop

Сохраненные в стеке значения регистров не выталкиваются сразу после возврата из функции, они накапливаются в стеке вместе с аргументами нескольких последовательно вызываемых функций. Начальные значения регистров восстанавливаются только после разгрузки стека. Эта опция действует по умолчанию. Для форсирования очистки стека после каждого вызова функции нужно применять обратную опцию **-fno-defer-pop**.

-fdelete-null-pointer-checks

При этой из программы убирается опции код проверки указателей на нулевое значение, если анализ потока данных показывает, что значение указателей не может быть нулевым. В некоторых вариантах среды окружения существует возможность обработки ситуации обнуления указателей (*dereference null pointer*). Поэтому опцию **-fdelete-null-pointer-checks** не следует использовать в программах, имеющих прямое отношение к проверке обнуления указателей. Этот флаг устанавливается автоматически при использовании опций **-O2**, **-O3** и **-Os**, он может быть отключен обратной опцией **-fno-delete-null-pointer-checks**.

-fdelayed-branch

Этот флаг действует только на машинах, которые имеют слоты задержки ветвлений (delayed branch slots). Он имеет отношение к загрузке и выполнению инструкций ветви кода до принятия решения о выполнении этой ветви. После вычисления условия результат вычисления инструкций может быть отброшен в зависимости от расположения инструкций и принятого решения. Флаг устанавливается каждым уровнем оптимизации, который его поддерживает. Он может быть отключен применением обратной опции `-fno-delayed-branch`.

-fdiagnostics-show-location=*where*

Предусмотрена возможность разбиения длинных сообщений диагностики (как предупреждений так и сообщений об ошибках) на несколько строк при их выводе. По умолчанию поле *where* имеет значение `once`. Оно определяет однократное включение в сообщение имени и пути расположения файла исходного кода, вызвавшего это сообщение. Значение `every-line` указывает включение информации о расположении исходного кода в каждую строку сообщения.

Опция в разных случаях ее применения действует по-разному. Она вообще имеет смысл только при ненулевом значении (или значении по умолчанию) параметра опции `-fmessage-length`.

-fdollar-ok**Fortran**

Разрешает использование в символьических ссылках (symbol names) знака "\$" ("доллар").

-fdollars-in-identifiers**C, C++**

Воспринимает знак "доллар" ("\$") как допустимый буквенный символ для использования в идентификаторах. При использовании обратной опции `-fno-dollars-in-identifiers` использование знака "\$" будет строго исключено.

Значение этого флага меняется в зависимости от платформы предназначения и языка программирования. Традиционный стандарт языка *C* разрешает использование знаков "\$", в то время как более новые стандарты не допускают этого. Поэтому, если вам нужно определить правило применения этого буквенного символа, то лучше это сделать явным образом.

-fdump-class-hierarchy[-format]**C++**

По этой опции компилятор для каждого класса выводит дамп иерархии и таблицу виртуальных функций в файл, имеющий в своем названии имя класса и суффикс `.class`. Необязательный параметр *format* может иметь одно из следующих значений:

- `address` — Выводит адрес каждого узла, этот адрес может быть использован для перекрестного сравнения с другими дампами. В том числе и с дампами, выводимыми по опции `-d`.

- **slim** — Уменьшает размер вывода за счет подавления такой информации, как код определения функций или область действия идентификаторов.
- **all** — Увеличивает размер вывода, определяя включение в дамп всей возможной информации.

-fdump-translation-unit[-format]

C, C++

По этой опции компилятор для каждого модуля выводит дерево внутреннего представления исходного кода. Информация выводится в файл, имеющий в своем названии имя исходного файла и суффикс **.tu**. Необязательный параметр **format** может иметь одно из следующих значений:

- **address** — Выводит адрес каждого узла дерева внутреннего представления исходного кода. Этот адрес может быть использован для перекрестного сравнения с другими дампами. В том числе и с дампами, выводимыми по опции **-d**.
- **slim** — Уменьшает размер вывода за счет подавления такой информации, как код определения функций или область действия идентификаторов.
- **all** — Увеличивает размер вывода, определяя включение в дамп всей возможной информации.

-fdump-tree-switch[-format]

C++

Выводит дамп различных этапов преобразования внутреннего представления дерева исходного кода на промежуточном языке. Информация выводится в файл, имя которого соответствует имени исходного файла и имеет суффикс, соответствующий значению параметра **switch**.

Параметр **switch** должен иметь одно из следующих значений:

- **original** — Выводит в файл **name.original** дерево внутреннего представления исходного кода до выполнения каких-либо преобразований на уровне промежуточного языка.
- **optimized** — Выводит в файл с именем **name.optimized** дерево внутреннего представления исходного кода после выполнения всех преобразований уровня промежуточного языка.
- **inlined** — Выводит дерево внутреннего представления исходного кода в файл с именем **name.inlined** после выполнения всех подстановок кода **inline** функций.

Необязательный параметр **format** может иметь одно из следующих значений:

- **address** — Выводит адрес каждого узла дерева. Этот адрес может быть использован для перекрестного сравнения с другими дампами. В том числе и с дампами, выводимыми по опции **-d**.
- **slim** — Уменьшает размер вывода за счет подавления такой информации, как код определения функций или область действия идентификаторов.
- **all** — Увеличивает размер вывода, определяя включение в дамп всей возможной информации.

-fdump-unnumbered

При отладке компилятора с опцией **-d** подавляет вывод в выходные файлы номеров инструкций ассемблера и номеров строк. Это упрощает использование утилиты **diff** для сравнения дампов.

-feliide-constructors**C++**

Действует по умолчанию. Упрощает генерируемый код, если он вызывает функцию, возвращающую объект значением его адреса. В результате оптимизации функция создает экземпляр объекта непосредственно в указанном расположении возвращаемого значения вместо использования конструктора копирования объекта, созданного внутри локальной области действия функции. Это может вызывать проблемы в случае, если конструктор оказывает побочное действие на результат. Отменить значение флага по умолчанию можно применением обратной опции **-fno-elide-constructors**.

-femulate-complex**Fortran**

Применяет эмуляцию арифметических действий с комплексными числами вместо использования встроенной прямой поддержки комплексной арифметики нижнего уровня GCC.

Эта опция была предусмотрена для обхода присутствовавших в предыдущих версиях ошибок с реализацией комплексной арифметики. Сейчас считается, что эти ошибки уже устранены.

-fencoding=name**Java**

Поле **name** указывает название набора буквенных символов применяемой при считывании исходных файлов кодировки. По умолчанию используется текущая установка компилятора, либо, при ее отсутствии, набор **UTF-8**.

-f enforce-eh-specs**C++**

Действует по умолчанию. GCC вырабатывает код, обрабатывающий исключения во время выполнения программы в соответствии с применяемым стандартом языка C++. Установка обратной опции **-fno-enforce-eh-specs** отключает генерирование кода обработки исключений. Это уменьшает размер скомпилированной программы.

-fexceptions

Включает поддержку обработки исключений. По этой опции компилятор генерирует дополнительный код, который вызывает и обрабатывает исключения. Без особого указания этой опции она автоматически применяется при компиляции программ на таких языках как *Ada*, *Java* и *C++*. Т.е. тех языках, стандарты которых предусматривают использование обработки исключений.

Код обработки исключений достаточно оптимален и не оказывает особого влияния на скорость выполнения программы. Но отключение генерирования этого кода опцией **-fno-exceptons** может существенно уменьшить размер программы на C++, которая не использует обработку исключений.

См. также `-fnon-call-exceptions`, `-funwind-tables` и `-fasynchronous-unwind-tables`.

-fexpensive-optimizations

Этот флаг включает применение нескольких оптимизаций вообще довольно эффективных, но требующих серьезного увеличения затрат времени на компиляцию программы. Например, общая оптимизация удаления общих подвыражений CSE (Common Subexpression Elimination) при этом флаге запускается снова после прохода удаления общих глобальных подвыражений. Некоторые другие оптимизации применяются глубже, чем обычно по умолчанию. Этот флаг устанавливается автоматически при определении опций `-O2`, `-O3` и `-Os`, но может при необходимости быть отключен применением обратной опции `-fno-expensive-optimizations`.

-fexternal-templates

C++

Опция распознается компилятором, но ее поддержка прекратилась, (deprecated option). В соответствии с этой опцией экземпляры шаблона кода могут подставляться или нет в зависимости от расположения кода его определения. В последних версиях GCC экземпляры шаблона включаются в точном соответствии с директивами `#pragma interface` и `#pragma implementation`. См. также опцию `-falt-external-templates`.

-ff2c

Fortran

Опция действует по умолчанию и назначает генерирование кода, совместимого с требованиями `f2c`. Утилита `f2c` применяется для трансляции исходного кода с языка *Fortran* на язык *C*.

Установка обратной опции `-fno-f2c` подавляет выработку совместимого с `f2c` кода и вместо этого применяет соглашения о вызовах GNU. Это не влияет на взаимодействие кода с библиотекой `libf2c`, если только отдельное определение не закрывает использование в качестве аргументов встроенных функций этой библиотеки. При использовании опции `-fno-f2c` ее следует применять при компиляции всех модулей, используемых в компоновке одной программы.

-ff2c-intrinsics-specs

Fortran

Значение поля `specs` определяет статус специфических встроенных функций (intrinsics) расширения `f2c`, которые имеют некорректный формат относительно определений применяемого стандарта языка. Утилита `f2c` применяется для трансляции исходного кода с языка *Fortran* на язык *C*. Возможны следующие значения `specs`:

- `enable` — Встроенные функции набора расширения `f2c` распознаются и их использование возможно. Это значение применяется по умолчанию.
- `hide` — Встроенные функции `f2c` распознаются, но для их использования при первом вызове такой функции следует применить оператор `INTRINSICS`.
- `disable` — Встроенные функции `f2c` распознаются, но их использование допускается только если перед именем каждой из них стоит оператор `INTRINSICS`.
- `delete` — Встроенные функции набора `f2c` не распознаются.

-ff66**Fortran**

Компилируемый исходный код считается соответствующим диалекту Fortran 66. См. также опции **-ff77** и **-ff90**.

-ff77**Fortran**

Компилируемый исходный код считается соответствующим диалекту Fortran 77. При этом также считается что этот диалект соответствует входным требованиям утилиты **f2c** и с ее помощью возможна трансляция исходного кода с языка *Fortran* на язык *C*. См. также опции **-ff66** и **-ff90**.

-ff90**Fortran**

При этой опции компилятор будет распознавать конструкции, свойственные диалекту Fortran 90. Применение опций **-fvxt** и **-ff90-intrinsics-specs** дает возможность использования дополнительных конструкций диалекта Fortran 90. См. также опции **-ff66** и **-ff77**.

-ff90-intrinsics-specs**Fortran**

Значение поля *specs* определяет возможность использования специфических встроенных функций (intrinsics) диалекта Fortran 90, которые имеют форму, конфликтующую с основными определениями стандарта языка *Fortran*. Возможны следующие значения *specs*:

- **enable** — Встроенные функции диалекта Fortran 90 распознаются и использование их возможно. Это значение применяется по умолчанию.
- **hide** — Встроенные функции диалекта распознаются, но для их использования при первом вызове такой функции следует применить оператор **INTRINSICS**.
- **disable** — Встроенные функции диалекта распознаются, но их использование допускается только если перед именем каждой из них стоит оператор **INTRINSICS**.
- **delete** — Встроенные функции диалекта Fortran 90 не распознаются.

-ffast-math

При этой опции некоторые математические вычисления выполняются быстрее за счет отступления от требований стандартов ISO и IEEE. Например, при установке этой опции считается что функции **sqrt()** не будут передаваться отрицательные аргументы или недопустимые значения с плавающей точкой. И, соответственно, при этом будет отключена обработка таких ситуаций.

Применение этой опции определяет макрос препроцессора **__FAST_MATH__** и устанавливает опции **-fno-math-errno**, **-funsafe-math-optimizations** и **-fno-trapping-math**.

При применении обратной опции **-fno-fast-math** автоматически устанавливается опция **-fmath-errno**.

-ffixed-register

Определенный этой опцией регистр процессора с именем `register` считается фиксированным регистром, он не может быть выделен для использования компилятором. Это не отменяет использование этого регистра для выполнения действий с основными фиксированными регистрами, такими как указатель вершины стека, указатель текущего кадра стека, и т.п.

Имена регистров зависят от платформы, они перечислены в макросе описания машины `REGISTER_NAMES`.

См. также `-fcall-used-register` и `-fcall-saved-register`.

-ffixed-form**Fortran**

Действует по умолчанию. Указывает, что исходный код написан в соответствии с традиционным фиксированным форматом языка *Fortran*, а не свободную форму наподобие исходного кода стандарта Fortran 90.

Равносильна опции `-fno-free-form`.

-ffixed-line-length-len**Fortran**

Поле `len` определяет номер последнего столбца исходного кода, за которым будут игнорироваться все символы в исходном файле фиксированного формата.

Указание опции `-ffixed-line-length-0` или `-ffixed-line-length-none` снимает ограничение на длину обрабатываемой компилятором строки входного исходного файла.

По умолчанию `len` имеет значение **72**. В этом случае при ширине страницы в 80 столбцов остающееся поле из 8 символов используется для последовательной нумерации строк исходного кода.

-ffloat-store

При этой опции компилятор не выделяет регистры общего назначения для хранения значений с плавающей точкой. На некоторых машинах это позволяет использовать специальные регистры, которые имеют более высокую точность представления чисел с плавающей точкой, чем это предусмотрено стандартом языка компилируемой программы. Благодаря этому выдерживается более высокая точность представления чисел, чем позволяет оперирование числами с сохранением их в памяти машины.

По умолчанию действует обратная опция `-fno-float-store`, разрешающая использование общих регистров.

Этот флаг будет действительно полезен только если ваша программа должна соответствовать требованиям стандарта IEEE по точности вычислений с плавающей точкой.

-ffor-scope**C++**

По умолчанию действуют установки, соответствующие применяемому стандарту языка. Эта опция определяет область видимости переменных, которые объявляются в разделе инициализации оператора цикла `for`.

Указание опции **-ffor-scope** ограничивает видимость этих переменных областью тела цикла.

Применение опции **-fno-for-scope** ограничивает область видимости переменных цикла от места их объявления до точки закрытия блока кода, который содержит оператор **for**. Следующий пример будет при этой опции компилироваться без сообщения об ошибке:

```
#include <stdio.h>
int main (int argc,char *argv[])
{
    for(int i=0; i<10; i++) {
        printf("Loop one %d\n",i);
    }
    printf("Out of loop %d\n",i);
    return(0);
}
```

-fforce-addr

Для выполнения арифметических операций с адресом, он должен быть скопирован в регистр. Эта опция обычно несколько ускоряет работу программы, потому что требуемый адрес зачастую уже находится в регистре, благодаря этому отпадает необходимость его повторной загрузки. По умолчанию применяется обратная опция **-fno-force-addr**. См. также **-fforce-mem**.

-fforce-classes-archive-check

Java

Опция включает принудительную проверку присутствия в файле класса `java.lang.Object` атрибута, указывающего на компиляцию этого класса компилятором GNU. Этот атрибут имеет нулевую длину и имя `gnu.gcj.gcj-compiled`. Атрибут будет проверяться во всех случаях, кроме компиляции в байт-код Виртуальной Машины Java.

-fforce-mem

Все значения для выполнения над ними арифметических операций должны предварительно быть скопированы в регистры. Эта опция несколько ускоряет работу программы, потому что требуемые значения часто уже находятся в регистрах и в таком случае отпадает необходимость их повторной загрузки.

Этот флаг устанавливается автоматически при использовании опций **-O2**, **-O3** и **-Os**, но может при необходимости быть отключен применением обратной опции **-fno-force-mem**. См. также опцию **-fforce-addr**.

-ffortran-bounds-check

Fortran

Вызывает генерирование дополнительного кода, который во время выполнения программы проверяет индексы массивов и обращений к подстрокам переменных типа `CHARACTER` на их соответствие допустимому диапазону между объявленными минимальным и максимальным значениями индекса.

Имеет то же значение, что и опция **-fbounds-check**.

-ffree-form**Fortran**

Указывает, что исходный код программы имеет свободный формат, наподобие применяемого для стандарта Fortran 90.

Эта опция равносильна **-fno-fixed-form**.

-ffreestanding**C**

Сообщает компилятору, что вырабатываемая им программа должна выполняться в отдельной среде окружения (freestanding environment). При этом она может не иметь доступа к стандартной системной библиотеке и ее выполнение не обязательно должно начинаться с функции `main()`. Эта опция автоматически устанавливает опцию **-fno-built-in**.

Применение этой опции использованию опции **-fno-hosted**.

-ffunction-cse

Действует по умолчанию. Вызовы функций выполняются с записью адреса функции в регистр. При использовании обратной опции **-fno-function-cse** подразумевается, что каждый оператор, выполняющий вызов функции, должен содержать адрес вызываемой функции. Применяется по умолчанию значение этого флага позволяет вырабатывать более эффективный код.

См. также опцию **--param**.

-ffunction-sections

По этой опции компилятор в ассемблерном выходе размещает каждую функцию в собственном именованном разделе. Название каждого такого раздела наследует имя соответствующей функции. Это дает преимущество только на тех машинах, которые имеют компоновщик, поддерживающий секционирование кода для оптимизации выделения памяти. Для применения такой же оптимизации по отношению к данным см. опцию **-fdata-sections**.

При установке опции **-ffunction-sections** для машины, не поддерживающей секционирование выделения памяти, будет выдано предупредительное сообщение и эта опция будет игнорирована. Даже на тех машинах, которые поддерживают секционирование, применение этой опции может не дать никакого преимущества, несмотря на возможность оптимизации компоновщиком. На деле такой подход может давать несбалансированный эффект из-за большего объема и медленной загрузки выполняемого объектного кода.

Эта опция не действует при установке опции **-P** для выполнения профилирования. Из-за реорганизации кода возможны проблемы при использовании **-ffunction-sections** совместно с опцией **-g**, как и вообще при любой отладке.

См. также раздел главы 4 "Атрибуты".

-fgcse

Выполняет оптимизацию CSE исключения общих глобальных подвыражений. Эта опция может полностью разупорядочить код в случае применения в программе опе-

раторов `goto` с вычисляемыми адресами. Опция автоматически устанавливается при использовании уровня оптимизаций `-O2`, `-O3` и `-Os`. При необходимости она может быть отменена обратной опцией `-fno-gcse`.

См. также `--param`.

-fgcse-lm

Выполняет оптимизацию CSE исключения общих глобальных подвыражений с определением операций загрузки и сохранения данных внутри цикла в случаях, когда операция загрузки не может быть вынесена перед началом цикла.

Опция автоматически устанавливается опциями `-O2`, `-O3` и `-Os`. При необходимости она может быть отключена обратной опцией `-fno-gcse-lm`.

См. также `--param`.

-fgcse-sm

Выполняет оптимизацию CSE исключения общих глобальных подвыражений с определением операций загрузки и сохранения данных внутри цикла, когда операция сохранения не может быть вынесена за конец цикла.

Опция автоматически устанавливается опциями `-O2`, `-O3` и `-Os`. При необходимости она может быть отключена обратной опцией `-fno-gcse-sm`.

См. также `--param`.

-fglobals

Fortran

Действует по умолчанию. Включает диагностику ситуации конфликта глобальных имен, когда уже имеется определенная с тем же именем процедура, использующая другие типы аргументов вызова.

Указание обратной опции `-fno-globals` отключает эту диагностику. При этом также отключается подстановка кода, что позволяет в ряде случаев избежать аварийного завершения компиляции из-за некорректной выработки кода.

-fgnu-intrinsics-specs

Fortran

Значение поля `specs` определяет возможность использования специфических встроенных функций (intrinsics) GNU, которые имеют некорректную форму относительно определений применяемого стандарта языка *Fortran*. Возможны следующие значения `specs`:

- `enable` — Встроенные функции расширения GNU распознаются и использование их возможно. Это значение применяется по умолчанию.
- `hide` — Встроенные функции GNU распознаются, но для их использования при первом вызове такой функции следует применить оператор `INTRINSICS`.
- `disable` — Встроенные функции GNU распознаются, но их использование допускается только если перед именем каждой из них стоит оператор `INTRINSICS`.
- `delete` — Встроенные функции расширения GNU не распознаются компилятором.

-fgnu-keywords**C++**

Действует по умолчанию. Обратная опция **-fno-gnu-keywords** отключает использование ключевого слова `typeof`. См. также **-ansi**.

-fgnu-linker

Действует по умолчанию. Использование обратной опции **-fno-gnu-linker** означает, что компоновщик GNU использоваться не будет. При этом не будет генерироваться код глобальной инициализации (такой, как основные конструкторы и деструкторы классов). При использовании стороннего компоновщика необходимо применение утилиты `collect2` для того чтобы удостовериться, что компоновщик подключает все требуемые конструкторы и деструкторы. На системах, заведомо того требующих, `gcc` конфигурируется с автоматическим использованием утилиты `collect2`.

-fgnu-runtime**ObjC**

Для многих систем применяется по умолчанию. Опция указывает компилятору генерировать код с использованием среды выполнения программ (runtime environment) GNU Objective-C.

-fguess-branch-probability

Действует по умолчанию. Определяет, что GCC будет проводить оценку сравнительной частоты использования ветвей кода и выполнять соответствующую оптимизацию.

GCC для оценки вероятностей ветвления использует случайное моделирование. Поэтому при компиляции одного и того же исходного кода возможно получение различных вариантов выходного объектного кода. Для выключения действия этой опции и получения при каждой компиляции одинакового выходного кода следует указывать обратную опцию **-fno-guess-branch-probability**. Другими опциями, которые также могут приводить к выработке неоднозначного выхода компилятора, являются **-fbranch-probabilities** и **-fprofile-arcs**.

-fhash-synchronization**Java**

По этой опции функции `Java synchronize`, `wait` и `notify` сохраняются в общей хэш-таблице (hash table) вместо размещения их внутри каждого объекта.

-fhosted**C**

Компилируемая программа предназначена для запуска в "дружественной" среде окружения (hosted environment). При этом предполагается полная доступность всех функций стандартной библиотеки. Главная процедура `main()` должна иметь тип возвращаемого значения `int`. При использовании этой опции автоматически устанавливается опция **-fbuiltin**.

Эта опция равнозначна опции **-fno-freestanding**.

-fident**Pre**

Действует по умолчанию. При указании обратной опции **-fno-ident** препроцессор будет игнорировать директивы **#ident**.

-fimplement-inlines**C++**

Применяется по умолчанию. Для inline-функций, т.е. генерируемых подстановкой кода, создается один экземпляр не подставляемого скомпилированного кода функции в месте их объявления. Это делается на случай их вызова из модулей, которые не содержат кода для их подстановки. При указании обратной опции **-fno-implement-inlines** для inline-функций под управлением директивы **#pragma implementation** будет подавляться выработка отдельного объектного кода реализации функции. Если отдельный код реализации функции не генерируется, то каждый вызов такой функции обязательно должен подставляться кодом ее определения.

-fimplicit-templates**C++**

Действует по умолчанию. Каждая ссылка на шаблон кода (template) или содержащаяся в шаблоне функцию будет генерировать отдельный экземпляр шаблона, если он еще не создан предыдущим обращением. Обратная опция **-fno-implicit-templates** подавляет явное (implicit) генерирование экземпляров шаблона, включая также и inline-шаблоны (т.е. расширяемые постановкой кода).

См. также **-fimplicit-inline-templates**.

-fimplicit-inline-templates**C++**

Действует по умолчанию. Каждая ссылка на шаблон кода (template) или на функцию, которая содержится в шаблоне, будет генерировать подстановку кода шаблона в случае, если такая подстановка еще не была выполнена по предыдущему обращению. Обратная опция **-fno-implicit-inline-templates** подавляет явную (implicit) подстановку кода шаблона, включая также и inline-шаблоны (т.е. шаблоны, расширяемые постановкой кода).

См. также **-fimplicit-templates**.

-finhibit-size-directive

Подавляет выработку директив ассемблера **.size** и других директив, которые могут вызывать проблемы в случаях, когда код функции при загрузке в память разбивается на несколько разделенных и отстоящих друг от друга частей.

Опция была специально предусмотрена для компиляции исходного файла **crtstuff.c**, который входит в состав GCC. Другие варианты ее использования не предусматривались.

-finit-local-zero**Fortran**

При этой опции все локальные переменные, определяемые в пределах модуля компиляции, инициализируются нулевыми значениями. Это не относится к полям общих блоков данных (common blocks) и аргументам вызова функций.

Рекомендуется использовать совместно с опцией `-fno-automatic` для предотвращения сбоев при автоматической инициализации переменных во время выполнения программы.

-finit-priority

Опция предназначена для внутреннего использования компилятором. Она определяет порядок инициализации переменных во время выполнения программы.

-finline

C, C++, ObjC, Fortran

Действует по умолчанию. Разрешает при объявлении функций использование ключевого слова `inline` для указания, что код определения такой функции должен подставляться в месте ее вызова. При указании обратной опции `-fno-inline` компилятор игнорирует использование в исходном коде программы ключевых слов `inline`. Учтите, что подстановка кода функций применяется только при назначении с помощью опции `-O` некоторого уровня оптимизации. См. также опцию `--param` и раздел главы 4 "Атрибуты".

-finline-functions

Разрешает компилятору самостоятельно выбирать функции достаточно малой степени сложности для их расширения подстановкой кода в местах их вызова. Если при этом функция объявлена таким образом, что все ее вызовы могут быть определены внутри модуля, то отдельный код тела этой функции не создается, потому что в действительности он никогда вызываться не будет. Хорошим примером такого объявления на языке `C` могут служить функции с атрибутом `static`, вызовы которых в общем случае допускаются только в пределах одного исходного файла.

Эта опция автоматически устанавливается при применении оптимизации `-O3`, если при этом не установлен флаг `-fno-inline-functions...`.

-finline-limit=size

При этой опции компилятор не будет подставлять функции кодом их определения, если размер этого кода превышает указанное в поле `size` количество псевдоинструкций. Если значение `size` не указано, то по умолчанию оно равно 600. См. также опцию `--param`.

-finstrument-functions

Вставляет вызовы особой встроенной функции в точках входа и выхода каждой вызываемой функции. Далее приводятся прототипы вызовов этой встроенной профилирующей функции:

```
void __cyg_profile_func_enter(void *this_fn,void *call_site);
void __cyg_profile_func_exit(void *this_fn,void *call_site);
```

Аргумент `this_fn` содержит адрес текущей вызываемой функции, определяемый по информации из таблицы программных символов. Аргумент `call_site` идентифицирует вызывающую подпрограмму. (На некоторых платформах информация, передаваемая значением переменной `call_site`, может быть недоступной.)

Если обращения к функции подставляется кодом ее определения, то вызовы профилирующей функции помещаются перед и после вставляемого кода подстановки. При этом для возможности идентификации вызываемой функции должна существовать и быть доступной ее отдельная, т.е. не подставляемая кодом определения (non-*inline*), версия. Даже если все ее вызовы генерируются подстановкой.

Для предотвращения расширения вызовов функции подстановкой ее кода при ее объявлении следует использовать атрибут `no_instrument_function`. Применение этого атрибута может быть необходимым для обработчиков прерываний и для функций, из которых не могут вызываться профилирующие подпрограммы.

См. также раздел главы 4 "Атрибуты".

-fintrin-case-spec

Fortran

Поле `spec` указывает регистр буквенных знаков (строчные/заглавные), используемый в именах встроенных функций (intrinsic names). Возможны следующие значения `spec`:

- `initcap` — Слово начинается с заглавной буквы и все остальные буквы — строчные.
- `upper` — Все слово должно быть написано заглавными буквами.
- `lower` — Все слово должно быть написано строчными буквами. Действует по умолчанию.
- `any` — Допускается любой порядок использования строчных и заглавных букв.

См. также `-fmath-case-`, `-fsource-case-`, `-fsymbol-case-` и `-fcase-`.

-fjni

Java

Для компиляции при этой опции системно-ориентированных методов (native methods) используется интерфейс JNI вместо применяемого по умолчанию интерфейса CNI. При этом также генерируются заглушки (stubs) — исходные файлы для определений системно-ориентированных методов JNI.

-fkeep-inline-functions

Компилятор будет генерировать тело функции даже если все обращения к ней расширяются подстановкой кода (*inline*) и действительные вызовы этой функции отсутствуют. По умолчанию действует обратная опция `-fno-keep-inline-functions`, по этой опции при отсутствии действительных вызовов отдельный код определения функции не генерируется.

-fkeep-static-consts

Если не применяются некоторые из уровней оптимизации, то опция действует по умолчанию. По этой опции всегда выделяется память для размещения значений локальных констант, обращения к которым возможны только в пределах своего компиляционного модуля (*private constants*). Память выделяется даже при отсутствии действительных обращений к ним. Для предотвращения выделения памяти неиспользуемым константам следует использовать обратную опцию `-fno-keep-static-consts`.

-fleading-underscore

Эта опция применяет модификацию имен символов, которые записываются в объектный файл. К началу таких имен символов добавляется знак подчеркивания "_". Действие этой опции можно отменить обратной опцией **-fno-leading-underscore**.

Допускается использование этой опции при выполнении попыток компоновки с наследуемым ассемблерным кодом (legacy assembler code).

-fmath-case-spec**Fortran**

Поле *spec* указывает регистр буквенных знаков (строчные/заглавные), используемый в ключевых словах (keywords) языка *Fortran*. Возможны следующие значения *spec*:

- **initcap** — Слово начинается с заглавной буквы и все остальные буквы — строчные.
- **upper** — Все слово должно быть написано заглавными буквами.
- **lower** — Все слово должно быть написано строчными буквами. Действует по умолчанию.
- **any** — Допускается любой порядок использования строчных и заглавных букв.

См. также **-fintrin-case-**, **-fsource-case-**, **-fsymbol-case-** и **-fcase-**.

-fmath-errno

Применяется по умолчанию. Код ошибки результата вычисления таких математических функций как `sqrt()` записывается в глобальную переменную с именем `errno`. Использование обратной опции **-fno-math-errno** отменяет использование `errno`. Это может повлиять на обработку исключений, применяемую в соответствии со стандартом IEEE.

См. также **-ffast-math**.

-fmem-report

По завершению компиляции выводится подробный отчет об использовании памяти для размещения каждого типа данных. В листинг также включается информация и о других выделениях памяти, используемой скомпилированной программой.

-fmemoize-lookups**C++**

Кэширует последние внутренние обращения к таблице символов (internal symbol lookups) для уменьшения затрат времени на следующие обращения.

-fmerge-all-constants

Эта опция автоматически применяет опцию **-fmerge-constants**. Кроме того, она применяет слияние дубликатов для строкового типа (strings) и для массивов (arrays). Стандарты языков C и C++ требуют выделения отдельного расположения для всех элементов данных, поэтому использование этой опции может приводить к выработке объектного кода, несоответствующего стандартам.

-fmerge-constants

Выполняется попытка слияния значений для всех типов констант, кроме строк. Слияние означает, что для всех констант, имеющих одинаковое значение, в памяти размещается только одна копия их значения. Опция действует по умолчанию при включении любого уровня оптимизации. Обратная опция **-fno-merge-constants** разрешает слияние констант только в пределах одного компиляционного модуля.

-fmessage-length=*size*

Применяет форматирование сообщений об ошибках, выводимых компилятором. Сообщения разбиваются на строки, длина которых не превышает *size*. При указании значения 0 ограничение длины строки вывода не применяется, каждое сообщение выводится в одну строку. По умолчанию применяется значение 72 для языка C++, и 0 — для всех остальных языков. В некоторых случаях реализация этой опции отсутствует.

-fmil-intrinsics-*specs***Fortran**

Значение поля *specs* определяет статус специфических встроенных функций (intrinsics) стандарта MIL-STD-1753. Возможны следующие значения *specs*:

- **enable** — Встроенные функции MIL-STD-1753 распознаются и использование их возможно. Это значение применяется по умолчанию.
- **hide** — Встроенные функции MIL-STD-1753 распознаются, но для их использования при первом вызове такой функции следует применить оператор **INTRINSICS**.
- **disable** — Встроенные функции MIL-STD-1753 распознаются, но их использование допускается только если перед именем каждой из них стоит оператор **INTRINSICS**.
- **delete** — Встроенные функции стандарта MIL-STD-1753 не распознаются компилятором.

-fmove-all-movables

Все инвариантные выражения выносятся за пределы кода цикла (loop). При этом вырабатываемый код может как улучшиться, так и ухудшиться. Это зависит от структуры и вложенности циклов в исходном коде. По умолчанию ко всем языкам, кроме **Fortran**, применяется обратная опция **-fno-move-all-movables**.

См. также опцию **-freduce-all-givs**.

-fms-extensions**C++**

Подавляет вывод предупредительных сообщений при использовании своеобразных конструкций, определенных для MFS (Microsoft Foundation Class). Как то явное определение значением при объявлении переменных типа **int**, или нестандартный синтаксис получения адресов методов класса.

-fnext-runtime

Генерирует выходной код, совместимый со средой выполнения программ системы NeXT. Действует по умолчанию при компиляции для систем, основанных на NeXT, таких как Darwin и Mac OS X.

-fno-*

Любая опция, которая начинается с префикса "**-fno-**" имеет противоположную форму своего представления без подстроки "по-", то есть соответствующую обратную ей флаговую опцию с префиксом "**-f**". Именно такой формой она и представлена в алфавитном списке настоящего Приложения. Например, информацию об опции **-fno-for-scope** вы сможете найти в описании опции **-ffor-scope**.

Было бы неверно при перечислении в этом списке использовать формы опций с префиксом "**-fno-**". Не для всех опций, которые начинаются с "**-f**", имеются противоположные формы с префиксом "**-fno-**".

-fnon-call-exceptions

Генерирует код, который делает возможным перехват инструкций для вызова исключений. (Таких инструкций, как, например, неправильные операции над числами с плавающей точкой или недопустимая адресация обращений к памяти.) Эта опция не может применяться универсально для всех платформ. Она требует наличия в среде выполнения программ специфической поддержки аппаратной базы.

Опция оказывает действие только на использование сигналов аппаратных прерываний, и не распространяется на применение общих сигналов. (Таких сигналов, как **SIGALM** или **SIGTERM**.)

См. также **-fexceptions**, **-funwind-tables** и **-fasynchronous-unwind-tables**.

-fnonansi-builtins

C++

Действует по умолчанию. При использовании **-fno-nonansi-builtins** отключается автоматическое генерирование встроенных функций, использование которых не предусмотрено стандартами "ANSI C" и "ISO C".

См. также **-ansi** и **-fbuiltin**.

-fomit-frame-pointer

Не сохраняет в регистре указатель кадра стека (frame pointer) для тех функций, которые не нуждаются в его использовании. При этом пропускается код сохранения и восстановления адреса и освобождается дополнительный регистр для общего использования. Опция устанавливается автоматически при применении опции **-O** для всех уровней оптимизации, но только если отладчик (debugger) способен работать без указателя кадра стека. Если вы используете отладчик, не поддерживающий подобный режим отладки, то для применения этой опции ее следует указывать явным образом. На некоторых plataформах указатель кадра стека не используется, в таких случаях опция не оказывает никакого действия. По умолчанию применяется обратная опция **-fno-omit-frame-pointer**.

-fonetrip**Fortran**

По этой опции каждый цикл DO будет выполняться по крайней мере один раз. Условие цикла проверяется после выполнения его кода, а не перед его выполнением.

Некоторые компиляторы до стандарта Fortran 77 ставили проверку условия в конце цикла, в то время как другие — в начале цикла. Начиная со стандарта Fortran 77 все компиляторы проверяют условие цикла перед его выполнением. Это означает, что когда вычисление выражения условия цикла дает результат "ложь", то тело цикла не будет выполнено ни разу.

-foperator-names**C++**

Действует по умолчанию. При обратной опции **-fno-operator-names** не допускается использование ключевых слов **and**, **bitand**, **bitor**, **compl**, **not**, **or** и **xor** — эти ключевые слова не распознаются компилятором. Вместо них следует употреблять соответствующие альтернативные формы этих операторов **&&**, **&**, **|**, **~**, **!**, **||** и **^**. Это предотвращает неправильную компиляцию некоторых старинных исходников, освобождая имена перечисленных ключевых слов для их использования в других целях.

-foptimize-register-move

Оптимизирует распределение регистров, изменяя назначение тех регистров, которые используются в операциях перемещения (move) данных из одного расположения памяти в другое. Такая оптимизация особенно эффективна на машинах, имеющих инструкции прямого перемещения данных в памяти. Флаг автоматически устанавливается при оптимизациях **-O2**, **-O3** и **-Os**, при необходимости его отключения применяется обратная опция **-fno-optimize-register-move**.

-foptimize-sibling-calls

Оптимизация вложенных вызовов типа "sibling call". В некоторых случаях, когда функция в последнем операторе рекурсивно вызывает сама себя (tail recursive call) или использует вложенный вызов другой функции, логика программы может быть преобразована так, чтобы вместо такой функции использовать некоторую циклическую структуру. Этот флаг автоматически устанавливается опциями оптимизации **-O2**, **-O3** и **-Os**, при необходимости его отключения применяется обратная опция **-fno-optimize-sibling-calls**.

Вот пример функции с рекурсивным вызовом в последнем операторе:

```
int rewhim(int x,int y) {
    ...
    return(rewhim(x+1,y));
}
```

При оптимизации этого кода вместо повторного вызова той же функции может быть поставлена команда, которая выполняет переход в начало функции. Следующий пример кода показывает схожую ситуацию с вложенным вызовом функции в последнем операторе:

```
int whim(int x,int y) {
    ...
    return(wham(x+1,y));
}
```

Это более общий случай ситуации типа "sibling call". Здесь требуется вложенный вызов функции `wham()`. Оптимизация может удалить текущий кадр стека функции `whim()`, при этом функция `wham()` будет возвращать свое значение непосредственно в процедуру, вызывающую `whim()`.

-foptimize-static-class-initialization

Java

Действует по умолчанию. В случаях, когда применяется оптимизация уровня `-O2`, `-O3` или `-Os`, и компилятор вырабатывает выход в объектном формате вместо байт-кода JVM, приводит к тому, что статические классы инициализируются при их первом использовании. Для отключения этого вида оптимизации применяется обратная опция `-fno-optimize-static-class-initialization`.

-foptional-diags

C++

Действует по умолчанию. Определяет вывод всех диагностических сообщений. При указании обратной опции `-fno-optional-diags` будет подавляться вывод компилятором дополнительных диагностических сообщений, не требуемых стандартом языка C++.

--for-assembler *optionlist*

Asm

То же, что и опция `-Wa`.

--for-linker *option*

Linker

То же, что и опция `-Xlinker`.

--force-link *name*

Linker

То же, что и опция `-u`.

-foutput-class-dir=*directory*

Java

При использовании совместно с опцией `-C` выводимые компилятором файлы классов сохраняются в каталоге `directory` (или соответствующем подкаталоге ниже указанного полем `directory` расположения). По умолчанию для вывода этих файлов в качестве базового каталога используется текущий.

-fpack-struct

Упаковывает поля членов структур (имеется в виду агрегатный тип `struct`) так, что между ними не остается требуемых для выравнивания промежутков.

Это может приводить к снижению эффективности кода, содержащего обращения к полям структур. Иногда приводит к нарушению совместимости с системными библиотеками.

Опцию `-fpack-struct` не следует применять при компиляции программ на языке *Fortran*.

См. также раздел главы 4 "Атрибуты".

-fpedantic

Fortran

Имеет то же значение, что и опция **-pedantic**, но применяется исключительно при компиляции программ на языке *Fortran*.

-fpeephole

Действует по умолчанию, при необходимости может быть отключена обратной опцией **-fno-peephole**. Опция **-fpeephole** применяет оптимизацию "peephole optimization" на этапе вывода компилятором ассемблерного кода. При этой оптимизации выполняется проверка соответствия предназначаемой машины и применяемого набора инструкций и замена в ассемблерном коде неэффективных последовательностей более продвинутыми инструкциями предназначаемой машины. Этот флаг действует только при назначении любого из уровней оптимизации опцией **-O**. Опция **-fpeephole** является зависимой от платформы и на ряде платформ может не иметь действия.

-fpeephole2

Приводит в действие оптимизацию типа "peephole optimization" на уровне RTL-кода. Эта оптимизация выполняется после распределения регистров, но до задействования планировщика инструкций (scheduling phase). Она состоит в специфичной по отношению к аппаратной платформе трансляции одного набора инструкций RTL в другой набор RTL-инструкций. Опция **-fpeephole2** является зависимой от платформы и может не иметь действия на ряде платформ. Этот флаг устанавливается автоматически при применении опций оптимизации **-O2**, **-O3** и **-Os**, при необходимости его можно отключить обратной опцией **-fno-peephole2**.

-fpermissive

C++

По этой опции во всех случаях отступлений от стандарта компилятор вместо сообщений об ошибках выдает предупреждения. Если не применяется **-fpermissive** или **-pedantic**, то по умолчанию действует опция **-pedantic-errors**.

-fpic

Компилятор генерирует независимый от положения в памяти, перемещаемый объектный код (position independent code, PIC). Такой формат необходим для получения модулей, используемых в составе динамической разделяемой библиотеки (shared library). Вся внутренняя адресация строится с использованием глобальной таблицы смещений (global offset table, GOT). При определении любого адреса содержащееся в таблице значение складывается с начальным адресом загрузки кода.

Опция используется при компиляции модулей, предназначенных для сохранения в разделяемых объектных библиотеках для их последующей динамической загрузки оттуда и использования программами во время их выполнения.

Если при использовании опции **-fpic** компоновщик выдает сообщение ошибки о том, что перемещаемый объектный код не работает, то следует перекомпилировать исходник с опцией **-fPIC**.

Некоторые системы имеют ограничение на размер таблицы смещений. Для процессора "Motorola m88k" предельный размер таблицы равен 16к, для "m68k" и "RS/6000" он равен 32к, для "Sparc" — 8к. PIC-код требует наличия определенной аппаратной поддержки и вследствие этого может работать только на платформах, имеющих такую поддержку.

См. также **-fPIC** и **-shared**.

-fPIC

Эта опция имеет то же значение, что и **-fpic**, но у нее есть дополнительные возможности, позволяющие обходить действующие на некоторых платформах ограничения размера таблиц смещений перемещаемого объектного кода. Такие ограничения действуют на машинах Motorola m88k, m68k и Sparc.

См. также **-fpic** и **-shared**.

-fppc-struct-return

Генерирует код, который всегда возвращает структуры данных (*structures*) сохранением их в памяти, даже если размер структур позволяет записывать их в регистр. Применение тех или иных соглашений о передаче структур через память или через регистры зависит от платформы.

Несмотря на некоторое снижение эффективности вырабатываемого кода, эта опция может быть очень полезной для обеспечения совместимости с другими компиляторами.

При компиляции программ на языке *Fortran* эта опция применяется только в случае компоновки с библиотекой *libg2c*, если эта библиотека тоже была скомпилирована с опцией **-fppc-struct-return**.

См. также **-freg-struct-return**.

-fprefetch-loop-arrays

Генерируются инструкции упреждающей выборки (*prefetch*) массивов для повышения производительности циклических вычислений. (Эта опция работает только при наличии соответствующей аппаратной поддержки.)

-fpreprocessed

Pre

Предобработка исходного кода не применяется, даже если имена входных файлов имеют суффиксы, показывающие на необходимость их обработки препроцессором. Распознаваемые суффиксы (расширения имен) файлов перечислены в таблице Г.1.

Несмотря на применение этой опции, указание **-C** по прежнему будет назначать удаление препроцессором всех комментариев из исходных файлов.

-fpretend-float

Во время перекрестной компиляции, т.е. выработки компилятором объектного кода другой системы, этот флаг указывает применение формата "домашней" машины (*host platform*) для генерирования инструкций математических операций с плавающей точкой. При этом код, получаемый в результате такой компиляции, скорее

всего не сможет выполняться пред назначаемой машиной (target machine). Но порядок следования инструкций скорее всего будет тем же, что и при выработке кода, точно соответствующего пред назначаемой платформе.

-fprofile-arcs

При использования опции **-fprofile-arcs** для компиляции программы и после запуска версии программы, скомпилированной с этой опцией, создается файл, который содержит результаты подсчета количества проходов выполнения для каждого блока кода. Затем программа может быть заново скомпилирована с опцией **-fbranch-probabilities**, и при этой новой компиляции информация из файла, уже записанного профилирующим кодом, может быть использована для оптимизации наиболее часто используемых ветвей программы. В случае отсутствия информации из такого файла GCC для оптимизации может примерно оценить вероятный путь выполнения программы. Информация о проходах блоков записывается в файл, который имеет то же имя, что и файл исходного кода, только с добавлением суффикса **.da**.

В другом варианте применения данная опция действует совместно с опцией **-ftest-coverage** для поддержки использования утилиты **gcov**. Сочетание этих опций для каждой функции создает граф потока выполнения программы (flow graph) и на основе информации графа строит дерево расстояний переходов между функциями (spanning tree). Затем в каждую функцию, которая не входит в дерево расстояний, помещается код для подсчета количества проходов выполнения функции. В блоки с простыми входом и выходом профилирующий код добавляется непосредственно в блок. Блоки с множеством входов и выходов разбиваются на простые блоки, структура которых обеспечивает трассировку всех входов и выходов первичного блока.

Программа, откомпилированная с этими опциями, затем запускается с утилитой **gcov**. При этом она выполняется несколько медленнее, чем та же программа, откомпилированная с опцией **-a** или **-ax**. Однако этот метод предпочтительнее. Подсчеты, получаемые с использованием опции **-a**, не дают достаточной информации для расчета вероятностей всех ветвлений.

См. также **-a**, **-fbranch-probabilities** и **-fguess-branch-probability**.

-freduce-all-givs

Усекает размер всех общих устанавливаемых переменных (таких как счетчики цикла). Эта опция может улучшать или ухудшать производительность вырабатываемого компилятором кода в зависимости от сложности структуры циклов в исходном коде. По умолчанию для всех языков программирования, кроме *Fortran*, действует обратная опция **-fno-reduce-all-givs**.

См. также **-fmove-all-movables**.

-freg-struct-return

Генерирует код, который возвращает структуры данных (structures) малой длины через регистры. Детальное определение соглашений о передаче структур через память или через регистры зависит от платформы.

При компиляции программ на языке *Fortran* эта опция применяется только в случае компоновки с библиотекой `libg2c`, если эта библиотека тоже была скомпилирована с опцией `-freg-struct-return`.

См. также опцию `-fpcc-struct-return`.

-fregmove

То же, что и опция `-foptimize-register-move`.

-frename-registers

Применяет такой способ оптимизации, который после планирования инструкций выполняет попытку исключения ложных зависимостей в ассемблерном коде. Это позволяет задействовать регистры, оставшиеся не использованными после распределения регистров и планирования инструкций. Опция дает заметные преимущества на машинах с большим количеством регистров. Код, сгенерированный с этой опцией, отлаживать довольно сложно. Флаг устанавливается автоматически при использовании опции оптимизации `-O3`, при необходимости его можно отключить обратной опцией `-fno-rename-registers`.

-frepo

C++

Применяет автоматическое включение в код всех экземпляров шаблона. Эта опция также устанавливает опцию `-fno-implicit-templates`, которая подавляет автоматическое включение в код шаблонов, не предназначенных для подстановки (non-inline templates).

-frerun-cse-after-loop

Эта опция отменяет повторный проход оптимизации исключения общих подвыражений (CSE) при последующей оптимизации циклов. Делается это потому, что оптимизация циклов может создавать новые общие подвыражения. Этот флаг автоматически устанавливается опциями `-O2`, `-O3` и `-Os`, но может быть замещен применением обратной опции `-fno-rerun-cse-after-loop`. См. также `--param`.

-frerun-loop-opt

Дважды запускает оптимизацию циклов. При втором проходе этой оптимизации линейное разложение циклического кода не происходит, но применяется повторный анализ циклов с учетом инструкций, исключенных при первом проходе. Эта опция автоматически устанавливается опциями `-O2`, `-O3` и `-Os`, но может быть отключена обратной опцией `-fno-rerun-loop-opt`.

-frtti

C++

Действует по умолчанию. Для каждого класса, содержащего виртуальные методы, генерируется код идентификации класса по времени (RunTime Type Identification, RTTI). Если вы не используете операторы `dynamic_cast` или `typeid`, то применение обратной опции `-fno-rtti`, отменяющей генерирование такого кода, позволяет несколько уменьшить размер памяти, занимаемой каждым классом. Опция

-fno-rtti не действует при использовании методов обработки исключений, которые требуют присутствия кода RTTI.

-fschedule-insns

Предпринимается попытка перестроения инструкций для предотвращения остановок (stalling) при выполнении кода. Это может требоваться на машинах, допускающих одновременное выполнение нескольких инструкций, и у которых операции с плавающей точкой или обращения к памяти выполняются медленно по сравнению с остальными операциями. Генерируемый код позволяет загружать и выполнять другие инструкции параллельно с выполнением медленных инструкций. Этот флаг автоматически устанавливается опциями **-O2**, **-O3** и **-Os**, но при необходимости может быть отключен применением обратной опции **-fno-schedule-insns**.

-fschedule-insns2

Действует также, как и опция **-fschedule-insns**, за исключением того, что она применяется уже после выделения для каждой функции как глобальных, так и локальных регистров. Эта опция может быть эффективной для машин с малым количеством регистров и относительно медленными инструкциями загрузки в регистры данных. Этот флаг автоматически устанавливается опциями **-O2**, **-O3** и **-Os**, при необходимости он может быть отключен применением обратной опции **-fno-schedule-insns2**.

-fshared-data

Опция применяет разделяемый доступ к объявленным с атрибутом **private** данным. Это имеет смысл для операционных систем, обеспечивающих разделяемый доступ к данным между параллельными процессами одной программы, когда для каждого процесса создается собственная копия элементов обрабатываемых **private**-данных.

-fshort-double

Применяет для данных типа **double** тот же размер, что и для типа **float**. Использование этой опции может вызывать проблемы при компиляции программ на языке *Fortran*.

-fshortEnums

Уменьшает размер перечисляемого типа **enum** до размера наименьшего типа целых чисел, который можно применить для хранения диапазона всех объявленных значений.

-fshort-wchar

C, C++

Преобразовывает тип данных **wchar_t** к типу **unsigned short int**, вместо применения для хранения таких данных того типа, который применяется по умолчанию для текущей платформы.

-fsigned-char**C**

При этой опции тип данных `char` по умолчанию считается типом со знаком `signed char` (с диапазоном значений от `-127` до `+128`). При отсутствии явного указания этого флага применение по умолчанию типа `signed` или `unsigned` зависит от платформы. Обратная опция `-fno-signed-char` равносильна опции `-funsigned-char`.

-fsigned-bitfields**C**

Действует по умолчанию. Битовые поля (`bitfields`) считаются относящимися к данным целочисленных типов `int` со знаком. При обратной опции `-fno-signed-bitfields` они воспринимаются как данные типов `unsigned int`. При указании опции `-traditional` все битовые поля не будут содержать знака. Обратная опция `-fno-signed-bitfields` равносильна опции `-funsigned-bitfields`.

-fsilent**Fortran**

Действует по умолчанию. При обратной опции `-fno-silent` на стандартное устройство вывода сообщений об ошибках `stderr` последовательно выдаются имена всех компилируемых модулей.

-fsingle-precision-constant

Все числовые константы с плавающей точкой сохраняются как числа с плавающей точкой обычной точности (тип `float`), вместо применения для этого двойной точности (тип `double`).

-fsource-case-spec**Fortran**

Значение `spec` указывает следует ли регистр буквенных знаков исходного текста программы преобразовывать к строчным или заглавным буквам, или он должен остаться в том же виде. Изменения не затрагивают константы формата Holerith. Возможны следующие значения `spec`:

- `upper` — Исходник транслируется с заменой строчных букв на заглавные.
- `lower` — Исходник транслируется с заменой заглавных букв на строчные. Это значение применяется по умолчанию.
- `preserve` — Регистр букв исходного текста программы не изменяется.

См. также `-fintrinsic-case-`, `-fmath-case-`, `-fsymbol-case-` и `-fcase-`.

-fssa

Экспериментальная опция. Все тело функции конвертируется в граф потока выполнения (flow graph) формата SSA (Static Single Assignment). Затем выполняется ряд оптимизаций. Затем код преобразовывается к прежнему формату.

-fssa-ccp

Экспериментальная опция. Применяет оптимизацию условной передачи кода CCP (Conditional Code Propagation) с преобразованием кода в граф формата SSA

(Static Single Assignment). Эта оптимизация заменяет на константы те переменные, значения которых нигде не изменяются, и устраниет не используемые ветви программы. Опция требует установки опции **-fssa** и назначения опцией **-O** любого уровня оптимизации.

-fssa-dce

Экспериментальная опция. Удаляет участки неиспользуемого кода, то есть применяет оптимизацию DCE (Dead Code Elimination). Оптимизация выполняется с преобразованием кода в граф формата SSA (Static Single Assignment). Эта опция требует установки опции **-fssa** и назначения опцией **-O** любого уровня оптимизации.

-fstack-check

Назначает генерирование необходимого проверочного кода для предотвращения ситуаций переполнения стека выполняемой программой. Этот код в действительности не проверяет стек, он использует встроенные возможности операционной системы для обработки ситуации возможного переполнения стека.

Необходимость в этой опции может возникать при компиляции программ для их выполнения в многопоточной среде (multi-threaded environment). Программы, которые выполняются в однопоточной среде определяют переполнение стека автоматически.

-fstack-limit-register=register

Указывает имя регистра, который содержит предельный адрес, ограничивающий допустимый размер стека. Опция может применяться для сокращения используемой программой области стека. Она не может применяться для увеличения области стека далее размера, предоставляемого операционной системой.

См. также **-fstack-limit-symbol**.

-fstack-limit-symbol=symbol

Указывает имя переменной, которая содержит адрес памяти, ограничивающий размер области стека. Опция может применяться для сокращения используемой программой области стека. Но не может применяться для увеличения области стека далее размера, предоставляемого операционной системой.

Величины, используемые для адресации памяти зависят от платформы. Например, если начальный адрес стека равен 0x80000000 и стек растет в сторону уменьшения адресов, то ограничение области стека размером 128k применяется установкой следующих опций:

```
-fstack-limit-symbol=__stack_limit -Wl,__stack_limit=0x7FFE0000
```

Также возможно объявление переменной адреса предела стека внутри программы. Но и тогда в командной строке на компиляцию программы необходимо параметром опции **-fstack-limit-symbol** указать имя этой переменной.

См. также опцию **-fstack-limit-register=register**.

-fstats**C++**

Выводит статистику обработки программы верхним уровнем компилятора. Эта информация относится к внутренним действиям компилятора. Опция не влияет на вырабатываемый компилятором код.

-fstore-check**Java**

Действует по умолчанию. Указание обратной опции **-fno-store-check** удаляет код, проверяющий во время выполнения программы соответствие типов объектов при их сохранении в массив.

-fstrength-reduce

Применяет такую оптимизацию циклов, которая сокращает их код и исключает избыточные переменные внутри цикла. Также происходит замена медленно выполняемых операций, таких как умножение и деление, на более простые и быстрые операции, такие как сложение и вычитание. Эта опция всегда автоматически применяется при использовании опций **-funroll-loops** и **-funroll-all-loops**. Ее автоматически устанавливают также опции оптимизации **-O2**, **-O3** и **-Os**. При необходимости данную опцию можно отключить обратной опцией **-fno-strength-reduce**.

Вот простой пример. В цикле используется временная переменная для хранения вычисляемого значения индекса массива и ее значение вычисляется из переменной цикла умножением на 2:

```
for (int i=0; i<10; i++) {
    index = i * 2;
    frammis (valarr[index]);
}
```

Здесь внутренняя переменная цикла **index** можно исключить. Кроме того, для умножения на 2 можно применить простую операцию логического сдвига. В результате получается такой код:

```
for (int i=0; i<10; i++) {
    frammis (valarr[i << 1]);
}
```

Сдвиг на одну двоичную позицию счетчика цикла умножает его значение на 2 и результат выражения непосредственно используется в качестве индекса массива без его хранения во временной переменной.

-fstrict-aliasing

При этой опции применяются наиболее строгие правила использования синонимов при адресации данных и функций (aliasing). Синонимами (alias) считаются различные имена программных символов, прямо или косвенно адресующиеся к одному расположению памяти. Применение строгих правил синонимов, к примеру, для языка C означает, что символ типа **int** не может быть синонимом символа типа **double** или указателя, но может быть синонимом символа типа **unsigned char**.

Даже при строгих правилах совмещения имен могут оставаться проблемы при обращениях к члену объединения (union member) через адрес объединения вместо использования для этого указателя на требуемый член объединения. Вот пример кода, который может вызывать проблемы:

```
int *iprt
union {
    int ivalue;
    double dvalue;
} migs;
...
migs.ivalue = 45;
iptr = &migs.ivalue;
frammis (*iptr);
migs.dvalue = 88.6;
frammis (*iptr);
```

В этом примере строгое совмещение имен может не определить возможное изменение значения, адресуемого указателем `iptr`, в промежутке между двумя вызовами функции. При прямой адресации членов объединения таких проблем не возникает.

-fsymbol-case-spec

Fortran

Поле `spec` указывает регистр буквенных знаков (строчные/заглавные), используемый в определяемых пользователем символах программы. Возможны следующие значения `spec`:

- `initcap` — Слово начинается с заглавной буквы, остальные буквы — строчные.
- `upper` — Все слово должно быть написано заглавными буквами.
- `lower` — Все слово должно быть написано строчными буквами.
- `any` — Допускается любой порядок использования строчных и заглавных букв. Это значение применяется по умолчанию.

См. также `-fmath-case-`, `-fsource-case-`, `-fintrin-case-` и `-fcase-`.

-fsyntax

Fortran

Компилятор только проверяет синтаксис исходного кода. Больше никаких действий при этой опции не предпринимается.

-ftemplate-depth-number

C++

Число `number` устанавливает наибольшую допустимую глубину вложенности экземпляров шаблона для определения ситуаций рекурсивной или циклической подстановки шаблонов кода. При достаточном соответствии программы стандарту нет необходимости указывать допустимую глубину более 17. По умолчанию она равна 500.

-ftest-coverage

При этой опции компилятор вырабатывает файлы, содержащие информацию для утилиты **gcov**. Эти файлы несут то же имя, что и исходный файл, но имеют суффиксы, указывающие на их содержание.

Файлы с суффиксом **.bb** содержат информацию о соответствии основных блоков объектного кода номерам строк исходного файла. Эта информация используется **gcov** для соотношения результатов подсчета проходов выполнения с номерами строк исходного кода.

Файлы с суффиксом **.bbg** содержат список связей графа потока выполнения программы (flow graph). Эта информация используется **gcov** для перестройки графа потока выполнения и расчета количества проходов выполнения блоков программы по данным из файлов с суффиксом **.da**, вырабатываемых по опции **-fprofile-arcs**.

См. также опции **-a**, **-ftest-coverage** и **-fprofile-arcs**.

-fthread-jumps

Возникают ситуации когда после вычислении условия перехода и его выполнения управление может передаваться в такое расположение программы, где из действующих на момент первичного перехода значений вычисляется новое условие, которое вызывает новый переход с вполне определенным при таких обстоятельствах назначением. В таких случаях возможна оптимизация последовательных переходов, которая перенаправляет цель первичного перехода в место окончательного предназначения. Эта опция устанавливается автоматически при всех уровнях оптимизации. В отличие от других подобных опций она не может быть замещена опцией **-fno-thread-jumps**.

-ftime-report

По этой опции после завершения компиляции программы печатается отчет о времени, затраченном на компиляцию. Выводится время использования отведенных пользователю ресурсов, время использования системы и отсчеты времени (wall clock) для каждого прохода. Выводятся суммарные итоги времени использования.

-ftrapping-math

Действует по умолчанию. При установке обратной опции **-fno-trapping-math** считается, что ошибки операций с плавающей точкой не могут вызывать исключения, обрабатываемые прерываниями, и порождать сигналы. Обратная опция **-fno-trapping-math** может привести к генерированию такого кода, который нарушает условия стандартных правил для операций с плавающей точкой.

-ftrapv

Генерирует код, который перехватывает и обрабатывает ситуации переполнения результата операций сложения чисел с учетом знака, а также операций деления и умножения. Эта опция может использоваться во время тестирования программы и создавать файлы **core** для каждого случая переполнения результата, которые обычно проходят незамеченными и могут вызывать проблемы.

По умолчанию действует обратная опция **-fno-trapv**.

-ftypeless-boz

Fortran

Указывает использование бестипового префикса, определяющего предельное значение (или основание счета, radix) недесятичных констант вместо применения по умолчанию типа **INTEGER(KIND=1)**. Имеются в виду константы наподобие **Z'ABCD'**.

-fugly-args

Fortran

Действует по умолчанию. Обратная опция **-fno-ugly-args** запрещает передачу в аргументах вызова функций бестиповых констант формата "Hollerith". По умолчанию будут допустимыми оба следующих примера вызова:

```
CALL FRED(4HABCN)
CALL SAM('123'0)
```

-fugly-assign

Fortran

При этой опции программа использует общие ячейки для хранения присваиваемых меток (assigned labels) и числовых данных (numeric data). Например два оператора следующего примера используют одно и то же расположение ячейки:

```
I = 3
ASSIGN 10 TO I
```

Эту опцию необходимо применять, если программа обращается к присваиваемому значению как элементу данных, потому что по умолчанию для хранения различных типов информации всегда отводятся отдельные ячейки.

-fugly-assumed

Fortran

Массив с указанным размером в один элемент считается объявленным с размером "*". Например, оператор **DIMENTION X(1)** при этой опции воспринимается как **DIMENTION X(*)**.

-fugly-comma

Fortran

Завершающая список аргументов запятая воспринимается как передача подпрограмме дополнительного нулевого аргумента. То есть, при этой опции оператор **CALL BLOG()** будет передавать вызываемой подпрограмме один нулевой аргумент, а **CALL RIM(,)** — два нулевых аргумента.

Без этой опции завершающая запятая в списке аргументов будет игнорироваться и подпрограммам не будут передаваться нуль-аргументы даже и в случае присутствия в списке других аргументов.

-fugly-complex

Fortran

Допускает использование с встроенными функциями (intrinsics) **REAL(expr)** и **AIMAG(expr)** любых типов комплексных выражений. По умолчанию допускаются только такие выражения с комплексными числами, которые дают результат типа **COMPLEX(KIND=1)**.

При совместном использовании этой опции с опцией `-ff90` такие встроенные функции не конвертируют возвращаемые ими действительную и мнимую части комплексного аргумента.

-fugly-init**Fortran**

Действует по умолчанию. При указании обратной опции `-fno-ugly-init` не допускается использование данных формата Hollerith в операторах `DATA` и `PARAMETER`. При этом также не разрешается использование буквенных констант для инициализации числовых типов данных и наоборот, использование числовых констант для инициализации буквенных типов данных.

-fugly-logint**Fortran**

Применяет автоматическое конвертирование типов `INTEGER` и `LOGICAL` практически во всех случаях, где это необходимо. Этого достаточно, чтобы перекрестно использовать два этих типа почти в любых выражениях.

-funderscoring**Fortran**

Действует по умолчанию. К именам с одним знаком подчеркивания "_" (underscore character) компилятор добавляет два подчеркивания, а к внешне определяемым именам без знака подчеркивания добавляет один такой знак. Два знака подчеркивания добавляются также и к внутренним именам, уже имеющим в конце имени знак подчеркивания, это делается для предупреждения возможных коллизий с внешними именами.

Установка обратной опции `-fno-underscoring` запрещает все преобразования имен прибавлением к ним знаков подчеркивания. Если же применить `-fno-second-underscore`, то эта установка отменит только добавление к именам второго знака подчеркивания.

Отменять преобразование компилятором имен не рекомендуется. Это можно делать только если в исходный код уже внесены особые поправки, обеспечивающие его совместимость с выходным кодом других компиляторов. Среди прочих проблем при отмене добавления подчеркиваний следует особо указать на возможность конфликтов имен с системными библиотеками.

-fuse-boehm-gc**Java**

Опция назначает использование метода "Boehm" разметки битовых полей в выгружаемой в swap информации в ходе выполнения программы. (Boehm garbage collection bitmap marking code.)

-fsecond-underscore**Fortran**

Действует по умолчанию. См. опцию `-funderscoring`.

-funix-intrinsics-spec**Fortran**

Значение поля `specs` определяет статус встроенных функций (intrinsics) UNIX. Возможны следующие значения `specs`:

- **enable** — Встроенные функции UNIX распознаются и их использование возможно. Это значение действует по умолчанию.
- **hide** — Встроенные функции UNIX распознаются, но для их использования при первом вызове такой функции следует применить оператор **INTRINSICS**.
- **disable** — Встроенные функции UNIX распознаются, но их использование допускается только когда перед именем каждой из них стоит оператор **INTRINSICS**.
- **delete** — Встроенные функции UNIX не распознаются.

-funroll-all-loops

Эта опция устанавливает флаг **-funroll-loops** и снимает ограничение на величину кода цикла и количество его итераций. При этом будут разворачиваться даже такие циклы, количество итераций которых во время компиляции не может быть определено. Установка этой опции обычно приводит к выработке компилятором кода большего размера, дольше выполняемого машиной.

Когда не удается определить количество итераций цикла, он преобразовывается следующим образом. Сначала цикл разворачивается определенное количество раз и между дубликатами кода первичного цикла вставляются проверки условий выхода. Полученная последовательность помещается в цикл, при этом создается цикл, размер кода которого увеличивается в несколько раз. Преимущество состоит в том, что итерации цикла будут происходить с меньшей частотой, чем итерации исходного цикла без такой обработки.

-funroll-loops

При этой опции в целях оптимизации программы достаточно простые циклы разворачиваются в линейный код при условии, что число итераций цикла и количество инструкций тела цикла достаточно малы. При разворачивании цикла из кода программы удаляется циклическая конструкция и последовательность инструкций цикла линейно повторяется в программе требуемое количество раз. Показатель простоты цикла (т.е. возможность его разворачивания) определяется как произведение числа итераций цикла на количество инструкций тела цикла (имеются в виду инструкции промежуточного кода на языке RTL — insns). Цикл может быть развернут, только если этот показатель меньше заданной величины. В описываемой версии компилятора этот показатель определяется константой, которая по умолчанию имеет значение 100. Эта опция всегда устанавливает опции **-fstrength-reduce** и **-frerun-cse-after-loop**.

-funsafe-math-optimizations

Убирает код проверок операций с плавающей точкой, при установке этой опции считается, что во всех случаях используются только допустимые значения. При этом возникает возможность нарушения стандартов IEEE и ANSI для точности операций с плавающей точкой. Эта опция позволяет компоновщику вставлять код обеспечения нестандартной оптимизации работы аппаратного блока FPU (FPU — Floating Point Unit, устройство для выполнения математических операций с плавающей точкой).

-funsigned-bitfields

C

При указании этой опции битовые поля (`bitfields`) считаются относящимися к данным целочисленных типов без знака `unsigned int`. По умолчанию битовые поля относятся к типу `signed int`. При применении опции `-traditional` все битовые поля в любом случае будут беззнаковыми. Обратная опция `-fno-unsigned-bitfields` равносильна опции `-fsigned-bitfields`.

-funsigned-char

C

При этой опции тип данных `char` по умолчанию считается типом без знака `unsigned char` (с диапазоном значений от 0 до 255). В отсутствие явного указания этого флага применение по умолчанию знакового или беззнакового типа `char` зависит от платформы. Обратная опция `-fno-unsigned-char` равносильна опции `-fsigned-char`.

-funwind-tables

Действие этой опции сходно с действием опции `-fexceptions`. Единственное отличие состоит в том, что при указании `-funwind-tables` генерируются необходимые для обработки исключений статические данные.

См. также `-fexceptions`, `-fnon-call-exceptions` и `-fasynchronous-unwind-tables`.

-fuse-sxa-atexit

C++

Опция приводит к применению порядка запуска глобальных деструкторов, обратного порядку выполнения соответствующих им конструкторов, вместо действующего по умолчанию порядка, обратного запуску конструкторов. Последовательность запуска будет изменяться только в случае использования вложенных вызовов конструкторов, когда один конструктор вызывает другой. Опция будет действовать только при наличии в разделяемой библиотеке стандартных функций языка C (C runtime library) функции `sxa_exit()`. Без опции `-fuse-sxa-atexit` компилятор вместо `sxa_exit()` использует функцию `atexit()`.

-fuse-divide-subroutine

Java

Для выполнения деления целых чисел будет использоваться вызов подпрограммы библиотеки. Это делает возможной обработку исключения в ситуации целочисленного деления на ноль.

-fverbose-asm

Вставляет более подробные чем обычно комментарии в выходной код на ассемблере. Это делает выходной листинг более читаемым.

Основное применение этой опции планировалось для отладки самого компилятора. Именно для облегчения чтения выходного ассемблерного кода. По умолчанию действует обратная опция `-fno-verbose-asm`. Ее применение дает более удобный код для сравнения выходных листингов.

-fversion**Fortran**

Выполняет запуск внутренних проверок правильности установки в составе GCC компилятора GNU Fortran и показывает информацию о его версии. Эта опция также устанавливает оба флага `-v` и `--verbose`.

-fvolatile

Для всех расположений памяти, адресуемых через указатели, применяет свойства вариантного типа языка C (объявляемого с квалификатором `volatile`).

См. также `-fvolatile-global` и `-fvolatile-static`.

-fvolatile-global

Для всех расположений памяти, адресуемым как локально так и глобально, применяет свойства вариантного типа языка C (объявляемого с квалификатором `volatile`). Это не распространяется на статические (static) элементы, то есть доступные для обращения к ним только в пределах своего модуля компиляции.

См. также `-fvolatile` и `-fvolatile~static`.

-fvolatile-static

Применяет свойства вариантного типа языка C (объявляемого с квалификатором `volatile`) для статических (static) расположений памяти, то есть доступных для обращения к ним только в пределах своего модуля компиляции.

См. также `-fvolatile-global` и `-fvolatile`.

-fvtable-gc**C++**

Вырабатывает информацию о перемещении символов (relocation information), которая позволяет компоновщику устранять из таблицы vtable записи для неиспользуемых виртуальных функций. Эта опция требует обязательного применения и компоновщика и ассемблера GNU.

Вырабатываемая по этой опции информация используется также для удаления определений не используемых функций. См. описания опции `-ffunction-sections` и `-W1`.

-fvxt**Fortran**

Некоторые конструкции исходного кода отличаются между диалектами GNU Fortran и VXT Fortran. Использование этой опции для интерпретации таких конструкций применяет правила диалекта VXT Fortran.

См. также опции `-ff90`, `-fvxt-intrinsics-` и `-ff90-intrinsics-`.

-fvxt-intrinsics-spec**Fortran**

Значение поля `spec` определяет статус встроенных функций (intrinsics) диалекта VXT Fortran. Возможны следующие значения `spec`:

- `enable` — Встроенные функции диалекта VXT распознаются и их использование возможно. Это значение действует по умолчанию.

- **hide** — Встроенные функции VXT распознаются, но для их использования при первом вызове такой функции следует применить оператор `INTRINSICS`.
- **disable** — Встроенные функции VXT распознаются, но их использование допускается только если перед именем каждой из них стоит оператор `INTRINSICS`.
- **delete** — Встроенные функции VXT не распознаются.

См. также опцию `-fvxt`.

-fweak

C++

Действует по умолчанию. Обратная опция `-fno-weak` отменяет поддержку замещения программных символов даже в случае их поддержки компоновщиком. Не следует без особой потребности применять `-fno-weak`, потому что при этом вырабатывается код низкого качества, пригодный разве только для тестирования компилятора.

-fwritable-strings

C, C++

При использовании этой опции компилятор допускает запись данных в строковые константы. Флаг устанавливается автоматически при установке опции `-traditional`.

Для получения действительной возможности записи в строковые константы при компиляции программ на языке C++ необходимо также указывать опцию `-fno-const-strings`.

-fzeroes

Fortran

При этой установке компилятор воспринимает нулевые значения так же, как и любые другие. Без этой опции существует возможность того, что компилятор не определит установки многими операторами `DATA` нулевых начальных значений переменных.

-g[leveL]

В выход компилятора включается отладочная информация в формате, распознаваемом отладчиком `gdb`. Точный формат этой информации зависит от формата вырабатываемого компилятором объектного кода (`stabs`, `COFF`, `XCOFF`, `DWARF` или `DWARF2`).

Параметр уровня отладочной информации `leveL` является необязательным. Числовое значение этого параметра от 1 до 3 указывает количество включаемой в выход отладочной информации. По умолчанию он имеет значение 2. Уровень, равный 1, вырабатывает только глобальную отладочную информацию, необходимую для выполнения отладчиком обратной трассировки кода. При уровне 2 кроме информации первого уровня включается также информация о локальных переменных и номера строк исходного кода. На третьем уровне кроме информации второго уровня в выход включается дополнительная отладочная информация, такая как использованные при компиляции макроопределения.

На системах, использующих формат объектного кода stabs, компилятор по этой опции вырабатывает такую отладочную информацию, которая может быть использована только отладчиком GNU `gdb`.

Допускается совместное применение этой опции с опцией `-o`, которая производит оптимизированный выходной код. Учтите, что оптимизация может серьезно усложнить отладку программы, она изменяет выходной объектный код и нарушает его однозначное соответствие исходнику. Часть объектного кода может быть перенесена. Некоторая часть исходного кода, возможно, вообще не будет транслирована в выполнимый формат.

Опция имеет другую форму представления `--debug`.

См. также `-ggdb`, `-gstabs`, `-gcoff`, `-gxcoff`, `-gdwarf` и `-gdwarf2`.

`-gcoff[/level]`

Вырабатывает отладочную информацию в формате COFF, если он поддерживается пред назначаемой системой. Этот формат наиболее часто используется отладчиком SDB на системах System V старших выпусков, чем SVR4. Параметр `level` — не обязательный. Значения `level` 1, 2 и 3 смотри в описании опции `-g`.

`-gdwarf[/level]`

Вырабатывает отладочную информацию в формате DWARF 1-й версии, если пред назначаемая система поддерживает такой формат. Параметр `level` — не обязательный. Расширенная информация для отладчика `gdb` включается только при указании символа "+" в качестве значения `level`. Это может сделать невозможным применение других отладчиков. Описания значений `level` 1, 2 и 3 смотри в опции `-g`.

Формат DWARF 1-й версии используется отладчиком SDB на многих системах SVR4.

`-gdwarf-2[/level]`

Вырабатывает отладочную информацию формата DWARF 2-й версии, если пред назначаемая система поддерживает такой формат. Параметр `level` не обязательный. Значения `level` 1, 2 и 3 описаны в опции `-g`.

Этот формат используется отладчиком DBX на системах IRIX 6.

`-gen-decls`

ObjC

Генерирует интерфейсное объявление класса и записывает его в файл с именем `x.decl`.

`-ggdb[/level]`

Вырабатывает подробную отладочную информацию, отформатированную специально для использования отладчиком `gdb`. В выход включаются любые доступные расширения, поддерживаемые `gdb`. Параметр `level` — не обязательный. Значения `level` 1, 2 и 3 описаны в опции `-g`.

-gnatoption**Ada**

При компиляции программ на языке Ada передает специфическую опцию (или набор специфических опций) для GNAT, драйвера языка Ada верхнего уровня GCC. Все эти опции определяются в поле **option** как отдельные буквы, добавляемые к общему префиксу таких опций **-gnat**. Например, для назначения опций GNAT "e" и "1" ("эль") их следует так поставить в командной строке компилятору:

```
$ gcc -gnate -gnat1
```

Две этих опции GNAT возможно объединить в одной опции команды **gcc**. Следующая командная строка равнозначна предыдущей:

```
$ gcc -gnatel1
```

Некоторые из опций драйвера GNAT требуют указания сопутствующих им значений. Например, буква опции "m" назначает наибольшее допустимое количество сообщений об ошибках. Следующая команда установливает это число равным 15-ти:

```
$ gcc -gnatm15
```

Опция, требующая указания ей значения, может быть объединена с другими буквенными опциями, но при этом она должна назначаться в последнюю очередь. Следующий пример команды показывает объединение в одной опции **gcc** спецификаций GNAT "e", "l" и "k":

```
$ gcc -gnatelk15
```

Опции GNAT двух следующих команд, указываемые как числа без предшествующей им литеры, применяют ограничения стандартов Ada 83 и Ada 95 соответственно:

```
$ gcc -gnat83
$ gcc -gnat95
```

По умолчанию действует **-gnat95**. В таблице Г.3 содержится список доступных буквенных кодов опций GNAT, применяемых в командной строке **gcc** с префиксом **-gnat**.

Таблица Г.3. Буквенные модификаторы и значения, используемые с опцией **-gnat**

Буква	Описание
a	Применяет при отладке использование логических выражений с определяемым утверждением (assertion), которые могут порождать сообщения об ошибках. Опция включает действие директив pragma Assert и pragma Debug . Без указания такой опции эти прагмы в исходных файлах игнорируются.
b	Все сообщения об ошибках будут выводиться в сокращенной форме даже при назначении опции verbose .
c	Выполняет только проверки семантики и синтаксиса программы. При этом генерируются файлы .ali , но не вырабатывается выполнимый код.
e	Сообщения об ошибках не накапливаются до завершения обработки исходника, а выдаются по ходу компиляции. Это позволяет гарантированно получать их в случае аварийного завершения компиляции.

Буква	Описание
E	Выполняет полные динамические проверки порядка выработки программы (dynamic elaboration checks).
f	Включает вывод всех возможных сообщений об ошибках. Позволяет определять несколько ошибок в одной строке программы. При необъявленной переменной сообщение выдается при каждом обращении к ней.
g	Применяет проверки соответствия стилей (вертикальные колонки, отступы, шаблоны разметки регистром букв и т.п.).
ichar	Символ <i>char</i> содержит идентификатор кодировки буквенных знаков. Положения знаков набора ASCII (от 1 до 127) не изменяются. Остальные значения 8-битной кодировки (от 128 до 255) могут изменяться. Распознаются следующие значения <i>char</i> : 1: Набор знаков Latin-1. 2: Latin-2. 3: Latin-3. 4: Latin-4. p: Набор IBM PC кодовой страницы 437. 8: Набор IBM PC кодовой страницы 850. f: Пользовательский набор знаков верхнего регистра клавиатуры (full uppercase). n: Набор пользовательских знаков без верхнего регистра (no uppercase). w: Расширенный набор знаков (wide character set).
jchar	Значение <i>char</i> указывает на применяемый метод кодировки расширенных буквенных знаков (wide characters). Допустимы следующие методы их представления: n: (none) Формат расширенных знаков не указывается. h: (Hex encoding) Шестнадцатиричная мультибайтная кодировка с применением escape-символа. u: (Upper-half coding) Перекодировка верхней половины алфавита. Первый бит начального байта кодовой последовательности установлен в единицу. Исключает применение знаков верхнего регистра набора Latin-1. s: (Shift JIS) Сдвигнутая кодировка JIS. Отличается от "u" тем, что представление расширенного символа состоит из двух последовательных знаков. Исключает использование верхнего регистра набора Latin-1. e: (EUC coding). Отличается от "u" тем, что каждый расширенный буквенный символ записывается двумя последовательными знаками ASCII, причем каждый из них имеет установленный верхний бит. Исключает использование верхнего регистра набора Latin-1.
knumber	Значение <i>number</i> ограничивает длину идентификаторов.
1 ("эль")	Выводит весь исходный код со всеми сообщениями об ошибках.
mnumber	Значение <i>number</i> ограничивает число выводимых сообщений об ошибках.
n	Активирует подстановку кода в пределах одного модуля при использовании прагмы <i>inline</i> . Замещается опцией <i>-fno-inline</i> .
N	Действует как <i>-gnatn</i> , но прагма <i>inline</i> автоматически применяется ко всем исходным файлам. Замещается опцией <i>-fno-inline</i> .
o	Задействует во время выполнения программы обычно не выполняемые проверки таких исключений как переполнение целочисленных операций, деление на ноль и обращение к значениям до завершения их вычислений. При этом вырабатывается более тяжелый и медленный код. (Действие этой опции не распространяется на операции с плавающей точкой.)
p	Подавляет генерирование кода любых проверок, действующих во время выполнения программы. Действует так же прагма <i>Supress (all_checks)</i> . Экономит размер программы и повышает ее производительность за счет снижения защиты от неправильной обработки данных.

Буква	Описание
q	Продолжает компиляцию независимо от присутствия синтаксических ошибок. Выходной код генерируется независимо от результатов работы парсера. Может приводить к выработке кода с непредсказуемым поведением.
g	Требует соответствия форматирования исходного кода соглашениям, перечисленным в официальном руководстве по программированию на языке Ada.
s	Выполняет только синтаксическую проверку исходника.
t	Выводит дерево внутреннего представления кода в файл .adt. Эта информация используется для исключения неиспользуемых участков кода (dead code elimination).
u	Выводит список всех модулей, задействованных в текущей компиляции.
v	Назначает режим вывода описаний (verbose mode). На стандартное устройство вывода направляются подробные сообщения об ошибках, содержащие строки ошибочного исходного кода.
wmode	Значение <i>mode</i> назначает способ обработки предупредительных сообщений. Если поле <i>mode</i> имеет значение "v", то вывод предупреждений подаётся. При значении "e" все предупреждения воспринимаются как сообщения об ошибках. При значении "1" ("эль") выводятся только те предупреждения, которые имеют отношение к соблюдению соответствия порядка выработки программы (elaboration order).
ztype	Определяет способ выработки отладочной информации в формате STABS. Если поле <i>type</i> имеет значение "x", то директивы симаольной таблицы вырабатываются для тех процедур, которые принимают вызовы, (receiver). При значении "v" они вырабатываются для посылающих вызовы процедур (sender).

-gstabs[/*level*]

Вырабатывает отладочную информацию в формате STABS, если пред назначенная система поддерживает такой формат. Параметр *level* — не обязательный. Расширенная информация для отладчика *gdb* включается только при указании символа "+" в качестве значения *level*. Значения *level* 1, 2 и 3 описаны в опции **-g**.

Этот формат может использоваться отладчиком DBX на многих системах BSD, но он не работает с отладчиками DBX или SDB на платформах MIPS, Alpha и в системе SVR4. При выработки информации формата STABS на системе SVR4 требуется также применение ассемблера GNU.

-gvms[/*level*]

Вырабатывает отладочную информацию в формате VMS, если пред назначенная система поддерживает такой формат. Параметр *level* — не обязательный. Значения *level* 1, 2 и 3 описаны в опции **-g**.

Этот формат используется отладчиком DEBUG на системах VMS.

-gxcoff[/*level*]

Вырабатывает отладочную информацию в формате XCOFF, если пред назначенная система поддерживает такой формат. Параметр *level* — не обязательный. Значения *level* 1, 2 и 3 описаны в опции **-g**.

Этот формат используется отладчиком DBX на системах RS/6000.

-H**Pre**

Выводит упорядоченный список всех использованных заголовочных файлов вместе со отдельным списком таких файлов, не имеющих кода предотвращения ситуации их множественного включения.

Имеется другая форма представления этой опции: **--trace-includes**.

--help

Показывает список распознаваемых **gcc** опций. Если также указана опция **-v**, то выводимый список будет включать и опции задействуемых **gcc** процессов. Если при этом дополнительно указать опцию **-W**, то в список будут включены и все недокументированные опции.

См. также описание опции **--target-help**.

--include-directory**Pre, Ada, Java**

То же, что опция **-I**.

-I name**Pre, Ada, Java**

Поле **name** этой опции указывает препроцессору имя верхнего каталога для поиска включаемых в программу заголовочных файлов (include files). В этом каталоге и ниже его должен проводиться поиск файлов, включаемых препроцессором в исходный код программы по директиве **#include**. Опция может быть использована в командной строке несколько раз для указания нескольких таких каталогов.

Установленные этой опцией каталоги просматриваются в первую очередь, что дает возможность замещения любых включаемых в программы системных заголовочных файлов.

Использование этой опции с точкой, т.е. в форме **-I.**, назначает компилятору текущий каталог в качестве рабочего каталога.

При компиляции программ на языке *Ada* поле **name** этой опции указывает путь для поиска исходных файлов.

При компиляции программ на языке *Java* указывает компилятору путь для поиска файлов классов. Возможно указание не только имени каталога, но и имени **.jar** или **.zip** архива. Установка опции **-I** имеет более высокий приоритет, чем опция **--classpath** и переменная окружения **CLASSPATH**. Указанное опцией **-I** расположение будет просматриваться в первую очередь. Разработчики компилятора рекомендуют отдавать предпочтение этой опции и использовать ее вместо **--classpath**.

См. также опции **-I-**, **-isystem**, **-B**, **-nostdinc**, **-withprefixbefore** и переменную окружения **CPATH**. По установке вторичного списка для поиска заголовочных файлов см. опцию **-idirafter**.

Эта опция другую форму представления **--include-directory**.

-I-**Pre**

Опция модифицирует порядок применения директив **-I**, стоящих перед ней в командной строке. Установки этих опций будут действовать на директивы, имеющие формат **#include "..."**, и не будут оказывать действия на директивы в фор-

мате `#include <...>`. При этом все опции `-I`, которые стоят в командной строке после `-I-`, будут работать с директивами `#include <...>`.

Кроме того, использование опции `-I-` отменяет выполняемый по умолчанию поиск включаемых заголовочных файлов в каталоге расположения исходного файла.

Другая форма представления этой опции: `--include-barrier`.

-idirafter directory

Pre

Добавляет указанное имя `directory` во второй список каталогов для поиска заголовочных файлов. При поиске включаемого файла компилятор GCC вначале просматривает каталоги из первого списка. И только затем, если файл не найден, поиск продолжается в каталогах из второго списка. В первый список каталоги добавляются опцией `-I`.

Другая форма представления этой опции: `--include-directory-after`.

-imacros filename

Pre

Указанный этой опцией файл с именем `filename` препроцессор считывает и обрабатывает прежде исходного кода программы. В этом файле пропускается вся информация, кроме директив макроопределений. Назначенные при этом макросы могут затем быть использованы в обрабатываемом исходном файле.

Любые опции `-D` или `-U` обрабатываются раньше любых опций `-imacros`. Опции `-include` и `-imacros` обрабатываются в том порядке, в котором они стоят в командной строке.

Другая форма этой опции: `--imacros`.

-include filename

Pre

Указанный этой опцией файл с именем `filename` препроцессор считывает и обрабатывает прежде исходного кода программы так, как если бы он включался по директиве `#include` в первой строке программы.

Любые опции `-D` или `-U` обрабатываются раньше любых опций `-include`. Опции `-include` и `-imacros` обрабатываются в том порядке, в котором они стоят в командной строке.

Другая форма этой опции: `--include`.

-include-barrier

Pre

Опция имеет то же значение, что и `-I-`.

-include-directory-after directory

Pre

То же, что и опция `-idirafter`.

-include-prefix prefix

Pre

То же, что и `-iprefix`.

-include-with-prefix directory

Pre

То же, что и `-iwithprefix`.

-include-with-prefix-after *directory* Pre

То же, что и `-iwithprefix`.

-include-with-prefix-before *directory* Pre

То же, что и `-withprefixbefore`.

-iprefix *prefix* Pre

Указывает значение `prefix` в качестве префикса, добавляемого для составления полных имен путей доступа перед именами каталогов, указанных опциями `-iwithprefix` и `-iwithprefixbefore`.

Другая форма этой опции: `--include-prefix`.

-isystem *directory* Pre

Добавляет указанный каталог `directory` в начало дополнительного списка каталогов для поиска включаемых заголовочных файлов. При этом каталог помечается как системный, при компиляции к нему применяется такое же отношение, как и к стандартным системным каталогам.

См. также опции `-I` и `-B`.

-iwithprefix *directory* Pre

Опция добавляет каталог в дополнительный список путей включаемых заголовочных файлов. При этом полное имя добавляемого каталога составляется из префикса, указанного опцией `-iprefix`, и значения поля `directory` этой опции. Если префикс не определен опцией `-iprefix`, стоящей в командной строке прежде рассматриваемой опции, то по умолчанию применяется такое значение префиксного каталога, которое действовало при инсталляции самого компилятора.

При поиске включаемого заголовочного файла компилятор GCC вначале просматривает каталоги из первого списка (каталоги в него добавляются опцией `-I`), затем, если файл не найден, поиск продолжается в каталогах из второго списка.

Эта опция может быть записана в форме `--include-with-prefix` или `--include-with-prefix-after`.

-iwithprefixbefore *directory* Pre

Добавляет каталог в основной список путей для поиска включаемых заголовочных файлов. При этом полное имя добавляемого каталога составляется из префикса, указанного в опции `-iprefix`, и значения поля `directory` этой опции. Если префикс не определен в командной строке до рассматриваемой опции, то по умолчанию применяется такое значение префиксного каталога, которое действовало при инсталляции самого компилятора.

Эта опция может быть записана как `--include-with-prefix-before`.

--library-directory *directory* Linker

Опция имеет то же значение, что и `-L`.

-L`directory`**Linker**

Добавляет указанный каталог `directory` в список каталогов для поиска библиотек, указанных опциями `-l` ("эль").

См. также опцию `-B` и в приложении 2 описание переменной окружения `LIBRARY_PATH`.

-l`library`**Linker**

Опция задает имя статической библиотеки, используемой компоновщиком для разрешения внешних ссылок. Полное имя библиотеки составляется добавлением к указанному имени `library` префикса `lib` и суффикса `.a`. Например, опция `-lconsole` сообщает компоновщику имя библиотеки `libconsole.a`.

Поиск библиотеки с именем, указанным опцией `-l`, будет выполняться среди библиотек стандартного набора (т.е. в стандартных каталогах для размещения библиотек) и в каталогах, указанных опциями `-L`.

При компоновке программ на языке *Objective-C* требуется указывать опцию `-lobjc`. Она сообщает компоновщику о необходимости использования основной библиотеки *Objective-C* `libobjc.a`.

Для разрешения внешних ссылок программы компоновщик просматривает библиотеки в том порядке, в котором они указаны опциями `-l` в командной строке. Порядок просмотра библиотек может иметь важное значение. Например, по следующей команде компоновщик сможет разрешить все ссылки из библиотеки `glower.o` на объекты библиотеки `jpeg.a` и не сможет разрешить такие же ссылки, если они есть в библиотеке `flower.o`.

```
gcc glower.o -ljpeg flower.o -o shawall
```

Порядок следования опций имеет значение и тогда, когда используемые библиотеки имеют перекрестные ссылки между собой. При наличии циркулярных ссылок между двумя библиотеками для их разрешения может потребоваться указание в команде их имен более одного раза. Как в следующем примере команды:

```
gcc spring.o -ldflat -lturbo -ldflat -o spring
```

Одна и та же библиотека может быть указана в командной строке как своим полным именем (`libjpeg.a`), так и опцией `-l` (`-ljpeg`). Но только при использовании опции `-l` компоновщик будет выполнять поиск библиотеки в стандартных каталогах и в каталогах, назначенных опциями `-L`.

Для совместимости с соглашениями POSIX допускается оставлять пробел между флагом опции и именем библиотеки.

См. также опцию `-L`.

--language `language`

Опция имеет то же значение, что и `-X`.

-M**Pre**

По этой опции препроцессор выводит правило зависимостей в формате, пригодном для его включения в компоновочный сценарий (makefile). Это правило состав-

ляется из имени объектного файла, стоящего за ним двоеточия, и, затм, имени исходного файла и имен всех включаемых заголовочных файлов. Каждый включаемый файл выводится в отдельной строке, вместе с полным путем расположения (path name). Если какие-либо файлы были указаны в командной строке опциями **-include** или **-imacros**, то их имена также будут выведены в этом списке.

Опция **-M** автоматически применяет опцию **-E**.

Выводимое по этой опции правило включает только имя объектного файла и список зависимостей, там нет никаких указаний, относящихся к компиляции исходного кода.

При отсутствии опций **-MT** и **-MQ** имя объектного файла для выводимого правила будет совпадать с именем исходного, только с соответствующей заменой суффикса.

Исходным файлом может быть файл любого из типов, перечисленных в таблице Г.1, если к нему допустимо применение предобработки кода. Например, если указать файл с исходным кодом на языке *Java*, то будет выведен путь к системному файлу типа **.jar**.

Другими опциями препроцессора, используемыми при выработке правил для компоновочных скриптов (makefiles), являются **-MD**, **-MMD**, **-MF**, **-MG**, **-MM**, **-MP**, **-MQ** и **-MT**.

Опция может быть записана как **--dependencies**.

--main=classname

Java

Указывает имя класса, содержащего метод с именем **main()**. Метод **main()** этого класса назначается входной точкой выполнения вырабатываемой компилятором программы. Опция используется при компиляции файлов исходного кода классов и входных файлов интерпретатора *Java* в объектный код.

-maligned-data

Fortran

Применяется только при компиляции программ на языке *Fortran* для платформ Intel x86.

Применение 64-битного выравнивания чисел с плавающей точкой двойной точности серьезно ускоряет работу на этих платформах программ, использующих много данных типа **REAL(KIND=2)** (**DOUBLE PRECISION**).

-MD

Pre

Действует в основном так же, как опция **-M**. Отличие состоит в том, что она не задействует опцию **-E**. Кроме того, правило зависимостей выводится в файл, имя которого наследует имя исходного файла и имеет суффикс **.d**. Возможно, назначение имени файла для вывода правила зависимостей опцией **-MF** или **-o**.

Опция может быть записана в форме **--write-dependencies**.

-MMD

Pre

Действует как опция **-MD**, только не выводит имена системных заголовочных файлов.

Опция может быть записана в форме **--write-user-dependencies**.

-MF *filename*

Pre

При использовании вместе с опцией **-M**, **-MM**, **-MD** или **-MMD** указывает имя файла для вывода правила зависимостей.

Другим способом назначения имени для такого файла является назначение переменной окружения **DEPENDENCIES_OUTPUT**.

-MG

Pre

Эта опция может использоваться вместе с опцией **-M** или **-MM** для указания, что пропущенные заголовочные файлы должны считаться генерируемыми файлами, созданными в том же каталоге, где находится исходный файл. Зависимости для них вырабатываются так, как будто они присутствуют и не включают других заголовочных файлов.

Опция может быть записана в форме **--print-missing-file-dependencies**.

-MM

Pre

Действует как опция **-MD**, только не выводит имена системных заголовочных файлов.

Опция может быть записана в форме **--user-dependencies**.

-MP

Pre

Эта опция может использоваться вместе с опцией **-M** или **-MM** для создания "пустой" цели для каждого включаемого файла. Единственной целью этой опции является предотвращение вывода программой **make** сообщений об ошибке в случае удаления из исходников ссылок на заголовочные файлы без обновления компоновочного скрипта (**makefile**).

-MQ *filename*

Pre

Действует в основном так же, как опция **-MT**. Отличие состоит в том, что имя целевого файла размечается в соответствии с требованиями оформления компоновочного скрипта (**makefile**). Например, по команде

```
gcc -M -MQ '$(OBJMRK)mrk.o' brink.c
```

будет выведено следующее правило:

```
$(OBJMRK)mrk.o: brink.c
```

-MT *filename*

Pre

Используется с опцией **-M** или опцией **-MM** для назначения имени целевого файла вырабатываемого правила зависимостей. По умолчанию файл цели наследует имя исходного файла и имеет суффикс **.o**. Опцию **-MT** можно использовать для назначения ему другого имени, добавления к имени файла пути расположения, или использования в названии файла значения какой-либо переменной окружения. Например, по команде

```
gcc -M -MT '$(OBJMRK)mrk.o' brink.c
```

будет выведено следующее правило:

```
$ (OBJMRK)mrk.o: brink.c
```

См. также **-MQ**.

--no-lineCommands

Pre

Опция имеет то же значение, что и **-P**.

--no-standardIncludes

Pre, Ada

То же, что и опция **-nostdinc**.

--no-standardLibraries

Linker

То же, что и **-nostdlib**.

--no-warnings

Linker

Опция указывает компилятору не выводить предупредительных сообщений.

То же, что и **-w**.

-nodefaultlibs

Linker

При этой опции компоновщик не будет использовать подпрограммы из стандартных системных библиотек. Будут использоваться только те библиотеки, которые явным образом указаны в командной строке.

Компилятор может генерировать вызовы системных функций **memcpy()**, **memcmp()** и **memset()** на системе System V или **bcopy()** и **bzero()** на BSD. Обычно эти внешние обращения разрешаются с помощью использования системной библиотеки языка **C libc.a**. Если вы отмените использование стандартных системных библиотек, то вам придется позаботиться о том, чтобы предоставить компоновщику эти подпрограммы.

Стандартная библиотека **libcgcc.a** содержит набор особых подпрограмм, специфичных для предназначаемой платформы. По существу, они являются необходимой частью компилятора. Поэтому следует указывать **-lgcc** даже при отмене использования стандартных системных библиотек.

См. также **-nostartfiles** и **-nostdlib**.

-nostartfiles

Linker

При этой опции компоновщик не будет включать в программу стандартные объектные файлы, содержащие код инициализации среды выполнения программы (startup object files). См. также **-nostdlib** и **-nodefaultlibs**.

-nostdinc

Pre, Ada

Предотвращает поиск компоновщиком заголовочных файлов в стандартных системных каталогах. При этой опции поиск может проводиться только в текущем каталоге и каталогах, указанных опциями **-I**.

При компиляции программ на языке *Ada* опция сообщает компилятору, что использование программой системной библиотеки не предполагается.

Опция может быть записана в форме `--no-standard-includes`.

-nostdinc++

Pre, C++

Предотвращает поиск компоновщиком заголовочных файлов в стандартных для программ на языке *C++* каталогах. Поиск в других стандартных системных каталогах при этом не отменяется. Опция специально предназначена для компиляции библиотек *C++*.

-nostdlib

Linker

Эта опция применяет обе опции `-nostartfiles` и `-nodefaultlibs`. При этом компоновщик будет использовать только те файлы, которые указаны ему в командной строке.

Опция может быть записана в форме `--no-standard-libraries`.

-Olevel

Устанавливает уровень оптимизации генерируемого компилятором кода. При оптимизации всегда приходится находить компромисс между сокращением размера кода и занимаемой памяти, и увеличением скорости выполнения программы. По умолчанию применяется `-O0`, что означает отказ от применения оптимизации. Если в опции значение `level` не указано то оно считается равным 1.

Если уровень оптимизации не установлен, то компилятор вырабатывает код, полностью соответствующий структуре входного исходного кода. Выполнение оптимизации не только отнимает существенно больше времени на обработку, но и требует значительно больше памяти.

Компиляция программы без использования оптимизации имеет два преимущества. Во-первых, она выполняется быстро (оптимизация может занимать намного больше времени). А во-вторых, вырабатываемый при этом код намного проще трассируется в отладчике. Конечно же, вы можете трассировать и оптимизированный код. Однако, при оптимизации переносятся многие участки кода, почти всегда пропускаются некоторые ветви и участки, а некоторые оптимизации при каждом проходе дают неоднозначный результат. Все это серьезно усложняет отладку программы.

Так что отказ от оптимизации создает наилучшие условия для процесса разработки программы.

Имеется другая форма представления этой опции: `--optimize`.

Уровни оптимизации программ, устанавливаемые этой опцией, перечислены в таблице Г.4.

Таблица Г.4. Шесть уровней оптимизации

Уровень	Описание
<code>-O</code>	Компилятор пытается сократить как размер кода, так и время его выполнения. И при этом не выполняет модификаций, которые могут затруднить отладку программы.

Уровень	Описание
	Включает Опции <code>-fno-optimize-size</code> , <code>-fdefer-pop</code> , <code>-fthread-jumps</code> , <code>-fguess-branch-probability</code> , <code>-cprop-registers</code> и <code>-fdelayed-branch</code> . Флаг <code>-fomit-frame-pointer</code> устанавливается только если применяемый отладчик способен работать без использования регистра указателя кадра стека.
<code>-O0</code>	Действует по умолчанию. Отключает любые оптимизации размера кода и устанавливает флаг <code>-fno-merge-constants</code> .
<code>-O1</code>	То же, что <code>-O</code> .
<code>-O2</code>	На этом уровне применяются все виды оптимизации, которые не требуют вычисления оптимального выбора между размером и скоростью кода. Кроме флагов, устанавливаемых при <code>-O</code> , дополнительно задействует следующие опции <code>-foptimize-sibling-calls</code> , <code>-fcse-follow-jumps</code> , <code>-fcse-skip-blocks</code> , <code>-fgcse</code> , <code>-fexpensive-optimizations</code> , <code>-fstrength-reduce</code> , <code>-frerun-cse-after-loop</code> , <code>-frerun-loop-opt</code> , <code>-fcaller-saves</code> , <code>-fforce-mem</code> , <code>-fpeephole2</code> , <code>-fschedule-insns</code> , <code>-fschedule-insns-after-reload</code> , <code>-fregmove</code> , <code>-fstrict-aliasing</code> , <code>-fdelete-null-pointer-checks</code> и <code>-freorder-blocks</code> . Этот уровень оптимизации не разворачивает циклы, не выполняет оптимизацию подстановок (<code>inlining</code>) и переназначение регистров.
<code>-O3</code>	В дополнение к опциям, включаемым при <code>-O2</code> , устанавливает также <code>-finline-functions</code> и <code>-frename-registers</code> .
<code>-Os</code>	Оптимизирует размер программы. Устанавливает все опции, действующие при <code>-O3</code> . Устанавливает опции <code>-falign-loops</code> , <code>-falign-jumps</code> , <code>-falign-labels</code> и <code>-falign-functions</code> с параметром <code>=1</code> , что не допускает вставку пустого пространства для применения выравнивания.

-o filename

Назначает имя для выходного файла. При тип выводимой информации не имеет значения. Это может быть исходный код после предобработки, ассемблерный код, объектный модуль или скомпонованный двоичный машинный код. Опция `-o` может назначать имя только одного выходного файла, поэтому при выработке нескольких файлов применять ее не следует.

Без указания этой опции выводимые компилятором файлы, которые содержат готовые к выполнению машиной скомпонованные программы, по умолчанию имеют имя `a.out`.

Опция может быть записана в форме `--output`.

--optimize level

То же, что и `-O`.

--output filename

То же, что и `-o`.

--output-class-directory=directory

Java

То же, что и `-f-output-class-dir`.

-p

Включает в программу дополнительный код, который выводит информацию, пригодную для анализа профилирующей программой `prof`. Эту опцию следует исполь-

зовать как при компиляции исходных, так и при компоновке объектных файлов. См. также `-pg`. Опция может быть записана в форме `--profile`.

-P**Pre**

По этой опции препроцессор при его задействовании с опцией `-E` не будет генерировать директивы `#line`. Опция может быть записана в форме `--no-line-commands`.

--param name=value

Существуют некоторые внутренние ограничения, которые компилятор GCC учитывает для определения допустимого количества оптимизаций программы. Эти ограничения устанавливаются этой опцией значением `value` для указываемого в поле `name` именованного параметра оптимизации. В таблице Г.5 перечислены имена и допустимые значения параметров оптимизации.

Другая форма представления этой опции: `-param`.

Таблица Г.5. Параметры оптимизации, используемые с опцией `--param`

Имя параметра	Значение
<code>max-delay-slot-insn-search</code>	Наибольшее количество просматриваемых инструкций при поиске инструкции для заполнения слота задержки (<code>delay slot</code>). Увеличение значения этого параметра может улучшить генерируемый код, но при этом увеличится время компиляции. По умолчанию равно 100.
<code>max-delay-slot-live-search</code>	Наибольшее количество просматриваемых блоков при поиске блока с подходящим временем жизни информации в регистрах. Увеличение значения этого параметра может улучшить генерируемый код, но увеличит время компиляции. По умолчанию равно 333.
<code>max-gcse-memory</code>	Максимальный размер памяти, которая может быть выделена для выполнения оптимизации CSE (Global Common Subexpression Elimination). При недостаточном объеме памяти оптимизация не проводится. По умолчанию установлено значение, равное 50 мегабайт (52248800).
<code>max-gcse-passes</code>	Максимальное количество проходов (итераций) оптимизации CSE (Global Common Subexpression Elimination). По умолчанию равно 1.
<code>max-inline-insns</code>	Ограничение максимального количества инструкций метода, к которому может применяться расширение подстановкой кода. По умолчанию равно 600.
<code>max-pending-list-length</code>	Максимальное количество элементов ветви кода, которые могут сохраняться планировщиком слотов (<code>slot scheduler</code>) в списке зависимостей, ожидающих результата вычисления условия, до перезапуска трассирующего механизма. Большой объем кода функции может порождать тысячи зависимостей. По умолчанию равно 32.

-pass-exit-codes

Компилятор будет игнорировать ненулевые коды завершения любой стадии компиляции. Такие значения кода завершения сообщают об ошибках выполнения. При

этом код завершения GCC будет равен наибольшему коду ошибки, возвращаемому запускаемыми процессами (стадиями обработки). Обычно, если любой процесс или стадия обработки возвращает ненулевой код завершения, то компиляция прерывается и компилятор возвращает код, соответствующий произошедшей ошибке.

-pedantic

При компиляции программ на языках *C* и *C++* с этой опцией любые отступления от требований стандартов ISO вызывают выдачу предупредительных сообщений, предусмотренных этими стандартами. Без указания этой опции допускается использование расширений GNU, однако при этом будут успешно компилироваться и программы, отвечающие стандартам ISO, (хотя для некоторых из них может потребоваться применение опции **-ansi**).

Для языка *C* применяемый стандарт зависит от установки опции **-std**. При указании опцией **-std** стандарта *gnu89* рассматриваемая опция применяет правила C89. Следует учесть, что опция **-pedantic** определяет выдачу только тех сообщений, которые предусмотрены стандартами ISO. Поэтому существует возможность того, что в некоторых случаях не соответствующий стандарту код будет скомпилирован без выдачи предупреждений.

При компиляции с языка *C* опция **-pedantic** не распространяется на любые выражения, которые стоят после **_extension_**.

Для программ на языке *C++* при отсутствии опций **-fpermissive** и **-pedantic** по умолчанию применяется опция **-fpedantic-errors**.

При компиляции с этой опцией программ на языке *Fortran* использование расширений, не отвечающих стандарту Fortran 77, приводит к выдаче компилятором предупредительных сообщений. Предупреждения в частности выдаются при использовании в строковых константах конструкций языка *C* (таких как "\n"). Определенные расширения GNU и даже некоторые конструкции традиционного стандарта языка *Fortran* также могут вызывать выдачу предупредительных сообщений. Программы стандарта Fortran 77 будут одинаково хорошо компилироваться в GCC как с этой опцией, так и без нее. Опция не требует строгого следования стандарту.

Другая форма записи этой опции **--pedantic**.

-pedantic-errors

C, C++, Fortran

Эта опция действует так же, как опция **-pedantic**. Отличие состоит в том, что сообщения диагностики выводятся как ошибки, а не как предупреждения.

При компиляции программ на языке *C++* при отсутствии опций **-fpermissive** и **-pedantic** опция **-fpedantic-errors** применяется по умолчанию.

Опция может быть записана в форме **--pedantic-errors**.

-pg

C, C++, Fortran

Включает дополнительный код, который выводит информацию, пригодную для ее дальнейшего анализа профилирующей программой *gprof*. Эту опцию следует использовать как при компиляции исходного кода, так и при компоновке объектных файлов. См. также опцию **r**.

-pipe

Использует вместо временных файлов (intermediate files) программные каналы потоков ввода-вывода (pipes) для передачи выхода одной стадии компиляции на вход другой ее стадии. В операционных системах Unix, OS/2 и др. каналы служат для передачи выхода одной программы на вход другой программы. Опция может вызывать сбой компиляции в случае, когда используемый ассемблер не способен принимать входной поток через программный канал.

Опция может быть записана в форме `--pipe`.

--prefix *prefix*

То же, что и опция `-B`.

--preprocess

Pre

То же, что и `-E`.

-print-file-name=*library*

Выводит путь расположения указанной библиотеки. При этом никаких дальнейших действий не предпринимается. См. также опции `-print-libgcc-file-name` и `-print-prog-name`.

Опция может быть записана в форме `--print-file-name`.

-print-libgcc-file-name

Выводит путь расположения библиотеки `libgcc.a`. Действует так же, как и опция `-print-file-name=libgcc.a`. Опция может быть записана в форме `--print-libgcc-file-name`.

--print-missing-file-dependencies

Pre

То же, что и опция `-MG`.

-print-multi-directory

Выводит каталог, соответствующий установке `multilib` для поиска используемых библиотек. Имя пути определяется значением переменной окружения `GCC_EXEC_PREFIX`. Никаких дальнейших действий не предпринимается.

Эта опция может быть записана в форме `--print-multi-directory`.

-print-multi-lib

Выводит установки `multilib`, определенные в командной строке, вместе с соответствующими опциями. При этом никаких дальнейших действий не предпринимается. В вырабатываемом по этой опции выходе в качестве разделителя списка используется точка с запятой ";", в опциях вместо дефисов стоят символы "@"". Это упрощает обработку выходного текста в командной оболочке (shell).

Опция может быть записана в форме `--print-multi-lib`.

-print-prog-name=program

Выводит полное имя расположения указанной в поле *program* программы (такой программы как *cc1* или *cpp0*.) Никаких дальнейших действий не предпринимается. См. также **-print-file-name**.

Опция может быть записана в форме **--print-prog-name**.

-print-search dirs

Выводит список путей расположения, где GCC проводит поиск программ подпроцессов, запускаемых им при компиляции. Также выводит список каталогов для поиска используемых в компоновке библиотек. Никаких дальнейших действий при этой опции компилятором не предпринимается. Если при компиляции не удается обнаружить требуемую программу или библиотеку, то можно как переписать ее в доступный для поиска каталог, так и добавить путь ее расположения в список переменной **GCC_EXEC_PREFIX**.

Опция может быть записана в форме **--print-search dirs**.

--profile

То же, что и **-p**.

--profile-blocks

То же, что и **-a**.

-Q

При этой опции по мере компиляции будут выводиться имена каждой пройденной функции и в конце каждого прохода — статистика, включающая время компиляции и время компоновки программы.

-remap**Pre**

Указывает препроцессору проверять в каждом каталоге возможного расположения включаемых заголовочных файлов наличие файла с именем **header.gcc**. Такие файлы, если они присутствуют, используются компилятором для определения замещений имен заголовочных файлов (header files). Каждая строка файла состоит из используемого в программах имени заголовочного файла и имени соответствующего ему в действительности используемого файла, находящегося на диске. Например, следующие строки файла **header.gcc** определяют замену длинных имен заголовочных файлов короткими именами.

```
NotSupportedException.h notsup.h
RollbackException.h rollbak.h
TransactionRequiredException.h transreq.h
```

-S**Linker**

Удаляет из выполнимого файла таблицу программных символов (symbol table) и информацию об их перемещаемой адресации (relocation information). Дает такой же результат, как применение утилиты **strip**.

-S

С этой опцией компилятор не задействует ассемблер и компоновщик. Выполняется компиляция исходных файлов в файлы с ассемблерным кодом, но дальнейшего асSEMBЛИРОВания в объектный код не происходит. Выход сохраняется в файлах с именами, соответствующими именам исходных файлов, но с суффиксом `.s`. При этом игнорируются все входные файлы с неизвестными компилятору суффиксами (см. таблицу Г.1), если тип их содержимого прямым образом не указан опцией `-x`.

Опция может быть записана в форме `--assemble`.

-save-tempS

Отменяет обычную процедуру удаления временных файлов, вырабатываемых на промежуточных стадиях компиляции. Файлы остаются в рабочем каталоге, обычно в текущем. Содержимое файлов соответствует суффиксам их имен (см. таблицу Г.1), или установкам опций `-x` в строке команды компилятору.

Опция может быть записана в форме `--save-tempS`.

-shared**Linker**

При этой опции компоновщик создает объектный модуль формата, который может компоноваться из разделяемой библиотеки во время выполнения программы. Если команда `gcc` используется для создания разделяемой библиотеки, то применение этой опции также отменяет выдачу компоновщиком ошибки из-за отсутствия метода `main()`.

Для успешной компиляции объектных модулей, предназначенных для размещения в разделяемых библиотеках (shared libraries), необходимо правильное использование соответствующей опции `-fPIC` или `-fPIIC`, а также специфичных опций пред назначаемой платформы. Опция `-shared` для правильной работы выходного кода может в частности требовать генерации специальных конструкторов. Выдаваемые из за неправильной установки флагов сообщения об ошибках компиляции разделяемых модулей могут быть довольно сложными, в большинстве случаев их можно игнорировать без вреда для вырабатываемого кода.

Опция может быть записана в форме `--shared`.

См. также `-shared-libgcc`, `-static-libgcc` и `-static`.

-shared-libgcc**Linker**

Опция указывает компоновщику использовать разделяемую (shared) версию библиотеки `libgcc`. На системах, не поддерживающих использование разделяемых библиотек, или при отсутствии скомпонованной разделяемой версии библиотеки `libgcc` эта опция не оказывает действия.

При задействовании компоновщика через `g++`, `gcj` или `g77` этот флаг действует автоматически для выполнения требований обработки исключений. Разделяемая версия библиотеки `libgcc` необходима при обработке с помощью пользовательской разделяемой библиотеки исключений, порождаемых кодом другой разделяемой библиотеки. Функции разделяемой `libgcc` используются при этом как кодом, вызывающим исключение, так и обрабатывающим это исключение кодом.

См. также `-shared`, `-static-libgcc` и `-static`.

-specs=filename

По этой опции драйвер `gcc` считывает файл спецификаций с именем `filename`. В файле содержатся опции с их назначением, в соответствии с которым они передаются при компиляции подпроцессам. Файл спецификаций обрабатывается после считывания стандартных спецификаций, он может быть использован для замещения действующих по умолчанию правил задействования подпроцессов.

Опция может быть записана в форме `--specs`.

-static**Linker**

Компоновщик будет игнорировать любые разделяемые библиотеки и разрешать все внешние ссылки непосредственным включением в вырабатываемый объектный код статических объектных файлов. На системах, не поддерживающих динамической компоновки, установка этой опции не изменяет вырабатываемый выходной код. Опция может быть записана в форме `--static`. См. также опцию `-shared`.

-static-libgcc**Linker**

Назначает использование статической версии библиотеки `libgcc`. Применение этой опции может создать проблемы с обработкой исключений при компиляции программ на языках `C++` и `Java`.

См. также `-shared`, `-shared-libgcc` и `-static`.

-std=name**C**

Указывает применяемый стандарт языка программирования `C`. Распознаваемые значения поля `name` перечислены в таблице Г.6. Опция отключает ключевые слова расширения GNU `asm`, `typeof` и `inline`. Альтернативные формы этих ключевых слов `__asm__`, `__typeof__` и `__inline__` остаются доступными.

Опция может быть записана в форме `-std`. См. также `-ansi`.

Таблица Г.6. Имена стандартов языка C для использования с опцией `-std`

Значение параметра	Описание
<code>iso9899:1990</code>	Стандарт ISO C89. При этом также устанавливаются флаги <code>-fno-traditional</code> , <code>-fno-writeable-strings</code> , <code>-fno-asm</code> , <code>-fno-nonansi-builtins</code> и <code>-fno-noniso-default-format-attributes</code> .
<code>iso9899:199409</code>	Усовершенствованная версия стандарта ISO C89. При этом также устанавливаются флаги <code>-fno-traditional</code> , <code>-fno-writeable-strings</code> , <code>-fno-asm</code> , <code>-fno-nonansi-builtins</code> и <code>-fno-noniso-default-format-attributes</code> .
<code>iso9899:1999</code>	Стандарт ISO C99. При этом также устанавливаются флаги <code>-fno-traditional</code> , <code>-fno-writeable-strings</code> , <code>-fno-asm</code> , <code>-fno-nonansi-builtins</code> и <code>-fno-noniso-default-format-attributes</code> .
<code>c89</code>	То же, что и <code>iso9899:1990</code> .
<code>c99</code>	То же, что и <code>iso9899:1999</code> .
<code>gnu89</code>	Стандарт ISO C89 с расширениями GNU и подключением некоторых определений стандарта ISO C99. При этом также устанавливаются флаги <code>-fno-traditional</code> , <code>-fno-writeable-strings</code> , <code>-fno-asm</code> , <code>-fno-nonansi-builtins</code> и <code>-fno-noniso-default-format-attributes</code> .

-symbolic**Linker**

Создает подшивки обращений к глобальным символам при сборке разделяемых объектов. Этот подход является альтернативой компоновке с использованием опций **-shared** и **-static**. Этот способ поддерживается только некоторыми платформами, такими как некоторые из систем SVR4 и DG/UX.

Опция может быть записана в форме **--symbolic**.

-syntax-only

По этой опции компилятор проверяет синтаксис входного исходного кода, выводит все сообщения об ошибках и все предупреждения. После этого обработка останавливается, никаких дальнейших действий не производится.

--target *machine*

То же, что и опция **-b**.

--target-help

Выводит список всех опций командной строки, специфичных по отношению к пред назначаемой целевой платформе.

См. также **--help**.

--trace-includes**Pre**

То же, что и **-H**.

-traditional**C**

Данная опция распознается компилятором, но дальнейшая ее поддержка прекращена, (deprecated option). Включает поддержку компилятором первичного стандарта "K&R C" (т.е. язык C Кернигана и Ритчи). Учтите, что программы на традиционном C не компилируются с заголовочными файлами стандарта "ISO C". Эта опция также устанавливает опции **-traditional-cpp** и **-f writable-strings**. Другая форма записи этой опции: **--traditional**. См. также **-f allow-single-precision**.

-traditional-cpp**C**

Включает поддержку препроцессором правил стандартного препроцессора языка C. Другая форма этой опции: **--traditional-cpp**.

-trigraphs**Pre**

Включает поддержку триграфов (trigraphs). Эта опция устанавливается автоматически при включении опций **-ansi** и **-std**.

При этой опции девять последовательностей из трех буквенных знаков, начинающиеся с двух знаков вопроса "??", транслируются в отдельные буквенные символы в соответствии со следующим списком:

??=	#	??([??<	{
??/	\	??)]	??>	}
??'	^	??!		??-	~

Другая форма записи этой опции: **--trigraphs**.

-time

Выводит отчет о времени, занятом каждым подпроцессом компиляции программы. В каждой строке выводится пользовательское время (user time, т.е. время занятное выполнением кода подпроцесса) и системное время (system time т.е. время затраченное на системные вызовы). Следующий пример показывает вывод по опции **-time** при компиляции программы на языке C++ в выполнимый объектный формат.

```
gcc -time fortest.cpp -o fortest.o
# cc1plus 0.14 0.05
# as 0.00 0.01
# collect2 0.10 0.03
```

Другая форма этой опции: **--time**.

-U *name*

Linker

Добавляет указанное имя в таблицу программных символов (symbol table) в качестве символа, предназначенного для разрешения компоновщиком при сборке объектного кода. Компоновщик будет разрешать эту ссылку загрузкой объектного модуля, содержащего определение символа с таким именем.

Другая форма этой опции **--force-link**.

-U*macro*

Linker

Удаляет ранее сделанное макроопределение с именем, указанным в поле *macro*. Все опции **-D** обрабатываются раньше опций **-U**, а опции **-U** в свою очередь обрабатываются раньше любых опций **-include** и **-imacros**.

Другая форма этой опции **--undefine-macro**.

-undef

Pre

При этой опции препроцессор не будет предопределять никаких нестандартных макросов. Опция подавляет такие архитектурные макроопределения как **_unix_**, **_OpenBSD_**, **_mips_**, **_linux_**, **_vax_** и т.п.

--undefine-macro *macro*

Pre

То же, что и **-U**.

--use-version *version*

Pre

То же, что и **-V**.

--user-dependencies

Pre

То же, что и **-MM**.

-V

Выводит номер текущей версии компилятора и список всех команд, запускаемых на каждой стадии преобразования исходного кода в готовую к выполнению программу.

му. При отдельном использовании этой опции будет выводиться только номер текущей версии компилятора. В сочетании с опцией `--help` выводится полный список команд.

При компиляции программ на языке *Fortran* опция устанавливает также флаг `-fversion`. Другая форма этой опции `--version`.

См. также `-###`.

-V version

Назначает версию `gcc`, которая должна быть запущена при компиляции. Эта опция имеет значение только если в вашей системе установлено несколько версий компилятора. По умолчанию запускается наиболее свежая версия.

Опция действует через изменение используемого компилятором и его компонентами префикса пути расположения. Обычно установленные версии компилятора находятся в каталогах `/usr/local/lib/gcc-lib/machine/version`.

Опция может быть записана в форме `--use-version`. См. также `-b` и `-B`.

--verbose

То же, что и `-v`.

-W

Отменяет выдачу всех предупредительных сообщений. То же, что `--no-warnings`.

-W

Включает выдачу семейства предупредительных сообщений, относящихся к коду, способному вызывать те или иные проблемы. Такие сообщения помогают программисту создавать более чистый и уверенно переносимый код. Опция включает обработку следующих ситуаций:

- **Сравнение.** (Comparison.) Предупреждение выдается при проверке беззнаковой величины на отрицательность (т.е., что ее значение меньше нуля). Например, следующее сравнение будет всегда давать положительный результат из-за того, что переменная `x` беззнакового типа никогда не будет меньше нуля:

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

- **Сравнение.** (Comparison.) Выдается предупреждение при сравнении величины со знаком с беззнаковой величиной. Возможно получение ошибочного результата, если при сравнении беззнаковая величина приводится к типу со знаком. Выдача этого вида предупреждения может быть отключена опцией `-Wno-sign-compare`.

- **Сравнение.** (Comparison.) Синтаксис языка *C* для числовых выражений отличается от синтаксиса вычисления условий. Для выражений, подобных следующему, будут выдаваться предупреждения:

```
if(a < b < c) . . .
```

Алгебраический синтаксис выражения условия допустим здесь только в случае, когда значение переменной **b** принадлежит открытому интервалу между значениями величин **a** и **c**. В языке *C* представленное выражение эквивалентно следующей конструкции кода, которая не будет давать непредвиденных ошибок:

```
int result;
result = a < b;
if (result < b) . . .
```

- **Возвращение функцией константы.** (*Const return.*) Предупреждение выдается, когда возвращаемое значение функции объявлено как константа. Объявление **const** в таких случаях не имеет смысла, потому что функции возвращают значения как *rvalue*, т.е. значением результата правой части выражения присваивания.
- **Инициализация агрегатных типов.** (*Aggregate initializers.*) Выдается предупредительное сообщение, когда начальные значения для сборного типа данных указаны не для всех его членов. В следующем примере такие предупреждения будут выданы как для массива, так и для структуры:

```
struct {
    int a;
    int b;
    int c;
} tmp = { 1, 2 };
int arr[10] = { 1, 2, 3, 4, 5};
```

- **Неиспользуемые результаты выражений.** (*No side effect.*) В случаях, когда вычисление выражения не изменяет никакой величины. В этом примере результат сложения не используется:

```
int a = 1;
int b = 1;
a + b;
```

- **Переполнение типа.** (*Overflow.*) Во время компиляции программ на языке *Fortran* выдаются предупреждения при переполнении числового типа с плавающей точкой в объявлениях констант.
- **Возвращаемые значения.** (*Return value.*) В случаях, когда код функции не обязательно возвращает результат. В следующем примере функция не возвращает результат при отрицательных значениях **x**:

```
ambigret(int x)
{
    if(x >= 0)
        return(x);
}
```

- **Синтаксис объявления **static**.** (*Static syntax.*) Выдаются предупреждения в случаях, когда ключевое слово **static** стоит не в начале строки объявления.
- **Неиспользуемые аргументы.** (*Unused arguments.*) При использовании опции **-Wall** или **-Wunused** совместно с **-W** выдаются предупреждения для всех аргументов функции, не используемых в коде определения этой функции.

Опция может быть записана в форме **--extra-warnings**.

-Wa,*optionlist*

Asm

Поле **optionlist** содержит список разделенных запятой опций, которые должны быть переданы ассемблеру. Все опции, отделенные запятыми, передаются ассемблеру как отдельные опции командной строки. Другая форма этой опции **--for-assembler**. См. также **-Wp** и **-Wl**.

-Waggregate-return

C, C++, ObjC

Опция определяет выдачу предупредительного сообщения в случае, когда функция возвращает структуру (агрегатный тип **struct**), объединение (**union**) или массив (**array**).

-Wall

При компиляции программ на языках *C* и *Objective-C* эта опция равносильна применению набора опций **-Wreturn-type**, **-Wunused**, **-Wimplicit**, **-Wswitch**, **-Wformat**, **-Wparentheses**, **-Wmissing-braces**, **-Wsign-compare** и **-Wmultichar**. Опция **-Wunknown-pragmas** устанавливается только для pragma-директив, отсутствующих в заголовочных файлах. При использовании опции **-O** также устанавливается **-Wunused**.

При компиляции программ на *C++* дополнительно к перечисленным устанавливается опции **-Wctor-dtor-privacy**, **-Wnon-virtual-dtor**, **-Wreorder** и **-Wnon-template-friend**.

При компиляции программ на языке *Fortran* устанавливается только опции **-Wunused** и **-Wunused**.

Для компиляции программ на *Java* эта опция равносильна совместному применению опций **-Wredundant-modifiers**, **-Wextraneous-semicolon** и **-Wunused**.

Опция может быть записана в другой ее форме **--all-warnings**.

--warn-

То же, что и **-W**.

-Wbad-function-cast

C

Опция включает выдачу предупреждений при несоответствии типа возвращаемого функцией значения типу левой части выражения. Иначе говоря, в случае приведения (cast) типа возвращаемого функцией результата. Следующий пример кода при этой опции будет генерировать предупреждение на вызове функции:

```
int glim()
{
    return(88);
}
...
char *cp;
cp = (char *)glim();
```

-Wcast-align**C, C++, ObjC**

Выдает предупреждение в случае проблем с выравниванием, возможных при приведении (cast) типов указателей. Например, на некоторых машинах возможно выравнивание адресации к данным типа `int` по границе 2 или 4 байта. В случае приведения указателя `char` к указателю `int` (т.е. фактически адресации к типу `char` как к `int`) из-за выравнивания возможно получение неправильного результата.

-Wcast-qual**C, C++, ObjC**

Выдает предупреждение, когда вызов функции отменяет действие квалификатора `const`. Пример:

```
const char *conchp;
char *chp;
. . .
chp = (char *)conchp;
```

-Wchar-subscripts**C, C++, ObjC**

Выдает предупреждение при использовании переменной типа `char` в качестве индекса массива. Тип `char` часто по умолчанию считается знаковым, что может приводить к ошибкам.

-Wcomment**C, C++, ObjC**

Выдает предупреждение, когда внутри комментария типа "/* . . . */" находится сочетание символов "/*". Предупреждение также выдается в случае, когда строка, содержащая комментарий отмеченный сочетанием "/*", заканчивается символом обратной наклонной черты '\'. Что, конечно, означает то, что текущий комментарий продолжается в следующей строке кода.

-Wconversion**C, C++, ObjC**

Предупреждение выдается, когда наличие прототипа требует конвертирования типов, которого не было бы при отсутствии прототипа. Имеются в виду такие случаи, когда требуется конвертирование между действительным и целым типом, или преобразование знаковых величин в беззнаковые, а также при изменении диапазона величин. Предупреждения выдаются только в случаях явного принудительного конвертирования данных (coercion), а не для назначенного приведения типов (cast). Например, в этом коде для первого оператора предупреждение будет выдано, а для другого — нет:

```
unsigned int rrecp;
rrecp = -1;
rrecp = (unsigned int)-1;
```

-Wctor-dtor-privacy**C++**

Выдает предупреждение для класса, который невозможно использовать из-за того, что его конструкторы и деструкторы объявлены с атрибутом `private`, либо класс не имеет доступных для использования методов.

-Wdeprecated**C++**

Действует по умолчанию. Выдаются предупреждения об использовании не поддерживаемых свойств (deprecated features) языка C++. Отменяется опцией **-Wno-deprecated**.

-Wdeprecated-declarations**C, C++, ObjC**

Действует по умолчанию. Выдаются предупреждения об использовании операторов и ключевых слов, отмеченных атрибутом, обозначающим прекращение поддержки, (deprecated). Отменяется опцией **-Wno-deprecated-declarations**.

-Wdisabled-optimization

Выдает предупреждения при запросах отключенных способов оптимизации. Эти обстоятельства связаны с ограничениями самого компилятора, а не с проблемами в коде программы. GCC может не поддерживать слишком сложных и/или слишком долго выполняемых оптимизаций.

-Wdiv-by-zero

Действует по умолчанию. Выдает предупреждения о делении целого числа на ноль в случаях, когда компилятор может определить такую ситуацию. Отменяется обратной опцией **-Wno-div-by-zero**. Для деления на ноль чисел с плавающей точкой предупреждений не предусмотрено.

-Weffc++**C++**

Предупреждения об отступлениях от правил разметки кода, изложенных в книге Скотта Маерса (Scott Myers, "Effective C++"). Учтите, что стандартные библиотеки написаны без соблюдения этих правил. Поэтому при этой опции вы можете увидеть множество сообщений, относящихся к коду библиотек.

-Werror

Преобразовывает все предупреждения в сообщения об ошибках компиляции.

-Werror-implicit-function-declaration**C**

Выдает предупреждение при всяком использовании функции до ее объявления. См. также **-Wimplicit-function-declaration**.

-Wextern-inline**C++**

Выдает предупреждение, если функция объявлена одновременно как **extern** и как **inline**.

-Wextraneous-seicolon**Java**

Выдает предупреждения о лишних символах точки с запятой ";", перед которыми нет оператора. Опция подавляет использование пустых операторов.

-Wfloat-equal

Выдается предупреждение при сравнении на равенство двух чисел с плавающей точкой, потому что такая ситуация скорее всего возникает из-за ошибки в программе.

Природа арифметических операций над числами с плавающей точкой такова, что равенство результатов вычислений встречается чрезвычайно редко и носит случайный характер. То есть точное сравнение таких чисел будет давать отрицательный результат даже тогда, когда числа отличаются настолько мало, что по логике программы следует их считать равными. Далее вам предлагается пример способа сравнения чисел с плавающей точкой на равенство с точностью до 10^{-5} :

```
double delta = 0.00001;
. . .
if((val > val2-delta) && (val < val2+delta) {
    /* здесь val1 и val2 считаются равными */
}
```

-Wformat**C, C++, ObjC**

Проверяет вызовы таких функций как `printf()` и `scanf()` и выдает предупреждение в случае, если типы аргументов не соответствуют формату их вывода. Например, в следующем примере показан оператор, в котором значение типа `double` предназначается к выводу в формате типа `int`:

```
double dvalue = 44.44
. . .
printf("The value %d is bad.\n", dvalue);
```

Формат вывода тестируется в соответствии со свойствами библиотеки GNU libc версии 2.2. Она включает в себя определения, соответствующие C89, C99, POSIX и некоторым расширениям GNU для BSD. При установленной опции `-pedantic`, предупреждения будут выдаваться при любых отступлениях от стандартных правил форматирования.

Будут проверяться функции, поддерживающие форматирующие строки, а именно следующие: `printf()`, `fprintf()`, `sprintf()`, `scanf()`, `fscanf()`, `strftime()`, `vprintf()`, `vfprintf()` и `vsprintf()`. Для стандарта C99 кроме приведенных еще функции `snprintf()`, `vsnprintf()`, `vscanf()`, `vfscanf()` и `vsscanf()`. Для систем X/Open также `strfmon()`, `printf_unlocked()` и `fprintf_unlocked()`.

См. опции `-Wformat-extra-args`, `-Wformat-nonliteral` и `-Wformat-security`. См. также раздел "Атрибуты" в главе 4. Опция автоматически устанавливается при применении опции `-Wall`. Ее действие можно отключить обратной опцией `-Wno-format`.

-Wformat2**C, C++, ObjC**

Действует так же, как и одновременное применение опций `-Wformat`, `-Wformat-nonliteral` и `-Wformat-security`.

-Wformat-extra-args**C, C++, ObjC**

Действует по умолчанию. Во время действия опции **-Wformat** указание рассматриваемой опции в ее обратной форме **-Wno-format-extra-args** подавляет вывод предупредительных сообщений о неиспользуемых аргументах, которые передаются функциям, подобным `printf()` и `scanf()`.

-Wformat-nonliteral**C, C++, ObjC**

При установленной опции **-Wformat** эта опция выдает предупреждения в случаях, когда форматирующий аргумент таких функций как `printf()` и `scanf()` не является строковой константой.

-Wformat-security**C, C++, ObjC**

При установленной опции **-Wformat** эта опция выдает предупреждения в случаях, когда обращение к таким функциям как `printf()` и `scanf()` может быть небезопасным. Использование переменной в качестве форматирующего аргумента при вызове таких функций считается ненадежным из-за возможности использования "%n".

-Wformat-y2k**C, C++, ObjC**

Действует по умолчанию. Указание **-Wno-format-y2k** отключает выдачу предупреждений о ситуациях, когда аргумент формата функции `strftime()` допускает вывод календарного года в виде двух десятичных цифр.

-Wglobals**Fortran**

Действует по умолчанию. Указание обратной опции **-Wno-globals** отключает выдачу предупреждений о совпадении глобальных имен подпрограмм, функций, блоков данных, или блоков общего назначения (common blocks) с именами встроенных функций (intrinsics). При этом также подавляются предупреждения о таких нарушениях правил вызова глобальных функций и подпрограмм, как неправильное количество или несоответствие типов аргументов.

-Wimplicit**Fortran**

Выдает предупреждения о явных (implicit) объявлениях переменных, массивов или функций.

-Wimplicit-int**C**

Выдает предупреждения для объявлений, в которых отсутствует прямое указание типа. Эта опция устанавливается автоматически при использовании опции **-Wimplicit** или **-Wall**.

-Wimplicit-function-declaration**C**

Выдает предупреждения об использовании функций до их объявления. См. также **-Werror-implicit-function-declaration**. Опция автоматически устанавливается при использовании **-Wimplicit** или **-Wall**.

-Wimplicit**C**

То же, что и одновременное применение опций `-Wimplicit-int` и `-Wimplicit-function-declaration`. Эта опция устанавливается автоматически при использовании опции `-Wall`.

-Wimport**C, C++, ObjC**

Действует по умолчанию. Указание обратной опции `-Wno-import` подавляет вывод препроцессором предупреждений об использовании директив `#import`.

-Winline**C, C++, ObjC**

Выдает предупреждения о невозможности подстановки кода функции, объявленной с атрибутом `inline`.

-Wl,*optionlist***Linker**

Список опций, находящийся в поле `optionlist`, передается компоновщику. Все элементы этого списка, разделенные запятыми, ставятся отдельными опциями командной строки вызова компоновщика.

См. также `-xlinker`, `-Wa` и `-Wp`.

-Wlarger-than-size**C, C++, ObjC, Java**

Выдает предупреждение о превышении допустимого размера объекта, а также когда размер возвращаемого функцией значения превышает `size` байт.

-Wlong-long**C, C++, ObjC**

Применяется по умолчанию, но действует только совместно с опцией `-pedantic`. Опция `-Wlong-long` назначает выдачу предупредительных сообщений об использовании типа данных `long long`. Действие этой опции можно отменить применением обратной опции `-Wno-long-long`.

-Wmain**C, C++**

Выдает предупреждение в случае, когда определение функции `main()` выглядит подозрительно. В общем случае это должна быть функция, имеющая внутреннюю компоновку, и возвращающая результат типа `int`. Она может не иметь аргументов, или иметь до трех аргументов подходящих для этого типов.

-Wmissing-braces**C, C++, ObjC**

Выдает предупреждения при неполной разметке скобками начальных значений элементов массивов. В следующем примере оба массива будут инициализированы корректно, но в определении массива `b` расположение начальных значений определено более точно.

```
int a[2][2] = { 0, 1, 2, 3 };
int a[2][2] = { { 1, 2 }, { 3, 4 } },
```

Опция `-Wmissing-braces` автоматически применяется при установке опции `-Wall`.

-Wmissing-declarations**C**

Выдает предупреждения в случаях, когда глобальная функция определяется без предварительного объявления, в котором могут быть указаны или не указаны типы ее аргументов. См. также **-Wstrict-prototypes** и **-Wmissing-prototypes**.

-Wmissing-format-attribute**C, C++, ObjC**

Выдает сообщения по функциям, которые могут быть кандидатами для назначения им атрибута `format`. Следует заметить, что эти предупреждения сообщают лишь о возможности установки атрибута. Опция действует только вместе с **-Wformat** или **-Wall**.

-Wmissing-noreturn**C, C++, ObjC**

Выдает сообщения по функциям, которые могут быть кандидатами для назначения им атрибута `noreturn`. Следует заметить, что эти предупреждения сообщают лишь о возможности установки атрибута. Каждый случай должен рассматриваться отдельно. Установка атрибута `noreturn` функции, которая в действительности так или иначе возвращает результат, может привести к трудно устранимым ошибкам в программе.

-Wmissing-prototypes**C**

Выдает предупреждения в случаях, когда глобальная функция определяется без предварительного объявления ее прототипа с типами аргументов.

См. также **-Wstrict-prototypes** и **-Wmissing-declarations**.

-Wmultichar**C, C++, ObjC**

Действует по умолчанию. Включает выдачу предупреждений при назначении буквенным константам (character constants) строк, имеющих длину более одного символа. Таких как '`ab`' или '`Plop`'. Код, генерируемый для такого типа объявлений, зависит от платформы. Поэтому следует избегать таких ситуаций в целях обеспечения переносимости программ. Действие этой опции можно отключить обратной опцией **-Wno-multichar**.

-Wnested-externs**C**

Включает предупреждения при использовании внутри функций объявлений `extern`.

-Wnon-template-friend**C++**

Действует по умолчанию. Выдает предупреждения в случаях, когда в качестве участника шаблона объявляется метод, не изменяющий типа аргументов (friend function), который не может быть преобразован в шаблон.

Расширение GNU языка C++, связанное с реализацией этих функций, имеет приоритет перед стандартными определениями. В GNU C++ имена friend-функций объявляются без квалификаторов. Сейчас такое поведение не применяется по умолчанию и опция служит для обеспечения совместимости с уже существующим кодом. Действие этой опции можно отключить обратной опцией **-Wno-non-template-friend**.

Опция `-Wnon-template-friend` применяется автоматически при установке опции `-Wall`.

-Wnon-virtual-dtor

C++

Выдает предупреждения в случаях, когда похоже, что не виртуальный дескриптор должен объявляться как виртуальный. Не виртуальный дескриптор не может быть выполнен при обращении к объекту из вышестоящих классов. Автоматически применяется при установке опции `-Wall`.

-Wold-style-cast

C++

Выдает предупреждения при использовании традиционного стиля приведения типов (стиля языка C) вместо более новых операторов стандарта C++ `static_cast`, `const_cast`, `reinterpret_cast`. Например:

```
class A { . . . };
class B: public A { . . . };

. . .

A* a = new A();
B* b = a;                                // конвертирование типов
A* a2 = static_cast<A*>(b);   // стандартное приведение C++
```

-Wout-of-date

Java

Действует по умолчанию. Обратная опция `-Wno-out-of-date` подавляет выдачу предупреждений о том, что используемый файл класса устарел по отношению к соответствующему ему исходному файлу.

-Woverloaded-virtual

C++

Выдает предупреждение, когда объявление функции скрывает виртуальную функцию основного класса. В следующем примере функция `fn()` класса `A` оказывается скрытой объявлением в классе `B`:

```
class A {
    virtual void fn();
};

class B public A {
    void fn(int);
};
```

-Wp,optionlist

Pre

Список опций, находящийся в поле `optionlist`, передается препроцессору. Все элементы этого списка, разделенные запятыми, ставятся отдельными опциями командной строки препроцессора.

См. также `-Wa` и `-wl`.

-Wpacked

C, C++, ObjC

Выдает предупреждение о том, что указанный атрибут `packed` не имеет действия. Например, следующая структура (`struct`) будет занимать четыре байта независимо от атрибута `packed`:

```
struct fourbyte {
    short x;
    char a;
    char b;
} __attribute__((packed));
```

См. раздел "Атрибуты" в главе 4, опции **-fpack-struct** и **-Wpadded**.

-Wpadded

C, C++, ObjC

Выдает предупреждение, когда компилятор вставляет свободное пространство (padding) между полями структуры (как для выравнивания в памяти полей, так и для выравнивания всей структуры). В некоторых случаях возможно переупорядочивание полей для уменьшения размера структуры и обеспечения должного выравнивания без вставки пустого пространства. В следующем примере для обеспечения выравнивания по границе 2-х байтов нужна вставка одного пустого байта перед полем **b** типа **short**:

```
struct pad {
    char a;
    short b;
    char c;
};
```

-Wparentheses

C, C++, ObjC

Выдает предупреждения для синтаксически допустимых конструкций кода, которые могут быть сложными для понимания программистом (из-за порядка следования операторов или пропущенных скобок в выражениях). Выражение в этом примере кода может вызвать предупреждение:

```
if (a && b || c) . . .
```

В следующем примере возможно заблуждение относительно отношений между операторами **if** и **else**:

```
if(a)
    if(b)
        m = p;
else
    a = 0;
```

Здесь разметка кода говорит о том, что **else** относится к первому оператору **if**, хотя на деле это не так. В таких случаях тоже выдается предупреждение.

Опция применяется автоматически при установке опции **-Wall**.

-Wpmf-conversions

C++

Действует по умолчанию. Указание обратной опции **-Wno-pmf-conversions** подавляет выдачу предупреждений при преобразовании (приведении) типа указателя к адресу метода класса (member function).

-Wpointer-arith**C, C++, ObjC**

Включает выдачу предупреждений при принятии компилятором решений, зависящих от размера типа указателя (`void`) или размера возвращаемого функцией результата. В GCC для обеспечения поддержки адресной арифметики размер зависимых величин по умолчанию в равен 1.

-Wprotocol**ObjC**

Действует по умолчанию. Указание обратной опции `-Wno-protocol` подавляет выдачу предупреждений при об отсутствии в протокольном классе метода, предусмотренного этим протоколом.

-Wredundant-decls**C, C++, ObjC**

Включает выдачу предупреждений при повторном объявлении символа в пределах одной области видимости переменных (scope). Предупреждения выдаются независимо от идентичности таких объявлений.

-Wredundant-modifiers**Java**

Выдает предупреждения об использовании лишних модификаторов. Например, если в объявлении метода одновременно использованы модификаторы `interface` и `public`.

-Wreorder**C++**

Выдает предупреждения в случае, если компилятор переупорядочивает методы инициализации в соответствии с последовательностью их объявления.

Например, следующий код представляет ситуацию, когда инициализаторы будут переупорядочены:

```
class Reo {
    int i;
    int j;
    Reo(): j(5), i(10) {}
};
```

Эта опция устанавливается автоматически при использовании опции `-Wall`.

-Wreturn-type**C, C++**

Включает выдачу предупреждений при объявлении функции без назначения типа возвращаемого результата и, соответственно, присвоении ему типа по умолчанию `int`.

--write-dependencies**Pre**

То же, что `-MD`.

--write-user-dependencies**Pre**

То же, что `-MMD`.

-Wselector**ObjC**

Выдает предупреждение об использовании переключателя (*selector*) для определения множества методов различных типов.

-Wsequence-points**C**

Включает выдачу предупреждений в случае использования в пределах выражения более одного обращения к одной и той же переменной, когда одно из этих обращений изменяет ее значение. Определения языка C допускают любой порядок вычисления промежуточных результатов выражения в пределах соблюдения требований приоритета операторов. При этом изменение значения переменной в одной части составного выражения может приводить к получению непредсказуемого результата другой части выражения.

Части выражения выделяются в соответствии в положением в нем следующих операторов:

```
; , && || ? :
```

Далее вам предлагается несколько примеров составных выражений, которые из-за неопределенного порядка вычисления их частей могут давать непредсказуемый результат:

```
s = a[s++];  
s = s--;  
a[s++] = b[s];  
a[s] = b[s += c];
```

-Wshadow**C, C++, ObjC, Java**

Включает выдачу предупреждений в случаях, когда объявление локальной переменной перекрывает использование аргумента вызова текущей функции, глобальной переменной или другой объявленной локальной переменной.

-Wsign-compare**C, C++, ObjC**

Выдает предупреждения о ситуациях, когда сравнение величины беззнакового типа со знаковой величиной может давать неправильный результат из-за приведения перед сравнением знакового типа к типу без знака. Опция автоматически устанавливается при использовании опции *-Wall*. Ее действие можно отменить использованием обратной опции *-Wno-sign-compare*.

-Wsign-promo**C++**

Выдает предупреждения при замещении объявления беззнакового или перечисляемого типа данных знаковым типом того же размера. Такое замещение предусмотрено стандартом языка, но в некоторых случаях может приводить к потере данных.

-Wstrict-prototypes**C**

Опция назначает выдачу предупреждений об объявлении или определении функций без предварительного указания количества и типов аргументов. См. также *-Wmissing-prototypes* и *-Wmissing-declarations*.

-Wsurprising**Fortran**

Предупреждения, выдаваемые при этой опции, указывают на конструкции исходного кода, которые могут неоднозначно интерпретироваться компилятором, и потому давать неожиданный для программиста результат. Речь идет о таких конструкциях, которые могут по-разному обрабатываться различными компиляторами и соответственно давать непредсказуемый результат при портировании кода. Предупреждения выдаются в следующих случаях:

- Выражения с двумя операторами в одной строке и неопределенным порядком их выполнения. Например, код `x***y*z` из-за пропущенных скобок может быть интерпретирован как `x** (y*z)` и как `(x**y) *z`. Установка флага `-fpedantic` также назначает выдачу предупреждений о таких ситуациях.
- Выражения с неоднозначной унарной операцией `"-"`. Например, выражение `-2**x` может быть истолковано как `(-2**)x` или как `- (2**x)`.
- Использование в качестве счетчика цикла `DO` действительного числа вместо целого. Это может давать неожиданный результат при использовании различных компиляторов.

-Wswitch**C, C++, ObjC**

Предупреждает об использовании перечисляемого типа в качестве индекса переключателей `case` оператора `switch`, когда отсутствует определение переключателя `default` и `case` определены не для всех возможных значений перечисляемого типа. Опция автоматически устанавливается при использовании опции `-Wall`.

-Wsynth**C++**

Выдает предупреждения о замещении (перегрузке) основных операций языка (operator synthesis different from cfront). Следующий пример показывает синтез в GCC оператора по типу `A& operator = (const A&)`, где препроцессор cfront использует оператор по умолчанию, определяемый пользователем, `operator =`:

```
class A {
    operator int()
    A& operator = (int)
};

main() {
    A a1;
    A a2;
    a1 = a2;
};
```

-Wsystem-headers**C, C++, ObjC**

Включает выдачу предупреждений при компиляции кода системных заголовочных файлов (system header files). Обычно все предупреждения, относящиеся к системным заголовочным файлам, подавляются.

Для выдачи предупреждений по неизвестным pragma-директивам в системных заголовочных файлах необходимо также указывать опцию `-Wunknown-pragmas`. По-

тому что опция `-Wall` тоже по умолчанию игнорирует системные файлы-заголовки.

-Wtraditional

C

Выдает предупреждения для стандартных конструкций языка, которые могут иметь различное значение, или не определены в традиционном стандарте языка C. Для обеспечения переносимости кода следует избегать использования некоторых неоднозначных или проблематичных конструкций. Вот список некоторых ситуаций, по которым возможна выдача предупреждений:

- **Преобразование типов** (Conversion). Конвертирование между типами чисел с фиксированной и с плавающей точкой, вызванное несоответствием определения функции объявлению ее прототипа. См. также `-Wconversion`. Пример вызова функции, который приведет к выдаче предупреждения:

```
void takedouble (double dval);  
.  
.  
int eighty = 80;  
takedouble(eighty);
```

- **Локальные функции** (External). Случай вызова функций, объявленных с атрибутом `external`, за пределами области их видимости (блока, в котором они были объявлены).
- **Инициализация** (Initial values). Присвоение начальных значений автоматическим агрегатным типам. Например, объединению (`union`) или структуре (`struct`), объявленному внутри функции.
- **Инициализация** (Initial values). Присвоение начального значения объединению (`union`) вызовет предупреждение. Даже при использовании для инициализации константы, равной нулю.
- **Метка** (Label). Совпадение имени метки с именем объявленной переменной.
- **Константы** (Constants). Случай объявления целочисленных констант с суффиксом `U`, а также констант вещественных чисел (real numbers) с суффиксами `F` или `L`.
- **Константы** (Constants). Объявления буквенных констант (символьных литералов) в форме десятичного числа при отличии диапазона и/или знака числа (+/-) от требований традиционного стандарта языка C. Предупреждения выдаются только для десятичной формы представления. Шестнадцатиричные и восьмиричные константы всегда воспринимаются как последовательность бит.
- **Буквенные строки** (Literal strings). Предупреждения выдаются при использовании метода объединения строк (`concatenation`) стандартного диалекта C.
- **Препроцессор** (Preprocessor). Использование имени макроса (`#macro`) в строковых литералах (`string literal`). Стандартный язык C не допускает этого, хотя в традиционном языке K&R C строки могут содержать имена макросов.
- **Препроцессор** (Preprocessor). Все директивы препроцессору, начинающиеся в первой позиции строки, не определенные в традиционном языке C.

- **Препроцессор (Preprocessor).** Макроопределения в форме функций без назначения аргументов.
- **Статические функции (Static).** При объявлении static-функции после объявления не статических функций. Некоторые компиляторы традиционного стандарта языка C не воспринимают такой код.
- **Оператор Switch.** При использовании в операторе `switch` переменной типа `long` в качестве операнда переключателя `case`.
- **Присутствие унарной операции "+"(Unary plus).**

-Wtrigraphs**C**

Включает выдачу предупреждений об использовании триграфов (trigraphs) в строковых константах. Есть замечательный пример. Одна из версий ядра Linux содержала такую строку: "`imm: parity error (???)\n`". Стандартным компилятором языка C она транслировалась в "`imm: parity error (?)\n`".

-Wundef**C, C++, ObjC**

Включает выдачу предупреждений при использовании не определенного идентификатора в выражении условия директивы `#if`.

-Wuninitialized

Выдает предупреждения при использовании автоматической переменной до ее инициализации. Также предупреждает о ситуации, когда вызов `setjmp()` может нарушить значение автоматической переменной. Опция должна использоваться только в сочетании с `-O`, потому что для определения таких случаев используются результаты оптимизации потока данных (data flow analysis).

Точность этих сообщений не гарантирована. В следующем примере показана трудно определяемая ситуация, когда функции `printf()` может быть передано как инициализированное, так и неинициализированное значение переменной `value`:

```
int value;
if(a < b);
    value = 5;
else if(a > c)
    value = 10;
printf("%d\n",value);
```

Из-за особенностей применяемых способов анализа потока данных действие рассматриваемой опции не распространяется на структуры (struct), объединения (union), массивы, любые переменные с атрибутом `volatile`, переменные адресуемые через ссылки на них, переменные, которые используются для вычисления значений, в дальнейшем нигде в программе не используемых.

Результаты анализа потока данных предупреждают об использовании оператора `setjmp()`, но не определяют места обращений к `longjmp()`. Поэтому предупреждения могут выдаваться и при отсутствии проблем.

Те же способы оптимизации потока данных используются и для исходного кода на языке *Fortran*. Следующий пример показывает ситуацию, когда точно определить

использование неинициализированной переменной `TVAL` не представляется возможным.

```
IF (IVAL .EQ. 1) TVAL = 5
IF (IVAL .EQ. 2) TVAL = 10
CALL SMON(TVAL)
```

Появление некоторых выдаваемых при этой опции ложных сообщений можно предотвратить объявлением функций, не возвращающих значение, с атрибутом `noreturn`.

Опция `-Wuninitialized` автоматически устанавливается при совместном использовании опций `-Wall` и `-O`.

-Wunknown-pragmas

Выдает предупреждения об использовании неизвестных директив `#pragma`. Если опция применена не автоматически вследствие установки `-Wall`, а указана явно, то ее действие распространяется и на системные заголовочные файлы (system header files).

-Wunreachable-code

Включает выдачу предупреждений о неиспользуемых при выполнении программы участках кода.

Следует с большой осторожностью удалять участки, по которым выдаются такие сообщения. Предупреждения могут выдаваться о подстановляемом (`inline`) коде функции или расширении имени макроса (`expantion of a macro`) при том, что другие экземпляры вхождений такого кода могут использоваться программой. Кроме того предупредительные сообщения могут выдаваться при намеренном пропуске участков условного кода установкой опций компилятора.

-Wunused

Эта опция устанавливает набор опций `-Wunused-function`, `-Wunused-label`, `-Wunused-parameter`, `-Wunused-value` и `-Wunused-variable`. При компиляции программ на языке *Fortran* при этой опции выдаются предупреждения по всем объявленным, но не использованным переменным. Опция автоматически применяется при установке `-Wall`.

-Wunused-function

Выдает предупреждения о неиспользуемых определениях статических (`static`) функций, а также при отсутствии определений объявленных функций. Опция `-Wunused-function` задействуется автоматически при использовании опции `-Wunused` или `-Wall`.

-Wunused-label

Включает выдачу предупреждений о неиспользуемых метках, объявленных в программе без атрибута `unused`. Опция `-Wunused-function` задействуется автоматически при использовании опции `-Wunused` или `-Wall`.

-Wunused-parameter

Выдает предупреждения о неиспользуемых аргументах функций, объявленных без атрибута `unused`. Опция `-Wunused-parameter` задействуется автоматически при использовании опции `-Wunused` или `-Wall`.

-Wunused-value

Выдает предупреждения о неиспользуемых локальных или не статических переменных, объявленных без атрибута `unused`. Автоматически задействуется при использовании опции `-Wunused` или `-Wall`.

-Wunused-variable

Выдает предупреждения о неиспользуемых локальных переменных или не статических переменных, объявленных без атрибута `unused`. Автоматически задействуется при использовании опции `-Wunused` или `-Wall`.

-Wwrite-strings**C, C++**

При компиляции программ на языке *C* предупреждает об использовании указателей типа `char *` для записи строковых констант (literal strings constant). При компиляции программ на языке *C++* выдает предупреждения о преобразовании (неявном приведении) строковых констант к типу `char *`.

Эта опция приносит ощутимую пользу, если требуется повышенное внимание к объявлению прототипов и типов данных с атрибутом `const`. В остальных случаях она приводит к появлению большого количества ненужных назойливых сообщений.

-xlanguage

Указывает на тип содержимого файлов, перечисленных в командной строке. Без этой опции тип файла определяется компилятором в соответствии с суффиксом имени этого файла. Опция действует на все имена файлов, которые стоят после нее в строке команды. В следующем примере команды файлы `morg.jmp` и `frampl` указаны компилятору как файлы с исходным кодом на языке *C*:

```
gcc -xc morg.jmp frampl
```

Опция может использоваться несколько раз в одной команде, каждый раз переключая применяемый язык программирования. Особое значение `none`, указанное в поле `language` служит для отключения действия предыдущей опции `-x`. Например, следующая командная строка указывает компилятору, что файлы `murk` и `stim.wad` — исходники на *C++*, файл с именем `hummer.c` в действительности содержит код на языке *Java*, в то время как `slamm.c` содержит исходный код на языке *C*:

```
gcc -xc++ murk stim.wad -xjava hummer.c -xnone slamm.c
```

Список возможных значений поля `language` представлен в таблице Г.7.

Опция может быть записана в форме `--language`.

Таблица Г.7. Спецификаторы языков для опции **-x**

Имя языка	Описание
ada	Исходный код на языке Ada.
assembler	Не требующий предобработки ассемблерный код.
assembler-with-cpp	Ассемблерный код, предназначенный к предобработке.
c	Требующий предобработки исходный код на языке C.
c++	Требующий предобработке исходный код на языке C++.
c++-cpp-out	Исходный код на языке C++, не подлежащий предобработке.
c-header	Заголовочный файл на языке C.
cpp-output	Исходный код на языке C, не подлежащий предобработке.
f77	Не подлежащий предобработке исходный код на языке Fortran.
f77-cpp-input	Исходный код на языке Fortran, предназначенный к предобработке.
java	Исходный код на языке Ada.
obj-cpp-output	Не требующий предобработки исходный код на языке Objective-C.
objective-c	Исходный код на языке Objective-C, предназначенный к предобработке.
ratfor	Исходный код на языке Fortran, предназначенный к предобработке препроцессором RATFOR.

-Xlinker option**Linker**

Служит для сквозной передачи опций компоновщику (linker). Обычно эта опция используется для указания компоновщику специфических опций предназначаемой системы. Например, если вы используете компоновщик системы System V и желаете передать ему опцию **-all**, то это можно сделать указанием `gcc` опции **-Xlinker -all**. Для передачи компоновщику нескольких опций следует использовать опцию **-Xlinker** несколько раз в одной командной строке последовательно с каждой передаваемой компоновщику опцией. Например, для указания компоновщику набора опций "**-woff 5,17**" их следует указывать `gcc` так: "**-Xlinker -woff -Xlinker 5,17**". Вариант "**-Xlinker -woff 5,17**" не работает.

Опция может быть записана в форме **--for-linker**. См. также **-Wl**.

Словарь терминов

Английские

ABI (Application Binary Interface) — Двоичный (машинного уровня) интерфейс приложений. Спецификация, описывающая форматы исполняемых файлов и правила взаимодействия прикладных программ с Центральным Процессорным Блоком, *CPU*.

absolute adress — См. *абсолютный адрес*.

Ada — Язык программирования высокого уровня. Был разработан французской компанией "Сии-Хониуэлл Буль0187" по заказу МО США как единый язык для встраиваемых и бортовых систем в 1973-1983 гг. Назван в честь математика и писательницы графини Августы Ады Лавлейс (Augusta Ada Lovelace, 1811-1852), дочери поэта Дж. Г. Байрона, первой женщины-программиста, написавшей в 1830 г. вместе с Чарльзом Беббиджем ряд программ для его Аналитической Машины и выполнившей перевод описания этой машины на французский язык. Последний стандарт — Ada-95. В GCC язык Ada поддерживается начиная с версии 3.1. Для компиляции программ на языке Ada-95 в Сборный Компилятор GNU был интегрирован компилятор *GNAT* оригинальной разработки компании "Ada Core Technologies", бесплатно переданный проекту GCC в октябре 2001 года.

adress — См. *абсолютный адрес* и *относительный адрес*.

aggregate — См. *агрегатный тип*.

alias — См. *псевдоним*.

ANSI (American National Standards Institute) — Организация, которая администрирует и координирует издание и соблюдение нормативных документов национальных стандартов США.

API (Application Programming Interface) — Интерфейс прикладного программирования. Набор стандартных программных прерываний, вызовов процедур (методов) и форматов данных, которые должны использовать прикладные программы для запроса и получения обслуживания от операционной системы. В настоящей книге больше употребляется в связи с переносом программного обеспечения на системы MS Windows.

archive — См. *библиотека*.

assembler — См. *ассемблер*.

Autoconf — Утилита настройки конфигурации и упаковки программного обеспечения с открытым исходным кодом для его выпуска и распространения.

back end — См. *нижний уровень*.

BFD (Binary File Descriptor) — Библиотека, содержащая подпрограммы для выполнения низкоуровневых действий с двоичными файлами различных форматов.

binutils — Пакет, включающий в себя основной набор программных утилит, которые используются GCC. Специально разработан для непосредственной работы с компилятором GCC.

BSD (Berkeley Software/Standard Distribution) — Операционная система семейства UNIX. На ней основываются некоторые современные системы UNIX. См. также *SVR4*.

bss — Имя сегмента (раздела) вырабатываемого компоновщиком системы UNIX исполняемого файла. Этот сегмент программы содержит неинициализированные данные. До загрузки программы в память переменная bss содержит только имя, размер и положение. Область памяти выделяется этому разделу при загрузке программы на выполнение. См. также *text* и *data*.

bytecode — См. *байт-код*.

C — Язык программирования высокого уровня. Разработан Деннисом Ричи (Dennis Ritchie) в начале 70-х гг. в "AT&T Bell Laboratories" специально для операционной системы Unix.

C++ — Объектно-ориентированный язык программирования высокого уровня. Разработан Бьерном Страуструпом (Bjarne Stroustrup) в "AT&T Bell Laboratories" в 1983 г. Язык C++ был создан для объединения возможностей языка C с поддержкой объектно-ориентированного программирования.

C89 — Стандарт ANSI языка программирования C 1989 года.

C99 — Стандарт ANSI языка программирования C 1999 года.

cast — См. *приведение типа*.

calling convention — См. *последовательность вызова*.

calling sequence — См. *последовательность вызова*.

CCP (Conditional Code Propagation) — Способ оптимизации, определяющий переменные, значения которых не изменяются при всех возможных путях выполнения программы и использующий эти сведения для исключения неиспользуемых участков кода.

cfront — Первоначальная версия компилятора C++ была реализована в корпорации "AT&T" в программе с именем cfront, которая транслировала исходный текст программ на языке C++ в исходный код на языке C.

Chill (CCITT High Level Language) — Язык программирования высокого уровня. Разработан в 1970-х годах под эгидой CCITT для программирования систем телекоммуникации реального времени. Последний стандарт выпущен в 1996 году. Начиная с версии 3.1 поддержка в GCC этого языка прекратилась.

class — См. *класс*.

CNI (Cygnus Native Interface) — Набор средств для описания *системно-ориентированных* методов Java-программ на языке C++. См. также *JNI*.

code (код) — Устоявшийся английский термин, обозначающий любую форму списка последовательных инструкций, представляющих реализацию алгоритма компьютерной программы. Широкое понятие, включающее в себя от понятных человеку исходных текстов на языке высокого уровня до последовательностей машинных операционных кодов — двоичных инструкций, передаваемых процессору.

coersion — Внутреннее автоматическое *конвертирование* одного основного типа данных в другой без использования стандартного синтаксиса *приведения* типа или вызова специальной функции. Термин указывает на ситуацию, когда исходный код может неоднозначно переноситься на другие компиляторы и платформы.

COFF (Common Object File Format) — Стандартный формат объектных файлов, которые могут сохранять переносимость между различными системами. Распознается многими ассемблерами и компоновщиками. Может содержать отладочную информацию, распознаваемую различными программами-отладчиками.

COMDAT (COMmon Data) — Данные или выполнимый элемент программы (набор выполнимых элементов), которые могут повторяться в нескольких используемых объектных файлах. Программа-компоновщик (linker) должна находить и исключать лишние копии таких блоков. В документации может обозначаться терминами *folding* и *comdat folding*.

common — Атрибут глобальной переменной, предназначенный для размещения в блоке *common block*.

common block — Компоновщик GNU создает блок данных общего использования как область памяти для размещения глобальных переменных. В случаях, когда в разных объектных модулях объявляются идентичные глобальные переменные, все ссылки на них разрешаются через одну переменную блока common.

compilation unit — См. *компиляционный модуль*.

compiler — См. *компилятор*.

Configure, сценарий конфигурации установки программного обеспечения с открытым исходным кодом. Входит в состав предоставляемого пользователям пакета.

copyleft — В противоположность знаку охраны авторского права copyright (c) эта отметка сообщающая о том, что программа свободна для использования и распространяется на условиях общественной лицензии (например, *GPL GNU*). Это значит, что программа распространяется свободно и все ее модифицированные и расширенные версии также должны подлежать свободному распространению.

CPP (C Preprocessor) — Основной препроцессор GCC. Программа, которая считывает исходный код, находит в нем и обрабатывает *директивы*, вырабатывая при этом на выходе определенным образом модифицированную версию исходного кода.

CPU — Центральный Процессорный Блок вычислительной машины. Часть компьютера, непосредственно исполняющая машинные команды программ. В общем случае состоит из арифметико-логического устройства, блока регистров, устройства управления памятью и др. функциональных устройств центрального блока.

cross compile — См. *кросс-компилятор*.

cruff — Неиспользуемый код, который остается в составе программы после нескольких циклов устранения ошибок или после модернизации программы. Устранение таких участков кода может быть затруднительным из-за сложности их однозначной идентификации.

CSE (Common Subexpression Elimination) — Устранение общих подвыражений. Способ оптимизации программы, при котором выполняется поиск повторяющихся выражений. Лишние их копии удаляются там, где возможно использование уже однажды найденного результата выражения вместо его повторного вычисления.

ctor — Конструктор. Распространенное сокращение от (ConstrucTOR). См. *конструктор*. См. также *dtor*.

CVS (Concurrent Version System) — Система Конкурирующих Версий позволяет управлять комплектностью разрабатываемых версий программ. CVS отслеживает и регистрирует изменения в исходных текстах программы. Система создана для обеспечения участия в разработке программного обеспечения многих добровольцев со всех уголков света.

Cygwin — Коммерческий проект условно-бесплатного решения для портирования программного обеспечения UNIX на системы *Microsoft Windows*.

data — Сегмент (раздел) вырабатываемого компоновщиком UNIX выполнимого файла. В сегменте с именем data содержатся данные с начальными значениями. Раздел содержит элементы данных, каждый из которых имеет имя, размер

и отведенный участок памяти, в котором содержится его начальное значение. См. также *bss* и *text*.

DBX — Программа интерактивной отладки (debugger), которая позволяет построчно трассировать выполнение программы. Изначально DBX существовал как отладчик для режима командной строки, однако в различных вариантах своего воплощения он имеет интерфейс X Window GUI и emacs-интерфейс.

DCE (Dead Code Elimination) — Метод оптимизации программ, который удаляет любые участки кода, неиспользуемого при выполнении программы. См. также *dead code*.

dead code — Во время оптимизации возможно определение участков программы, которые никогда не используются при любых возможных вариантах ее выполнения. Именно эти участки называются "мертвым кодом". Удаление "мертвого кода" делает программу эффективнее, сокращая ее размер, объем используемой памяти, время загрузки и выполнения.

demangle — Процесс извлечения описательной информации, закодированной в замененных (mangled) именах функций, которые содержатся в файлах объектного формата. См. также *mangle* и *деманглер*.

demangler — См. *деманглер*.

deprecated — Нерекомендуемое для использования свойство или опция программы. Опция или свойство, которые потеряли свое значение и поддержка их прекращена. Они еще распознаются и работают, но вероятно, что уже в следующей версии программы их уже не будет. Поэтому их использование не рекомендуется.

dereference — Разъименование указателя. Вычисление выражения может изменить значение адреса, хранящегося в указателе. В таком случае говорят о разъименовании указателя

directive — См. *директива*.

distention — Термин документации GCC по поддержке языка Fortran. Применяется по отношению к специфическим расширениям отдельных диалектов Fortran, именам встроенных функций, которые в более современных реализациях не используются и называются также "ugly", т.е. "странными", "мерзкими"... Некоторые из "distensions" поддерживаются компилятором g77 через использование флагов -fugly-*.

dtor — Деструктор. Распространенное сокращение от (DestrucTOR). См. *деструктор*. См. также *ctor*.

DWARF (Debugging With AttRIBUTE Format) — Формат размещения в объектном коде информации, используемой при отладке программ.

DWARF2 (Debugging With AttRIBUTE Format 2) — Наиболее современная выпущенная версия формата *DWARF*.

dynamic library — См. *библиотека*.

EABI (Embedded Application Binary Interface) — Двоичный интерфейс системного уровня приложений встроенных систем.

ECOFF (Extended Common Object File Format) — Расширенный стандарт *COFF*. Стандартный формат объектных файлов, портируемых между различными системами. Поддерживается многими современными ассемблерами и компоновщиками. То же, что *XCOFF*.

EH — Сокращение от "exception handling". См. *исключения*.

ELF (Executable Linux Format, Executable and Linkable Format) — Формат объектных файлов системы Linux, которые вместе с исполняемым кодом содержат информацию, необходимую для загрузки разделяемых библиотек. Этот объектный формат происходит от формата COFF и имеет много общих с ним свойств.

embedded system — См. *встроенная система*.

entry point — См. *точка входа в программу*.

Escape-последовательность — В языке C применяется метод экранирования символов, которые могут быть неправильно интерпретированы компилятором. Это так называемые escape-коды. Например, символ конца строки может обозначаться в строковых литералах как "/n", а табуляции — "/t".

exception handling — См. *обработка исключений*.

f2c — Транслятор исходного кода на языке *Fortran* в исходный код на языке C. Стандарт для Fortran-кода, предназначенного для обработки этим транслятором. Стандарт f2c имеет в GCC соответствующие опции поддержки.

folding — См. *COMDAT*.

Fortran — FORTRAN. Процедурный язык высокого уровня для реализации численных методов. Разработан в 1954–1957 гг. Джоном Бэкусом (John W. Backus) и его коллегами в компании "IBM". Название происходит от сокращения "FORmula TRANslation". Последний стандарт — Fortran-95. В GCC также поддерживаются стандарты RATFOR, ANSI Fortran-66, f2c, Fortran-77 (ANSI X3.9). Программы на языке фортран могут иметь *фиксированный* (*традиционный*) или *свободный* формат. В GCC имеется поддержка всех основных стандартов языка Fortran и многих из его известных расширений.

FPU (Floating Point Unit) — Аппаратный процессорный блок, который действует совместно с Центральным Процессорным Блоком (*CPU*) при выполнении компьютером операций с числовыми данными в формате представления с плавающей точкой.

frame, stack frame — См. *кадр стека*.

front end — См. *верхний уровень*.

FTP (File Transfer Protocol) — Клиент-серверный протокол из набора протоколов IP, обеспечивающий поиск и передачу файлов между системами по транспортному протоколу TCP.

function — См. *функция*.

garbage collection — Дословно "сборка мусора". Процесс динамического распределения памяти между выполняемыми в системе процессами.

GCSE (Global Common Subexpression Elimination) — Метод оптимизации программ, который распознает повторяющиеся выражения и использует уже полученный результат для исключения их лишних экземпляров.

GNAT (GNU Ada Translator) — Верхний уровень GCC для компиляции программ на языке *Ada*. Передан проекту GCC компанией "Ada Core Technologies" в 2001 г.

GNATS (GNU Bug Tracking System) — Доступная для пользователей система мониторинга ошибок GCC и другого программного обеспечения GNU.

GNU — Действующий с 1984 года открытый добровольческий проект по созданию свободно распространяемых и доступных общественности программ с открытым исходным кодом.

GUI (Graphic User's Interface) — Графический интерфейс пользователя.

GOT (Global Offset Table) — Таблица в объектном файле, которая содержит список смещений (*относительных адресов*), обеспечивающий динамическую загрузку и перемещение выполняемого кода в памяти машины.

GPL GNU (General Public License GNU) — Общественная лицензия GNU. Лицензия, которая применяется к программному обеспечению условия свободного распространения. См. также *copyleft*.

header, header file — См. *заголовочный файл*.

Hollerith field — В программах на языке *Fortran* заключенная в кавычки строка буквенных знаков, которая в начале содержит число, указывающее количество знаков в строке, и затем — саму текстовую строку. В начале текстовой строки стоит символ 'H'. Например, для передачи текстовой строки "Philips66" может быть использовано поле формата Hollerith со значением "10HPhilips66".

host — См. *целевая платформа*. Обычно сочетание операционной системы с аппаратным обеспечением, то сочетание, на котором предназначается к выполнению вырабатываемая программа.

HTML (HyperText Markup Langage) — Язык гипертекстовой разметки. Поддерживаемый протоколом HTTP из набора интернет-протоколов IP международный формат представления документов.

i18n — Сокращение от "internationalization". См. *интернационализация*.

if-conversion — Процедура оптимизации, которая обеспечивает наибольшую эффективность выполнения для наиболее вероятного пути ветвлений алгоритма программы.

include guard — Способы предотвращения ситуаций повторного включения препроцессором содержимого заголовочных файлов. Обычно применяется проверка инициализации специальной переменной окружения, уникально определя-

емой в коде заголовочного файла. Основная часть кода файла помещается в блок условной компиляции, который не включается в выходной код, если переменная уже определена.

inline — Расширение имени подстановкой кода. В частности, подстановка вызова функции целым блоком кода определения значения функции.

insn — Оператор языка машинного уровня или, чаще всего, инструкция (узел) кода на промежуточном языке *RTL*.

instantiation — Создание экземпляра *объекта* на основе определения *класса*.

intrinsic — Встроенная функция *Fortran*, которая используется программами как часть языка. В GCC поддерживается использование целого ряда наборов встроенных функций для различных стандартов языка.

invariant expression — Так называется выражение или участок кода внутри цикла, связанные в вычислением одного и того же (инвариантного) значения при каждом проходе. Инвариантные выражения при оптимизации могут быть вынесены из цикла.

ISO (International Organization of Standardization) — Международный комитет стандартизации, действующий с 1946-го года. В него входят национальные организации по выпуску стандартов из 75 стран мира (включая также и *ANSI*).

jar — Архивный файл формата, подобного zip, который содержит выполнимые классы для интерпретатора *JVM*. Java-программы могут выполняться непосредственно из архивов *.jar. Такой архив также содержит стандартный файл, содержащий список имен классов, — *манифест*.

Java — Интерпретируемый объектно-ориентированный язык программирования компании Sun Microsystems. Разработан в период 1990–1995 гг. группой под управлением Джеймса Гослинга (James Gosling) для создания небольших платформо-независимых Интернет-приложений, распространяемых на клиентские машины с web-сервера. Java-программы в виде классов в особом объектном формате предназначены для выполнения в среде *Виртуальной Машины Java*, *JVM*. В GCC возможна компиляция программ на языке Java не только в байт-код, выполнимый интерпретатором *JVM*, но и в объектный код машинного формата. Возможно также использование методов библиотек Java в программах на других языках.

JNI (Java Native Interface) — Стандартный интерфейс программирования для реализации *системно-ориентированных* методов Java-программ и встраивания (интеграции) *JVM* в системно-ориентированные приложения. См. также *CNI*.

intrinsic — См. *встроенная функция*.

JVM (Java Virtual Machine) — См. *Виртуальная Машина Java*.

K&R C — Первоначальный традиционный стандарт языка программирования C, описанный Д. Ритчи и Б. Керниганом в книге "Язык программирования C" ("The C Programming Language" B. Kernighan, D. Ritchie. Prentice Hall, 1978, 1988).

l10n (localization) — См. локализация.

lexical analysis — См. лексический анализ.

Lex-сканер — См. лексический анализ.

LGPL (Lesser General Public License) — "Сокращенный вариант" *GPL* GNU. Лицензия, используемая для многих, хотя и не всех, библиотек проекта GNU. Условия этой лицензии разрешают использование подпрограмм из состава библиотек для производства программного обеспечения, на которое действуют исключительные права собственности. *GPL* таких условий не допускает.

library — См. библиотека.

life analysis — Понятие связано с оптимизацией программ. Это — процесс, который определяет значения, которые должны дольше других храниться в регистрах. Он также находит возможности освобождать регистры от более не используемых величин.

linker, link editor — См. компоновщик.

localization (l10n) — См. локализация.

lvalue — Значение левой части оператора присваивания. В общем случае — любого рода выражение, результатом вычисления которого должен быть адрес памяти. См. также *rvalue*.

macro — См макрос.

make — Утилита сборки программного пакета. Используется для компиляции и компоновки программ с учетом выполнения различных условий (проверки даты и времени, определения актуальности уже скомпилированных объектных файлов, использования зависимостей компоновки). Действует на основании правил зависимостей и других условий, определяемых в особых компоновочных сценариях — *make-файлах*.

makefile, Makefile — См. *make-файл*.

make-файл — Сценарий выработки программы или программного пакета, выполняемый утилитой *make*. Обычно называется *makefile* или *Makefile*. Входит в состав пакета программного обеспечения, распространяемого с открытым исходным кодом. Содержит определения целей и связанные с целями зависимости компоновки. Строки зависимостей определяют приоритет и порядок компоновки частей программы. В случае обнаружения несоответствия объектных файлов правилам выработки программы выполняется повторная компиляция необходимых для этого исходных файлов.

mangle — Дословно "искажать". Процесс замены имен функций и методов для их представления в таблицах символических имен объектных файлов. Компиляторы *C++* и *Java* модифицируют имена функций и методов для их уникальной идентификации в соответствии с принадлежностью классу, количеством и типами аргументов. Это обеспечивает возможность *перегрузки* имен функций (ме-

тодов) и поддерживает выполнение условий полиформизма объектно-ориентированных программ. См. также *demangle* и *деманглер*.

manifest — См. *jar*.

marshaling — См. *маршалинг*.

MFC (Microsoft Foundation Class) — Иерархия классов оболочки интерфейса *API* систем *Microsoft Windows*.

Microsoft Windows, MS Win32 — Общее название семейства систем, поставляемых компанией "Microsoft" для персональных компьютеров.

NaN (Not a Number) — Термин стандарта IEEE для любого случая недопустимых действий над числами с плавающей точкой.

NEXTSTEP — Операционная среда окружения, предоставляющая графический (GUI) интерфейс пользователя. Может использоваться на компьютерах HP, NeXT, Sun и других.

NLM (NetWare Loadable Module) — Исполняемые программы для систем NetWare.

NLS (National Language Support) — Способность компилятора GCC выводить диагностические сообщения на других национальных языках, кроме основного — американского диалекта английского языка.

noop — Инструкция языка ассемблера, которая не выполняет никакого действия. Код этой "пустой" инструкции часто используется в исполняемых программах в качестве заполняющего байта.

object — См. *объект*.

Objective-C — Язык программирования. Появился в результате объединения языка C с поддержкой объектов языка Smalltalk. Реализация *Objective-C* в GCC представляет собой расширение языка C добавлением в него синтаксиса поддержки объектов, сходного с применяемым в Smalltalk.

opcode — См. *откод*.

ordered comparison — Упорядоченное сравнение двух чисел с плавающей точкой. Вызывает исключение в случае, когда одно из сравниваемых чисел имеет значение "*NaN*". См. также *unordered comparison*.

overload — См. *перегрузка*.

package — В языке Java — набор классов. В Ada — набор процедур.

parser — Процедура *синтаксического разбора*.

pass — Проход (по коду). Каждый последовательный этап считывания компилируемого кода, связанный с его той или иной обработкой, будь то предобработка, синтаксический разбор, генерация кода и т.д. Проход может как модифицировать входной код для следующих этапов обработки программы, или только генерировать таблицы и другого рода информацию.

peephole optimization — Метод оптимизации программ, при котором неэффективные последовательности инструкций заменяются более усовершенствованными операциями.

PIC (Position Independent Code) — Независимый от положения загрузки код. Объектный код, пригодный для размещения в разделяемой библиотеке. Вся внутренняя адресация перемещаемых объектных модулей разрешается через относительные адреса или ссылки на таблицу глобальных символов.

platform — См. *платформа*.

PMF (Point to Member Function) — Особый тип языка C++, способный содержать адреса функций — членов класса определенного объекта.

POSIX — Стандартная спецификация UNIX, возникшая в результате объединения стандарта IEEE со спецификацией "Open Group's Single UNIX".

postmortem — "Посмертный анализ" аварийно завершившейся программы в отладчике.

pragma — Специфическая команда компилятору, помещаемая в исходный код, распознается только предназначенным компилятором и игнорируется остальными компиляторами.

preprocessor — См. *препроцессор*.

pseudo-op — Псевдо-опкод. Директивы *ассемблеру* в коде на языке ассемблера. См. также *директива*.

Ratfor (Rational Fortran) — Доступный для общественного использования препроцессор исходного кода, который допускает использование в программах на языке Fortran синтаксиса, подобного языку C, и конвертирует их в Fortran-код *стандартного формата*.

relative adress — См. *относительный адрес*.

relocatable adress — Перемещаемый адрес. См. *относительный адрес*.

RM (Reference Manual) — Сокращение, применяемое в области программирования на языке Ada. Является ссылкой на документ, содержащий определение стандарта Ada-95.

RMI (Remote Method Invocation) — Набор средств поддержки вызова удаленных методов для реализации выполнения Java-программ в распределенной системе (*distributed system*) взаимодействующих между собой *Виртуальных Машин Java* (JVM).

RTL (Register Transfer Language) — Язык Регистрового Переноса. Портируемый псевдо-ассемблерный язык гипотетической обобщенной машины. Внутренний промежуточный язык, код на котором генерируется верхним уровнем GCC из входного исходного текста программы. Затем нижний уровень компилятора GCC транслирует RTL-код на язык ассемблера целевой машины. В форме RTL программа проходит основные этапы оптимизации и другие действия общей обработки.

RTTI (RunTime Type Identification) — В объектно-ориентированном программировании существует возможность маскировки одного объекта другим. Средства RTTI позволяют определять действительный тип объекта во время выполнения программы.

rvalue — Значение правой части оператора присваивания. В общем случае — любого рода выражение, результатом вычисления которого должна быть величина.

scheduler — Планировщик инструкций. Если машина способна выполнять несколько инструкций одновременно, то линейная последовательность инструкций может быть переупорядочена (распланирована) для упреждающего выполнения нескольких более быстрых инструкций в то время, пока выполняется одна медленная.

scope — "Область видимости". Структура или блок кода, в пределах которой возможно использование переменной.

shared library — См. *библиотека*.

Single UNIX Specification — См. *POSIX*.

slot scheduler — См. *scheduler*.

Smalltalk — Объектно-ориентированный язык программирования, разработанный компанией "Xerox PARC" в 1970-х. Это был язык программирования интерфейсной системы SIMULA, представлявшей собой среду взаимодействия окон и манипулятора (мыши).

spec file — Файл, который содержит набор правил, определяющих назначение параметров, передаваемых gcc вызываемым подпроцессам.

SSA (Static Single Assignment) — Особый формат представления логического потока выполнения программы в блоке кода. Построение графа в формате SSA используется некоторыми способами оптимизации, например оптимизацией исключения неиспользуемого кода (dead code elimination).

STABS (Symbol TABle directiveS) — Операторы и данные, помещаемые в код на языке ассемблера для представления отладочной информации. Ассемблер и компоновщик помещают эти данные в объектный код и исполняемые файлы в виде таблиц для обеспечения символьной отладки.

stack frame, frame — См. *кадр стека*.

static library — См. *библиотека*.

stderr (Standard Error) — Стандартное устройство сообщений об ошибках.

stdin (Standard Input) — Стандартное устройство ввода.

stdout (Standard Output) — Стандартное устройство вывода.

strip — Команда strip применяется к исполняемым файлам для удаления из них всей отладочной информации.

struct — См. *структурата*.

SVR4 (System Five Release Four) — Выпущенная компанией "AT&T" версия UNIX.

Легла в основу многих современных версий UNIX. См. также BSD.

table (virtual function table) — См. *таблица виртуальных функций*.

target — Целевая платформа. См. *платформа*.

text — Содержащий инструкции раздел (сегмент) выполнимого кода, вырабатываемого компоновщиком UNIX.

token — См. *лексема*.

translation unit — См. *компиляционный модуль*.

translator — См. *транслятор*.

ugly — Забавный термин, обычно применяется в связи с компиляцией древних программ на языке Fortran. См. *distortion*.

union — См. *объединение*.

unordered comparison — Неупорядоченное сравнение двух чисел с плавающей точкой. В случае, когда одно из сравниваемых чисел (или оба) имеют значение "*NaN*" не вызывает исключение. См. также *ordered comparison*.

variadic macro — См. *вариативный макрос*. Макрос, который может иметь переменное количество аргументов. В GCC поддерживается возможность передачи макросу переменного количества аргументов через их сохранение в переменной окружения с именем `__VA_ARGS__`.

void — В языках *C*, *C++*, *Objective-C*, *Java* — тип функций, не имеющих возвращаемого значения. Применяется как указатель на экземпляр любого типа.

volatile — Изменяемый тип данных. Отмечает расположение памяти, которое может быть модифицировано любой подпрограммой. Использование в программах этого типа позволяет лучше использовать регистры и избежать лишних перемещений данных из памяти в регистры и обратно.

vtable, (virtual function table) — Таблица виртуальных функций. В технологии объектно-ориентированного программирования объекты поддерживают внутренние таблицы, которые содержат адреса функций.

VXT — Диалект языка *Fortran*, подобный VAX Fortran, включающий в себя некоторые свойства стандарта Fortran-90.

Windows — См. *Microsoft Windows*.

word — Размер машинного слова архитектуры процессора. Определяет основной тип целых чисел, обрабатываемых машиной.

XCOFF (Extended Common Object File Format) — Расширенный стандарт *COFF*. Стандартный формат объектных файлов, портируемых между различными системами. Поддерживается многими современными *ассемблерами* и *компоновщиками*. То же, что *ECOFF*.

Русские

абсолютный адрес — числовое значение, точно и однозначно определяющее расположение байта в памяти машины. См. также *относительный адрес*.

агрегатный тип — Тип данных, состоящий из набора полей одного или более основных типов. Например, общий для многих языков тип "массив" (array), или "структура" (struct) в языке C.

алиас — См. *псевдоним*.

архив (статическая библиотека) — См. *библиотека*.

ассемблер — Специфичная к платформе программа,читывающая исходный код в виде последовательности ассемблерных инструкций (мнемоническое представление кодов машинных операций) и транслирует его в двоичный объектный код машинного формата, который затем может быть передан *компоновщику*.

баг (bug) — Устоявшийся жargonный термин, означающий ошибку в коде программы. Программистам тоже бывает нелегко признавать свои ошибки, и тогда они называют их "жукаами", "мухами"... Термин получил широкое распространение благодаря одному древнему и всем известному анекдоту. Отсюда же происходит устоявшееся название программ-отладчиков — "дебагер" (debugger).

байт-код Java — Специфический формат переносимого объектного кода, вырабатываемого при компиляции программ на языке Java. Байт-код загружается и выполняется интерпретатором JVM (Виртуальной Машиной Java), что является основным способом выполнения Java-программ.

библиотека — Библиотекой (library) называется отдельный файл, содержащий один или более объектных файлов, которые могут быть использованы при компоновке выполнимых программ. Статическая библиотека (static library) содержит модули, включаемые *компонентовщиком* непосредственно в исполнимую программу. Она также имеет другое название — архив. Разделяемая библиотека (shared library) содержит набор объектных модулей, которые временно загружаются в память и присоединяются к программе во время ее выполнения. Она также называется динамической библиотекой (dynamic library). Стандартными библиотеками называются библиотеки любого типа, поставляемые в комплекте с операционной системой.

биндер — Программа создания подшивки пакета (набора процедур) Ada, gnatbind.

вариативный макрос — Макрос, которому можно передавать переменное количество аргументов. В GCC поддерживается возможность передачи макросу переменного количества аргументов через их сохранение в переменной окружения с именем __VA_ARGS__.

верхний уровень компилятора (front end) — Верхний уровень GCC преобразовывает входной код на языке высокого уровня в промежуточную структуру, пере-

даваемую ориентированному на целевую платформу *нижнему уровню* (порту) компилятора.

Виртуальная Машина Java (Java Virtual Machine, JVM) — Программа, способная считывать объектные файлы стандартного формата *байт-кода* Java и выполнять находящиеся в таком файле инструкции. Переносимость Java-программ решается за счет эффективного портирования Виртуальной Машины Java и поддержки этой программой стандартного набора классов. Является основной средой выполнения программ на языке *Java*.

встроенная система — (Embedded system.) Встроенной (или встраиваемой) системой называется компьютерная система, которая работает совместно с другим оборудованием и размещается в одном конструктивном блоке с этим оборудованием. Это может быть встроенная среда выполнения управляющих задач автоматического оборудования или система для микроконтроллера. Это название также употребляется по отношению к ограниченной изолированной среде выполнения задач для портирования приложений.

встроенная функция — Стандартная функция языка программирования высокого уровня, действующая как часть языка. В технологии программирования на языке Fortran по отношению к таким функциям применяется определяющий термин "intrinsics".

выравнивание адреса — Перемещение элемента данных, машинной команды или блока кода для установки его начального адреса кратным показателю выравнивания. Это может быть критичным для времени обработки и даже для правильности считывания процессором данных и команд из памяти. Если адрес является числом, кратным 16 (т.е. деление этого числа на 16 не дает остатка), то мы говорим, что этот адрес имеет выравнивание по границе 16-бит, или 16-битное выравнивание.

генерирование кода — В общем случае программная выработка исходного кода по заданным характеристикам или конфигурации. В качестве примера программы-генератора кода можно назвать стандартную утилиту lex систем UNIX и ее аналог проекта GNU flex. Термин также применяется по отношению к выработке компилятором выходного кода на ассемблерном или промежуточном языке.

деманглер (demangler) — Программа, выполняющая преобразование замененных имен функций и методов в их полную форму представления, соответствующую именам в исходном коде. Полные имена включают список типов аргументов. Замена имен, помещаемых в таблицах символов объектных файлов, (mangling) практикуется для обеспечения поддержки замещения (*перегрузки*) функций и методов по типам обрабатываемых аргументов. С компилятором GCC для этого используются утилиты `c++filt` и `nm`.

деструктор — Специальный метод, обеспечивающий разрушение объекта.

динамическая библиотека — См. *библиотека*.

директива — 1) Команда в исходном коде, которой предшествует символ "решетка" ('#, hash) в первой позиции строки. Такие команды исполняются (и, затем, после их обработки, убираются из исходного кода) *препроцессором*. 2) В коде на языке ассемблера *директивой* называется команда-инструкция, обрабатываемая ассемблером. Директивы ассемблера также называются псевдо-опкодами (*pseudo-opcode*).

зависимость — См. *make-файл*.

заголовочный файл — Файл исходного кода, помещаемого препроцессором в тело других программ. Препроцессор *CPP* использует для этого директиву *include* с именем включаемого файла. В языках *C*, *C++* и *Objective-C* существует традиция давать заголовочным файлам имена с суффиксом ".h".

замена имен символов в объектном коде — См. *mangling*.

интернационализация (i18n) — Обеспечение возможности программы или набора программ, составляющих отдельный пакет, поддерживать различные национальные языки, в частности для выдачи сообщений интерактивного пользовательского интерфеса.

исключения — См. *обработка исключений*.

кадр стека — Область *стека*, в которой во время выполнения текущей функции содержатся локальные переменные, передаваемые этой функции в качестве аргументов, а также хранятся значения регистров, восстанавливаемых при возврате. Формат кадра стека зависит от типа и версии процессора и от соглашений о вызове функций, применяемых в системе.

класс — 1) В объектно-ориентированном программировании классом называется определительное описание типа и структуры объекта. Вырабатываемые на основе определения класса экземпляры объектов имеют одинаковый интерфейс и поведение. 2) В *Java*-файле скомпилированного исходного кода класса, содержащий *байт-код* интерпретатора *JVM*, также называется классом.

код (code) — Термин, обозначающий любую форму списка последовательных инструкций, представляющих алгоритм компьютерной программы. Широкое понятие, включающее в себя от понятных человеку исходных текстов на языке высокого уровня до последовательностей машинных операционных кодов — двоичных инструкций, передаваемых процессору.

комментарий — Стока программы, содержащая пояснения исходного кода для лучшей читаемости и понимания программы. Комментарии убираются из исходных текстов при их обработке препроцессором.

конструктор — Специальный метод, обеспечивающий инициализацию объекта.

компилятор (compiler) — Набор программного обеспечения, которое считывает исходный текст компьютерной программы, и транслирует (переводит) содержащиеся в нем инструкции в выполнимый машиной формат. В современной документации на английском языке термины *compiler* и *translator* могут применяться как синонимы.

компиляционный модуль — Отдельный модуль исходного текста программы, который компилируется в отдельный объектный файл. Чаще всего — это отдельный исходный файл. Иногда он включает содержимое других файлов (в языке C — с помощью *директивы препроцессора #include*). В этом случае такой набор файлов составляет один компиляционный модуль.

компоновщик (linker) — Компоновщик, редактор компоновочных связей. Утилита, которая объединяет объектные файлы (часть из которых может находиться в составе библиотек) и разрешает находящиеся в них обращения к внешним объектам. В результате компоновки вырабатывается готовая к запуску программа. Компоновщик входит в комплект поставки аппаратного обеспечения. Утилита-компонентовщик проекта GNU называется *ld*, она способна поддерживать большой набор целевых платформ.

конвертирование типов (coersion) — Внутреннее преобразование одного основного типа данных в другой без использования стандартных способов *приведения* типа или вызова специальной функции.

кросс-компилятор — Специальная конфигурация компилятора для *перекрестной компиляции* — выработки выполнимых файлов, предназначенных для совершенно другой платформы.

лексема — См. *лексический анализ*.

лексический анализ (lexical analysis, lexical scan). — Начальный этап компиляции исходного кода программы. На этом этапе программа считывается и разделяется на элементы языковых конструкций процессом лексического сканирования (лексическим сканером). Считанные символы собираются в цепочки, из которых выделяются элементы исходного кода программы — *лексемы* (tokens). Выделенные последовательности сортируются по категориям и передаются дальше *синтаксическому разделителю*. На этом этапе принимается решение о возможности дальнейшей обработки кода.

лексический сканер — См. *лексический анализ*.

локализация (l10n) — Поддержка программой местных (национальных) форматов мер, времени, денежных расчетов и т.д..

макроопределение — Директива препроцессора для назначения макроса (*#define*). Содержит имя макроса и присваиваемое ему значение. См. *макрос*.

макрос (macro) — Используемая препроцессором переменная среды окружения. Объявляется с соответствующим ему значением использованием в программе директивы *#define*. Может затем использоваться в программе для подстановки текста и в проверках условий управляющих директив.

манифест — Файл описания архива классов Java. См. *jar*.

маршалинг — Преобразование Маршалла. Последовательное упорядочивание аргументов вызова удаленного метода в распределенной системе.

массив — Агрегатный тип, состоящий из набора адресуемых индексами однотипных переменных.

нижний уровень компилятора (*back end*) — Нижний уровень GCC компилирует промежуточную структуру, передаваемую ему *верхним уровнем* компилятора в код объектного формата предназначаемой (целевой) машины. Также называется *портом* GCC, потому что обеспечивает перенос компилируемых в GCC программ на поддерживающие платформы.

обработка исключений (exception handling, EH) — Обработка исключительных ситуаций (исключений, exceptions), чаще всего — ошибок выполнения программы. Часть кода, которая автоматически вызывается для обработки таких ситуаций, называется обработчиком исключений (exception handler).

объединение (union) — Агрегатный тип языков программирования C и C++.

объект — В объектно-ориентированном программировании это понятие обозначает совокупность данных и связанных с ними действий. Объекты взаимодействуют через посылаемые друг другу сообщения.

объектный код — Результат компиляционного процесса. Выполнимый машиной, то есть "объективный" код. Для построения из модулей объектного кода загружаемых на выполнение файлов и для разрешения внешних обращений, имеющихся в объектном коде, требуется их обработка *компоновщиком*.

опкод (opcode) — Операционный код. Определительная часть машинной инструкции.

отладчик (debugger) — Программа для поиска ошибок и изучения хода выполнения разрабатываемых программ. В проекте GNU используется отладчик gdb.

относительный адрес — числовое значение, представляющее смещение от известного расположения памяти. Этот тип адресов используется в перемещаемых модулях *разделяемых библиотек*. Адрес того или иного объекта при выполнении программы разрешается относительно точки загрузки модуля в память машины. См. также *абсолютный адрес*.

парсер — Процедура *синтаксического разбора*.

патч (patch) — Широко известный жargonный термин, обозначающий набор исправлений к программе, обычно предоставляемый разработчиком.

перегрузка имен функций и методов (overload) — Замещение имен функций (методов) в соответствии с типами передаваемых параметров. Это позволяет использовать различные реализации кода функции при вызове ее по имени с различными наборами аргументов. См. также *tangle*, *demangle* и *деманглер*.

перекрестная компиляция — См. *кросс-компилятор*.

платформа — Сочетание определенного компьютерного процессора с запущенной на нем операционной системой. GCC позволяет, используя специально сконфигурированную версию компилятора, на одной "домашней" (т.е. базовой) системе производить выполнимые программы, предназначенные для запуска на другой системе (целевой платформе). См. также *кросс-компилятор*.

порты GCC — Поддерживаемые в GCC целевые платформы. См. *нижний уровень*.

портирование — Перенос программного обеспечения на другие *платформы*.

последовательность вызова (calling sequence) — Последовательность операторов на языке ассемблера, используемая для вызова функций. Этот код устанавливает передаваемые аргументы, сохраняет адрес возвращения так, чтобы он был доступен из вызываемой функции, выполняет вызов и затем управляет передачей возвращаемого значения (если такое имеется). Последовательность вызова иногда называют соглашением о вызовах (calling convention).

поток (thread) — Порожденный процесс выполнения в многозадачной системе. Использует предоставленные ему ресурсы системы и имеет в ней уникальный идентификатор.

прагма — См. *pragma*.

прагма-директива — Особая *директива препроцессора*, сообщающая компилятору указанную *прагму*, то есть команду, предусмотренную интерфейсом этого компилятора. См. *pragma*.

предобработка — Обработка исходного кода программы *препроцессором*.

препроцессор — Программа, которая считывает исходный код, и обрабатывая его, отвечает на имеющиеся в коде *директивы*. Вырабатывает определенным образом модифицированную версию исходного кода.

приведение типов (cast) — Указание с помощью специальных синтаксических правил на необходимость преобразования значения переменной к другому типу.

проверочный набор GCC — Тестовые исходные коды для проверки правильности работы *компилятора*.

прототип функции (function prototype) — Объявление в начале программы имен и типов *функций* вместе со списком типов аргументов (параметров) их вызова и типом возвращаемого значения.

процессы — Набор одного или более *потоков* (threads) и ассоциированных с ними системных ресурсов.

псевдоним (alias) — Одно и то же расположение памяти может быть прямо или косвенно адресовано несколькими различными символическими именами, возможно разных типов. Такие имена и являются *псевдонимами* по отношению друг к другу.

разделяемая библиотека — См. *библиотека*.

реестр Win32 (MS Win32 Registry) — База данных конфигурации системы и пользовательских настроек среды выполнения GUI-приложений систем Microsoft Windows.

свободный формат Fortran — Более новый стандарт форматирования исходных текстов на языке *Fortran*. Отменяет строгости фиксированного (или *традиционного*) формата Fortran).

свойства системы, Java — Стандартный набор параметров, передаваемых Виртуальной Машиной Java исполняемому Java-классу при его запуске. Возможно ис-

пользование дополнительных свойств для передачи исполняемому классу параметров, которые могут обрабатываться в коде Java-программы.

синтаксический разбор — *Синтаксический разбор* (parsing). Этап компиляции программы. Лексемы исходного кода имеют некоторые отношения между собой, определяемые их взаимным относительным положением в выходном потоке лексического сканера. Процесс синтаксического разделителя, иначе называемого парсером (parser), определяет тип каждого элемента и транслирует исходный код программы в древовидную логическую структуру, которая (возможно, после оптимизации) используется для генерирования кода на языке машинного уровня. В GCC на выходе парсера создается промежуточный код на языке *RTL*.

синтаксический разделитель — См. *синтаксический разбор*.

системно-ориентированный метод Java — Реализованный на языке системного уровня (обычно C или C++) метод, использующий особенности того или иного компьютерного оборудования. См. *CNI* и *JNI*.

статическая библиотека — См. *библиотека*.

стек — Область памяти, куда записываются данные, связанные с вызовом функций. См. *кадры стека*.

структура (struct) — Агрегатный тип языков программирования C и C++.

строкура кода — Грамматически выделяемая часть исходного текста программы, которая содержит блок или элемент реализуемого алгоритма. Типичный участок исходного кода, обладающий определенными свойствами.

суффикс (расширение) имени файла — Часть имени файла (подстрока), стоящая после последнего символа точки ('.') в строке имени. Используется для указания компилятору GCC типа и формата содержимого файла и контекстного определения компилятором соответствующих команде действий с этим файлом.

таблица виртуальных функций (vtable) — В технологии объектно-ориентированного программирования объекты поддерживают внутренние таблицы, которые содержат адреса функций.

таблица символов — В объектном коде содержатся таблицы разрешения адресов программных символов. Эта информация используется компоновщиками и отладчиками.

точка входа в программу (entry point) — Адрес памяти внутри выполнимой программы, на который передается управление при запуске программы.

традиционный формат Fortran — Правила форматирования исходных текстов программ на языке *Fortran*, предназначенных для обработки компиляторами более старых выпусков. Также называется *фиксированным* форматом Fortran-программ. Этот стандарт устанавливает строгие правила форматирования текста, соблюдение которых критично для правильной компиляции программ.

транслятор (translator) — Программное обеспечение, которое преобразовывает исходный текст компьютерной программы в формат кода на другом языке про-

граммирования или на промежуточный язык. В современной английской документации часто используется как синоним понятия *компилятор* (compiler).

трассировка — Пошаговое выполнение программы в отладчике.

фиксированный формат Fortran — См. *традиционный формат*.

функция — подпрограмма, возвращающая в точку вызова результат своего выполнения, присваиваемый имени функции.

целевая платформа (target) — Обычно сочетание определенной версии операционной системы с определенным аппаратным обеспечением, то сочетание для выполнения, на котором предназначается вырабатываемая программа. См. также *платформа*.

Язык Регистрового Переноса, См. *RTL*.

Предметный указатель

A

Ada 27, 173
инсталляция 173
типы файлов 176
утилиты 183

B

binutils 29, 53, 300
установка 53

C

C (язык) 25, 77
атрибуты 88
вставка строк ассемблерного кода 281
расширения GNU 84
соглашения об именах 83
стандарты 84
типы файлов 78
C++ 26, 108
атрибуты 116
библиотеки 121
иерархия классов программы 320
классы 118
промежуточный код компиляции 320
расширения GNU 116
типы файлов 108
COMDAT 125
Chill 28
CNI 162, 202, 207
configure, сценарий конфигурации
опции 41
префиксный каталог 42
CVS 37
Cygwin 35, 54

E

Embedded systems 306

F

Fortran 26, 136
Rarfor, транслятор 142

встроенные функции 144
препроцессор 139
расширения GNU 143
свободный формат 137
типы файлов 136
традиционный (фиксированный) формат 137

G

GCC 20
время, затраченное на компиляцию 322
выпуски 38
компиляция 39
контакты 31
поддерживаемые языки 25
проверочный набор 56
установка 34, 40
FTP-файлы 36
загрузка по FTP 35
прекомпилированной версии 34
через CVS 37
экспериментальная версия 39
части компилятора 28
стадии компиляции 329
добавление языка верхнего уровня 328
поддержка встраиваемых систем 309
выход компилятора 316
опции подпроцессов 323
отладочная информация 324
файлы и каталоги 326
используемая память 318
переменные окружения 476
опции 490, 493
управления ассемблером 279
машинно-зависимые 275
типы обрабатываемых файлов 492
GNAT 27, 173
GNU 16, 20
GPL, Общественная лицензия GNU. 460

I

Inline 90, 125

J

Java 27, 152
jar-архивы 159, 161
байт-код 152, 154
виртуальная машина Java 152
свойства (system properties) 171
типы файлов 153
утилиты 160
JVM 152
JNI 163, 205

L

GPL 460
Lvalue 102

M

Mangling (замена имен в объектном коде) 122, 135
Make-файлы 266, 321
правила суффиксов 268
написание 269

N

NLS, система поддержки национальных языков 220

O

Objective-C 26, 127
типы файлов 127

P

Postmortem 258

R

Ratfor 142
RMI (Remote Method Invocation) 169
RTL, Язык Регистрового Переноса 342
инструкции 347
коды типов выражений 343
типы инструкций 344
типы operandов 346
коды режимов выражений 370
классы режимов выражений 372

T

Template 125

V

Vtable, таблица виртуальных функций 124

W

Windows, MS Win32
MinGW (компилятор) 303
Cygwin (компилятор) 304
консольные приложения 304
GUI Win32 305

X

xgettext, опции программы 223

Русские термины

A

Адресация абсолютная и относительная 280
Ассемблер 278, 310
выработка ассемблерного модуля 112, 139, 157
вставка ассемблерного кода в программы 281
директивы ассемблера GNU as 285
Атрибуты 88, 116

Б

Библиотеки 80, 121, 232
разделяемые (динамические) 82, 114, 133, 141, 159, 232, 236
статические (архивы) 80, 112, 131, 140, 158, 232, 233
размещение 236
для встраиваемых систем 311, 312
libgcc1.a 302

В

Встраиваемые системы 306
Выравнивание адреса 280

Д

Демонглер 124

З

Зависимость компоновочная (make-файлы) 265, 321
Заголовочные файлы 60, 117, 318

И

Исключения 206
Интернационализация 220

К

Компоновочный файл, *makefile*. 75
Кросс-компиляция 298, 299

Л

Лексический анализ 330
Локализация 220
Локальные установки 221

М

Макрос, макропределение 61
 вариативный 103
 предопределенный 71
Макросы *make*-файлов 267
Маршалинг 168

О

Объектные файлы 232
Отладчик 250
 запуск программы под отладчиком 255
 команды отладочного режима 263
 компиляция для отладки 253
 подключение к программе во время
 выполнения 260
 "посмертный" анализ аварийного завершения
 программы" 258
 форматы отладочной информации 250
Оптимизация 329

П

Парсер 332
Платформа целевая 298
Прагма 69
Поддержка платформ (опции поддержки) 375
 Alpha 376
 Alpha/VMS 382
 ARC 383
 AVR 390
 CRIC 391
 D30V 394
 H8/300 395
 Intel IA-64 397
 Intel 386 и AMDx86-64 399
 Intel 960 406
 M32R/D 408
 M680x0 409
 M68HC1x 413
 M88K 414

Mcore 417

MIPS 418
MMIX 425
MN 10200, MN 1030 427
NS32K 428
PDP-11 430
RS/6000 и PowerPC 432
RT PC 443
S300 и zSeries 444
SH. 445
SPARC 447
System V 452
TMS320C3x/C4x 453
V850 456
Xstormy 457

Препроцессор CPP 60, 111, 139

директивы 60
 ## 70
 #define 61
 #error и #warning 64
 #if, #elif, #else и #endif 64
 #ifndef, #ifndef, #else и #endif 66
 #include 66
 #include_next 68
 #line 68
 #pragma 69
 #undef 70
 _Pragma 69
 defined 65

С

Синтаксический разбор 316, 332, 338
Совмещение языков 197
 Ada и C 216
 C и C++ 198
 Fortran и C 213
 Java и C 207
 Java и C++ 202
 Objective-C и C 200

Сценарий

сборки (*make*-файл) 266
компонентов (утилиты *ld*) 312
конфигурации (*configure*) 41
Стек, кадры стека 99, 100

У

Утилиты
 ar 233

gdb 250
ld 237, 312
ldconfig 241
ldd 238
ldd 246
make 265
nm 243
objdump 247
ranlib 234
strip 245
autoconf 265

flex 331
as 278
Autoconf 273

Ф

Форматы

отладочной информации 250
STABS 251
DWARF 252
COFF 252
XCOFF 253

Сборный компилятор GNU является основой всего мирового программного обеспечения с открытым исходным кодом. Так или иначе, все свободно распространяемые программные продукты, включая и компиляторы, основаны на GCC.

GCC поддерживает широкий набор языков программирования, предусматривает участие неограниченного количества программистов при разработке программ и обеспечивает перенос программ практически на любые операционно-аппаратные платформы.

Его всегда можно получить, это просто и бесплатно.

Разработка и поддержка GCC не прекращается с 1987 года.

В книге детально рассматривается:

- Загрузка, конфигурирование и инсталляция GCC. Использование тестового набора. Получение документации и подписка на списки рассылки. Способы разрешения проблем.
- Создание приложений для Unix, Linux, Windows и встраиваемых систем.
- Кросс-компиляция приложений. Генерирование кода для других машин.
- Условная компиляция и использование директив препроцессора.
- Компиляция программ на языках C, C++, Objective-C, Fortran, Java и Ada. Объединение частей программы, написанных на разных языках программирования. Включение в программы ассемблерного кода и частей, написанных на языках системного уровня. Компиляция байт-кода Виртуальной Машины Java в машинную программу, использование jar-библиотек.
- Применение установок локализации и интернационализации приложений. Поддержка национальных языков и стандартов.
- Оптимизация кода на различных этапах компиляции программ.
- Проверка и обработка объектных файлов. Работа со статическими объектными архивами, совместно используемыми библиотеками и полностью скомпонованными программами.
- Поддержка генерации машинного кода для различных платформ. Использование ассемблеров и компоновщиков, предоставляемых поставщиками оборудования.
- Обработка ошибок компиляции и компоновки. Использование отладчика GNU.
- Необходимые сведения для участия в разработке и поддержке GCC. Разделение верхнего и нижнего уровней компилятора. Промежуточный язык регистраного переноса RTL. Уровни оптимизации. Способы расширения GCC. Добавление пользовательских языков верхнего уровня и средств поддержки целевых платформ.

e-mail: books@diasoft.kiev.ua web-сайт: www.diasoft.kiev.ua

КАТЕГОРИЯ ➤

Программирование / Компиляторы

ОБОЛОЧКА ➤

GCC, версии 3.1 или более новой

УРОВЕНЬ ➤

Знанияющий средний мастер эксперта

Об авторе

Артур Гриффитс – известный автор книг компьютерной тематики и эксперт по компьютерным технологиям. Он участвовал в работах по созданию трансляторов, компиляторов, компоновщиков и ассемблеров с 1977 года.

Артур Гриффитс разрабатывал ассемблеры и компоновщики для специальных вычислительных машин. Принимал участие в создании и поддержке компиляторов языков PLEXUS, SATS, Forth, расширений компилятора COBOL. Он также участвовал в разработке систем реального времени, в частности, командного языка для управления распределенными системами космической связи.

После 5 лет Артур Гриффитс делает книги по компьютерной тематике, занимается дистанционным обучением программированию и пишет программы на языке Java. После включения в GCC компилятора Java он сам решил написать эту книгу.

ISBN 966-7992-34-9



9 789667 992347 >