Тема 5. Инструменты статического и динамического анализа кода

5.1 Введение

Практически все программы содержат дефекты. Одна часть из них обнаруживается сравнительно легко, другая – почти никогда, как правило потому, что они практически никогда никак себя не проявляют. Некоторые дефекты, которые возникают относительно часто, остаются незамеченными даже просто потому, что они не воспринимаются как ошибки или не являются достаточно серьезными. Дефекты программ могут привести к ошибкам различных видов: от логических/функциональных (программа иногда выдает неправильные значения) до ошибок времени выполнения (в программе постоянно происходят сбои) или утечек ресурсов (производительность программы деградирует до тех пор, пока программа не «застынет» окончательно или аварийно завершится). Программы также могут содержать тонкие уязвимости безопасности, которые могут быть использованы злоумышленниками для получения контроля над компьютерными системами.

Исправление дефектов, которые появляются внезапно, может оказаться чрезвычайно дорогостоящим мероприятием, в частности, в конце жизненного цикла разработки, или что еще хуже, после развертывания системы. Многие простые дефекты в программах обнаруживаются современными компиляторами, однако преобладающим методом для обнаружения дефектов является, конечно же, **тестирование**. Оно обладает большим потенциалом для нахождения дефектов большинства типов, однако, тестирование является дорогостоящим и не гарантирует, что будут найдены **все** дефекты. Тестирование также проблематично, поскольку может быть применено только к исполняемому программному коду, то есть довольно поздно в процессе разработки. Альтернативы тестированию, такие как **анализ потоков данных** или **формальная верификация**, известны с 1970 гг., но до сих пор не получили широкого распространения в индустрии разработки, в последнее время появились несколько свободно-распространяемых и коммерческих инструментов для обнаружения дефектов во время компиляции, а не во время выполнения программы. Инструменты автоматизированного статического анализа могут быть использованы для поиска ошибок времени выполнения, утечек ресурсов и даже некоторых уязвимостей без выполнения программного кода.

5.2 Статический анализ программного кода

Такие языки, как C и в меньшей степени C++ предназначены в первую очередь для создания эффективного и переносимого кода, которой требует поддержки для того, чтобы избежать или справиться с ошибками времени выполнения. Скажем, в C нет никаких проверок операций границ массивов, а также того, что разыменование переменной-указателя возможно (т.е. она не равна NULL) или того, что приведение типов определено корректно, поэтому такие проверки должны быть обеспечены программистом. С другой стороны, мы должны убедиться, что они действительно не нужны, т.е. гарантировать, что

условия ошибки никогда не встретятся на практике.

Итак, под термином «**статический анализ**» мы имеем в виду автоматические методы исследования свойств программного кода без его фактического выполнения. Свойства, которые мы рассматриваем, включают те, что приводят к преждевременному прекращению или плохо определенным результатам работы, но не включает, например, синтаксические ошибки, простые ошибки типов, а также ошибки, связанные с функциональной корректностью программы. Таким образом, статический анализ можно использовать для проверки того, что работа программы не прервана преждевременно из-за неожиданных событий во время выполнения, но он не гарантирует, что программа выдает правильные результаты. В то время как статический анализ может быть использован, например, для проверки возникновения тупиковой ситуации, своевременности или досрочного завершения работы, есть и другие, более специализированные методы проверки таких свойств. Они базируются на тех же (или похожих) принципах. Статический анализ необходимо отличать от **динамического анализа**, который сосредоточен на исполнении программ и включает, например, тестирование, мониторинг производительности, локализацию ошибок и отладку.

Статический анализ в общем случае не гарантирует отсутствия ошибок времени выполнения, и, хотя он может уменьшить необходимость проведения тестирования и даже обнаружить ошибки, которые на практике не могли быть найдены с помощью тестирования, он **не предназначен для замены тестирования**. СЛАЙД 2.

Далее приведен далеко не полный перечень проблем времени выполнения, которые обычно не обнаруживаются с помощью традиционных компиляторов и трудно обнаруживаются при тестировании, но могут быть найдены путем статического анализа:

- Неправильное управление ресурсами: утечки ресурсов различных видов. Например, динамически выделяемой памяти, которая не освобождается; файлы или сокеты, которые должным образом не закрываются, если в них нет нужды в дальнейшем;
- Неверные операции: Деление на ноль, вызов математических функций с некорректными значениями (например, отсутствие положительных значений у логарифма), переполнение, антипереполнение или потеря значимости в арифметических выражениях, обращение к элементам массивов за их границами, разыменование нулевых указателей, освобождение памяти, которая уже освобождена;
- Мертвый код и данные: код и данные, которые не могут быть достигнуты или не используются. Это может быть всего лишь плохой стиль кодирования, но может также сигнализировать о наличии логических ошибок или опечаток в коде;
- Неполный код: Это подразумевает использование неинициализированных переменных, функций с неопределенными возвращаемыми значениями (вследствие, например, отсутствующих операторов *return*) и неполных операторов ветвления(например, отсутствует *case* в операторе *switch* или *else* в условном операторе).

Другие проблемы, выявляемые статическим анализатором, включают незавершение программы, неперехваченные исключения, условия гонок и т.д. СЛАЙД 3.

В дополнение к обнаружению ошибок, статический анализ можно использовать для получения более эффективного кода, в частности, для «безопасных» языков, таких как Java, где эффективность не является главной целью. Многих проверок, осуществляемых во время выполнения *Java*-программ, можно на практике избежать, если предоставлена определенная информация о поведении программы во время выполнения. Например, проверки индексов элементов массивов можно опустить, если мы знаем, что значения индексов ограничены значениями в указанных пределах. Статический анализ может предоставить такую информацию.

Статический анализ также может быть использован для вывода типов в нетипизированных или слабо типизированных языках или для проверки типов в языках без статических систем типов. И наконец, статический анализ может быть использован для отладки, для автоматической генерации тестов, для анализа влияния, обнаружения

вторжений и метрик программного обеспечения. Тем не менее, в этой части нашей лекции мы сосредоточим наше внимание на использование статического анализа для поиска дефектов и уязвимостей программ, которые обычно не обнаруживаются до тех пор, пока код не будет выполнен.

Самые интересные свойства, проверяемые при статическом анализе, неразрешимы, а это означает, что невозможно, даже теоретически, определить, является ли произвольная программа обладающей тем или иным свойством или нет. Как следствие, статическому анализу свойственна неточность. Он, как правило, делает вывод, что некоторое свойство (например, ошибка времени выполнения) **возможно**. Это означает, что (СЛАЙД 4):

1. если у программы есть конкретное свойство, анализатор, как правило, имеет возможность сделать вывод лишь о том, что «программа может иметь свойство». В некоторых (особых) случаях анализатор в состоянии сделать вывод о том, что «программа не обладает свойством».

2. если у программы нет конкретного свойства, то появляется шанс определить, что (а) наш анализатор на самом деле в состоянии вывести это (т.е. у программы нет свойства), но может также случиться, что (б) анализатор делает вывод, что программа может обладать свойством (но может и не обладать).

Если свойство, которое мы проверяем, есть дефект, то мы ссылаемся на случай (2 б) как **ложноположительный**. Таким образом, если анализатор сообщает, что программа может делить на ноль, то мы не можем сказать в целом, является ли это реальной проблемой (пункт 1) или это ложное срабатывание (пункт 2 б). Точность анализа определяет количество ложных срабатываний. Чем более неточным является анализ, тем выше вероятность генерации *false positives*.

К сожалению, точность обычно зависит от времени анализа. Чем точнее анализ, тем более он ресурсоемкий, и тем больше времени потребуется. Следовательно, точность анализа можно «обменять» на время. Это очень тонкий компромисс – если анализ быстрый, то будет получено большое количество ложных срабатываний, и в этом случае предупреждения не могут вызывать доверие. С другой стороны, очень точный анализ вряд ли прекратится через разумное время для больших и очень больших программ.

Один из способов избежать ложных срабатываний заключается в фильтрации результатов анализа с удалением тех ошибок, которыми они потенциально не являются (с некоторой долей вероятности). Однако это может привести к удалению положительных (реальных) дефектов. Это известно как «**ложная отрицательность**» (*false negatives*), т.е. действительная проблема, о которой не сообщается. Ложная отрицательность может происходить, по крайней мере, по двум причинам. Во-первых, если анализ слишком оптимистичен, то он делает неоправданные предположения о влиянии тех или иных операций, не принимая, например, во внимание, что *malloc()* может вернуть значение NULL. Во-вторых, может прийти к ложной отрицательности, если анализ является неполным, т.к. не принимает во внимание все возможные пути выполнения в программе. СЛАЙД 5.

Есть несколько общепризнанных методов, которые могут быть использованы как компромисс между точностью и временем анализа. СЛАЙД 6.

Потокозависимый анализ учитывает граф потока управления программы, в то время как **потоконезависимый** – нет. Потокозавсимый анализ, как правило, более точен, он может сделать вывод, например, о том, что имена *x* и *y* могут быть псевдонимами только после строки 10. В то же время, потоко-независимый анализ делает вывод только о том, что *x* и *y* могут быть псевдонимами (в любом месте в пределах своей области видимости). С другой стороны, потокозависимый анализ обычно более трудоемкий.

Путезависмый анализ рассматривает только допустимые пути внутри программы. Он учитывает значения переменных и логических выражений в условиях и циклах, чтобы сократить выполнение тех ветвей кода, которые не будут достигнуты на любом пути. **Путенезависимый** анализ учитывает все пути выполнения, даже те, которые не

достигаются никогда. Путезависимость обычно подразумевает более высокую точность, но, как правило, затрачивает больше времени.

Контекстно-зависимый анализ включает информацию о контексте, например, глобальные переменные и фактические параметры вызова функции при анализе функции. Это известно еще как **межпроцедурный анализ**, в отличие от **внутрипроцедурного анализа**, который анализирует функции без каких-либо предположений о контексте. Внутрипроцедурный анализ гораздо быстрее, но страдает от большей неточности, чем межпроцедурный.

Путе- и контекстная зависимость полагается на отслеживание возможных значений переменных программы: например, если мы не знаем значений переменных в логическом выражении условного оператора, то мы не знаем, выполнять ветвь *then* или *else*. Такой анализ значений может быть более или менее сложным; обычно ограничивается некоторым интервалом (например, $0 < x < 10$), но некоторые подходы полагаются на более общие отношения между несколькими переменными (например, $x > y + z$). Другим важным вопросом являются **псевдонимы** (или «**алиасы**»); при использовании указателей или массивов значение переменной может быть изменено путем изменения значения другой переменной. Без тщательного анализа переменных и псевдонимов мы, как правило, будем получать большое количество ложных срабатываний, или придется делать необоснованные оптимистические предположения о значениях переменных.

Неразрешимость свойств времени выполнения, подразумевает, что невозможно произвести анализ, который всегда бы находил все дефекты и не производил ложных срабатываний. Средства статического анализа (фреймворки) называют «**точными**» (или **консервативными** или **безопасными**), если они сообщают обо всех проверяемых дефектах, т.е. нет ложной отрицательности, но при этом могут быть ложные срабатывания. Традиционно большинство фреймворков для статического анализа направлены на полноту, пытаясь избежать чрезмерной отчетности ложных срабатываний. Тем не менее, большинство современных коммерческих систем (например, *Coverity* и *Klocwork* K7) не являются полными (т.е. они не найдут все фактические дефекты), а также обычно дают ложные срабатывания. СЛАЙД 7.

Иногда утверждают, что статический анализ можно применять к неполному коду (к отдельным файлам и/или процедурам). Хотя в этом есть доля правды, качество такого анализа может быть хуже. Например, если анализ не знает, как вызываются в существующем коде процедуры или подпрограммы, он должен, чтобы быть точным, допустить, что процедура вызывается произвольным образом, анализируя те пути, которые, вероятно, не будут выполняться, когда добавляется отсутствующий код. Это, как правило, приводит к ложной положительности. Аналогично незавершенный код может содержать вызов функции, который не доступен либо потому, что она еще не написана, либо является частью закрытой (как правило, проприетарной) библиотеки. Такой неполный код **можно** проанализировать, но это может привести к большому числу ложных срабатываний и/или ложной негативности в зависимости от того, делает ли анализ пессимистические или оптимистические предположения о недостающем коде.

С другой стороны («позитивной»), обычно нет необходимости предоставлять полный код отсутствующих функций или их вызовов. Часто бывает достаточно обеспечить заглушки или функции верхнего уровня, который имитирует эффект исследуемых свойств программы.

Инструменты для языков C/C++ (СЛАЙД 8):

- *lint (adlint, splint);
- *cppcheck*;
- *viva64*;
- и др.

Инструменты для Java:

- *JLint*;
- *FindBugs*

См. более или менее полный список инструментов, например, в [7].

Проиллюстрируем выполнение статического анализа программного кода на примере системы *cppcheck* (СЛАЙДЫ 9-12).

5.3 Динамический анализ программного кода

Динамический анализ кода – анализ программного обеспечения, выполняемый при помощи запуска программ на реальном или виртуальном процессоре (в отличие от статического анализа). СЛАЙД 13.

Инструменты динамического анализа обнаруживают программные ошибки в коде, запущенном на исполнение. При этом разработчик имеет возможность наблюдать или диагностировать поведение приложения во время его исполнения, в идеальном случае – непосредственно в целевой среде.

Во многих случаях в инструменте динамического анализа производится модификация исходного или бинарного кода приложения, чтобы установить ловушки, или **процедуры-перехватчики** (*hooks*), для проведения инструментальных измерений. С помощью этих ловушек можно обнаружить программные ошибки на этапе выполнения, проанализировать использование памяти, покрытие кода и проверить другие условия. Инструменты динамического анализа могут генерировать точную информацию о состоянии стека, что позволяет отладчикам отыскать причину ошибки. Поэтому, когда инструменты динамического анализа находят ошибку, то, скорее всего, это настоящая ошибка, которую программист может быстро идентифицировать и исправить. Следует заметить, что для создания ошибочной ситуации на этапе выполнения должны существовать точно необходимые условия, при которых проявляется программная ошибка. Соответственно, разработчики должны создать некоторый контрольный пример для реализации конкретного сценария.

Плюсы динамического анализа (СЛАЙД 14):

1) Редко возникают *false positives* – высокая продуктивность по нахождению ошибок.

2) Для отслеживания причины ошибки может быть произведена полная трассировка стека и среды исполнения.

3) Захватываются ошибки в контексте работающей системы.

Минусы динамического анализа:

1) Происходит вмешательство в поведение системы в реальном времени; степень вмешательства зависит количества используемых инструментальных вставок. Это не всегда приводит к возникновению проблем, но об этом нужно помнить при работе с критическим ко времени кодом.

2) Полнота анализа ошибок зависит от степени покрытия кода. Таким образом, кодовый путь, содержащий ошибку, должен быть обязательно пройден, а в контрольном примере должны создаваться необходимые условия для создания ошибочной ситуации.

Одним из известных инструментов динамического анализа программного кода является система *valgrind*. СЛАЙД 14.

Мы продемонстрируем работу одного из элементов этого инструмента – *memcheck*, т.к. с его помощью можно найти многие *run time* ошибки при работе с памятью. *Valgrind* запускает виртуальную машину, которая лучше следит за использованием памяти, чем реальная.

Ниже приводится список проверок памяти, которые можно выполнить с помощью *Valgrind* (СЛАЙД 15):

- Использование неинициализированной памяти

- Утечки памяти
- Переполнения памяти
- Повреждение стек
- Использование указателей памяти после того, как соответствующая память была освобождена
- Несоответствующие указатели в *malloc*/*free*.

Демонстрация (СЛАЙДЫ 16-20).

В первом примере (*valg1.c*) отлавливается непроинициализированная переменная. Во втором (*valg2.c*) – обнаруживается разыменование указателя памяти после освобождения памяти. Третий (*valg3.c*) – двойное освобождение памяти.

Верхний кадр в обеих трассах — это библиотечный код освобождения памяти, но мы уверены, что стандартная библиотека отлажена хорошо. Перенеся внимание на ту часть стека, которая относится к написанному мной коду, видим, что трасса указывает на строки 13 и 14 в исходном тексте, где действительно находятся оба обращения к *free*(*cmd*).

5.4 Целесообразность профилирования

Требования к производительности программ очень разные, но нет, конечно, секрета в том, что многие из них имеют очень жесткие требования по скорости выполнения. Например, медиапроигрыватели: пользователя должно обеспокоить, если тот воспроизводит видео на 50 или 75% от требуемой скорости.

Другие приложения, выполняющие длительные операции, предпочтительнее использовать в фоновом режиме, занимаясь в это время чем-нибудь еще. Хотя подобные приложения не имеют жестких ограничений по производительности, ее увеличение дает некоторые преимущества. Если таким приложением является видеокодировщик, было бы не плохо, чтобы он мог кодировать большее количество видеоинформации за определенный промежуток времени или кодировать с более высоким качеством за то же время.

В общем случае для всех приложений, кроме простейших, работает правило: чем выше производительность, тем более полезным и популярным будет приложение. По этой причине анализ производительности является (или должен являться) важнейшей задачей для многих разработчиков программного обеспечения.

К сожалению, большинство усилий, затрачиваемых на попытки ускорить приложение, проходит даром, поскольку разработчики часто оптимизируют свои программы на уровне одной или нескольких машинных инструкций (**микроуровень**) без полного исследования их функционирования на более высоком уровне абстракции (**макроуровень**). Можно затратить много усилий на уменьшение времени работы определенной функции в 2-3 раза (и это разумно), но если эта функция вызывается 1-2 раза (скажем при открытии и закрытии файла), то уменьшение времени ее выполнения с 200 мс до 100 мс существенно не скажется на общем времени выполнения программы.

Более эффективным использованием рабочего времени была бы оптимизация определенных участков программы, а именно тех, которые вызываются наиболее часто. Например, если программа затрачивает 50% времени на обработку строк, и функции можно оптимизировать на 10%, то это даст 5%-ное улучшение общей производительности.

Значит, весьма важно иметь точную информацию о том, где именно расходуется время в нашей программе для реальных входных данных, если мы на самом деле желаем эффективно оптимизировать наш код. Такие действия называются **профилированием кода**. Можно выполнять ручное профилирование кода, однако предпочтительнее поручить эту работу какому-либо автоматизированному средству, которое называется **профилировщиком**. СЛАЙД 21.

Существует достаточно большое количество программ для профилирования кода, но

мы сосредоточимся на одном из элементов GNU Toolchain, а именно на так называемом GNU-профайлере (*gprof*).

5.5 Профилирование в жизненном цикле программного обеспечения

Перед началом изучения использования *gprof* важно отметить место, которое профилирование занимает во всем цикле разработки. В общем случае код должен создаваться для решения следующих задач, перечисленных в порядке их важности (СЛАЙД 22):

- *Корректность работы программного обеспечения*. Понятно, что главная цель разработки – правильно выполнение программами своих функций. Вообще нет смысла проектировать сверхбыструю программу, если она не делает того, что должна делать! Корректность можно считать чем-то вроде «серой зоны», т.к., например, медиаплейер, работающий с 90% файлов, или воспроизводящий видео с иногда случающимися сбоями, все-таки можно использовать, однако корректность более важна, чем производительность.

- *Удобство в обслуживании программного обеспечения*. В общем случае, если программа не написана для удобного обслуживания, то даже если она вначале работает так, как нужно, но рано или поздно кто-нибудь (скажем, сам разработчик) оставит попытки исправить опечатки или добавить новые функции.

- *Производительность программного обеспечения*. Здесь самое место для профилировщика. Если программное приложение работает корректно, можно начать профилирование для ускорения его работы.

Допустим, у нас уже есть работающее приложение. Рассмотрим, как использовать *gprof* для точного измерения и поиска участков, на которые расходуется время при выполнении приложения, для того чтобы наиболее эффективно потратить усилия на оптимизацию.

Профилировщик *gprof* может профилировать приложения, написанные на языках C, C++, Pascal и Fortran 77. Приведенные ниже примеры используют «плоский» C.

5.6 Использование *gprof*

Использование *gprof* – не очень сложный процесс (СЛАЙД 23). Нам нужно лишь выполнить три следующих шага:

1. Разрешить добавлять информацию для профилировщика при компиляции программного кода.
2. Выполнить программный код для того, чтобы создать профильные данные.
3. Запустить *gprof* с передачей ему файла с профильными данными, сгенерированными на 2-ом шаге.

На последнем шаге будет создан файл, который содержит результаты анализа в читабельной форме (СЛАЙД 24). Помимо прочей информации, он также содержит две таблицы (простой профиль и граф вызовов). **Простой профиль** содержит обзор временно́й информации по работе функций. Это может быть, например, время, затраченное определенной функцией, количество ее вызовов и т.д. Со своей стороны **граф вызовов** показывает все вызовы функций, которые привели к данной функции, какие функции вызываются из данной функции и т.д. И при этом способе можно вычислить время, затраченное подпрограммами.

Попробуем разобраться, как выполняются перечисленные выше три шага, на практическом примере. Следующий код будет использован и далее (СЛАЙДЫ 25 и 26).

Важно отметить, что показанный код содержит функции с длительно выполняющимися циклами *for*.

Итак, на первом шаге нам надо убедиться, что опция поддержки профилирования (*-pg*) включена при вызове компилятора, у нас – *gcc*. Вот, что сообщает *man* по поводу этой опции (СЛАЙД 27).

```
-pg : Generate extra code to write profile information
suitable for the analysis program gprof. You must use this
option when compiling the source files you want data about, and
you must also use it when linking.
```

Итак, компилируем и компонуем наш код (СЛАЙД 28):

```
$ gcc -Wall -pg test_gprof.c test_gprof_new.c -o test_gprof
$
```

Обратим внимание на то, что опцию ‘*-pg*’ можно использовать: с командой *gcc*, которая только компилирует (опция *-c*); с командой *gcc*, которая только компонует (опция *-o* с объектными модулями); с командой, которая делает и то, и другое, как в нашем случае.

На втором шаге двоичный файл, созданный в результате выполнения первого шага, запускается так, чтобы могла быть сгенерирована профильная информация.

```
$ ls
test_gprof  test_gprof.c  test_gprof_new.c

$ ./test_gprof

...

$ ls
gmon.out  test_gprof test_gprof.c  test_gprof_new.c

$
```

Как видно, после выполнения двоичного модуля в текущем каталоге появился новый файл ‘*gmon.out’*.

Важно отметить, что если во время выполнения программы изменяется текущий рабочий каталог (например, командой *chdir*), то файл *gmon.out* будет появляться в новом текущем рабочем каталоге. Кроме того, у нашей программы должны быть достаточные полномочия (*permissions*) на создание файла *gmon.out*.

На третьем шаге запускается инструмент *gprof*, которому передается в качестве аргументов имя исполняемого модуля программы и файл ‘*gmon.out*’. Будет создан файл с результатами анализа, который содержит всю желаемую для профилирования информацию.

```
$ gprof test_gprof gmon.out > analysis.txt
```

Можно явно указать имя выходного файла, как это сделали мы, или информация выводится на стандартный вывод.

```
$ ls
analysis.txt  gmon.out  test_gprof  test_gprof.c  test_gprof_new.c
```

Видно, что появился файл ‘*analysis.txt*’. Его содержимое представлено на СЛАЙДАХ 29-35.

Итак, вся интересующая информация представлена в этом файле. Можно вывести его содержимое. Действительно весь файл разбит на две части (простой профиль и граф вызовов). Содержимое каждой из них достаточно хорошо поясняется в самом файле.

Внимательный слушатель/читатель, наверное, заметил, что здесь никакой

информации, скажем, о функции *printf*. Дело в том, что стандартная библиотека языка C (*libc*) в нашем случае не компилировалась и не компоновалась с флагом *-pg*. Значит, никакой информации для профилировщика получить не удастся. Единственный способ – получить нужную сборку библиотеки с указанием опции *-pg*.

5.7 Кастомизация вывода профильной информации

Существует несколько флагов, которые позволяют изменить формат вывода аналитической информации (СЛАЙДЫ 36-41).

1. Можно подавить вывод статических функций.

```
$ gprof -a test_gprof gmon.out > analysis.txt
$ cat analysis.txt
...
$
```

Видно, что теперь не выводится информация, относящаяся к функции *func2*.

2. Как мы видели ранее, *gprof* выдает очень много подробной информации. В том случае, когда такого количества информации не требуется, можно ее подавить с помощью флага *-b*.

```
$ gprof -b test_gprof gmon.out > analysis.txt
```

3. В том случае, когда нам достаточно только простого профиля, мы можем использовать флаг *-p*. Его рекомендуется использовать совместно с флагом *-b*.

```
$ gprof –p -b test_gprof gmon.out > analysis.txt
```

4. Вывод информации по конкретной функции в простом профиле выполняется указанием имени функции с опцией *-p*.

```
$ gprof –pfunc2 -b test_gprof gmon.out > analysis.txt
```

Как видно, выдается информация только по функции *func2*.

5. Если не требуется информация о простом профиле, то ее вывод может быть подавлен указанием опции *-P*:

```
$ gprof -P -b test_gprof gmon.out > analysis.txt
```

Если просмотреть содержимое файла, то видно, что выводится лишь граф вызовов. Похожим образом можно организовать вывод простого профиля, за исключением некоторой функции. Для этого так же используется опция *-P* совместно с именем функцию, которую нужно исключить из вывода.

```
$ gprof –Pfunc2 -b test_gprof gmon.out > analysis.txt
```

В этом примере мы пытаемся исключить функцию ‘*func2*', передавая ее имя профилировщику совместно с опцией *-P*. Если мы выведем содержимое текстового файла, то простой профиль будет показан без функции *func2*.

6. Можно выводить информацию о графе вызовов с использованием опции *-q*:

```
gprof -q -b test_gprof gmon.out > analysis.txt
```

Если теперь вывести содержимое *analysis.txt*, то можно убедиться в том, что там действительно лишь информация о графе вызовов.

7. Можно выводить в графе вызовов информацию только об определенной функции. Это выполняется указанием имени функции после флага *-q*.

```
$ gprof –qfunc2 -b test_gprof gmon.out > analysis.txt
```

Будет выведена информация только о функции *func2*.

8. Если вообще не требуется информация о графе вызовов, то следует использовать опцию *-Q*.

```
$ gprof -Q -b test_gprof gmon.out > analysis.txt
```

При выводе файла аналитики будет показана только информация о простом профиле.

Так же можно подавить вывод информации в графе вызовов об определенной функции путем указания имени функции вместе с флагом *-Q*.

```
$ gprof –Qfunc2 -b test_gprof gmon.out > analysis.txt
```

Здесь мы подавим вывод информации о функции *func2*. В этом можно будет убедиться при демонстрации содержимого файла аналитики.

9. Мы можем получить листинг «**аннотированного исходного кода**» (*annotated source*), в котором выводится исходный код приложения с отметками о количестве вызовов каждой функции.

Для использования этой возможности нужно откомпилировать исходный код с разрешенной отладочной информацией, для того чтобы исходный код был помещен в исполняемый файл:

```
$ gprof –A test_gprof gmon.out > analysis.txt
```

Более или менее полный список инструментов профилирования см. в [23].

5.8 Покрытие кода

Каково тестовое покрытие нашего кода? Существуют ли в нем строки, которые не выполнялись при прогоне тестов? Вместе с *gcc* поставляется программа *gcov*, которая подсчитывает, сколько раз выполнялась каждая строка во время работы программы. СЛАЙД 42.

Процедура выглядит следующим образом:

- Добавить в переменную *CFLAGS* для *gcc* флаги *-fprofile-arcs* и *-ftest-coverage*. Стоит также добавить флаг -O0 , чтобы оптимизатор не исключал строки.

- В ходе работы программы для каждого исходного файла *yourcode.c* порождается один или два файла: *yourcode.gcda* и *yourcode.gcno*.

- Команда *gcov yourcode.gcda* выводит в *stdout* процентную долю исполняемых строк, которые действительно исполнялись во время прогона программы (объявления, директивы # *include* и т. п. не считаются), и создает файл *yourcode.c.gcov*.

- В первом столбце таблицы в файле *yourcode.c.cov* показано, сколько раз каждая исполняемая строка исполнялась тестами, а строки, которые не исполнялись ни разу, помечены большим жирным маркером #####. Именно на них следует обращать внимание при написании очередного теста.

Использование *gcov* мы демонстрируем на коде, показанном на СЛАЙДЕ 43.

Литература и дополнительные источники к Теме 5

1. Lint man page - http://www.unix.com/man-page/FreeBSD/1/lint
2. cppcheck - http://cppcheck.sourceforge.net/
3. FindBugs - http://findbugs.sourceforge.net/
4. JLint - http://jlint.sourceforge.net/
5. Klocwork TruePath - http://www.klocwork.com/products/insight/klocwork-truepath/
6. Coverity SAVE - http://www.coverity.com/products/coverity-save.html
7. List of tools for static code analysis - http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis
8. Выявление ошибок работы с памятью при помощи valgrind - http://www.opennet.ru/base/dev/valgrind_memory.txt.html
9. Использование статического и динамического анализа для повышения качества продукции и эффективности разработки - http://www.swd.ru/print.php3?pid=828
10. Dynamic program analysis - http://en.wikipedia.org/wiki/Dynamic_program_analysis
11. PurifyPlus - http://unicomsi.com/products/purifyplus/
12. Intel Inspector XE - https://software.intel.com/en-us/intel-inspector-xe
13. Clang Static Analyzer - http://clang-analyzer.llvm.org/
14. GNU Binutils - http://www.gnu.org/software/binutils/
15. GNU gprof - http://sourceware.org/binutils/docs-2.23.1/gprof/index.html
16. Ускорение кода при помощи GNU-профайлера - http://www.ibm.com/developerworks/ru/library/l-gnuprof/
17. Романенко, А. А. Профилирование программ. - http://ccfit.nsu.ru/arom/data/PP_ICaG/06_Profiling_txt.pdf
18. Sysprof, System-wide Performance Profiler for Linux - http://sysprof.com/
19. OProfile - A System Profiler for Linux (News) - http://oprofile.sourceforge.net/
20. Руководство по настройке производительности - http://www.regatta.cs.msu.su/doc/usr/share/man/info/ru_RU/a_doc_lib/aixbman/prftungd/2365cf1.htm
21. VTune Amplifier XE by Intel Corporation - http://software.intel.com/en-us/intel-vtune-amplifier-xe
22. Java Profiler - JProfiler - http://www.ej-technologies.com/products/jprofiler/overview.html
23. List of performance analysis tools - http://en.wikipedia.org/wiki/List_of_performance_analysis_tools
24. Gcov docs - http://gcc.gnu.org/onlinedocs/gcc/Gcov.html