# Descriptors

## Part One

# Introduction

## A Symbian OS string is known as a descriptor

- As it is self-describing
- A descriptor holds the length of the string of data it represents as well as its type (which identifies the underlying memory layout of the descriptor data)

## Descriptors have something of a reputation among Symbian OS programmers

- The key point to remember is that they were designed to be very efficient on low-memory devices,
- Use the minimum amount of memory necessary to store a string while describing it in terms of its length and layout

# Introduction

Descriptors are not like standard C++ strings, Java strings or the MFC CString

- Their underlying memory allocation and cleanup must be managed by the programmer

Descriptors are also are also not like C strings

- They protect against buffer overrun and do not rely on NULL terminators to determine the length of the string

symbian Academy

# Descriptors

## Features of Symbian OS Descriptors

- Understand that Symbian OS descriptors may contain text or binary data
- Know that descriptors may be narrow (8-bit), wide (16-bit) or neutral (which is 16-bit since Symbian OS is built for Unicode)
- Understand that descriptors do not dynamically extend the data area they reference, so will panic if too small to store data resulting from a method call

# Character Size

## Symbian OS

- Has been built to support Unicode character sets with wide (16-bit) characters by default
- The descriptors in early releases of Symbian OS (EPOC) up to Symbian OS v5 had 8-bit native characters

## The character width of descriptor classes can be identified from their names.

- Class names ending in 8 (e.g. TPtr8) it have narrow (8-bit) characters
- Class name ending with 16 (e.g. TPtr16) manipulates 16-bit character strings

# Character Size

There is also a set of neutral classes

- Which have no number in their name (e.g. **`TPtr`**)
- The neutral classes are **`typedef`** 'd to the character width set by the platform
- The neutral classes were defined for source compatibility purposes to ease the switch between narrow and wide builds

Today Symbian OS is always built with wide characters

- It is recommended practice to use the neutral descriptor classes where the character width does not need to be stated explicitly.

# Memory Management

Descriptor classes do not dynamically manage the memory used to store their data

- The modification methods check that the maximum length of the descriptor is sufficient for the operation to succeed
- If it is not, they do not re-allocate memory for the operation
- But instead panic to indicate that an overflow would have occurred

# Memory Management

## The contents of the descriptor can shrink and expand

- up to the maximum length allocated to the descriptor

## Before calling a descriptor method which expands the data

- The developer should implement the necessary checks to ensure that there is sufficient space available in the descriptor
- The contents of the descriptor can shrink and expand up to the maximum length allocated to the descriptor

# Descriptors

## The Symbian OS Descriptor Classes

- Know the characteristics of the **`TDesC, TDes, TBufC, TBuf, TPtrC, TPtr, RBuf`** and **`HBufC`** descriptor classes
- Understand that the descriptor base classes **`TDesC`** and **`TDes`** implement all generic descriptor manipulation code, while the derived descriptor classes merely add construction and assignment code specific to their type
- Identify the correct and incorrect use of modifier methods in the **`TDesC`** and **`TDes`** classes
- Recognize that there is no **`HBuf`** class, but that **`RBuf`** can be used instead as a modifiable dynamically allocated descriptor

# Key Descriptor Class Concepts

These will be covered later, but to proceed it is helpful to be aware of the following points

- TDesC is the base class for all descriptors
- There are currently six derived descriptor classes (TPtrC, TPtr, TBufC, TBuf, HBufC and RBuf)

Each subclass

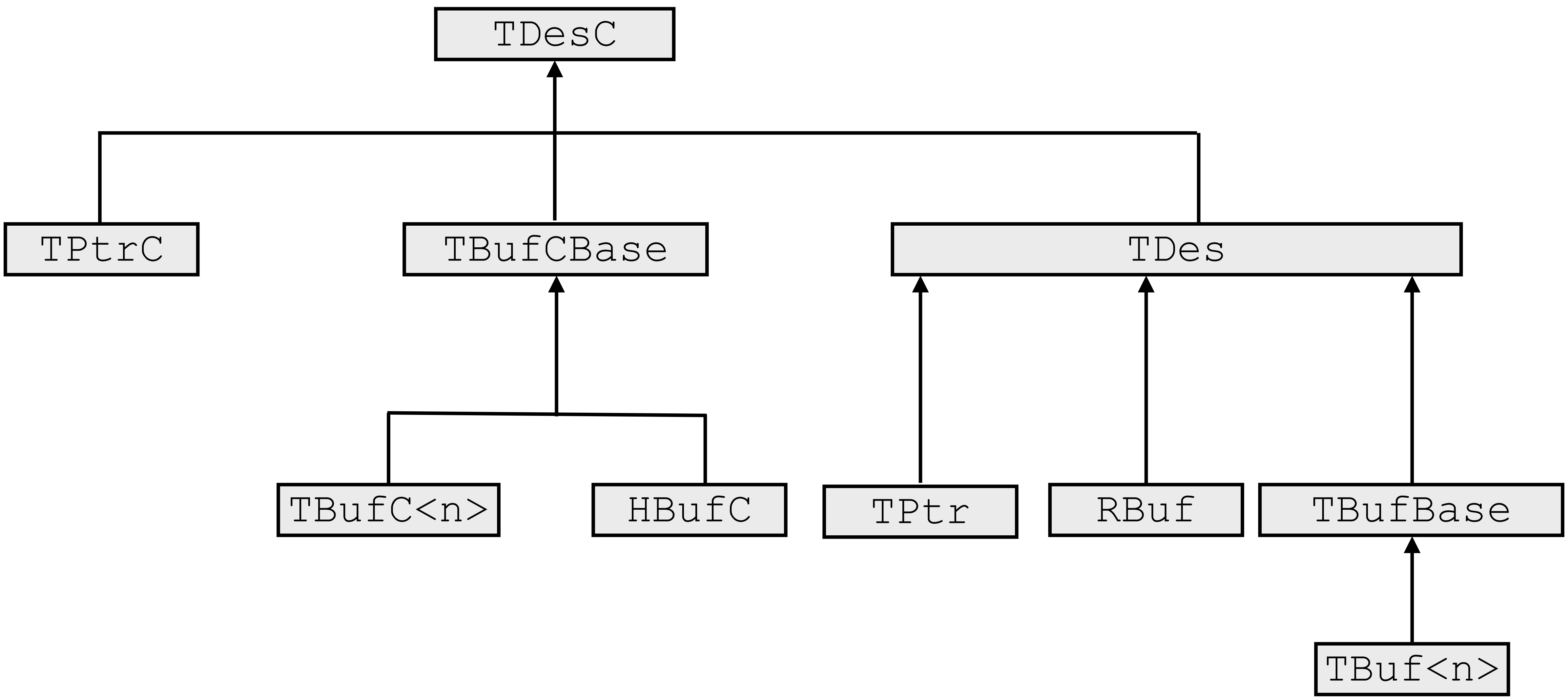
- Does not implement its own data access method using virtual function overriding
- This would add an extra 4 bytes to each derived descriptor object for a virtual pointer (vptr) to access the virtual function table
- 4 bits of the 4 bytes that store the length of the descriptor object are used to indicate the class type of the descriptor

# Key Descriptor Class Concepts

Descriptors come in two basic layouts:

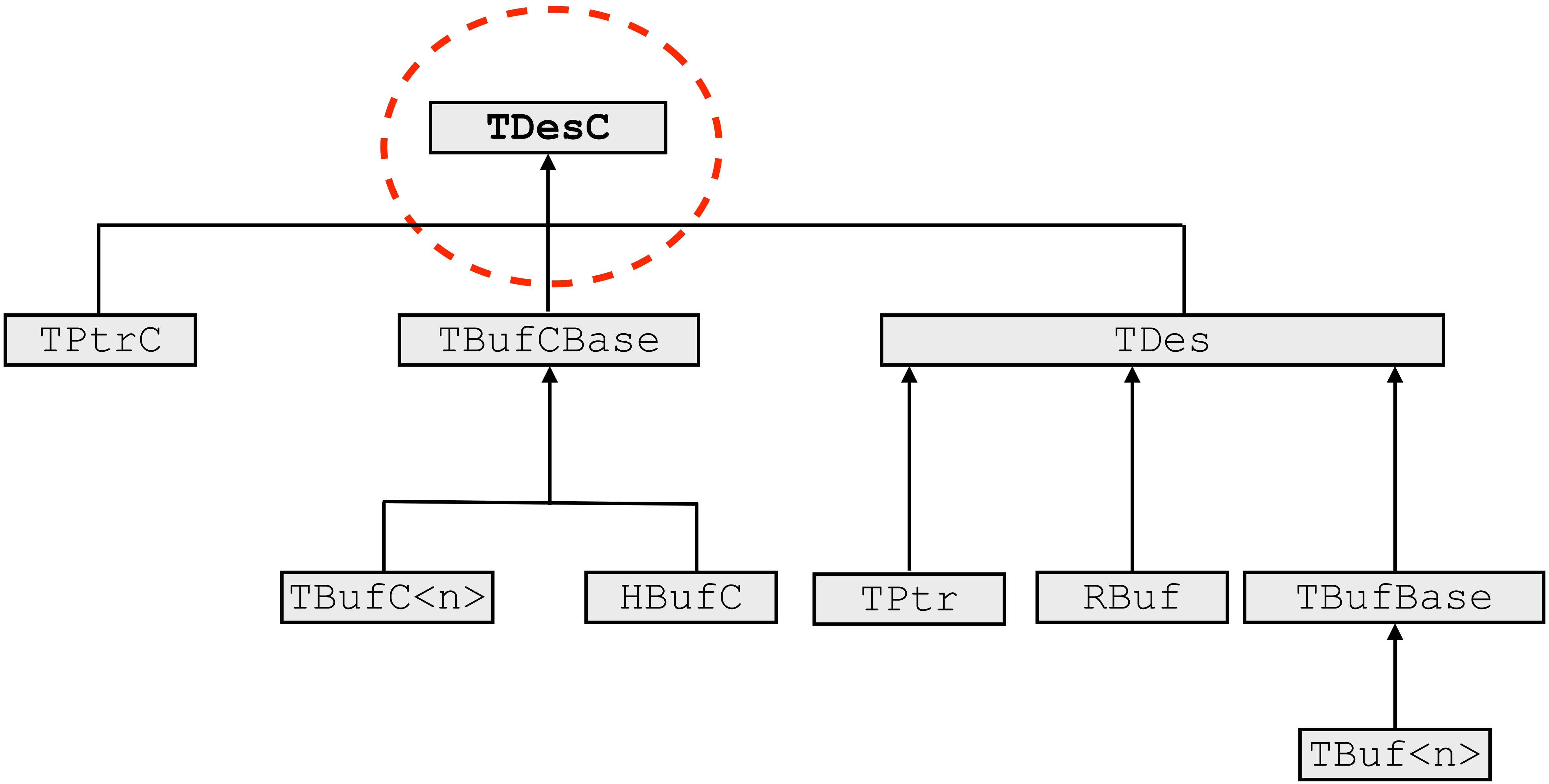
- Pointer descriptors - in which the descriptor holds a pointer to the location of a character string stored elsewhere
- Buffer descriptors - where a string of characters forms part of the descriptor

# Descriptor Class Inheritance Hierarchy




A first glance at the inheritance tree

# Symbian OS Descriptor Class: **`TDesC`**






symbian Academy

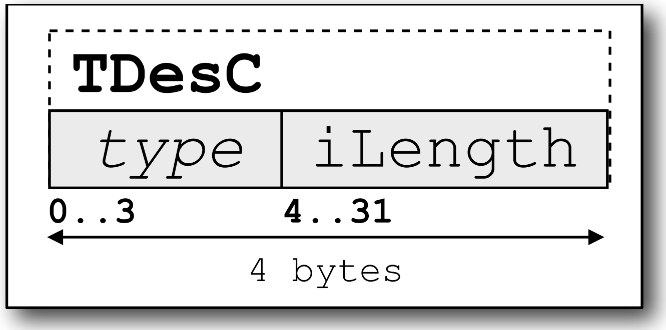
# Symbian OS Descriptor Class: `TDesC`

All the descriptor classes derive from the base class **`TDesC`**

- Apart from the literal descriptors (discussed later)
- The T prefix indicates a simple type class
- The C suffix reflects that the class defines a non-modifiable type of descriptor i.e. constant

# Symbian OS Descriptor Class: **`TDesC`**

**`TDesC`** defines the fundamental layout of every descriptor type:




To identify each of the derived classes

- The top 4 bits are used to indicate the type of memory layout of descriptor
- Remaining 28 bits are used to hold the length of the descriptor data
- Thus the maximum length of a descriptor is limited to 2^28 bytes (256 MB) Plenty!

# Symbian OS Descriptor Class: `TDesC`

## `TDesC` provides methods

- For determining the length of the descriptor - **`Length()`**
- Accessing the data - **`Ptr()`**

## Using `Length()` and `Ptr()` methods

- The **`TDesC`** base class can implement all the operations typically possible on a constant string object, such as data access, comparison and search

## The derived classes

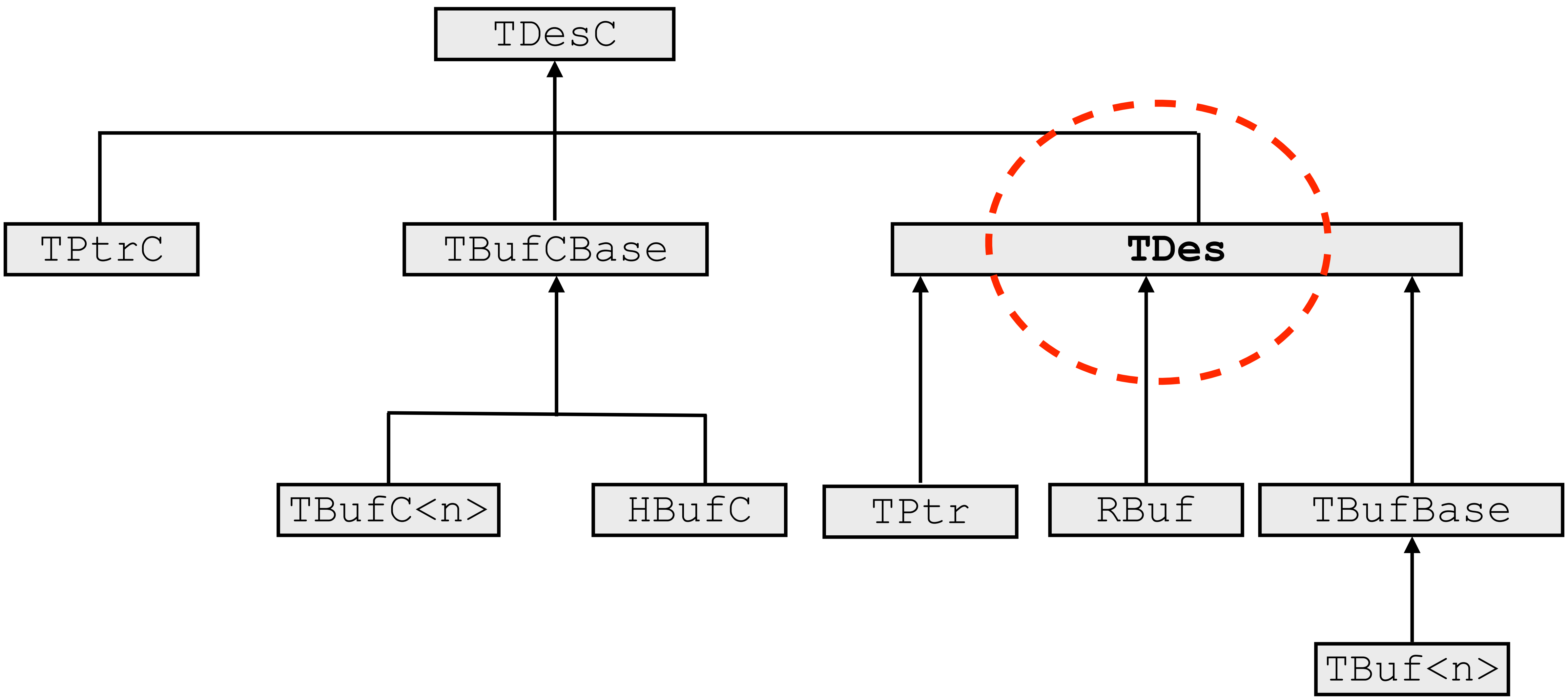
- All inherit these methods but they are never overridden since they are equally valid for all types of descriptor.
- In consequence all constant descriptor manipulation is implemented by **`TDesC`** regardless of the type of the descriptor
  - A good example of code reuse

# Symbian OS Descriptor Class: `TDesC`

## Access to the descriptor data

- Is different depending on the implementation of the derived descriptor classes (buffer or pointer)
- When a descriptor operation needs the correct address in memory for the beginning of the descriptor data
- It uses the `Ptr()` method of `TDesC` which looks up the type (the top 4 bits) of descriptor
- And returns the correct address for the beginning of the data

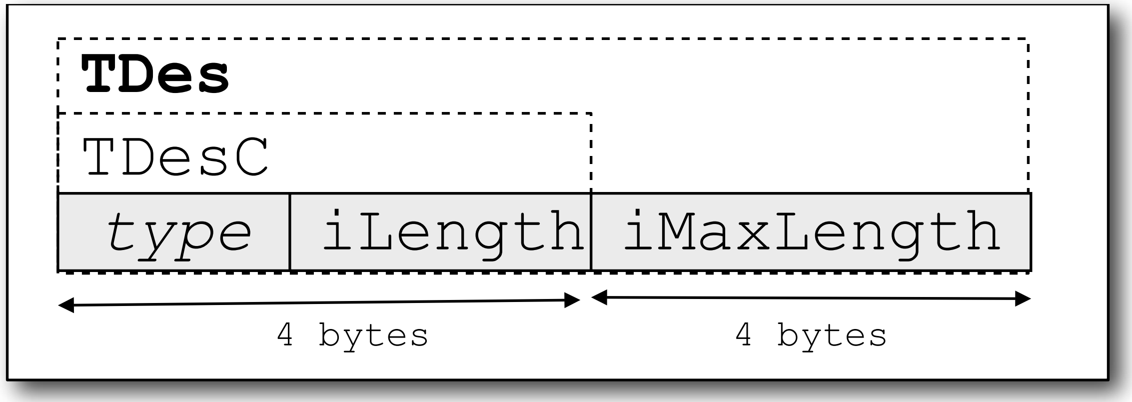
# Symbian OS Descriptor Class: **TDes**



# Symbian OS Descriptor Class: **TDes**

The modifiable descriptor types all derive from the base class TDes

- A subclass of **TDesC**
- **TDes** has an additional member variable to store the maximum length of data allowed for the current memory allocated to the descriptor
- The **MaxLength()** method of **TDes** returns this value - it is not overridden by derived classes




**TDes** defines a range of methods to manipulate modifiable string data

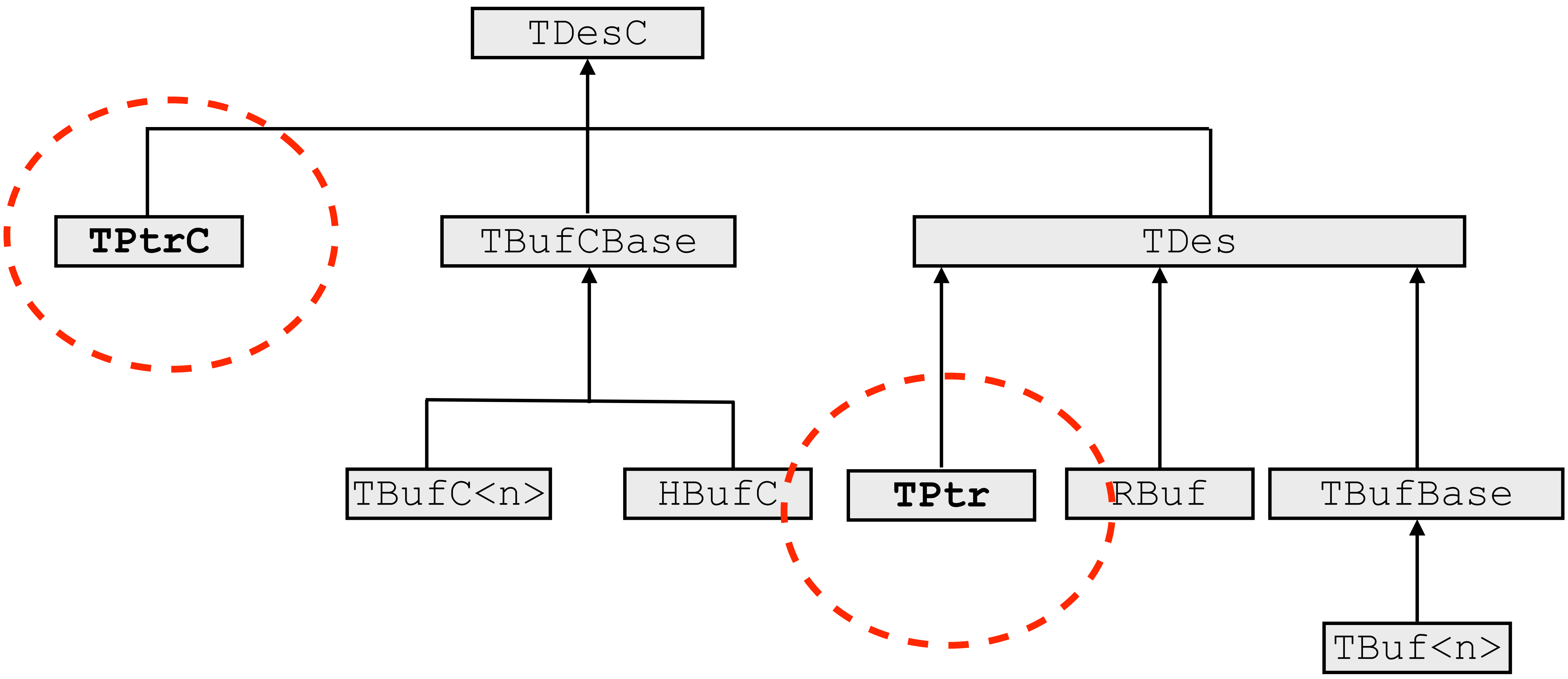
- Including appending, filling and formatting the descriptor data
- All the manipulation code is implemented by **TDes** and inherited by the derived classes.

# Derived Descriptor Classes

Descriptors come in two basic layouts:

- Pointer descriptors in which the descriptor holds a pointer to the location of a character string stored elsewhere
- Buffer descriptors where a string of characters forms part of the descriptor.

# Pointer Descriptors: **TPtrC** and **TPtr**



# Pointer Descriptors: `TPtrC` and `TPtr`

## The string data of a pointer descriptor

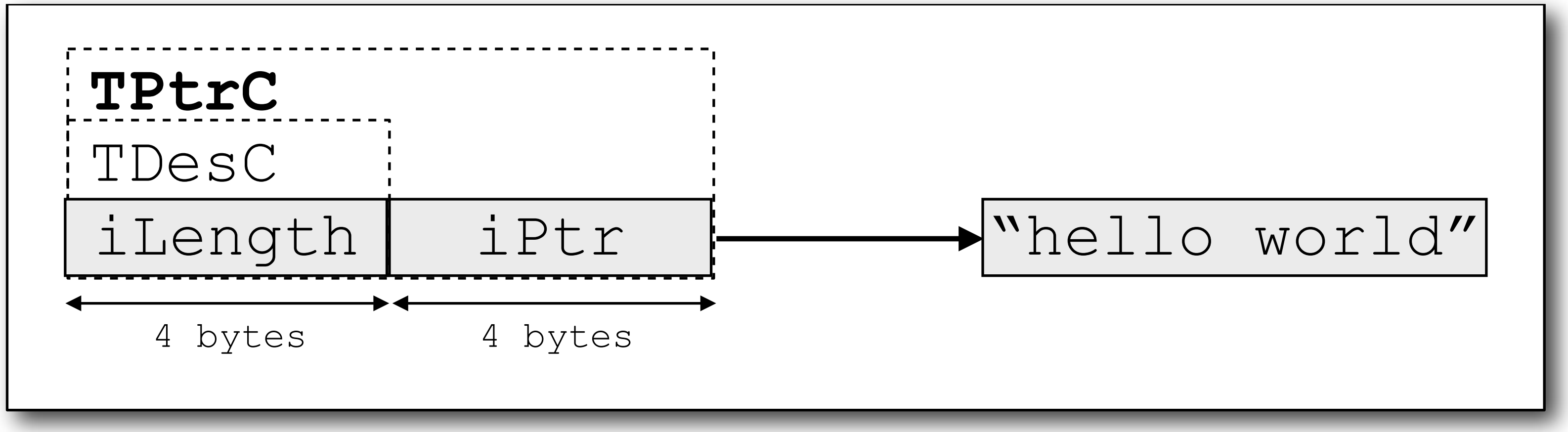
- Is separate from the descriptor object itself
- Can be stored in ROM, on the heap or on the stack
- The memory that holds the data is not "owned" by the descriptor

## Pointer descriptors

- Are agnostic about where the memory they point to is actually stored
- The pointer descriptors themselves are usually stack-based
- But they can be used on the heap for example as a member variable of a `CBase`-derived class

# Pointer Descriptors: **`TPtrC`** and **`TPtr`**

The non-modifiable pointer descriptor **`TPtrC`**




- The pointer to the data follows the length word thus the total size of the descriptor object is two words (8 bytes)
- **`TPtrC`** is the equivalent of using **`const char*`** when handling strings in C
- The data can be accessed but not modified i.e. the data in the descriptor is constant
- All the non-modifying operations defined in the **`TDesC`** base class are accessible to objects of type **`TPtrC`**

# Pointer Descriptors: `TPtrC` and `TPtr`

The class also defines a range of constructors to allow `TPtrC` to be constructed from another:

- Descriptor
- A pointer into memory
- A zero-terminated C string

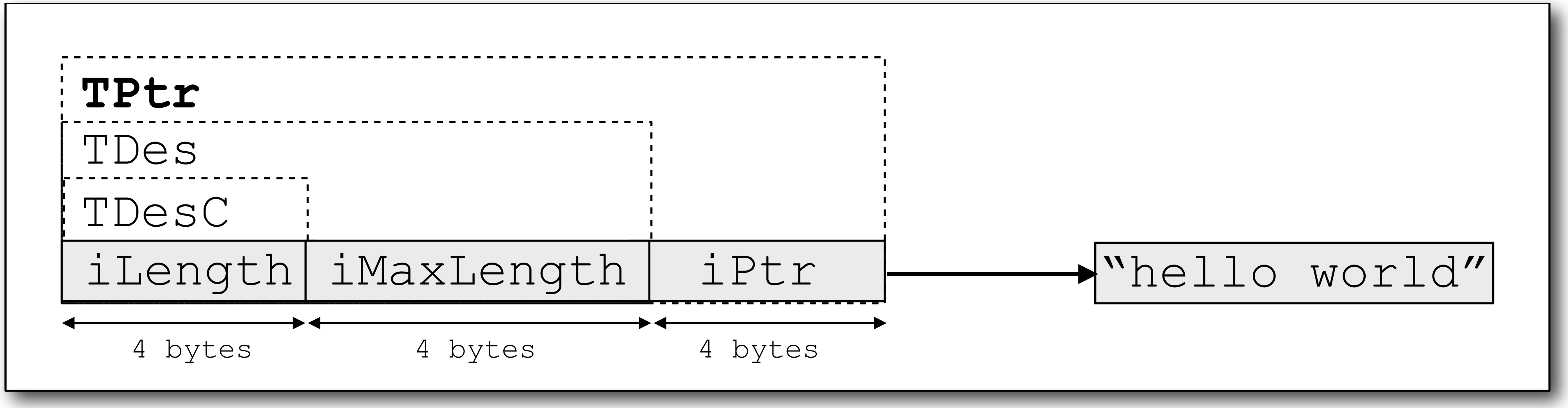
# Pointer Descriptors: **TPtrC** and **TPtr**

## Examples of **TPtrC** constructors

| | |
|---|---|
| | `_LIT(KDes, "abc ....");` |
| Constructed from a literal descriptor | `TPtrC ptr(KDes);` |
| Copy constructed from another **TPtrC** | `TPtrC copyPtr(ptr);` |
| Constant buffer descriptor | `TBufC<100> constBuffer(KDes);` |
| Constructed from a **TBufC** | `TPtrC myPtr(constBuffer);` |
| **TText8** is a single (8-bit) character, equivalent to **unsigned char** | `const TText8* cString = (TText8*)"abc ...";` |
| Constructed from a zero-terminated C string | `TPtrC8 anotherPtr(cString);` |
| Pointer into memory initialized elsewhere | `TUint8* memoryLocation;` |
| Length of memory to be represented | `TInt length;` |
| | `...` |
| | `TPtrC8 memPtr(memoryLocation,length);` |

# Pointer Descriptors: **`TPtrC`** and **`TPtr`**

## A modifiable pointer descriptor **`TPtr`**




- The data location pointer (4 bytes) follows the maximum length of **`TDes`** (4 bytes), which itself follows the length word of **`TDesC`** (4 bytes)
- The descriptor object is three words in total (12 bytes)
- The **`TPtr`** class can be used for access to, and modification of, a character string or binary data
- All the modifiable and non-modifiable base-class operations of **`TDes`** and **`TDesC`** may be performed on a **`TPtr`**

# Pointer Descriptors: `TPtrC` and `TPtr`

## The class defines constructors

- To allow objects of type `TPtr` to be constructed from pointer into an address in memory
- Setting the length and maximum length as appropriate

## The compiler

- Also generates implicit default and copy constructors
- Since they are not explicitly declared `protected` or `private` in the class

## A `TPtr` object

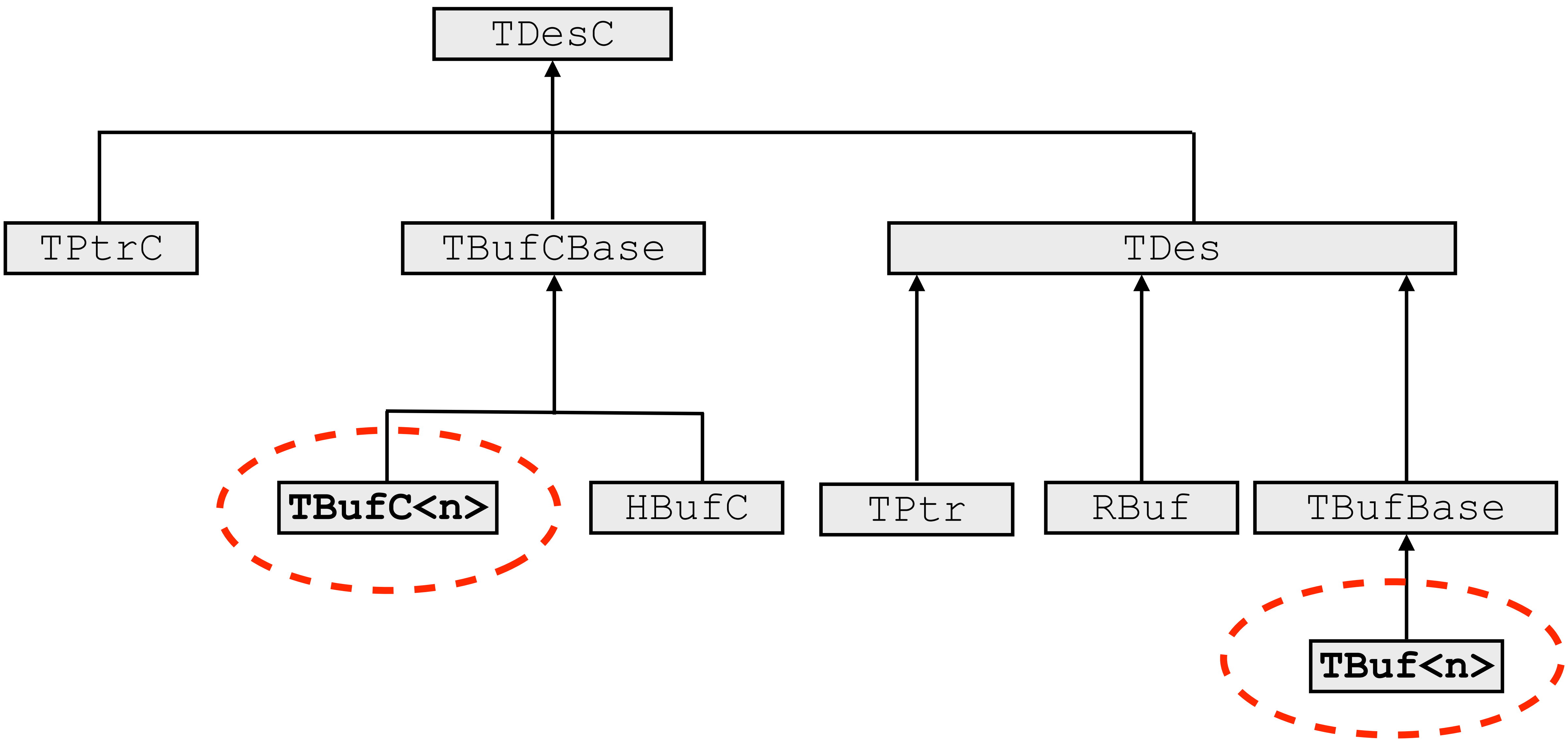
- May be copy-constructed from another modifiable pointer descriptor
- For example by calling the `Des()` method on a non-modifiable buffer

# Pointer Descriptors: **TPtrC** and **TPtr**

## Examples of **TPtr** constructors

| | |
|---|---|
| | `_LIT(KLiteralDes1, "abc ..");` |
| **TBufC** is described later | `TBufC<60> buf(KLiteralDes1);` |
| Copy construction – can modify the data in **buf** | `TPtr ptr(buf.Des());` |
| **Length=37** characters | `TInt length = ptr.Length();` |
| Maximum **length=60** chars, as for **buf** | `TInt maxLength = ptr.MaxLength();` |
| Valid pointer into memory | `TUint8* memoryLocation;` |
| ... | `...` |
| Length of data to be represented | `TInt len = 12;` |
| Maximum length to be represented | `TInt maxLen = 32;` |
| Construct a pointer descriptor from a pointer into memory **length=0, max=32** | `TPtr8 memPtr(memoryLocation, maxLen);` |
| **length=12, max=32** | `TPtr8 memPtr2(memoryLocation, len, maxLen);` |

# Stack-Based Buffer Descriptors **`TBufC`** and **`TBuf`**



# Stack-Based Buffer Descriptors `TBufC` and `TBuf`

The stack-based buffer descriptors may be modifiable or non-modifiable

- The string data forms part of the descriptor object
  - Located after the length word in a non-modifiable descriptor
  - Located after the maximum-length word in a modifiable buffer descriptor

These descriptors are useful for fixed-size relatively small strings (i.e. stack-based)

- They may be considered equivalent to `char[]` in C
- But with the benefit of overflow checking

# Stack-Based Buffer Descriptors **TBufC** and **TBuf**

**TBufC** is the non-modifiable buffer class


- Used to hold constant string or binary data.
- The class derives from **TBufCBase** (which derives from **TDesC** and exists as an inheritance convenience rather than to be used directly)
- **TBufC<n>** is a thin template class which uses an integer value to determine the size of the data area allocated for the buffer descriptor object

# Stack-Based Buffer Descriptors `TBufC` and `TBuf`

## `TBufC` has several constructors

- One to allow non-modifiable buffers to be constructed from a copy of any other descriptor or from a zero-terminated string
- `TBufC` can also be created empty and filled later

## As the data is non-modifiable

- The entire contents of the buffer is replaced by calling the assignment operator defined by the class
- The replacement data may be another non-modifiable descriptor or a zero-terminated string

## In each case

- The new data length must not exceed the length specified in the template parameter when the buffer was created.

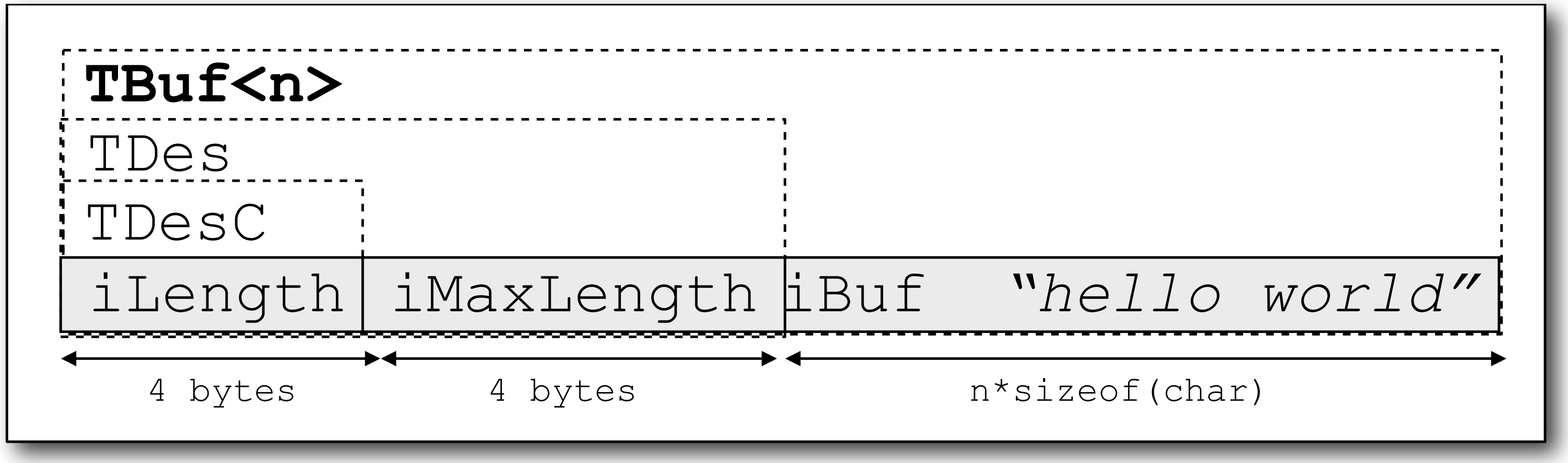
# Stack-Based Buffer Descriptors **TBufC** and **TBuf**

## Examples of **TBufC** constructors

| | |
|---|---|
| | `_LIT(KPalindrome, "aabbaa");` |
| Constructed from literal descriptor | `TBufC<50> buf1(KPalindrome);` |
| Constructed from **buf1** | `TBufC<50> buf2(buf1);` |
| | `...` |
| Constructed from a NULL-terminated C string | `TBufC<30> buf3((TText16*)"Never odd or even");` |
| Constructed empty **length = 0** | `TBufC<50> buf4;` |
| | `...` |
| Copy and replace **buf4** contains data copied from **buf1**, length modified | `buf4 = buf1;` |
| **buf1** contains data copied from **buf3**, length modified | `buf1 = buf3;` |
| Panic! Max length of **buf3** is insufficient for **buf2** data | `buf3 = buf2;` |

# Stack-Based Buffer Descriptors **TBufC** and **TBuf**

The **TBuf<n>** class for modifiable buffer data is a thin template class




- The integer value determining the maximum allowed length of the buffer
- It derives from **TBufBase** which itself derives from **TDes**
- Inherits the full range of non-modifiable and modifiable descriptor operations in **TDes** and **TDesC**

# Stack-Based Buffer Descriptors **TBufC** and **TBuf**

**TBuf<n>** defines a number of constructors and assignment operators

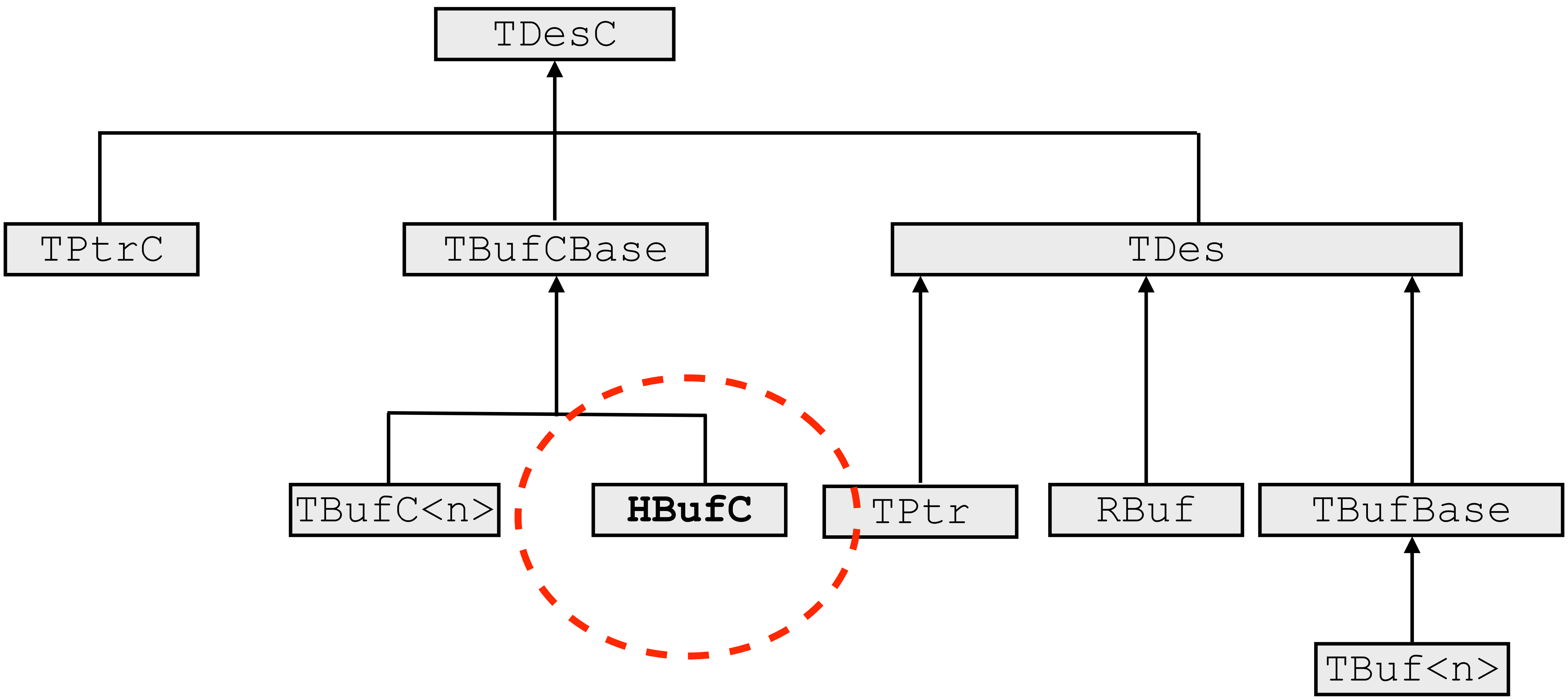
- Similar to those offered by its non-modifiable counterpart **TBufC<n>**

| | |
|---|---|
| | `_LIT(KPalindrome, "aabbaa");` |
| Constructed from literal descriptor | `TBuf<40> buf1(KPalindrome);` |
| Constructed from constant buffer descriptor | `TBuf<40> buf2(buf1); //` |
| Constructed from a NULL-terminated C string | `TBuf8<40> buf3((TText8*)"Do Geese see God?");` |
| Constructed empty **length = 0 maximum length = 40** | `TBuf<40> buf4;` |
| Copy and replace | `...` |
| **buf2** copied into **buf4**, updating length and max length | `buf4 = buf2;` |
| updated from C string | `buf3 = (TText8*)"Murder for a jar of red rum";` |



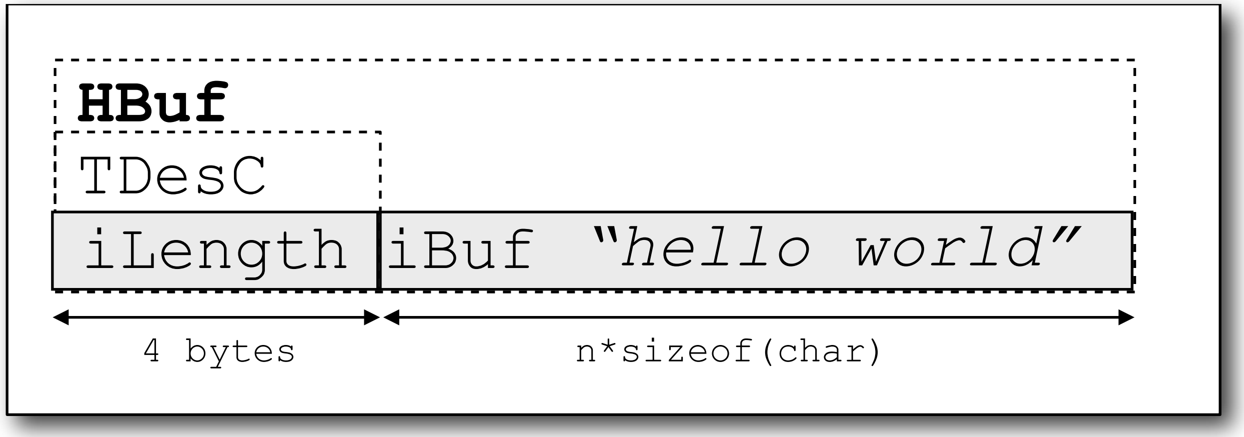
symbian Academy

# Dynamic Descriptors: **`HBufC`**



# Dynamic Descriptors: **`HBufC`**

The class layout is similar to **`TBufC`**




Heap-based descriptors can be used

- for string data that cannot be placed on the stack because it is too big or its size is not known at compile time
- where **`malloc`**'d data would be used in C

# Dynamic Descriptors: `HBufC`

## The `HBufC8` and `HBufC16` classes

- Export a number of `static NewL()` factory functions to create the descriptor on the heap
- These methods follow the two-phase construction model and may leave if there is insufficient memory available

## There are no public constructors

- All heap buffers must be constructed using one of these methods

## `TDesC::Alloc()` or `TDesC::AllocL()`

- May be used - these spawn an `HBufC` copy of any existing descriptor

# Dynamic Descriptors: `HBufC`

## These descriptors are not modifiable

- In common with the stack-based non-modifiable buffer descriptors the class provides a set of assignment operators
- These replace the entire contents of the buffer
- Objects of the class can also be modified at run-time by creating a modifiable pointer descriptor `TPtr` using the `Des()` method

## Heap descriptors

- Can be created dynamically to the size required
- But they are not automatically resized
- The buffer must have sufficient memory available for the modification operation to succeed or a panic will occur

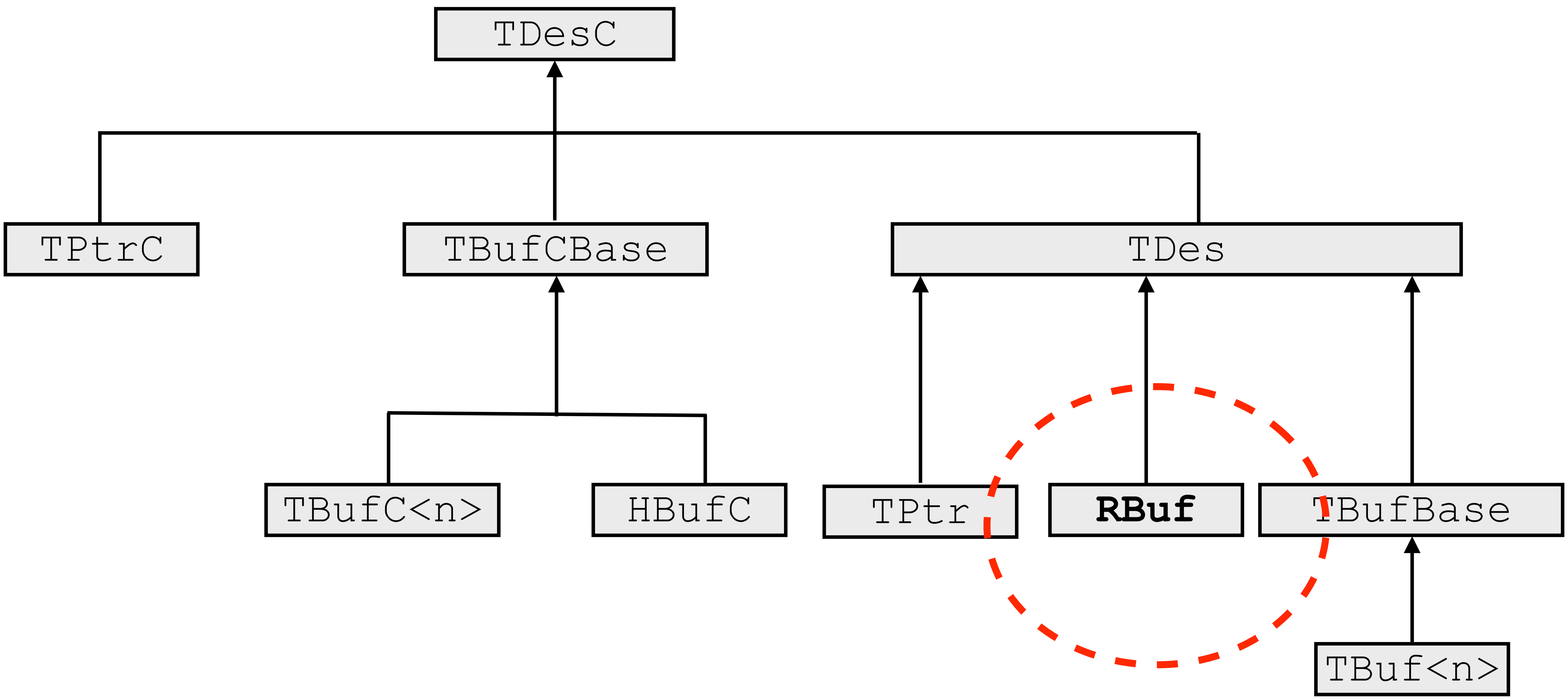
# Dynamic Descriptors: **HBufC**

## **HBufC** Constructor example

- Pay close attention to the modifiable **ptr** from **heapBuf->Des()**

| | |
|---|---|
| | `_LIT(KPalindrome, "Do Geese see God?");` |
| | `TBufC<20> stackBuf(KPalindrome);` |
| Allocates an empty heap descriptor of max length | `...` |
| 20 | `HBufC* heapBuf = HBufC::NewLC(20);` |
| | `TInt length = heapBuf->Length();` |
| Current length = 0 | `TPtr ptr(heapBuf->Des());` |
| Modification of the heap descriptor | `ptr = stackBuf;` |
| Copies **stackBuf** contents into **heapBuf** | `length = heapBuf->Length();` |
| **length** = 17 | `HBufC* heapBuf2 = stackBuf.AllocLC();` |
| From stack buffer | `length = heapBuf2->Length();` |
| **length** = 17 | `_LIT(KPalindrome2, "Palindrome");` |
| | `*heapBuf2 = KPalindrome2;` |
| Copy and replace data in **heapBuf2** | `length = heapBuf2->Length();` |
| **length** = 10 | `CleanupStack::PopAndDestroy(2, heapBuf);` |

# Dynamic Descriptors: **`RBuf`**



# Dynamic Descriptors: `RBuf`

## Class `RBuf` behaves like `HBufC`

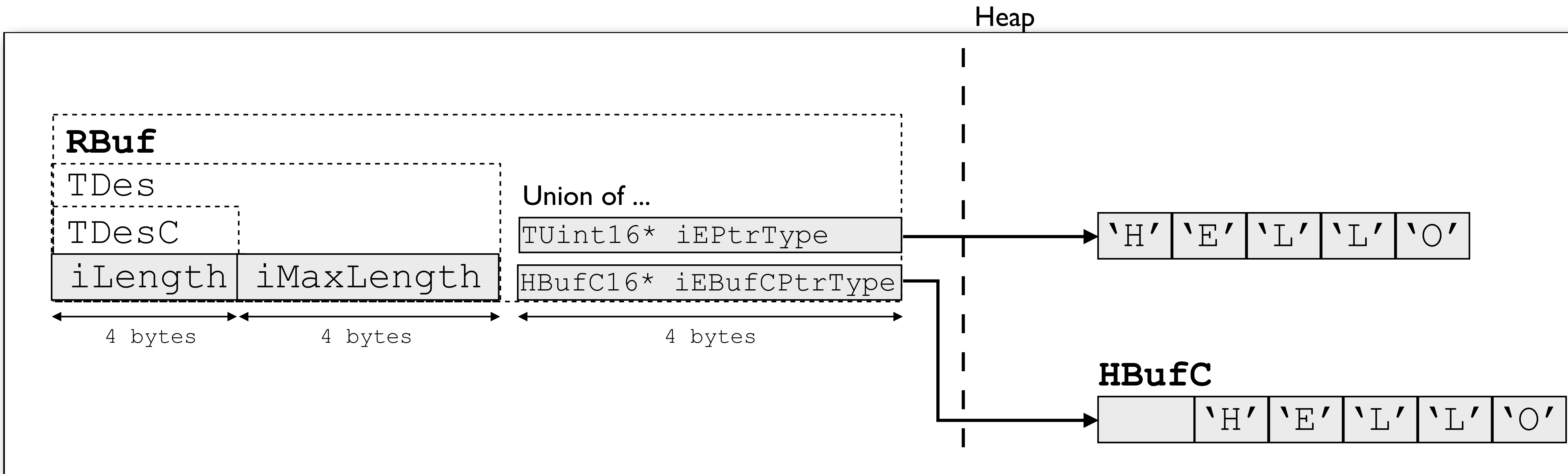
- The maximum length required can be specified dynamically
- On instantiation, an `RBuf` object can allocate its own buffer, take ownership of pre-allocated memory or a pre-existing heap descriptor
- `RBuf` descriptors are typically created on the stack
- But maintain a pointer to memory on the heap

## `RBuf` is derived from `TDes`

- An `RBuf` object can easily be modified and can be passed to any function where a `TDesC` or `TDes` parameter is specified
- There is no need to create a `TPtr` around the data in order to modify it
- This makes it preferable to `HBufC` when dynamically allocating a descriptor which is later modified.

# Dynamic Descriptors: **`RBuf`**

Internally **`RBuf`** behaves in one of two ways:




- As a **`TPtr`** descriptor type which points to a buffer containing only descriptor data

  The **`RBuf`** object allocates or takes ownership of memory existing elsewhere

- As a pointer to a heap descriptor **`HBufC*`**

  The **`RBuf`** object takes ownership of an existing heap descriptor thus the object pointed to contains a complete descriptor object

# Dynamic Descriptors: `RBuf`

## The handling of the internal union representation is transparent

- There is no need to know how a specific **`RBuf`** object is represented internally
- The descriptor operations correspond to the usual base-class methods of **`TDes`** and **`TDesC`**

## The class is not named `HBuf`

- As the objects are not directly created on the heap

## It is an R class

- As it manages a heap-based resource and is responsible for freeing the memory at cleanup time

# Descriptors: Part One

- ✓ Features of Symbian OS Descriptors
- ✓ The Symbian OS Descriptor Classes

# Descriptors

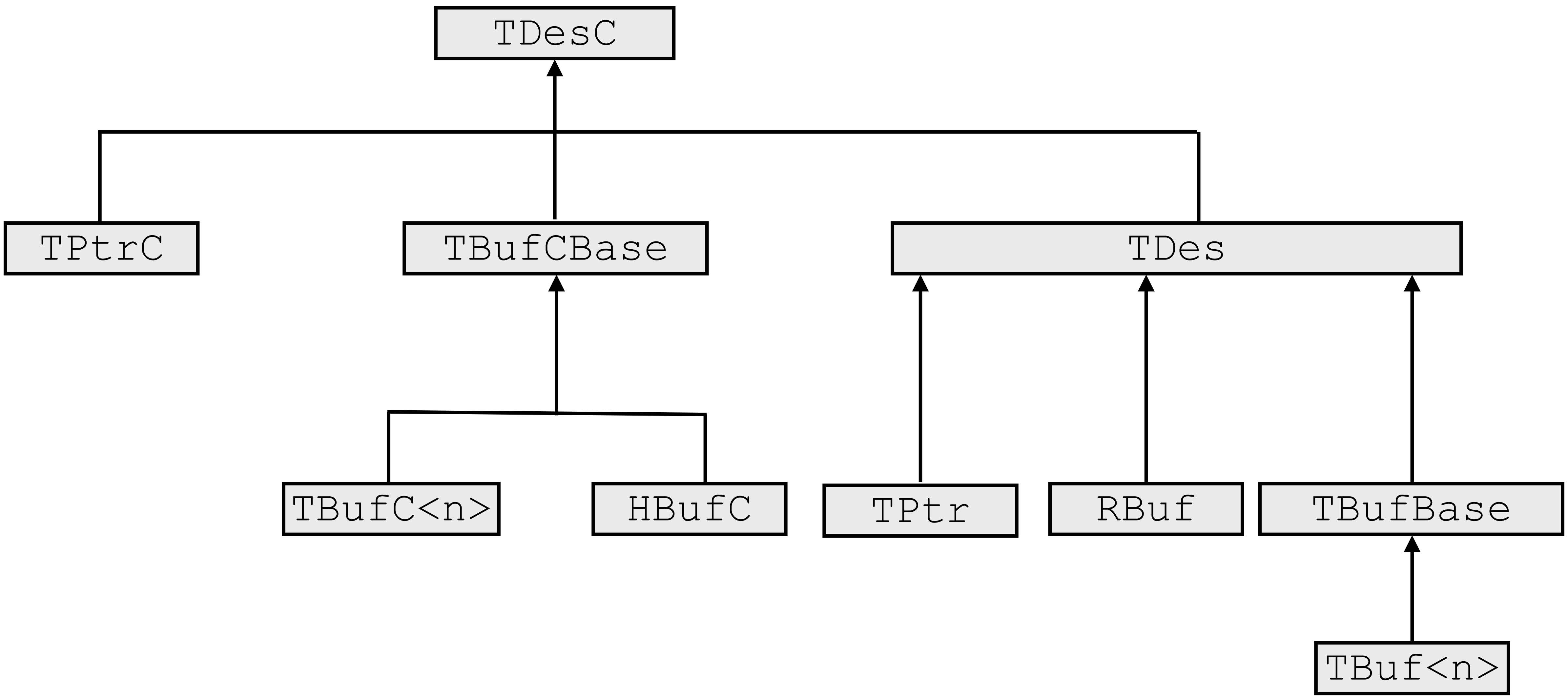
## Part Two

# Descriptors

## The Inheritance Hierarchy of the Descriptor Classes

- Know the inheritance hierarchy of the descriptor classes
- Understand the memory efficiency of the descriptor class inheritance model and its implications

# The Inheritance Hierarchy of the Descriptor Classes



# The Inheritance Hierarchy of the Descriptor Classes

## The `TDesC` and `TDes` base classes

- Provide the descriptor manipulation methods
- Must know the type of derived class they are operating on in order to correctly locate the data area

## Each subclass

- Does not implement its own data access method using virtual function overriding
  - This would add an extra 4 bytes to each derived descriptor object for a virtual pointer (vptr) to access the virtual function table

# The Inheritance Hierarchy of the Descriptor Classes

Descriptors were designed to be as efficient as possible

- The size overhead to accommodate a C++ vptr was considered undesirable

To allow for the specialization of derived classes

- The top 4 bits of the 4 bytes that store the length of the descriptor object are used to indicate the class type of the descriptor

# The Inheritance Hierarchy of the Descriptor Classes

There are currently six derived descriptor classes: **`TPtrC`**, **`TPtr`**, **`TBufC`**, **`TBuf`**, **`HBufC`** and **`RBuf`**

- Each sets the identifying **`iType`** bits as appropriate upon construction
- The use of 4 bits to identify the type limits the number of different types of descriptor to 2^4 (= 16)
- It seems unlikely that the range will need to be extended significantly in the future

# The Inheritance Hierarchy of the Descriptor Classes

For all descriptors, access to the descriptor data goes through the non-virtual `Ptr()` method of the base class `TDesC`

- It uses a `switch` statement to check the 4 bits, identify the type of descriptor and return the correct address for the beginning of its data.
- This requires that the `TDesC` base class has knowledge of the memory layout of its subclasses hard-coded into `Ptr()`

# Descriptors

## Using the Descriptor APIs

- Understand that the descriptor base classes `TDesC` and `TDes` cannot be instantiated
- Understand the difference between `Size()`, `Length()` and `MaxLength()` descriptor methods
- Understand the difference between `Copy()` and `Set()` descriptor methods and how to use assignment correctly

# Using the Descriptor APIs

## The `TDesC` and `TDes` descriptor base classes

- Provide and implement the APIs for all descriptor operations
- Typically the derived descriptors only implement specific methods for construction and copy assignment.

## Objects of type `TDesC` and `TDes`

- Cannot be instantiated directly because their default constructors are protected.
- It is the derived descriptor types that are actually instantiated and used

## The descriptor API methods

- Are fully documented in the Symbian OS Library of each SDK and the course examples
- This lecture focuses on some of the trickier areas of descriptor manipulation

# Using the Descriptor APIs

## The difference between `Length()` and `Size()`

- The `Size()` method returns the size of the descriptor in bytes
- The `Length()` method returns the number of characters it contains
- For 8-bit descriptors this is the same as the size i.e. the size of a character is a byte

## From Symbian OS v5u (version 5 unicode)

- The native character is 16 bits wide i.e. each character occupies two bytes
- `Size()` always returns a value double that of `Length()` for neutral and explicitly wide descriptors

# `MaxLength()` and Length Modification Methods

The **`MaxLength()`** method

- Of **`TDes`** returns the maximum length of a modifiable descriptor value
- Like the **`Length()`** method of **`TDesC`** it is not overridden by the derived classes

The **`SetMax()`** method

- Sets the current length of the descriptor to the maximum length allowed
- It does not change the maximum length of the descriptor thereby expanding or contracting the data area

# **`MaxLength()`** and Length Modification Methods

### The **`SetLength()`** method

- Can be used to adjust the descriptor length to any value between zero and its maximum length

### The **`Zero()`** method

- Sets the length to zero

# `Set()` and the Assignment Operator

The pointer descriptor types

- Provide **`Set()`** methods which update them to point at different string data

```
_LIT(KDes1, "Sixty zippers were quickly picked from the woven jute bag");
_LIT(KDes2, "Waltz, bad nymph, for quick jigs vex");
TPtrC alpha(KDes1);
TPtrC beta(KDes2);
alpha.Set(KDes2); // alpha points to the data in KDes2
beta.Set(KDes1);  // beta points to the data in KDes1
```

Literal descriptors are described later

# `Set()` and the Assignment Operator

**`TDes`** provides an assignment operator **`TDes::operator =()`**

- Used copy data into the memory referenced by any modifiable descriptor
- Provided the length of the data to be copied does not exceed the maximum length of the descriptor which would cause a panic

It is easy to confuse **`Set()`** with **`TDes::operator =()`**

- **`Set()`** resets a pointer descriptor to point at a new data area
  - Changes the length and maximum length members
- **`TDes::operator =()`** merely copies data into an existing descriptor
  - Modifies the descriptor length but <u>not its maximum length</u>

# `Set()` and `operator =()`

Points to the contents of **`buf`**

Valid pointer into memory

...

Maximum length to be represented

max length=40

Copy and replace

**`memPtr`** data is **`KLiteralDes1`** (37 bytes), max length=40

Points to the data in **`buf2`**

Replace what **`ptr`** points to

**`ptr`** points to contents of **`buf2`**, max length = 100

Attempt to update **`memPtr`**, which panics because the contents of **`ptr2`** (43 bytes) exceeds max length of **`memPtr`** (40 bytes)

```
_LIT(KLiteralDes1, "Jackdaws love ...");
TBufC<60> buf(KLiteralDes1);
TPtr ptr(buf.Des());
TUint16* memoryLocation;
...
TInt maxLen = 40;
TPtr memPtr(memoryLocation, maxLen);

memPtr = ptr; // Assignment
_LIT(KLiteralDes2, "The quick brown fox ...");
TBufC<100> buf2(KLiteralDes2);
TPtr ptr2(buf2.Des());

ptr.Set(ptr2);
memPtr = ptr2;
```

# Modifying Data with `Des()`

The content of a non-modifiable buffer descriptor cannot be altered directly other than by complete replacement of the data

- The stack- and heap-based **`TBufC`** and **`HBufC`** descriptors provide a method which returns a modifiable pointer descriptor to the data represented by the buffer

But it is possible to change the data indirectly

- by calling **`Des()`** and then operating on the data via the pointer

When the data is modified via the return value of **`Des()`**

- The length members of both the pointer descriptor and the constant buffer descriptor are updated

# Modifying Data with **Des()**

Note: All descriptors are 8 bit

Constructed from literal descriptor

Data is the string in **buf**, max length = 40

Use **ptr** to replace contents of **buf**

Panic!

**KPal2** exceeds max length of **buf** (=40)

```
_LIT8(KPalindrome, "Satan, oscillate...");
TBufC8<40> buf(KPalindrome);
TPtr8 ptr(buf.Des());
...
ptr = (TText8*)"Do Geese see God?";
ASSERT(ptr.Length()==buf.Length());
_LIT8(KPal2, "Are we not drawn onward...");
ptr = KPal2; // Assignment
```

# Descriptors

## Descriptors as Function Parameters

- Understand that the correct way to specify a descriptor as a function parameter is to use a reference, for both constant data and data that may be modified by the function in question.

# Descriptors as Function Parameters

## The `TDesC` and `TDes` descriptor base classes

- Provide and implement the APIs for all descriptor operations
- They can be used as function arguments and return types, allowing descriptors to be passed around in code without forcing a dependency on a particular type

## An API client

- Should not be constrained to using a `TBuf` because a particular function requires it

## Function providers

- Should remain agnostic to the type of descriptor passed to them

## Unless a function takes or returns ownership

- It should not need to specify whether a descriptor is stack- or heap-based

# Descriptors as Function Parameters

When defining functions

- The `TDesC` and `TDes` base classes should always be used as parameters or return values
- For efficiency, descriptor parameters should be passed by reference
- Either as `const TDesC&` for constant descriptors or `TDes&` when modifiable

Except: When returning ownership of a heap-based descriptor as a return value

- It should be specified explicitly so that the caller can clean it up appropriately and avoid a memory leak

# Descriptors

## Correct Use of the Dynamic Descriptor Classes

- Identify the correct techniques and methods to instantiate an HBufC heap buffer object
- Recognize and demonstrate knowledge of how to use the new descriptor class RBuf

# Correct Use of the Dynamic Descriptor Classes

## **HBufC** construction and usage

- **HBufC** can be spawned from existing descriptors using the **Alloc()** or **AllocL()** overloads implemented by **TDesC**
- This example which shows how to replace inefficient code with use of **TDesC::AllocL()**

```
void CSampleClass::UnnecessaryCodeL(const TDesC& aDes)
    {
    iHeapBuffer = HBufC::NewL(aDes.Length());
    TPtr ptr(iHeapBuffer->Des());
    ptr.Copy(aDes);
    ...
    }
```

Could be replaced by a single line

```
    {
    iHeapBuffer = aDes.AllocL();
    }
```

# HBufC Construction and Usage

## An HBufC object

- Can also be instantiated using the **static NewL()** factory methods specified for the class
- For **HBufC16** the methods available are as follows:

```
static IMPORT_C HBufC16* NewL(TInt aMaxLength);

static IMPORT_C HBufC16* NewLC(TInt aMaxLength);
```

- These methods create a new heap-based buffer descriptor with maximum length as specified
- Leave if there is insufficient memory available for the allocation
- The **NewLC** method leaves the successfully created descriptor object on the cleanup stack
- The heap descriptor is empty and its length is set to zero

# **HBufC** Construction and Usage

## An **HBufC16** object

- Can also be instantiated using the non leaving method **static New()** factory:

```
static IMPORT_C HBufC16* New(TInt aMaxLength);
```

- This method creates a new heap-based buffer descriptor with maximum length as specified
- It does not leave if there is no heap memory available to allocate the descriptor
- The caller must compare the returned pointer against **NULL** to confirm that it has succeeded before dereferencing it
- The heap descriptor is empty and its length is set to zero

# **HBufC** Construction and Usage

**NewMax** methods also set **HBufC16**'s length to the maximum value

```
static IMPORT_C HBufC16* NewMax(TInt aMaxLength);

static IMPORT_C HBufC16* NewMaxL(TInt aMaxLength);

static IMPORT_C HBufC16* NewMaxLC(TInt aMaxLength);
```

- These methods create a new heap-based buffer descriptor with maximum length as specified and set its length to the maximum value
- No data is assigned to the descriptor

## If insufficient memory is available for the allocation:

- **NewMax()** return a **NULL** pointer ()
- **NewMaxL()** and **NewMaxLC()** will leave
- **NewMaxLC()** leaves the successfully allocated descriptor on the cleanup stack

# **HBufC** Construction and Usage

Initializing an **HBufC16** from the contains of a **RReadStream**

```
static IMPORT_C HBufC16* NewL(RReadStream& aStream, TInt aMaxLength);

static IMPORT_C HBufC16* NewLC(RReadStream& aStream, TInt aMaxLength);
```

- These methods allocate a heap-based buffer descriptor and initialize it from the contents of a read stream
- Reads from the stream up to the maximum length specified (or the maximum length of the stream data - whichever is shortest) and allocates a buffer to hold the contents
- Typically used to reconstruct a descriptor that has previously been externalized to a write stream using the stream operators

# Descriptor Externalization and Internalization

An example of descriptor externalization and internalization using streams

Writes the contents of **iHeapBuffer** to a writable stream

Write the descriptor's length
Write the descriptor's data

```
void CSampleClass::ExternalizeL(RWriteStream& aStream) const
    {
    aStream.WriteUint32L(iHeapBuffer->Length());
    aStream.WriteL(*iHeapBuffer, iHeapBuffer->Length());
    }
/**  */
void CSomeClass::InternalizeL(RReadStream& aStream)
    {

    TInt size = aStream.ReadUint32L();
    iHeapBuffer = HBufC::NewL(size);

    TPtr ptr(iHeapBuffer->Des());

    aStream.ReadL(ptr,size);

    }
```

Instantiates **iHeapBuffer** by reading the contents of the stream

Read the descriptor's length
Allocate **iHeapBuffer**

Create a modifiable descriptor over **iHeapBuffer**
Read the descriptor data into **iHeapBuffer** via **ptr**

# Descriptor Externalization and Internalization

The following code

- Shows the use of the stream operators as a more efficient alternative
- The stream operators have been optimized to compress the descriptor's meta-data as much as possible for efficiency and space conservation

```
void CSampleClass::ExternalizeL(RWriteStream& aStream) const
    {// Much more efficient, no wasted storage space
    aStream << iHeapBuffer;
    }

void CSampleClass::InternalizeL(RReadStream& aStream)
    {// KMaxLength indicates the maximum length of
     // data to be read from the stream
    iHeapBuffer = HBufC::NewL(aStream, KMaxLength);
    }
```

# **RBuf**

While there is a non-modifiable heap descriptor class **`HBufC`**

- There is no corresponding modifiable **`HBuf`** class

  Which might have been expected in order to make heap buffers symmetrical with **`TBuf`** stack buffers.

The **`RBuf`** class was first introduced in Symbian OS v8.0

- But first documented in Symbian OS v8.1 and used most extensively in software designed for phones based on Symbian OS v9 and later

# **RBuf** Construction and Usage

**RBuf** objects can be instantiated using

- **Create()**, **CreateMax()** or **CreateL()** to specify the maximum length of descriptor data that can be stored
- It is also possible to instantiate an **RBuf** to store a copy of the contents of another descriptor:

```
RBuf myRBuf;
_LIT(KHelloRBuf, "Hello RBuf!");
myRBuf.Create(KHelloRBuf());
```

- **Create()** allocates a buffer for the **RBuf** to reference
- If the **RBuf** previously owned a buffer, **Create()** will <u>not clean it up</u> before assigning the new buffer reference

  Cleanup must be done explicitly by calling **Close()** first

An **RBuf** can alternatively take ownership of a pre-existing section of memory using the **Assign()** method

- **Assign()** will also orphan any data already owned

  (**Close()** should be called first to avoid memory leaks)

# RBuf Construction and Usage

Using **Assign()** to take ownership

Taking ownership of **HBufC**

Use and clean up

```
HBufC* myHBufC = HBufC::NewL(20);
RBuf myRBuf.Assign(myHBufC);
...
myRBuf.Close();
```

Taking ownership of pre-allocated heap memory

Use and clean up

```
TInt maxSizeOfData = 20;
RBuf myRBuf;
TUint16* pointer =
    static_cast<TUint16*>(User::AllocL(maxSizeOfData*2));

myRBuf.Assign(pointer, maxSizeOfData);
...
myRBuf.Close();
```

# **`RBuf`** Construction and Usage

The **`RBuf`** class

- Does not manage the size of the buffer and reallocate it if more memory is required
- If **`Append()`** is called on an **`RBuf`** object for which there is insufficient memory available - a <u>panic will occur</u>
- This should be clear from the fact that the base-class methods are non-leaving
- i.e. there is no scope for the reallocation to fail in the event of low memory

# RBuf Construction and Usage

The memory for **RBuf** operations must be managed by the programmer

- The **ReAllocL()** method can be used as follows:

**myRBuf** is the buffer to be resized e.g. for an Append() operation

Push onto cleanup stack for leave-safety

Extend to **newLength**

Remove from cleanup stack

```
myRBuf.CleanupClosePushL();
myRBuf.ReAllocL(newLength);
CleanupStack::Pop ();
```

# `RBuf` Construction and Usage

Using an **`RBuf`** is preferable to using an **`HBufC`** in that:

- If the **`ReAllocL()`** method is used on the **`HBufC`** and causes the heap cell to move any associated **`HBufC*`** and **`TPtr`** variables need to be updated
- This update is not required for **`RBuf`** objects as the pointer is maintained internally

# **`RBuf`** Construction and Usage

The class is not named **`HBuf`**

- Unlike **`HBufC`** objects of this type are not themselves directly created on the heap

It is instead an R class

- As it manages a heap-based resource and is responsible for freeing the memory at cleanup time
  - As is usual for other R classes cleanup is performed by calling **`Close()`**

If the **`RBuf`**

- Was pushed onto the cleanup stack by a call to **`CleanupClosePushL()`** use **`CleanupStack::PopAndDestroy()`**

# `RBuf` Construction and Usage

It is possible to create an **`RBuf`** from an existing **`HBufC`**

- The **`RBuf`** class is both modifiable and dynamically allocated and thus is an advantage over **`HBufC`**
- Being able to create an **`RBuf`** from an **`HBufC`** object is a good migration route from pre-Symbian OS v8.1

Previously

- It would have been necessary to instantiate an **`HBufC`** and then use a companion **`TPtr`** object - constructed by calling **`Des()`** on the heap descriptor

When to use **`RBuf`** or **`HBufC`**?

- **`RBuf`** is recommended for use when a dynamically allocated buffer is required to hold data that changes frequently
- **`HBufC`** should be preferred when a dynamically allocated descriptor is needed to hold data that rarely changes



symbian Academy

# HBufC vs RBuf

## Using **HBufC** for modifiable data

```
HBufC* socketName = NULL;
// KMaxNameLength is defined elsewhere

if(!socketName)
    {
    socketName = HBufC::NewL(KMaxNameLength);
    }
// Create writable 'companion' TPtr
TPtr socketNamePtr(socketName->Des());
message.ReadL(message.Ptr0(), socketNamePtr);
```

## Using **RBuf** for modifiable data

```
RBuf socketName;
...

if(socketName.Compare(KNullDesC)==0)
    {
    socketName.CreateL(KMaxNameLength);
    }



message.ReadL(message.Ptr0(), socketName);
```

The code is simpler

- It is easier to understand and maintain

# Descriptors

## Common Inefficiencies in Descriptor Usage

- Know that **`TFileName`** objects should not be used indiscriminately, because of the stack space each consumes
- Understand when to dereference an **`HBufC`** object directly, and when to call **`Des()`** to obtain a modifiable descriptor (**`TDes&`**)

# Common Inefficiencies in Descriptor Usage

## **`TFileName`** objects waste stack space

- The **`TFileName`** type is **`typedef`**'d as a modifiable stack buffer with maximum length 256 characters
- It can be useful when calling various file system functions to parse filenames into complete paths - since the exact length of a filename is not always known at compile time
- For example to print out a directory's contents

## However, since each character is of 16-bit width

- Every time a **`TFileName`** object is declared on the stack it consumes 2 × 256 = 512 bytes
- Plus the 12 bytes required for the descriptor object itself
- That's just over 0.5 KB!
  - The default stack size for an application is only 8 KB

# `TFileName` Objects Waste Stack Space

On Symbian OS the stack space for each process is limited

- By default it is just 8 KB
- On the Windows emulator if more stack space is needed the stack will just expand
- This is not the case on a phone - a panic will be raised when the stack overflow occurs
- This can be hard to track down since <u>it will not be seen when testing on the emulator</u> so cannot be easily debugged

# **`TFileName`** Objects Waste Stack Space

At 0.5 KB a single **`TFileName`** object

- Can consume and potentially waste a significant proportion of the stack space

It is good practice to use one of the dynamic heap descriptor types

- Or limit the use of **`TFileName`** objects to members of C classes as these types are always created on the heap

# Referencing **HBufC** through **TDesC**

## The **HBufC** class derives from **TDesC**

- The **HBufC*** pointer can simply be dereferenced when a reference to a non-modifiable descriptor **TDesC&** is required

## A common mistake is to call the **Des()** method on the heap descriptor

- This creates a separate **TPtr** referencing the descriptor data
- This is not incorrect it returns a **TDes&**
- But it is clearer and more efficient simply to return the **HBufC** object directly

```
const TDesC& CSampleClass::AccidentalComplexity()
    {
    return (iHeapBuffer->Des());

    // could be replaced more efficiently with

    return (*iHeapBuffer);
    }
```

# Descriptors

## Literal Descriptors

- Know how to manipulate literal descriptors and know that those specified using `_L` are deprecated
- Specify the difference between literal descriptors using `_L` and those using `_LIT` and the disadvantages of using the former

# Literal Descriptors

Literal descriptors are different from the other descriptor types

- They are equivalent to `static char[]` in C and can be built into program binaries in ROM (if the code is part of the system) because they are constant
- There is a set of macros defined in `e32def.h` which can be used to define Symbian OS literals of two different types, `_LIT` and `_L`

# Literal Descriptors

## **_LIT** macro

- The **_LIT** macro is preferred for Symbian OS literals since it is the more efficient type
- It has been used in the sample code throughout these lectures - typically as follows:

```
_LIT(KFieldMarshalTait, "Field Marshal Tait");
```

- **KFieldMarshalTait** can then be used as a constant descriptor - for example to write to a file or display to a user

# The _LIT Macro

## The _LIT macro

- Builds a named object (**KFieldMarshalTait**) of type **TLitC16** into the program binary
- Storing the appropriate string in this case "**Field Marshal Tait**"
- The explicit macros **_LIT8** and **_LIT16** behave similarly except that **_LIT8** builds a narrow string of type **TLitC8**

# The _LIT Macro

**TLitC8** and **TLitC16** do not derive from **TDesC8** or **TDesC16**

- But they have the same binary layouts as **TBufC8** or **TBufC16**
- This allows objects of these types to be used wherever **TDesC** is used
- The string stored in the program binary has a **NULL** terminator because the native compiler string is used to build it
- The **_LIT** macro adjusts the length to the correct value for a non-terminated descriptor

# The _LIT Macro

Symbian OS also defines literals yo represent a blank string

- There are three variants of the null descriptor, defined as follows:
- Build independent:

```
_LIT(KNULLDesC,"");
```

- 8-bit for non-Unicode strings:

```
_LIT8(KNULLDesC8,"");
```

- 16-bit for Unicode strings:

```
_LIT16(KNULLDesC16,"");
```

# _L Macro

## Use of the _L macro

- Is now deprecated in production code
- It may still be used in test code (where memory use is less critical)
- The advantage of using **`_L`** is that it can be used in place of a **`TPtrC`** without having to declare it separately from where it is used:

```
User::Panic(_L("telephony.dll"), KErrNotSupported);
```

- For the example above, the string ("**`telephony.dll`**") is built into the program binary as a basic, **`NULL`**-terminated string

# _L Macro

## Unlike the TLitC built for the _LIT macro

- It has no initial length member
- This means the layout of the stored literal is not like that of a descriptor i.e. no length word

## Thus when the code executes

- Each instance of _L will result in construction of a temporary TPtrC
- With the pointer being set to the address of the first byte of the literal as it is stored in ROM

# _LIT and _L Memory Layouts

## The difference memory layouts

- In ROM for literals created using **_LIT** and **_L** macros

`_LIT(KHello,"Hello World!")`

ROM

| 12 | Hello World!\0 |
|---|---|

`TPtrC KHello(_L("Hello World!"))`

Stack Temporary → ROM

| iLength 12 | iPtr | → | Hello World!\0 |
|---|---|---|---|



symbian Academy

# Descriptors

## Descriptor Conversion

- Know how to convert 8-bit descriptors into 16-bit descriptors and vice versa using the descriptor `Copy()` method or the `CnvUtfConverter` class
- Recognize how to read data from file into an 8-bit descriptor and then 'translate' the data to 16-bit without padding, and vice versa
- Know how to use the `TLex` class to convert a descriptor to a number, and `TDes::Num()` to convert a number to a descriptor

# Descriptor Conversion

## Conversion between narrow and wide descriptors

- **`TDes`** implements an overloaded set of **`Copy()`** methods which allow copying directly into descriptor data from:
  - Another descriptor
  - A NULL-terminated string
  - A pointer

## The **`Copy()`** methods

- Copy the data into the descriptor setting its length accordingly
- The methods will panic if the maximum length of the receiving descriptor is shorter than the incoming data

# Descriptor Conversion

## The `Copy()` method

- Is overloaded to take either an 8- or 16-bit descriptor

## It is possible

- To copy a narrow-width descriptor onto a narrow-width descriptor
- To copy a wide descriptor onto a wide descriptor

## It is also possible

- To copy between descriptor widths
- i.e. carrying out an implicit conversion in the process

# Wide to Narrow

## The `Copy()` method implemented by `TDes8`

- Is to copy an incoming wide descriptor into a narrow descriptor
- It strips out alternate characters assuming them to be zeroes - the data values do not exceed 255 (decimal).
- The `Copy()` method which copies a narrow descriptor into the data area is a straight data copy

| C | A | T |
|---|---|---|

| D | \0 | O | \0 | G | \0 |
|---|---|---|---|---|---|

NULL characters stripped out in wide-to-narrow descriptor copy

| D | O | G |
|---|---|---|

```
TBuf8<3> cat(_L8("CAT"));

TBuf16<3> dog(_L16("DOG"));

cat.Copy(dog);
```

# Narrow to Wide

## For class **`TDes16`**

- An incoming 16-bit descriptor can be copied directly onto the data area
- The **`Copy()`** method that takes an 8-bit descriptor pads each character with a trailing zero as part of the copy operation

| S | M | A | L | L |
|---|---|---|---|---|

| L | \0 | A | \0 | R | \0 | G | \0 | E | \0 |
|---|---|---|---|---|---|---|---|---|---|

NULL characters used to pad narrow-to-wide descriptor copy

| S | \0 | M | \0 | A | \0 | L | \0 | L | \0 |
|---|---|---|---|---|---|---|---|---|---|

```
TBuf8<5> small(_L8("SMALL"));

TBuf16<5> large(_L16("LARGE"));

large.Copy(small);
```

# `Copy()` methods

## The `Copy()` methods

- Form a rudimentary means of copying and converting between 8- and 16-bit descriptors
- When the character set is encoded by one byte per character
- And the last byte of each wide character is simply a NULL character padding

# The `CnvUtfConverter` Class

## Is used to perform a proper conversion

- In both directions between 16-bit Unicode (UCS-2) and 8-bit non-Unicode character sets
- Or between Unicode and the UTF-7 and UTF-8 transformation sets

## The `CnvUtfConverter` class

- Supplied by `charconv.lib` (see header file `utf.h`) is available
- The class supplies a set of `static` methods such as `ConvertFromUnicodeToUtf8()` and `ConvertToUnicodeFromUtf8()`

# Converting Descriptors to Numbers

A descriptor can be converted to a number using the **`TLex`** class

- This class provides general-purpose lexical analysis and performs syntactical element parsing and string-to-number conversion
- Using the locale-dependent functions of **`TChar`** to determine whether each character is a digit, a letter or a symbol

# Converting Descriptors to Numbers

**`TLex`** is the build-width neutral class

- Implicitly **`TLex16`**
- **`TLex8`** and **`TLex16`** can also be used explicitly
- The neutral form should be preferred unless a particular variant is required
- **`TLex`** can be constructed with the data for constructed empty and later assigned the data
- Both construction and assignment can take:
  - Another **`TLex`** object
  - A non-modifiable descriptor
  - A **`TUint16*`** or **`TUint8*`** (for **`TLex16`** or **`TLex8`** respectively) pointer to string data

# Converting Descriptors to Numbers

## At the very simplest level

- When the string contains just numerical data, the descriptor contents can be converted to an integer using the **`Val()`** function of **`TLex`**

```
_LIT(KTestLex, "54321");
TLex lex(KTestLex());
TInt value = 0;
TInt err = lex.Val(value); // value == 54321 if no error occurred
```

- The **`Val()`** function is overloaded for different signed integer types (**`TInt, TInt8, TInt16, TInt32, TInt64`**) with or without limit checking
- There are also **`Val()`** overloads for the unsigned integer types, passing in a radix value (decimal, hexadecimal, binary or octal), and for **`TReal`**
- **`TLex`** provides a number of other API methods for manipulation and parsing
- These are documented in the Symbian OS Library in each SDK

# Converting Numbers to Descriptors

## The descriptor classes

- Provide several ways to convert a number to a descriptor
- The various overloads of **`AppendNum()`**, **`AppendNumFixedWidth()`**, **`Num()`** and **`NumFixedWidth()`** convert the specified numerical parameter to a character representation
- Either completely replacing the contents of or appending the data to the descriptor

# Converting Numbers to Descriptors

## Formatting with conversion directives

- The **`Format()`**, **`AppendFormat()`**, **`FormatList()`** and **`AppendFormatList()`** methods of **`TDes`** each take a format string
- Containing literal text embedded with conversion directives
- And a trailing list of arguments

## Each formatting directive

- Consumes one or more arguments from the trailing list
- And can be used to convert numbers into descriptor data

# Packaging Objects in Descriptors

## Flat data objects

- Can be stored within descriptors using the package buffer **`TPckgBuf`** and package pointer **`TPckg`** and **`TPckgC`** classes
- This is useful for inter-thread or inter-process data transfer when making a client–server request

## A T-class object

- May be packaged whole into a descriptor ("*descriptorized*") so it may be passed in a type-safe manner between threads or processes

# Packaging Objects in Descriptors

The **`TPckgBuf`**, **`TPckg`** and **`TPckgC`** classes

- Are thin template classes
- Derived from **`TBuf<n>`**, **`TPtr`** and **`TPtrC`** respectively (see **`e32std.h`**)
- The classes are type-safe and are templated on the type to be packaged

# Packaging Objects in Descriptors

There are two package pointer classes

- Modifiable **`TPckg`** or non-modifiable **`TPckgC`** pointer descriptors which refer to the existing instance of the template-packaged class
- Functions may be called on the enclosed object (**`TSample theSample`** next slides)
- If it is enclosed in a **`TPckgC`** a constant reference to the packaged object is returned from **`operator()`**

# Packaging Objects in Descriptors

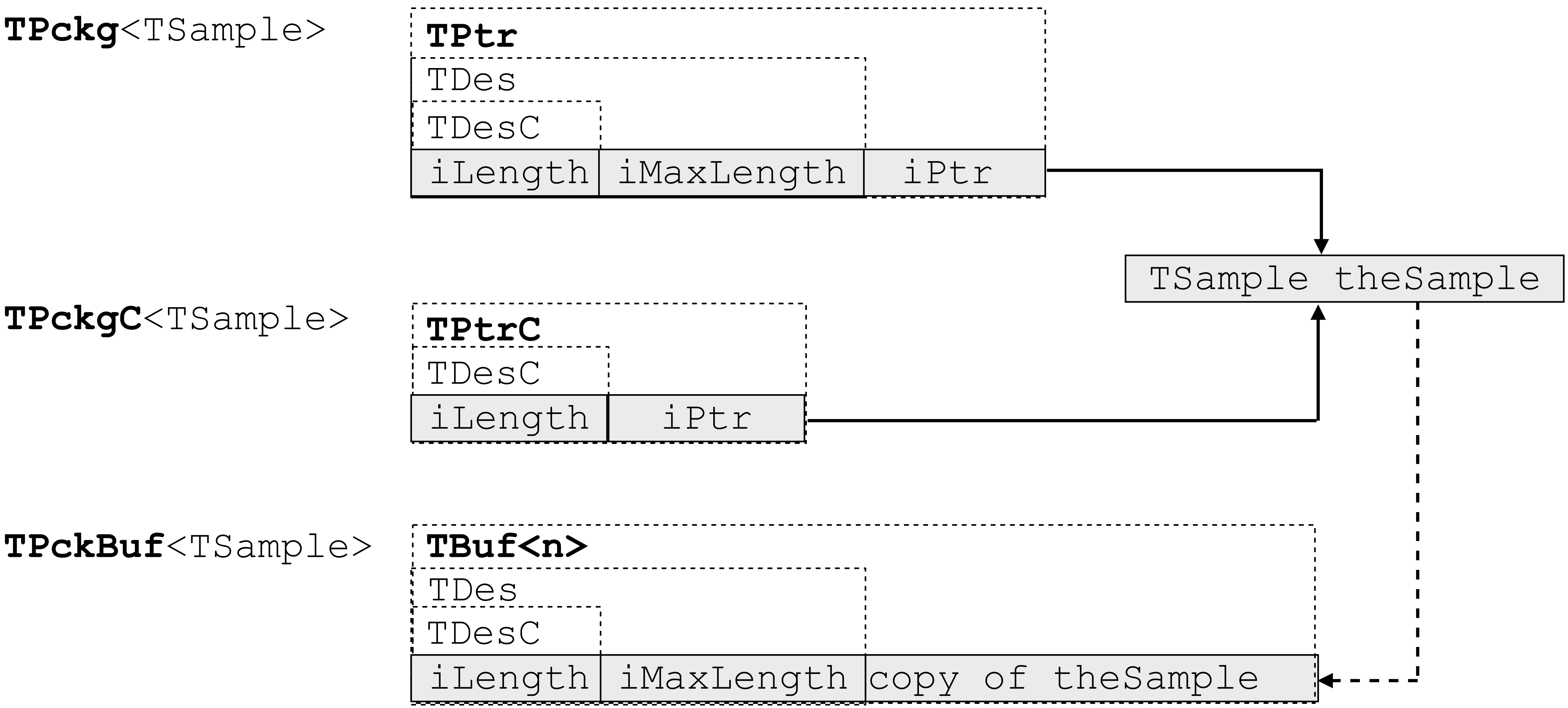
## There is one package buffer class

- The package buffer `TPckgBuf` creates and stores a new instance of the type to be encapsulated in a descriptor
- The copied object is owned by the package buffer (next slide)

## It is modifiable

- Functions may be called on it after calling `operator()` on the `TPckgBuf` object to retrieve it
- The package buffer contains a copy of the original object thus only the copy is modified, the original is unchanged

# Memory Layout of the **TPckg**, **TPckgC** and **TPckgBuf** Classes



# Packaging Objects in Descriptors

A simple T class encapsulated in each of the package types

TSample is the class to be packaged up inside the descriptor packages

Methods that maybe called on the (enclosed) object

Modifiable package containing **theSample**

Non-modifiable package containing **theSample**

Modifiable package containing a copy of **theSample**

Calling methods directly on the enclosed **theSample** object

Compile error! Non-const method

Fine a const method called

Modifying a copy of **theSample**

```
class TSample
    {
public:
    void SampleFunction();
    void ConstantSampleFunction() const;
private:
    TInt iSampleData;
    };
TSample theSample;
TPckg<TSample> packagePtr(theSample);
TPckgC<TSample> packagePtrC(theSample);
TPckgBuf<TSample> packageBuf(theSample);
packagePtr().SampleFunction();
packagePtrC().SampleFunction();
packagePtrC().ConstantSampleFunction();
packageBuf().SampleFunction();
```

# Descriptors: Part Two

- ✓ The Inheritance Hierarchy of the Descriptor Classes
- ✓ Using the Descriptor APIs
- ✓ Descriptors as Function Parameters
- ✓ Correct Use of the Dynamic Descriptor Classes
- ✓ Common Inefficiencies in Descriptor Usage
- ✓ Literal Descriptors
- ✓ Descriptor Conversion