Рецепты по использованию дескрипторов в Symbian OS

Алексей Гусев

Перевод с английского: Александр Смирнов

Рецензенты: Александр Труфанов, Денис Григоренко

Опубликовано Symbian Developer Network

Версия: 1.0 – Сентябрь 2008

1 О статье

Данная статья иллюстрирует использование одного из самых сложных понятий в Symbian OS – дескрипторов. Дескрипторы – это набор классов, предназначенных для работы со строками. Использование этих классов зачастую вызывает затруднения у новичков, которые только приступили к изучению программирования под Symbian OS. Возможно, даже более опытным инженерам придется потратить некоторое время для решения обычных задач при работе с классами дескрипторов.

Данная статья содержит подробное руководство, которое поможет вам научиться эффективно использовать дескрипторы, понять как они работают, а так же найти решения наиболее распространенных проблем, с которыми разработчикам приходится сталкиваться ежедневно. Статья составлена в виде рецептов, в том же стиле, что и недавно изданная книга «Quick Recipes on Symbian OS» издательства Symbian Press (дополнительную информацию вы можете найти на сайте developer.symbian.com/quick).

Рецептурная структура статьи позволяет отдельно знакомиться с любой ее частью, и дает читателю возможность быстро найти необходимое решение.

Все фрагменты кода в рецептах освещают определенные области API дескрипторов: от простых манипуляций строками, до конвертации между различными типами дескрипторов и лексического анализа.

Предполагается, что читатель имеет некоторые базовые знания C++, а также знаком с иерархией классов дескрипторов. Информацию о ней вы сможете найти в документации Symbian Development Library (по адресу developer.symbian.com/main/documentation/sdl) и в публикациях издательства Symbian Press (информацию о которых вы можете найти по адресу developer.symbian.com/books).

2 Вступление

Если использование дескрипторов дескрипторов Symbian OS вызывает у вас затруднение, то с помощью данной статьи мы надеемся приободрить вас и изменить ваше мнение о них к лучшему. Однако прежде чем мы начнем разбираться с нижеприведенными рецептами, позвольте нам напомнить пару вещей, которые необходимо иметь в виду при чтении данной статьи.

Главное, о чем нужно помнить, это то, что хотя дескрипторы в Symbian OS идентичны классам для работы со строками в других операционных системах или платформах, управление выделяемой для их хранения памятью целиком ложится на плечи разработчика. Этот факт сильно влияет на стиль программирования, особенно, если в программе возможно возникновение «сброса», «паники», и т.д.

Для простоты и понятности, примеры в данной статье не включают в себя код обработки ошибок. Однако в прилагающимся к ней проекте с примерами используются стандартные приемы Symbian C++ для обработки сбросов. Ознакомьтесь с ним, чтобы понять как нужно работать с дескрипторами на практике.

В большинстве рецептов используются одни и те же объявления, которые во избежание нежелательных повторений описаны только в первом примере кода.

После всего вышесказанного мы, наконец, готовы начать наше путешествие и исследовать мир дескрипторов Symbian OS.

3 Простые рецепты

3.1 Как объединить две строки

Проблема: у вас есть две строки, которые вы хотели бы объединить.

Решение: если вы раньше программировали на каком-либо объектно-ориентированном языке, то вы должно быть ожидаете решение, столь же простое, как использование `operator+()`.К счастью для нас, в Symbian OS действительно есть такой оператор для дескрипторов. Ну, или почти такой же. Вместо оператора operator+() вы должны использовать `TDesC::operator+=()`:

```
// обычные объявления
_LIT(KStringOne, "StringOne");
_LIT(KStringTwo, "StringTwo");

TBuf<128> buf1(KStringOne);
TBuf<128> buf2(KStringTwo);

// вызов оператора "+="
buf1 += buf2;
```

Но с тем же успехом вы могли бы вызвать метод `TDes::Append()`:

```
buf1.Append(buf2);
```

В некоторых, достаточно редких случаях может быть удобнее применить метод `TDesC::Insert()`. В следующем примере для наглядности используется класс `HBufC`. Приведенный ниже код гарантирует, что дескриптор содержит символ `NULL` в качестве своего последнего элемента (это может пригодиться, если ваше приложение взаимодействует с сервером, работающим с обычными С-строками):

```
HBufC *pBuf = HBufC::NewLC(100);

// добавление строки
pBuf->Des().Append(buf1);

// добавить NULL-символ в конец строки
pBuf->Des().PtrZ();

// увеличиваем длину pBuf для учета символа NULL
TInt bufLen = pBuf->Des().Length();
pBuf->Des().SetLength(bufLen + 1);

// вызываем метод Insert() класса TDes, чтобы вставить строку перед NULL
pBuf->Des().Insert(bufLen, buf2);

// удалим указатель pBuf из стека очистки,
// и освободим занимаемую им память
CleanupStack::PopAndDestroy(pBuf);
```

Обратите внимание на то, что метод `PtrZ()` не изменяет длину дескриптора, и поэтому в примере она увеличивается вручную.

Еще один способ объединить две строки – форматирование. Более подробно данный метод будет рассмотрен в главе 4.1. Нижеприведенный код демонстрирует как можно воспользоваться одним из многочисленных методов, доступных в классе `TDesC`:

```
buf1 = KStringOne;

// добавим вторую строку при помощи форматирования
buf1.AppendFormat(_L("%S"), &KStringTwo);
```

В отступлении от темы добавим, что макрос `_L()` не рекомендуется использовать в окончательном варианте кода, так как он не так эффективен как `_LIT()`. В приведенных примерах `_L()` встречается не так уж и часто, но на практике вы можете предпочесть его макросу `_LIT()`, в особенности там, где символьная константа используется всего один раз.

Что может пойти не так: обычной проблемой при объединении двух строк является недостаточный размер дескриптора (buf1 в нашем случае) для хранения объединенной строки. Если длина дескриптора будет недостаточна, ваш код вызовет панику с кодом USER 11.

3.2 Как конвертировать число в строку

Проблема: у вас есть целое или дробное число, которое вы хотите конвертировать в строку.
Решение: класс TDes содержит ряд методов для конвертации, которыми вы можете воспользоваться в любом из унаследованных от него классов:

- TDes::Num()
- TDes::NumUC()
- TDes::NumFixedWidth()
- TDes::NumFixedWidthUC()
- TDes::AppendNum()
- TDes::AppendNumUC()
- TDes::AppendNumFixedWidth()
- TDes::AppendNumFixedWidthUC()

Приведенные выше методы используют в качестве параметра целое или дробное значение и позволяют конвертировать их в строку, в соответствии с формой записи в указанной системе счисления (для целых чисел) или предопределенным форматом. Следующий фрагмент кода демонстрирует несколько возможных вариантов их использования:

```
TReal realValue = 16.0;
TUint value = 16;
TBuf<128> buf;

// конвертирование 64-битного беззнакового целого числа
buf.NumUC(value);

// конвертирование 64-битного знакового целого числа в двоичную форму
buf.Num(value, EBinary);

// конвертирование 64-битного знакового целого числа в шестнадцатеричную форму
buf.Num(value, EHex);

// конвертирование числа с плавающей точкой
TRealFormat fmt(KDefaultRealWidth, 2);
buf.Num(realValue, fmt);
```

Использование Num-методов более эффективно чем использование метода Format(), поэтому вы должны использовать их везде, где это возможно.

3.3 Как конвертировать дескриптор в число

Проблема: вы хотите конвертировать содержащуюся в дескрипторе строку в число.
Решение: в Symbian OS это весьма простая задача. Достаточно посмотреть следующий пример:

```
_LIT(KNumber, "16");

// объявление лексографического анализатора
TLex lex(KNumber);

// объявление переменной для хранения числа
TInt value = 0;


// конвертирование дескриптора в целое число
TInt err = lex.Val(value);

if (err == KErrNone)
```

```
        {
        // конвертирование прошло успешно
        console->Printf(_L("Use TLex to convert: Value = %i\n"), value);
        }
    else
        {
        // при конвертировании произошла ошибка
        console->Printf(_L("Use TLex returned err: %i\n"), err);
        }

    TReal32 realNum = 16.0;
    err = lex.Val(realNum, '.');
```

Приведенный пример демонстрирует использование класса `TLex`, который мы рассмотрим более детально в главе 5.5. Значение, возвращаемое методом `Val()`, указывает на результат операции конвертирования строки в число, включая коды возможных ошибок. Такие ошибки, к примеру, могут возникнуть, если формат символьного представления был неправильным, т.е. содержал нечисловые символы.

3.4 Как сериализовать или десериализовать дескриптор

Проблема: вы хотели бы сериализовать дескриптор в бинарное представление или восстановить дескриптор из имеющегося бинарного представления.
Решение: вам следует воспользоваться либо операторами << и >>, либо методами `ReadL()` и `WriteL()` потоковых классов `RDesReadStream`, `RFileReadStream`, `RMemReadStream`, и их `RXWriteX` аналогами.

Для того чтобы операторы << и >> смогли сохранить или восстановить данные некоторого класса из потока, в этом классе необходимо реализовать два метода: `ExternalizeL()` и `InternalizeL()`. Эти методы позволяют записать или прочитать класс из двоичной формы. Применив это правило для дескрипторов, мы можем использовать для этих целей код, похожий на тот, что приведен ниже:

```
void ExtIntSampleL()
    {
    _LIT8(KDescExtInt, "This is a descriptor");

    // создание 8-мим битного дескриптора для хранения записываемых данных
    RBuf8 rBufToExternalize;
    rBufToExternalize.CreateL(KDescExtInt(), 32);
    rBufToExternalize.CleanupClosePushL();

    // создание 8-ми битного буфера обмена
    RBuf8 rBufToWriteInto;
    rBufToWriteInto.CreateL(64);
    rBufToWriteInto.CleanupClosePushL();

    // создание объекта потока для записи дескриптора
    RDesWriteStream writeStream(rBufToWriteInto);
    writeStream.PushL();

    // сериализация данных
    writeStream << rBufToExternalize;

    // завершение работы с потоками
    writeStream.Close();
    writeStream.Pop();

    // создание 8-ми битного дескриптора для хранения считываемых данных
    RBuf8 rBufToInternalize;
    rBufToInternalize.CreateL(64);
    rBufToInternalize.CleanupClosePushL();
```

```
// создание объекта потока для чтения дескриптора
RDesReadStream readStream(rBufToWriteInto);
readStream.PushL();

// десериализация данных
readStream >> rBufToInternalize;

// завершение работы с потоками
readStream.Close();
readStream.Pop();

// закрытие и уничтожение всех объектов RBuf
CleanupStack::PopAndDestroy(3);
}
```

Что может пойти не так: операторы << и >> могут вызвать сброс (leave), учтите это при разработке кода.

3.5 Как записать двоичные данные в дескриптор

Проблема: вы хотели бы сохранить двоичные данные внутри дескриптора.
Решение: вы можете воспользоваться последовательностями вида "\xHH" (в шестнадцатеричной системе счисления) чтобы поместить двоичные данные в макрос _LIT:

```
//                    B       i       n       a       r       y
_LIT(KBinaryStuff, "\x42\x00\x69\x00\x6e\x00\x61\x00\x72\x00\x79\x00");
```

Приведенная в примере последовательность определяет строку «Binary» в Unicode. Тот же самый фокус вы можете проделать и с 8-ми битными символьными дескрипторами. Так как дескриптор всегда знает о размере своих данных, его поведение очень схоже с поведением строк в Бейсике и BSTR из мира Windows.

Для обычных дескрипторов вы можете использовать тот же прием и метод Append():

```
TBuf8<32> buffer;
TUint8 data[6] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05};

// сохранение данных в дескриптор
buffer.Append(data, sizeof(data));

// добавление нескольких байтов
buffer.Append(0x06);
buffer.Append(0x07);
buffer.Append(0x08);
```

Что может пойти не так: вам, возможно, придется столкнуться с проблемой очередности байтов при использовании Unicode дескрипторов. Эта проблема затрагивает макросы _LIT и _LIT16, а так же все 16-ти битные дескрипторы. Причиной всему этому служит тот факт, что "\xHH" в Unicode строке означает "0x00HH", а в Symbian OS будет сохранен как "0xHH00".

4 Более сложные рецепты

4.1 Как форматировать строки

Проблема: у вас есть несколько различных переменных, которые вы хотели бы отформатировать и конвертировать в дескриптор.
Решение: класс TDes имеет несколько методов, содержащих слово Format в своих названиях. Нижеприведенный код демонстрирует лишь несколько возможных директив форматирования для различных типов данных. Полный список форматирующих директив вы можете найти в документации Symbian Development Library. Итак:

```
_LIT(KFmtBin, "Binary: %bb\n");
```

```
_LIT(KFmtOct, "Octal: %o\n");
_LIT(KFmtInt32, "Integer: %i\n");
_LIT(KFmtInt64, "64-bit integer: %Ld\n");
_LIT(KFmtReal, "Real: %f\n");
_LIT(KFmtUint, "Uint: %u\n");
_LIT(KFmtHex, "Hex: 0x%x\n");
_LIT(KFmtUint64, "Uint64: %Lu\n");
_LIT(KFmtCStr, "C-String: %s\n"); // обратите внимание на маленькую букву "s"
_LIT(KFmtDesc, "Descriptor: %S\n"); // обратите внимание на большую букву "S"

TBuf<128> buf1;
buf1 = KStringOne;

_LIT(KFormatSample, "FormatSample\n");
console->Write(KFormatSample);

TInt intVal = 16;

buf1.Format(KFmtBin, intVal);
console->Write(buf1);

buf1.Format(KFmtOct, intVal);
console->Write(buf1);

buf1.Format(KFmtInt32, intVal);
console->Write(buf1);

buf1.Format(KFmtInt64, (TInt64)intVal);
console->Write(buf1);

buf1.Format(KFmtHex, intVal);
console->Write(buf1);

buf1.Format(KFmtUint, -intVal);
console->Write(buf1);

buf1.Format(KFmtUint64, (TUint64)-intVal);
console->Write(buf1);

TBuf<128> buf2(KStringTwo);

buf1.Format(KFmtDesc, &buf2);
console->Write(buf1);

buf2.Append(0);

// или то же самое: const TUint16* pData = buf2.Ptr();

const TText* pData = buf2.Ptr();

buf1.Format(KFmtCStr, pData);
console->Write(buf1);
```

Когда вы используете группу методов `AppendFormat()`, то возможных переполнений буфера можно избежать при передаче указателя на объект, унаследованного от класса `TDesOverflow`:

```
class TSampleOverflow : public TDesOverflow
    {
    void Overflow(TDes& aDes);
    };

void TSampleOverflow::Overflow(TDes& aDes)
    {
    // данная функция обрабатывает возникновение переполнения буфера
    _LIT(KTextDescOverflow, "Got descriptor overflow - maxlength = %d\n");
```

```
    console->Printf(KTextDescOverflow, aDes.MaxLength());
    }

...

TSampleOverflow overflowHandler;
TBuf<9> buf1;
TBuf<128> buf2(KStringTwo);

// здесь мы достигаем максимально длины дескриптора
buf1.AppendFormat(KFmtDesc, &overflowHandler, &buf2);

// а здесь будет вызван наш обработчик переполнения
buf1.AppendFormat(KFmtDesc, &overflowHandler, &buf2);
```

Что может пойти не так: используя дескрипторы для указания параметров форматирования, вы можете ошибочно воспользоваться директивой %S, вместо директивы %s, и наоборот. Директива %S используется для указателя на дескриптор, в то время как директива %s означает C-строку, или char*. Также, позабытый знак & перед переменной дескриптора будет означать потерю большого количества времени в поисках причины сбоя в работе кода.

4.2 Как конвертировать C-строку в дескриптор (и наоборот)

Проблема: у вас имеется буфер с C-строкой, и вы хотите конвертировать его в дескриптор, либо наоборот.
Решение: дескрипторы Symbian OS обычно не содержат в конце символ NULL (как, например, BSTR-строки в Windows). Тем не менее, вы можете: 1) легко добавить NULL-символ в конец данных дескриптора; 2) выполнять конвертацию данных из дескрипторной формы в форму C-строки и наоборот.

```
_LIT(KFmtCStr, "C-string: %s\n");   // в директиве маленькая буква s
_LIT(KFmtDesc, "Descriptor: %S\n"); // в директиве большая буква S

TBuf<128> buf(KStringOne);
TUint cstrBuffer[100];

// обнуление буфера
memset(cstrBuffer, 0, sizeof(cstrBuffer));

// копирование данных дескриптора в C-строку
memcpy(cstrBuffer, buf.Ptr(), buf.Size());

console->Printf(KFmtCStr, cstrBuffer);

// выполнение обратного преобразования
// метод PtrZ() автоматически добавляет NULL-символ в конец данных дескриптора
const TText* text = buf.PtrZ();
console->Printf(KFmtCStr, text);
```

Что может пойти не так: если размер дескриптора будет недостаточным для дополнительного NULL-символа, вызов метода TDes::PtrZ() вызовет панику USER 23.

4.3 Как конвертировать 8-ми битную строку в 16-ти битную (и наоборот)

Проблема: у вас есть 8-ми битный или 16-ти битный дескриптор, который вы хотели бы конвертировать в дескриптор с другим размером символов.
Решение: применяемое решение будет зависеть от конкретной задачи. В простейшем случае, такой задачей является конвертирование ASCII-строки в ее Unicode аналог. В этом случае все, что вам будет нужно сделать – это выполнить следующий код:

```
_LIT8(K8bitDesc, "8-bit");

TBuf8<128> buf1(K8bitDesc);

// конвертирование 8-ми битного дескриптора в 16-ти битный
TBuf16<128> buf2;
buf2.Copy(buf1);
// вывод дескриптора на печать
console->Write(buf2);
```

Как вы можете убедиться, метод `TDes::Copy()` делает всю работу за вас. Подобная конвертация могла бы быть удобной во многих случаях, например, когда вы работаете с сетью, где данные часто передаются в 8-ми битной форме.

Методы `Collapse()` и `Expand()` так же могут быть использованы для конвертаций между 8-ми битными и 16-ти битными дескрипторами:

```
_LIT(K16bitDesc, "16-bit");

TBuf<128> buf1(K16bitDesc);

// конвертирование 16-ти битного дескриптора в 8-ми битный,
// и возвращение указателя на него
TPtr8 narrowPtr = buf1.Collapse();

// конвертирование 8-ми битного дескриптора в 16-ти битный,
// и возвращение указателя на него
TPtr widePtr = narrowPtr.Expand();
```

Для более сложных преобразований, в особенности, когда вы работаете с национальными или UTF кодировками, вам необходимо воспользоваться классом `CnvUtfConverter` (для компиляции нижеприведенного кода необходимо подключить библиотеку `charconv.lib`):

```
// конвертирование из Unicode в UTF-8
HBufC8* pHeapUTF = CnvUtfConverter::ConvertFromUnicodeToUtf8L(buf2);
CleanupStack::PushL(pHeapUTF);


// конвертирование из UTF-8 в Unicode
HBufC* pHeapUCS2 = CnvUtfConverter::ConvertToUnicodeFromUtf8L(*pHeapUTF);
CleanupStack::PushL(pHeapUCS2);

// вывод Unicode строки на печать
console->Write(*pHeapUCS2);

// удаление использованных указателей из стека очистки,
// и освобождение занимаемой ими памяти
CleanupStack::PopAndDestroy(2);
```

Что может пойти не так: если вы работаете с национальными кодировками, то некоторые из символов могут быть конвертированы неправильно.

4.4 Как передавать дескрипторы в качестве аргументов функций

Проблема: вы хотели бы использовать дескриптор в качестве параметра функции.
Решение: стандартный подход при решении этой проблемы – использование классов `TDesC` и `TDes`. С его помощью, вы так же разрешаете пользователям вашей функции передавать в нее и другие типы дескрипторов:

```
// функция с неизменяемым параметром
void FuncWithReadOnlyParameter(const TDesC& aConstData);
```

```
// функция с изменяемым параметром
void FuncWithModifiableParameter(TDes& aData);
```

Используя вышеприведенные объявления, вы можете передавать в функции переменные типов TPtr, RBuf, или любых других типов, унаследованных от классов TDesC и TDes. Очевидно, что подобные объявления ограничивают набор доступных операций над передаваемыми аргументами до набора методов базовых классов TDesC и TDes. В случае, когда вы сознательно хотите работать с переменными типа TPtr, а не с TDes, вы можете объявить аргументы функций так, как вам будет удобно. Но все же, при этом вам следует хорошенько подумать – для отказа от использования базовых классов в качестве параметров должна быть действительно хорошая причина, ведь они и так имеют большое количество методов. Использование аргументов специализированных классов редко бывает обосновано.

А теперь давайте более детально рассмотрим параметры вышеприведенных функций. Возможно, в первом методе вам покажется несколько необычным использование оператора const для немодифицируемого дескриптора. Что ж, действительно, дескриптор является «немодифицируемым» согласно терминологии Symbian OS[1], однако его данные все же можно редактировать через дескриптор-указатель TPtr:

```
// получим указатель на данные "немодифицируемого" дескриптора
TPtr hackedTDesC(aConstData.Des());

// изменим данные "немодифицируемого" дескриптора
hackedTDesC.Append(anotherTPtr);
```

Использование оператора const усиливает синтаксис C++ и предотвращает подобные коварные операции уже на этапе компиляции. Вдобавок, объявления с const позволяют передавать в функции символьные дескрипторы, заданные с помощью макроса _L(), что просто невозможно в любых других случаях.

Что может пойти не так: совершенно очевидный прием в среде с ограниченными ресурсами – передача потенциально больших блоков данных по ссылке, а не по значению. Передача по ссылке позволяет не перегружать стек, как это обычно происходит при передаче данных по значению. Поэтому мы и воспользовались оператором & в вышестоящем примере объявления функций. Если мы отодвинем на задний план тему неправильного использования параметров (а в данной области мы можем быть весьма искусны), то часто возникающей проблемой при передаче дескрипторов в функции будет их переполнение. Помните о том, что управление памятью – обязанность разработчика!

Кроме того, никогда не пытайтесь создавать и инициализировать объекты базовых дескрипторных классов. Хотя документация в SDK и описывает их как «абстрактные», они все же отличаются в этом смысле от абстрактных объектов в Си++. Возможно ваш код и скомпилируется, но работать не будет.

4.5 Как передавать и возвращать модифицируемые дескрипторы

Проблема: вы хотели бы передавать в функцию и возвращать из нее модифицируемые дескрипторы.
Решение: как вы уже знаете из предыдущей главы, все, что нужно сделать – это объявить аргумент функции как TDes:

```
void SomeFunction(TDes& aData);
```

В теле функции вы можете считывать и записывать данные в аргумент, следя за тем, чтобы он имел достаточный размер для последующих операций.

---

[1] Содержащиеся в «немодифицируемом» дескрипторе данные можно получить, но их нельзя изменить при помощи методов самого дескриптора. Однако они могут быть изменены через дескриптор TPtr или даже заменены целиком с помощью оператора присваивания.

Теперь рассмотрим возвращение модифицируемого дескриптора из функции. Нижеприведенный пример показывает один из возможных способов:

```
TPtr TDesCDerivedClass::RightL(TInt aLength) const
    {
    if (aLength < 0)
        {
        // отрицательная длина
        User::Leave(KErrArgument);
        }

    TInt len = Length();

    if (aLength > len)
        {
        aLength = len;
        }

    return (TPtr((TUint16*)Ptr() + len - aLength, aLength, aLength));
    }
```

Как и в обычном C++, возвращаемые значения должны быть похожи на указатели, т.е. необходимо использовать `TPtr`, `HBufC*`, или любые другие классы, которые не размещают свои данные в стеке.

Что может пойти не так: самая коварная ошибка, которую вы можете допустить, – это забыть объявить аргументы функции как ссылки.

5 Сложные рецепты

5.1 Как конвертировать дескриптор из одного типа в другой

Проблема: у вас есть дескриптор одного типа, и вы хотите привести его к другому.
Решение: посмотрите на нижеприведенный код, конвертирующий дескрипторы всевозможных типов. Вы увидите комбинацию конструкторов, операторов присваивания, а также методов `Des()` и `Ptr()`:

```
_LIT(KTestBuf, "Buffer");

TBufC<16> bufC1(KTestBuf);
TBuf<16> buf1(bufC1.Des());

// TBuf -> TBufC
TBuf<16> buf2(KTestBuf);
TBufC<16> bufC2(buf2);


// TBuf/TBufC -> TPtr
TPtr ptr1(0, 0);
ptr1.Set((TUint16*)(buf2.Ptr()), buf2.Length(), 32);
TPtr ptr2(bufC2.Des());

// TPtr -> TBuf/TBufC
TBuf<16> buf3 = ptr1;
TBufC<16> bufC3 = ptr2;

// TBuf/TBufC -> TPtrC
TPtrC ptrC1(ptr1);
TPtrC ptrC2(buf2);
TPtrC ptrC3(bufC2);
```

```
// всё вышеперечисленное -> HBufC
HBufC* pHeap1 = ptrC1.AllocLC();
HBufC* pHeap2 = ptr1.AllocLC();
HBufC* pHeap3 = bufC1.AllocLC();
HBufC* pHeap4 = buf1.AllocLC();

// то же самое можем проделать и через оператор присваивания
HBufC* pHeap5 = HBufC::NewLC(32);
*pHeap5 = ptr1;

// либо просто через метод Copy()
HBufC* pHeap6 = HBufC::NewMaxLC(32);
pHeap6->Des().Copy(ptr2);

// удаление использованных указателей из стека очистки,
// и освобождение занимаемой ими памяти
CleanupStack::PopAndDestroy(6);
```

Что может пойти не так: базовые и унаследованные классы дескрипторов так же содержат в себе методы `Ptr()` и `Des()`, так что вы можете легко запутаться, каким из них пользоваться и когда. В результате возможны ошибки компилирования.

5.2 Как использовать RBuf, когда нужен дескриптор, размер которого меняется динамически

Проблема: вы хотите воспользоваться динамически изменяемым дескриптором.
Решение: класс `RBuf` легок в использовании, данный класс сочетает в себе характеристики как дескриптора-указателя – `TPtr`, так и дескриптора, хранящего данные в памяти кучи, – `HBufC`. Класс `RBuf` может обладать собственной памятью, что демонстрируется в следующем примере:

```
// создание RBuf
RBuf rBuf;
rBuf.CreateL(6);

// помещение в стек очистки
rBuf.CleanupClosePushL();

// теперь мы можем копировать, изменять размер, и добавлять новые данные в буфер
_LIT(KHello, "Hello ");
_LIT(KWorld, "World!");


// запишем в rBuf первое слово
rBuf.Copy(KHello());


// увеличим размер rBuf так, чтобы в нем хватило места и для второго слова
rBuf.ReAllocL(12);


// добавим второе слово к содержимому rBuf
rBuf.Append(KWorld);


// удаление rBuf из стека очитки, и освобождение занимаемой им памяти
CleanupStack::PopAndDestroy();

// теперь rBuf можно использовать снова для указания на другой буфер,
// с помощью метода RBuf::Assign()
HBufC* hBuf = KHello().AllocL();
rBuf.Assign(hBuf);

// сделаем что-нибудь еще, и снова освободим занимаемую им память
...
```

Более подробную информацию о классе RBuf вы можете найти в статье «Introduction to RBuf», доступной на сайте Symbian Developer Network по адресу developer.symbian.com/main/documentation/symbian_cpp/symbian_cpp/index.jsp.

Что может пойти не так: как и в случае с любыми другими типами дескрипторов, главная опасность таится в работе с памятью. Например, вы можете переполнить дескриптор данными. Или дескриптор, на который вы ссылаетесь, может быть случайно удален в тот самый момент, когда ваш код начинает работу с ним. Другой частой проблемой является необходимость вызова метода RBuf::Close(), и освобождения связанной с объектом RBuf памяти. Использование метода RBuf::CleanupClosePushL() частично решает эту проблему, однако все же будьте осторожны с присваиванием новой строки уже существующему экземпляру RBuf, либо при смене владельца выделенной цепочки памяти.

5.3 Как найти подстроку в дескрипторе, сопоставляя шаблонам поиска

Проблема: вы хотели бы найти заданную подстроку в имеющемся дескрипторе.
Решение: Различные Find(), Match(), и Locate() методы должны удовлетворить любым вашим потребностям в простом поиске строк внутри дескрипторов. Дополнительно к основной функциональности, понятной из их названий, некоторые варианты этих методов позволяют производить нормализацию (folding) и сопоставление (collation).[2] Нижеприведенный пример иллюстрирует использование только двух методов из этой группы – Match(), и MatchF():

```
//
// поисковые запросы (шаблоны)
//

// найти слово "World", обрамленное любыми другими символами
_LIT(KTxtMatchstr1, "*World*");

// найти слово, начинающееся на "W", затем содержащее один любой символ,
// и заканчивающееся на "rld"
_LIT(KTxtMatchstr2, "*W?rld*");

// найти слово, начинающееся на "Wor"
_LIT(KTxtMatchstr3, "Wor*");

// найти слово "Hello", и только
_LIT(KTxtMatchstr4, "Hello");

// найти букву "W", обрамленную любыми другими символами
_LIT(KTxtMatchstr5, "*W*");

// найти слово, начинающееся на "hello"
_LIT(KTxtMatchstr6, "hello*");

// найти любой символ, или слово,
// в том числе и отсутствие каких-либо символов или слов
_LIT(KTxtMatchstr7, "*");

_LIT(KMatchStr, "Match %S\n");
```

---

[2] «Нормализация», или «folding», являет собой относительно простой метод нормализации текста, т.е. удаления из него всех различий в регистрах, замены диакритических знаков на знаки без диакритических символов, и т.д. «Сопоставление», или «collation», представляет собой улучшенный и более действенный способ сравнения строк, выполняющегося по определенным правилам. Каждая локализация Symbian OS обычно содержит в себе собственный стандартный набор таких правил.

```
_LIT(KTxtNotFound, "Not Found");
_LIT(KTxtFound, "Found");
_LIT(KFormatStr, "%S %S (idx = %d)\n");
_LIT(KTxtHelloWorld, "Hello World!");

const TBufC<16> bufc(KTxtHelloWorld);
TInt index;
TInt pos;
TPtrC genptr;
TBufC<8> matchstr[7] = {
                        *&KTxtMatchstr1, // "*World*"
                        *&KTxtMatchstr2, // "*W?rld*"
                        *&KTxtMatchstr3, // "Wor*"
                        *&KTxtMatchstr4, // "Hello"
                        *&KTxtMatchstr5, // "*W*"
                        *&KTxtMatchstr6, // "hello*"
                        *&KTxtMatchstr7  // "*"
                        };


// поиск строк с использованием метода Match()
for (index = 0; index < 7; index++)
    {
    pos = bufc.Match(matchstr[index]);

    if (pos < 0)
        {
        // ничего не найдено
        genptr.Set(KTxtNotFound);
        }
    else
        {
        // что-то нашли
        genptr.Set(KTxtFound);
        }

    console->Printf(KFormatStr, &genptr, &matchstr[index], pos);
    }

// поиск при помощи метода MatchF():
// в данном случае будет найдена шестая строка
for (index = 0; index < 7; index++)
    {
    pos = bufc.MatchF(matchstr[index]);

    if (pos < 0)
        {
        // ничего не найдено
        genptr.Set(KTxtNotFound);
        }
    else
        {
        // что-то нашли
        genptr.Set(KTxtFound);
        }

    console->Printf(KFormatStr, &genptr, &matchstr[index], pos);
    }
```

Что может пойти не так: используя методы `Match()` и `MatchF()`, вы не сможете искать в строках символы ? и *, т.к. эти методы не поддерживают никаких escape-символов. Однако метод `MatchC()` не имеет таких ограничений, и вы можете им воспользоваться.

5.4 Как сравнивать дескрипторы

Проблема: вы хотите сравнить содержимое двух дескрипторов.

Решение: класс TDesC имеет несколько методов для сравнения как текстовых, так и двоичных данных. Нижеприведенный пример иллюстрирует несколько возможных сценариев:

```
void CompareSample()
    {
    // методы Compare() и CompareF() могут сравнивать любые типы данных,
    // для двоичных данных используйте просто Compare(),
    // а для текстовых данных используйте Compare(), CompareF() или CompareC()
    const TBufC<16> compstr[7] =
        {
        *&KTxtCompstr1, // "Hello World!@@"
        *&KTxtCompstr2, // "Hello"
        *&KTxtCompstr3, // "Hello Worl"
        *&KTxtCompstr4, // "Hello World!"
        *&KTxtCompstr5, // "hello world!"
        *&KTxtCompstr6, // "Hello World "
        *&KTxtCompstr7  // "Hello World@"
        };

    for (index = 0; index < 7; index++)
        {
        if ((bufc.Compare(compstr[index])) < 0)
            {
            // "меньше чем"
            genptr.Set(KTxtlessthan);
            }
        else
            {
            if ((bufc.Compare(compstr[index])) > 0)
                {
                // "больше чем"
                genptr.Set(KTxtGreaterThan);
                }
            else
                {
                // строки одинаковы
                genptr.Set(KTxtEqualTo);
                }
            }

        _LIT(KFormat,"\"%S\"%S\"%S\"\n");
        console->Printf(KFormat, &bufc, &genptr, &compstr[index]);
        }

    // метод CompareF() игнорирует регистры символов,
    // поэтому четвертая и пятая строки окажутся равными
    for (index = 3; index < 5; index++)
        {
        if ((bufc.CompareF(compstr[index])) < 0)
            {
            // "меньше чем"
            genptr.Set(KTxtLessThan);
            }
        else
            {
            if ((bufc.CompareF(compstr[index])) > 0)
                {
                // "больше чем"
                genptr.Set(KTxtGreaterThan);
                }
            else
                {
                // строки одинаковы
                genptr.Set(KTxtEqualTo);
```

```
                }
            }

        _LIT(KTxtusingCF," (using CompareF())");
        _LIT(KFormatF,"\"%S\"%S\"%S\"%S\n");
        console->Printf(KFormatF, &bufc, &genptr, &compstr[index], &KTxtusingCF);
        }
```

Что может пойти не так: вы можете обнаружить, что нормализация не всегда работает так, как вы того ожидаете, особенно при использовании некоторых кодировок.

5.5 Как использовать TLex

Проблема: у вас есть дескриптор, и вы хотели бы провести лексический анализ хранимых в нем данных.

Решение: анализ строковых данных – достаточно распространенная задача. Например, когда ваше приложение обрабатывает какие-то команды или работает с данными GPS. В таких случаях лучше всего воспользоваться классом `TLex`. Нижеприведенный код выполняет простой разбор команд:

```
// функция анализирует входящую строку,
// и пытается найти в ней команды "exit" или "help"
TInt ParseCommand(const TDesC& aCommand)
    {
    TLex lex(aCommand);
    TInt res = KErrNone;
    TBool help = EFalse;
    TPtrC ptr;

    for (ptr.Set(lex.NextToken()); ptr.Length(); ptr.Set(lex.NextToken()))
        {
        if (!ptr.CompareF(_L("q")) || !ptr.CompareF(_L("quit"))
                                   || !ptr.CompareF(_L("exit")))
            {
            return KErrEof;
            }

        if (!ptr.CompareF(_L("help")))
            {
            help = ETrue;
            }
        else if (ptr.Length() == 2)
            {
            if (!ptr.CompareF(_L("-H")))
                {
                help = ETrue;
                }
            else
                {
                // обработка других возможных команд
                }
            }
        }
    }
```

Возможности класса `TLex` могут быть лучше продемонстрированы на примере более сложного разбора данных. Например, при анализе данных GPS:

```
// функция анализирует строку формата GGA, например, такую:
// $GPGGA,123519,4807.038,N,01131.000,E,1,08,0.9,545.4,M,46.9,M,,*47
void GpsParseGGASentence(const TDesC &aNmea)
```

```
{
TReal latitude, longitude;

_LIT(KSpace, " ");
RBuf tmpBuf;

if (tmpBuf.Create(aNmea) != KErrNone)
    {
    // ошибка при инициализации RBuf
    return;
    }

// заменим все запятые в предложении на пробелы,
// чтобы иметь возможность использовать метод TLex::NextToken()
TInt commaPos = -1;

while ((commaPos = tmpBuf.Locate(',')) > KErrNotFound)
    {
    tmpBuf.Replace(commaPos, 1, KSpace);
    }

// создаем объект TLex с подготовленными данными
TLex nmeaSentence(tmpBuf);

// элементы строки сохраняются в переменных
// исключительно в демонстрационных целях
TPtrC firstToken = nmeaSentence.NextToken();
TPtrC secondToken = nmeaSentence.NextToken();
TPtrC thirdToken = nmeaSentence.NextToken();


// географическая широта
TLex tokenLat(thirdToken);


// получение численного значения географической широты
tokenLat.Val(latitude);

// проверяем, северная это широта, или южная
TPtrC auxLat = nmeaSentence.NextToken();

if (auxLat.Compare(_L("S")) == 0)
    {
    // южная широта
    latitude *= -1;
    }

// географическая долгота
TLex tokenLon(nmeaSentence.NextToken());

// получение численного значения географической долготы
tokenLon.Val(longitude);

// проверяем, западная это долгота, или восточная
TPtrC auxLon = nmeaSentence.NextToken();

if (auxLon.Compare(_L("W")) == 0)
    {
    // западная долгота
    longitude *= -1;
    }
}
```

Что может пойти не так: класс `TLex` использует пробелы в качестве разделителя содержащихся в строке элементов. Поэтому, если сами элементы могут содержать пробелы, то вам следует производить разбор с большей осторожностью. Вас также может немного

озадачить работа с отмеченной и текущей позициями при разборе данных, и т.д.

6 Заключение

В этой статье мы кратко описали некоторые операции над дескрипторами. С прочими вы можете познакомиться в ходе собственных экспериментов. Я надеюсь, что данная статья ответила на многие из ваших вопросов касательно того, как решить ту или иную задачу и сделала вас более благосклонными к дескрипторам. Совершенно очевидно, что мы не могли раскрыть все возможные вопросы за один раз, поэтому вам следует обратиться за дополнительной информацией к документам Symbian Development Library и коду демонстрационного примера.

Исходный код всех изложенных в данной статье примеров вы можете найти по адресу:

- developer.symbian.com/main/downloads/papers/Descriptors/Descriptors.zip

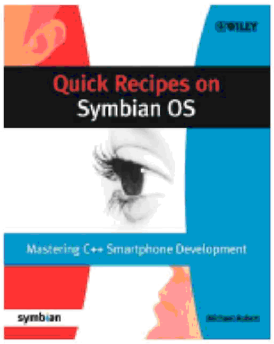
Дополнительную полезную информацию о дескрипторах вы можете найти здесь:

- descriptors.blogspot.com
- forum.nokia.com/info/sw.nokia.com/id/9c61aadd-8ae7-4c19-9484-2b165f0cb55e/S60_Platform_Descriptor_Example_v2_1_en.zip.html

Больше информации вы так же можете найти в книгах издательства Symbian Press, описание которых расположено по адресу:

- developer.symbian.com/books

7 Об авторе



Алексей начал работать с мейнфреймами в конце 80-х, используя языки программирования Pascal и REXX, однако вскоре переключился на C/C++ и Java для различных платформ. Затем он обратился к мобильным технологиям. После почти десятилетней работы в качестве начальника отдела и руководителя проектов для платформы Windows Mobile, он решил присоединиться к команде разработчиков Symbian Core Development, изначально работавшей над системой безопасности Symbian OS, а затем – над USB.

Алексей имеет диплом магистра прикладной математики и физики Московского физико-технического института. Он так же является аккредитованным разработчиком Symbian и автором многих статей на сайте www.developer.com. Помимо этого, он является автором нескольких глав популярной книги издательства Symbian Press «Qiuck Recipes on Symbian OS» (более подробную информацию о которой вы можете найти по адресу developer.symbian.com/quick).