

Descriptors_as_Function_Parameters

Descriptors are very powerful and efficient, but you have to be aware of their properties to use them correctly. Once you have understood the general concept of descriptors, you will sooner or later need to pass a descriptor to a function as a parameter. The easiest way to do this would be to define your function as follows:

```
void myFunction(TBuf<8> aText);
```

The problem with this solution is that you limit the possible parameter type to a TBuf (modifiable, stack-based descriptor) with a length of 8. If the caller uses a different descriptor type, like the new RBuf, he won't be able to directly call your function.

This is where the general TDes (modifiable) and TDesC (constant) base types are important. If you do not need specific functionality, you should always use the generic base types for your parameters.

```
void SomeFunction(  
const TDesC& aReadOnlyDescriptor,  
    & aReadWriteDescriptor);
```

This example above shows two different descriptors:

- 1. Read-only descriptor parameter:** TDesC is the base class of all other descriptor types, even the modifiable TDes. Therefore, if you're using a modifiable RBuf, you can still pass it to this parameter. The definition with the constant type plus the constant reference ensures that it can not be altered.
- 2. Read/Write descriptor parameter:** In contrast to the first parameter, this is a non-const TDes descriptor. This means the caller can modify the contents, but it's not important for him exactly where the descriptor content is stored (stack or heap).

Another important thing to note about the use of descriptors as parameters is that they must be passed by reference (e.g. TDes&) and not by value (e.g. TDes). This is because passing by value uses static binding, so the receive function receives an object of the base class type, which contains no access to the descriptor's data. It will all compile OK, but nothing when you run the code, will fail. Look at the descriptor in the debugger and you'll see why - it contains rubbish where the data should be.

API Example

Of course, the Symbian OS API uses the same paradigm for the system functions. Take a look at the code to read from a file:

```
TInt errorCode = RFile.Read(TDes8& aDes);
```

The function will know the maximum available length of the descriptor through aDes.MaxLength() and won't read more than possible through string size limitations. On the other hand side, the caller can get the number of bytes read by using des.Length(), which can be the same as the maximum length or also less than that.

If you compare this to the Win32-API without a good string API, the same function is a lot more complicated - as the function can not find out how many bytes to read and the caller can't get the number of bytes actually read through the string class:

```
BOOL ReadFile(HANDLE hFile, LPVOID lpBuffer, DWORD nNumberOfBytesToRead,  
    LPDWORD lpNumberOfBytesRead, LPOVERLAPPED lpOverlapped);
```

Important Links

[HBufC as method parameter](#)