Московский Государственный Университет имени М.В. Ломоносова
Факультет вычислительной математики и кибернетики

С.В. Герасимов, И.В. Машечкин, М.И. Петровский, И.С. Попов, А.Н. Терехин, А.В. Чернов

Инструментальные средства разработки ПО в ОС UNIX

(учебно-методическое пособие)

Москва
2011

УДК 004.4'23 004.432.2 004.428.2 004.4'42 004.416.2 004.451.9

ББК 32.973-018.2

С40

В пособии рассматриваются популярные инструментальные средства поддержки жизненного цикла разработки ПО в ОС UNIX. Материал подготовлен авторами на основании практического опыта применения соответствующих инструментов. Пособие создано в поддержку курса «Операционные системы», читаемому студентам 2-го курса на факультете ВМК МГУ имени М.В.Ломоносова. Материал рассчитан на широкий круг читателей: студентов, аспирантов и преподавателей, а также всех интересующихся программированием в ОС UNIX.

УДК 004.4'23 004.432.2
004.428.2 004.4'42

004.416.2 004.451.9

ББК 32.973-018.2

Рецензенты:

доцент И.А.Волкова

доцент Е.А.Кузьменкова

Герасимов С.В., Машечкин И.В., Петровский М.И., Попов И.С., Терехин А.Н., Чернов А.В.

С40 **Инструментальные средства разработки ПО в ОС UNIX: учебно-методическое пособие**.

Издательский отдел факультета ВМК МГУ 2011, - 68 с.

Печатается по решению Редакционно-издательского Совета факультета вычислительной математики и кибернетики МГУ им. М.В. Ломоносова

ISBN 978-5-89407-458-0

Оглавление

Введение

Результатами работы над большими программными проектами являются сотни тысяч, иногда миллионы строк исходного кода, создаваемые на протяжении нескольких лет командами из десятков разработчиков.

Для продуктивной работы в сфере разработки ПО современный специалист должен обладать глубокими знаниями в области *программной инженерии (англ., software engineering)* – инженерных подходах, позволяющих систематизировать работу в рамках *жизненного цикла ПО (англ., software development life cycle)* и повысить качество результата.

Данное методическое пособие является кратким обзором популярных средств инструментальной поддержки процессов жизненного цикла ПО, используемых разработчиками в операционных системах семейства UNIX. Наряду с ключевыми инструментами, такими как компилятор и редактор внешних связей из набора GNU Compiler Collection, отладчик GNU Debugger и интегрированная среда разработки Code::Blocks, в пособии рассматриваются дополнительные средства, позволяющие повысить эффективность работы при разработке ПО: система автоматизации сборки GNU Make, профилировщик Valgrind, система модульного тестирования CUnit и система документирования исходных текстов Doxygen.

Компиляция и компоновка с помощью GNU Compiler Collection

GNU compiler collection (GCC) (http://gcc.gnu.org) – это инструментальное средство разработки программ на языках Си, Си++, Фортран и других. GCC включает в себя:

- Препроцессоры программ на языках Си и Си++.
- Компиляторы для поддерживаемых языков. В мире Unix под компилятором (в узком смысле) понимается программа, выдающая в качестве результата текст программы на языке ассемблера.
- Стандартные библиотеки языков Си++ и других (кроме Си).

- Программы-драйверы компиляции, которые предоставляют универсальный интерфейс командной строки ко всем компонентам GCC и связанным с ними системным утилитам. Например, программа gcc позволяет управлять компиляцией программ на Си, g++ - компиляцией программ на Си++ и т. д.

В состав GCC не входят:

- Ассемблер (GNU Assembler, команда as), компоновщик (GNU linker, команда ld[1]) и некоторые другие утилиты для работы с объектными и исполняемыми файлами. В Linux они находятся в инсталляционном пакете binutils.
- Заголовочные файлы и объектные модули стандартной библиотеки языка Си. В Linux они находятся в инсталляционных пакетах glibc, glibc-devel, glibc-static.

Тем не менее, они необходимы для компиляции программ на Си, ввиду чего будут рассмотрены наряду с инструментами GCC.

Команда запуска GCC для языка Си в общем виде выглядит следующим образом:

```
gcc <параметры>
```

, где в параметрах могут идти вперемешку имена входных файлов для компиляции и опции, управляющие компиляцией. В дальнейших разделах использование gcc описывается более подробно.

Схема трансляции программ, написанных на Си

Рассмотрим схему трансляции программы на языке Си, которая традиционно используется в системах Unix.

Трансляция программы состоит из следующих этапов:

- препроцессирование;
- трансляция в ассемблер;
- ассемблирование;
- компоновка.

Традиционно исходные файлы программы на языке Си имеют суффикс имени файла .c, заголовочные файлы для программы на Си имеют суффикс .h. В файловых системах Unix регистр букв значим,

[1] Название команды происходит от английского слова «load» - «загрузка».

и если, например, имя файла имеет суффикс .C, такой файл считается содержащим текст программы на языке Си++, и будет компилироваться компилятором языка Си++, а не Си.

Препроцессирование. Препроцессор просматривает входной .c файл, исполняет содержащиеся в нём директивы препроцессора, в частности, включает в него содержимое других файлов, указанных в директивах #include.

Файл-результат препроцессирования не содержит директив препроцессора, не раскрытых макросов, вместо директив #include в файл-результат подставлено содержимое соответствующих заголовочных файлов. Файл с результатом препроцессирования обычно имеет суффикс .i, однако после завершения трансляции все промежуточные временные файлы по умолчанию удаляются, поэтому чтобы увидеть результат препроцессирования (что, например, бывает полезно при отладке ошибок, связанных с небрежным использованием макросов) нужно использовать опцию -E командной строки gcc. Результат препроцессирования называется *единицей трансляции (англ., translation unit)* или *единицей компиляции (англ., compilation unit)*

Трансляция в ассемблер. На вход подаётся одна единица трансляции, а на выходе (при отсутствии синтаксических и семантических ошибок) выдаётся файл на языке ассемблера для (как правило) машины, на которой ведётся трансляция. Файл с оттранслированной программой на языке ассемблера имеет суффикс имени .s, но точно так же, как и результат работы препроцессора, он по умолчанию удаляется.

Ассемблирование. На этой стадии работает ассемблер. Он получает на входе результат работы предыдущей стадии и генерирует на выходе объектный файл. Объектные файлы в UNIX имеют суффикс .o.

Компоновка. Компоновщик получает на вход набор объектных файлов, соответствующим единицам трансляции, составляющим программу, подключает к ним стандартную библиотеку языка Си и библиотеки, указанные пользователем, и на выходе получает исполняемую программу.

Запуск транслятора gcc

Рассмотрим основные возможности транслятора GNU Си. В командной строке задаётся список файлов для обработки. Какие операции необходимо выполнить с файлами – зависит от суффикса имен файлов. Возможные суффиксы перечислены в таблице ниже. Если имя файла имеет нераспознанный суффикс, это имя передаётся компоновщику.

Суффикс имени файла	Выполняемые действия
.h	Заголовочный файл на языке Си. Не должен использоваться в аргументах команды gcc. Попытка трансляции такого файла вызывает сообщение об ошибке.
.c	Файл на языке Си. Выполняется препроцессирование, трансляция, ассемблирование и компоновка.
.i	Препроцессированный файл на языке Си. Выполняется трансляция, ассемблирование и компоновка.
.s	Файл на языке ассемблера. Выполняется ассемблирование и компоновка.
.S	Файл на языке ассемблера. Выполняется препроцессирование, ассемблирование и компоновка.
.o	Объектный файл. Выполняется компоновка.
.a	Файл статической библиотеки. Выполняется компоновка.

Действия по трансляции файла определяются для каждого указанного в командной строке файла индивидуально. Например, если в командной строке указаны имена файлов 1.c и 2.o, то для первого файла будут выполнены все шаги трансляции, а для второго – только компоновка. Исполняемый файл будет содержать результат трансляции первого файла, скомпонованный со вторым файлом и стандартными библиотеками.

Пользователь может явно задать, на какой фазе нужно остановиться. По умолчанию транслятор пытается выполнить все необходимые фазы, включая компоновку программы. Конечная фаза трансляции программы определяется для всех транслируемых за один вызов gcc файлов указанием одной из опций, перечисленных в таблице.

Опция	Описание
-E	Остановиться после препроцессирования. Результат работы препроцессора выводится по умолчанию на стандартный поток вывода. Имя выходного файла можно указать с помощью опции -o. При этом если в командной строке указано несколько файлов, то в выходной файл будет помещён результат препроцессирования последнего файла.
-S	Остановиться после трансляции в ассемблер. По умолчанию имя выходного файла получается из имени входного файла заменой суффикса .c или .i на суффикс .s. Явное имя выходного файла можно указать с помощью опции -o. Попытка использования опции -o и нескольких имён входных файлов вызывает сообщение об ошибке.
-c	Остановиться после ассемблирования. По умолчанию имя выходного файла получается из имени входного файла заменой суффикса его имени на суффикс .o. Явное имя выходного файла можно указать с помощью опции -o, которая несовместима с указанием одновременно нескольких транслируемых файлов.
	Если ни одной из перечисленных выше опций не задано, выполняются все стадии трансляции. Имя выходного файла по умолчанию равно a.out, но может быть изменено с помощью опции -o.
-o	Позволяет задать явное имя выходного файла для любой стадии трансляции.

Например, командная строка

```
gcc 1.c 2.c -o 1
```

транслирует два файла на языке Си, объединяя их в одну программу с именем 1.

Командная строка

```
gcc 3.o 4.o -o 3 -lm
```

компонует два объектных файла, добавляя к ним стандартную библиотеку языка Си и стандартную математическую библиотеку (опция -lm), и помещает результат в исполняемый файл с именем 3.

Прочие полезные опции транслятора gcc перечислены в таблице.

Опция	Описание
-I PATH	Добавляет каталог PATH в начало списка каталогов, которые просматриваются препроцессором при поиске файлов, подключаемых директивой #include. В командной строке может быть указано несколько опций -I, тогда каталоги просматриваются в порядке, в котором они указаны в командной строке.
-D NAME	Определяет макрос с именем NAME, который получает значение 1.
-D NAME=VALUE	Определяет макрос с именем NAME, который получает заданное значение VALUE.
-Wall	Включает выдачу большого количества предупреждающих сообщений, которые по умолчанию не выдаются. Опция должна использоваться при компиляции программ, все предупреждающие сообщения компилятора должны быть внимательно проанализированы, поскольку могут указывать на ошибки в программе.
-g	Включает генерацию отладочной информации в исполняемую программу. Наличие отладочной информации позволяет отлаживать программу в терминах исходного языка, а не машинного кода.
-O2	Включает большинство оптимизаций программы, которые одновременно уменьшают размер программы и увеличивают скорость её

	выполнения.
-L PATH	Добавляет путь PATH в начало списка каталогов, которые просматриваются редактором связей при поиске библиотек, указанных с помощью опции -L. Если в командной строке указано несколько опций -L, они добавляются в том же порядке, в котором указаны в командной строке.
-lname	Добавляет библиотеку name к списку библиотек, которые участвуют в компоновке программы (обратите внимание на отсутствие пробела между опцией и именем библиотеки). В системах Unix редактор связей просматривает библиотеки *один раз*, поэтому неправильный порядок задания библиотек может привести к тому, что некоторые имена останутся неопределёнными, и компиляция завершится с ошибкой. Файл, хранящий библиотеку с именем name, называется libname.a, если библиотека статическая, и libname.so, если библиотека динамическая.
-static	Указывает, что при компоновке не должны использоваться динамические библиотеки. Реализации всех используемых в программе функций будут добавлены непосредственно в исполняемый файл. Размер исполняемого файла программы может вырасти на сотни килобайт, зато такая программа перестанет быть зависимой от динамических библиотек. На некоторых системах могут отлаживаться только статически скомпонованные программы.

Использование стандартных библиотек языка Си

В языках Си и Си++ библиотеки состоят из двух частей:

- Заголовочных файлов, содержащих объявления типов данных, констант, прототипов функций и внешних переменных, которые подключаются к исходным файлам на этапе препроцессирования, формируя единицы трансляции.
- Файлов реализации, содержащих тела функций и определения переменных, которые подключаются к программе на этапе

компоновки (в случае статических библиотек) или на этапе выполнения (в случае динамических библиотек).

Заголовочные файлы стандартной библиотеки находятся в каталоге /usr/include и его подкаталогах, например, /usr/include/stdio.h или /usr/include/sys/types.h. Программа-драйвер gcc автоматически добавляет этот каталог в список для поиска заголовочных файлов, поэтому каталог /usr/include не нужно задавать в опции –I.

Файлы динамических библиотек размещаются в каталоге /lib или /usr/lib, а файлы статических библиотек – в каталоге /usr/lib. Они задаются автоматически и опция –L для них не нужна. Файл динамической библиотеки языка Си называется libc.so и полный путь к нему – /lib/libc.so.

Таким образом, если выписать явно пути и библиотеки, задаваемые при компиляции программы на Си с помощью gcc неявно, мы получим примерно следующую командную строку:

```
gcc -I/usr/include -L/lib -L/usr/lib jeltz.c –lc
```

Исключением являются математические функции стандартной библиотеки Си, объявленные в заголовочном файле <math.h>, например, sin. Их реализации вынесены в отдельную библиотеку libm.so (libm.a), которая не указывается в списке подключаемых библиотек по умолчанию. Для компоновки программ, использующих математические функции, необходимо в командной строке gcc указать опцию -lm:

```
gcc -Wall -O2 marvin.c -omarvin –lm
```

Для того, чтобы увидеть все пути, передаваемые драйвером компиляции gcc препроцессору, компилятору, ассемблеру и компоновщику, можно использовать опцию –v:

```
gcc -g -O0 -v prosser.c -o prosser
```

Компоновка программы

Если исполняемая программа компонуется из нескольких единиц трансляции, компоновщик использует свои правила видимости имён, которые приведены ниже:

- Все имена, объявленные с классом памяти static, видимы только в пределах своей единицы трансляции и не влияют на компоновку.
- Если некоторая единица трансляции использует внешнее имя (переменной или функции), которое не определено ни в какой единице трансляции, выдаётся сообщение об ошибке.
- Если несколько единиц трансляции определяют нестатическую функцию с одним и тем же именем, выдаётся сообщение об ошибке.
- Если некоторое нестатическое имя определяется и как переменная, и как функция, выдаётся сообщение об ошибке.
- Если несколько единиц трансляции определяют нестатическую инициализированную переменную с одним и тем же именем, выдаётся сообщение об ошибке.
- Если несколько единиц трансляции определяют переменную с одним и тем же именем, которая инициализируется не более чем в одной единице трансляции, все определения размещаются, начиная с одного адреса.

Последнее правило можно продемонстрировать на следующем примере. Предположим, что в трёх файлах определена переменная var следующим образом:

Первый файл:	Второй файл:	Третий файл

`int var = 1;`
`int add1(void)`
`{`
`  return var++;`
`}`
	`int var;`
	`int add2(void)`
	`{`
	`  return var += 2;`
	`}`
		`int var;`
		`int add3(void)`
		`{`
		`  return var+=3;`
		`}`

Если все три единицы компиляции объединяются в одну программу, то переменная var каждого из трёх файлов будет располагаться по одному и тому же адресу, и каждая из трёх функций будет работать, по сути, с общей переменной. Чтобы предотвратить такое слияние переменных можно использовать явную инициализацию переменной var, тогда компоновщик выдаст сообщение об ошибке как показано ниже.

Первый файл:	Второй файл:	Третий файл

`int var = 1;`
`int add1(void)`
`{`
`  return var++;`
`}`
	`int var = 1;`
	`int add2(void)`
	`{`
	`  return var += 2;`
	`}`
		`int var = 1;`
		`int add3(void)`
		`{`
		`  return var+=3;`
		`}`

Видимость глобальных переменных при компоновке можно также регулировать с помощью спецификаторов класса памяти extern и static. Спецификатор extern запрещает компилятору выделять память под переменную в данном модуле. Спецификатор static локализует область видимости переменной единицей трансляции, в которой она определена.

Программы из нескольких единиц трансляции

Как правило, сложные программы состоят из нескольких исходных файлов, которые объединяются компоновщиком. При написании таких программ полезно следовать следующим рекомендациям.

При группировке функций и переменных по исходным файлам логически сильно связанные функции объединяются в один исходный файл. Данная рекомендация соответствует способу реализации на Си парадигмы *модульного программирования*, предполагающего разбиение программы на независимые части – *модули*. Например, если речь идет о программе-редакторе табличных данных, функции обеспечивающие сохранение таблицы в файл, могут быть помещены в один исходный файл, функции, которые выводят на экран содержимое таблицы, – в другой файл, функции, которые анализируют ввод пользователя, – в третий.

Чем больше переменных объявлено в единице компиляции с классом памяти static вместо класса памяти по умолчанию, тем лучше. Программа может быть легче модифицирована, если доступ к данным всегда происходит с помощью вызовов функций. Чем меньше «чужих» переменных использует некоторая единица компиляции, тем она проще для понимания.

Для каждого .c файла должен существовать интерфейсный файл с тем же именем, но суффиксом .h, в котором определяются переменные, функции, типы данных и пр., которые могут использоваться извне данной единицы компиляции.

Исходный .c файл должен обязательно подключать свой собственный .h файл. В этом случае транслятор обнаружит рассогласования между объявлениями в .h-файлах и определениями в .c файле.

Интерфейсный .h файл должен быть обязательно защищён от повторного включения т.н. *стражем включения (англ., include guard)*:

```
#ifndef __NAME_H__
#define __NAME_H__
<здесь находится текст файла>
#endif
```

Здесь NAME – это имя файла (без суффикса). Некоторые .h-файлы могут включать другие .h файлы, поэтому, когда программа становится большой, человек не может отследить, какие файлы уже включались, а какие – ещё нет. При использовании стражей включения можно не задумываясь включать в .c файл все заголовочные файлы, необходимые данной единице компиляции. Защита от повторного включения предотвращает появление ошибок о переопределённых типах, переменных и функциях.

В заголовочном файле помещаются макроопределения и типы данных, являющиеся интерфейсом данной единицы компиляции, то есть необходимые для использования функций и переменных этой единицы компиляции. С классом памяти extern помещаются необходимые переменные и прототипы функций, объявленные в соответствующей единице компиляции.

В заголовочный файл никогда не помещаются определения переменных с классом памяти, отличным от класса extern, и тела функций. В заголовочный файл никогда не помещаются прототипы функций с классом памяти static. Если некоторый тип или константа используются только в теле какой-либо функций и не требуется для корректного использования функциональности данной единицы

компиляции другими модулями, этот тип или константа также не помещаются в заголовочный файл.

Отладка в The GNU Project Debugger

Отладчик (англ., debugger) – это программа, позволяющая контролировать выполнение другой (отлаживаемой) программы. Отладчик позволяет выполнять программу по шагам (пошаговая отладка), просматривать значения ячеек памяти, устанавливать точки останова и т. д.

Отладчик, разработанный в рамках проекта GNU, называется The GNU Project Debugger (GDB) (http://www.gnu.org/s/gdb/). Он будет рассмотрен в данном разделе.

Отладчик —инструмент, к которому следует прибегать, когда другие способы найти ошибку в программе (например, просмотр исходных текстов, отладочная печать, изучение логов работы программы) не дали результатов. Если при разработке программы использовано автоматическое тестирование (например, модульное тестирование), и реализована подсистема детального логирования, потребность в использовании отладчика уменьшается. В некоторых ситуациях, например, при поиске ошибок в «боевой» системе, скомпилированной без отладочной информации, отладка программы с помощью отладчика вообще невозможна. Тем не менее, отладчик является популярным средством поиска ошибок и умение им пользоваться необходимо любому квалифицированному разработчику программного обеспечения.

Можно выделить три основных способа использования отладчика:

- запуск исследуемой программы из-под отладчика;
- подключение отладчика к работающей программе;
- «посмертная» отладка.

В первом случае сначала запускается отладчик, настраиваются параметры запускаемой программы, затем отлаживаемая программа запускается под контролем отладчика.

Во втором случае отладчик подключается к работающей программе (процессу), которая приостанавливается в некоторой

точке. После завершения сеанса отладчик отключается от программы, и программа продолжает выполнение как обычно.

При «посмертной» отладке исследуется т.н. файл *дампа памяти (англ., core dump*) программы. Файл дампа памяти содержит значения ячеек памяти адресного пространства процесса, он создается ядром операционной системы (ОС), когда программа получает сигнал какой-либо «фатальной» ошибки. При посмертной отладке процесса отлаживаемой программы уже не существует, поэтому возможности отладчика ограничены, тем не менее такая форма отладки может быть полезна.

С помощью отладчика можно отлаживать любой исполняемый файл в терминах адресов ОЗУ, регистров ЦП и инструкций процессора, но удобнее вести отладку в терминах языка высокого уровня (ЯВУ), то есть в терминах операторов ЯВУ, переменных и т. п. Для этого в скомпилированной программе должна присутствовать отладочная информация. У компилятора GCC за подключение отладочной информации отвечает опция -g. Например,

```
gcc -g prog.c -oprog
```

программа в файле marvin.c будет скомпилирована в исполняемый файл marvin с отладочной информацией.

Рассмотрим подробнее три сценария использования отладчика.

Запуск исследуемой программы из-под отладчика

При запуске отладчика указывается только исполняемый файл с запускаемой программой.

```
gdb prog
```

Исполняемый файл prog будет загружен, но не будет запущен. Отладчик GDB выдаст приглашение ко вводу команды. Перед запуском программы на выполнение можно задавать точки останова, отслеживаемые переменные. Для запуска программы надо выполнить команду отладчика run. Примерный сценарий работы приведен ниже. Команды, вводимые пользователем, выделены полужирным шрифтом. Комментарии к вводимым командам выделены курсивом.

```
$ gcc -g prog.c -oprog     компилируем с отладочной информацией
```

```
$ gdb prog                  запускаем отладчик
GNU gdb (GDB) Fedora (7.2-51.fc14)
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and
redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/cher/gdb/prog...done.
(gdb) b main                устанавливаем точку останова на
функцию  main
Breakpoint 1 at 0x80483bd: file prog.c, line 5.
(gdb) run                   запускаем программу на
выполнение
Starting program: /home/cher/gdb/prog
Breakpoint 1, main () at prog.c:5
5               printf("Hello, I'm Eddie!\n");
(gdb)                       выполнение приостановлено в
начале функцию main
```

Подключение отладчика к работающей программе

Иногда предположение о наличии ошибки в программе возникает когда программа уже запущена и работает существенное время, а перезапуск программы с начала нецелесообразен. Например, вычислительная программа работает дольше ожидаемого времени и возникло подозрение, что она зациклилась, или серверная программа работает не так, как ожидается. В этих случаях можно подключиться к работающей программе отладчиком. Отладчик GDB в этом случае запускается с двумя аргументами: первый — это имя исполняемого файла, второй аргумент — число – идентификатор процесса (PID). Примерный сценарий работы с отладчиком приведен ниже.

```
$ gdb ./ej-users 7096       программа называется ej-users и
работает как процесс 7096
GNU gdb (GDB) Fedora (7.2-51.fc14)
Copyright (C) 2010 Free Software Foundation, Inc.
```

```
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and
redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/cher/ejudge/ej-users...done.
Attaching to program: /home/cher/ejudge/ej-users, process
7096
0xb78c1424 in __kernel_vsyscall ()    выполнение
программы приостановлено в этой точке
(gdb) bt              печатаем стек вызовов функций
#0  0xb78c1424 in __kernel_vsyscall ()
#1  0x00a8788d in ___newselect_nocancel ()
    at ../sysdeps/unix/syscall-template.S:82
#2  0x0806e502 in do_work () at userlist-server.c:10244
#3  0x0806fee5 in main (argc=5, argv=0xbfb06754) at
userlist-server.c:10816
(gdb) up 2            поднимаемся на два уровня вверх по
стеку вызовов
#2  0x0806e502 in do_work () at userlist-server.c:10244
10244       val = select(max_fd, &rset, &wset, NULL,
&timeout);
(gdb) p max_fd        печатаем значение переменной max_fd
$1 = 7                значение переменной равно 7
(gdb) quit            завершаем отладку
A debugging session is active.

        Inferior 1 [process 7096] will be detached.

Quit anyway? (y or n) y   подтверждаем завершение
отладки, программа продолжит выполнение
Detaching from program: /home/cher/ejudge/ej-users,
process 7096
$
```

«Посмертная» отладка

После аварийного завершения программы, например, в результате попытки доступа к области памяти, не принадлежащей процессу, либо при делении на 0, либо при вызове функции abort(),

еще остается возможность узнать состояние, в котором находилась программа в момент ошибки.

Для этого при аварийном завершении программы ядром ОС должен быть сгенерирован файл дампа памяти аварийно-завершившейся программы (так называемый core-файл). Создание core-файлов может быть по умолчанию отключено. Например,

```
$ ./prog                    запускаем программу
Hello, I'm Eddie!
Aborted                     выполнение программы аварийно
завершилось, но core-файла нет
$
```

Для того, чтобы разрешить создание core-файлов нужно снять ограничение на максимальный размер core-файлов.

```
$ ulimit -c unlimited
```

Тогда работа программы может выглядеть примерно так:

```
$ ./prog                    запускаем программу
Hello, I'm Eddie!
Aborted (core dumped)
$ ls                        смотрим список файлов в текущем
каталоге
core.7911  prog  prog.c     появился core-файл core.7911
$
```

В данном примере имя core-файла включает в себя PID процесса, при завершении которого был создан core-файл, но иногда такой файл называется просто core.

Для посмертной отладки GDB запускается с двумя аргументами: первый — это имя исполняемого файла, а второй — это имя core-файла. Примерный сценарий работы с отладчиком в этом случае приведен ниже.

```
$ gdb ./prog core.7911      запускаем посмертную отладку
GNU gdb (GDB) Fedora (7.2-51.fc14)
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and
redistribute it.
```

```
There is NO WARRANTY, to the extent permitted by law.
Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/cher/gdb/prog...done.
[New Thread 7911]
Core was generated by `./prog'.
Program terminated with signal 6, Aborted.
#0  0xb782a424 in __kernel_vsyscall ()
(gdb) bt                          просматриваем стек вызовов
#0  0xb782a424 in __kernel_vsyscall ()
#1  0x009e32f1 in raise (sig=6) at
../nptl/sysdeps/unix/sysv/linux/raise.c:64
#2  0x009e4d5e in abort () at abort.c:92
#3  0x080483fe in main () at prog.c:6
(gdb) quit                        завершаем отладку
$
```

Основные команды отладчика

Ниже описываются основные команды отладчика GDB. Команды вводятся с консоли после отображения отладчиком “(gdb)” – строки-приглашения ко вводу очередной команды.

quit – завершение работы отладчика. Если программа запускалась из-под отладчика (первый способ отладки), то вместе с завершением отладки завершится и программа. Если отладчик подключался к работающей программе, то программа продолжит выполнение в обычном режиме. Если отладчик запускался для исследования core-файла, то отладчик просто завершается, так как никакого отлаживаемого процесса не существует. Вместо набора команды quit можно просто нажать комбинацию *Ctrl-D* – стандартная комбинация, обозначающая признак конца файла (конец ввода) в Unux.

run – запустить программу на выполнение. Команда доступна только, если программа еще не запущена. При подключении к работающему процессу или исследовании core-файлов данная команда не доступна. Допускается указывать аргументы командной строки и перенаправление потоков ввода-вывода, как если бы программа запускалась командным процессором. Пример:

```
(gdb) run < 001.in > 001.out
```

Здесь программа запускается на выполнение с файлом 001.in, перенаправленным на стандартный поток ввода, и файлом 001.out, в который перенаправляется вывод на стандартный поток вывода.

Выполнение программы может быть прервано в любой момент нажатием на комбинацию *Ctrl-C*. Отладчик приостановит выполнение программы и выдаст приглашение ко вводу очередной команды отладчика.

Выполнение программы также приостанавливается при получении программой любого сигнала, при завершении программы либо достижении точки останова.

bt (или *backtrace*) – распечатать стек вызовов. Стек вызовов печатается от самой вложенной функции к функции main. Для каждого стекового фрейма печатается адрес в коде точки вызова, название функции и параметры, переданные в функцию, а также позиция в исходном коде. Команда *bt full* дополнительно печатает значения локальных переменных.

```
(gdb) bt full
#0  0xb78c1424 in __kernel_vsyscall ()
No symbol table info available.
#1  0x00a8788d in ___newselect_nocancel ()
    at ../sysdeps/unix/syscall-template.S:82
No locals.
#2  0x0806e502 in do_work () at userlist-server.c:10244
        addr = {sun_family = 1,
          sun_path = "\000\060\060\060\071b", '\000'
<repeats 101 times>}
        val = 0
        max_fd = 7
        timeout = {tv_sec = 0, tv_usec = 702429}
        rset = {fds_bits = {80, 0 <repeats 31 times>}}
        wset = {fds_bits = {0 <repeats 32 times>}}
        p = 0x0
        q = 0x9286e68
        saved_fd = 6
```

up – переход на указанное количество фреймов вверх по стеку вызовов функций. Если аргумент у команды не указан, он

принимается равным 1 (переход на один фрейм вверх, то есть переход к функции, которая вызвала текущую функцию).

down – переход на указанное количество фреймов вниз по стеку вызовов функций. Если аргумент у команды не указан, он принимается равным 1 (переход на один фрейм вниз, то есть переход к функции, которая была вызвана в текущей точке текущей функции).

info frame – получить информацию о текущем стековом фрейме.

info locals – получить информацию о значениях локальных переменных текущего стекового фрейма.

p (или *print*) – напечатать значение выражения. Аргументом команды может быть почти произвольное выражение языка Си, даже включающее в себя вызовы функций программы, если, конечно, отлаживаемый процесс существует. Таким образом, вызовы функций недоступны при «посмертной» отладке. Если аргумент команды не указан, берется аргумент, который был указан в команде *p* в последний раз.

```
(gdb) p user              печатаем значение переменной user
$1 = (const unsigned char *) 0x0
(gdb) p                   еще раз печатаем значение переменной user
$2 = (const unsigned char *) 0x0
(gdb)
```

l (или *list*) – напечатать исходный код. Команду можно использовать во многих вариантах, часть из которых перечислена ниже.

```
(gdb) l                   напечатать очередные 10 строк исходного файла
(gdb) l -                 напечатать предыдущие 10 строк исходного файла
(gdb) l 200               напечатать 10 строк в окрестности 200 строки текущего
файла
(gdb) l prog.c:200        напечатать 10 строк в окрестности 200 строки файла prog.c
(gdb) l main              напечатать 10 строк в окрестности начала функции main
(gdb) l *0x0806e502       напечатать 10 строк в окрестности кода по указанному адресу
```

b (или *break*) – установка точки останова. Параметром команды является точка в программе, помечаемая как точка останова. Команду можно использовать во многих вариантах, часть из которых перечислена ниже.

(gdb) **b main**	*установить точку останова в начале функции main*
(gdb) **b 200**	*установить точку останова на 200 строке текущего файла*
(gdb) **b prog.c:200**	*установить точку останова на 200 строке файла prog.c*
(gdb) **b *0x0806e502**	*установить точку останова по указанному адресу*

c (или *continue*) – продолжить выполнение программы.

finish — продолжить выполнение программы до достижения конца текущей функции

n (или *next*) — сделать указанное количество шагов выполнения (по умолчанию 1). Вызовы функций рассматриваются как одна инструкция, то есть вызванные функции выполняются в обычном, а не пошаговом режиме.

s (или *step*) — сделать указанное количество шагов выполнения программы (по умолчанию 1). При вызовах функций выполнение входит в функции в пошаговом режиме.

Сборка программы с помощью GNU Make

Программа make предназначена для отслеживания зависимостей одних файлов от других файлов, выявления «устаревших» файлов при помощи сравнения времен модификации файлов и выполнения команд для «обновления» устаревших файлов. Make является представителем *систем автоматизации сборки (англ., build automation system)*, позволяющих автоматизировать процессы трансляции, компоновки, запуска модульных тестов и развертывания системы за счет описания соответствующих сценариев на специальном языке.

Сценарии Make описываются в т.н. *файле проекта*. *Проектом* называется совокупность файлов, зависящих друг от друга. Файл описания проекта перечисляет зависимости между файлами и задает команды для обновления зависимых файлов. Имя файла описания проекта задается опцией –f командной строки программы make и по умолчанию предполагается равным Makefile или makefile. Если имя файла проекта явно не задано, при запуске утилита ищет в текущем каталоге файл с указанными выше именами, и, если такой файл существует, выполняет команды из него.

Таким образом, в простейшей форме команда

```
make
```

по описанию проекта в файле Makefile или makefile определяет, какие файлы устарели и нуждаются в обновлении и запускает соответствующие команды.

Форма описания проектов в файле Makefile достаточно сильно варьируется от версии к версии. Утилита make на системах FreeBSD плохо совместима с утилитой GNU Make, которая на FreeBSD обычно называется gmake. В данном разделе будет рассматриваться именно GNU make (http://www.gnu.org/software/make/).

Одно из применений утилиты make – отслеживание зависимостей между исходными и объектными файлами в проектах на языке Си или Си++ и управление компиляцией программы – то есть сборка проекта. Использование make с этой целью рассматривается в данном разделе.

Обычно программы на языках Си или Си++ представляют собой совокупность нескольких .c (.cpp) файлов с реализациями функций и .h файлов с прототипами функций и определениями типов данных. Как правило, каждому .c файлу соответствует .h файл с тем же именем.

Предположим, что разрабатываемая программа называется earth и состоит из файлов arthur.c, arthur.h, trillian.c, trillian.h, prosser.c, prosser.h. Разработка программы ведется в POSIX-среде с использованием компилятора GCC.

Простейший способ скомпилировать программу — указать все исходные .c файлы в командной строке gcc:

```
gcc arthur.c trillian.c prosser.c -o earth
```

Компилятор gcc выполнит все этапы компиляции исходных файлов программы и компоновку исполняемого файла earth. Обратите внимание, что в командной строке gcc указываются только .c файлы и никогда не указываются .h файлы.

Компиляция и компоновка при помощи перечисления всех исходных файлов в аргументах командной строки GCC допустима лишь для совсем простых программ. С ростом числа исходных файлов ситуация очень быстро становится неуправляемой. Кроме того, каждый раз все исходные файлы будут компилироваться от начала до конца, что в случае больших проектов занимает много

времени. Поэтому обычно компиляция программы выолняется в два этапа: компиляция объектных файлов и компоновка исполняемой программы из объектных файлов. Каждому .c файлу теперь соответствует объектный файл, имя которого в POSIX-системах имеет суффикс .o. Таким образом, в рассматриваемом случае программа earth компонуется из объектных файлов arthur.o, trillian.o и prosser.o следующей командой:

```
gcc arthur.o trillian.o prosser.o -o earth
```

Каждый объектный файл должен быть получен из соответствующего исходного файла следующей командой:

```
gcc -c arthur.c
```

Обратите внимание, что явно задавать имя выходного файла необязательно. Оно будет получено из имени компилируемого файла заменой суффикса .c на суффикс .o. Итак, для компиляции программы earth теперь необходимо выполнить четыре команды:

```
gcc -c arthur.c
gcc -c trillian.c
gcc -c prosser.c
gcc arthur.o trillian.o prosser.o -o earth
```

Хотя теперь для компиляции программы необходимо выполнить четыре команды вместо одной, взамен получаются следующие преимущества:

- если изменение внесено в один файл, например, в файл prosser.c, нет необходимости перекомпилировать файлы trillian.o или arthur.o; достаточно перекомпилировать файл prosser.o, а затем выполнить компоновку программы earth;
- компиляция объектных файлов arthur.o, trillian.o и prosser.o не зависит друг от друга, поэтому может выполняться параллельно на многопроцессорном (многоядерном) компьютере.

В случае нескольких исходных .c и .h файлов и соответствующих промежуточных .o файлов отслеживать, какой файл нуждается в перекомпиляции, становится сложно, и здесь на помощь приходит программа make. По описанию файлов и команд для компиляции программа make определяет, какие файлы

нуждаются в перекомпиляции, и может выполнять перекомпиляцию независимых файлов параллельно.

Файл A *зависит* от файла B, если для получения файла A необходимо выполнить некоторую команду над файлом B. Можно сказать, что в программе существует зависимость файла A от файла B. В нашем случае файл arthur.o зависит от файла arthur.c, а файл earth зависит от файлов arthur.o, trillian.o и prosser.o. Можно сказать, что файл earth транзитивно зависит от файла arthur.c.

Зависимость файла A от файла B называется *удовлетворенной*, если:

- все зависимости файла B от других файлов удовлетворены;
- файл A существует в файловой системе;
- файл A имеет дату последней модификации не раньше даты последней модификации файла B.

Если все зависимости файла A удовлетворены, то файл A не нуждается в перекомпиляции. В противном случае сначала удовлетворяются все зависимости файла B, а затем выполняется команда перекомпиляции файла A.

Например, если программа earth компилируется в первый раз, то в файловой системе не существует ни файла earth, ни объектных файлов arthur.o, trillian.o, prosser.o. Это значит, что зависимости файла earth от объектных файлов, а также зависимости объектных файлов от .c файлов не удовлетворены, то есть все они должны быть перекомпилированы. В результате в файловой системе появятся файлы arthur.o, trillian.o, prosser.o, даты последней модификации которых будут больше дат последней модификации соответствующих .c файлов (в предположении, что часы на компьютере идут правильно, и что в файловой системе нет файлов «из будущего»). Затем будет создан файл earth, дата последней модификации которого будет больше даты последней модификации объектных файлов.

В получившейся конфигурации все зависимости всех файлов друг от друга удовлетворены, и поэтому для компиляции программы earth не нужно выполнять никаких команд.

Предположим теперь, что в процессе разработки был изменен файл prosser.c. Его время последнего изменения теперь больше времени последнего изменения файла prosser.o. Зависимость prosser.o от prosser.c становится неудовлетворенной, и, как следствие, зависимость earth от prosser.o также становится неудовлетворенной. Чтобы удовлетворить зависимости необходимо перекомпилировать файл prosser.o, а затем файл earth. Файлы arthur.o и trillian.o можно не трогать, так как зависимости этих файлов от соответствующих .c файлов удовлетворены.

Такова общая идея работы программы make и, на самом деле, всех программ управления сборкой проекта: ant (http://ant.apache.org/), scons (http://www.scons.org/) и др.

Хотя утилита make присутствует во всех системах программирования, вид управляющего файла или набор опций командной строки могут сильно различаться. Далее будет рассматриваться командный язык и опции командной строки программы GNU make. В дистрибутивах операционной системы Linux программа называется make. В BSD, как правило, программа GNU make доступна под именем gmake.

Файл описания проекта может содержать описания переменных, описания зависимостей и описания команд, которые используются для компиляции. Каждый элемент файла описания проекта должен, как правило, располагаться на отдельной строке. Для размещения элемента описания проекта на нескольких строках используется символ продолжения \ точно так же, как в директивах препроцессора языка Си.

Определения переменных записываются следующим образом:

```
<имя> = <определение>
```

Использование переменной записывается в одной из двух форм:

```
$(<имя>)
```

или

```
${<имя>}
```

Эти формы равнозначны.

Переменные рассматриваются как макросы, то есть использование переменной означает подстановку текста из определения переменной в точку использования. Если при определении переменной были использованы другие переменные, подстановка их значений происходит при использовании переменной (опять так же, как и в препроцессоре языка Си).

Зависимости между компонентами определяются следующим образом:

```
<цель> : <цель1> <цель2> ... <цельn>
```

Где <цель> - имя цели, которое может быть либо именем файла, либо некоторым именем, обозначающим действие, которому не соответствует никакой файл, например clean. Список целей в правой части задает цели, от которых зависит <цель>.

Если описание проекта содержит циклическую зависимость, то есть, например, файл A зависит от файла B, а файл B зависит от файла A, такое описание проекта является ошибочным.

Команды для перекомпиляции цели записываются после описания зависимости. Каждая команда должна начинаться с символа табуляции (\t). Если ни одной команды для перекомпиляции цели не задано, будут использоваться стандартные правила, если таковые имеются. Для определения, каким стандартным правилом необходимо воспользоваться, обычно используются суффиксы имен файлов. Если ни одна команда для перекомпиляции цели не задана и стандартное правило не найдено, программа make завершается с ошибкой.

Для программы earth простейший пример файла Makefile для компиляции проекта может иметь вид:

```
earth : arthur.o trillian.o prosser.o
     gcc arthur.o trillian.o prosser.o -o earth
arthur.o : arthur.c
     gcc -c arthur.c
trillian.o : trillian.c
     gcc -c trillian.c
prosser.o : prosser.c
     gcc -c prosser.c
```

Однако, в этом описании зависимостей не учтены .h файлы. Например, если файл arthur.h подключается в файлах arthur.c и trillian.c, то изменение файла arthur.h должно приводить к перекомпиляции как arthur.c, так и trillian.c. Получается, что .o файлы зависят не только от .c файлов, но и от .h файлов, которые включаются данными .c файлами непосредственно или косвенно. С учетом этого файл Makefile может приобрести следующий вид:

```
earth : arthur.o trillian.o prosser.o
     gcc arthur.o trillian.o prosser.o -o earth
arthur.o : arthur.c arthur.h
     gcc -c arthur.c
trillian.o : trillian.c trillian.h arthur.h
     gcc -c trillian.c
prosser.o : prosser.c prosser.h arthur.h
     gcc -c prosser.c
```

Первой в списке зависимостей обычно записывается «главная» зависимость, а затем записываются все остальные файлы-зависимости.

В командной строке программы make можно задать имя цели, которую требуется (при необходимости) перекомпилировать. Так, при запуске

```
make prosser.o
```

будет при необходимости перекомпилирован только файл prosser.o и те файлы, от которых он зависит, все прочие файлы затронуты не будут. Если в командной строке имя цели не указано, берется первая цель в файле. В нашем случае это будет цель earth.

Если придерживаться хорошего стиля написания Makefile, то каждый Makefile должен содержать как минимум два правила: all – основное правило, которое соответствует основному предназначению файла, и правило clean, которое предназначено для удаления всех рабочих файлов, создаваемых в процессе компиляции. В случае программы earth рабочими файлами можно считать сам исполняемый файл программы earth, а также все объектные файлы.

С учетом этих дополнений файл Makefile примет вид:

```
all : earth
earth : arthur.o trillian.o prosser.o
     gcc arthur.o trillian.o prosser.o -o earth
```

```
arthur.o : arthur.c arthur.h
     gcc -c arthur.c
trillian.o : trillian.c trillian.h arthur.h
     gcc -c trillian.c
prosser.o : prosser.c prosser.h arthur.h
     gcc -c prosser.c
clean :
     rm -f earth *.o
```

Обратите внимание, что у правила clean отсутствует список файлов, от которых этот файл зависит. Поскольку существование файла с именем clean в рабочем каталоге не предполагается, команда rm -f … будет выполняться каждый раз, когда make запускается на выполнение командой

```
make clean
```

Данный файл, безусловно, решает задачу автоматизации сборки программы earth. Теперь можно придать этому файлу более общий вид, чтобы в этот файл легче было вносить изменения.

Во-первых, можно параметризовать название используемого компилятора, а также предоставить возможность управлять параметрами командной строки компилятора. Для задания компилятора можно определить переменную CC, для задания опций командной командной строки компиляции объектных файлов — переменную CFLAGS, а для задания опций командной строки компоновки выходной программы — переменную LDFLAGS. Получим следующий файл:

```
CC = gcc
CFLAGS = -Wall -O2
LDFLAGS = -s
all : earth
earth : arthur.o trillian.o prosser.o
     $(CC) $(LDFLAGS) arthur.o trillian.o prosser.o -o earth
arthur.o : arthur.c arthur.h
     $(CC) $(CFLAGS) -c arthur.c
trillian.o : trillian.c trillian.h arthur.h
     $(CC) $(CFLAGS) -c trillian.c
prosser.o : prosser.c prosser.h arthur.h
     $(CC) $(CFLAGS) -c prosser.c
clean :
     rm -f earth *.o
```

Теперь можно изменить используемый компилятор, не только отредактировав Makefile, но и из командной строки. Например, запуск программы make в виде

```
make CC=icc
```

позволит для компиляции программы использовать не gcc, а Intel компилятор Си. Аналогично запуск

```
make CFLAGS=”-g” LDFLAGS=”-g”
```

позволит включить отладочную информацию в генерируемые объектные файлы и исполняемую программу.

Во-вторых, можно избавиться от дублирования имен файлов сначала в зависимостях, а потом в выполняемых командах. Для этого могут быть использованы специальные переменные $^, $< и $@. Переменная $@ раскрывается в имя цели, стоящей в левой части правила. Переменная $< раскрывается в имя первой зависимости в правой части правила. Переменная $^ раскрывается в список всех зависимостей в правой части. Правило для компиляции файла arthur.o приобретет следующий вид:

```
arthur.o : arthur.c arthur.h
     $(CC) $(CFLAGS) -c $<
```

Именно такое правило для компиляции .o файлов из .c файлов уже встроено в make, поэтому строку компиляции можно просто удалить. Останется следующий Makefile:

```
CC = gcc
CFLAGS = -Wall -O2
LDFLAGS = -s
all : earth
earth : arthur.o trillian.o prosser.o
     $(CC) $(LDFLAGS) $^ -o $@
arthur.o : arthur.c arthur.h
trillian.o : trillian.c trillian.h arthur.h
prosser.o : prosser.c prosser.h arthur.h
clean :
     rm -f earth *.o
```

При желании можно создавать новые шаблонные зависимости, то есть зависимости не конкретных файлов друг от друга, а файлов, имена которых удовлетворяют заданному шаблону. Тогда команды в зависимостях конкретных файлов также могут быть опущены.

Например, стандартное шаблонное правило для зависимостей .o файлов от .c файлов может быть определено следующим образом:

```
%.o : %.c:
	$(CC) -c $(CFLAGS) $<
```

Тем не менее, в этом файле проекта осталось слабое место. Оно связано с тем, что зависимости объектных файлов включают в себя помимо .c файлов и .h файлы, подключаемые .c файлами непосредственно или транзитивно. Представим себе, что в файл prosser.c была добавлена директива

```
#include "trillian.h"
```

но Makefile не был соответствующим образом изменен. Теперь может получиться так, что в файле trillian.h будет изменена некоторая структура данных, но файл prosser.o не будет перекомпилирован и код модуля prosser.o будет продолжать работать со старой версией структуры данных, в то время как остальная программа — с новой версией структуры данных. Такое расхождение в описании данных в рамках одной программы может привести к «загадочным» ошибкам при ее работе.

Хотелось бы каким-либо образом строить списки зависимостей объектных файлов от .c и .h файлов автоматически. Для этого мы воспользуемся специальными опциями компилятора gcc и расширенными возможностями GNU make.

Предположим, что автогенерируемые зависимости не находятся в самом файле Makefile, а подключаются из внешнего файла deps.make. Для подключения содержимого внешнего файла в Makefile необходимо добавить директиву

```
include deps.make
```

Для генерации файла deps.make с зависимостями воспользуемся опцией -MM компилятора gcc:

```
deps.make:arthur.c trillian.c prosser.c arthur.h trillian.h prosser.h
	gcc -MM arthur.c trillian.c prosser.c > deps.make
```

Файл deps.make зависит от всех .c и .h файлов, из которых собирается программа. Может показаться, что это правило не будет работать, так как в Makefile необходимо включить файл deps.make, для генерации которого необходимо выполнить Makefile, то есть

возникает циклическая зависимость, однако GNU make умеет корректно обрабатывать такие ситуации.

Для того, чтобы не выписывать списки .c и .h файлов несколько раз, в начале Makefile можно определить переменные:

```
CFILES = arthur.c trillian.c prosser.c
HFILES = arthur.h trillian.h prosser.h
```

Более того, список объектных файлов можно получать из списка .c файлов заменой суффикса .c на .o:

```
OBJECTS = $(CFILES:.c=.o)
```

В итоге получили следующий Makefile:

```
CC = gcc
CFLAGS = -Wall -O2
LDFLAGS = -s
CFILES = arthur.c trillian.c prosser.c
HFILES = arthur.h trillian.h prosser.h
OBJECTS = $(CFILES:.c=.o)
TARGET = earth
all : $(TARGET)
earth : $(OBJECTS)
     $(CC) $(LDFLAGS) $^ -o $@
include deps.make
deps.make : $(CFILES) $(HFILES)
     gcc -MM $(CFILES) > deps.make

clean :
     rm -f $(TARGET) *.o
```

Этот файл можно легко модифицировать для сборки других проектов с помощью изменения значений переменных CFILES, HFILES и TARGET.

Разработка в Code::Blocks

Интегрированная среда разработки (англ., integrated development environment, IDE) призвана объединить в общем графическом пользовательском интерфейсе инструменты разработки программного обеспечения: основные – такие как редактор исходных текстов, компилятор, компоновщик, отладчик и, во многих случаях, вспомогательные – средство автоматизации сборки,

профилировщик, среду для запуска модульных тестов, генератор документации и др.

Code::Blocks (http://www.codeblocks.org) – свободная кросс-платформенная интегрированная среда разработки, написанная на Си++ и поддерживаемая языки Си, Си++ и др. В состав Code::Blocks входит редактор исходных текстов с подсветкой синтаксиса и функцией *автозаполнения (англ., autocomplete)* – подсказки имен функций и переменных при наборе первых букв имени.

Исходные тексты, относящиеся к одной единице компоновки (исполняемый модуль, библиотека), составляют *проект (англ., project)* Code::Blocks. По каждому проекту могут заданы параметры трансляции и компоновки компилятором gcc для *отладочной (англ., Debug)* и *рабочей (англ., Release)* конфигураций. Несколько логически связанных проектов (например, проект-приложение с графическим пользовательским интерфейсом, проект-библиотека, содержащая бизнес-логику, и используемая приложением) могут быть объединены в общее *рабочее пространство (workspace)*. Для проектов рабочего пространства может быть задан порядок их сборки.

Одним из популярных сценариев использования интегрированной среды разработки является отладка. Code::Blocks предоставляет графический интерфейс к популярным функциям отладчика GDB: пошаговой отладке (включая подключение к работающему процессу), управлению контрольными точками, просмотру переменных, стека вызовов, значений ячеек памяти.

Функциональность Code::Blocks может быть расширена за счет установки дополнительных модулей.

Профилирование в Valgrind

Во многих случаях перед программистом встает проблема поиска в программе «узких» мест – фрагментов кода, негативно влияющих на производительность программы, а также ошибок, приводящих к нарушению функционирования программы из-за нерационального, а порой безконтрольного использования памяти. Конечно же, путем изучения исходных текстов, внесения в

программу кода сбора статистики по числу обращений к функциям, времени их работы, количеству неосвобожденных блоков памяти могут быть обнаружены как некорректные использования и *утечки памяти (англ., memory leak)*, так и нерациональная реализация алгоритмов. Однако более эффективным способом поиска «узких» мест является применение *профилировщиков (англ. profiler)* – программных средств, собирающих значения статистических характеристик работы программы.

В данном методическом пособии рассматриваются популярный многоцелевой профилировщик Valgrind (http://valgrind.org) и следующее подмножество инструментов, входящие в его состав:

- отладчик памяти memcheck;
- профилировщик кучи massif;
- профилировщик кэш-памяти и точек ветвления cachegrind;
- профилировщик вызовов функций callgrind.

Принцип работы Valgrind заключается в преобразовании скомпилированной программы в промежуточное представление, динамически транслируемое во время работы инструмента в машинные команды с одновременным подсчетом необходимой статистики. Архитектура Valgrind допускает написание новых инструментов сторонними разработчиками.

Формат командной строки:

```
valgrind [опции] программа-и-ее-аргументы
```

Опция Valgrind

--tool=<memcheck|massif|cachegrind|callgrind|…> [по умолчанию: memcheck]

позволяет выбрать инструмент

Memcheck

Memcheck – исторически первый инструмент Valgrind - отладчик памяти, позволяет отслеживать следующие дефекты программы:

- утечка памяти – не освобождение памяти, выделенной в куче (ситуация, при которой для блока памяти, выделенного функцией malloc/new, отсутствует вызов free/delete);
- обращение к памяти после ее освобождения;
- выход за границы выделенной памяти;
- использование неинициализированной памяти.

В программе:

```
#include <stdlib.h>

int main()
{
  char * p = malloc(10);
  p[10] = 1; // Ошибка – обращение к памяти «за»
             // выделенным блоком
  return 0;
}
```

memcheck, запущенный с помощью командной строки

```
valgrind --leak-check=full ./a.out
```

обнаружит выход за границы памяти и утечку (выделено жирным):

```
==1671== Memcheck, a memory error detector
==1671== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et
al.
==1671== Using Valgrind-3.6.1 and LibVEX; rerun with -h for copyright
info
==1671== Command: ./a.out
==1671==
==1671== Invalid write of size 1
==1671==    at 0x8048446: main (memcheck1.c:6)
==1671==  Address 0x17803a is 0 bytes after a block of size 10
alloc'd
==1671==    at 0x59278: malloc (in
/usr/local/lib/valgrind/vgpreload_memcheck-x86-freebsd.so)
==1671==    by 0x804843C: main (memcheck1.c:5)
==1671==
==1671==
==1671== HEAP SUMMARY:
==1671==     in use at exit: 10 bytes in 1 blocks
==1671==   total heap usage: 1 allocs, 0 frees, 10 bytes allocated
==1671==
==1671== 10 bytes in 1 blocks are definitely lost in loss record 1 of
133
==1671==    at 0x59278: malloc (in
/usr/local/lib/valgrind/vgpreload_memcheck-x86-freebsd.so)
==1671==    by 0x804843C: main (memcheck1.c:5)
```

```
==1671==
==1671== LEAK SUMMARY:
==1671==    definitely lost: 10 bytes in 1 blocks
==1671==    indirectly lost: 0 bytes in 0 blocks
==1671==      possibly lost: 0 bytes in 0 blocks
==1671==    still reachable: 0 bytes in 0 blocks
==1671==         suppressed: 0 bytes in 0 blocks
==1671==
==1671== For counts of detected and suppressed errors, rerun with: -v
==1671== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from
0)
```

Для получения информации о строке, содержащей проблему, программа должна быть скомпилирована с отладочной информацией (опция gcc -g).

В начале каждой строки выводится номер процесса.

Одной из опций memcheck является leak-check:

--leak-check=<no|summary|yes|full> [по умолчанию: summary]

Опция задает степень детализации отображения информации по утечкам памяти. No – отсутствие контроля утечек. Full - полная информация по утечкам с указанием конкретных проблемных мест в коде. Summary и yes – промежуточные режимы.

Massif

Massif - *профилировщик кучи (англ., heap)* и *стека (англ., stack)*. Собирает статистику по использованию памяти в куче и стеке через фиксированные промежутки времени.

Для следующего примера программы:

```
#include <stdlib.h>

void f()
{
  malloc(4000);
  malloc(4000);
}

int main()
{
  int i;

  char * p[10];
  for(i = 0; i < 10; i++)
```

```
  {
     p[i] = malloc(1000);
  }

  f();

  for(i = 0; i < 10; i++)
  {
     free(p[i]);
  }

  return 0;
}
```

профилировщик, запущенный командной строкой

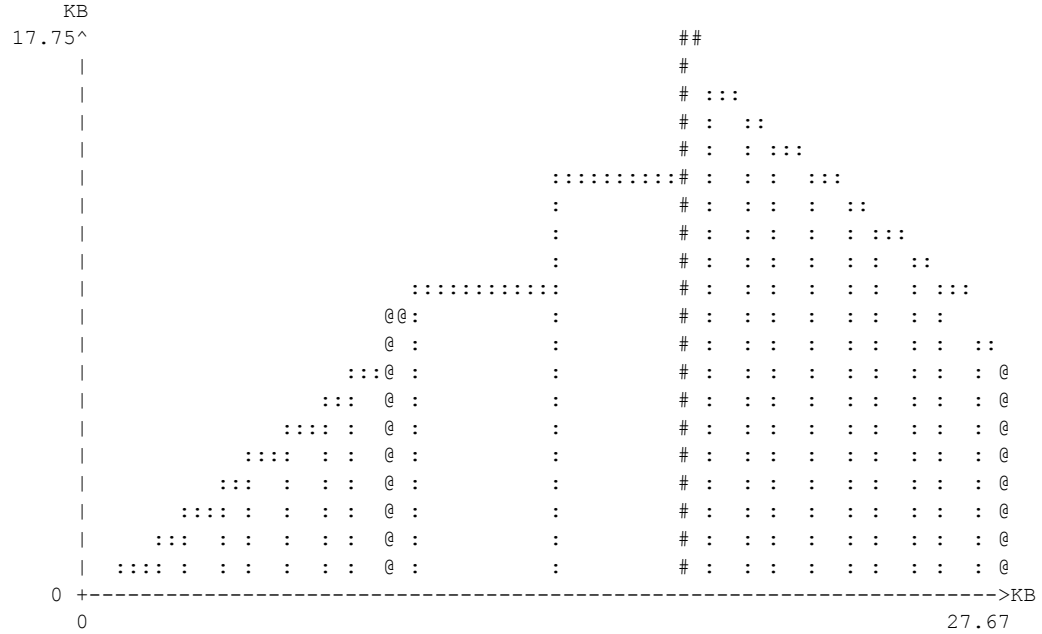
```
valgrind --tool=massif --time-unit=B ./a.out
```

сформирует файл с именем massif.out.<PID процесса>, содержащий статистику использования памяти.

Для визуализации отчета можно использовать утилиту ms_print. Команда

```
ms_print massif.out.<PID процесса>
```

приведет к выводу на экран графика использования памяти в куче:



В режиме –time-unit=B сбор статистики производится через фиксированное число выделенных / освобожденных байт (в режиме по умолчанию, предполагающем замеры расходования памяти через

фиксированные интервалы времени, график может получиться не читаемым из-за продолжительных временных интервалов, в которые выделение / освобождение памяти не производилось). Символом «:» обозначается «нормальный» замер. «@» соответствует подробному замеру, описание которого находится в листинге ms_print под графиком. По умолчанию каждый 10-й замер является подробным. Частота замеров регулируется с помощью опции --detailed-freq. Знак «#» соответствует замеру с пиковым расходованием памяти.

Под таблицей отображаются характеристики каждого замера: порядковый номер, время, текущий расход памяти, полезный расход памяти, дополнительная память (обычно выделяется объем памяти больше запрошенного за счет использования памяти под системную информацию либо выравнивание), размер стека. По детальным замерам также отображается вклад каждой функции в расходование памяти:

```
Number of snapshots: 24
 Detailed snapshots: [9, 13 (peak), 23]

--------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
  0              0                0              0             0            0
  1          1,016            1,016          1,000            16            0
  2          2,032            2,032          2,000            32            0
  3          3,048            3,048          3,000            48            0
  4          4,064            4,064          4,000            64            0
  5          5,080            5,080          5,000            80            0
  6          6,096            6,096          6,000            96            0
  7          7,112            7,112          7,000           112            0
  8          8,128            8,128          8,000           128            0
  9          9,144            9,144          9,000           144            0
98.43% (9,000B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->98.43% (9,000B) 0x8048498: main (massif.c:16)

--------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
 10         10,160           10,160          10,000           160            0
 11         14,168           14,168          14,000           168            0
 12         18,176           18,176          18,000           176            0
 13         18,176           18,176          18,000           176            0
99.03% (18,000B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->55.02% (10,000B) 0x8048498: main (massif.c:16)
|
->22.01% (4,000B) 0x8048460: f (massif.c:5)
| ->22.01% (4,000B) 0x80484AB: main (massif.c:19)
|
->22.01% (4,000B) 0x804846C: f (massif.c:6)
  ->22.01% (4,000B) 0x80484AB: main (massif.c:19)

--------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
 14         19,192           17,160          17,000           160            0
```

```
 15         20,208           16,144           16,000           144            0
 16         21,224           15,128           15,000           128            0
 17         22,240           14,112           14,000           112            0
 18         23,256           13,096           13,000            96            0
 19         24,272           12,080           12,000            80            0
 20         25,288           11,064           11,000            64            0
 21         26,304           10,048           10,000            48            0
 22         27,320            9,032            9,000            32            0
 23         28,336            8,016            8,000            16            0
99.80% (8,000B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->49.90% (4,000B) 0x8048460: f (massif.c:5)
| ->49.90% (4,000B) 0x80484AB: main (massif.c:19)
|
->49.90% (4,000B) 0x804846C: f (massif.c:6)
| ->49.90% (4,000B) 0x80484AB: main (massif.c:19)
|
->00.00% (0B) in 1+ places, all below ms_print's threshold (01.00%)
```

В 13-м замере сообщается, что 18000 байт было выделено методами malloc, new и т.д., из них 10000 байт были запрошены в функции main в 16-й строке и еще 4000 байт были два раза запрошены в 5й и 6й строках функцией f().

Опция –stacks=yes включает режим профилирования размера стека (по умолчанию, выключен).

Cachegrind

Cachegrind – симулятор кэш-памяти и точек ветвления. Эффективность использования кэш-памяти процессором во многих случаях оказывает существенное влияние на производительность программы. При этом узкие места в использовании кэш-памяти обычно скрыты как за алгоритмами трансляции с языка высокого уровня в ассемблер, так и за особенностями архитектуры процессора. Аналогичная ситуация обстоит с *предсказанием точек ветвления (англ., branch prediction)* – механизмом, позволяющим повысить производительность программы за счет досрочного предсказания и выполнения условных переходов в конвейерной архитектуре.

В процессе симуляции работы кэш-памяти cachegrind собирает статистику по числу *промахов (англ., miss)* и *обращений (англ., ref)* для следующих типов кэш-памяти:

- I1 – кэш команд 1-го уровня
- D1 – кэш данных 1-го уровня

- LL – кэш последнего уровня[2]

Информация об алгоритмах симуляции кэш и предсказании переходов приведена в документации на сайте разработчика.

Некоторые опции cachegrind:

- --I1=<размер>,<ассоциативность>,<размер строки>
  Определяет размер, ассоциативность и размер строки I1 кэш
- --D1=<размер>,<ассоциативность>,<размер строки>
  Определяет размер, ассоциативность и размер строки D1 кэш
- --LL=<размер>,<ассоциативность>,<размер строки>
  Определяет размер, ассоциативность и размер строки LL кэш
- --cache-sim=no|yes [по умолчанию: yes]
  Симуляция и сбор статистики по кэш.
- --branch-sim=no|yes [по умолчанию: no]
  Симуляция и сбор статистики по точкам ветвления.

Пример программы:

```
#include <stdlib.h>
#include <time.h>

int main()
{
  int i;
  int a[1000];
  double avg = 0.0;

  srand(time(NULL));
  for(i = 0; i < 1000; i++)
     a[i] = rand();

  for(i = 0; i < 1000; i++)
     avg += a[i];

  avg /= 1000;

  return 0;
}
```

[2] В современных процессорах используются двух- либо трехуровневая кэш-память. В cachegrind симулируется первый и последний уровень кэш, т.к. они оказывают наибольшее влияние на производительность.

Программа, скомпилированная в исполняемый модуль a.out, может быть проанализирована cachagrind с помощью следующей строки запуска:

```
valgrind --tool=cachegrind --branch-sim=yes ./a.out
```

Инструмент сформирует следующую информацию:

```
==2752== Cachegrind, a cache and branch-prediction profiler
==2752== Copyright (C) 2002-2010, and GNU GPL'd, by Nicholas
Nethercote et al.
==2752== Using Valgrind-3.6.1 and LibVEX; rerun with -h for copyright
info
==2752== Command: ./a.out
==2752==
==2752==
==2752== I   refs:      282,092
==2752== I1  misses:        412
==2752== LLi misses:         57
==2752== I1  miss rate:    0.14%
==2752== LLi miss rate:    0.02%
==2752==
==2752== D   refs:      136,029  (96,662 rd   + 39,367 wr)
==2752== D1  misses:      2,133  ( 1,685 rd   +    448 wr)
==2752== LLd misses:      1,838  ( 1,425 rd   +    413 wr)
==2752== D1  miss rate:     1.5% (   1.7%     +    1.1%  )
==2752== LLd miss rate:     1.3% (   1.4%     +    1.0%  )
==2752==
==2752== LL refs:         2,545  ( 2,097 rd   +    448 wr)
==2752== LL misses:       1,895  ( 1,482 rd   +    413 wr)
==2752== LL miss rate:      0.4% (   0.3%     +    1.0%  )
==2752==
==2752== Branches:       43,997  (40,660 cond +  3,337 ind)
==2752== Mispredicts:     3,032  ( 2,982 cond +     50 ind)
==2752== Mispred rate:      6.8% (   7.3%     +    1.4%   )
```

В файл cachegrind.out.<PID процесса> попадут значения счетчиков характеристик использовании кэш и предсказания точек ветвления: Ir, I1mr, ILmr, Dr, D1mr, DLmr, Dw, D1mw, DLmw, Bc, Bi,

где «m» – означает «промах», «r»/«w» - чтение / запись соответственно, Bc – условный переход, Bi – косвенный переход (переход по адресу, значение которого заранее неизвестно).

Запуск инструмента аннотирования с помощью команды

```
cg_annotate cachegrind.out.2752
```

позволяет получить статистику с разбиением по функциям:

```
    Ir I1mr ILmr     Dr  D1mr  DLmr     Dw D1mw DLmw     Bc   Bcm    Bi Bim
file:function
```

```
--------------------------------------------------------------------------------
257,536  382   34 86,500 1,683 1,423 33,227  400  376 38,580 2,923 3,329  44  ???:???
 17,033    2    2  9,009     1     1  4,012   46   36  2,002    22     0   0
/usr/home/test/valgrind/cachegrind.c:main
  7,000    0    0  1,000     0     0  2,000    0    0      0     0     0   0  ???:rand
```

Режим построчного аннотирования позволяет оценить эффективность использования кэш-памяти и предсказания точек ветвления для строк исходных текстов и реализуется опцией –auto=yes:

```
-- Auto-annotated source: /usr/home/test/valgrind/cachegrind.c
--------------------------------------------------------------------------------
   Ir I1mr ILmr    Dr D1mr DLmr    Dw D1mw DLmw    Bc Bcm Bi Bim

    .    .    .     .    .    .     .    .    .     .   .  .   .  #include <stdlib.h>
    .    .    .     .    .    .     .    .    .     .   .  .   .  #include <time.h>
    .    .    .     .    .    .     .    .    .     .   .  .   .
    .    .    .     .    .    .     .    .    .     .   .  .   .  int main()
    8    1    1     1    0    0     4    0    0     0   0  0   0  {
    .    .    .     .    .    .     .    .    .     .   .  .   .    int i;
    .    .    .     .    .    .     .    .    .     .   .  .   .    int a[1000];
    2    0    0     0    0    0     1    0    0     0   0  0   0    double avg = 0.0;
    .    .    .     .    .    .     .    .    .     .   .  .   .
    4    0    0     0    0    0     4    1    1     0   0  0   0    srand(time(NULL));
3,004    1    1 2,001    0    0     1    0    0 1,001  12  0   0    for(i = 0; i <
1000; i++)
3,000    0    0 1,000    0    0 2,000   45   35     0   0  0   0       a[i] = rand();
    .    .    .     .    .    .     .    .    .     .   .  .   .
3,004    0    0 2,001    0    0     1    0    0 1,001  10  0   0    for(i = 0; i <
1000; i++)
8,000    0    0 4,000    0    0 2,000    0    0     0   0  0   0       avg += a[i];
    .    .    .     .    .    .     .    .    .     .   .  .   .
    4    0    0     2    1    1     1    0    0     0   0  0   0    avg /= 1000;
    .    .    .     .    .    .     .    .    .     .   .  .   .
    1    0    0     0    0    0     0    0    0     0   0  0   0    return 0;
    6    0    0     4    0    0     0    0    0     0   0  0   0  }
--------------------------------------------------------------------------------
Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw Bc Bcm Bi Bim
--------------------------------------------------------------------------------
 6    0    4  9    0    0 10   10    9  5   1  0   0  percentage of events annotated
```

cg_annotate может быть использован для построчного аннотирования и машинных команд. Для этого программа, написанная, например, на Си, должна быть предварительно транслирована в ассемблер с помощью опции –S, после чего скомпилирована в режиме отладки (–g). Результат аннотирования:

```
-- Auto-annotated source: /usr/home/test/valgrind/cachegrind.s
--------------------------------------------------------------------------------
   Ir I1mr ILmr    Dr D1mr DLmr    Dw D1mw DLmw    Bc Bcm Bi Bim

    .    .    .     .    .    .     .    .    .     .   .  .   . .file  "cachegrind.c"
    .    .    .     .    .    .     .    .    .     .   .  .   . .text
    .    .    .     .    .    .     .    .    .     .   .  .   . .p2align 4,,15
    .    .    .     .    .    .     .    .    .     .   .  .   . .globl main
    .    .    .     .    .    .     .    .    .     .   .  .   . .type main,@function
    .    .    .     .    .    .     .    .    .     .   .  .   . main:
    1    1    1     0    0    0     0    0    0     0   0  0   0   leal 4(%esp), %ecx
    1    0    0     0    0    0     0    0    0     0   0  0   0   andl    $-16, %esp
    1    0    0     1    0    0     1    0    0     0   0  0   0   pushl   -4(%ecx)
```

```
    1    0    0     0    0    0     1    0    0     0   0  0   0    pushl   %ebp
    1    0    0     0    0    0     0    0    0     0   0  0   0    movl    %esp, %ebp
    1    0    0     0    0    0     1    0    0     0   0  0   0    pushl   %ebx
    1    0    0     0    0    0     1    0    0     0   0  0   0    pushl   %ecx
    1    0    0     0    0    0     0    0    0     0   0  0   0    subl    $4032,%esp
    1    0    0     0    0    0     0    0    0     0   0  0   0    fldz
    1    0    0     0    0    0     1    0    0     0   0  0   0    fstpl   -16(%ebp)
    1    0    0     0    0    0     1    1    1     0   0  0   0    movl    $0, (%esp)
    1    0    0     0    0    0     1    0    0     0   0  0   0    call    time
    1    0    0     0    0    0     1    0    0     0   0  0   0    movl   %eax,(%esp)
    1    0    0     0    0    0     1    0    0     0   0  0   0    call    srand
    1    0    0     0    0    0     1    0    0     0   0  0   0    movl $0, -20(%ebp)
    1    0    0     0    0    0     0    0    0     0   0  0   0    jmp     .L2
    .    .    .     .    .    .     .    .    .     .   .  .   .  .L3:
1,000    0    0 1,000    0    0     0    0    0     0   0  0   0   movl -20(%ebp),%ebx
1,000    0    0     0    0    0 1,000    0    0     0   0  0   0   call rand
1,000    0    0     0    0    0 1,000   45   35     0   0  0   0   movl %eax,-
4020(%ebp,%ebx,4)
1,000    0    0 1,000    0    0     0    0    0     0   0  0   0   addl $1, -20(%ebp)
    .    .    .     .    .    .     .    .    .     .   .  .   .  .L2:
1,001    1    1 1,001    0    0     0    0    0     0   0  0   0   cmpl $999,-20(%ebp)
1,001    0    0     0    0    0     0    0    0 1,001  12  0   0   jle     .L3
    1    0    0     0    0    0     1    0    0     0   0  0   0   movl  $0, -20(%ebp)
    1    0    0     0    0    0     0    0    0     0   0  0   0   jmp     .L5
    .    .    .     .    .    .     .    .    .     .   .  .   .  .L6:
1,000    0    0 1,000    0    0     0    0    0     0   0  0   0   movl -20(%ebp),%eax
1,000    0    0 1,000    0    0     0    0    0     0   0  0   0   movl -
4020(%ebp,%eax,4), %eax
1,000    0    0     0    0    0 1,000    0    0     0   0  0   0   pushl   %eax
1,000    0    0 1,000    0    0     0    0    0     0   0  0   0   fildl   (%esp)
1,000    0    0     0    0    0     0    0    0     0   0  0   0   leal 4(%esp), %esp
1,000    0    0 1,000    0    0     0    0    0     0   0  0   0   fldl    -16(%ebp)
1,000    0    0     0    0    0     0    0    0     0   0  0   0   faddp   %st, %st(1)
1,000    0    0     0    0    0 1,000    0    0     0   0  0   0   fstpl   -16(%ebp)
1,000    0    0 1,000    0    0     0    0    0     0   0  0   0   addl $1, -20(%ebp)
    .    .    .     .    .    .     .    .    .     .   .  .   .  .L5:
1,001    0    0 1,001    0    0     0    0    0     0   0  0   0    cmpl    $999, -
20(%ebp)
1,001    0    0     0    0    0     0    0    0 1,001  10  0   0   jle     .L6
    1    0    0     1    0    0     0    0    0     0   0  0   0   fldl    -16(%ebp)
    1    0    0     1    1    1     0    0    0     0   0  0   0   fldl    .LC1
    1    0    0     0    0    0     0    0    0     0   0  0   0   fdivrp  %st, %st(1)
    1    0    0     0    0    0     1    0    0     0   0  0   0   fstpl   -16(%ebp)
    1    0    0     0    0    0     0    0    0     0   0  0   0   movl    $0, %eax
    1    0    0     0    0    0     0    0    0     0   0  0   0   addl    $4032, %esp
    1    0    0     1    0    0     0    0    0     0   0  0   0   popl    %ecx
    1    0    0     1    0    0     0    0    0     0   0  0   0   popl    %ebx
    1    0    0     1    0    0     0    0    0     0   0  0   0   popl    %ebp
    1    0    0     0    0    0     0    0    0     0   0  0   0   leal -4(%ecx), %esp
    1    0    0     1    0    0     0    0    0     0   0  0   0   ret
    .    .    .     .    .    .     .    .    .     .   .  .   .  .size  main, .-main
    .    .    .     .    .    .     .    .    .     .   .  .   .  .section
.rodata
    .    .    .     .    .    .     .    .    .     .   .  .   .  .align 8
    .    .    .     .    .    .     .    .    .     .   .  .   .  .LC1:
    .    .    .     .    .    .     .    .    .     .   .  .   .    .long   0
    .    .    .     .    .    .     .    .    .     .   .  .   .    .long   1083129856
    .    .    .     .    .    .     .    .    .     .   .  .   .    .ident  "GCC:
(GNU) 4.2.1 20070719  [FreeBSD]"

--------------------------------------------------------------------------------
Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw Bc Bcm Bi Bim
--------------------------------------------------------------------------------
 6    0    4  9    0    0 10   10    9  5   1  0   0  percentage of events annotated
```

С точки зрения информативности, целесообразно запускать cachegrind на программе, скомпилированной с отладочной информацией (опция –g), однако тонкую оптимизацию производительности имеет смысл проводить на программе, скомпилированной в рабочей конфигурации с включенной оптимизацией кода. Чтобы преодолеть это противоречие существует утилиты cg_merge и cg_diff, позволяющие объединять либо находить различия между двумя версиями файла вывода cachegrind. Итоговый файл может быть передан в cg_annotate для анализа.

Callgrind

При разработке высоконагруженных приложений либо реализации алгоритмов обработки больших массивов данных часто возникает необходимость в поиске функциональности, критически влияющей на производительность программы. Узкие места могут быть найдены с помощью замеров времени в коде программы. Другим способом является использование *профилировщика вызовов функций*. Инструмент callgrind строит дерево вызовов функций программы. Для каждой функции рассчитывается *суммарная стоимость (англ., inclusive cost)* и *собственная стоимость (англ., exclusive cost)*, выраженные в числе машинных инструкций. Если функция f вызывает функцию g, то число инструкций g войдут в суммарную стоимость f. Собственную стоимость f составят инструкции функции без учета инструкций по обращению к g. Таким образом, суммарная стоимость функции main составляет 100% стоимости программы. Собственная стоимость функции – это число вызванных инструкций данной функции, за исключением инструкций по обращению к другим функциям.

Командная строка для запуска callgrind:

```
valgrind –tool=callgrind имя-программы [аргументы]
```

Инструмент создает выходной файл с именем callgrind.out.<PID процесса>, который может быть обработан утилитой callgrind_annotate с помощью командной строки:

```
callgrind_annotate [опции] callgrind.out.<pid>
```

callgrind_annotate во многом аналочичен cg_annotate. Список функций упорядочен по собственной стоимости. Некоторые опции callgrind_annotate:

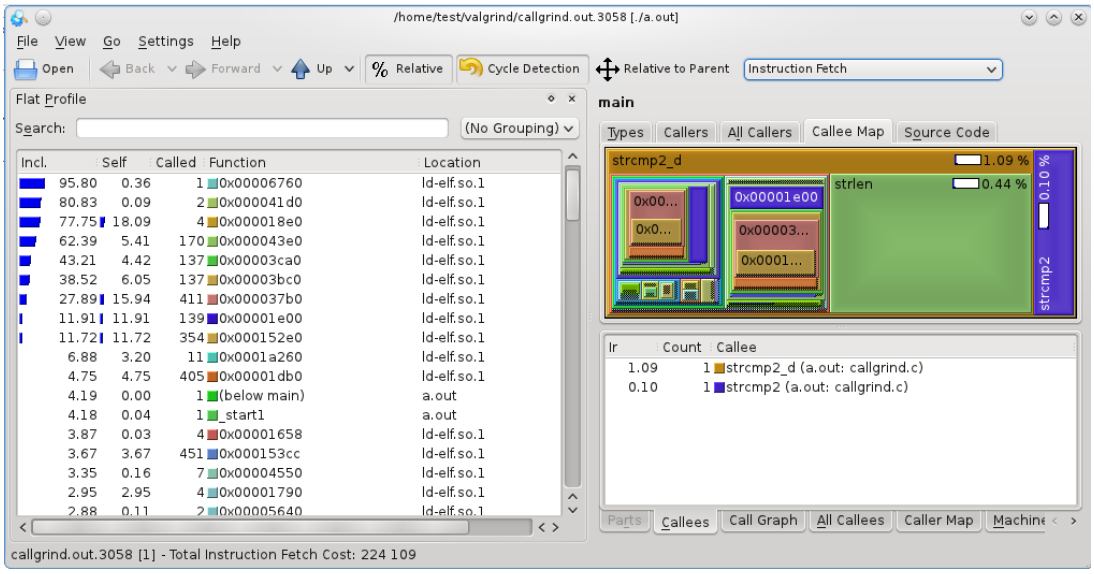
- --inclusive=<yes|no> [по умолчанию: no]
  yes - вместо сортировки по собственной стоимости используется сортировка по суммарной стоимости.
- --tree=<none|caller|calling|both> [по умолчанию: none]
  Способ отображения дерева вызовов: caller - по каждой функции выводится список функций, вызвавших ее; calling – отображаются вызванные функции, both – комбинация режимов caller и calling. Ниже приведен фрагмент дерева, построенного в режиме calling, для следующего примера программы:

```
#include <string.h>

int strcmp2(char *p, char *q)
{
  while(*p && *p == *q)
  {
    p++;
    q++;
  }
  return *p - *q;
}

int strcmp2_d(char *p, char *q)
{
  int i = 0;
  while(i < strlen(p) && i < strlen(q) && p[i] == q[i])
    i++;
  return p[i] - q[i];
}

int main()
{
  char * p = "qwertyasdfghzxcvbn";
  char * q = "qwertyasdfghzxcvbN";
  strcmp2(p, q);
  strcmp2_d(p, q);
  return 0;
}
```

Фрагмент вывода:

```
2,704  *  callgrind.c:main
[/usr/home/test/valgrind/a.out]
    228  >   callgrind.c:strcmp2 (1x)
[/usr/home/test/valgrind/a.out]
  2,452  >   callgrind.c:strcmp2_d (1x)
[/usr/home/test/valgrind/a.out]
  2,452  *  callgrind.c:strcmp2_d
[/usr/home/test/valgrind/a.out]
    980  >   ???:strlen (35x) [/lib/libc.so.7]
  1,074  >   ???:0x00001658 (1x) [/libexec/ld-elf.so.1]
```

Для наглядного отображения дерева вызовов рекомендуется использовать средство графической визуализации KCacheGrind:



Callgrind также включает в себя функции по симуляции кэш и предсказанию переходов, аналогичные cachegrind.

Автоматическое тестирование в CUnit

На сегодняшний день любое реально используемое программное обеспечение находится в состоянии непрерывной доработки – разработчиками выпускаются все новые и новые версии, содержащие исправления ошибок, реализацию новых требований заказчика, изменения, обусловленные обновлениями сопутствующего ПО: выходом новых версий операционных систем и библиотек. В такой ситуации «ручное» тестирование каждой новой модификации не является хорошей идей в виду большого

объема трудозатрат и рутинности процесса. На помощь приходит *автоматическое тестирование*, суть которого заключается в использовании программных средств для выполнения тестов и контроля результатов их выполнения, что помогает сократить время тестирования и упростить его процесс. Разновидностями автоматического тестирования являются:

- *Системное тестирование (англ., system testing)* – тестирование системы как «черного ящика» (без знаний деталей реализации) с помощью сценариев, близких к сценариям использования системы в реальной работе.
- *Модульное тестирование (англ., unit testing)* – тестирование т.н. методом «белого ящика», подразумевающего тестирование внутренних функций и методов, непосредственно реализующих систему. Цель модульного тестирования – показать, что отдельные части программы сами по себе функционируют без ошибок. Модульное тестирование реализуется с помощью создания набора *тестов*, описывающих сценарии вызова функций и сравнивающих фактические результаты работы с ожидаемыми при помощи набора *проверок (англ., assert)*, входящих в состав системы модульного тестирования. Тесты являются обычными функциями/методами, написанными на языке программирования высокого уровня, и включаются в состав исходных текстов программы. Как правило, система тестирования также включает в себя подсистему автоматического запуска тестов (в т.ч. с графическим пользовательским интерфейсом). Для проверки корректности программы после внесения очередных модификаций необходимо лишь активировать запуск тестов.

Широкую популярность сегодня приобрел подход *разработка через тестирование (англ., test-driven development)*, в основе которого лежит идея реализации/доработки функциональности минимальными частями, причем до реализации новой функции должны быть написаны модульные тесты на нее. Написание тестов до реализации позволяет заранее продумать программные интерфейсы и сценарии использования новой функциональности.

Автоматическое тестирование и разработка через тестирование обладают следующими преимуществами:

- Инкрементально пополняемый набор тестов снижает риск внесения ошибки при модификации ПО.
- Автоматическая проверка тестов позволяет уменьшить издержки по сравнению с рутинным «ручным» тестированием.
- Наличие набора автоматических тестов положительно влияет на психологическое состояние программистов, вносящих изменение в ПО.
- Опережение реализации написанием тестов позволяет спроектировать программные интерфейсы новой функциональности на раннем этапе цикла разработки.
- Использование модульных тестов неизбежно приводит к необходимости разумной декомпозиции программного кода и уменьшению связности между его частями и, как результат, к коду более высокого качества.

Для начала использования модульного тестирования, вообще говоря, не требуется какое-либо специализированное средство, достаточно описать проверочные сценарии в виде функций на языке программирования и реализовать простейший метод, вызывающий сценарии и отображающий результаты их выполнения. Для реализации модульного тестирования на Си программисту может пригодиться макрос стандартной библиотеки assert(cond), аварийно завершающий программу при невыполнении условия cond.

Одной из распространенных систем модульного тестирования программ, написанных на языке Си, является CUnit (http://cunit.sourceforge.net). В CUnit тест является функцией языка Си, не имеющей параметров и возвращающего значения. Логически каждый тест является листом следующей иерархии:

- *Реестр тестов (англ., test registry)*
    - *Набор тестов (англ., test suite)*
        - *Тест (англ., test)*

Для тестов, входящих в набор, могут быть заданы функция *установки (англ., setup)* и функция *очистки (англ., teardown)* контекста. Функция установки вызывается один раз до запуска всех

тестов набора, функция очистки также один раз после работы тестов. Указанные функции могут быть полезны для создания/освобождения вспомогательных структур данных, открытия/закрытия временных файлов и т.п. CUnit позволяет запускать как тесты из заданных реестров и наборов, так и отдельные тесты.

Реестры

CUnit создает один реестр тестов по умолчанию. До его использования должна быть вызвана функция CU_initialize_registry. После завершения тестирования пользователь должен вызвать функцию CU_cleanup_registry для освобождения памяти, занятой реестром. Данная функция должна быть последней вызванной функцией CUnit. Отсутствие вызова CU_cleanup_registry() приведет к утечке памяти.

Функции CU_initialize_registry и CU_cleanup_registry оперируют единственным реестром, создаваемым по умолчанию. Для создания и управления дополнительными реестрами следует использовать функции:

- CU_create_new_registry,
- CU_get_registry,
- CU_set_registry,
- CU_destroy_existing_registry

, о которых можно получить дополнительную информацию в документации по CUnit.

Наборы

Функция

```
CU_pSuite CU_add_suite(const char* strName,
CU_InitializeFunc pInit, CU_CleanupFunc pClean)
```

создает новый набор тестов с указанными именем, функциями установки и очистки. Созданный набор добавляется в реестр, соответственно реестр должна быть инициализирован до вызова данной функции.

Имена наборов должны быть уникальны в рамках реестра. Функции инициализации и очистки необязательны, передаются в

виде указателя на функцию без параметров, возвращающую int. Функции должны возвращать 0 в случае успеха. Если набор не нуждается в установке и/или очистке в качестве значения соответствующего указателя нужно передать NULL.

Функция возвращает указатель на созданный набор. В последствие указатель используется для добавления тестов в набор. В случае ошибки возвращается NULL и устанавливается значение кода ошибки, который может быть считан с помощью функций:

```
CU_ErrorCode   CU_get_error(void);
const char*    CU_get_error_msg(void);
```

С кодами ошибок, устанавливаемыми функциями CUnit, можно ознакомиться в документации по CUnit.

Тесты

Функция

```
CU_pTest CU_add_test(CU_pSuite pSuite, const char*
strName, CU_TestFunc pTestFunc)
```

создает новый тест с заданным именем и реализующей его функцией и добавляет его в заданный набор. Набор pSuite должен быть создан с помощью CU_add_sute. Имена тестов должны быть уникальны в рамках набора. Тип CU_testFunc определен как:

```
typedef void (*CU_TestFunc)(void);
```

pTestFunc не может быть NULL. CU_add_test возвращает указатель на новый тест. В случае ошибки возвращается NULL и устанавливается код ошибки.

Макрос CU_ADD_TEST автоматически генерирует уникальное имя теста, основываясь на имени функции, реализующей тест, и добавляет тест в заданный набор:

```
#define CU_ADD_TEST(suite, test) (CU_add_test(suite,
#test, (CU_TestFunc)test))
```

Проверки

В таблице приведены некоторые проверки CUnit:

Название макроса	Контролируемое условие
CU_ASSERT(int expression)	Проверка на ИСТИНУ
CU_ASSERT_FALSE(value)	Проверка на ЛОЖЬ

CU_ASSERT_EQUAL(actual, expected)	Проверка на равенство
CU_ASSERT_NOT_EQUAL(actual, expected))	Проверка на неравенство
CU_ASSERT_PTR_EQUAL(actual, expected)	Проверка указателей на равенство
CU_ASSERT_STRING_EQUAL(actual, expected)	Проверка строк на равенство
CU_ASSERT_NSTRING_EQUAL(actual, expected, count)	Проверка первых N символов строк на равенство
CU_ASSERT_DOUBLE_EQUAL(actual, expected, granularity)	Проверка чисел с плавающей точкой на равенство с заданной точностью

CUnit предоставляет макросы с суффиксом “FATAL”, например, CU_ASSERT_FATAL, обеспечивающие останов процесса тестирования на текущем тесте в случае невыполнения проверки.

Запуск тестов

Базовый режим

```
CU_ErrorCode CU_basic_run_tests(void)
```

Запуск всех тестов. Возвращает код первой ошибки, возникшей при запуске тестов.

```
CU_ErrorCode CU_basic_run_suite(CU_pSuite pSuite)
```

Запуск всех тестов заданного набора. Возвращает код первой ошибки, возникшей при запуске тестов.

```
CU_ErrorCode CU_basic_run_test(CU_pSuite pSuite, CU_pTest
pTest)
```

Запуск заданного теста и заданного набора. Возвращает код первой ошибки, возникшей при запуске теста.

Интерактивный режим

Выбор, запуск наборов и тестов, просмотр результатов происходят в интерактивном режиме. Для запуска режима консоли необходимо вызвать функцию

```
void CU_console_run_tests(void)
```

Получение результатов прохождения тестов

```
const CU_pRunSummary CU_get_run_summary(void)
```

Возвращает результаты запуска тестов. Возвращаемое значение – указатель на структуру CU_RunSummary, содержащую статистику запуска тестов (поля структуры самодокументируемы):

```
typedef struct CU_RunSummary
{
     unsigned int nSuitesRun;
     unsigned int nSuitesFailed;
     unsigned int nTestsRun;
     unsigned int nTestsFailed;
     unsigned int nAsserts;
     unsigned int nAssertsFailed;
     unsigned int nFailureRecords;
} CU_RunSummary;
typedef CU_Runsummary* CU_pRunSummary;
```

Функция

```
const CU_pFailureRecord CU_get_failure_list(void)
```

возвращает информацию о тестах, не пройденных за последний запуск (NULL, если все тесты прошли) в виде списка структур, содержащих данные о местоположении теста и не пройденной проверке:

```
typedef struct CU_FailureRecord
{
     unsigned int  uiLineNumber;
     char*         strFileName;
     char*         strCondition;
     CU_pTest      pTest;
     CU_pSuite     pSuite;

     struct CU_FailureRecord* pNext;
     struct CU_FailureRecord* pPrev;

} CU_FailureRecord;

typedef CU_FailureRecord*  CU_pFailureRecord;
```

Пример использования CUnit

Рассмотрим функции по работе со множеством чисел (структура Set):

```
#include <stdlib.h>

/* Множество */
struct _Set
{
    int * data;
    int size;
};

typedef struct _Set Set;

/* Код успеха */
#define SUCCESS 1
/* Код неуспеха */
#define FAIL 0

typedef int Result;

/* ИНТЕРФЕЙС МНОЖЕСТВА */
/* Проверка вхождения, возвращает 1, если множество содержит элемент,
иначе 0 */
int contains(Set * set, int elem);
/* Вставка, вставляет элемент и возвращает SUCCESS, если исходное
множество не содержит элемент, иначе FAIL */
Result insert(Set * set, int elem);
/* Удаление, удаляет и возвращает SUCCESS, если исходное множество
содержит элемент, иначе FAIL */
Result remove(Set * set, int elem);

/* РЕАЛИЗАЦИЯ */
/* Двоичный поиск в упорядоченном массиве. Возвращает индекс (начиная
с 0) найденного элемента
   либо -1. В случае нескольких вхождений одного и того же элемента,
возвращается индекс
   произвольного.
   При отсутствии искомого элемента в массиве через posBefore
возвращается индекс ближайшего
   элемента, меньшего искомого  */
int _bsearch (int * array, int len, int elem, int * posBefore)
{
    int low = 0, high = len - 1;
    while(low <= high)
    {
        int mid = (low + high) / 2;

        if (elem == array[mid])
            return mid;
        else if (elem < array[mid])
```

```
            high = mid - 1;
        else
            low = mid + 1;
    }
    if(NULL != posBefore)
        *posBefore = high;
    return -1;
}

int contains(Set * set, int elem)
{
    return indexOf(set, elem) != -1;
}

int indexOf(Set * set, int elem)
{
    return _bsearch(set->data, set->size - 1, elem, NULL);
}

Result insert(Set * set, int elem)
{
    int index, posBefore;
    index = _bsearch(set->data, set->size, elem, &posBefore);

    if(index != -1)
        return FAIL;

    set->data = realloc(set->data, sizeof(int) *  (set->size + 1));
    memmove(set->data + posBefore + 2, set->data + posBefore + 1,
sizeof(int) * (set->size - posBefore - 1));
    set->data[posBefore + 1] = elem;

    set->size++;

    return SUCCESS;
}

Result remove(Set * set, int elem)
{
    int index, posBefore;
    index = _bsearch(set->data, set->size, elem, &posBefore);

    if(index == -1)
        return FAIL;


    set->data = realloc(set->data, sizeof(int) * (set->size - 1));
    memmove(set->data + index, set->data + index + 1, sizeof(int) *
(set->size - index - 1));

    set->size--;

    return SUCCESS;
```

```
}
```

Автоматические тесты могут созданы как на функции интерфейса библиотеки:

```
void test1(void)
{
    Set s = {0};
    CU_ASSERT_EQUAL(s.size, 0);
    insert(&s, 3);
    CU_ASSERT_EQUAL(s.size, 1);
    insert(&s, 1);
    CU_ASSERT_EQUAL(s.size, 2);
    CU_ASSERT_EQUAL(s.data[0], 1);
    CU_ASSERT_EQUAL(s.data[1], 3);
}

void test2(void)
{
    Set s = {0};

    CU_ASSERT_EQUAL(insert(&s, 1), SUCCESS);
    CU_ASSERT_EQUAL(insert(&s, 1), FAIL);
}

void test3(void)
{
    Set s = {0};

    insert(&s, 1);

    CU_ASSERT_EQUAL(remove(&s, 2), FAIL);
    CU_ASSERT_EQUAL(remove(&s, 1), SUCCESS);
}
```

, так и на функцию _bsearch, используемую реализацией библиотеки:

```
void test4()
{
    int array[] = {1, 3, 5, 11};
    int size = sizeof(array) / sizeof(int);

    int posBefore;
```

```
    CU_ASSERT_EQUAL(_bsearch(array, size, 4, &posBefore),
-1);
    CU_ASSERT_EQUAL(posBefore, 1);

    CU_ASSERT_EQUAL(_bsearch(array, size, 19,
&posBefore), -1);
    CU_ASSERT_EQUAL(posBefore, 3);

    CU_ASSERT_EQUAL(_bsearch(array, size, 5, &posBefore),
2);
}
```

Следует отметить, что в реальных программах обычно создается большее число тестов, чем в рассмотренном примере. Тестовое покрытие должно по возможности описывать все многообразие ситуаций, в которых от функции хочется ожидать детерминированный результат. Например, «пустые» параметры, граничные точки, вообще говоря, любые параметры, обладающие свойствами, прямо или косвенно используемыми реализацией (например, четность/нечетность числа элементов массива в _bsearch). Иногда бывает полезно включить в тестовый набор тесты, проверяющие обработку функцией входных некорректных параметров.

Пример кода помещения тестов в набор и их автоматического запуска:

```
#include <CUnit/Basic.h>

int main()
{
    CU_pSuite suite;

    CU_initialize_registry();

    suite = CU_add_suite("main_suite", NULL, NULL);

    CU_ADD_TEST(suite, test1);
    CU_ADD_TEST(suite, test2);
    CU_ADD_TEST(suite, test3);
    CU_ADD_TEST(suite, test4);

    CU_basic_run_tests();
```

```
    CU_cleanup_registry();

    return CU_get_error();
}
```

Данная тестирующая программа, скомпилированная в исполняемый файл testrunner в последствии может стать отдельной целью Makefile:

```
test:
     testrunner
```

Пример запуска среды интерактивной проверки CUnit:

```
#include <CUnit/Console.h>

int main()
{
    CU_pSuite suite;

    CU_initialize_registry();

    suite = CU_add_suite("main_suite", NULL, NULL);

    CU_ADD_TEST(suite, test1);
    CU_ADD_TEST(suite, test2);
    CU_ADD_TEST(suite, test3);
    CU_ADD_TEST(suite, test4);

    CU_console_run_tests();

    CU_cleanup_registry();

    return 0;
}
```

Документирование исходных текстов программ с использованием Doxygen

Возвращаясь к примеру функции _bsearch из предыдущего раздела, реализующей бинарный поиск, следует обратить внимание на тот факт, что имея перед глазами один только прототип этой функции

```
int _bsearch (int * array, int len, int elem, int *
posBefore)
```

, сложно догадаться о смысле некоторых параметров, например, параметра posBefore. Конечно же, можно заглянуть в реализацию функции, однако это требует дополнительного времени, сил и сопряжено с ошибками интерпретации кода. Кроме этого, не во всех случаях исходный текст реализации доступен программисту. Помочь решить проблему может документирование данной функции – снабжение ее текстом, описывающим семантику параметров и возвращаемого значения. Документирование, как правило, производится в исходном тексте программы, используя специальным образом размеченный комментарий языка программирования, расположенный непосредственно перед описываемым элементом программы: типом либо функцией.

Системы генерации документации позволяют упростить процесс документирования. Основной их задачей является автоматическое построение файлов документации по документирующим комментариям. Принимая на вход документированные исходные тексты программы, генератор документации сопоставляет документирующие комментарии с документируемыми элементами программы, осуществляет разбор и контроль корректности разметки и формирует итоговую документацию в одном из популярных текстовых форматов.

Кросс-платформенная система генерации документации Doxygen (http://www.doxygen.org) поддерживает языки программирования Си++, Си, Objective-C, Python, Java, IDL, PHP, C#, Fortran, VHDL, D и генерирует документацию на основе набора исходных текстов, содержащих документирующие комментарии в стиле Си/Си++. Кроме этого Doxygen может быть использован для извлечения структуры программы из недокументированных исходных кодов. Возможно автоматическое составление диаграмм классов и графов зависимостей сущностей программы. Doxygen поддерживает генерацию документации в форматах HTML, LATEX, man, RTF и XML.

Doxygen является консольной программой. Параметры построения документации хранятся в конфигурационном файле, имеющем текстовый формат.

Документирование исходного кода

Блоки комментариев

Для документирования исходного кода используются блоки комментариев в стиле Си/Си++ с использованием дополнительной разметки. Для каждой сущности программы могут быть заданы описания двух типов: *подробное* и *краткое* (для функций также существует третий тип описания – «в теле»). Краткое описание – однострочное, подробное – более детализированное многострочное, описание «в теле», как правило, используется для описания реализации функции.

Существует несколько способов отметить блок комментария как подробное описание:

1. Многострочный комментарий, начинающийся с *двух или более* «*», т.н. JavaDoc стиль описания:

```
/**
 * Подробное
 * описание
 */
```

Промежуточные ‘*’ не обязательны:

```
 /**
 Подробное
 описание
 */
```

2. Многострочный комментарий, начинающийся с «/!*». Описание в стиле QT.

```
/*!
 * Подробное
 * описание
 */
```

3. Три или более однострочных комментария Си/Си++, где каждая строка начинается с «///» или «//!»:

```
///
/// Подробное
/// описание
///
```

и:

```
//!
//! Текст
//!
```

4. Более выделенные в исходном коде блоки комментариев:

```
/*********
 * Подробное
 * описание
 **********/
```

и:

```
////////////////////
/// Подробное
/// описание
////////////////////
```

Способы создания краткого описания:

1. Использование команды brief:

```
/**
 * @brief Краткое
 * описание
 *
 * Полное описание
 */
```

В данном случае краткое и полное описания разделяются пустой строкой.

2. Использование режима JAVADOC_AUTOBRIEF=YES в конфигурационном файле.

В данном режиме краткое описание не требует отдельной команды и завершается точкой либо новой строкой:

```
/**
 * Краткое описание. Полное описание
 */
```

3. Разделение на несколько блоков:

```
/// Краткое описание
/**
 * Полное
 * описание
 */
```

или:

```
/// Краткое описание

/// Полное
/// описание
```

Обязательна пустая строка между кратким и полным описанием.

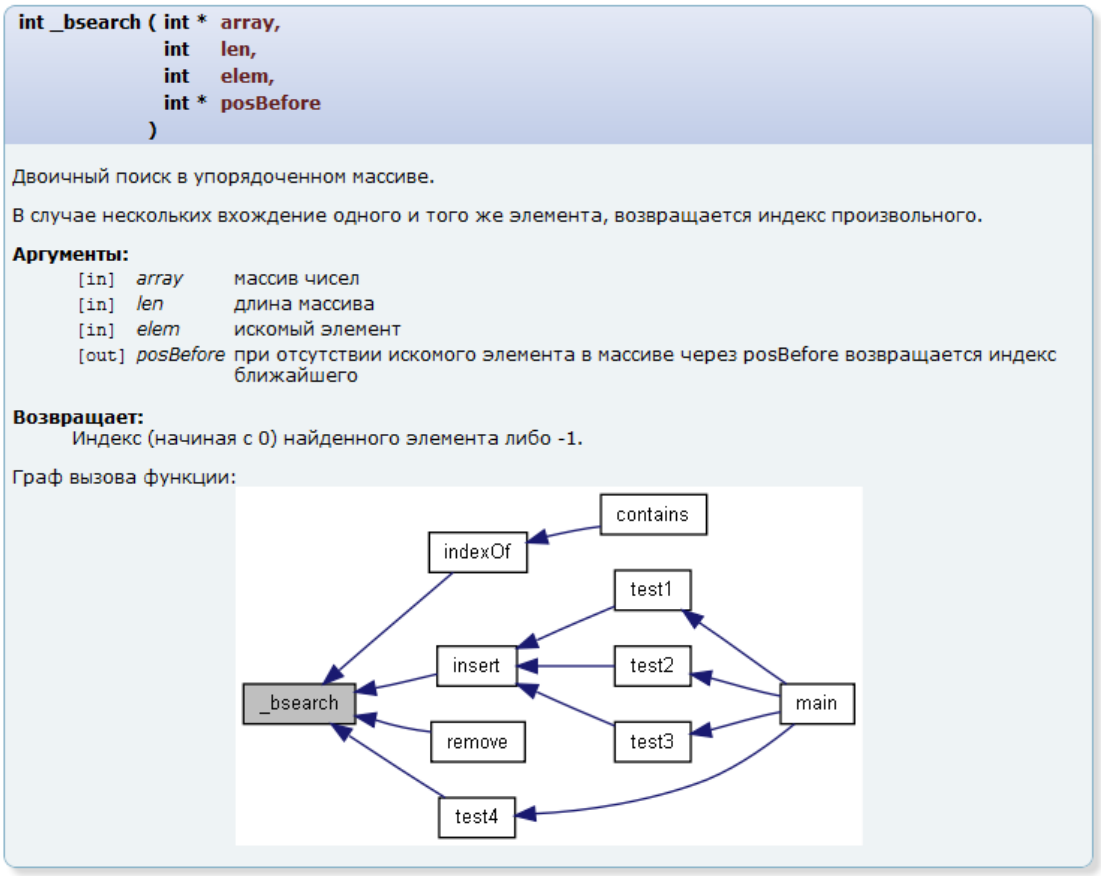
Документирование функций

Для документирования параметров функции используется команда param, возвращаемых значений - return:

```
/**
 * Двоичный поиск в упорядоченном массиве.
 *
 * В случае нескольких вхождение одного и того же элемента,
возвращается индекс произвольного.
 *
 * @param[in] array массив чисел
 * @param[in] len длина массива
 * @param[in] elem искомый элемент
 * @param[out] posBefore при отсутствии искомого элемента в массиве
 * через posBefore возвращается индекс ближайшего
 *
 * @return Индекс (начиная с 0) найденного элемента либо -1.
 */
 int _bsearch (int * array, int len, int elem, int * posBefore);
```

Задавать тип параметра in (входной) / out (выходной) не обязательно.

Фрагмент сгенерированного Doxygen описания данной функции (с диаграммой вызова данной функции):



Запуск Doxygen

В результате выполнения команды

```
doxygen -g <имя конфигурационного файла>
```

Doxygen создаст новый конфигурационный файл. Имя конфигурационного файла по умолчанию Doxygen.

Для использования Doxygen при документировании исходных файлов на Си может пригодиться опция «OPTIMIZE_OUTPUT_FOR_C = YES». Также может быть полезна опция «JAVADOC_AUTOBRIEF = YES» для интерпретации первой строки комментария как краткого комментария без указания команды @brief. Для выбора кодировки исходных текстов и локализации файла документации существуют параметры «INPUT_ENCODING» и «OUTPUT_LANGUAGE».

После редактирования файла можно использовать команду

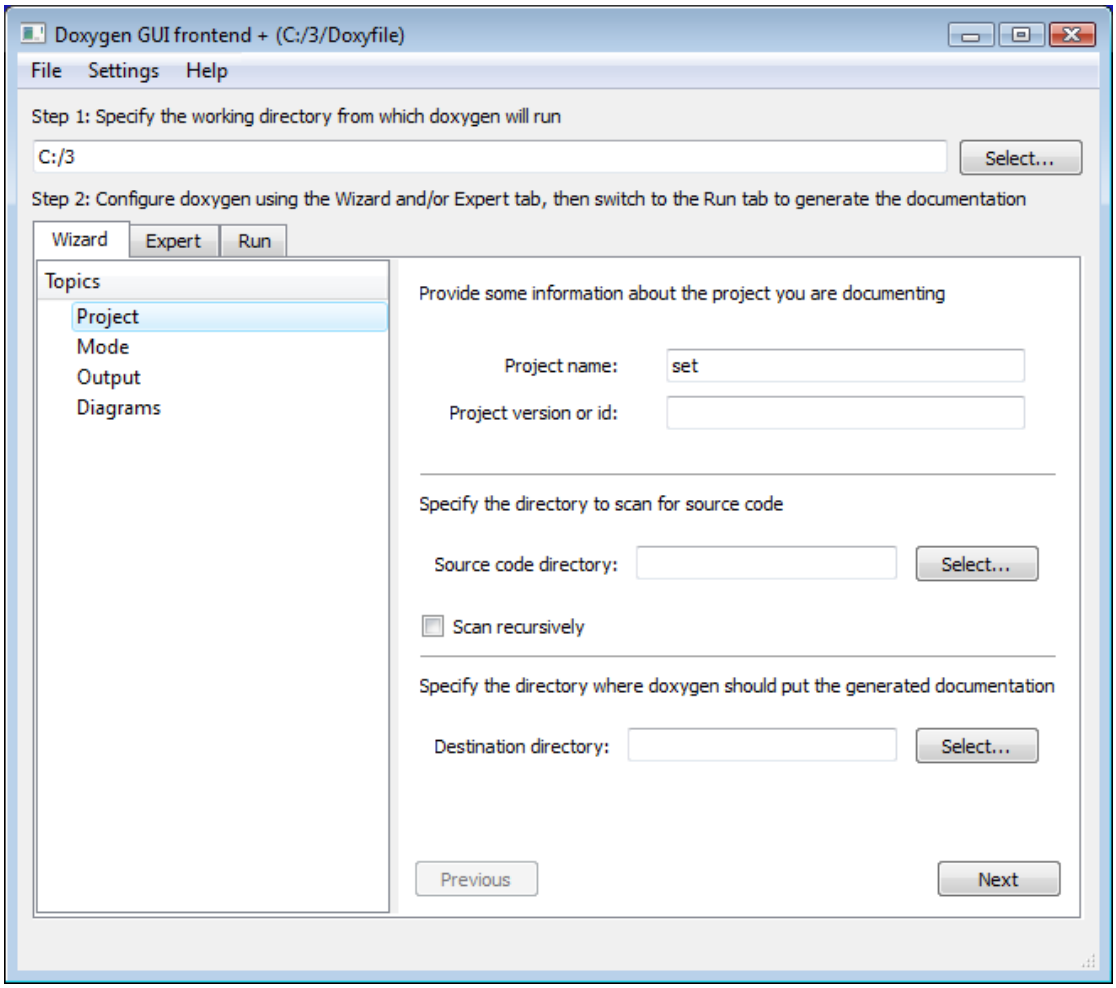
```
doxygen <имя конфигурационного файла>
```

для генерации документации.

В Makefile может быть создана соответствующая цель, отвечающая за генерацию документации:

```
doc:
	doxygen
```

Для визуального редактирования опций конфигурационного файла и запуска генерации документации в состав Doxygen входит графическая утилита doxywizard:



Полный перечень команд разметки и опций конфигурационного файла можно найти в документации Doxygen.