Язык C

в XXI веке

Бен Клеменс

ДМК ПРЕСС ИЗДАТЕЛЬСТВО

O'REILLY®

УДК 004.6
ББК 32.973.26
К48

Клеменс Бен
К48 Язык С в XXI веке / пер. с англ. А. А. Слинкина. – М.: ДМК Пресс, 2015. – 376 с.: ил.

ISBN 978-5-97060-101-3

Язык С — не просто фундамент всех современных языков программирования, он и сам — современный язык, идеальный для написания эффективных приложений передового уровня. Последние 20 лет С не стоял на месте. Сам язык и окружающая его экосистема подвергаются пересмотру. Эта книга начинается там, где другие заканчиваются. В ней рассказано, как изменилась функциональность, поддерживаемая любым компилятором, благодаря двум новым стандартам С, вышедшим со времен оригинального ANSI. Цель книги – рассмотреть то, чего нет в других учебниках по С: инструменты и окружение; библиотеки для работы со связанными списками и анализаторами XML; написание удобочитаемого кода с дружественным программным интерфейсом.

Издание предназначено для программистов, имеющих опыт работы на каком-либо языке и обладающими базовыми знаниями о С.

УДК 004.6
ББК 32.973.26

ISBN 978-1-491-90389-6 (анг.)
ISBN 978-5-97060-101-3 (рус.)

Содержание

Предисловие 11

Часть I ❖ Окружение 23

Глава 1 ❖ Настраиваем среду для компиляции 24

Работа с менеджером пакетов 25
Компиляция программ на C в Windows 27
POSIX в Windows 27
Компиляция программ на C при наличии подсистемы POSIX 28
Компиляция программ на C в отсутствие подсистемы POSIX 29
Как пройти в библиотеку? 30
Несколько моих любимых флагов 32
Пути 33
Компоновка во время выполнения 36
Работа с файлами makefile 36
Задание переменных 37
Правила 40
Сборка библиотек из исходного кода 43
Сборка библиотек из исходного кода (даже если системный администратор против) 45
Компиляция C-программы с помощью встроенного документа 46
Включение файлов-заголовков из командной строки 46
Универсальный заголовок 47
Встроенные документы 48
Компиляция из stdin 50

Глава 2 ❖ Отладка, тестирование, документирование 51

Работа с отладчиком 51
Отладка программы как детективная история 53
Переменные GDB 62
Распечатка структур 63
Использование Valgrind для поиска ошибок 67
Автономное тестирование 69
Использование программы в качестве библиотеки 72
Покрытие 73
Встроенная документация 74
Doxygen 74
Грамотное программирование с помощью CWEB 76
Проверка ошибок 78
Ошибки и пользователи 78
Учет контекста, в котором работает пользователь 80
Как следует возвращать уведомление об ошибке? 81

Глава 3 ❖ Создание пакета для проекта 83
Оболочка 84
Замена команд оболочки их выводом 84
Применение циклов for в оболочке для обработки набора файлов 86
Проверка наличия файла 88
Команда fc 90
Файлы makefile и скрипты оболочки 92
Создание пакета с помощью Autotools 95
Пример работы с Autotools 96
Описание Makefile с помощью Makefile.am 100
Скрипт configure 104

Глава 4 ❖ Управление версиями 108
Получение списка отличий с помощью diff 109
Объекты Git 110
Тайник 114
Деревья и их ветви 115
Объединение 116
Перемещение 117
Дистанционные репозитории 118

Глава 5 ❖ Мирное сосуществование 121
Динамическая загрузка 121
Ограничения динамической загрузки 124
Процесс 124
Писать так, чтобы можно было понять 124
Функция-обертка 125
Контрабанда структур данных через границу 126
Компоновка 128
Python как включающий язык 128
Компиляция и компоновка 129
Условный подкаталог для Automake 130
Distutils при поддержке Autotools 131

Часть II ❖ Язык 134

Глава 6 ❖ Ваш приятель – указатель 136
Автоматическая, статическая и динамическая память 136
Автоматическая 137
Статическая 137
Динамическая 137
Переменные для хранения постоянного состояния 140
Указатели без malloc 142

Структуры копируются, для массивов создаются псевдонимы ... 143
malloc и игрища с памятью ... 146
Виноваты звезды ... 147
Все, что нужно знать об арифметике указателей ... 148
Typedef как педагогический инструмент ... 150

Глава 7 ❖ Несущественные особенности синтаксиса C, которым в учебниках уделяется чрезмерно много внимания ... 153

Ни к чему явно возвращать значение из main ... 154
Пусть объявления текут свободно ... 154
Меньше приведений ... 157
Перечисления и строки ... 159
Метки, goto, switch и break ... 160
К вопросу о goto ... 161
Предложение switch ... 163
Нерекомендуемый тип float ... 164
Сравнение чисел без знака ... 167
Безопасное преобразование строки в число ... 168

Глава 8 ❖ Важные особенности синтаксиса C, которые в учебниках часто не рассматриваются ... 171

Выращивание устойчивых и плодоносящих макросов ... 172
Приемы работы с препроцессором ... 176
Проверочные макросы ... 179
Защита заголовков ... 181
Компоновка с ключевыми словами static и extern ... 183
Переменные с внешней компоновкой в файлах-заголовках ... 184
Ключевое слово const ... 186
Форма существительное–прилагательное ... 187
Конфликты ... 187
Глубина ... 188
Проблема char const ** ... 189

Глава 9 ❖ Текст ... 192

Безболезненная обработка строк с помощью asprintf ... 192
Безопасность ... 195
Константные строки ... 196
Расширение строк с помощью asprintf ... 197
Песнь о strtok ... 199
Unicode ... 203
Кодировка для программ на C ... 205
Библиотеки для работы с Unicode ... 206
Пример кода ... 208

Глава 10 ❖ Улучшенная структура 211

Составные литералы 212
Инициализация с помощью составных литералов 213
Макросы с переменным числом аргументов 213
Безопасное завершение списков 215
Несколько списков 216
Foreach 217
Векторизация функции 218
Позиционные инициализаторы 219
Инициализация массивов и структур нулями 221
Псевдонимы типов спешат на помощь 222
К вопросу о стиле 224
Возврат нескольких значений из функции 225
Извещение об ошибках 226
Гибкая передача аргументов функциям 228
Объявление своей функции по аналогии с printf 229
Необязательные и именованные аргументы 231
Доведение до ума бестолковой функции 233
Указатель на void и структура, на которую он указывает 239
Функции с обобщенными входными параметрами 239
Обобщенные структуры 244

Глава 11 ❖ Объектно-ориентированное программирование на C 249

Расширение структур и словарей 251
Реализация словаря 253
C без зазоров 257
Функции в структурах 261
V-таблицы 265
Область видимости 270
Закрытые элементы структуры 271
Перегрузка 272
_Generic 274
Подсчет ссылок 277
Пример: объект подстроки 277
Пример: основанная на агентах модель формирования групп 281
Заключение 288

Глава 12 ❖ Параллельные потоки 290

Окружение 291
Составные части 292
OpenMP 293
Компиляция для использования OpenMP 294

Интерференция ... 295
Map-reduce ... 296
Несколько задач ... 297
Поточная локальность ... 299
Локализация нестатических переменных ... 300
Разделяемые ресурсы ... 300
Атомы ... 305
Библиотека pthread ... 307
Атомы C ... 311
Атомарные структуры ... 315

Глава 13 ❖ Библиотеки ... 320

GLib ... 320
Стандарт POSIX ... 321
Разбор регулярных выражений ... 321
Использование mmap для очень больших наборов данных ... 326
Библиотека GNU Scientific Library ... 328
SQLite ... 331
Запросы ... 332
libxml и cURL ... 334

Эпилог ... 338

Приложение ❖ Основные сведения о языке C ... 339

Структура ... 339
В C необходим этап компиляции, состоящий из одной команды ... 340
Существует стандартная библиотека, это часть операционной системы ... 341
Существует препроцессор ... 341
Существуют комментарии двух видов ... 342
Нет ключевого слова print ... 342
Объявления переменных ... 342
Любая переменная должна быть объявлена ... 342
Даже функции необходимо объявлять или определять ... 343
Базовые типы можно агрегировать в массивы и структуры ... 344
Можно определять новые структурные типы ... 345
Можно узнать размер типа ... 346
Не существует специального типа строки ... 346
Функции и выражения ... 347
Правила видимости в C очень просты ... 347
Функция main имеет особый смысл ... 348
Большая часть работы программы на C сводится к вычислению выражений ... 348
При вычислении функций используются копии входных аргументов ... 349

Выражения заканчиваются точкой с запятой 349
Есть много сокращенных способов записи арифметических операций 349
В C понятие истины трактуется расширительно 350
Результатом деления двух целых всегда является целое 350
В C имеется тернарный условный оператор 351
Ветвления и циклы несильно отличаются от других языков 351
Цикл for – просто компактная форма цикла while 352
Указатели 353
Можно напрямую запросить блок памяти 354
Массивы – это просто блоки памяти, любой блок памяти можно использовать как массив 354
Указатель на скаляр – это по существу массив с одним элементом 355
Существует специальная нотация для доступа к полям структур по указателю 356
Указатели позволяют изменять аргументы функции 356
Любой объект где-то находится, и, значит, на него можно указать 357

Глоссарий 358

Библиография 363

Предметный указатель 365

Глава 2

Отладка, тестирование, документирование

Ползу
По твоему окну
Ты мнишь, что мне неловко,
А я вот жду...
Чтоб завершить свою уловку.
— Wire «I Am the Fly»

В этой главе мы рассмотрим средства отладки, тестирования и документирования вашего творения – то, что необходимо для доведения потенциально полезного собрания скриптов до чего-то, на что вы сами и другие люди смогут положиться.

Поскольку C оставляет программисту свободу творить в памяти совершенно немыслимые вещи, отладка является одновременно банальной проверкой логики (с помощью gdb) и технически более сложной задачей вылавливания неправильного выделения и утечек памяти (с помощью Valgrind). В плане документирования мы рассмотрим один инструмент на уровне интерфейса (Doxygen) и другой, которой помогает документировать и разрабатывать каждый шаг программы (CWEB).

В этой главе мы также вкратце коснемся *тестовой оснастки* – комплекта средств, позволяющего быстро писать многочисленные тесты программы. И завершим главу рассмотрением стратегии уведомления об ошибках и принципов обработки данных, вводимых пользователем, и ошибок в них.

Работа с отладчиком

Первый совет касательно отладчика будет прост и краток:

Пользуйтесь отладчиком, обязательно.

Кто-то скажет, что это и не совет вовсе, потому что кто же не пользуется отладчиком? Но поскольку это второе издание книги, могу сообщить, что одной из самых часто повторяющихся просьб было включить более подробное введение в работу с отладчиком – для многих читателей это было откровением.

Некоторые считают, что обычно ошибки – результат неправильного понимания в широком смысле, тогда как отладчик дает только низкоуровневую информацию о состоянии переменных и стека вызовов. Действительно, отыскав место ошибки в отладчике, необходимо остановиться и подумать о том, что привело к этой

ошибке и не проявится ли та же проблема в каком-нибудь другом месте. Иногда в свидетельство о смерти включают глубокий анализ причины: *смерть произошла в результате _____, что явилось результатом _____, что явилось результатом _____, что явилось результатом _____*. После того как с помощью отладчика вы провели такой анализ и стали лучше понимать свою программу, обретенное знание следует инкапсулировать в дополнительных автономных тестах.

Теперь что касается слова «обязательно». Прогон программы под отладчиком обходится практически бесплатно. И не стоит извлекать отладчик на свет божий, только когда что-то ломается. Линус Торвальдс говорит: «Я пользуюсь отладчиком постоянно... вроде как накачанным дизассемблером, который еще и программировать можно»[1]. Ну неужели вас не прельщает возможность остановиться в любом месте, повысить уровень детализации вывода простой командой `print verbose++`, досрочно выйти из цикла `for (int i=0; i<10; i++)`, введя команды `print i=100` и `continue`, или протестировать функцию, подав ей на вход различные значения? Любители интерактивных языков правы в том, что взаимодействие с кодом улучшает процесс разработки во всех отношениях; но они так и не удосужились дочитать учебник C до главы об отладке и не знают, что все эти интерактивные штучки применимы и к C.

Для чего бы вы ни использовали отладчик, он должен уметь представлять хранящуюся в программе отладочную информацию (например, имена переменных и функций) в понятном человеку виде. Чтобы включить в исполняемый файл отладочные символы, при компиляции следует указать флаг `-g` (например, в переменной `CFLAGS`). Причин не использовать флаг `-g` очень мало – он не замедляет работу программы, а увеличение размера файла на килобайт-другой в большинстве случаев несущественно. Отладка также упрощается при отключении оптимизации с помощью флага `-O0` (O нуль), потому что иногда оптимизатор устраняет переменные, которые могли бы оказаться полезны при отладке, да и вообще видоизменяет код всякими неожиданными способами.

Я в основном рассматриваю GDB, потому что в большинстве POSIX-совместимых систем ничего другого просто нет[2]. Отладчик LLDB (поставляемый вместе с LLVM/Clang) постепенно набирает популярность, и я расскажу о нем тоже. Компания Apple перестала включать GDB в свою среду Xcode, но его можно установить с помощью менеджера пакетов, например Macports, Fink или Homebrew. В Mac сеансы отладки, возможно, придется запускать через sudo(!), например `sudo lldb stddev_bugged`.

Быть может, вы работаете в IDE или другой графической среде, которая запускает вашу программу под отладчиком всякий раз, как вы выбираете из меню команду «Выполнить». Я буду демонстрировать только работу из командной строки, но

---

[1] Из письма Торвальдса к коллеге от 6 сентября 2000 года.

[2] Кстати, компилятор C++ подправляет (mangle) имена функций. В GDB это видно, и я всегда считал отладку кода на C++ в GDB мучительным делом. Но компилятор C ничего подобного не делает, поэтому для него с GDB работать куда проще, и не нужны вспомогательные средства для восстановления имени в исходном виде.

вряд ли вас затруднит перевод команд на язык щелчков мышью. Некоторые графические фасады позволяют использовать макросы, определенные в файле *.gdbinit*.

При работе непосредственно с командной строкой вам, возможно, будет удобно видеть код в текстовом редакторе в соседнем открытом окне или на другом терминале. Простая комбинация отладчика с редактором дает многие преимущества IDE и, может статься, больше вам ничего и не понадобится.

Стек кадров

Для запуска программы необходимо попросить систему выполнить функцию `main`. Компьютер генерирует *кадр*, в котором хранится информация о вызове функции, в том числе ее входные параметры (которые в случае `main` принято называть `argc` и `argv`) и созданные внутри нее локальные переменные.

Допустим, что в процессе выполнения `main` вызывает функцию `get_agents`. В этот момент выполнение `main` приостанавливается и генерируется новый кадр для `get_agents`, где хранятся детали ее вызова. Быть может, `get_agents`, в свою очередь, вызывает функцию `agent_address`, и таким образом мы получаем растущий *стек* кадров. Рано или поздно выполнение `agent_address` завершится, в этот момент ее кадр будет вытолкнут из стека, и возобновится выполнение `get_agents`.

На вопрос «Где я нахожусь?» проще всего ответить, указав номер строки в программе, и иногда этого достаточно. Однако чаще вас интересует «Как я сюда попал?», и ответом на этот вопрос является *обратная трассировка*, или *стек вызовов*, то есть стек кадров. Вот пример обратной трассировки:

```
#0 0x0000000000413bbe in agent_address (agent_number=312) at addresses.c:100
#1 0x00000000004148b6 in get_agents () at addresses.c:163
#2 0x0000000000404f9b in main (argc=1, argv=0x7fffffffe278) at addresses.c:227
```

На вершине стека находится кадр 0, а на самом дне – вызов `main`, который в данный момент находится в кадре 2 (но номер кадра будет меняться по мере роста и сокращения стека). Шестнадцатеричное число после номера кадра – это адрес, с которого возобновится выполнение после возврата из вызванной функции; для меня как прикладного программиста это только зрительный шум, я на него не обращаю внимания. Далее показаны имя функции, ее входные параметры (в случае `argv` это опять-таки шестнадцатеричный адрес) и номер строки в исходном коде.

Если вы обнаружили, что номер дома в адресе агента (`agent_address`) заведомо неправилен, то, быть может, это потому, что передан неправильный номер агента (`agent_number`), и в этом случае стоит перейти в кадр 1 и поинтересоваться, в каком состоянии находилась `get_agents` и почему получилось такое странное состояние `agent_address`. В значительной мере искусство исследования программы заключается в перемещении по стеку и прослеживании причин и следствий между кадрами.

Отладка программы как детективная история

В этом разделе мы рассмотрим воображаемый сеанс вопросов и ответов с применением GDB или LLDB. В примерах к этой книге имеется файл `stddev_bugged.c`, вариант в примере 7.4 с ошибкой. Как в любом хорошем детективе, все ключи, необходимые для изобличения преступника, перед вами. Правильно выстроенная последовательность вопросов поможет исключить подозреваемых одного за другим, пока не останется всего один и ошибка не станет очевидной.

После компиляции программы (с помощью команды `CFLAGS="-g" make stddev_bugged`) приступим к расследованию и для начала запустим отладчик:

```
gdb stddev_bugged
# или
lldb stddev_bugged
```

Перед нами приглашение к вводу команд отладчика, можно задавать вопросы.

В. Что делает эта программа?

О. Команда `run` запускает программу. Ее, как и все команды GDB и LLDB, можно записать в сокращенном виде:

```
(gdb) r
mean: 5687.496667 var: 194085710
mean: 0.83 var: 4.1334
[Inferior 1 (process 22734) exited normally]
```

Похоже, программа вычисляет какие-то средние значения и дисперсии. Она дошла до конца без нарушения защиты памяти и прочих ошибок и вернула нуль, что означает нормальное завершение.

В. Проверяет ли код в `main` правильность результата?

О. Посмотреть на исходный код проще всего, открыв его в текстовом редакторе. Существуют способы расположить редактор рядом с отладчиком даже при удаленной работе на машине, оснащенной только текстовыми терминалами (см. врезку «Визуальные средства» на стр. 116). Но GDB и LLDB тоже умеют показывать фрагменты кода, для этого служит команда `list`:

```
(gdb) l main

28       }
29       return (meanvar){.mean = avg,
30                        .var = avg2 - pow(avg, 2)}; //E[x^2] - E^2[x]
31    }
32
33    int main(){
34       double d[] = { 34124.75, 34124.48,
35                      34124.90, 34125.31,
36                      34125.05, 34124.98, NAN};
37
```

Выведено десять строк программы выше и ниже запрошенной. При повторном выполнении команды `list` без аргументов будут выведены следующие 10 строк:

```
(gdb) l
38       meanvar mv = mean_and_var(d);
39       printf("mean:%.10g var:%.10g\n", mv.mean, mv.var*6/5.);
40
41       double d2[] = { 4.75, 4.48,
42                       4.90, 5.31,
43                       5.05, 4.98, NAN};
44
45       mv = mean_and_var(d2);
46       mv.var *= 6./5;
47       printf("mean:%.10g var:%.10g\n", mv.mean, mv.var);
```

В строке 38 мы видим обращение к функции `mean_and_var`, которой передается список `d`. И тут, очевидно, имеет место проблема: все числа в `d` расположены в окрестности 34 125, а напечатанное программой среднее значение приближенно равно 5687 (не говоря уже о какой-то несуразной дисперсии). Аналогично при втором обращении функции `mean_and_var` передается список чисел в окрестности 5, а среднее оказалось равно 0.83.

Вся оставшаяся часть сеанса посвящена поиску ответа на единственный вопрос: *где то первое место в программе, начиная с которого все пошло наперекосяк*? Но чтобы ответить на этот главный вопрос, нам нужны дополнительные детали.

В. Как узнать, что происходит внутри `mean_and_var`?

О. Мы хотим приостановить программу при входе в `mean_and_var`, чтобы поставить там точку останова:

```
(gdb) b mean_and_var
Breakpoint 1 at 0x400820: file stddev_bugged.c, line 16.
```

Поставив точку останова, заново запустим программу – она остановится в этой точке:

```
(gdb) r
Breakpoint 1, mean_and_var (data=data@entry=0x7fffffffe130) at
stddev_bugged.c:16
16 meanvar mean_and_var(const double *data){
(gdb)
```

Сейчас мы стоим в строке 16, в самом начале функции, и можем задавать дальнейшие вопросы о том, что в ней происходит.

В. В переменной `data` находится то, что мы думаем?

О. Посмотреть на переменную `data` в текущем кадре позволяет команда `print`, сокращенно `p`:

```
(gdb) p *data
$2 = 34124.75
```

Печально – нам показали только первый элемент. Однако в GDB имеется специальная конструкция `@-` для печати последовательности элементов массива. Вот как запросить первые 10 элементов [LLDB: `mem read -tdouble -c10 data`]:

```
(gdb) p *data@10
$3 = {34124.75,
  34124.480000000003,
  34124.900000000001,
  34125.309999999998,
  34125.050000000003,
  34124.980000000003,
  nan(0x8000000000000),
  7.7074240751234461e-322,
  4.9406564584124654e-324,
  2.0734299798669383e-317}
```

Обратите внимание на звездочку в начале выражения, без нее мы получили бы последовательность из десяти шестнадцатеричных адресов.

Я запросил 10 элементов, потому что было лень считать, сколько элементов хранится в наборе данных, но первые семь из десяти выглядят правильно: последовательность чисел, в конце которой находится маркер NaN. После него мы видим мусор – неинициализированную память за концом массива.

В. Соответствует ли это тому, что мы передали из `main`?

О. Команда `bt` печатает обратную трассировку:

```
(gdb) bt
#0 mean_and_var (data=data@entry=0x7fffffffe130) at stddev_bugged.c:16
#1 0x0000000000400680 in main () at stddev_bugged.c:38
```

В стеке находятся всего два кадра: текущий и вызвавший его, `main`. Посмотрим на данные в кадре 1 и для начала переключимся на него:

```
(gdb) f 1
#1 0x0000000000400680 in main () at stddev_bugged.c:38
38 meanvar mv = mean_and_var(d);
```

Сейчас отладчик находится в кадре функции `main`, в строке 38. Это ожидаемое место, так что порядок выполнения правильный (и не изменен оптимизатором). Находясь в этом кадре, посмотрим на массив данных с именем `d`:

```
(gdb) p *d@7
$5 = {34124.75,
  34124.480000000003,
  34124.900000000001,
  34125.309999999998,
  34125.050000000003,
  34124.980000000003,
  nan(0x8000000000000)}
```

Данные совпадают с теми, что мы видели в кадре `mean_and_var`, так что с набором данных вроде бы ничего странного не произошло.

Нам нет необходимости явно возвращаться в кадр 0, чтобы продолжить выполнение программы, но это можно было бы сделать командой `f 0` или командой перемещения по стеку относительно текущего кадра:

```
(gdb) down
```

Отметим, что в командах `up` и `down` предполагается числовой порядок. Если считать, что в списке, который выводит `bt` (как в GDB, так и в LLDB), кадр с наименьшим номером располагается сверху, то `up` идет вниз, а `down` вверх по списку обратной трассировки.

В. Эта проблема случайно не связана с параллельными потоками?

О. Получить список потоков позволяет команда `info threads` [LLDB: `thread list`]:

```
(gdb) info threads
Id   Target Id          Frame
```

```
* 1  Thread 0x7ffff7fcb7c0 (LWP 28903) "stddev_bugged" mean_and_var
(data=data@entry=0x7fffffffe180) at stddev_bugged.c:16
```

В данном случае существует всего один активный поток, поэтому проблема никак не может быть связана с многопоточностью. Символ * показывает, в каком потоке сейчас находится отладчик. Если бы существовал поток 2, то мы могли бы перейти в него командой `thread 2` (GDB) или `thread select 2` (LLDB).

Если до сих пор вы в своих программах не запускали несколько потоков, то после прочтения главы 12 ситуация обязательно изменится. Пользователям GDB рекомендуется добавить в файл `.gdbinit` показанную ниже команду, чтобы отключить надоедливые уведомления о каждом создании нового потока:

```
set print thread-events off
```

В. Что делает функция `mean_and_var`?

О. Мы можем пройти функцию в пошаговом режиме, повторно выполняя следующую команду:

```
(gdb) n
18              avg2 = 0;
(gdb) n
16    meanvar mean_and_var(const double *data){
```

Простое нажатие клавиши **Enter** повторяет предыдущую команду, так что даже букву `n` вводить необязательно:

```
(gdb)
18              avg2 = 0;
(gdb)
20          size_t count= 0;
(gdb)
16    meanvar mean_and_var(const double *data){
(gdb)
21          for(size_t i=0; !isnan(data[i]); i++){
(gdb)
21          for(size_t i=0; !isnan(data[i]); i++){
(gdb)
22            ratio = count/(count+1);
(gdb)
26            avg += data[i]/(count +0.0);
```

Номера строк показывают, что программа выполняется не последовательно. Объясняется это тем, что на каждом шаге отладчик выполняет машинные команды, которые необязательно точно соответствуют коду на C, из которого сгенерированы. Это нормально даже в случае, когда уровень оптимизации равен нулю. Переходы могут также отражаться на переменных – их значения ненадежны до момента выполнения второй или третьей строки после той, что выбивается из общего порядка.

Существуют и другие способы пошагового выполнения, чаще всего используются команды `s`, `n`, `u`, `c` (см. таблицу ниже). Однако на такой проход по программе может уйти весь день. Мы видим, что обход массива `data` производится в цикле `for`, так давайте поставим еще одну точку останова внутри этого цикла:

```
(gdb) b 25
Breakpoint 2 at 0x400875: file stddev_bugged.c, line 25.
```

Теперь у нас есть две точки останова, их можно посмотреть командой `info break` (GDB) или `break list` (LLDB):

```
(gdb) info break
Num     Type           Disp  Enb Address            What
1       breakpoint     keep  y   0x0000000000400820 in mean_and_var
                                                    at stddev_bugged.c:16
        breakpoint already hit 1 time
2       breakpoint     keep  y   0x0000000000400875 in mean_and_var
                                                    at stddev_bugged.c:25
```

Точка останова в начале функции `mean_and_var` нам больше не нужна, поэтому деактивируем ее [LLDB: `break dis 1`]:

```
(gdb) dis 1
```

После этого в столбце `Enb` таблицы, печатаемой командой `info break`, для точки останова 1 будет стоять `n`. Впоследствии точку останова можно реактивировать командой `enable 1` (GDB) или `break enable 1` (LLDB). А если вы точно знаете, что точка останова больше не понадобится, то ее можно вообще удалить командой `del 1` (GDB) или `break del 1` (LLDB).

В. Какие значения имеют переменные внутри цикла?

О. Можно начать выполнение программы с самого начала командой `r` или продолжить с места, где мы остановились, командой `c`:

```
(gdb) c
Breakpoint 2, mean_and_var (data=data@entry=0x7fffffffe130) at
stddev_bugged.c:25
25            avg2 *= ratio;
```

Сейчас мы остановились в строке 25 и можем просмотреть все локальные переменные [LLDB: `frame variable`]:

```
(gdb) info local
i = 0
avg = 0
avg2 = 0
ratio = 0
count = 1
```

Можно также проверить входные аргументы функции с помощью команды GDB `info args`, хотя ранее мы уже и так выводили массив `data`. Команда LLDB `frame variable` выводит как локальные переменные, так и входные аргументы.

В. Мы знаем, что выведенное среднее неправильно, а как изменяется переменная `avg` на каждой итерации цикла?

О. Можно было бы выполнять команду `p avg` при каждом попадании в точку останова, но этот процесс можно автоматизировать с помощью команды `display`:

```
(gdb) disp avg
1: avg = 0
```

Теперь, когда мы продолжим выполнение, отладчик будет крутиться в цикле и каждый раз в точке останова печатать текущее значение avg:

```
(gdb) c
Breakpoint 2, mean_and_var (data=data@entry=0x7fffffffe130) at stddev_bugged.c:25
25            avg2 *= ratio;
1: avg = 0

(gdb)
Breakpoint 2, mean_and_var (data=data@entry=0x7fffffffe130) at stddev_bugged.c:25
25            avg2 *= ratio;
1: avg = 0
```

Плохой знак: в программе есть строки

```
avg *= ratio;
...
avg += data[i]/(count +0.0);
```

поэтому avg должна бы изменяться на каждой итерации, но она как была равна нулю, так и остается. Установив, что это ошибка, мы можем больше не отвлекаться на переменную avg (которая в списке команды display значится под номером 1) и отменить ее автоматическую печать командой undisp 1.

В. Чему равны переменные, используемые при вычислении avg?

О. Мы уже убедились, что с data все в порядке, а как насчет ratio и count?

```
(gdb) disp ratio
2: ratio = 0

(gdb) disp count
3: count = 3
```

Выполнив еще несколько итераций цикла, мы увидим, что count, как и положено счетчику, каждый раз увеличивается на 1, а вот ratio не изменяется:

```
(gdb) c
Breakpoint 2, mean_and_var (data=data@entry=0x7fffffffe130) at
stddev_bugged.c:25
25            avg2 *= ratio;
3: count = 4
2: ratio = 0
```

В. Где присваивается значение ratio?

О. Посмотрев код в текстовом редакторе или командой l, мы увидим, что переменной ratio присваивается значение только в строке 22:

```
ratio = count/(count+1);
```

Мы уже убедились, что count увеличивается, но что-то же в этой строке неправильно. И сейчас должно быть понятно, что именно: если count целое число, то

в выражении `count/(count+1)` применяется целочисленное деление и, стало быть, возвращается целое число (3/4==0), хотя должна бы выполняться операция деления чисел с плавающей точкой, хорошо нам известная со школьной скамьи (3/4==0.75). Чтобы получить правильный результат (см. раздел «Меньше приведений» на стр. 157), нужно сделать так, чтобы либо числитель, либо знаменатель был числом с плавающей точкой. Для этого достаточно заменить целую константу `1` константой с плавающей точкой `1.0`:

```
ratio = count/(count+1.0);
```

Отладчик не предупредил нас об этой распространенной ошибке, но помог отыскать то место в программе, где что-то впервые пошло не так, и, безусловно, найти ошибку в одной строке проще, чем во фрагменте из 50 строк. Попутно мы получили возможность проверить разнообразные аспекты поведения программы и лучше понять порядок ее выполнения и структуру стека кадров.

Ниже приведен список наиболее употребительных команд отладчика. И в GDB, и в LLDB команд гораздо больше, но показанные ниже – это те 10%, которые используются в 90% случаев. Имена переменных по большей части взяты из программы скачивания заголовков газеты «Нью-Йорк таймс», описанной в разделе «libxml и cURL» ниже.

Таблица 2.1 ❖

Группа	Команда	Назначение
Запуск	`run`	Запустить программу с начала
	`run args`	Запустить программу с начала с указанными аргументами в командной строке
Останов	`b get_rss`	Приостановить выполнение в начале функции
	`b nyt_feeds.c:105`	Приостановить выполнение перед указанной строкой
	`break 105`	То же, что `b nyt_feeds.c:105`, если мы уже остановились в файле `nyt_feeds.c`
	`info break` [GDB] `break break` [LLDB]	Вывести список точек останова
	`watch curl` [GDB] `watch set var curl` [LLDB]	Остановиться, если значение указанной переменной изменилось
	`dis 3` / `ena 3` / `del 3` [GDB] `break dis 3` / `break ena 3` / `break del 3` [LLDB]	Деактивировать/реактивировать/удалить точку останова `3`. Если точек останова много, то можно деактивировать все командой `disable` без параметров, а затем активировать одну-две, нужные в данный момент; то же относится к командам `enable` и `delete`
Просмотр переменных	`p url`	Напечатать значение переменной `url`. Можно задать любое выражение, в том числе содержащее вызов функции
	`p *an_array@10` [GDB]	Напечатать первые десять элементов массива `an_array`. Для печати следующих десяти элементов выполните команду `*(an_array+10)@10`

Таблица 2.1 ❖ (окончание)

Группа	Команда	Назначение
	`mem read -tdouble -c10 an_array`	Прочитать 10 элементов типа `double` из массива `an_array`. Для чтения следующих десяти элементов выполните команду `mem read -tdouble -c10 an_array+10`
	`info args` / `info vars` [GDB] `frame var` [LLDB]	Получить значения аргументов функции или всех локальных переменных
	`disp url`	Печатать значение `url` при каждом останове программы
	`undisp 3`	Прекратить печать переменной с порядковым номером `3`. GDB: если номер не указан, перестают печататься значения всех переменных
Потоки	`info thread` [GDB] `thread list` [LLDB]	Вывести список активных потоков
	`thread 2` [GDB] `thread select 2` [LLDB]	Переключиться на поток `2`
Кадры	`bt`	Вывести стек кадров
	`f 3`	Показать кадр `3`
	`up` / `down`	Перейти на кадр с номером, на единицу большим или меньшим текущего
Пошаговое выполнение	`s`	Шаг на одну строку, даже если она находится в другой функции
	`n`	Перейти к следующей строке, но не заходить внутрь функции
	`u`	Вперед до следующей строки, считая от текущей (поэтому на повторных итерациях текущего цикла останова не происходит, программа останавливается на строке, следующей за циклом)
	`c`	Продолжить до следующей точки останова или до завершения программы
	`ret` или `ret 3` [GDB]	Немедленно выйти из текущей функции, вернув указанное значение (если оно задано)
	`j 105` [GDB]	Перейти к произвольной (с разумными ограничениями) строке
Просмотр кода	`l`	Напечатать 10 строк, окружающих текущую
Повтор	Клавиша **Enter**	Нажатие клавиши **Enter** без ввода каких-либо данных приводит к повтору последней команды, что упрощает пошаговое выполнение. Если последней командой была `l`, то печатаются следующие 10 строк
Компиляция	`make` [GDB]	Запустить `make`, не выходя из GDB. Можно также указать цель, например: `make myprog`
Справка	`help`	Посмотреть, что предлагает отладчик

Переменные GDB

В этом разделе мы рассмотрим некоторые полезные возможности отладчика, позволяющие просматривать данные с максимальными удобствами. Все описываемые команды выполняются из командной строки; встроенные в IDE отладчики на основе GDB часто предлагают средства для встраивания их в собственный интерфейс.

Ниже приведен пример программы, которая не делает ничего полезного, но в ней есть переменная, которую можно опросить. Поскольку это программа-пустышка, не забудьте при компиляции задать флаг отключения оптимизации `-O0`, иначе от переменной `x` не останется никаких следов.

```
int main(){
    int x[20] = {};
    x[0] = 3;
}
```

Первый совет покажется новым только тем, кто не читал руководства по GDB [Stallman 2002], и есть подозрение, что это вы все. Чтобы меньше нажимать клавиши, можно завести вспомогательные переменные. Например, чтобы опросить элемент, до которого нужно долго добираться по цепочке структур, можно поступить следующим образом:

```
(gdb) set $vd = my_model->dataset->vector->data
p *$vd@10

(lldb) p double *$vd = my_model->dataset->vector->data
mem read -tdouble -c10 $vd
```

В первой строке создается вспомогательная переменная, вместо которой подставляется длинный путь. Как и в оболочке, переменные обозначаются знаком доллара. Но, в отличие от оболочки, в GDB при первом определении переменной используются команда `set` и знак доллара, а в LLDB – синтаксический анализатор Clang для вычисления выражений, поэтому объявление в LLDB синтаксически ничем не отличается от обычного объявления в C. Во второй строке в обоих случаях демонстрируется пример использования. Здесь мы несильно сэкономили на количестве ударов по клавишам, но если вы подозреваете некую переменную в ошибке, то наличие у нее короткого имени ускорит ввод команд опроса.

Но это не просто имена, а настоящие переменные, которые можно изменять. Остановившись в третьей или четвертой строке этой программы-пустышки, попробуйте ввести такие команды:

```
(gdb) set $ptr=&x[3]
p *$ptr = 8
p *($ptr++)  # напечатать то, на что ведет указатель, и продвинуться на следующий элемент

(lldb) p int *$ptr = &x[3]
p *$ptr = 8
p *($ptr++)
```

Команда во второй строке изменяет значение по указанному адресу. Увеличение указателя на единицу сдвигает указатель на следующий элемент списка (как описано в разделе «Все, что нужно знать об арифметике указателей» на стр. 148), поэтому после выполнения третьей строки `$ptr` указывает на `x[4]`.

Последняя форма особенно удобна, потому что нажатие клавиши **Enter** без ввода данных повторяет последнюю команду. Поскольку указатель продвигается вперед, при каждом нажатии **Enter** мы будем получать новое значение, пока не переберем всего массива. Это полезно также при обходе связанного списка. Представьте, что имеется функция `show_structure`, которая отображает элемент связанного списка и устанавливает переменную `$list` равной текущему элементу. Пусть также указатель на начало списка хранится в переменной `list_head`. Если выполнить команду

```
p $list=list_head
show_structure $list->next
```

а затем просто нажимать **Enter**, то мы обойдем весь список. Ниже мы воплотим в жизнь идею воображаемой функции для показа структуры данных.

Но сначала рассмотрим еще один прием работы с `$`-переменными. Позволю себе скопировать пару строк из сеанса работы с отладчиком на другом экране:

```
(gdb|lldb) p x+3
$17 = (int *) 0xbffff9a4
```

Обратите внимание, что результат, выведенный командой печати, начинается с `$17`. На самом деле каждый выведенный результат присваивается переменной, которую можно использовать как любую другую:

```
(gdb|lldb) p *$17
$18 = 8
(gdb|lldb) p *$17+20
$19 = 28
```

Более того, в GDB переменной с именем `$` присваивается последний выведенный результат. Поэтому если вы получили некий шестнадцатеричный адрес, то для печати хранящегося по этому адресу значения нужно просто выполнить команду `p *$`. Таким образом, показанные выше шаги можно было записать так:

```
(gdb) p x+3
$20 = (int *) 0xbffff9a4
(gdb) p *$
$21 = 8
(gdb) p $+20
$22 = 28
```

Распечатка структур

Можно определять простые макросы, что особенно полезно для отображения нетривиальных структур данных – а именно ради этого мы чаще всего и работаем с отладчиком. Даже простой двумерный массив режет глаз, если вывести его в виде длинной строки чисел. В идеальном мире для каждой структуры данных

существовала бы отдельная команда отладчика, позволяющая распечатать ее в наиболее удобном виде (или разных видах).

Но, быть может, у вас уже есть C-функция, которая распечатывает сложную структуру, встречающуюся в программе. Тогда можно написать макрос, который просто вызовет эту функцию. Средство довольно примитивное, но удобное.

В отладчике невозможно воспользоваться макросами препроцессора C, потому что они расширяются задолго до того, как отладчик видит вашу программу. Поэтому если в программе имеются полезные макросы, то их придется повторно реализовать в отладчике.

Ниже показана функция GDB, которую можно испытать, поставив точку останова в том месте функции `parse`, описанной в разделе «libxml и cURL» на стр. 334, где имеется структура `doc`, представляющая дерево XML-документа. Поместите следующие макросы в файл `.gdbinit`.

```
define pxml
   p xmlElemDump(stdout, $arg0, xmlDocGetRootElement($arg0))
end
document pxml
Распечатать на экране дерево уже открытого XML-документа (к примеру, xmlDocPtr). Скорее всего, распечатка займет несколько страниц.
Например, если дано: xmlDocPtr doc = xmlParseFile(infile);
то следует написать: pxml doc
end
```

Обратите внимание, что документация находится сразу после самой функции; просмотреть ее можно с помощью команды `help pxml` или `help user-defined`. Макрос экономит всего несколько нажатий клавиш, но поскольку работа с отладчиком сводится в основном к просмотру данных, эта экономия суммируется.

Варианты этих макросов для LLDB я покажу позже.

В библиотеке GLib имеется структура связанного списка, поэтому хотелось бы иметь средство просмотра такого списка. В примере 2.1 оно реализовано в виде двух доступных пользователю макросов (`phead` для просмотра начала списка и `pnext` для перехода к следующему элементу) и одного макроса, которого пользователь вызывать не должен (`plistdata` – для устранения избыточности при реализации `phead` и `pnext`).

Пример 2.1 ❖ Набор макросов для отображения связанного списка в GDB – пожалуй, ничего сложнее этого отладочного макроса вам никогда не понадобится (gdb_showlist)

```
define phead
   set $ptr = $arg1
   plistdata $arg0
end
document phead
Напечатать первый элемент списка. Если имеется объявление
   Glist *datalist;
   g_list_add(datalist, "Hello");
то для просмотра списка можно использовать такие команды:
```

```
gdb> phead char datalist
gdb> pnext char
gdb> pnext char
В этом макросе $ptr – указатель на текущий элемент списка, а $pdata -
данные, хранящиеся в этом элементе.
end

define pnext
   set $ptr = $ptr->next
   plistdata $arg0
end
document pnext
Сначала необходимо вызвать phead; при этом устанавливается $ptr.
Этот макрос переходит к следующему элементу списка и показывает хранящееся
в нем значение. Единственным аргументом должен быть тип данных в списке.
В этом макросе $ptr – указатель на текущий элемент списка, а $pdata -
данные, хранящиеся в этом элементе.
end

define plistdata
   if $ptr
      set $pdata = $ptr->data
   else
      set $pdata= 0
   end
   if $pdata
      p ($arg0*)$pdata
   else
      p "NULL"
   end
end
document plistdata
Предназначен для вызова из phead и pnext, см. выше. Устанавливает
переменную $pdata и печатает ее значение.
end
```

В примере 2.2 приведен пример кода, в котором в списке `GList` хранятся данные типа `char *s`. Можете поставить точки останова перед строкой 8 и после строки 9 и вызвать в них показанные выше макросы.

Пример 2.2 ❖ Пример кода для экспериментов с отладкой. Можно рассматривать как сверхкраткое введение в связанные списки из библиотеки GLib (glist.c)

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

GList *list;

int main(){
    list = g_list_append(list, "a");
    list = g_list_append(list, "b");
    list = g_list_append(list, "c");

    for ( ; list!= NULL; list=list->next)
        printf("%s\n", (char*)list->data);
}
```

Можно определить функции, которые будут выполняться до или после каждого использования некоторой команды. Например, команды (GDB):

```
define hook-print
echo <----\n
end

define hookpost-print
echo ---->\n
end
```

будут выводить специального вида скобки до и после любого напечатанного значения. Самая интересная точка подключения – `hook-stop`. Команда `display` печатает значение произвольного выражения при каждом останове программы, но если вы хотите при каждом останове выполнять некоторый макрос или какую-нибудь другую команду GDB, то переопределите макрос `hook-stop`:

```
define hook-stop
pxml suspect_tree
end
```

После того как возникшие подозрения проверены, восстановите стандартное поведение:

```
define hook-stop
end
```

Для пользователей LLDB: см. `target stop-hook add`.

Ваша очередь. Макросы GDB могут включать также команду `while`, которая очень похожа на команды `if` из примера 2.1 (в начале строка вида `$ptr`, а в конце `end`). Воспользуйтесь ей, чтобы написать макрос, который выводит сразу весь список.

В LLDB все это делается немного иначе.

Во-первых, как вы, наверное, заметили, команды LLDB довольно многословны, потому что авторы ожидают, что для наиболее часто используемых команд вы сами напишете псевдонимы. Например, вот как можно было бы написать псевдонимы для команд печати массивов типа `double` или `int`:

```
(lldb) command alias dp memory read -tdouble -c%1
command alias ip memory read -tint -c%1

# Примеры использования:
dp 10 data
ip 10 idata
```

Механизм псевдонимов предназначен для сокращенной записи существующих команд. Не существует способа назначить псевдониму команды строку справки, поскольку LLDB пользуется справкой, ассоциированной с исходной командой. Для написания макросов, аналогичных показанным выше макросам GDB, в LLDB применяются регулярные выражения.

Вот версия для LLDB, которую можно поместить в файл `.lldbinit`:

```
command regex pxml
        's/(.+)/p xmlElemDump(stdout,%1, xmlDocGetRootElement(%1)0)/'
        -h "Dump the contents of an XML tree."
```

Подробное обсуждение регулярных выражений выходит за рамки этой книги (в Сети можно найти сотни пособий по регулярным выражениям). Отметим лишь,

что содержимое строки между первой и второй косой чертой будет подставлено вместо маркера%1, встречающегося между второй и третьей косой чертой.

Профилирование

Как бы быстро ни работала программа, всегда хочется, чтобы она была еще быстрее. В большинстве языков сразу дают совет: переписывайте все на C. Но мы-то и так уже пишем на C. Следующий шаг – найти функции, на которые уходит больше всего времени, их оптимизация даст наибольший эффект.

Прежде всего включите в переменную `CFLAGS` для gcc или icc флаг `-pg` (действие этого флага зависит от компилятора; gcc подготовит программу для работы с gprof, а компилятор Intel – для работы с prof, в остальном порядок действий аналогичен, поэтому я ограничусь только деталями для gcc). Программа, откомпилированная с этим флагом, будет приостанавливаться каждые несколько микросекунд и смотреть, в какой функции она сейчас находится. Эта информация записывается в двоичном формате в файл `gmon.out`.

Профилируется только сам исполняемый файл, но не скомпонованные с ним библиотеки. Поэтому если необходимо профилировать также библиотеку при исполнении тестовой программы, то придется скопировать код программы и библиотеки в одно место, а затем перекомпилировать их и собрать в один большой исполняемый файл.

После прогона программы выполните команду `gprof your_program > profile` (или `prof …`), затем откройте файл `profile` в текстовом редакторе, вы увидите список функций с указанием места, откуда они вызывались, и доли времени, проведенной программой в каждой функции. Возможно, информация о том, где в программе узкие места, станет для вас неожиданностью.

Использование Valgrind для поиска ошибок

При работе с отладчиком большая часть времени уходит на поиск места, где программа впервые начинает вести себя подозрительно. Хорошая система сама найдет это место. Иными словами, в хорошей системе программа быстро «грохнется».

Язык C в этом отношении противоречив. В некоторых языках опечатка вида `conut=15` просто приведет к созданию новой переменной, не имеющей ничего общего с переменной `count`, которую вы имели в виду; в C это будет обнаружено на этапе компиляции. С другой стороны, C позволит присвоить значение десятому элементу массива, содержащего всего 9 элементов, и долго еще будет радостно работать, пока вы не обнаружите, что там, где, по вашему мнению, должен был находиться элемент 10, на самом деле мусор.

Подобные проблемы с управлением памятью вызывают много хлопот, и потому существуют инструменты для борьбы с ними. И на одном из первых мест стоит Valgrind. Это средство перенесено на большую часть POSIX-совместимых систем (включая OS X), и установить его можно с помощью менеджера пакетов. А пользователи Windows могут поэкспериментировать с программой Dr. Memory.

Valgrind запускает виртуальную машину, которая лучше следит за использованием памяти, чем реальная, и потому знает, что вы обратились к десятому элементу массива, в котором всего 9 элементов.

Откомпилировав программу (в gcc и Clang, разумеется, с флагом `-g` для включения отладочных символов), выполните команду:

```
valgrind your_program
```