Версия 0.91 release candidate 2 от 06.02.2014. Возможны незначительные изменения.

Тема 1. Введение в системное программирование

1.1. Введение в системное программирование.

Главные вопросы, которые мы обсудим, представлены на СЛАЙДЕ 1.

Прикладная программа – программа, предназначенная для решения задачи или класса задач в определенной предметной области (области применения).

Системная программа – программа, предназначенная для поддержания работоспособности прикладных и других системных программ или повышения эффективности из использования.

В соответствии с этой терминологией, **системное программирование** – это процесс разработки системных программ (в том числе, управляющих и обслуживающих).

С другой стороны, система – единое целое, состоящее из множества компонентов и множества связей между ними. Следовательно, системное программирование – это разработка программных систем сложной структуры.

Эти два определения не противоречат друг другу, так как разработка программ сложной структуры ведется именно для обеспечения работоспособности или повышения эффективности программных систем.

Подразделение программного обеспечения (далее – ПО) на системное и прикладное на сегодня является в некоторой степени устаревшим. Сейчас говорят о трех классах ПО (СЛАЙД 2):

- Системное.
- Промежуточное.
- Прикладное.

Промежуточное ПО (*middleware*) определяется как совокупность программ, осуществляющих управление вторичными (конструируемыми самим ПО) ресурсами, ориентированными на решение определенного (широкого) класса задач. К такому ПО относятся менеджеры транзакций, системы управления базами данных, серверы коммуникаций и другие программные серверы. С точки зрения инструментальных средств разработки промежуточное ПО ближе к прикладному, так как не работает напрямую с первичными ресурсами, а использует для этого сервисы, предоставляемые системным ПО.

С точки зрения алгоритмического обеспечения и технологий разработки промежуточное ПО ближе к системному, так как всегда является сложной программной системой многократного и многоцелевого использования и в нем применяются те же или сходные алгоритмы, что и в системном ПО.

Современные тенденции развития ПО состоят в снижении объема как системного, так и прикладного программирования. Основная часть работы программистов выполняется в промежуточном ПО. Снижение объема системного программирования определено современными концепциями ОС, объектно-ориентированным подходом и наиболее популярной гибридной архитектурой ядра, а также микроядерной архитектурой ОС, в соответствии с которыми большая часть функций выносится в утилиты, которые можно отнести и к промежуточному ПО. Снижение объема прикладного программирования обусловлено тем, что современные продукты промежуточного ПО предлагают все больший набор инструментальных средств и шаблонов для решения задач своего класса.

Версия 0.91 release candidate 2 от 06.02.2014. Возможны незначительные изменения.

Значительная часть системного и практически все прикладное ПО пишется на языках высокого уровня, что обеспечивает сокращение расходов на их разработку/модификацию и переносимость.

Системное ПО подразделяется на системные управляющие программы и системные обслуживающие программы. См. также СЛАЙД 3.

Управляющая программа – системная программа, реализующая набор функций управления, который включает в себя управление ресурсами и взаимодействие с внешней средой, восстановление работы системы после проявления неисправностей в технических средствах.

Программа обслуживания (**утилита**) – программа, предназначенная для оказания услуг общего характера пользователям и обслуживающему персоналу программных систем конкретного назначения.

Управляющая программа совместно с набором необходимых для эксплуатации системы утилит составляют **операционную систему** (ОС).

Кроме входящих в состав ОС утилит могут существовать и другие утилиты (того же или стороннего производителя), выполняющие дополнительное (опциональное) обслуживание. Как правило, это утилиты, обеспечивающие разработку программного обеспечения для операционной системы.

Система программирования – система, образуемая языком программирования, компилятором или интерпретатором программ, представленных на этом языке, соответствующей документацией, а также вспомогательными средствами для подготовки программ к форме, пригодной для выполнения.

При разработке программ, а тем более – сложных, используется принцип модульности, разбиения сложной программы на составные части, каждая из которых может подготавливаться отдельно. Модульность является основным инструментом структурирования программного изделия, облегчающим его разработку, отладку и сопровождение.

1.2. Модульная разработка программ.

Программный модуль – программа или функционально завершенный фрагмент программы, предназначенный для хранения, трансляции, объединения с другими программными модулями и загрузки в оперативную память.

При выборе модульной структуры должны учитываться следующие основные соображения: (СЛАЙД 4)

- Функциональность – модуль должен выполнять законченную функцию
- Несвязность – модуль должен иметь минимум связей с другими модулями, связь через глобальные переменные и области памяти нежелательна
- Специфицируемость – входные и выходные параметры модуля должны четко формулироваться

Программа пишется в виде исходного модуля.

Исходный модуль (ИМ) – программный модуль на исходном языке, обрабатываемый транслятором и представляемый для него как целое, достаточное для проведения трансляции.

Первым (не для всех языков программирования обязательным) этапом подготовки программы является обработка ее **Макропроцессором** (или препроцессором). Макропроцессор обрабатывает текст программы, и на выходе получается его новая редакция текста. В большинстве систем программирования Макропроцессор совмещен с транслятором, и для программиста его работа и промежуточный ИМ прозрачны.

Следует иметь в виду, что Макропроцессор выполняет обработку текста. Это означает, с одной стороны, что он «не понимает» операторов языка программирования и «не знает» о переменных программы, с другой, что все операторы и переменные Макроязыка (тех выражений в программе, которые адресованы Макропроцессору) в

промежуточном ИМ уже отсутствуют и для дальнейших этапов обработки «не видны».

Так, если Макропроцессор заменил в программе некоторый текст *A* на текст *B*, то транслятор уже видит только текст *B*, и не знает, был этот текст написан программистом «своей рукой» или подставлен Макропроцессором.

Следующим этапом является трансляция.

Трансляция – преобразование программы, представленной на одном языке, в другое ее представление (например, на другом языке программирования). В определенном смысле мы должны получить программу, равносильную (семантически эквивалентную) первой. Однако, как правило, выходным языком транслятора является машинный язык целевой вычислительной системы (**Целевая** ВС – та ВС, на которой программа будет выполняться).

Машинный язык – язык программирования, предназначенный для представления программы в форме, позволяющей выполнять ее непосредственно техническими средствами обработки информации.

Трансляторы – общее название для программ, осуществляющих трансляцию. Иногда подразделяются на ассемблеры, компиляторы и интерпретаторы – в зависимости от исходного языка программы, которую они обрабатывают. Ассемблеры, как правило, работают с языками Ассемблера, компиляторы – с языками высокого уровня.

Автокод – символьный язык программирования, предложения которого по своей структуре в основном подобны командам и обрабатываемым данным конкретного машинного языка.

Язык Ассемблера – язык программирования, который представляет собой символьную форму машинного языка с рядом возможностей, характерных для языка высокого уровня (обычно включает в себя макросредства).

Язык высокого уровня – язык программирования, понятия и структура которого удобны для восприятия человеком.

Объектный модуль (ОМ) – программный модуль, получаемый в результате трансляции исходного модуля.

Поскольку результатом трансляции является модуль на языке, близком к машинному, в нем уже не остается признаков того, на каком исходном языке был написан программный модуль. Это создает принципиальную возможность создавать программы из модулей, написанных на разных языках. Специфика исходного языка, однако, может сказываться на физическом представлении базовых типов данных, способах обращения к процедурам/функциям и т.п. Для совместимости разноязыковых модулей должны выдерживаться общие соглашения. Большая часть ОМ – команды и данные машинного языка именно в той форме, в какой они будут существовать во время выполнения программы. Однако программа в общем случае состоит из многих модулей. Поскольку транслятор обрабатывает только один конкретный модуль, он не может должным образом обработать те части этого модуля, в которых запрограммированы обращения к данным или процедурам, определенным в другом модуле. Такие обращения называются **внешними ссылками**. Те места в объектном модуле, где содержатся внешние ссылки, транслируются в некоторую промежуточную форму, подлежащую дальнейшей обработке. Говорят, что объектный модуль представляет собой программу на машинном языке с неразрешенными внешними ссылками.

Разрешение внешних ссылок выполняется на следующем этапе подготовки, который обеспечивается **Редактором Связей** (или Компоновщиком). Редактор Связей соединяет (компонует) вместе все объектные модули, входящие в программу. Поскольку Редактор Связей «видит» уже все компоненты программы, он имеет возможность обработать те места в объектных модулях, которые содержат внешние ссылки. Результатом работы Редактора Связей является загрузочный модуль.

Загрузочный модуль (ЗМ) – программный модуль, представленный в форме, пригодной для загрузки в оперативную память для выполнения.

ЗМ сохраняется в виде файла на внешней памяти. Для выполнения программа должна быть перенесена (загружена) в оперативную память. Иногда при этом требуется некоторая дополнительная обработка (например, настройка адресов в программе на ту область оперативной памяти, в которую программа загрузилась). Эта функция выполняется **Загрузчиком**, который обычно входит в состав ОС.

Возможен также вариант, когда редактирование связей выполняется при каждом запуске программы на выполнение и совмещается с загрузкой. Это делает **Связывающий Загрузчик**. Вариант связывания при запуске более расходный, т.к. затраты на связывание тиражируются при каждом запуске. Но он обеспечивает:

- большую гибкость в сопровождении, так как позволяет менять отдельные ОМ программы, не меняя остальных модулей;
- экономию внешней памяти, т.к. ОМ, используемые во многих программах, не копируются в каждый ЗМ, а хранятся в одном экземпляре.

Вариант интерпретации подразумевает прямое исполнение исходного модуля.

Интерпретация – реализация смысла некоторого синтаксически законченного текста, представленного на конкретном языке.

Интерпретатор читает из ИМ очередное предложение программы, переводит его в машинный язык и выполняет. Все затраты на подготовку тиражируются при каждом выполнении, следовательно, интерпретируемая программа принципиально менее эффективна, чем транслируемая. Однако интерпретация обеспечивает удобство разработки, гибкость в сопровождении и переносимость.

Не обязательно подготовка программы должна вестись на той же ВС и в той же ОС, в которых программа будет выполняться. Системы, обеспечивающие подготовку программ в среде, отличной от целевой, называются **кросс-системами**. В кросс-системе может выполняться вся подготовка или ее отдельные этапы (СЛАЙД 4):

- Макрообработка и трансляция.
- Редактирование связей.
- Отладка.

Типовое применение кросс-систем – для тех случаев, когда целевая вычислительная среда просто не имеет ресурсов, необходимых для подготовки программ, например, встроенные системы. Программные средства, обеспечивающие отладку программы на целевой системе можно также рассматривать как частный случай кросс-системы.

1.3. Обязанности системных программистов.

Некоторые обязанности системного программиста (согласно профессиональному стандарту) представлены на СЛАЙДЕ 6, что показывает актуальность этой специализации и предмета нашего изучения.

1.4. Особенности дальнейшего процесса изучения СП.

Все примеры здесь и далее, если не оговорено особо, приводятся в виртуальной машине *Oracle VirtualBox* с гостевой операционной системой *Debian GNU/Linux*. Эта UNIX-подобная система выбрана по нескольким причинам. Во-первых, очень развитые **средства командной строки** (*Command Language Interface*, CLI). Именно путем подачи команд совершается большинство действий в UNIX.

Во-вторых, многообразие графических оболочек. Дело в том, что **графический интерфейс пользователя** (GUI) здесь не является частью ОС; поэтому пользователь может выбрать тот внешний вид и функциональность оконной системы, которые ему удобнее. Хотя для нас это не очень важно, т.к. мы в основном используем CLI.

В-третьих, подавляющее большинство программ в *NIX-системах распространяется в исходных текстах и часто собирается уже на вычислительной системе конечного пользователя, что делает, например, компилятор языка С обязательной частью почти любого дистрибутива ОС UNIX и UNIX-подобных систем.

Версия 0.91 release candidate 2 от 06.02.2014. Возможны незначительные изменения.

Для разработки мы будем пользоваться GNU *Toolchain* – набор созданных в рамках проекта GNU пакетов программ, необходимых для компиляции и генерации выполняемого кода из исходных текстов. Являются стандартным средством разработки программ и ядра ОС Linux.

Состав приведен на СЛАЙДЕ 7.

1.5.Автоматическое тестирование в CUnit

На сегодня любое практически полезное программное обеспечение, в том числе системное и промежуточное, находится в состоянии непрерывной доработки – разработчики выпускают новые версии программ, содержащие исправления ошибок, реализацию новых требований заказчика, изменения, обусловленные обновлениями сопутствующего ПО, т.е. появлением новых версий операционных систем и библиотек. В такой ситуации «ручное» тестирование каждой новой модификации становится крайне неэффективным вследствие большого объема трудозатрат и рутинности процесса. На помощь приходит **автоматическое тестирование**. Его смысл заключается в использовании программных средств для выполнения тестов и контроля результатов их выполнения. Это помогает сократить время и упростить процесс тестирования. Известными разновидностями автоматического тестирования являются (СЛАЙД 9):

1. **Системное тестирование** – тестирование системы как «черного ящика» (без знаний деталей реализации) с помощью сценариев, близких к сценариям использования системы в реальной работе.

2. **Модульное тестирование** (**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. Эта разновидность реализуется в виде набора тестов, описывающих сценарии вызова функций и сравнивающих фактические результаты работы с ожидаемыми результатами при помощи набора ***проверок*** (*assert*), входящих в состав системы модульного тестирования. Тесты являются функциями, написанными на языке реализации, и включаются в состав исходных текстов программы. Типичная система тестирования включает подсистему автоматического запуска тестов (часто – с графическим пользовательским интерфейсом). Для проверки корректности программы после внесения очередных модификаций необходимо лишь активировать запуск тестов.

Одной из широко-распространенных практик разработки является **разработка через тестирование** (*test-driven development*, *TDD*). В его основе лежит идея реализации и/или доработки функциональности программного обеспечения минимальными частями. 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 (СЛАЙД 10):

- Инкрементально-пополняемый набор тестов снижает риск внесения ошибки при модификации программного кода.
- Автоматическая проверка тестов позволяет уменьшить издержки по сравнению с «ручным» тестированием.
- Наличие набора автоматических тестов положительно влияет на психологическое состояние программистов, модифицирующих программный код.
- Идея «сначала тест, потом код» 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 обычно к коду более высокого качества.

Версия 0.91 release candidate 2 от 06.02.2014. Возможны незначительные изменения.

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 С-программисту может пригодиться стандартная библиотечная функция *assert*(*cond*), которая завершает выполнение программы аварийно при невыполнении условия *cond*. См. СЛАЙД 11.

Функция *assert()* обычно используется, чтобы убедиться в правильном выполнении программы, причем выражение составляется таким образом, что оно истинно только при отсутствии ошибок.

Пример использования этой функции также приведен на СЛАЙДЕ 11.

Но в реальных проектах завершение работы программы при неверных аргументах ничем не отличиться от того же «зависания», поэтому *assert()* следует использовать только для отладки. К примеру, мы пишем программу и полагаем, что она должна работать корректно. Однако при запуске наблюдается ошибка сегментации. Следует включить вызовы *assert()* в наиболее опасные места, чтобы проверить значения переменных и указателей, возвращаемые значения функций и т.д. Таким образом, у нас повышается шанс найти ошибку в коде

Одной из распространенных систем модульного тестирования программ, написанных на языке С, является CUnit. В CUnit тест является функцией языка С, не имеющей параметров и возвращаемого значения. Логически каждый тест является листом следующей иерархии (СЛАЙД 12):

Реестр тестов (*test registry*)

Набор тестов (*test suite*)

Тест (*test*)

Для тестов, входящих в набор, могут быть заданы функция **установки** (*setup*) и функция **очистки** (*teardown*) контекста. Функция установки вызывается единовременно 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 определенных реестров или наборов, так и отдельные тесты.

Реестры

CUnit создает один реестр тестов по умолчанию (СЛАЙД 13). 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.

Функции *CU_initialize_registry* и *CU_cleanup_registry* работают с 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*.

Об особенностях их использования можно получить дополнительную информацию в документации по CUnit.

Наборы

Функция (СЛАЙД 14):

```
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 должен быть инициализирован до вызова данной функции.

Имена наборов должны быть уникальны в рамках реестра.

Функции инициализации и очистки необязательны, они 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, то в качестве значения соответствующего указателя нужно передать *NULL*.

Функция возвращает указатель на созданный набор. Впоследствии указатель можно использовать для добавления тестов в требуемый набор. В случае ошибки возвращается *NULL* и устанавливается значение кода ошибки, который может быть получен 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 официальной документации.

Тесты

Функция (СЛАЙД 15):

```
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_suite()*. Имена тестов должны быть уникальными в пределах набора. Тип *CU_testFunc* определен как:

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

pTestFunc не может быть *NULL*. Функция *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 требуемый набор:

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

Проверки

В таблице приведены некоторые проверки CUnit (СЛАЙД 16):

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,`	Проверка строк на равенство

Название макроса	Контролируемое условие
`expected)`	
`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

```
CU_ErrorCode CU_basic_run_tests(void);
```

Запуск всех тестов. Возвращает код **первой** ошибки, возникшей при запуске тестов (СЛАЙД 17).

```
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 из заданного набора. Возвращает код **первой** ошибки, возникшей при запуске теста.

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

```
void CU_console_run_tests(void);
```

Функция

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

возвращает результаты запуска тестов. Возвращаемое значение – указатель на структуру *CU_RunSummary*, содержащую статистику запуска тестов (имена полей этой структуры самодокументируемы, СЛАЙД 18):

```
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

Версия 0.91 release candidate 2 от 06.02.2014. Возможны незначительные изменения.

пройденной проверке (СЛАЙД 19):

```
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

Рассмотрим функции по работе с множеством целых чисел (структура *Set*). См. СЛАЙДЫ 20-23.

Автоматические тесты могут созданы как на функции интерфейса библиотеки (СЛАЙД 24), так и на функцию *_bsearch*, используемую реализацией библиотеки (СЛАЙД 25).

Пример кода помещения тестов в набор и их автоматического запуска см. на СЛАЙДЕ 26.

Пример кода для запуска среды интерактивной проверки CUnit (СЛАЙД 27).

Более или менее полный список фреймоворков юнит-тестирования программ на C и других языках см. в [17]. СЛАЙД 28.

Литература и дополнительные источники к Теме 1

1. Oracle VM VirtualBox - https://www.virtualbox.org/
2. Debian – Универсальная Операционная Система - http://www.debian.org/
3. Операционная система GNU - http://www.gnu.org/
4. GNU toolchain - http://ru.wikipedia.org/wiki/GNU_toolchain
5. Кузнецов, А. С. Операционные системы и системное программное обеспечение: учеб. пособие / А. С. Кузнецов, И. В. Ковалев. – Красноярск: ИПЦ КГТУ, 2005. – 302 с.
6. Курячий, Г.В. Операционная система Linux: Курс лекций. Учебное пособие / Г.В.Курячий, К.А. Маслинский. М.: ДМК-Пресс, 2010. – 348 с. – Доступ через электронную библиотечную систему Издательства «Лань» - http://e.lanbook.com/
7. Профессиональный стандарт. Системный программист - http://www.apkit.ru/committees/education/PS_SP_4.0.pdf
8. CUnit - A Unit Testing Framework for C - http://cunit.sourceforge.net/
9. CuTest: C Unit Testing Framework - http://cutest.sourceforge.net/
10. Check: a unit test framework for C - http://check.sourceforge.net/
11. JUnit - A programmer-oriented testing framework for Java - http://junit.org/
12. CppUnit - C++ port of JUnit - http://apps.sourceforge.net/mediawiki/cppunit/
13. Тестирование программного обеспечения: модульные тесты - http://openquality.ru/software-testing/unit-tests.php
14. Unit Testing Guidelines - http://geosoft.no/development/unittesting.html
15. OpenSourceTesting.org lists many unit testing frameworks, performance testing tools and other tools programmers/developers may find useful - http://opensourcetesting.org/
16. Testing Framework - http://c2.com/cgi/wiki?TestingFramework
17. List of unit testing frameworks - https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks
18. googletest - Google C++ Testing Framework - Google Project Hosting -

Версия 0.91 release candidate 2 от 06.02.2014. Возможны незначительные изменения.

http://code.google.com/p/googletest/