03: Two-Phase Construction and Object Destruction

Tutorial

Goal

In this module, you will see how two-phase construction works in Symbian OS and what can happen if you do not implement it correctly.

Introduction

Every application you will write for Symbian OS will use own or pre-written C type classes. Usually, these will require two phased construction, so it's important that you know how this works and how it is correctly implemented.

Structure of this Exercise

In this exercise, you will start with a nearly empty framework. It's based on the same framework as the last module (*Leaves and the Cleanup Stack*), so it should be familiar to you.

An incomplete definition of a C type class has been added at the beginning of the code. The class is called `CShopEntry`. Only one function is pre-written: `PrintEntry()`, which outputs the two instance variables to the console. Those two variables (`iName` and `iPrice`) contain the code. `iName` is a heap based descriptor (`RBuf`) which is owned by the class.

To complete this module, you have to work through this document and complete the described tasks. You will first create a standard constructor and destructor and then see in a step by step process why this is not sufficient in this case – and which mechanisms are proposed by Symbian OS to write a totally memory leak free code.

Note that we will not do everything in the correct way right from the beginning – this is to explain and to experience hands-on what happens if common errors are made and why the approach was wrong.

Tutorial Tasks

Preparations

Open the sample framework (project "Construction") in Carbide.c++. Our goal is to create a class called `CShopEntry`, which can store the name of an item as well as its price.

Some parts of the class have already been pre-written. This includes the `CShopEntry::PrintEntry()`-function, which just prints a line to the console containing the name of the article as well as its price.

Those two variables have already been declared as private instance variables called `iName` (of type `RBuf`, more on that in the Descriptors-module) and `iPrice` (of type `TInt`).

At the beginning, the (incomplete) definition of the class looks like this:

```
class CShopEntry
	{
public:
	// Constructors & Destructors

private:
	// Private construction code

public:
	// Public functions
	void PrintEntry();

private:
	// Private instance variables
	RBuf iName;
	TInt iPrice;
	};
```

You will already know the structure of the code in `PrintEntry()` from previous modules. Note that the formatting process of the text inserts the text at a position that is marked with `%S`.

```
void CShopEntry::PrintEntry()
	{
	_LIT(KShopEntry, "Item: %S, Price: %d\n");
	console->Printf (KShopEntry, &iName, iPrice);
	}
```

Constructor

The first thing that is needed to actually use the `CShopEntry` class is to add a constructor and a destructor. Define them in the first public section of the `CShopEntry` class definition. We will pass the name of the item and the price to the class right in the constructor.

```
class CShopEntry
	{
public:
	// Constructors & Destructors
	CShopEntry(const TDesC& aName, TInt aPrice);
	~CShopEntry();
```

Note that the name is passed as `const TDesC&`, which is essentially a reference to any type of string, no matter if it's heap-, stack- or ROM-based. Descriptors will be covered in detail in the according module.

Also make sure that you get the naming conventions right. Parameters should have a preceding `a`, while instance variables are marked with an `i`. This makes it easier to differentiate those two, helps preventing errors and understanding the source code better.

The next task is to actually implement those two functions:

```
CShopEntry::CShopEntry(const TDesC& aName, TInt aPrice) :
	iPrice(aPrice)
	{
	iName.CreateL (aName);
	}
```

For the `TInt` parameter of the constructor, the class is already initialized with the parameter (`aPrice`) during its initialization. In the actual source code of the constructor, the `iName` descriptor is allocated; it copies the text to the heap memory that is automatically reserved an managed by the `RBuf`.

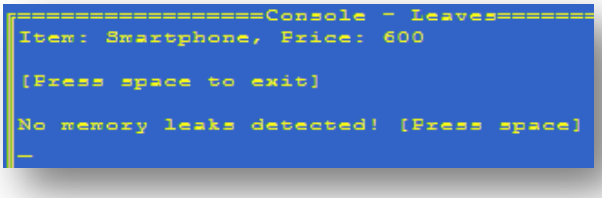
The destructor has to clean up the memory that is owned by the class instance. As we know, R type classes have to be closed, to make them free their memory / close their connections. Therefore, we call `iName.Close()` to free the memory allocated by the descriptor:

```
CShopEntry::~CShopEntry()
	{
	iName.Close ();
	}
```

The code will generally work fine, but it is not safe – we will shortly see why. First, we need some testing code. Add this to the `MainL()`-function at the end of the file:

```
	CShopEntry* entry1 = new (ELeave) CShopEntry(KEntry1, 600);
	entry1->PrintEntry();
	delete entry1;
```

The code works fine and you should see the following output when you run the application:



Memory Leaks

It's ok that we didn't put the test instance (`entry1`) on the cleanup stack, as no function could cause a leave until the class is deleted again. However, we will now assume that the code between construction and destruction of the object could potentially cause a leave, forcing us to put the object on the cleanup stack.

The new code is therefore:

```
	CShopEntry* entry1 = new (ELeave) CShopEntry(KEntry1, 600);
	CleanupStack::PushL(entry1);
	entry1->PrintEntry();
	CleanupStack::PopAndDestroy(entry1);
```

The code would be fine – however, if you execute the application now, you will notice that there's a memory leak. If you use a debug run configuration, the application will quit. In release mode, you will

get a serious Windows error message, which reports that *Construction.exe* has caused an error and has to be shut down. In any case, you won't see the debug message telling you about no memory leaks that is still part of the framework.

The memory leak can only be caused by our `entry1` object, which is not properly cleaned up by the cleanup stack. The reason is that the class is not derived from `CBase`, so the cleanup stack doesn't know that it should call the destructor of the class when cleaning up. Fix this mistake and the cleanup stack will work!

In case you're wondering – it would also be possible to push the object onto the cleanup stack through `CleanupDeletePushL(entry1);`. This would tell the cleanup stack to call delete `entry1` even if it isn't derived from `CBase`. However, as our object is indeed a C type class, it should be derived from `CBase` in any case!

Safe Construction

The current code has another problem. To reveal it, modify the constructor of the `CShopEntry`-class to simulate a memory allocation error for the descriptor using the `__UHEAP_FAILNEXT(1);` macro:

```
CShopEntry::CShopEntry(const TDesC& aName, TInt aPrice) :
	iPrice(aPrice)
	{
	__UHEAP_FAILNEXT(1);
	iName.CreateL (aName);
	}
```

When you execute the application now, you will notice a memory leak again. The reason is that the object itself has already been allocated in memory at the time when the code inside of the constructor (→ `iName.CreateL()`) leaves. Because of this, the constructor is left immediately and the memory allocated for the object is orphaned.

Therefore, no code within a C++ constructor should ever leave. This is not only true for memory allocations, but also for any other applications that could cause a leave – for example, opening a file.

Two Phase Construction

This leads us to the two phase construction scheme of Symbian OS. It splits the constructor into two phases:

- ***Constructor:*** The first phase is the normal constructor, which can still be used to assign arguments to instance variables, invoke functions that cannot leave, etc.

- ***Initialization:*** The second phase is usually a class method called `ConstructL()`. It completes the construction of the object and may perform operations that can leave.

The idea is that object construction is only complete when both phases are completed. Therefore, the instance of the object shouldn't exist when the constructor was executed but initialization fails.

We'll now adapt our application to use two phase construction:

1. Add the `ConstructL()` function to the `CShopEntry` class definition (public, for now) and adapt the parameter of the constructor so that it only requires the price.

```
class CShopEntry : public CBase
	{
public:
	// Constructors & Destructors
	CShopEntry(TInt aPrice);
	~CShopEntry();
	void ConstructL(const TDesC& aName);
```

2. Implement `ConstructL()` and move the `aName` parameter plus the `iName` initialization from the constructor to the new `ConstructL()` function.

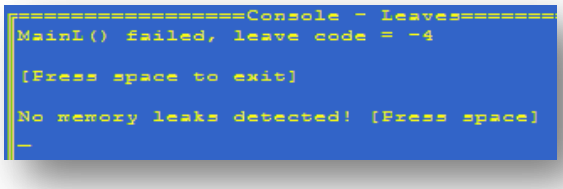
```
CShopEntry::CShopEntry(TInt aPrice) :
	iPrice(aPrice)
	{
	}

void CShopEntry::ConstructL(const TDesC& aName)
	{
	//__UHEAP_FAILNEXT(1);
	iName.CreateL (aName);
	}
```

3. Adapt the test code to push the object onto the cleanup stack after the constructor has successfully completed and then immediately call `ConstructL()`.

```
	CShopEntry* entry1 = new (ELeave) CShopEntry(600);
	CleanupStack::PushL(entry1);
	entry1->ConstructL(KEntry1);
	entry1->PrintEntry();
	CleanupStack::PopAndDestroy(entry1);
```

If you activate the `__UHEAP_FAILNEXT(1)`-macro in the `ConstructL()`-function, the leave will of course still be sent out as the allocation of `iName` fails. The framework therefore displays the according error message. The great thing is that the memory leak is gone – so, even though memory allocation during object construction failed, our code is still safe from memory leaks. After you're finished with testing, remove the macro again.



NewL and NewLC

Of course, it's not really convenient to call `ConstructL()` every time you want to create an instance of an object. Also, it might lead to errors if you provide your code to someone else and he/she forgets to do the two phase construction manually.

Because of this, classes that require two phase construction (e.g. they allocate memory during their construction) usually provide comfort functions called `NewL()` and `NewLC()`. Those functions encapsulate the whole object creation process. The difference is that `NewLC()` leaves the object on the cleanup stack, which is important if you plan to use it as a local variable. `NewL()` is mainly intended for instance variables, where the object is not left on the cleanup stack.

To implement those two methods, you should make the standard C++ constructor and `ConstructL()` private, so that the caller can no longer instantiate objects except through `NewL(C)`.

Both `NewL()` and `NewLC()` are static functions that return an instance of the class. Their definition reflects this behaviour; the top part of the class definition should therefore look like this:

```
class CShopEntry : public CBase
	{
public:
	// Constructors & Destructors
	static CShopEntry* NewLC(const TDesC& aName, TInt aPrice);
	static CShopEntry* NewL(const TDesC& aName, TInt aPrice);
	~CShopEntry();

private:
	// Private construction code
	CShopEntry(TInt aPrice);
	void ConstructL(const TDesC& aName);
```

The implementation of `NewLC()` is exactly like the manual object construction that we have done in the test code up to now. It first calls the constructor of the object, pushes it onto the cleanup stack and calls `ConstructL()`. Finally, the fully created instance is returned to the caller. The object is left on the cleanup stack, as is also indicated by the trailing `C` of the function name.

```
CShopEntry* CShopEntry::NewLC(const TDesC& aName, TInt aPrice)
	{
	CShopEntry* self = new (ELeave) CShopEntry(aPrice);
	CleanupStack::PushL (self);
	self->ConstructL (aName);
	return self;
	}
```

As the only difference between `NewL()` and `NewLC()` is whether the new instance of the object is left on the cleanup stack, their code is rather similar. Therefore, `NewL()` will use the same code as `NewLC()` and just pops the object from the cleanup stack after construction – without destroying it!

```
CShopEntry* CShopEntry::NewL(const TDesC& aName, TInt aPrice)
	{
	CShopEntry* self = CShopEntry::NewLC (aName, aPrice);
	CleanupStack::Pop (self);
	return self;
	}
```

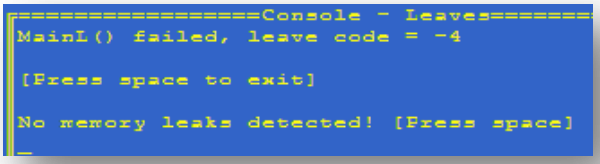
As the constructor is now private, we have to adapt the test code to use `NewLC()` instead. The construction code gets a lot shorter and easier:

```
	CShopEntry* entry1 = CShopEntry::NewLC(KEntry1, 600);
	entry1->PrintEntry();
	CleanupStack::PopAndDestroy (entry1);
```

The good thing is that the code stays safe even if the first object construction of the following updated test code succeeds, but the second produces an error:

```
CShopEntry* entry1 = CShopEntry::NewLC(KEntry1, 600);
__UHEAP_FAILNEXT(1);
CShopEntry* entry2 = CShopEntry::NewLC(KEntry2, 450);
entry1->PrintEntry();
entry2->PrintEntry();
CleanupStack::PopAndDestroy(2);
```

The result is still a memory leak safe code:



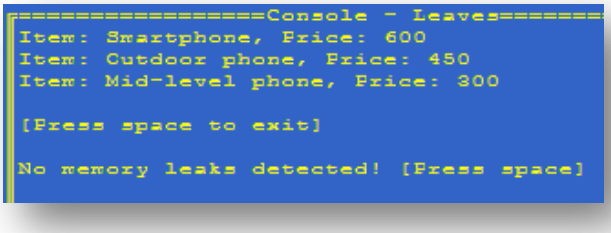
If we'd create the entry as an instance variable or if there is no potentially leaving code between the object construction and destruction, you can also use the `NewL()` function.

If your final test code looks like this...

```
CShopEntry* entry1 = CShopEntry::NewLC(KEntry1, 600);
CShopEntry* entry2 = CShopEntry::NewLC(KEntry2, 450);
entry1->PrintEntry();
entry2->PrintEntry();
CleanupStack::PopAndDestroy(2);

CShopEntry* entry3 = CShopEntry::NewL(KEntry3, 300);
entry3->PrintEntry();
delete entry3;
```

... the output should be like this:



Glossary

You might encounter the following definitions, which will not be familiar to you as of now. The following table lists them, along with a short description:

Term	Description
RBuf	One of the Symbian OS descriptors, available since Symbian OS 8. Descriptors are the Symbian OS way of C++ strings.
_LIT	Defines a literal = a fixed descriptor (string) that's stored directly in the compiled application.
TDesC	Base class for descriptors (= strings). Allows generic parameters, independent on the real type of the descriptor.