Знакомство с C-классами

Авторы: разработчики компании Penrillian
Перевод: Александр Смирнов

Опубликовано в Symbian Developer Network

Версия: 1.0 – Март 2009

1 Знакомство с C-классами

Стандарты Symbian OS определяют некоторый набор типовых классов, имеющих разные определенные характеристики. Эти типы образуют группы классов со схожими свойствами и поведением. Упомянутые типы представляют собой:

- M-классы, похожие на интерфейсы языка программирования Ява. Эти классы могут иметь только виртуальные методы, т.е. методы, которые обязательно должны будут реализованы в наследующих классах. M-классы всегда имеют в названии начальную букву «M», как например, в классе `MNotify`.

- T-классы являются «простыми», т.е. они не содержат в себе никаких «внешних» данных, расположенных на «куче». Как правило, для таких классов не нужен деструктор. T-классы, конечно же, могут хранить внутри указатели на внешние данные, однако они не должны отвечать за управление этими указателями (например, создавать, переопределять или удалять их). T-классы представляют собою простые типы, и их имя всегда начинается с буквы «T», как например, в классах `TRect` и `TPoint`.

- L-классы представляют собой новую идиому в Symbian OS, и являются самоуправляющимися строковыми шаблонными классами, поддерживающими автоматическую очистку памяти.

- R-классы имеют доступ к внешним ресурсам, например, к файлам. Название R-класса всегда начинается с буквы «R», как в классах `RFile` и `RSocket`.

Однако самым распространенным типом классов являются C-классы. C-классы могут содержать внутри данные, расположенные на «куче», и всегда наследуются от базового класса `CBase`. Любой класс, унаследованный от `CBase`, имеет в своем названии начальную букву «C», означающую слово «class» [1].

Как же нам выбрать подходящий тип класса? Тут может помочь одно простое правило: если вам нужен деструктор, значит, вам нужен C-класс. Если вы собираетесь делать что-то необычное, или просто нуждаетесь в классе, объекты которого будут располагаться на «куче», то вам так же потребуется C-класс.

2 Анатомия класса, унаследованного от CBase

Класс, унаследованный от `CBase`, имеет следующие свойства:

- Все данные внутри C-класса автоматически обнуляются при инициализации объекта этого класса. За обнуление данных отвечает перегруженный оператор `new`, переопределенный в классе `CBase`.

- Все объекты C-классов размещаются на «куче». Одной из причин для такого размещения является нужда в одинаковом механизме инициализации C-классов. Так как объекты в стеке не создаются при помощи оператора `new`, их данные могут быть и не обнулены при создании объектов. Если бы разрешалось размещать C-классы в стеке, мы могли бы оказаться в ситуации, когда одни из только что созданных C-классов обнулили свои данные, а другие – нет.

- В случае возникновения «сброса» (leave), C-классы должны самостоятельно очистить занимаемую ими память, т.к. они располагаются на «куче». Данное обстоятельство накладывает на нас обязательства сознательной и аккуратной очистки памяти при работе с C-классами (сбросы будут обсуждаться в главе 3.3).

- C-классы передаются через указатель или по ссылке, и поэтому для них не нужен явный конструктор копирования или оператор присваивания, если, конечно, такое копирование явно не понадобится какому-нибудь C-классу.

- Так как C-классы имеют необычное конструирование, и так как во время создания

объекта С-класса может возникнуть сброс, для защиты от утечек памяти в С-классах используется двухфазное конструирование объектов. Первой фазой такого конструирования является обычный конструктор Си++, код которого не должен вызывать сброс. А второй фазой конструирования объекта С-класса является вызов метода `ConstructL()`, содержащий в себе код, способный привести к сбросу.

Некоторые из приведенных пунктов будут более детально раскрыты в дальнейших главах.

Обычно С-классы не могут использоваться в качестве встраиваемых типов данных других С-классов (хотя некоторые исключения из этого правила еще будут обсуждаться в дальнейшем). Причиной такого ограничения является тот факт, что если С-класс будет использован в качестве встроенного члена другого класса, для его создания будет использован конструктор по умолчанию. Таким образом, двухфазное конструирование станет невозможным, и мы потеряем все выгоды этого приема. Для предотвращения такого неприятного сценария, все С-классы должны держать свои конструкторы приватными.

2.1 Что такое CBase?

Класс `CBase` является родителем всех остальных С-классов. Этот класс определен в заголовочном файле `e32base.h`, и имеет три основных свойства:

1. Он обладает виртуальным деструктором, который используется стеком очистки для стандартного освобождения памяти, занимаемой всеми объектами, унаследованными от класса `CBase`. Виртуальный деструктор базового класса `CBase` позволяет больше не объявлять деструкторы наследующих классов виртуальными, так как это уже сделано в базовом классе.

2. Класс `CBase` перегружает оператор `new` для обнуления данных С-классов при их инициализации. Удобства такого приема будут обсуждаться далее.

3. Класс `CBase` имеет приватный конструктор копирования и оператор присваивания. Подобный прием позволяет предостеречь пользователей С-классов от случайного создания нежелательных копий объектов. Теперь, для создания копий объектов С-класса понадобится явно объявлять и реализовывать конструктор копирования и оператор присваивания.

2.2 Стек очистки

Стек очистки является стандартным способом управления памятью в Symbian OS в случаях возникновения каких-либо ошибок или проблем. Стек очистки сохраняет у себя указатели на объекты, размещенные на «куче», и освобождает занимаемую ими память, в случае возникновения каких-либо внештатных ситуаций, предотвращая таким образом утечку памяти.

Класс `CleanupStack` представляет собой набор функций для добавления и удаления ресурсов из стека очистки. Класс `CBase` имеет встроенную поддержку работы со стеком очистки (о чем буде больше рассказано в главе 4). Класс `CleanupStack` определяет следующие методы:

- `Check()` – проверяет, находится ли ожидаемый элемент на самой вершине стека очистки, или нет.

- `Pop()` – извлекает самый верхний элемент из стека очистки. Метод так же может использовать в качестве аргумента переменную `TInt`, указывающую на количество элементов, которые нужно извлечь из стека за один прием. Так же есть вариант этого метода, который использует в качестве аргумента указатель на последний элемент, который нужно будет извлечь из стека вместе с другими элементами, количество которых будет так же указываться при помощи `TInt`-аргумента `aCount`. При этом из стека сначала будет извлечено элементов в количестве `aCount-1`, а затем будет произведена проверка корректности очередности при помощи переданного через аргументы указателя. Если последний элемент в стеке не будет равен элементу, переданному в качестве аргумента, то Symbian OS вызовет панику с кодом «`USER 90`».

- `PopAndDestroy()` извлекает элементы из стека, и освобождает занимаемую ими

память. Этот метод может использовать те же аргументы, что и метод Pop().

- PushL() – помещает указанный элемент в стек очистки.

Когда объекты C-классов, помещенные в стек очистки, извлекаются оттуда при помощи различных методов PopAndDestroy(), занимаемая ими память освобождается при помощи вызова их деструкторов.

2.3 Наследование класса CBase

2.3.1 Помните о главной причине наследования от CBase!

Что будет, если мы забудем произвести класс от CBase? Когда указатель на объект такого класса помещается в стек очистки, стек очистки будет интерпретировать его как указатель на объект класса TAny, а не как указатель на объект класса CBase. И когда стек очистки будет освобождать память, занимаемую таким объектом, он воспользуется методом User::Free(), в результате чего мы получим утечку памяти.

2.3.2 Вначале – CBase, потом – все остальное

Хорошей практикой является указание наследуемых классов в правильном порядке. Класс CBase, или любой другой класс, унаследованный от CBase, должен находиться первым в списке наследуемых классов:

```
class COurClass : public CBase, public MClass1
```

то же самое правило должно выполняться и для других классов, стоящих далее в иерархии наследования:

```
class CAnotherClass : public COurClass, public MClass2
```

Подобная практика не только улучшает иерархию наследования, но еще так же исключает проблемы при приведении наследующего класса к базовому. Если на первом месте в списке наследуемых классов окажется класс, унаследованный не от класса CBase, у вас могут возникнуть проблемы при помещении объектов такого класса в стек очистки. Все дело в том, что когда объект, унаследованный от CBase, помещается в стек очистки при помощи одного из методов CleanupStack::PushL(), в дело пускается неявное приведение типов, и указатель на объект вашего класса конвертируется в указатель на объект класса CBase. При этом указатель, полученный таким образом, адресует в памяти объект типа CBase, а не объект класса COurClass, к примеру.

Когда объект извлекается из стека очистки, для него вызывается метод CleanupStack::Pop(TAny* aExpectedItem). Данный метод сравнивает адрес аргумента aExpectedItem с адресом элемента, находящегося на самой вершине стека очистки. Если эти адреса совпадают, то объект извлекается из стека очистки.

Что может случиться, если класс CBase не указан первым в списке наследуемых классов? Давайте рассмотрим такую ситуацию шаг за шагом:

1. У нас имеется следующий класс:

   ```
   class CTestClass : public MCallback, public CBase
   ```

2. Как уже было сказано выше, когда указатель на объект класса CTestClass будет помещен в стек очистки, для него будет вызван метод PushL(CBase*). Для того, чтобы указатель, передаваемый в стек очистки, имел адрес объекта класса CBase, а не адрес объекта M-класса, его значение будет автоматически увеличено на 4 байта (см. рис. 1):

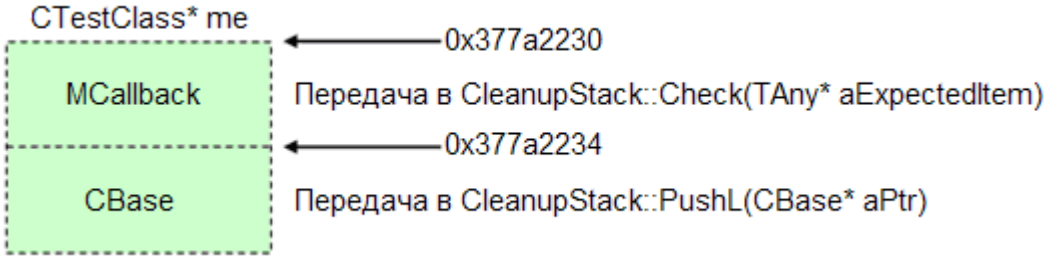

Рис. 1: Размещение в памяти объекта класса *CTestClass* [2].

3. К сожалению, стек очистки не имеет соответствующего метода `Pop(CBase*)`. Метод, при помощи которого указатели удаляется из стека, объявлен как `Pop(TAny*)`. Наш М-класс может быть без проблем интерпретирован как `TAny`, и поэтому метод `Pop(TAny*)` с удовольствием принимает переданный ему параметр без необходимого нам увеличения значения указателя. В действительности же, метод `Pop(TAny*)` сверяет адреса переданного ему в качестве аргумента указателя и указателя, находящегося на самой вершине стека очистки. Данная проверка выполняется при помощи метода `Check()`, и если адреса указателей окажутся неравными (в нашем случае разница будет равна 4 байтам), Symbian OS объявит панику с кодом «`E32USER-CBase 90`», и остановит ваше приложение. (Обратитесь к источнику [2] за дополнительной информацией.)

2.3.3 Сравнивая CBase с другими…

Достаточно интересно сравнить класс `CBase` и его место в иерархии классов с решениями в других языках программирования и системах. Например, все классы языка Java неявным образом берут свое начало от класса `Object`, и при объявлении Java-класса этот факт даже не нужно описывать в коде. С другой стороны, стандарт ISO C++ не определяет какой-либо базовый класс, поэтому разработчики имеют полную свободу в определении своих базовых классов. Однако библиотека Си++ классов от Microsoft Foundation имеет класс `CObject`, являющийся базовым для всех остальных классов, и определяет следующие базовые параметры классов [3]:

- Поддержка сериализации объекта
- Информация о классе во время выполнения программы
- Получение диагностической информации об объекте
- Совместимость с классами массивов и коллекций

Таким образом, здесь мы с вами можем видеть схему, схожую с решением на основе `CBase`. В том смысле схожую, что класс `CBase` так же предоставляет разработчикам определенные характеристики, полезные при разработке кода и решении различных задач. (Основные полезные характеристики класса `CBase` будут детально рассмотрены в главе 4.)

2.4 Виртуальный деструктор

Виртуальный деструктор, унаследованный нашим классом `COurClass` от `CBase`, позволяет создавать `CBase`-указатели на объекты класса `COurClass`. И при уничтожении такого `CBase`-указателя будет вызываться деструктор класса `COurClass`:

```
class COurClass : public CBase
    {
    // реализация нашего класса
    }

void SomeOtherObject::SomeFunction()
    {
    CBase* anObject = new (ELeave) COurClass();
    delete anObject;
    }
```

В вышеприведенном примере указатель на объект класса `COurClass`, унаследованного от `CBase`, сохраняется в переменной указателя типа `CBase`. Когда будет вызываться оператор `delete`, код автоматически вызовет деструктор класса `COurClass`.

Обратите внимание на то, что в Symbian OS используется перегруженный оператор new, или new (ELeave). Этот оператор вызывает сброс, если для нового объекта не хватает места в памяти устройства.

3 Основные шаблоны использования C-классов

3.1 Очистка памяти

Деструктор C-класса должен освобождать все ресурсы, занимаемые объектом. Конечно, есть вероятность, что очистка памяти будет производиться частично инициализированным объектом, однако и в этом случае не должно возникнуть никаких проблем, так как классы, наследуемые от CBase, обнуляют свои члены, если при их создании был использован оператор new. А удаление NULL-объектов в Symbian OS никогда не приводит к ошибкам.

Ресурсы класса обычно определяются при помощи указателей или дескрипторов. Очень важно, чтобы эти указатели и дескрипторы были обнулены, если они больше не указывают и не содержат в себе никаких данных. Однако данное условие автоматически выполняется в классах, унаследованных от CBase. Следует, правда, обратить внимание на то, что обнуление происходит только в момент создания объекта. В дальнейшем программисты должны сами позаботиться об очистке и обнулении указателей, если эти указатели должны будут использоваться заново в течении жизни объекта. Когда объекты, на которые ссылаются такие указатели, будут успешно удалены, указатели так же должны быть обнулены перед своим новым использованием:

```
void CMyClass::SomeFunctionL()
    {
    delete iMemberData;                  // 1
    iMemberData = NULL;                  // 2
    iMemberData = CMyOtherClass::NewL(); // 3
    }
```

После вызова оператора delete, переменная указателя iMemberData все еще указывает на область памяти, которую раньше занимал удаленный объект. Если перед новым использованием указателя ему не присвоить значение NULL (строка 2), дальнейшие попытки вызвать оператор delete для iMemberData, приведут к двойному удалению указателя, что, в свою очередь, вызовет панику системы. Если, опять же, вызов CMyOtherClass::NewL() в третьей строке приведет к сбросу, то системой тут же будет вызван деструктор класса CMyClass, который, в свою очередь, постарается удалить уже удаленный указатель iMemberData, что опять приведет к двойному удалению указателя. Поэтому после того как вы удалили указатель, всегда присваивайте ему значение NULL.

В деструкторах уже нет надобности присваивать удаленным указателям значение NULL, так как сам объект скоро будет удален, и поэтому указатель никто не сможет заново использовать. Например:

```
CMyClass::~CMyClass()
    {
    delete iMyDataMember;
    // нет больше надобности присваивать указателю NULL!
    }
```

Будьте осторожны при удалении объектов, взаимодействующих между собой. Очень важно удалять такие объекты в правильном порядке, иначе мы можем столкнуться с ситуацией, когда один объект будет зависеть от другого, уже несуществующего в памяти устройства!

3.2 Конструирование

C-классы в Symbian OS обычно конструируются в два шага:

1. Вначале создается объект при помощи вызова оператора new (ELeave) CMyTestClass(). Этот оператор, в свою очередь, вызывает стандартный

конструктор класса, CMyTestClass().
2. Затем объект инициализируется при помощи метода ConstructL().

В конце-концов, когда работа с объектом закончена, мы удаляем его при помощи оператора delete.

Правда каждый раз ожидать от разработчиков сначала вызова оператора new (ELeave), а затем – метода ConstructL(), когда они хотят создать какой-нибудь объект в программе, – это уже слишком. Если бы объекты всегда создавались таким образом, то работа программиста превратилась бы в тяжелую рутину (хотя, для справедливости стоит сказать, что иногда такой способ создания объектов является наиболее эффективным, так как в данном случае нам не приходится помещать объекты в стек очистки перед вызовов метода ConstructL()). Для автоматизации этого рутинного процесса, в Symbian OS используется весьма удобный способ создания объектов путем вызова фабричных методов С-класса:

```
// объявление класса
class CMyTestClass : public CBase
    {
public:
    static CMyTestClass* NewL(); // фабричный метод
    static CMyTestClass* NewLC(); // фабричный метод
    ~CTest();
private:
    CMyTestClass();
    void ConstructL();

    TInt iSomeInt;
    RFile iMyMemberFile;
    CAClass* iMyMemberCClass;
    };
```

Обратите внимание на то, что конструктор первой фазы CMyTestClass(), и конструктор второй фазы ConstructL(), объявлены в классе приватными. Таким образом пользователи класса будут вынуждены использовать для создания объектов только фабричные методы NewL() и NewLC() (и еще реже – New()):

```
// реализация класса
CMyTestClass* CMyTestClass::NewL()
    {
    CMyTestClass* self = NewLC(); // создаем и помещаем в стек очистки
    CleanupStack::Pop(self); // извлекаем из стека очистки
    return self;
    }

CMyTestClass* CMyTestClass::NewLC()
    {
    CMyTestClass* self = new (ELeave) CMyTestClass(); // создаем и обнуляем
    CleanupStack::PushL(self); // помещаем в стек очистки
    self->ConstructL(); // инициализируем созданный объект
    return self;
    }
```

Обратите внимание на различия между методами NewL() и NewLC(): NewLC() помещает указатель на новый объект в стек очистки, и оставляет его там (по терминологии Symbian OS, буква «C» в названии метода означает то, что метод помещает некие объекты в стек очистки, и оставляет их там), в то время как метод NewL() помещает указатель в стек, но затем извлекает его оттуда. Когда вам нужно инициализировать члены класса, используйте метод NewL(), иначе объект будет удален дважды. Если метод NewLC() будет применен к данным самого класса, то и стек очистки и сам объект класса получат указатель на данные. И оба попытаются удалить его в свое время: однако один из них это сделает успешно, а другой попытается освободить память, которая уже до него была освобождена, что, естественно, приведет к панике.

```
CMyTestClass::CMyTestClass() : iSomeInt(100)
```

```
    {
    // здесь только код, не вызывающий сбросов
    }


CMyTestClass::~CMyTestClass()
    {
    /* здесь очистка памяти: удаление объектов C-классов и освобождение ресурсов R-
    классов */
    delete iMyMemberCClass;
    iMyMemberRFile.Close();
    }

void CMyTestClass::ConstructL()
    {
    // код, способный вызвать сброс, следует помещать только сюда
    }
```

Статические методы `New()` следует использовать следующим образом:

```
CMyTestClass* myObject = CMyTestClass::NewL();
```

или

```
CMyTestClass* myObject = CMyTestClass::NewLC();
```

Обратите внимание еще раз на то, что методы `NewL()` и `NewLC()` были объявлены статическими. Это позволяет вызывать их еще до того, как объект создан.

Причина, по которой мы используем двухфазное конструирование, заключается в нашем желании получать полностью инициализированные и готовые к использованию объекты. Если бы обычный Си++ конструктор содержал код, способный вызывать сбросы, то любой сброс смог бы легко нарушить процесс инициализации объекта. Это, в свою очередь, привело бы к тому, что для такого частично инициализированного объекта в свое время не был бы вызван деструктор, и мы получили бы обычную утечку памяти. Поэтому наиболее безопасным способом создания объектов является двухфазное конструирование.

3.3 Сбросы

Исторически сложилось так, что в Symbian OS не использовались механизмы Си++ для обработки исключений и непредвиденных ситуаций. Все дело в том, что Си++ исключения еще не были стандартизированы на момент создания Symbian OS. (Когда Си++ исключения были стандартизированы, они все равно требовали слишком много ресурсов в виде больших запросов к количеству используемой оперативной памяти и размерам скомпилированного кода, поэтому так и не были приняты на вооружение в Symbian OS. [4]) С 9-й версии Symbian OS, для обработки всех внештатных ситуаций используется уже встроенный механизм обработки Си++ исключений, и разработчики могут пользоваться им, если захотят. Однако стандартным способом обработки ошибок в Symbian OS, по прежнему является работа со «сбросами».

Сброс генерируется во время появления внештатного события в системе, как, например, при возникновении недостатка оперативной памяти. Любая функция, способная привести к возникновению внештатной ситуации, и следовательно, к появлению сброса, должна иметь в своем названии окончание «`L`». Любая функция, вызывающая такую `L`-функцию, сама становится способна вызвать сброс, и поэтому так же должна иметь в своем названии окончание «`L`» (если, конечно, она не отлавливает и не обрабатывает внутри себя все возможные сбросы). Данный способ наименования функций является жизненно важным: любой разработчик, вызывающий такие функции, будет знать о возможных сбросах, и таким образом сможет предотвратить возможные утечки памяти.

Сбросы должны ловиться и обрабатываться при помощи «ловушек» – макросов `TRAPD` и `TRAP`:

```
// данная функция обрабатывает сбросы внутри себя
ThisFunctionDoesNotLeave()
    {
    TRAPD(result, ThisFunctionMayLeaveL());
```

```
    // обработать любую ошибку, возникшую в методе ThisFunctionMayLeave()
    }
```

Макрос TRAPD объявляет переменную, в которой будет сохранен код возникшей ошибки. При использовании макроса TRAP, вы должны будете сами объявить переменную для хранения возможного кода ошибки:

```
// данная функция обрабатывает сбросы внутри себя
ThisFunctionDoesNotLeave()
    {
    TInt result; // переменная для хранения кода ошибки
    TRAP(result, ThisFunctionMayLeaveL());
    // обработать любую ошибку, возникшую в методе ThisFunctionMayLeave()
    }
```

Любая программа должна использовать в своем теле хотя бы один-единственный вызов макроса TRAP для поимки и обработки всех возможных сбросов. Такой единичный вызов макроса TRAP может находиться на самом верхнем уровне программы, и обрабатывать все возможные сбросы. Так же можно размещать вызовы TRAP-макросов в отдельных местах кода, и обрабатывать таким образом более специфические проблемы и конкретные ошибки.

До адаптации в Symbian OS Си++ исключений для обработки сбросов, использование TRAP-макросов было весьма прожорливым в отношении циклов CPU и используемой памяти, поэтому разработчикам советовалось сводить использование TRAP-макросов к минимуму. К счастью, с появлением Си++ исключений, TRAP-макросы стали намного эффективнее.

Еще одной особенностью использования TRAP-макросов является невозможность помещения объектов в стек очистки при их вызове. Например, когда вы хотите вызвать LC-метод, помещающий объекты в стек очистки, и способный привести к сбросу:

```
TRAPD(err, myVar = CSomething::NewLC()); // вызовет панику
```

Причиной такого ограничения является особенность, с которой TRAP-макросы работают со стеком очистки. Вначале TRAP-макрос (TRAPD в данном случае) метит стек очистки перед вызовом потенциально опасной функции. Затем вызывается функция, и если возникает сброс, то свою полезную работу начинает TRAP-макрос. А он предполагает, что все объекты, помещенные в стек очистки во время работы потенциально опасной функции, будут удалены из него. Так как вызов метода CSomething::NewLC() поместит объект класса в стек очистки, и не удалит его оттуда, Symbian OS вызовет панику с кодом «E32User Panic 71».

4 Кодирование без C-классов

C-классы, при помощи своего базового класса, CBase, предоставляют нам множество удобств. Для того чтобы в этом полностью убедиться, давайте рассмотрим случай, когда мы хотим обойтись без использования класса CBase. Для примера мы создадим на «куче» объект, и удалим его. Возможно, пример кажется слишком простым, но, разбив его на части, мы получим:

1. Создать объект на куче.

2. Проверить, если полученный объект равен NULL.

3. Обнулить переменные объекта – нам приходится это делать самим, т.к. у нас нет класса CBase, выполняющего данную работу за нас.

4. Написать код для удаления указателя объекта из стека очистки, чтобы любая ситуация возникновения сброса была грамотно обработана. А так же создать объект типа TCleanupItem для размещения объекта в стеке очистки.

5. Поместить указатель объекта в стек очистки.

6. Использовать объект, как нам будет необходимо.

7. По завершении работы с объектом, удалить из стека очистки его указатель, и

освободить занимаемую объектом память.

В коде приведенная последовательность будет выглядеть следующим образом:

```
// использование класса, не унаследованного от CBase
void UseNonCBaseClass()
    {
    SomeObject* myObject = new SomeObject();          // шаг 1
    if (myObject)                                     // шаг 2
        {
        Mem::FillZ(myObject, sizeof(myObject));       // шаг 3
        TCleanupItem item(FreeAnObject, myObject);    // шаг 4
        CleanupStack::PushL(item);                    // шаг 5
        /* использовать объект */                     // шаг 6
        CleanupStack::PopAndDestroy(myObject);        // шаг 7
        }
    else
        {
        /* обработать проблемы, возникшие при инициализации объекта */
        }
    }

// метод для освобождения указателя из стека очистки
void FreeAnObject(TAny* aObject)
    {
    SomeObject* object = static_cast<SomeObject*>(aObject);
    delete object;
    }
```

Как вы можете заметить, отказ от использования класса `CBase` делает жизнь разработчика намного сложнее. Если бы вы использовали класс `CBase` в качестве базового, некоторые из вышеперечисленных шагов были бы автоматизированы. Например:

- Шаг 3 был бы реализован перегруженным оператором `new` класса `CBase`.
- Шаг 4 был бы полностью автоматизирован благодаря встроенной поддержке стека очистки классом `CBase`. Иначе нам приходится создавать объект типа `TCleanupItem` с указателем на функцию очистки и сам объект. Когда наш объект `myObject` будет удален из стека очистки, для освобождения занимаемой им памяти будет вызвана функция `FreeAnObject()`.

Фактически, если бы мы унаследовали наш класс от `CBase`, и дополнительно воспользовались двухфазным конструированием, нам не пришлось бы заботиться о шагах со второго по пятый. Конечно, во втором шаге мы могли бы воспользоваться вызовом оператора `new (ELeave)`, и решить, таким образом, множество проблем безопасной инициализации памяти. Однако смысл заключается в том, что правильно реализованное двухфазное конструирование отбрасывает необходимость использования такого, и ему подобных приемов при создании объектов С-классов.

5 Исключения в основных шаблонах использования

Хотя правила определения и использования С-классов подходят для большинства случаев, все же в этих правилах имеются некоторые исключения. Например, некоторые С-классы могут быть использованы как встроенные члены данных (inline data members) в теле других классов, унаследованных от `CBase`. Одним из таких классов является `CDesCArrayFlat`.

Главной причиной, по которой объекту класса `CDesCArrayFlat` не нужно быть указателем, является отсутствие в нем конструктора второй фазы. Инициализация объекта выполняется в тот момент, когда вы добавляете в него новые элементы. Для возможности размещения такого объекта в стеке, пользователь сначала должен обнулить размер массива сразу же после его объявления, так как при попытке вложить в массив новые элементы, объект будет проверять наличие имеющейся в его распоряжении памяти, обращаясь к `CArrayFixBase::iBase`. Если размер массива при создании не был обнулен, то при любых попытках его использования, будет возникать паника с кодом «KERN-EXEC 3».

Класс CActiveSchedulerWait, как и CDesCArrayFlat, может быть использован в C-классе без наличия указателя на него. Опять же, по причине отсутствия у этого класса конструктора второй фазы. И если объект этого класса будет находиться внутри C-класса, то он будет автоматически обнулен. Если вы захотите разместить его в стеке, то вам сначала придется обнулить его при помощи метода Mem::FillZ().

6 Некоторые общие C-классы

6.1 Семья классов CArray

Семья классов CArray включает в себя классы для массивов постоянной и переменной длины, хранящей как объекты, так и указатели на них, содержащиеся как в последовательных, так и в сегментированных областях памяти. Элементы массива могут быть удалены либо по одиночке:

```
myVariableCArray.Delete(aIndex);
```

либо все сразу:

```
myVariableCArray.Reset();
```

Вместе с удалением всех элементов, метод Reset() так же освободит всю память, отведенную для массива.

В данной семье есть классы, хранящие объекты как по значению, так и через указатели. Последний тип массивов берет свое начало от класса CArrayPtr. Когда объекты в массиве сохраняются в виде указателей, вам следует быть очень осторожными с управлением памятью, и точно знать, является ли данный массив владельцем объектов через хранимые в нем указатели, или же просто имеет через указатели удобный доступ к объектам. Если массив является владельцем объектов, тогда будет весьма полезна функция:

```
myVariableCArray.ResetAndDestroy();
```

которая сначала применит для каждого элемента в массиве оператор delete, а потом вызовет функцию Reset().

6.2 Графические классы: устройства, контексты и битмапы

Для графических устройств базовым классом является CGraphicsDevice. Данный класс наследуется от двух других классов, первым из которых является CBase:

```
class CGraphicsDevice : public CBase, public MGraphicsDeviceMap
```

От класса CGraphicsDevice производятся три других основных класса графических устройств: CFbsBitmapDevice, CFbsScreenDevice и CWsScreenDevice. Первый из них используется для прямой работы с битмапами, в то время как два остальных используются как для прямой работы с экраном, – в случае с CFbsScreenDevice, – так и для работы с экраном через сервер окон (Windows Server) в Symbian OS.

Понятие битмапа определено в классе CFbsBitmap. Данный класс представляет собой блок памяти, в котором могут храниться пиксельные данные, и который может быть одновременно доступен разным процессам. Класс содержит переменные размера экрана (длину и ширину), а так же переменную глубины цвета, определяющую формат данных для всех пикселей в битмапе.

Понятие «контекста» используется при отображении графических данных на экране или при рисовании пикселей на битмапе. Для контекстов существует множество базовых классов, включающих такие классы как CBitmapContext и CGraphicsContext. Главными контекстными классами являются CFbsBitGc и CWindowGc. Первый из них может отображать графические данные прямо на битмапе или на экране, в то время как последний может использоваться для отображения графических данных в окнах программ. Данные классы предоставляют своим

пользователям независимый от устройства абстрактный графический интерфейс, а так же предоставляют доступ ко всем необходимым настройкам, использующимся при рисовании и отображении графических данных на различных устройствах. Настройки включают в себя:

- Цвет, стиль и размер «кисти» или «ручки», используемой при рисовании
- Выравнивание текста
- Режим рисования, как, например, инвертирование используемого цвета, и т.д.

Объекты контекстов создаются не как большинство С-классов, а с использованием фабричных функций, доступных из соответствующих графических устройств:

```
TInt CreateContext(CGraphicsContext*& aGc);
```

Однако объект контекста, возвращаемый такой функцией, переходит в собственность класса, вызвавшего эту функцию, и поэтому по завершении работы с таким объектом, класс должен будет освободить память, занимаемую объектом контекста при помощи оператора `delete`.

6.3 CRichText

Класс `CRichText` позволяет создавать текст в расширенном текстовом формате. В таком тексте каждый параграф и каждый символ могут иметь свой собственный формат. Форматирование текста происходит при помощи как глобальных слоев форматирования (format layers), так и при помощи специального форматирования, применяемого к отдельным элементам текста с использованием методов `ApplyCharFormat()` и `ApplyParaFormat()`. В случае возникновения конфликтов между слоями форматирования, предпочтение отдается специальному форматированию, которое будет использовано поверх глобального. Текст с расширенным форматированием так же поддерживает встраиваемые объекты. Встраиваемые объекты будут представляться в тексте при помощи классов, унаследованных от класса `CPicture`.

Класс `CRichText` демонстрирует принципы хорошего объектно-ориентированного дизайна. Его широкая функциональность заложена в его базовых классах. Например, его базовый класс `CGlobalText` предоставляет пользователям возможность работы с глобальными слоями форматирования. В свою очередь, сам `CGlobalText` берет свое начало от класса `CPlainText`, предоставляющий своим пользователям возможности хранения текста.

6.4 CServer2

Класс `CServer2` представляет собой абстрактный базовый класс для создания Symbian-серверов. Сам по себе он является активным объектом (см. главу 6.7), и поэтому при своем создании требует определения приоритета исполнения. Класс `CServer2` принимает запросы от своих клиентов и перенаправляет их в соответствующую клиентскую сессию на стороне сервера. Этот класс отвечает за процесс создания клиентских сессий на стороне сервера, как результат ответа на клиентские запросы. Обычная процедура обработки запросов клиентов выглядит следующим образом:

- Вначале сервером создается объект класса `CServer2`.
- В качестве результата на запрос клиента по установке соединения создается новая сессия. Что, в свою очередь, приводит к вызову метода `CServer2::NewSessionL()`.
- Вызов метода `CServer2::NewSessionL()` создает объект, унаследованный от класса `CSession2`.

6.5 CSession2

Класс `CSession2` представляет собой сессию соединения между клиентом и сервером Symbian OS. Сессия работает в качестве канала коммуникаций между клиентом и сервером, и один клиент может иметь несколько параллельных сессий с сервером, что, впрочем, является большой редкостью. Чаще всего разные клиенты имеют сессии с сервером в одно и то же время. Класс `CSession2` имеет следующие методы:

- `ServiceL()` – используется для обработки любых сообщений, за исключением запросов на подключение и отключение, которые обрабатываются функциями `CreateL()` и `Disconnect()`, соответственно. Простая реализация этих функций в

классе – все, что необходимо сделать разработчику для получения полноценного класса сессии. Метод `ServiceL()` использует объект класса `RMessage2` в качестве параметра, который так же должен быть определен разработчиком сервера.
- `CountResources()` – считает количество ресурсов, использующихся в данный момент.
- `ServiceError()` – обрабатывает ситуации возникновения каких-либо ошибок, если метод `ServiceL()` вызывает сброс.

6.6 CDictionaryStore

Абстрактный класс `CDictionaryStore` представляет собой интерфейс хранилища, который доступен через свой идентификатор UID, а не прямо через ID потока. Интерфейс включает в себя следующие методы:

- `IsPresentL()` – принимает в качестве аргумента UID, и проверяет, имеется ли в хранилище поток с указанным UID.
- `RevertL()` – возвращает словарь к состоянию, имевшемуся на момент последнего изменения его содержимого, и вызывает сброс, если такая операция оказалась неуспешной.
- `Commit()` и `CommitL()` – фиксируют изменения в словаре. Метод `CommitL()` вызывает сброс, а метод `CommitL()` возвращает код ошибки, если операция фиксирования содержимого словаря оказалась неуспешной.

6.7 CActive

Класс `CActive` является абстрактным базовым классом, заключающим в себе функциональность передачи запроса какому-либо асинхронному сервису, и обработки результатов, выполненных этим сервисом запросов. Мобильное приложение может иметь несколько активных объектов (active objects), чья деятельность будет контролироваться активным планировщиком задач (active scheduler). Для того чтобы класс мог стать активным объектом, ему нужно быть унаследованным от класса `CActive`, и при этом указать приоритет выполнения в качестве параметра конструктора по умолчанию:

```
CMyActiveObject::CMyActiveObject() : CActive(CActive::EPriorityStandard)
    {
    }
```

Если в течении жизни объекта вам понадобится изменить приоритет его выполнения, вы всегд сможете воспользоваться методом `SetPriority()`.

Активный объект может быть зарегистрирован планировщиком задач в конструкторе второй фазы – `ConstructL()` (о котором мы говорили в главе 3.2):

```
void CMyActiveObject::ConstructL()
    {
    // зарегистрировать наш активный объект планировщиком задач
    CActiveScheduler::Add(this);
    }
```

или даже в конструкторах первой фазы, так как вызов метода `CActiveScheduler::Add()` не может привести к сбросу.

Для реализации функциональности активного объекта, в классе должны быть реализованы два метода: `RunL()`, который будет вызван планировщиком по завершении асинхронного запроса от активного объекта; и `DoCancel()`, который вызывается методом `CActive::Cancel()`, когда владелец активного объекта больше не заинтересован ни в каких асинхронных сервисах планировщика.

6.8 CEikAppUi

Классы, унаследованные от `CEikAppUi`, являются частью фреймворка приложений Symbian OS. Данные классы обрабатывают события и действия, производимые пользователями мобильных приложений, такие, как например, обработка нажатия клавиш, обработка действий пользователя с меню приложений, управление выпадающими меню, открытие и закрытие

файлов, а так же завершение работы приложений в Symbian OS. Команды, активизированные пользователем через меню приложений, обрабатываются в методе `HandleCommandL()` этого класса. В классах, унаследованных от `CEikAppUi`, этот метод переопределяется согласно необходимости и специфике работы класса. Вдобавок класс `CEikAppUi` наследует от класса `CCoeAppUi` определенное число методов, используемых для обработки различных событий:

- `HandleKeyEventL()` и `HandleSystemEventL()` – для обработки системных событий и событий от клавиатуры.
- `HandleSwitchOnEventL()` – для обработки события включения устройства.
- `HandleForegroundEventL()` – для обработки события вывода мобильного приложения в активное состояние.

6.9 CCoeEnv

Класс `CCoeEnv` предоставляет разработчику функции создания элементов управления графического интерфейса, таких, как например, списки (list box). Данный класс является активным объектом, и содержит в себе весь необходимый код работы с активным планировщиком. Класс `CCoeEnv` предоставляет множество вспомогательных функций, которые будут полезны многим приложениям, например:

- `ReadResource()` – считывает ресурс в дескриптор.
- `SystemGc()` – возвращает системный графический контекст.
- `WsSession()` – возвращает сессию с сервером окон, принадлежащую данному приложению.
- `FsSession()` – возвращает сессию с файловым сервером, владельцем которой является класс `CCoeEnv`.
- `Format256()` – считывает 256-байтовый ресурс в отформатированную строку.

Для более подробной информации обо всех вышеперечисленных и других классах, обратитесь к последней версии SDK.

7 Наилучшая практика работы с C-классами

Как вы уже, наверное, знаете, существуют определенные рекомендации касательно использования C-классов:

- Используйте двухфазное конструирование – это позволит вам безопасно создавать объекты C-классов.
- В C-классе должен быть реализован деструктор, для освобождения всех ресурсов, владельцем которых является объект. Таким образом, вы избежите возможных утечек памяти.
- Вообще, C-классы не могут быть использованы в качестве встраиваемых типов данных других C-классов. Причиной этому служит тот факт, что для встраиваемых типов данных используется конструктор по умолчанию, в то время как C-классы необходимо создавать при помощи двухфазного конструирования.
- C-классы должны наследоваться от `CBase` для того, чтобы избежать возможных проблем со стеком очистки. Следует помнить, что наследуемый класс `CBase` должен стоять первым в списке наследуемых классов.

8 Заключение

Данная статья является лишь беглым обзором C-классов, некоторых правил их использования, а так же некоторых исключений из этих правил.

C-классы являются основой устойчивости приложений Symbian OS. Совместно с R-, M-, L- и T-классами, они позволяют разделить функциональность приложения на отдельные «кусочки» – простые типы данных, доступ к ресурсам, интерфейсы, и всё остальное, что может быть обработано одним из этих типов классов. Эти классы оказывают Symbian-разработчику значительную помощь при создании кода, и вместе с правильно названными функциями (оканчивающихся на `L` и `LC`), позволяют легко разобраться в коде других разработчиков.

C-классы являются наиболее распространенным типом классов, с которыми вы будете встречаться при работе с Symbian OS. Что ж, тогда желаем вам удачи, помните об этом

руководстве, и вы не ошибетесь!

9 Дополнительные материалы

Помощь Symbian OS SDK версии 9.4 – полный ресурс помощи для Symbian-разработчика:

- developer.symbian.com/main/documentation/sdl/symbian94/sdk/doc_source/index.html

Symbian Developer Network – скорая помощь для двухфазного конструирования:

- developer.symbian.com/main/support/code_clinic/clinic_june2008/index.jsp

Forum Nokia – обсуждение двухфазного конструирования:

- discussion.forum.nokia.com/forum/showthread.php?t=132775&highlight=construction

NewLC – шесть главных вопросов касательно C-классов:

- www.newlc.com/inside-cbase-class-six-essential-questions

Концепции Symbian C++ – классы C, T, M и R:

- pf128.krakow.sdi.tpnet.pl/symbdev/tut/tut5.php

NewLC – обзор различных типов классов в Symbian OS:

- www.newlc.com/T-C-R-M-classes.html

Symbian Developer Library – описание C-, T-, M- и R-классов:

- developer.symbian.com/main/documentation/sdl/symbian94/sdk/doc_source/guide/EssentialIdioms/ClassTypes.guide.html

Forum Nokia wiki – обработка исключений в Symbian OS:

- wiki.forum.nokia.com/index.php/Exception_handling_in_Symbian_OS

Книга «*Symbian OS Explained: Effective C++ Programming for Smartphones*». Автор – Jo Stichbury, опубликовано издательством «John Wiley & Sons, Ltd.» в 2004-м году:

- developer.symbian.com/main/documentation/books/books_files/sose/index.jsp

SDN – «Знакомство с R-классами»:

- developer.symbian.com/main/downloads/papers/R_Classes_v1.09.pdf

SDN – «Знакомство с M-классами»:

- developer.symbian.com/main/downloads/papers/M_Classes.pdf

SDN – «Знакомство с T-классами»:

- developer.symbian.com/main/downloads/papers/TClasses.pdf

SDN – «Знакомство с L -классами»:

- developer.symbian.com/main/downloads/papers/LClasses.pdf

SDN – «Обработка исключений в Symbian OS»:

- developer.symbian.com/main/downloads/papers/Exception_Handling_in_Symbian_OS-v1.02.pdf

SDN – «Сбросы и исключения»:

- developer.symbian.com/main/downloads/papers/Leaves and Exceptions.pdfv1.02.pdf

10 Ссылки

[1] Stichbury, Jo (2004), «*Symbian OS Explained: Effective C++ Programming for Smartphones*», страница 4. Издательство «John Wiley & Sons, Ltd»:

- developer.symbian.com/main/documentation/books/books_files/sose/index.jsp

[2] Symbian Developer Network – детальное обсуждение того, почему класс `CBase` должен стоять первым в списке наследуемых классов:

- developer.symbian.com/main/support/code_clinic/clinic_may2008/index.jsp

[3] Microsoft MSDN Visual C++ Developer Center – страница описания класса `CObject`:

- msdn.microsoft.com/en-us/library/7k3448yy(VS.80).aspx

[4] Stichbury, Jo (2004), «*Symbian OS Explained: Effective C++ Programming for Smartphones*», страница 13. Издательство «John Wiley & Sons, Ltd»:

- developer.symbian.com/main/documentation/books/books_files/sose/index.jsp

11 О компании

Компания Penrillian создает мобильные приложения. Мы являемся экспертами в разработке и портировании приложений на все основные мобильные платформы, быстро и эффективно превращая отличные идеи в реальные продукты. Наша разнообразная база клиентов включает в себя глобальных поставщиков услуг и программных компаний, таких как Handmark, Sybase, T-Mobile и Vodafone. Они выбрали нас из-за нашего желания и тяги всегда соответствовать изменяющимся требованиям рынка, и всегда предоставлять наши решения вовремя и согласно бюджету.

Наша ключевая компетентность заключается в разработках для Symbian OS, S60, UIQ, Java, Windows Mobile, приложений безопасности, связи, RFID, а так же в создании мобильных пользовательских интерфейсов. Мы имеем тесные связи с главными игроками и центрами знаний в индустрии мобильных приложений, и мы рады быть платиновым партнером Symbian, партнером UIQ Alliance, а так же членом Forum Nokia PRO.

За более подробной информацией о наших консультационных услугах, портировании программ, и создания приложений, обращайтесь к нашему веб-сайту: www.penrillian.com.