Symbian OS

Descriptors

Disclaimer

- **These slides are provided free of charge at http://www.symbianresources.com and are used during Symbian OS courses at the University of Applied Sciences in Hagenberg, Austria (http://www.fh-hagenberg.at/)**
- **Respecting the copyright laws, you are allowed to use them:**
  - **for your own, personal, non-commercial use**
  - **in the academic environment**
- **In all other cases (e.g. for commercial training), please contact andreas.jakl@fh-hagenberg.at**
- The correctness of the contents of these materials cannot be guaranteed. Andreas Jakl is not liable for incorrect information or damage that may arise from using the materials.
- Parts of these materials are based on information from Symbian Press-books published by John Wiley & Sons, Ltd. This document contains copyright materials which are proprietary to Symbian, UIQ, Nokia and SonyEricsson. “S60™” is a trademark of Nokia. “UIQ™” is a trademark of UIQ Technology. Pictures of mobile phones or applications are copyright their respective manufacturers / developers. “Symbian ™”, “Symbian OS ™” and all other Symbian-based marks and logos are trademarks of Symbian Software Limited and are used under license. © Symbian Software Limited 2006.

Contents

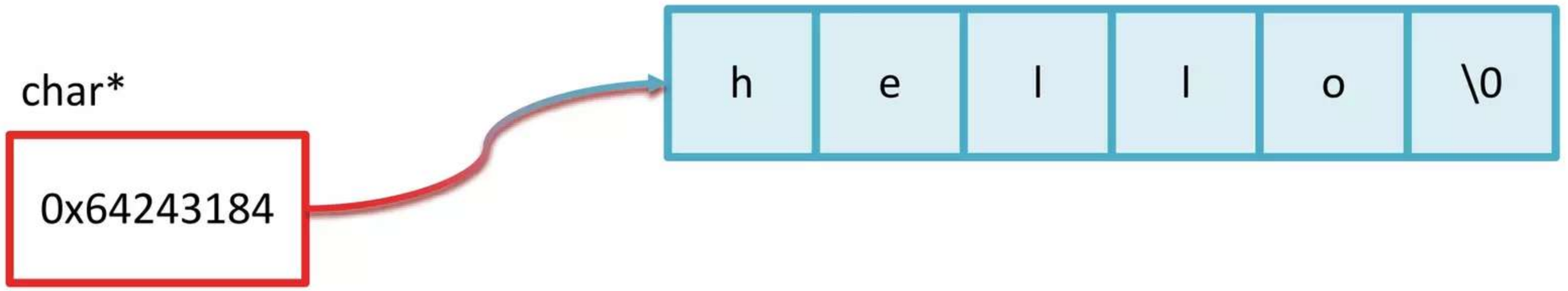
- Literals
- Descriptors
  - TDes / TDesC
  - TBuf / TBufC
  - TPtr / TPtrC
  - HBufC
  - RBuf
- Descriptors as function parameters / return values
- Converting to/from Unicode
- String ↔ Number conversion

Why are they different to Strings?

Introduction

Strings in C

- `char* hello = "hello";`
- **Memory view:**



- Function `strlen()`
  - Reads from the beginning to '\0' and counts number of chars

Descriptors – Introduction

- **Descriptor = Symbian OS string**
- **“self-describing”**
  - Holds length as well as type (= memory layout)
- **Different to:**
  - CString (C++), Java Strings, ...
    → for descriptors, **allocation and cleanup is managed by programmer**
  - C strings
    → Descriptors **protect against buffer overrun and don’t use \0 termination**

Descriptors – Motivation

- Why no normal strings?
  - Minimal **memory usage**
  - Required **efficiency**
    - ROM (Literals: _LIT!) – Stack – Heap?
    - Constant – modifiable?
  - **Unicode** (TBuf16 – TBuf8)
- Take a long time to get used to ☺

Unicode

- Symbian OS uses **16-bit text** since Symbian OS v5u+
- All descriptors available twice
  - **8 bit** (e.g. TBuf**8**)
    - **Used for binary data** and ASCII text
  - **16 bit** (e.g. TBuf**16**)
    - Normally not used explicitly, instead:
- **Build independent** (e.g. TBuf)
  - Currently always Unicode
  - typedef'd to 16 bit version
  - **Use this for normal strings**

_L, _LIT

Literals

Literals

- In reality _LIT is a **macro**, expands to create a:
- **Constant descriptor, compiled to program binary**
  - Not localisable → only use for testing, very simple applications or fixed strings (e.g. for protocols)!
- **_LIT(KHello, “Hello”);**
  - Builds a named object called `KHello` of type `TLitC16`
  - Stores the string `Hello` into the object
  - The string is written to the program binary

_LIT-Macro

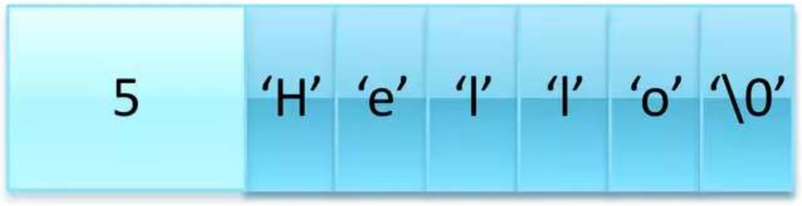
- `TLitC` has the **same memory layout** like a descriptor
- → can be used as a constant descriptor for **function parameters** (`const TDesC&`)
  - **_LIT(**KHello**,** "Hello!"**);**
    console->Printf(KHello);
    iLabel->SetTextL(KHello);
- → can be **casted to a descriptor** through ()-operator
  - TInt length = KHello**().**Length(); // == 6

_L-Macro

- **_L(“Hello”)**
  - Can be used directly as parameter
    - console->Printf(**_L(**“Hello”**)**);
    - User::Panic(**_L(**“example.dll”**)**, KErrNotSupported);
  - Creates temporary TPtrC → **inefficient**, deprecated!
- Both macros defined in e32def.h

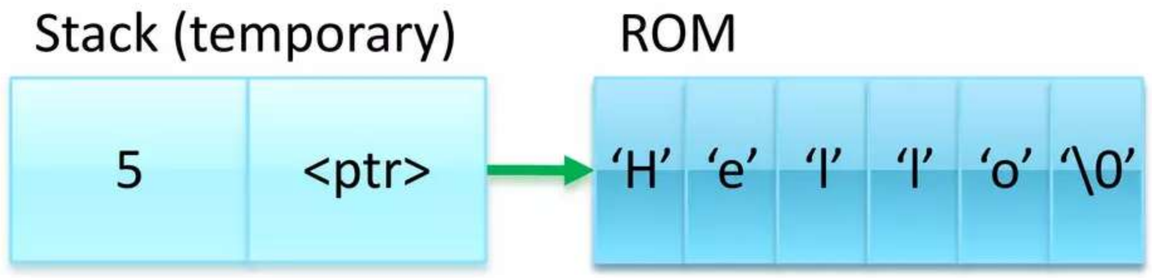
Comparison

_LIT(KHello, “Hello”);



_L(“Hello”)

(through macro) → **TPtrC hello(_L(“Hello”))**



- **For completeness:**
  - Literals are actually ‘\0’-terminated because of the standard C++ compiler, but the additional ‘\0’-char not reflected in the saved length
  - All other descriptors are not ‘\0’-terminated!

TDes, TDesC

Descriptors

Modifiable?

- Do these methods modify the data?

Function	Constant	Modify
Find ()	☐	☐
Copy ()	☐	☐
Append ()	☐	☐
Mid ()	☐	☐
Trim ()	☐	☐
Compare ()	☐	☐
Replace ()	☐	☐
Capitalize ()	☐	☐

Modifiable – Solution

- Do these methods modify the data?

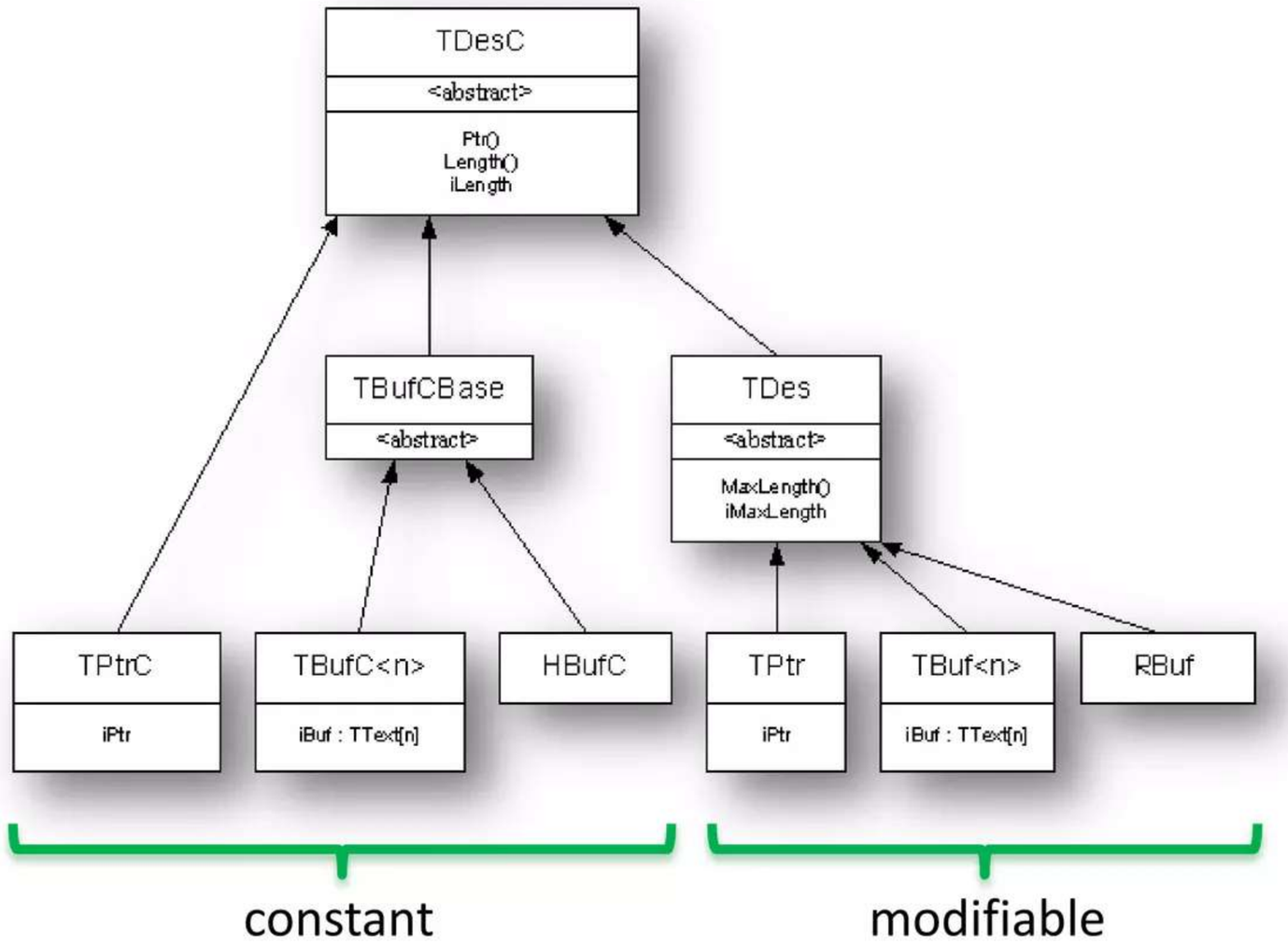
Function	Constant	Modify
Find ()	☑	☐
Copy ()	☐	☑
Append ()	☐	☑
Mid ()	☑	☐
Trim ()	☐	☑
Compare ()	☑	☐
Replace ()	☐	☑
Capitalize ()	☐	☑
defined in ...	**→ TDesC**	**→ TDes**

Constant – Modifiable?

- **Modifiable** (derived from `TDes`)
  - Allow to modify data (replace chars, appending, ...)
- **Non-modifiable** (derived from `TDesC`)
  - Constant data (read-only)
  - Modify it anyway? Possible, if data is not in ROM:
    - Get modifiable pointer descriptor to data using myText.Des();

Inheritance Hierarchy

- Abstract base class because of:
  - **Generalisation** (use base type for parameters!)
  - Provide **basic functions** shared by all types (e.g. `Compare()`, `Find()`,`Mid()`,...)



TDesC / TDes

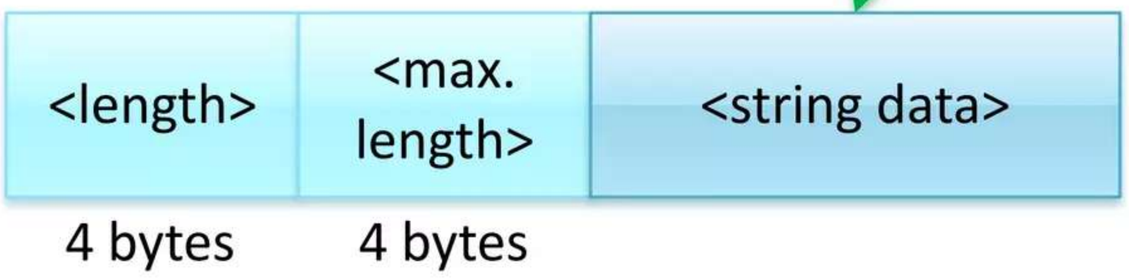
- **Base class of all other descriptors** (except Literals)
  - **Provide basic functionality**
    - **TDesC:** Read-only access (compare, search, ...)
    - **TDes:** Inherits from `TDesC` and adds modification functions. Saves maximum length to prevent overflow.
  - **Cannot be instantiated**
  - Allow **polymorphic** use of descriptors

    → Commonly used in **function parameters**
  - Don't care where **data is stored**

Storing the Length

- **Constant (TDesC)**

- **Modifiable (TDes)**



Depending on descriptor type:
can directly contain the data or be a pointer to data on the stack, heap or ROM

Type

The type is normally not relevant for the developer, therefore it is omitted in following slides for clarity

- **For Completeness:**
  - All Descriptors and Literals also store the **type of the Descriptor** (Stack, Heap, ROM, …)
  - **First 4 bits of iLength** → reduces max. length to 28 bits
  - Used in common methods of `TDes` and `TDesC` to execute correct code depending on type using `switch()`-statement
  - (**Alternative:** virtual functions, but that'd cause overhead)

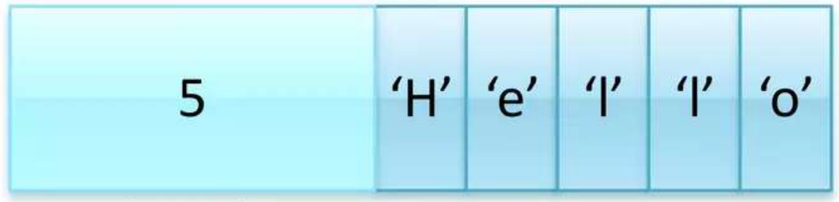
TBuf, TBufC

Stack-Based Buffer Descriptors

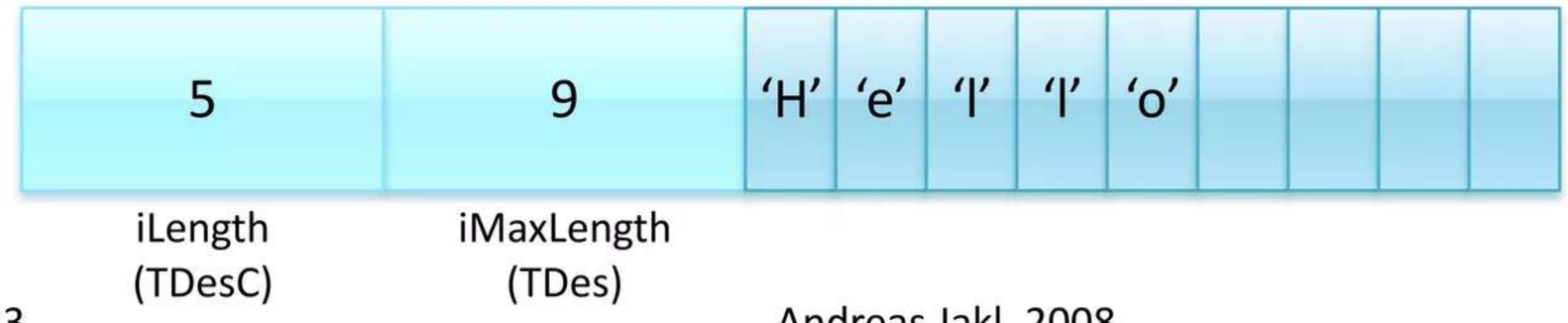
Buffer Descriptors

- Comparable to `(const) char[]` of C
- Directly contain the string
- Use C++ templates to specify length (parameter)

Constant: TBufC<5>

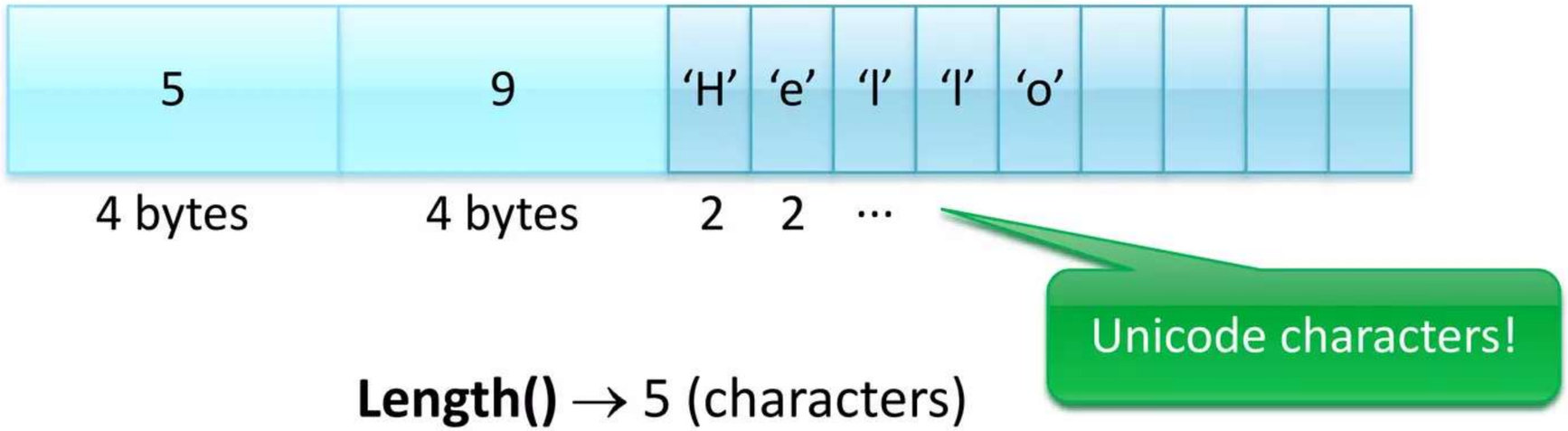


Modifiable: TBuf<9>



Size, Length, MaxLength

- TBuf<9> with text “Hello”



Length() → 5 (characters)
MaxLength() → 9 (characters)
Size() → 10 (bytes)
MaxSize() → 18 (bytes)

Initializing the TBuf

- Debugging in Carbide.c++: Assigning Strings

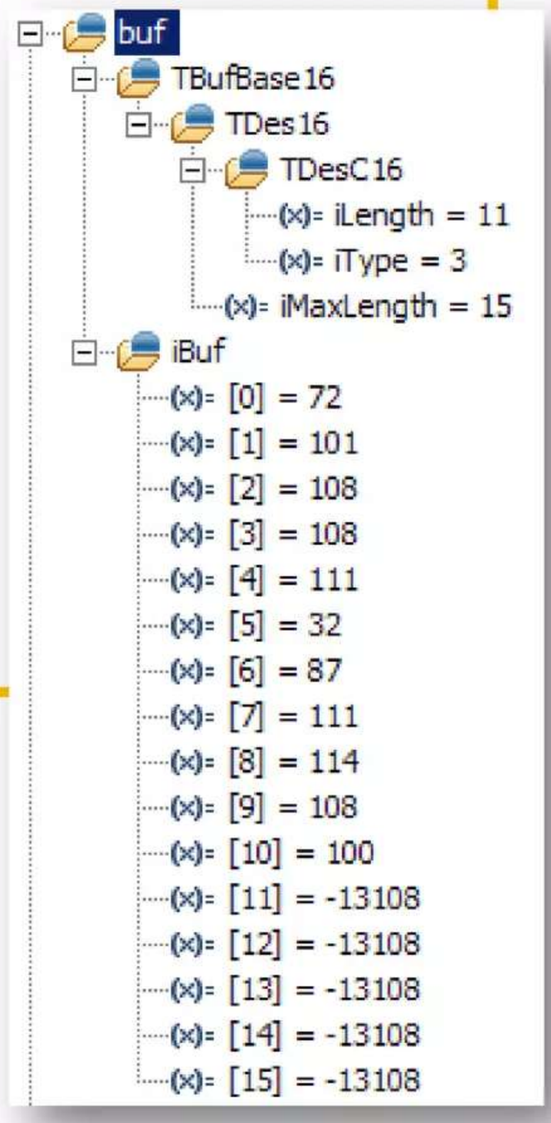
```
// Define constant string literal
_LIT(KString1, "Hello ");
_LIT(KString2, "World");
// Copy KString1 into new TBuf with max. length of 15
TBuf<15> buf1(KString1);
// Same as above, this time using Copy() and KString2
TBuf<15> buf2;              // Initialize empty TBuf with max. length of 15
buf2.Copy(KString2);
// Append contents of buf2 to buf1
buf1.Append(buf2);
// Create constant descriptor based on KString1
TBufC<15> cBuf1(KString1);
// Replace contents of cBuf1 with contents of buf1
cBuf1 = buf1;
```

Using the TBuf

```
_LIT(KHelloWorld, "Hello World");// Defines constant string literal
const TInt maxLength = 15;
// Create a modifiable buffer
TBuf<maxLength> buf;
TInt curLength = buf.Length();              // ..... ?
TInt maxLength2 = buf.MaxLength();          // ..... ?
// Set the contents of the buffer
buf = KHelloWorld;
curLength = buf.Length();                   // ..... ?
TInt curSize = buf.Size();                  // ..... ?
TText ch = buf[1];                          // ..... ?
```

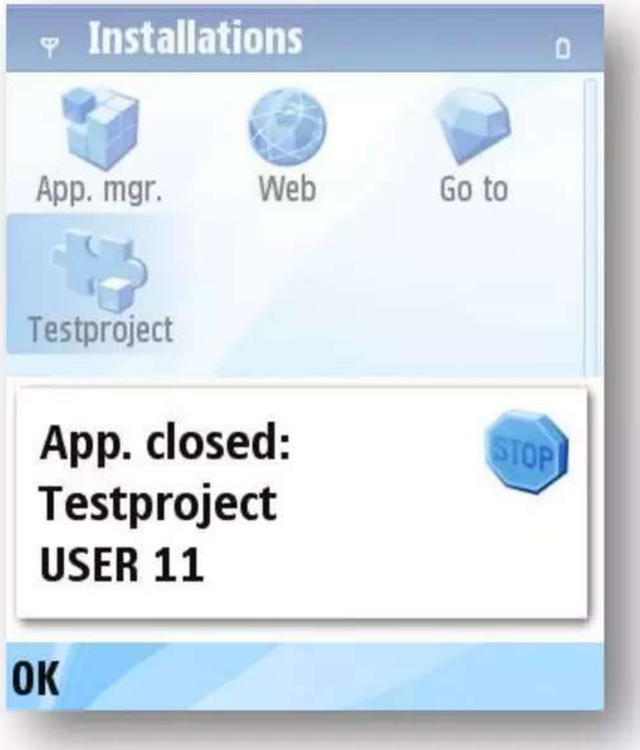
Using the TBuf

```
_LIT(KHelloWorld, "Hello World");// Defines constant string literal
const TInt maxLength = 15;
// Create a modifiable buffer
TBuf<maxLength> buf;
TInt curLength = buf.Length();          // == 0
TInt maxLength2 = buf.MaxLength();      // == 20
// Set the contents of the buffer
buf = KHelloWorld;
curLength = buf.Length();               // == 11
TInt curSize = buf.Size();              // == 22
TText ch = buf[1];                      // == 'e'
```



Maximum Length

```
_LIT(KHello, "Hello ");                 // Defines constant string literal
TBuf<15> buf(KHello);                   // buf == “Hello ”
buf.Append(KHello);                     // buf == “Hello Hello “
buf.Append(KHello);                     // Exceeds maximum length → Panic!
```



SDK-Doc for USER 11-panic:

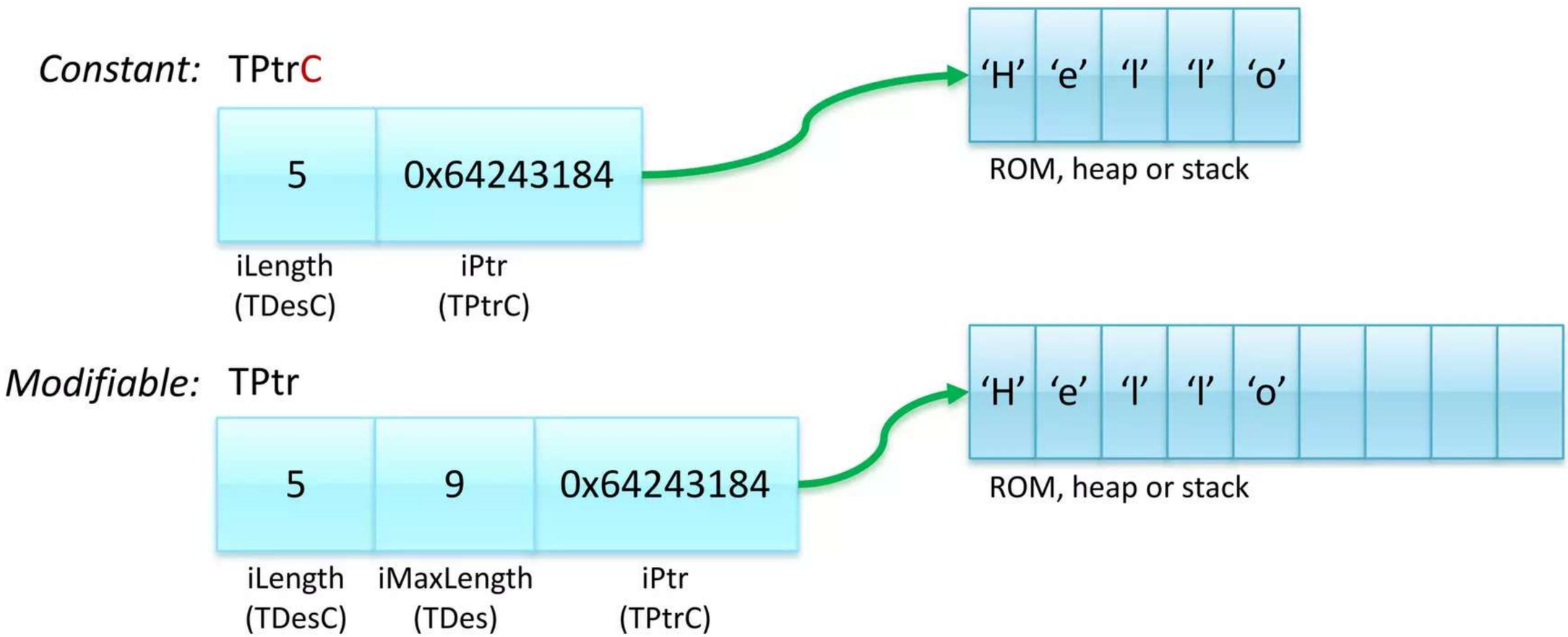
“This panic is raised when any operation that moves or copies data to a 16-bit variant descriptor, causes the length of that descriptor to exceed its maximum length. […]”

TPtr, TPtrC

Pointer Descriptors

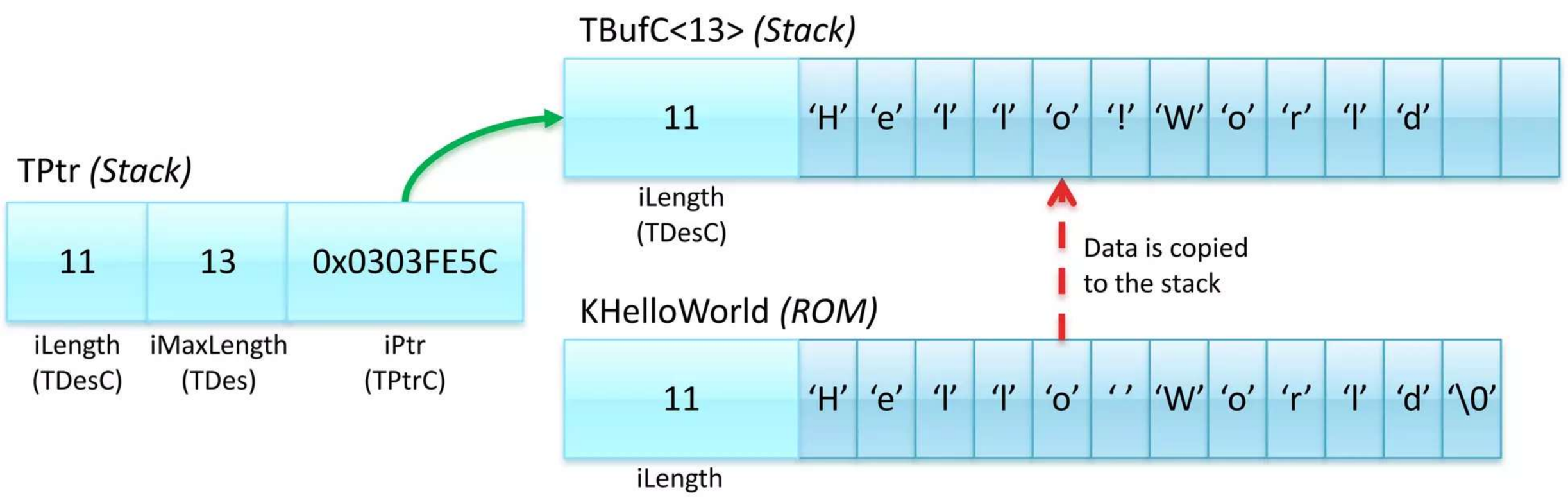
Pointer Descriptors

- Comparable to `(const) char*` of C
- Can **point to text on the heap, stack or ROM**
- Do **NOT own the data** they point to!

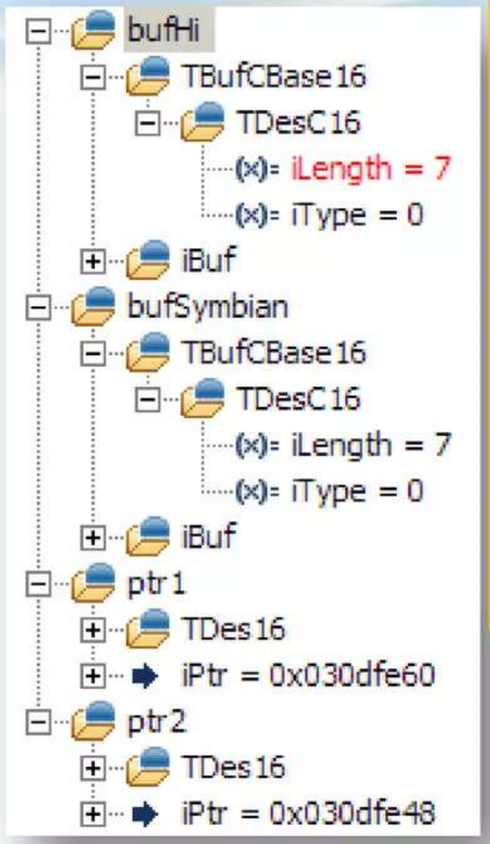


TPtr – Example

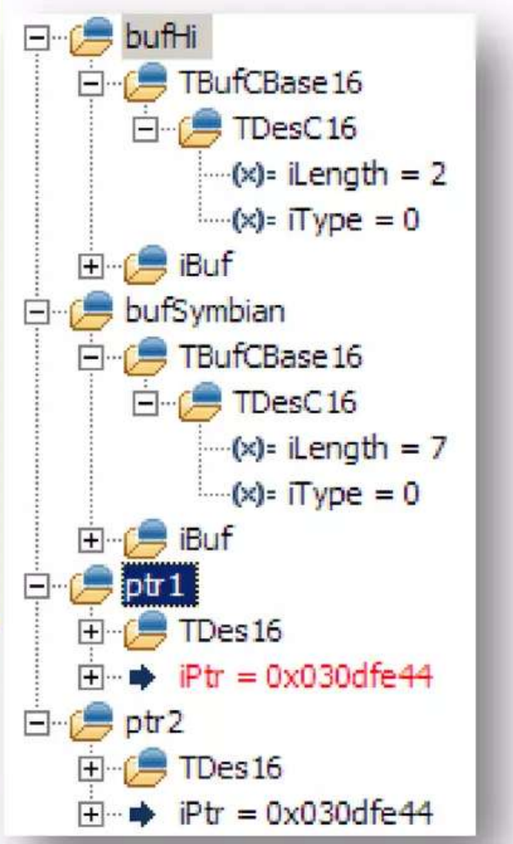
```
_LIT(KHelloWorld, "Hello World");// Defines constant string literal
// Create a constant buffer with contents of string literal
TBufC<13> buf (KHelloWorld);
// Get a pointer descriptor for the text
TPtr ptr = buf.Des();
// Replace the 6th char with an '!', even though TBufC is actually constant!
ptr[5] = '!';                                          // buf: "Hello!World"
```



TPtrC: “=” versus “Set“




```
// Create two literals
_LIT(KHi, "Hi");
_LIT(KSymbian, "Symbian");
// Constant buffer descriptors, copy data from ROM to Stack
TBufC<10> bufHi(KHi);
TBufC<10> bufSymbian(KSymbian);
// ptr1 and ptr2 point to string data owned by TBufC's
TPtr ptr1(bufHi.Des());
TPtr ptr2(bufSymbian.Des());
```



ptr1 = ptr2;

ptr1 → **bufHi**: “Symbian”
ptr2 → **bufSymbian**: “Symbian”
bufHi: “Symbian”
bufSymbian: “Symbian”
… copies Data from buffer pointed to ptr2 to buffer pointed to by ptr1

ptr1.Set(ptr2);

ptr1 → **bufSymbian**: “Symbian”
ptr2 → **bufSymbian**: “Symbian”
bufHi: “Hi”
bufSymbian: “Symbian”
… sets ptr1 to point to same buffer as ptr2

HBufC, RBuf

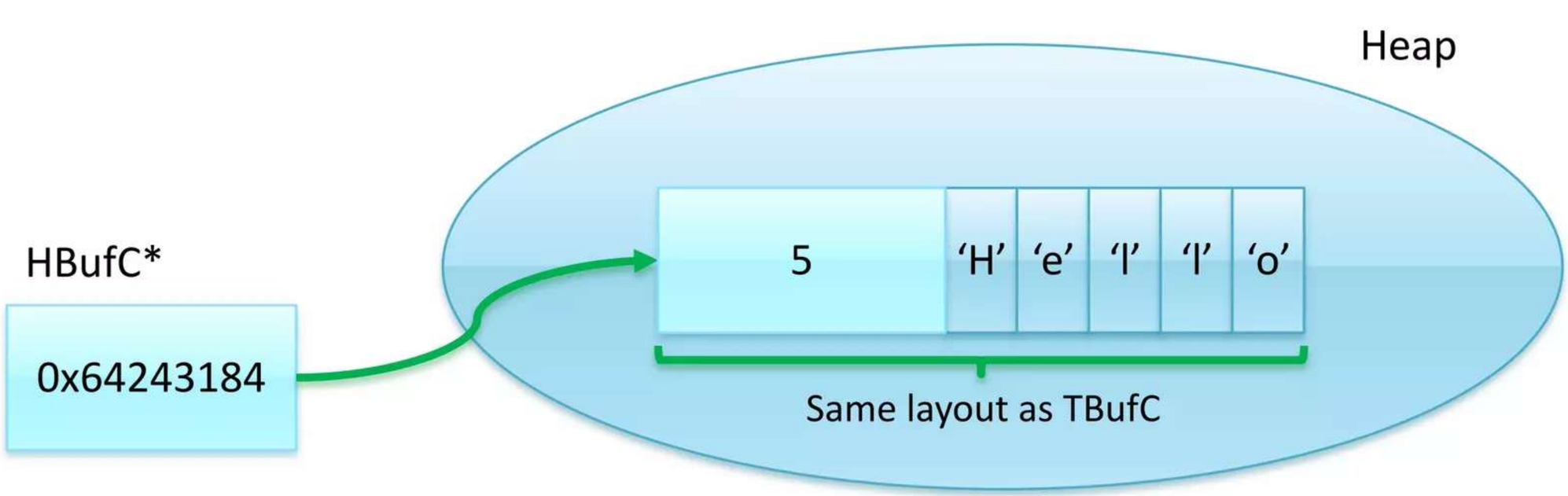
Heap-Based Buffer Descriptors

Use Heap-Based Buffer when…

- **String data** is not in ROM and not on stack
  - Stack size is a lot more limited than the heap!
- **Length of buffer** is not known at compile time, e.g.
  - Loading strings from resource file
  - Getting strings from UI (query dialogs, ...)
  - Receiving response from the web
- Longer **lifetime** than creator
  - e.g. passing to an asynchronous function

Constant Heap Descriptor

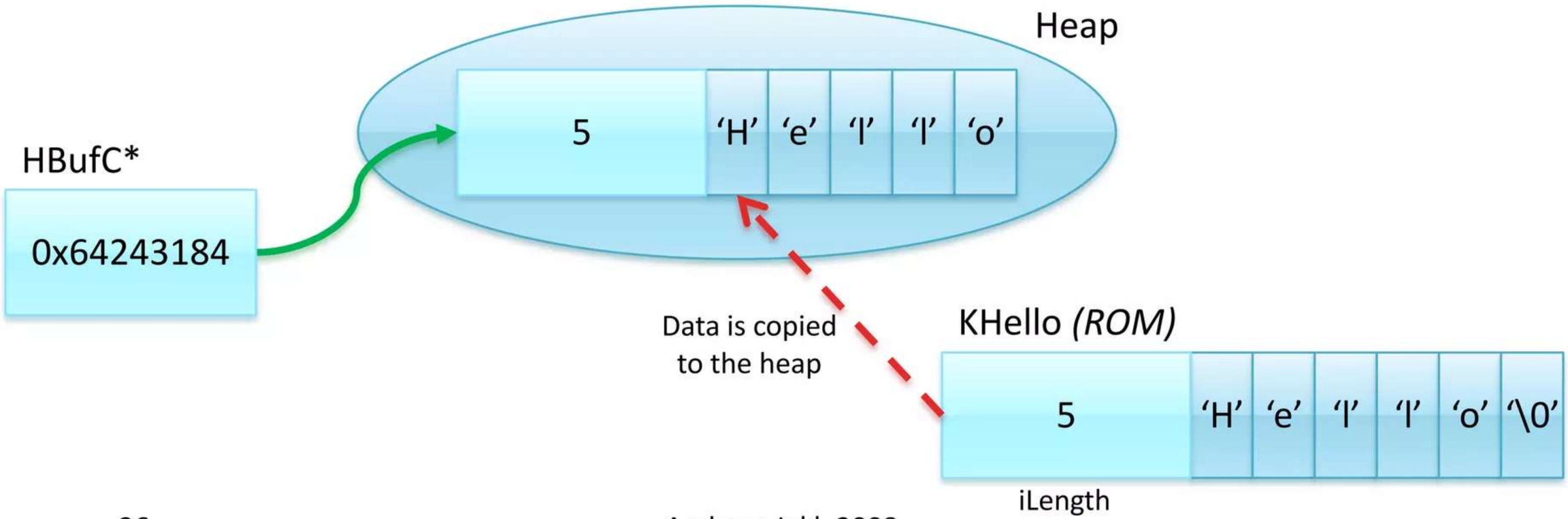
- Comparable to `(char*) malloc(length+1)` of C
- Data is stored on the heap



HBufC Example

```
_LIT(KHello, "Hello");
// Create new HBufC with length of 5
HBufC* hBuf = HBufC::NewLC(KHello().Length());
// Copy data of KHello to heap allocated by hBuf
*hBuf = KHello;
[…]
// Do cleanup
CleanupStack::PopAndDestroy(1);
```

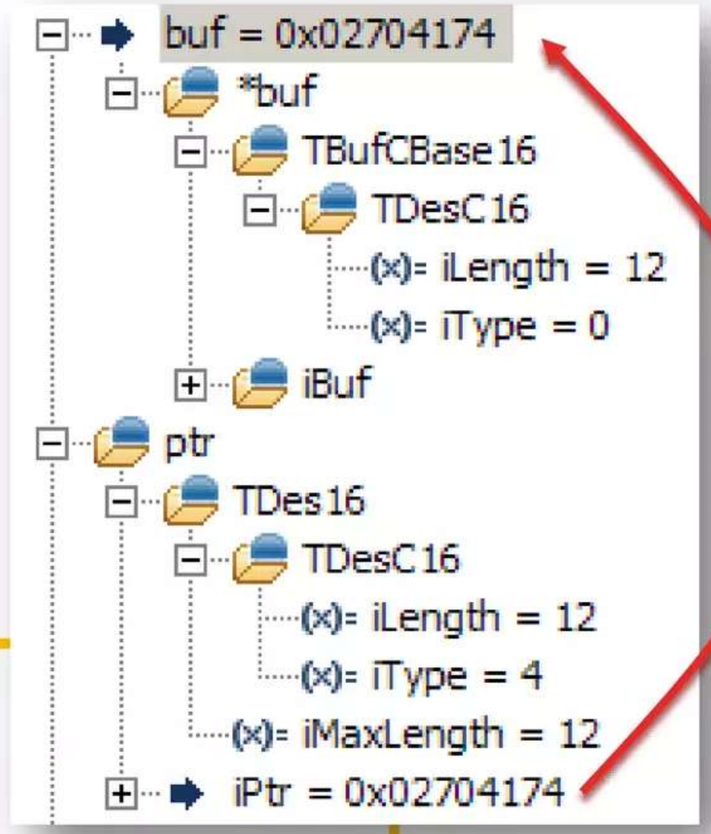
Shorter variant:
HBufC* hBuf = KHello().AllocLC();
(Function of TDesC, creates and returns heap buffer based on a copy of its own data)



Modifying HBufC

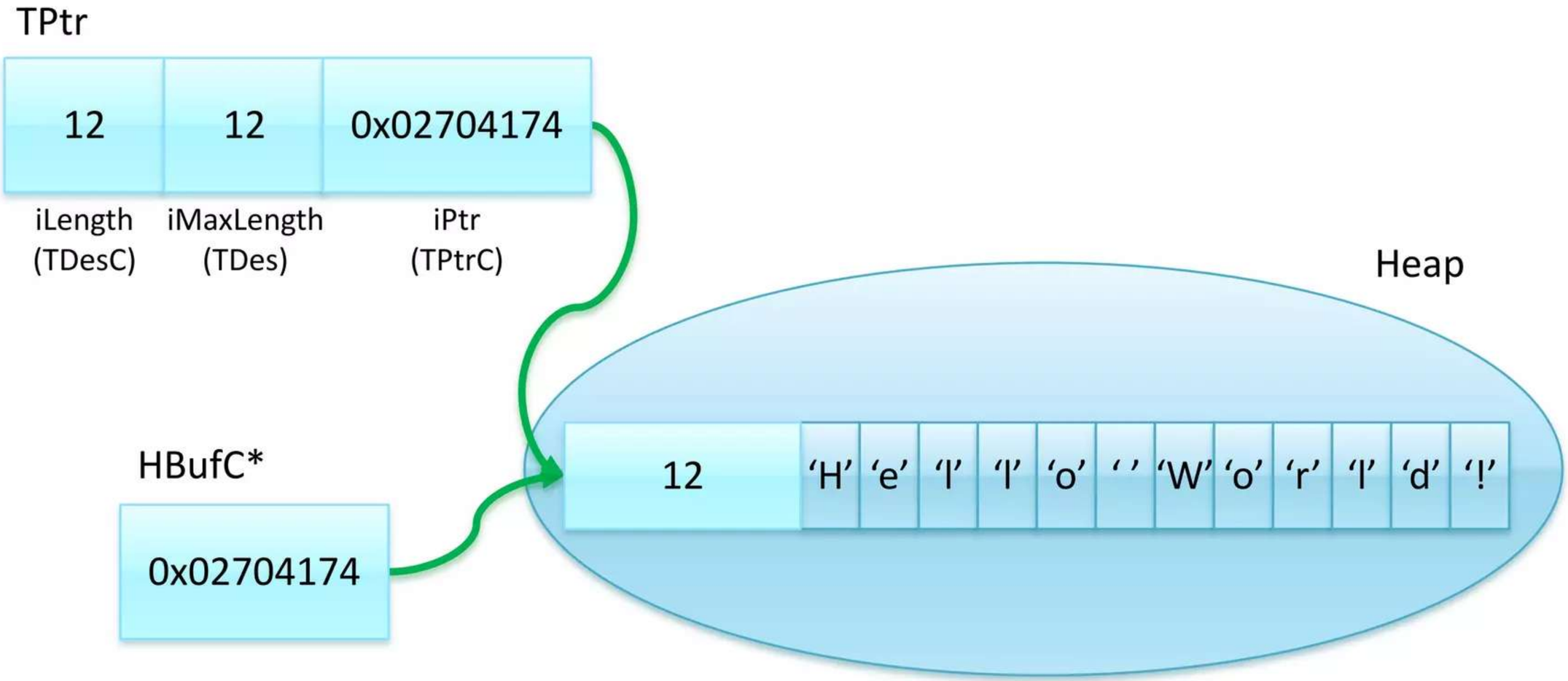
- Modify HBufC through a pointer descriptor:

```
_LIT(KHello, "Hello!");
_LIT(KWorld, "World!");
HBufC* buf = HBufC::NewLC(KHello().Length());
*buf = KHello;                          // buf holds "Hello!", Length = 6
// Increase heap buffer size - still holds "Hello!"
buf = buf->ReAllocL(KHello().Length() + KWorld().Length());
// New memory area has been reserved – update reference on CleanupStack!
CleanupStack::Pop(buf);
CleanupStack::PushL(buf);
// Create pointer descriptor based on heap descriptor
TPtr ptr(buf->Des());                   // Length = 6, MaxLength = 12
ptr[KHello().Length() - 1] = ' ';       // buf holds "Hello "
ptr.Append(KWorld);                     // buf holds "Hello World!"
CleanupStack::PopAndDestroy(buf);
```



To access the data of an HBufC* (e.g. to use it for a TDesC&-parameter), you can write *hptr instead of creating a TPtrC with hptr->Des();

HBufC – Memory View

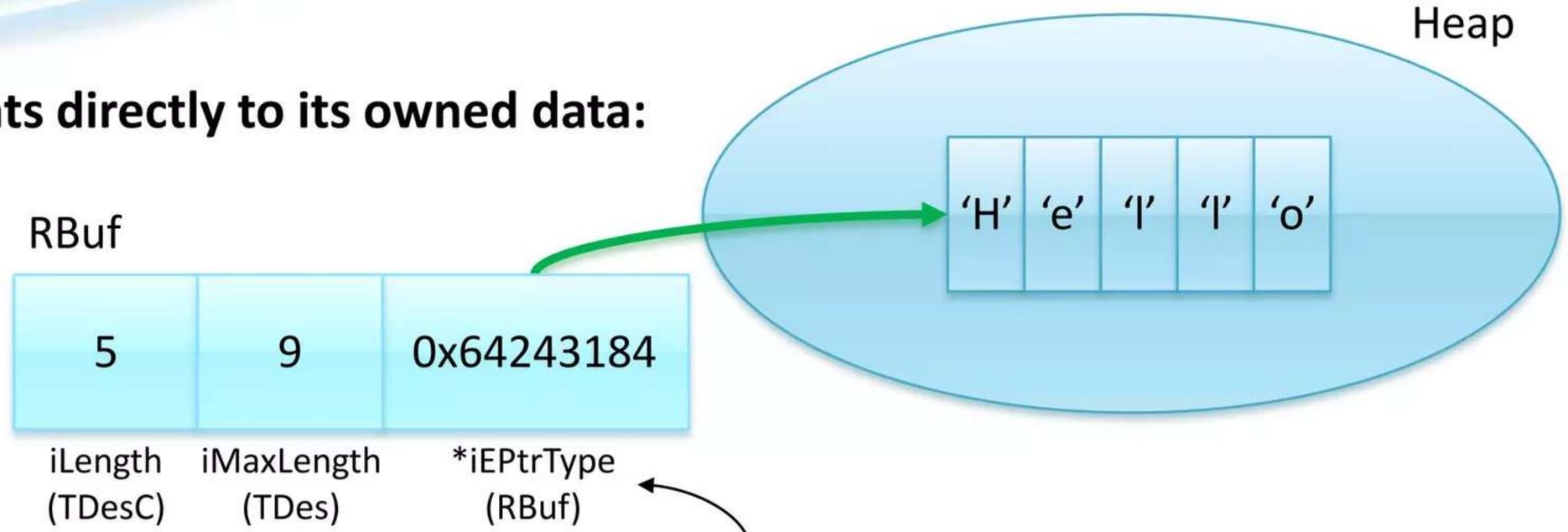


RBuf

- Available since Symbian OS 8
- **Owns and points to modifiable string data on the heap**
- Behaves like a **handle to a resource** (Close() to free)
- **Resize possible**, but has to be done **manually**
- Similar to:
  - `TPtr`, but owns data it points to
  - `HBufC`, but easier to use
- Can be **wrapper** for **HBufC**-Object

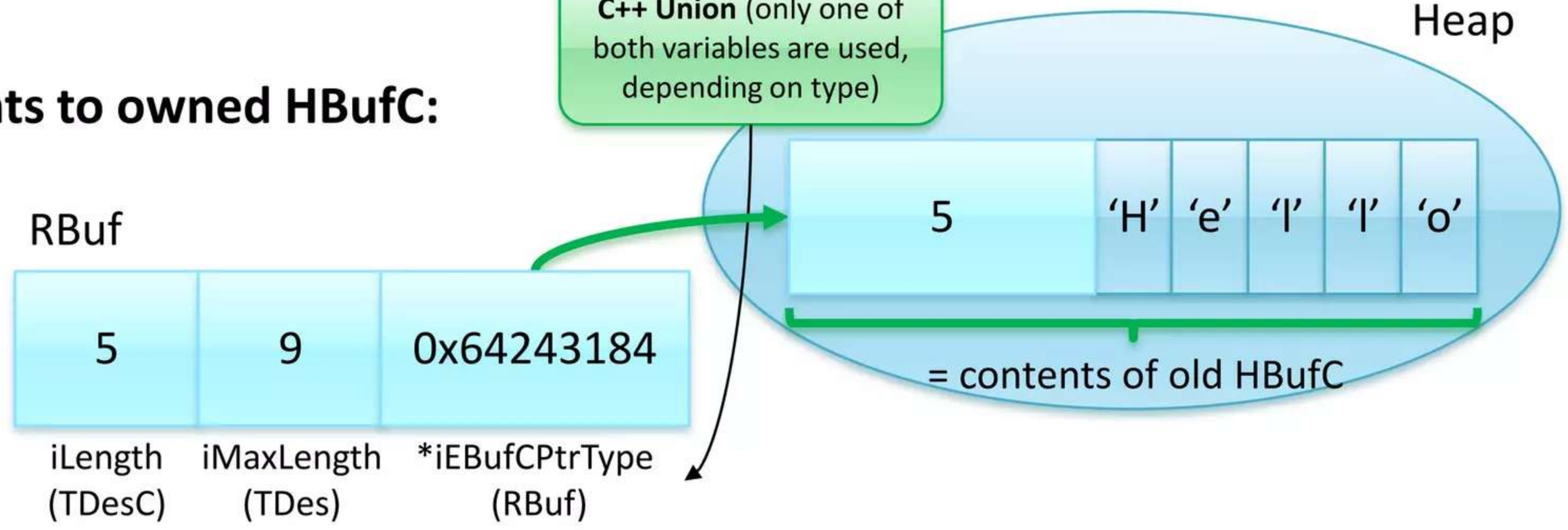
RBuf – Memory Layout

RBuf points directly to its owned data:



C++ Union (only one of both variables are used, depending on type)

RBuf points to owned HBufC:



RBuf – Construction

- **Creating an RBuf:**
  - **Allocate own memory**

```
RBuf buf;
buf.CreateL(TInt aMaxLength);    // Create RBuf with max. length
or: buf.CreateL(const TDesC &aDes);// Copy data to own memory
```

  - **Transfer ownership of HBufC** (wrapper for HBufC)

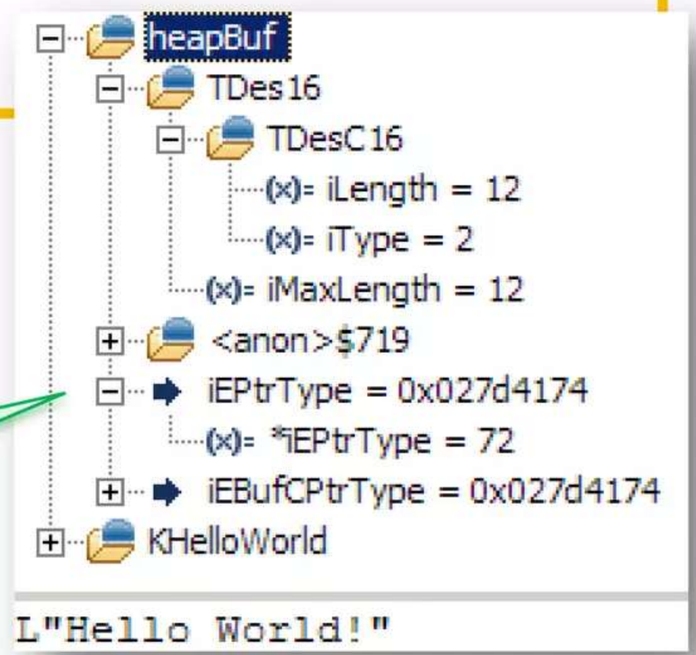
```
RBuf buf(HBufC* aHBuf);       // Transfer ownership of HBufC-data
or: buf.Assign(HBufC* aHBuf);
```

  - **Transfer ownership of existing memory area**

```
buf.Assign(TUint16 *aHeapCell, TInt aMaxLength);
```

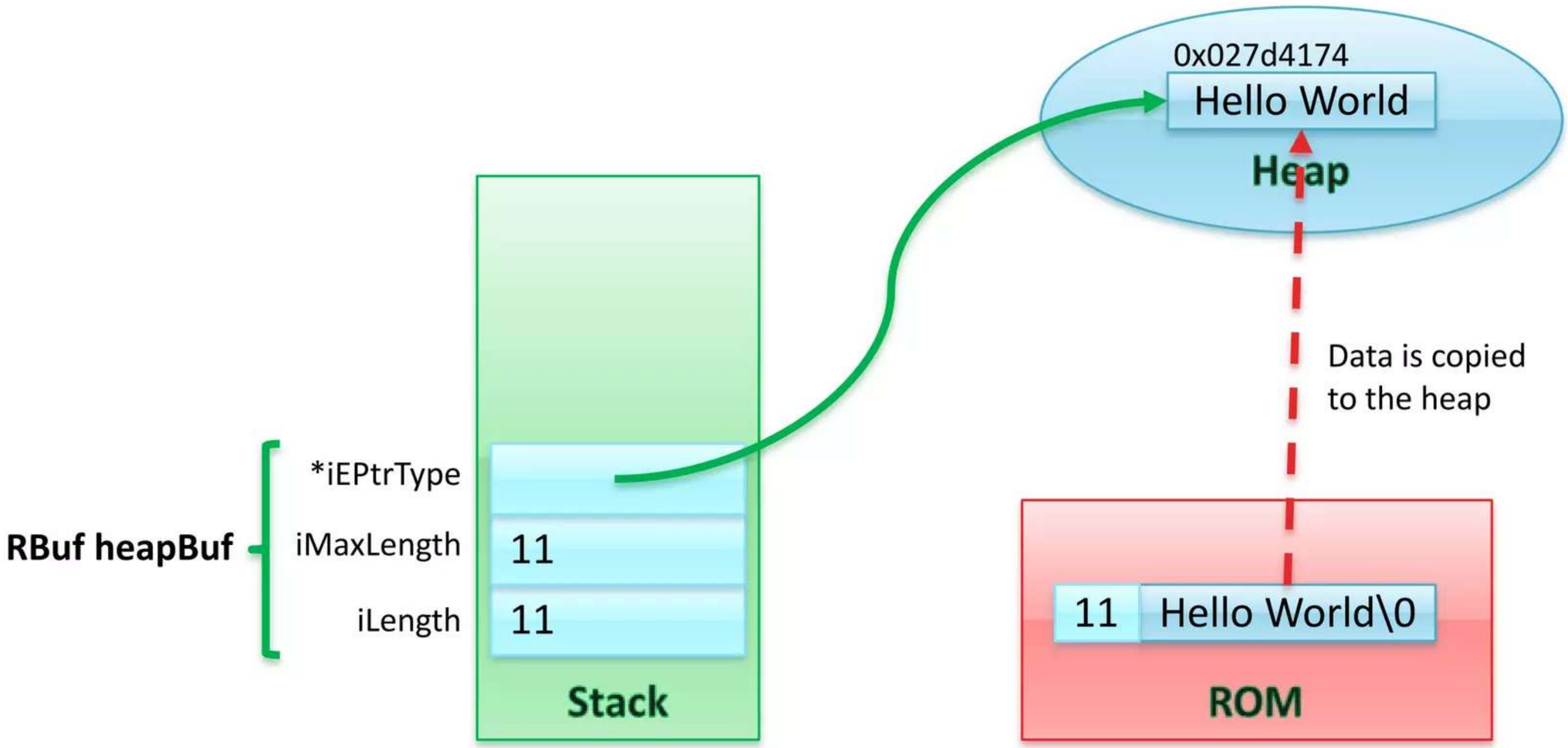
Using the RBuf

```
_LIT(KHelloWorld, "Hello World!");
RBuf heapBuf;
// Create RBuf, copy contents of KHelloWorld
heapBuf.CreateL(KHelloWorld);
// Make sure Close() is called on RBuf in case of Leave
heapBuf.CleanupClosePushL();
[…]
// Calls Close() on the RBuf
CleanupStack::PopAndDestroy();
```



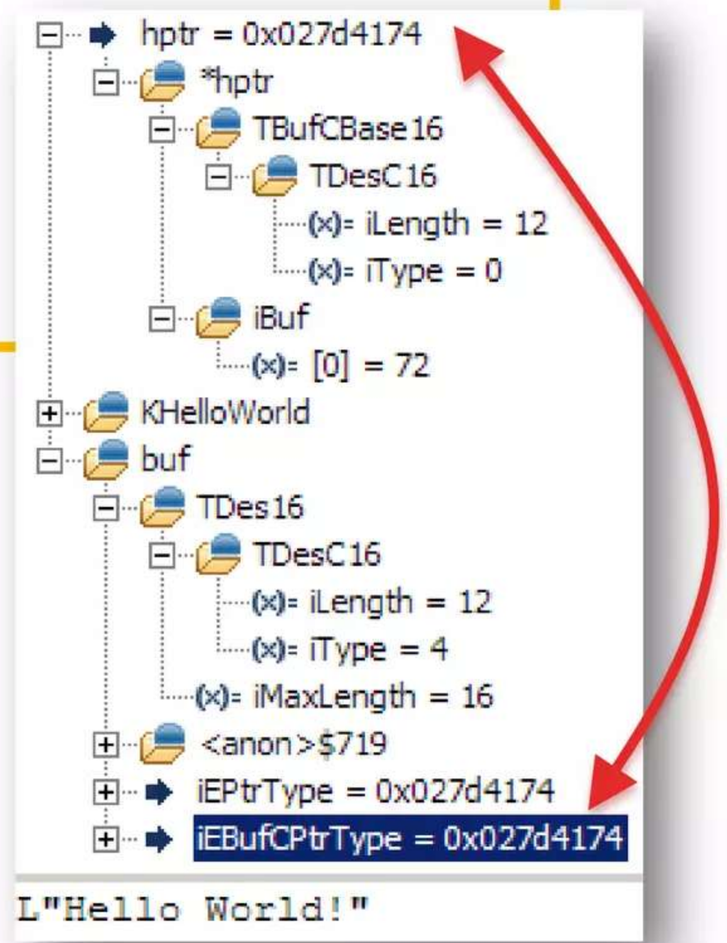
Directly points to string data
(ASCII 72 = “H” of “Hello World”)

RBuf – Memory view

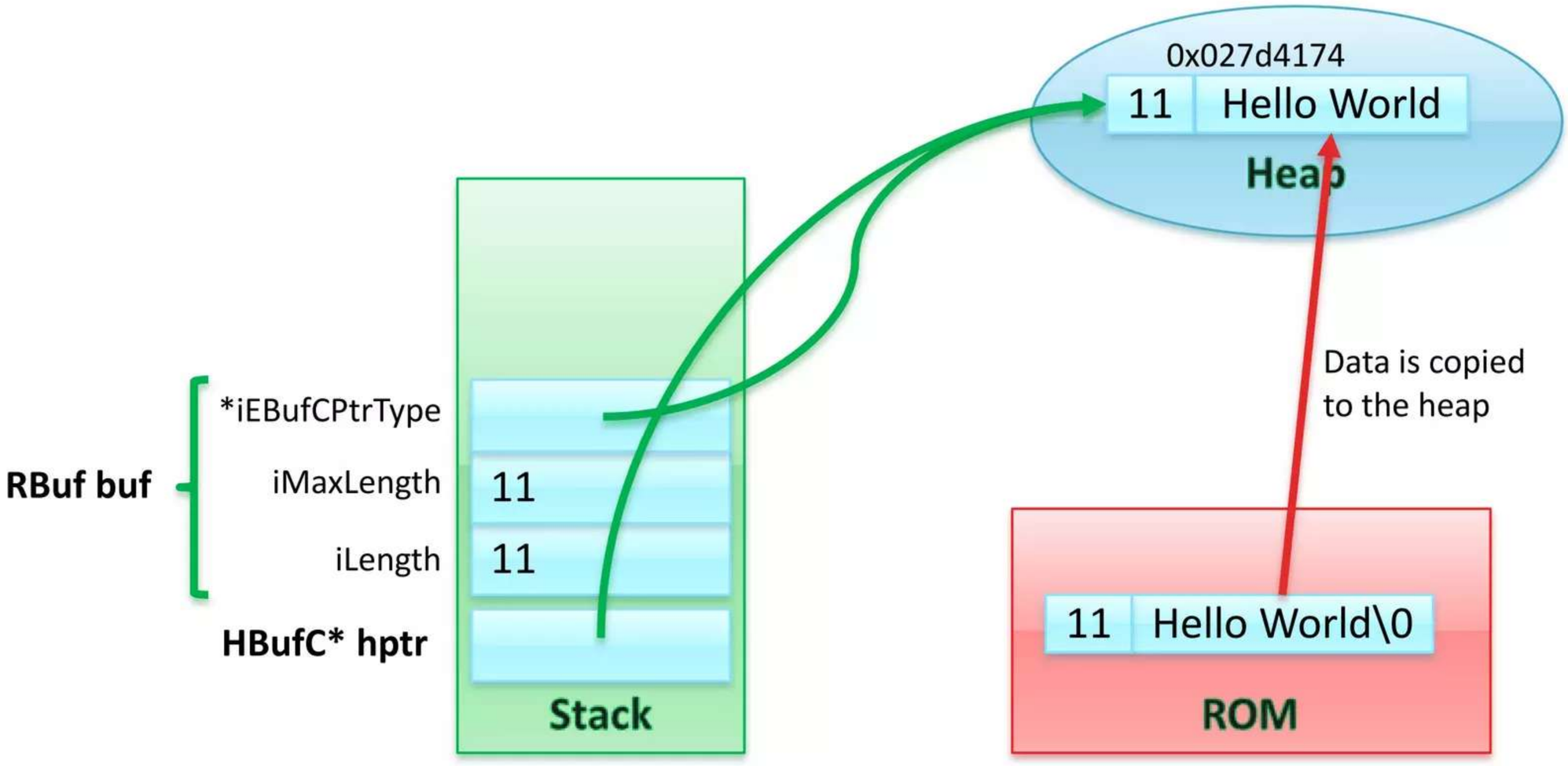


RBuf and HBufC

```
_LIT(KHelloWorld, "Hello World!");
HBufC* hptr;
// Create heap descriptor which can hold up to 15 items. Current lenght = 0
hptr = HBufC::NewL(15);
// Assigns data to heap descriptor. Current length = 12
*hptr = KHelloWorld;
// Ownership of heap descriptor passed to RBuf
RBuf buf(hptr);
// Close buffer and free resources - do not delete HBuf!
buf.Close();
```



RBuf & HBufC – Memory View



... don't delete twice, now the RBuf owns the memory!

One more Example

- It’s recommended to use RBuf instead of HBufC*

```
// Function returns HBufC*, we use RBuf to wrap it → simpler handling for us!
RBuf resString (iEikonEnv->AllocReadResourceL(someResourceId));
// Leaving functions ahead, so push the RBuf on the CleanupStack
resString.CleanupClosePushL();
// Use modifiable descriptor to append new text...
resString.ReAllocL(resString.Length() + 4);
resString.Append(_L("-Foo"));
[…]
// Assign text to a label, resString is a TDesC as well, so it works without changes!
SetLabelL(ELabel, resString);
CleanupStack::PopAndDestroy();          // calls resString.Close();
```

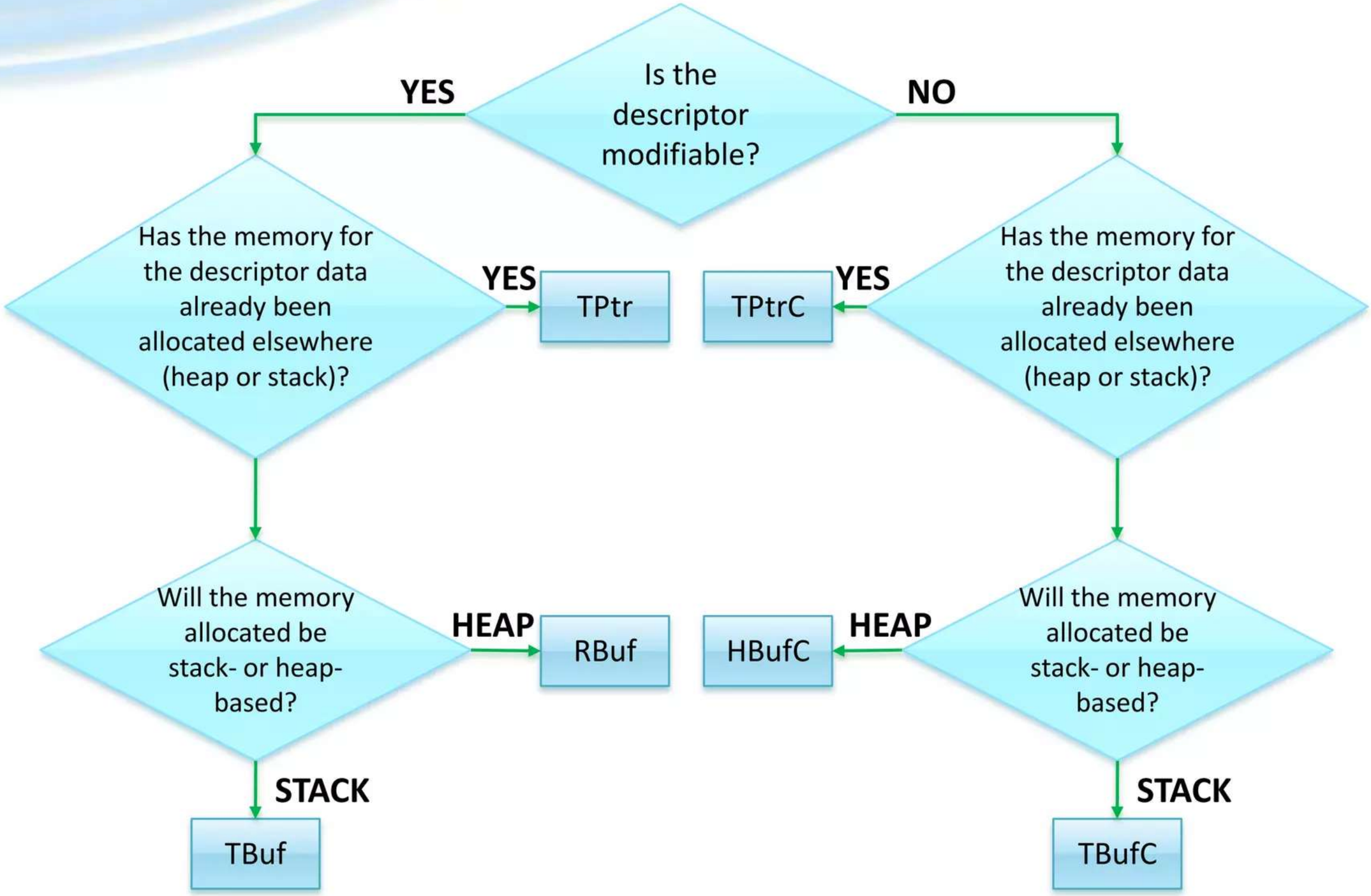
RBuf vs. HBufC?

- **Theoretically:**
  - Use **HBufC** for **constant** data
  - Use **RBuf** for **modifiable** data
- **However:**
  - As **RBuf is a lot easier to use**, consider always using RBuf
  - Most **Symbian OS APIs** operate using **HBufC**
    - Either also use HBufC for these cases
    - Or wrap HBufC with RBuf

Summary

- **Abstract** (TDes, TDesC)
  - Base class of other descriptors
  - Cannot be instantiated
  - Used for function parameters
- **Literal** (TLitC _LIT())
  - Used to store non-modifiable, literal strings in code
- **Buffer** (TBuf, TBufC)
  - Data is stored on the stack
  - Size specified at compile time
- **Heap** (HBufC, RBuf)
  - Data stored on the heap
  - Size specified at run-time
- **Pointer** (TPtr, TPtrC)
  - References data stored outside the class

Lots of types? Decide:

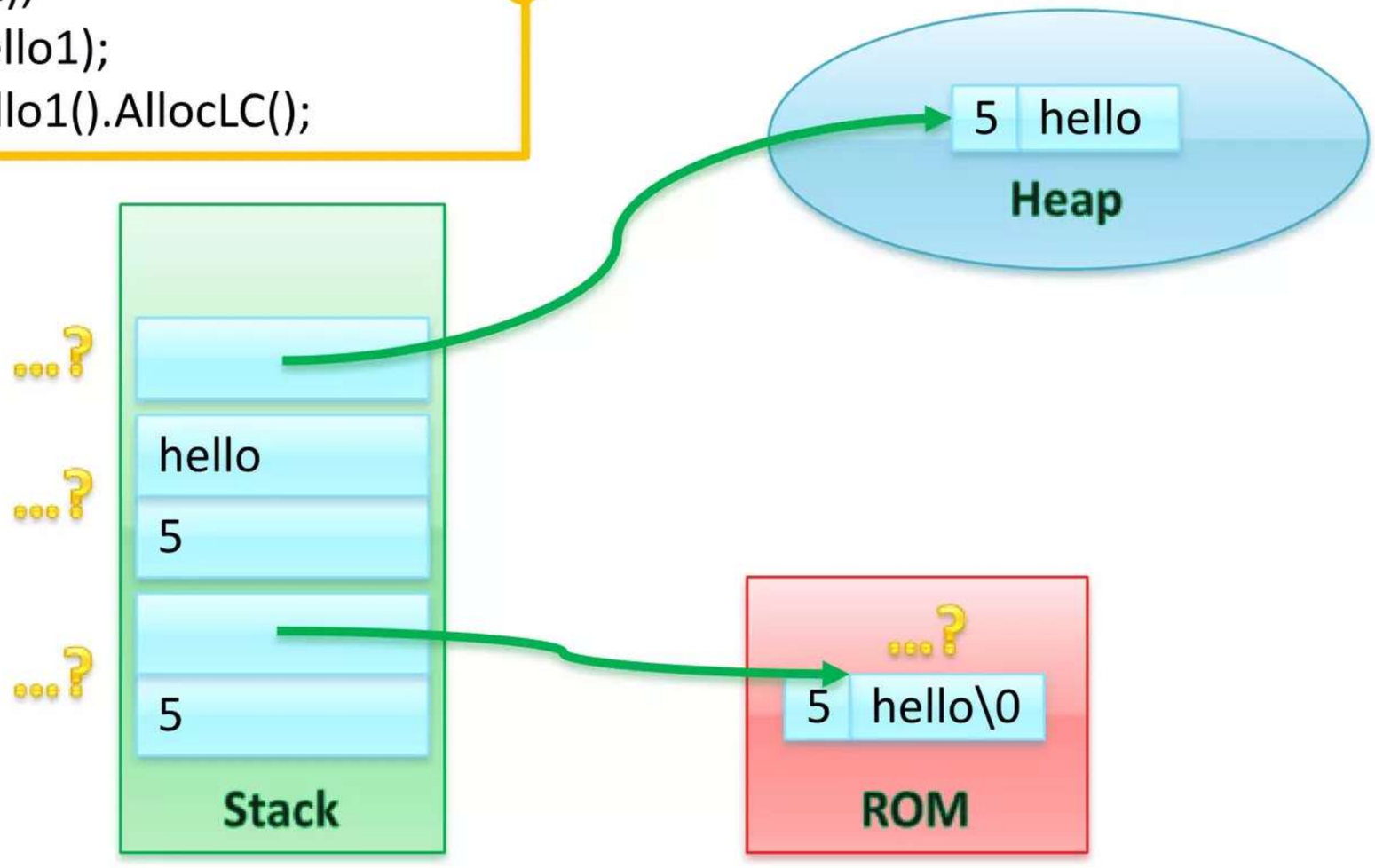


Quiz

TDesC::AllocLC()
"Creates a new 16-bit heap descriptor, initializes it with a copy of this descriptor's data, and puts a pointer to the descriptor onto the cleanup stack."

- Assign variables to the memory view:

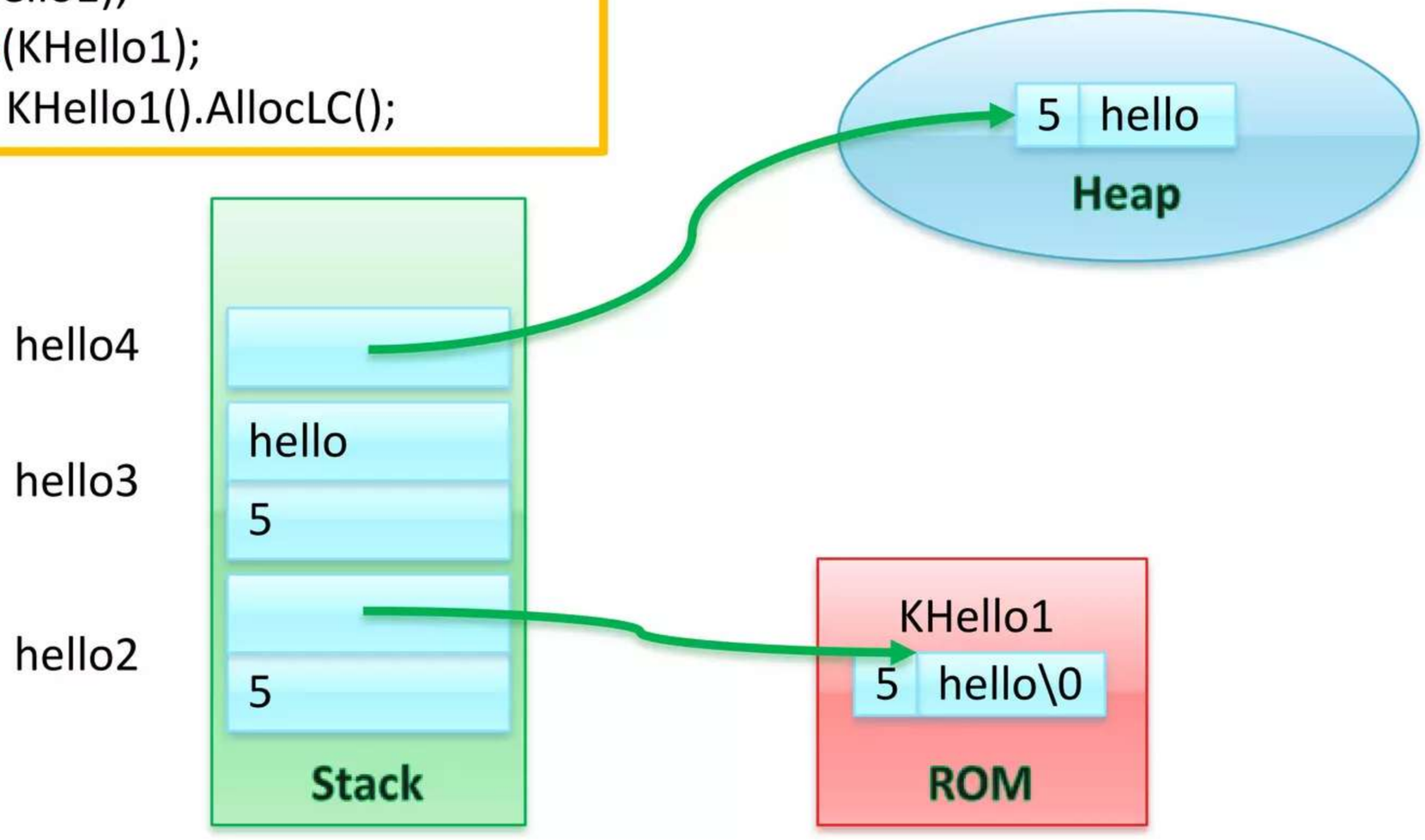
```
_LIT(KHello1, "Hello");
TPtrC hello2 (KHello1);
TBufC<5> hello3(KHello1);
HBufC* hello4 = KHello1().AllocLC();
```



Quiz – Solution

- Assign variables to the memory view:

```
_LIT(KHello1, “Hello”);
TPtrC hello2 (KHello1);
TBufC<5> hello3(KHello1);
HBufC* hello4 = KHello1().AllocLC();
```



The daily life of a Symbian OS developer …

Using Descriptors in Functions

Function Arguments

- void MyFunction(TBuf<8> aText)
  - What if caller uses an RBuf, TPtr or TBuf<9>?

→ References for base classes (TDes&, const TDesC&) should always be used

- void SomeFunction(
  const TDesC& aReadOnlyDescriptor,
  TDes& aReadWriteDescriptor);

Example

- **TInt errorCode = RFile.Read(TDes8& aDes);**
  - Function can find out max. length (aDes.MaxLength())
  - Caller can get number of bytes read: des.Length()
- **Compare that to the Win32-API …**
  - BOOL ReadFile(HANDLE *hFile*, LPVOID *lpBuffer*, DWORD *nNumberOfBytesToRead*, LPDWORD *lpNumberOfBytesRead*, LPOVERLAPPED *lpOverlapped*);

Return HBufC*

- Make sure calling function knows it takes ownership!

```
HBufC* CreateSomeDescriptorL() {
        _LIT(KBert, "bert");
        // Create heap-based descriptor
        HBufC* newBert = KBert().AllocL();
        // Return our HBufC*
        return (newBert);
}
```

- Don’t forget the cleanup stack if it is necessary!

```
HBufC* CreateSomeDescriptorLC() {
        _LIT(KBert, "bert");
        HBufC* newBert = KBert().AllocLC();
        return (newBert);
}
```

Better: RBuf

- Creating a specific RBuf in a function:

```
RBuf myBuf;
myBuf.CleanupClosePushL();
GetSomeDataL(myBuf);
console->Printf(myBuf);
CleanupStack::PopAndDestroy(myBuf);

void GetSomeDataL(RBuf& aBuf) {
        _LIT(KHello, "Hello");
        // Allocate memory and copy „Hello“ to RBuf
        aBuf.CreateL(KHello);
}
```

Returning a TPtrC

- What's wrong with the following code?

```
TPtrC GetText() {
        // Create TBufC based on literal descriptor
        _LIT(KMyText, "My Text");
        TBufC<7> buf(KMyText());
        // Create a TPtrC based on buf
        TPtrC bufPtr(buf);
        // Return the TPtrC
        return (bufPtr);
}
```

Returning a TPtrC – Solution

- **bufPtr** does not own the data
- **buf** is a local variable and ceases to exist when function is left
- → Target of `bufPtr` is no longer valid!

```
TPtrC GetText() {
        // Create TBufC based on literal descriptor
        _LIT(KMyText, "My Text");
        TBufC<7> buf(KMyText());
        // Create a TPtrC based on buf
        TPtrC bufPtr(buf);
        // Return the TPtrC
        return (bufPtr);
}
```

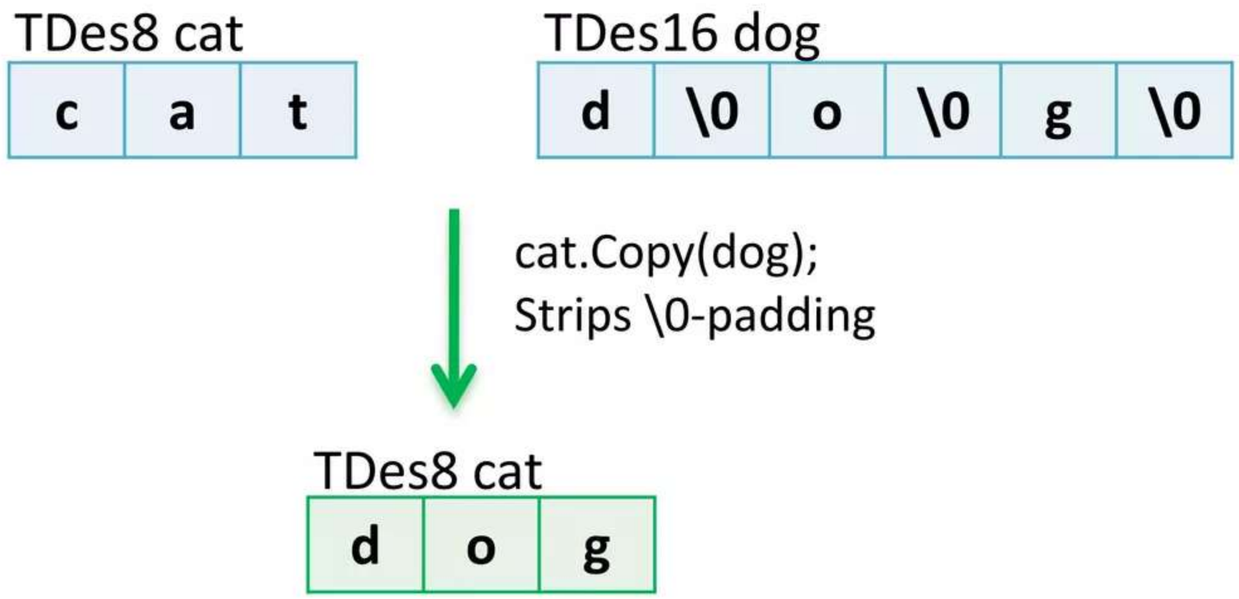
Painless conversion

Unicode and Binary

Converting from Unicode

- Simple conversion by stripping alternate characters:

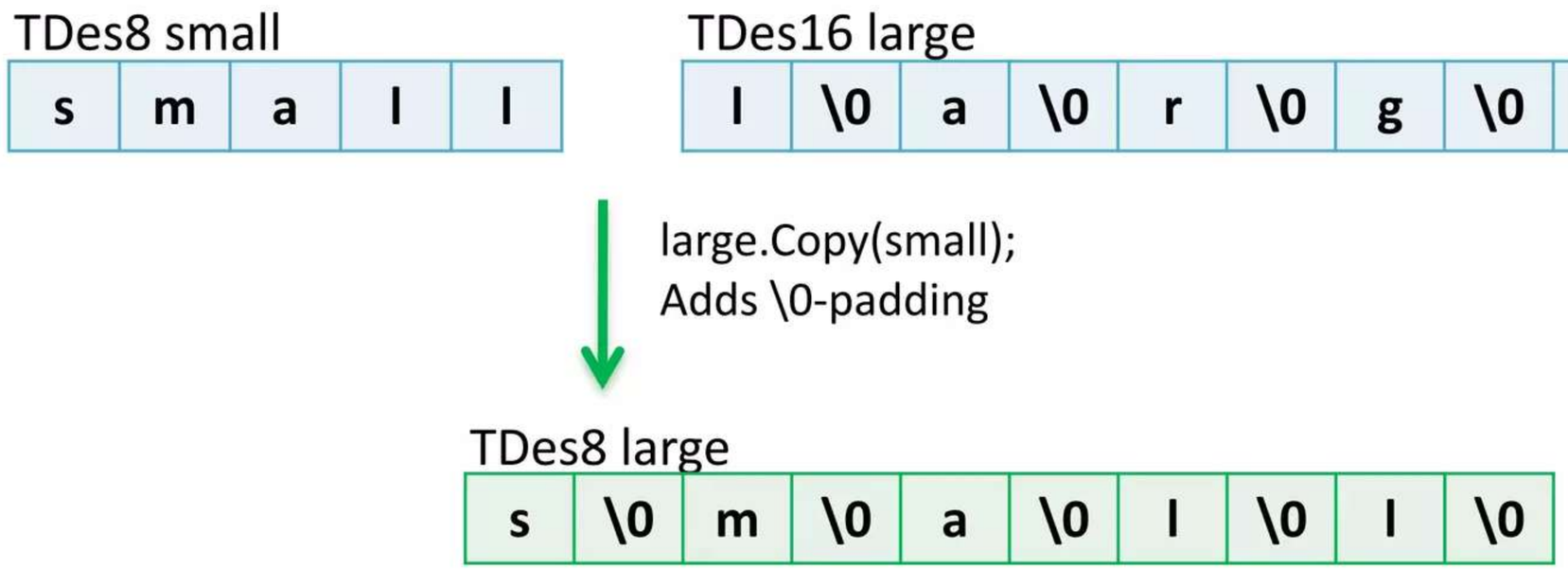
```
TBuf8<3> cat(_L8("cat"));              // _L used for simplicity
TBuf16<3> dog(_L16("dog"));
cat.Copy(dog);                         // cat now contains "dog"
```



Converting to Unicode

- Simple conversion by padding each character with trailing zero:

```
TBuf8<5> small(_L8("small"));
TBuf16<5> large(_L16("large"));
large.Copy(small);                    // large now contains "small"
```



Proper Conversion

- Use conversion library (charconv.lib)
- Requires:
  - #include <charconv.h>
  - #include <utf.h>
  - Library: charconv.lib


```
// Russian text saved as UTF8
_LIT8(KString8, "Телефон");
TBufC8<20> source8(KString8);

// Create target 16 bit buffer for Unicode string
RBuf target16;
target16.CreateL(source8.Length());
target16.CleanupClosePushL();

// Use conversion library to convert from UTF8 to Unicode
CnvUtfConverter::ConvertToUnicodeFromUtf8(target16,
source8);

// Print results and do cleanup
console->Printf(target16);
CleanupStack::PopAndDestroy(1);
```

Other things you should know

More about Descriptor Usage

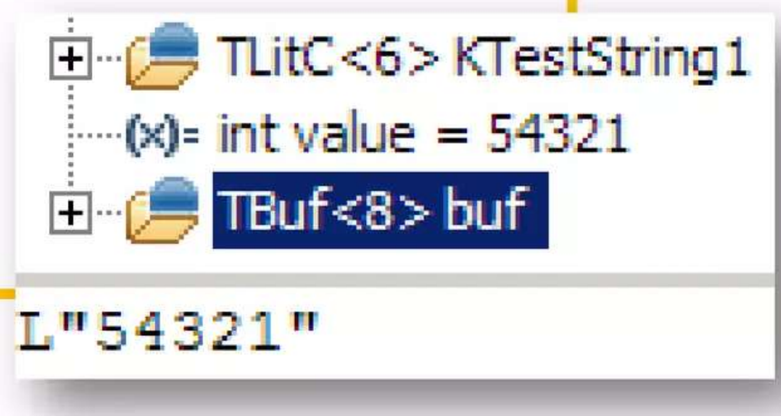
TLex

- General string-parsing functions suitable for numeric format conversions and syntactical-element parsing
  - e.g. parse GPS data (NMEA)
- Powerful, but very complex
- More information at:
  - http://descriptors.blogspot.com/2005/08/35-how-do-i-use-tlex.html

Conversion String ↔ Number

- **String → Number:** Provided by `TDes`
- **Number → String:** Simple use of `TLex`

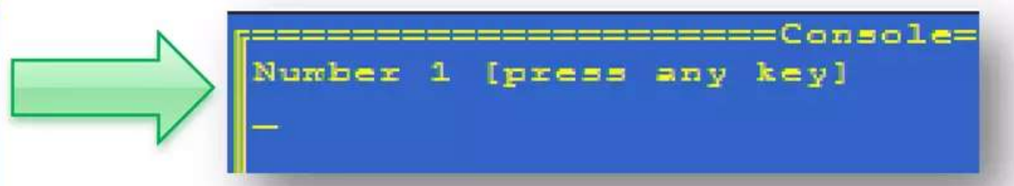
```
_LIT(KTestString1, "54321");

// Convert string to number
TLex lex(KTestString1());
TInt value = 0;
User::LeaveIfError(lex.Val(value));
ASSERT(value==54321);

// Convert number to string
TBuf<8> buf;
buf.Num(value);
```



Formatting Text

- **Integrate variables into text**
  1. Appending text or numbers
     - Simplest way: use the AppendNum()-function defined in TDes
     - Make sure the target descriptor is large enough!

```
_LIT(KText, "Number ");
TBuf<15> myText(KText);
myText.AppendNum(1);
console->Printf(myText);
```

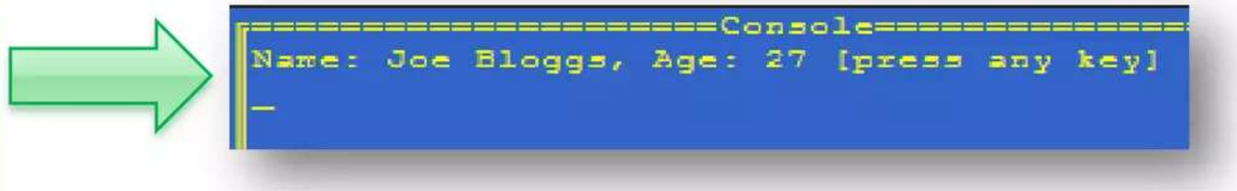


Formatting Text

2. Format strings

- Use **placeholders** for elements
- Full syntax in the **SDK help**: » Symbian OS v*XX* » Symbian OS guide » Base » Using User Library (E32) » Buffers and Strings » Using Descriptors » How to Use Descriptors » Format string syntax

```
_LIT(KName, "Joe Bloggs");
_LIT(KText, "Name: %S, Age: %d");
TBuf<30> myText;
myText.Format(KText, &KName, 27);
console->Printf(myText);
```



3. **Console** (format syntax directly integrated)

```
TInt result = 0;
_LIT(KFormatCompare1, "Compare() using str1 and str2 = %d\n");
console->Printf(KFormatCompare1, result);
```

TFileName

- **Definition of TFileName** (e32const.h / e32cmn.h):
  - const TInt KMaxFileName=0x100; // = Decimal 256
    typedef TBuf<KMaxFileName> TFileName;
- **Required size on the stack:** 520 bytes
  - 2 x 256 data bytes (Unicode), 8 bytes for descriptor obj.
- **Standard stack size in Symbian OS:** 8kB
- **Therefore:**
  - Do not allocate `TFileName` on the stack
    (however: instance var. of `CBase`-class is on the heap!)
  - Do not pass it by value!
  - … use `RBuf` instead!

Did you understand everything?

Test your Knowledge

ASD-like Question – Easy

- **Which of the following statements about descriptors are correct?**
  - **A.** All descriptor classes, except `RBuf`, derive from `TDesC`.
  - **B.** The first four bytes of a descriptor store the descriptor's length and type.
  - **C.** The descriptor classes do not use virtual functions to avoid the overhead of a virtual function pointer in every descriptor object.
  - **D.** Modifiable descriptors use the `NULL` terminator to indicate the end of the descriptor data.
  - **E.** All descriptors have "wide", 16-bit characters.

Solution

- **A.** Incorrect. `RBuf` derives from `TDesC` as well.
- **B.** Correct.
- **C.** Correct.
- **D.** Incorrect. The `NULL` terminator is not used by descriptors. Only Literals are \0-terminated because of the C++-compiler, but this is hidden from the developers.
- **E.** Incorrect. Descriptors are available as 8 bit and 16 bit variants.

ASD-like Question – Medium

```
1.  _LIT(KHello, "Hello!");
2.  TBufC<6> hello(KHello);
3.  _LIT(KBye, "Goodbye!");
4.  TBufC<8> bye(KBye);

5.  TPtr foo(hello.Des());
6.  TInt len = foo.Length();
7.  TInt maxLen = foo.MaxLength();

8.  TPtr bar(bye.Des());
9.  foo.Set(bar);
10. len = foo.Length();
11. maxLen = foo.MaxLength();

12. foo.Copy(KHello);
13. len = foo.Length();
14. maxLen = foo.MaxLength();
```

Which of the following statements are correct?

A. After executing line 2, a call to `hello.MaxLength()` returns 6.

B. After executing line 6, `len = 6;` After line 7, `maxLen = 6`.

C. After executing line 9, hello contains "`Goodbye!`".

D. After executing line 10, `len = 8;` After line 11, `maxLen = 8`.

E. After executing line 13, `len = 6;` After line 14, `maxLen = 6`.

Solution

- **A.** Incorrect. `TBufC` does not have a `MaxLength()`-function. This is defined in `TDes` base class.
- **B.** Correct.
- **C.** Incorrect. After the `Set()`-function, the `foo-TPtr` points to the same memory as the `bar-TPtr`. The buffers are not altered however. This would be the case with =.
- **D.** Correct.
- **E.** Incorrect. `MaxLength = 8`, as `foo` is now pointing to the buffer of the `bar-TPtr`, which is the `bye-TBuf` with a max. length of `8`.

ASD-like Question – Hard

Which of the following uses of a descriptor should be reconsidered?

A. USE 1
B. USE 2
C. USE 3
D. USE 4
E. USE 5

```
class CTestClass : public CBase {
public:
	static CTestClass* NewL(const TFileName aFilename); // USE 1
public:
	inline const TDesC& FileName() {return (iFileName);}; // USE 2
	void SendDataL(const TBufC<20>& aData); // USE 3
	void ReceiveDataL(TDes& aData); // USE 4
private:
	// Construction code omitted for clarity
private:
	TFileName iFileName; // USE 5
};
```

Solution

- **A.** Reconsider. `TFileName` should be passed by reference, not by value.
- **B.** OK.
- **C.** Reconsider. Use generic parameter instead (`const TDesC& aData`).
- **D.** OK.
- **E.** OK.

Resources

- Famous blogs with **answers to common problems**:
  - http://descriptors.blogspot.com/
  - http://descriptor-tips.blogspot.com/
- **Detailed information in the books:**
  - Symbian OS C++ for Mobile Phones, Vol. 3
  - Symbian OS Explained

Unfortunately this book do not cover the most recent addition to Symbian OS, the RBuf. Apart from this minor disadvantage, it is very useful.

That’s it!

Thanks for your attention