



Memory Management is an important issue on Symbian OS. This page describes issues related to memory leaks.

## Two-phase construction prevents memory leaks

Because of the Symbian-specific error handling mechanism, leaving, compound classes should be constructed in two phases to prevent memory from leaking. Let's consider the following C++ kind of construction:

```
CExample* example = new CExample();
if (example) {
    // Do something with example
}
```

First of all, on Symbian OS it is possible to use an overloaded version of new operator, *new(ELeave)*, which leaves if there is not enough memory available. Thus it is possible to use the returned pointer without further testing that the memory allocation was successful, which simplifies the code.

```
CExample* example = new(ELeave) CExample();
// Do something with example
```

Still, whether you use new or new(ELeave), you have to pay special attention to memory usage because of the leaving mechanism of Symbian. If the constructor leaves, the memory already allocated by the new operator will leak, because there is no way of cleaning it up. Hence, C++-type constructors and destructors should never leave.

*Note: A key rule of Symbian OS memory management: no code within a C++ constructor (or destructor) should ever leave.*

This problem can be avoided by breaking construction code into two phases:

In the first phase the code that cannot leave is executed. In practice this means calling base-class constructors, invoking functions that cannot leave and so on. This part is typically done in a basic constructor of the class. (Naturally, if the constructor cannot leave in the first place, it is appropriate to use mere new(ELeave) operator instead of the two-phase construction.)

In the second phase any part of the construction that might leave is performed. In practice this includes allocating memory, using resources such as files, invoking functions that might leave and so on. This part is typically done in a class method called *ConstructL*.

## Memory\_leaks

The two phases are performed separately and between them the object, allocated and constructed by the new operator, is pushed onto the cleanup stack. If a leave occurs during the second phase the cleanup stack calls the destructor to free any resources that have been successfully allocated and destroys the memory allocated for the object itself. A commonly used pattern in Symbian OS code is to provide a static function which wraps both phases of construction. This function is typically called *NewL*, or *NewLC* if it leaves the constructed object on the cleanup stack.

Here's an example of the two-phase construction:

```
class CExample : public CBase
{
public:
    static CExample* NewL();
    static CExample* NewLC();
    ~CExample();
private:
    CExample();           // Guaranteed not to leave
    void ConstructL();   // Second phase construction code, may leave
};
```

The non-leaving constructor *CExample* and second-phase constructor *ConstructL* have been made private so a caller cannot instantiate objects of the class except through *NewL* (or *NewLC*). If you intend your class to be subclassed, you should make the default constructor protected rather than private so the compiler may construct the deriving classes.

Typical implementations of *NewL* and *NewLC* may be as follows:

```
CExample* CExample::NewLC()
{
    CExample* me = new(ELeave) CExample(); // First phase construction
    CleanupStack::PushL(me);
    me->ConstructL(); // Second phase construction
    return (me);
}
```

```
CExample* CExample::NewL()
{
    CExample* me = CExample::NewLC();
    CleanupStack::Pop(me);
    return (me);
}
```

In addition to object construction, you should consider destruction code carefully.

## Issues related to destructors

A destructor must be coded to release all the resources that an object owns. However, if the second-phase construction fails, the cleanup stack may call the destructor to cleanup partially constructed objects. Thus, the destructor code cannot assume that the object is fully initialized, and you should beware of calling functions on pointers which may not yet be set to point to valid objects.

Here's an example of a safe destructor:

```
CExample::~CExample()
{
    if (iMyAllocatedMember) {
        iMyAllocatedMember->DoSomeCleanupPreDestruction();
        delete iMyAllocatedMember;
    }
}
```

## Links

[Memory Management](#)

[Debugging techniques](#)

[Cleanup stack support for non-CBase classes](#)

[Leave](#)