Symbian OS

Memory Management

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.

Schedule

- Leaves, Panics and TRAP(D)
- Cleanup stack
- Object construction using ELeave
- Two-phase construction
- Debugging tools

Handling problems

Leaves & Panics

Exceptions – Java

- **Try & Catch** for handling exceptions
- Functions can throw "**Exceptions**"

Calling function

```
Try {
    int x = Integer.parseInt("1234");
} catch (NumberFormatException e) {
    System.out.println("Unable to convert this
        String to a number.");
}
```

Integer Class

```
…
static int parseInt throws
NumberFormatException {
…
}
…
```

Leave – Symbian

The TRAP(D) macros are defined in e32cmn.h

- **TRAP(D)** catches exceptions ("**Leave**")
- Functions send out leave
- Function name marked by an **L**-suffix

Main-Function

```
TRAPD(err, DoExampleL());
if (err)
   {
   console->Printf(KTxtFailed, err);
   }
```

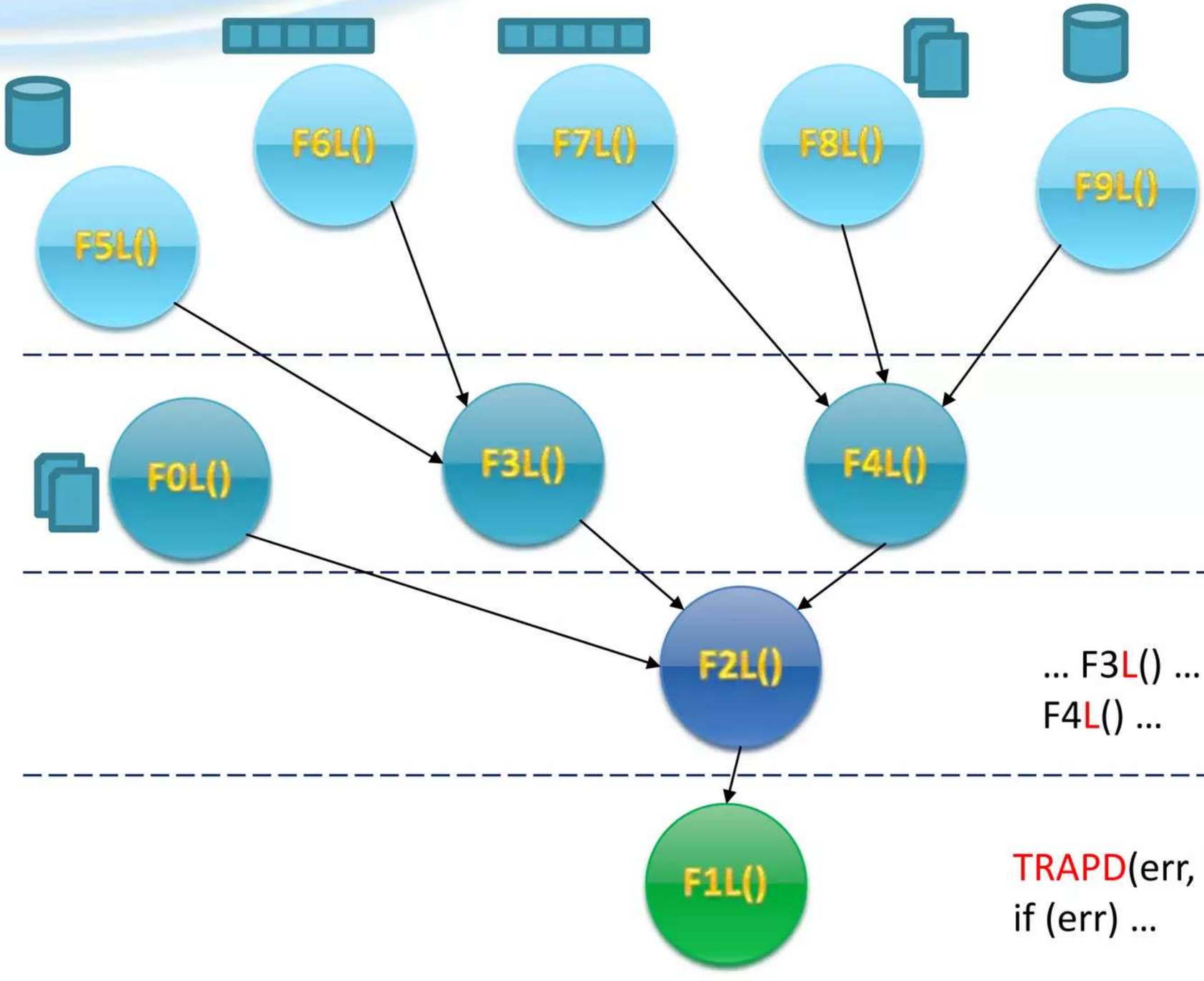
TRAPD-Makro declares **err** as **TInt** and = **KErrNone**

DoExampleL()-Function

```
void DoExampleL()
   {
   RFs fsSession;  // Connect to the file server
   User::LeaveIfError(fsSession.Connect());
   // ...
   fsSession.Close();
   }
```

Leaves if the Connect() function does not return KErrNone

Central Exception Handling



New (ELeave) …
… NewL() …
… User::Leave() …
… ConstructL() …

F5L() …
… F6L() ….
… F8L() ….

… F3L() …
F4L() …

TRAPD(err, F2L());
if (err) …

Handling Leaves

- Try to implement central leave-handling
- If leave not handled by your code → **error-message shown by the UI-framework!**
- **Therefore:** Only handle leaves yourself if they influence your application

When can a function leave?

1. **Caused by your own code:**
   - `User::Leave()`,
     `User::LeaveIfError()`,
     `User::LeaveNoMemory()` or
     `User::LeaveIfNull()`
2. **Failed object-construction**
   - when using the “`new (ELeave)`”-operator
3. **Calling a function that potentially causes a leave**
   - e.g. `x->DoSomethingL()`

Details: Causing a Leave

- **User::Leave(TInt aReason)**
  - Error code (aReason) = value that will be received by TRAP
  - Causes leave in any case
- **User::LeaveIfError(TInt aReason)**
  - Causes leave if parameter is negative, e.g.:

```
TInt foundAt = iElementArray.Find(example, aRelation);
User::LeaveIfError(foundAt); // Leaves if foundAt == KErrNotFound (-1)
```

- **User::LeaveNoMemory()**
  - Is the same as: User:Leave(KErrNoMem);
- **User::LeaveIfNull(TAny* aPtr)**

TRAP / TRAPD

- Two trap harness macros:
  - **TRAP**: declares the variable in which the leave code is returned
  - **TRAPD**: declare a `TInt` variable yourself

```
TRAPD(result, MayLeaveL());
if (KErrNone!=result)
...
is equivalent to:
TInt result;
TRAP(result, MayLeaveL());
if (KErrNone!=result)
...
```

- If a leave occurs inside `MayLeaveL()`, which is executed inside the harness, the program code will return immediately to the TRAP harness macro
- The variable result will contain the error code associated with the leave or will be `KErrNone` if no leave occured

Tips

- **TRAPs are expensive**
  - Don’t use several TRAPs right after each other
  - **Instead:**
    - Make the function leave (append an L to the function name)
    - Handle all errors in a single TRAP-call one level above!
- In UI apps, many leaves don’t have to be handled by you
  → they can go up to the topmost level (= Active Scheduler)

```
void InsertData()
  {
  TInt err;
  TRAP(err, iData->InsertL(23));
  if (err) { ... }
  TRAP(err, iData->InsertL(24));
  if (err) { ... }
  }
```

prefer

```
void HandleCommand()
  {
  TRAPD(err, InsertDataL());
  if (err) { ... }
  }

void InsertDataL()
  {
  iData->InsertL(23));
  iData->InsertL(24));
  }
```

Panics

- … **cannot be caught and handled!**
- Terminates thread (= usually the whole application)
- Use them **for checking code** logic only
- Can also be sent out by the system for critical errors
- **If a panic happens:**
  - Make sure you fix it, as you can't handle it!

```
// Stray signal detected!
_LIT(KMsgStraySignal, "Stray signal\n");
User::Panic(KMsgStraySignal, 1);          // Panic with code 1
```

Stack, Heap and the Cleanup Stack

Resource Handling

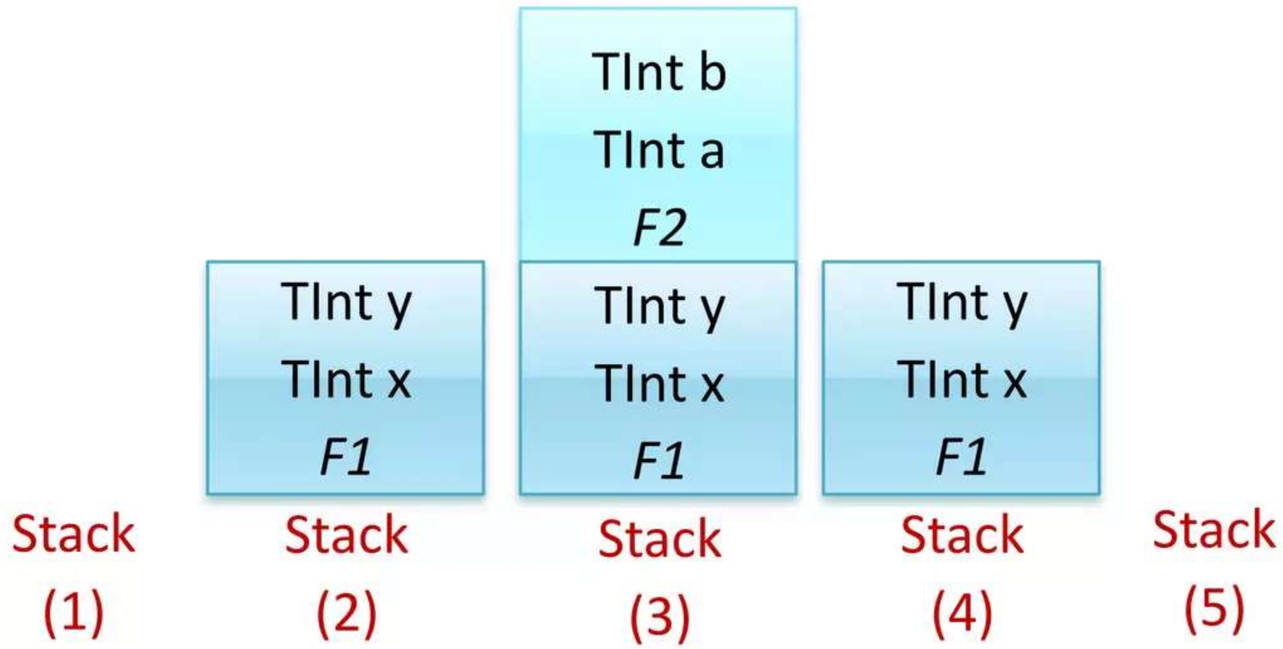
Practice in a Nutshell

- All strategies are present in this function:

```
CClass* CClass::NewL(TInt aInt, CBase& aObj)
{
        CClass* self = new (ELeave) CClass(aInt);
        CleanupStack::PushL(self);
        self->ConstructL(aObj);
        CleanupStack::Pop(self);
        return self;
}
```

Recap: Stack

```
// Stack (1)
void CTest::F1()
{
        TInt x = 0;
        TInt y = 1;          } Stack (2)
        for(x = 0; x < 10; x++)
                y++
        y = y + F2();
        // Stack (4)
}
// Stack (5)

TInt CTest::F2()
{
        TInt a = 2;
        TInt b = 1;
        return a + b;   // Stack (3)
}
```



Recap: Heap

- **Use for:**
  - Objects that need a **lot of memory**
    (→ stack-size is limited!)
  - Required amount of **memory only known at runtime**
- Scope of heap-objects is not limited to a function
  - Use **pointers** to pass to other functions/objects

Motivation

- Symbian OS designed to run for many years
  - → No opportunity to reclaim leaked memory through regular restarts!
- Small memory leaks accumulate over time
- We only have a small memory to start with
- **Therefore: Simply write perfect, leak-free code!**

Motivation – How?

- **By keeping track of all allocated objects!**
- By making sure:
  - All heap-objects are pointed to by at least one pointer, all the time – even in the constructor of an object!
  - Free heap memory as soon as possible after use.

Safeguard

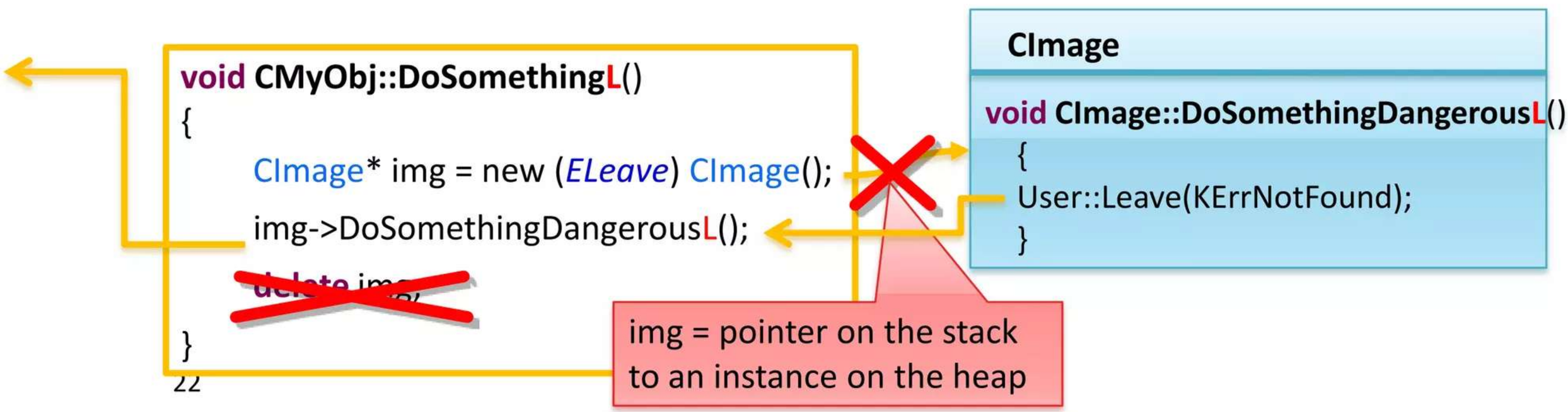
The Cleanup Stack

Resource Handling – Rule 1

Every **local** pointer to an instance of a **heap-based object** also has to be pushed on the **cleanup stack**, if there‘s the risk that the pointer gets lost because of a **leave**.

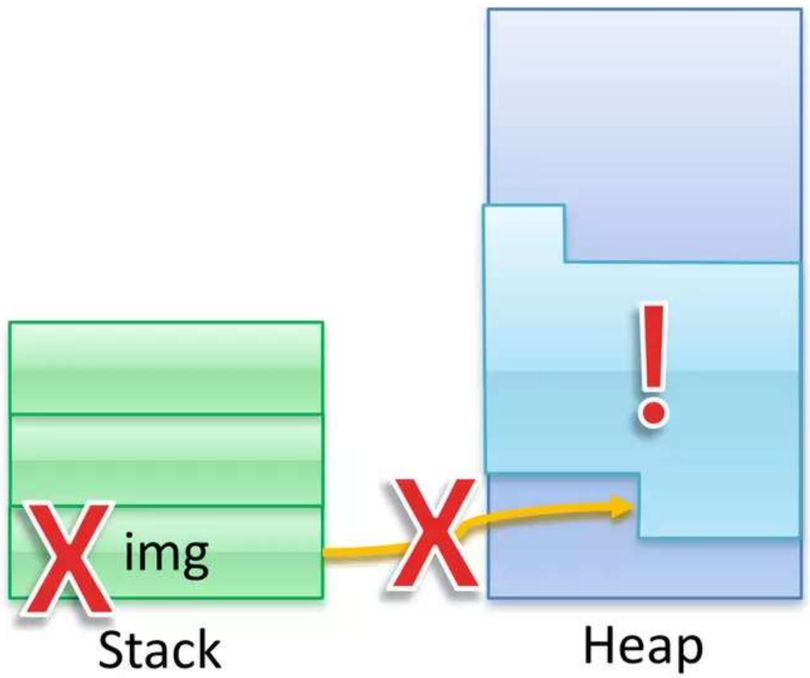
Cleanup Stack

- **Situation:** function creates local heap-object
- Before code gets to the delete-statement: error
  - Function is left (Leave)
  - Pointer-address on the stack is freed
  - Object itself is orphaned → **memory leak!**



Cleanup Stack

- Memory situation if a leave occurs:



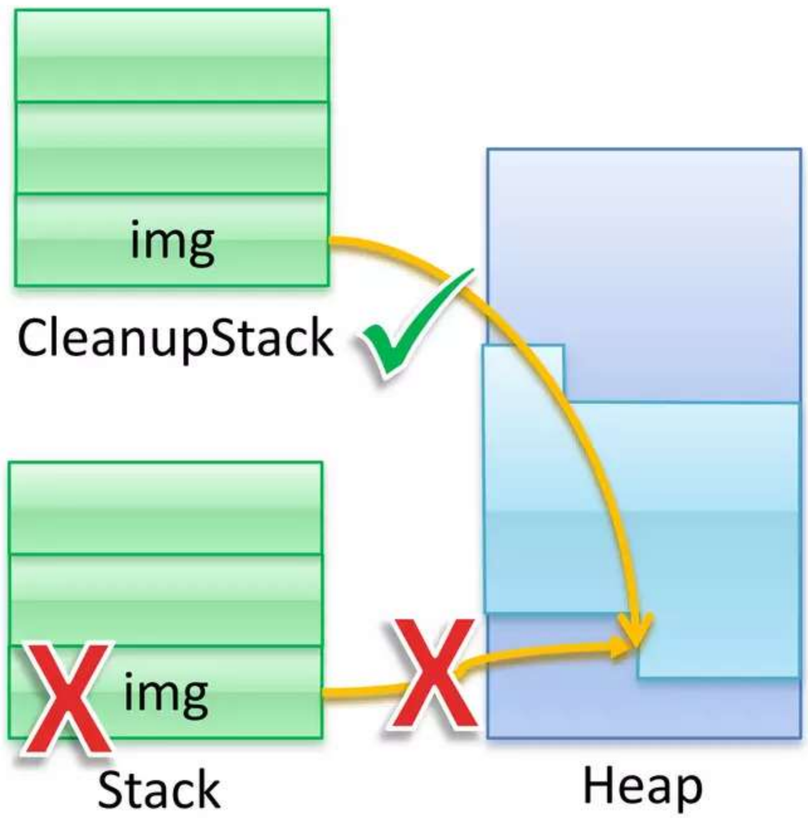
Object stays on the heap;
Pointer to delete the instance is lost
→ **memory leak**

Cleanup Stack

- **Solution:** Cleanup Stack

```
void CMyObj::DoSomethingL()
{
        CImage* img = new (ELeave) CImage();
        CleanupStack::PushL(img);
        img->DoSomethingDangerousL();
        CleanupStack::PopAndDestroy();
}
```

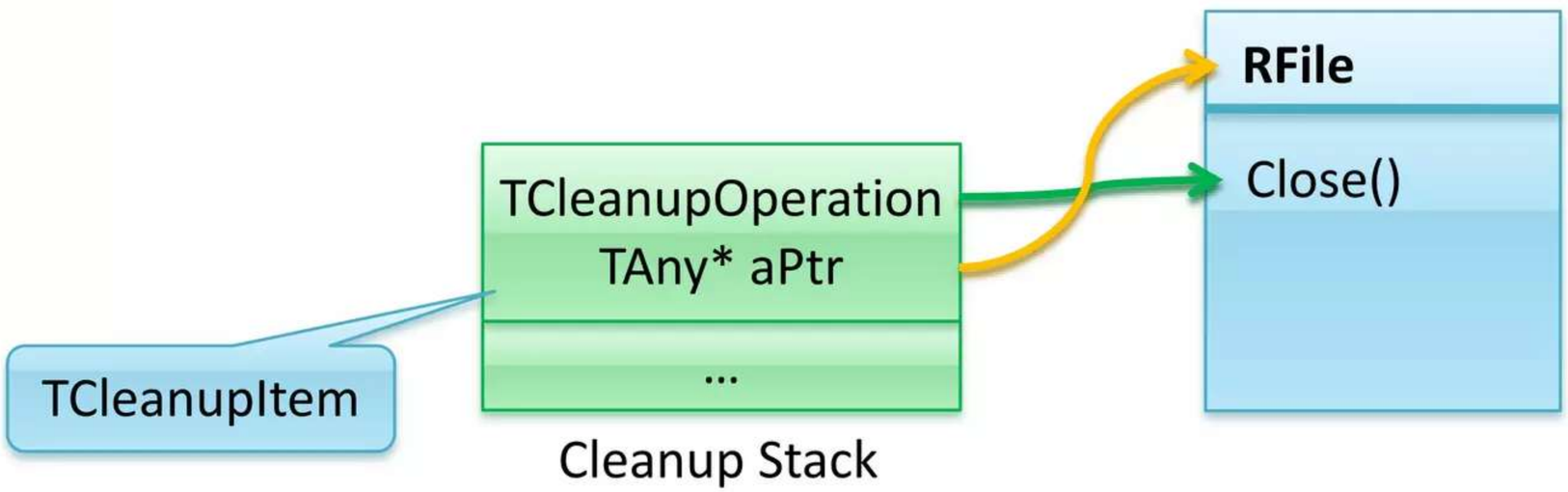
When no leave occurs: object still has to be deleted by you + removed from the cleanup stack!



Cleanup stack saves a second pointer to the object on the heap.

Advanced Cleanup

- For resources that have to be closed / freed (e.g.: files, sockets, …)
  - **Close():** Cleanup**Close**PushL(T&)
  - **Release():** Cleanup**Release**PushL(T&)
  - **Destructor:** Cleanup**Delete**PushL(T*)



Advanced Cleanup – Example

```
void CMyClass::TransferDataL()
   {
   RSocketServ socketServer;

   // Connect to SocketServer
   User::LeaveIfError( socketServer.Connect() );
   // Make sure Close() is called at the end
   CleanupClosePushL( socketServer );

   // …

   CleanupStack::PopAndDestroy();  //  socketServer
   }
```

Cleanup Stack and Ownership Transfer

- It should never be possible for an object to be deleted more than once!

```
void TransferOwnershipExampleL()
   {
   // The stack variable ptr points to memory allocated on the heap
   CItem* ptr = new (ELeave) CItem();
   // The following function may leave -> place the pointer on the
   // cleanup stack, so that the heap memory is freed in case
   // AppendL() leaves.
   CleanupStack::PushL(ptr);
   // iItemPtrArray takes ownership of the CItem object.
   // This could fail, as it needs to allocate a new slot to store the pointer,
   // therefore the object was placed on the cleanup stack in advance
   iItemPtrArray->AppendL(ptr);
   // iItemArray now owns the heap object, so ptr may be safely popped off the stack.
   // It shouldn't be destroyed, as this would make the item in iItemPtrArray invalid!
   CleanupStack::Pop(ptr);
   }
```

Practice in a Nutshell

- Strategies explained up to now:

```
CClass* CClass::NewL(TInt aInt, CBase& aObj)
{
        CClass* self = new (ELeave) CClass(aInt);
        CleanupStack::PushL(self);
        self->ConstructL(aObj);
        CleanupStack::Pop(self);
        return self;
}
```

Resource Handling – Rule 2

Never push instance variables on the cleanup stack!

The owning class is also on the cleanup stack somewhere (at least indirectly). This would lead to a double deletion of the object pointed to by the instance variable → Panic!

Coding Error Example

1. **CSimple** is created and pushed onto the cleanup stack as the next function may leave

```
CSimple* simple = new (ELeave) CSimple();

CleanupStack::PushL(simple);
```

```
class CSimple : CBase
    {
public:
    ~CSimple();
    void MayLeaveFuncL();
private:
    void PrivateMayLeaveL();
    CItem* iItem;
    };
```

2. A leaving method is called on simple

```
TRAPD(res,simple->MayLeaveFuncL());
...
```

```
void CSimple::MayLeaveFuncL()
    {
    iItem = new (ELeave) CItem();
    CleanupStack::PushL(iItem);

    PrivateMayLeaveL();
    CleanupStack::Pop(iItem);
    }
```

3. The member variable is pushed onto the clean up stack (oops!)

4. What happens if a leaves occurs?

5. The TRAP does the right thing and clears the cleanup stack; i.e. **CSimple::iItem** is deleted

```
...
```

6. The code logic completes with the popping and deleting of the **simple** object.

```
CleanupStack::PopAndDestroy(simple);
```

BUT this calls the **CSimple** destructor, which deletes the **iItem** which has already been deleted by the TRAP

```
CSimple::~CSimple
    {
    delete iItem;
    }
```

7. PANIC!

Example taken from the Symbian Academy slides

Leave-handling during

Object Construction

New Objects

- **new**-operator allocates memory and runs **constructor**
- Returns **null-pointer** if object creation fails (e.g. not enough memory)
- → **manual test** required to see if it was successful – no automated leave!

```
void CMyObj::DoSomethingL()
   {
   CImage* img = new CImage();
   if (img)
      {
      CleanupStack::PushL(img);
      img->DoSomethingDangerousL();
      CleanupStack::PopAndDestroy();
      }
   else
      {
      User::LeaveNoMemory();
      }
   }
```

New Objects – ELeave

```
void CMyObj::DoSomethingL()
   {
   CImage* img = new CImage();
   if (img)
      {
      CleanupStack::PushL(img);
      img->DoSomethingDangerousL();
      CleanupStack::PopAndDestroy();
      }
   else
      {
      User::LeaveNoMemory();
      }
   }
```

identical

```
void CMyObj::DoSomethingL()
   {
   CImage* img = new (ELeave) CImage();
   CleanupStack::PushL(img);
   img->DoSomethingDangerousL();
   CleanupStack::PopAndDestroy();
   }
```

new-Operator overloaded with **ELeave**: automated leave if there's not enough memory!

Practice in a Nutshell

- Strategies explained so far:

```
CClass* CClass::NewL(TInt aInt, CBase& aObj)
{
        CClass* self = new (ELeave) CClass(aInt);
        CleanupStack::PushL(self);
        self->ConstructL(aObj);
        CleanupStack::Pop(self);
        return self;
}
```

The ultimate combination …

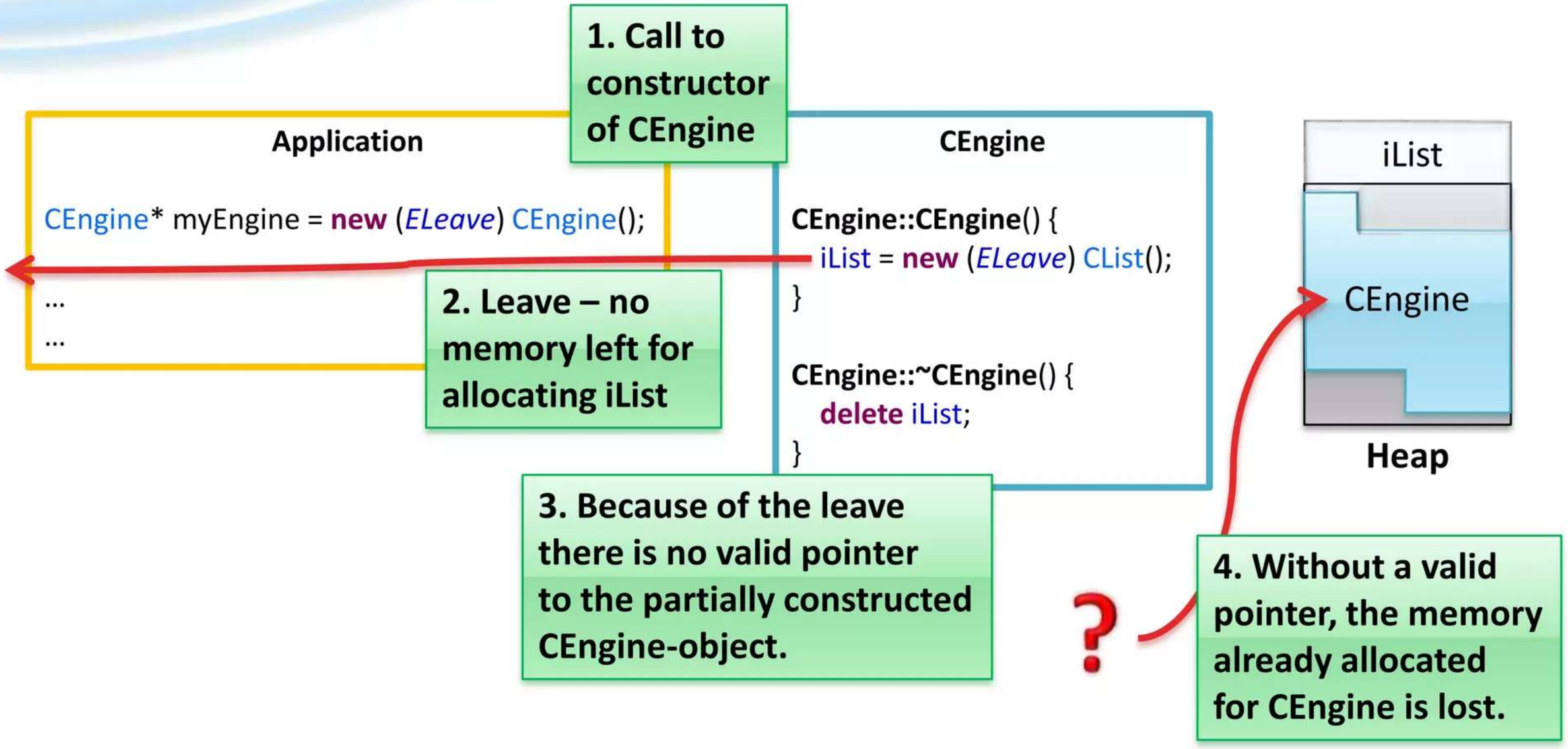
Two-Phase Construction

Resource Handling – Rule 3

Neither a constructor nor a destructor may cause a leave!

The destructor must not assume that the object was fully initialized.

Leaves in the Constructor



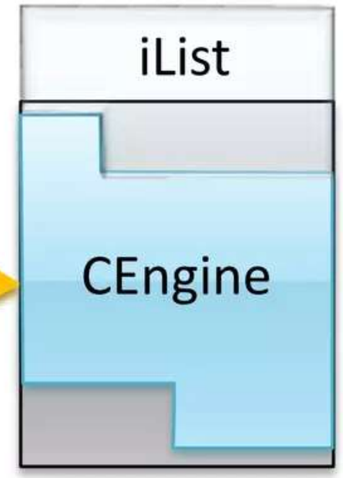
Solution: Two-phase Construction

Application

```
CEngine* myEngine = new (ELeave) CEngine();
CleanupStack::Push(myEngine);
myEngine->ConstructL();

...
// Note that the following line won't be
// reached in case of a leave!
CleanupStack::PopAndDestroy(myEngine);
```

CEngine

```
CEngine::CEngine() {
   // 1. Phase
}

CEngine::ConstructL() {
   // 2. Phase
   iList = new (ELeave) CList();
}

CEngine::~CEngine() {
   delete iList;
}
```



Heap

Object is fully allocated and on the cleanup stack
→ the destructor can be executed, the class is deleted by the cleanup stack
→ all memory is properly cleaned up in case of a leave!

Simplification: Two-phase Construction

- Less complicated creation of objects:

Application – Local Variable

```
CEngine* myEngine = CEngine::NewLC();
...
CleanupStack::PopAndDestroy(myEngine);
```

or

Application – Instance variable

```
iMyEngine = CEngine::NewL();

// Destructor:
delete iMyEngine;
```

CEngine

```
CEngine* CEngine::NewLC() {
   CEngine* self = new (ELeave) CEngine();      // 1. Phase
   CleanupStack::PushL(self);
   self->ConstructL();                          // 2. Phase
   return self;
}

CEngine* CEngine::NewL() {
   CEngine* self = CEngine::NewLC();
   CleanupStack::Pop(self);
   return self;
}
```

- Trailing **C** at the end of the function name:
  an object is left on the cleanup stack (as in `NewLC()`)

Practice in a Nutshell

- Strategies explained so far:

```
CClass* CClass::NewL(TInt aInt, CBase& aObj)
{
        CClass* self = new (ELeave) CClass(aInt);
        CleanupStack::PushL(self);
        self->ConstructL(aObj);
        CleanupStack::Pop(self);
        return self;
}
```

Derivation

- **Note:**
  - Some thoughts are necessary when deriving from a class that uses two-phase construction
  - See the literature or Symbian Academy-slides for details!

Resource Handling – Rule 4

If memory for a pointer (instance variable) is reallocated, set the old pointer to NULL beforehand.

NULL

```
void CEngine::DoStuffL(const TDesC& aName)
  {
  CElement* element = CElement::NewLC();
  TRAPD(err, element->SetNameL(aName));
  CleanupStack::PopAndDestroy(element);
  }
```

- Situation (**without NULL**):
  - AllocL() causes a leave, which propagates up …
  - Instance of CElement is deleted
  - Destructor of CElement is called
  - iName still points to already deleted memory
  - Deleted 2x (1) (2) → Panic

```
void CElement::SetNameL(const TDesC& aName)
  {
  delete iName; (1) // Deletes object, does not change pointer
  iName = NULL;      // Deletes pointer on stack
  iName = aName.AllocL();   // Re-Allocation
  }
```

```
void CElement::~CElement()
  {
  delete iName; (2)
  }
```

Note: delete does not delete a NULL pointer.

Summary

Summary

1. Catch leaves (= exceptions) with TRAP(D)
2. Use cleanup stack for local heap-based variables
3. Do not use the cleanup stack for instance variables
4. No leaves in constructors or destructors
5. Use two-phase construction for objects with data on the heap
6. Set a pointer to NULL before re-allocating memory

What's wrong?

```
CleanupStack::PushL(iList);
```

What's wrong?

```
CleanupStack::PushL(iList);
```

Never push instance variables on the cleanup stack!
Twice as safe isn't safe at all…

What's wrong?

```
CEngine* engine = new (ELeave) CEngine();
...
CleanupStack::PopAndDestroy(engine)
```

What's wrong?

```
CEngine* engine = new (ELeave) CEngine();
...
CleanupStack::PopAndDestroy(engine)
```

`engine` wasn't pushed on the cleanup stack!

`new (ELeave)` is only responsible for leaving if memory allocation for the object fails. `new (ELeave)` has nothing to do with the cleanup stack.

Solution:

Either: add the object to the cleanup stack:

```
CleanupStack::PushL(engine);
```

Or: don't use the cleanup stack for deleting – in case no leaving operation is called between creation and deletion of the `engine` object:

```
delete engine;
```

What's wrong?

```
void CGomokuViewGame::ChangeViewContextText(TInt aResourceId)
    {
    RBuf newText(iEikonEnv->AllocReadResourceL(resourceId));
    newText.CleanupClosePushL ();

    MQikViewContext* viewContext = ViewContext ();

    // Changing an already existing view context text
    viewContext->ChangeTextL (EGomokuViewContext, newText);

    CleanupStack::PopAndDestroy (1);       // newText
    }
```

What's wrong?

```
void CGomokuViewGame::ChangeViewContextText(TInt aResourceId)
    {
    RBuf newText(iEikonEnv->AllocReadResourceL(resourceId));
    newText.CleanupClosePushL ();

    MQikViewContext* viewContext = ViewContext ();

    // Changing an already existing view context text
    viewContext->ChangeTextL (EGomokuViewContext, newText);

    CleanupStack::PopAndDestroy (1);       // newText
    }
```

The trailing **L** of the function name is missing, as calls within the function can leave.

Solution:

void CGomokuViewGame::ChangeViewContextTextL(…)

On your way to perfection

Testing your Code

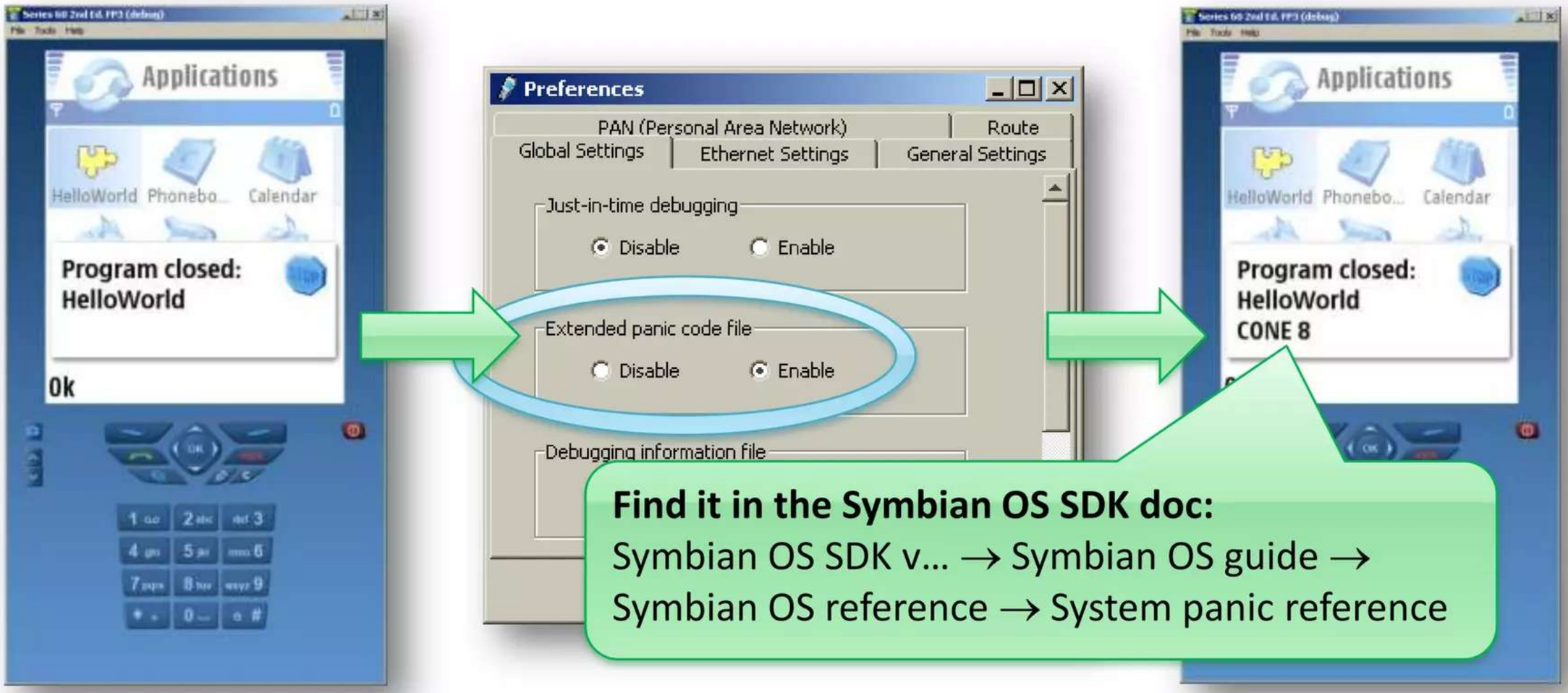
Macros for Testing

- Wrap code you want to test within `__UHEAP_MARK` and `__UHEAP_MARKEND`:

```
CClass* p1 = new (ELeave) CClass;
__UHEAP_MARK;                  // Mark start of test-area
CClass* p2 = new (ELeave) CClass;
CClass* p3 = new (ELeave) CClass;
__UHEAP_CHECK(2);              // 2 Objects (p2, p3) on the heap since the start-mark
__UHEAP_CHECKALL(3);               // In total 3 objects on the heap
delete p3;
_UHEAP_MARKEND;                // Result: p2 is still here – Memory Leak!
// or:  __UHEAP_MARKENDC(1);   // Expects one cell on the heap
```

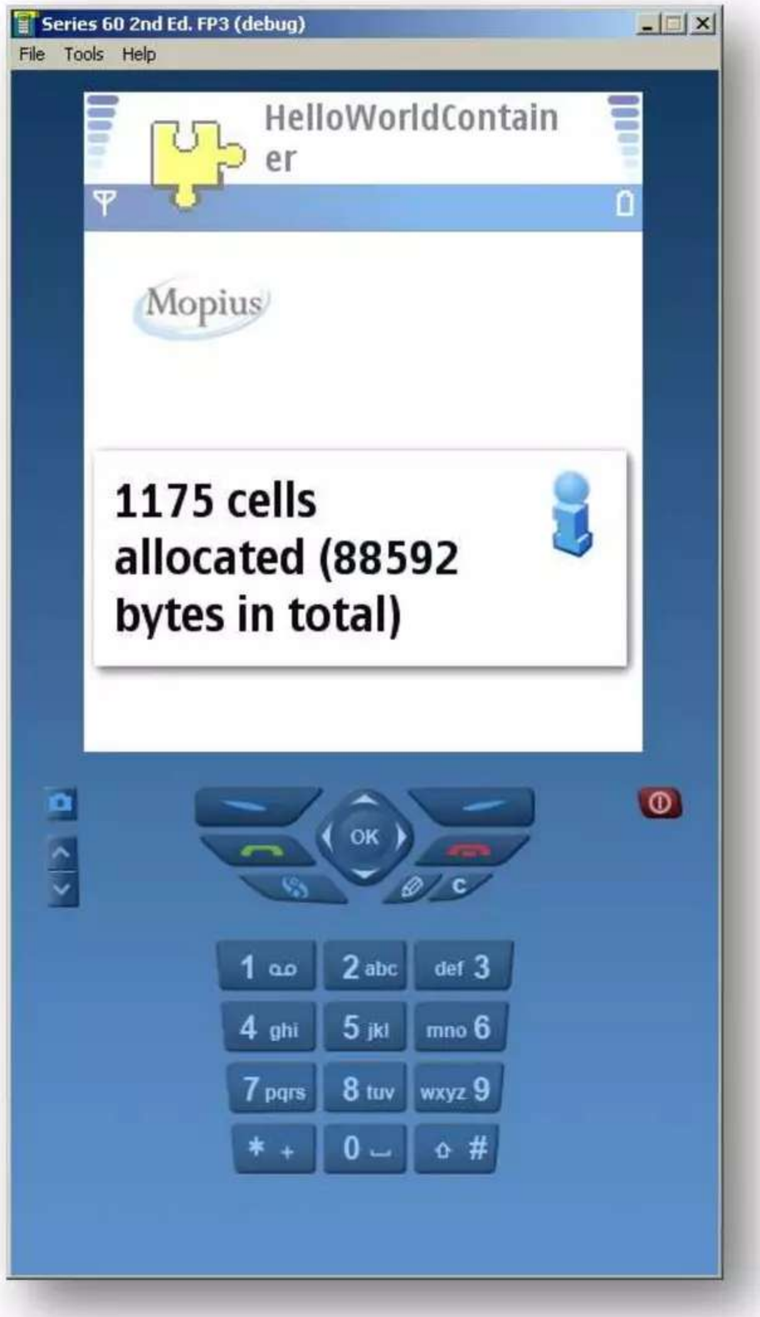
Finding Memory Leaks

- Emulator checks heap automatically if you **exit the program BEFORE closing the emulator window**!



Memory Information

- Show number of allocated cells:
  - **Ctrl+Alt+Shift+A**



Allocation Failure

- Intentional failing of memory allocation:
- **__UHEAP_SETFAIL(aType, aValue);**
  - **EDeterministic:** fail every nth request
  - **ERandom:** fail randomly once within a specified range – always using the same seed
  - **ETrueRandom:** random seed taken from system time

```
__UHEAP_SETFAIL(RHeap::EDeterministic, 2);
CObj* c1 = new (ELeave) CObj;          // will allocate
CObj* c2 = new (ELeave) CObj;          // will fail
CObj* c3 = new (ELeave) CObj;          // will allocate
CObj* c4 = new (ELeave) CObj;          // will fail
__UHEAP_RESET;                         // Deactivate
```

FAILNEXT

- Fails the next allocation request

```
CObj* c1 = new (ELeave) CObj;          // will allocate
__UHEAP_FAILNEXT(1);                   // fail next allocation
CObj* c2 = new (ELeave) CObj;          // will fail
```

Allocation Failure

- In the emulator, without writing code:
- **Heap Failure Tool**
  - **Activate:**
    Ctrl+Alt+Shift+P
  - **Deactivate:**
    Ctrl+Alt+Shift+Q
- Can fail heap, WindowServer-allocations or file-access

Try it for your own

… let’s move to the Challenges!