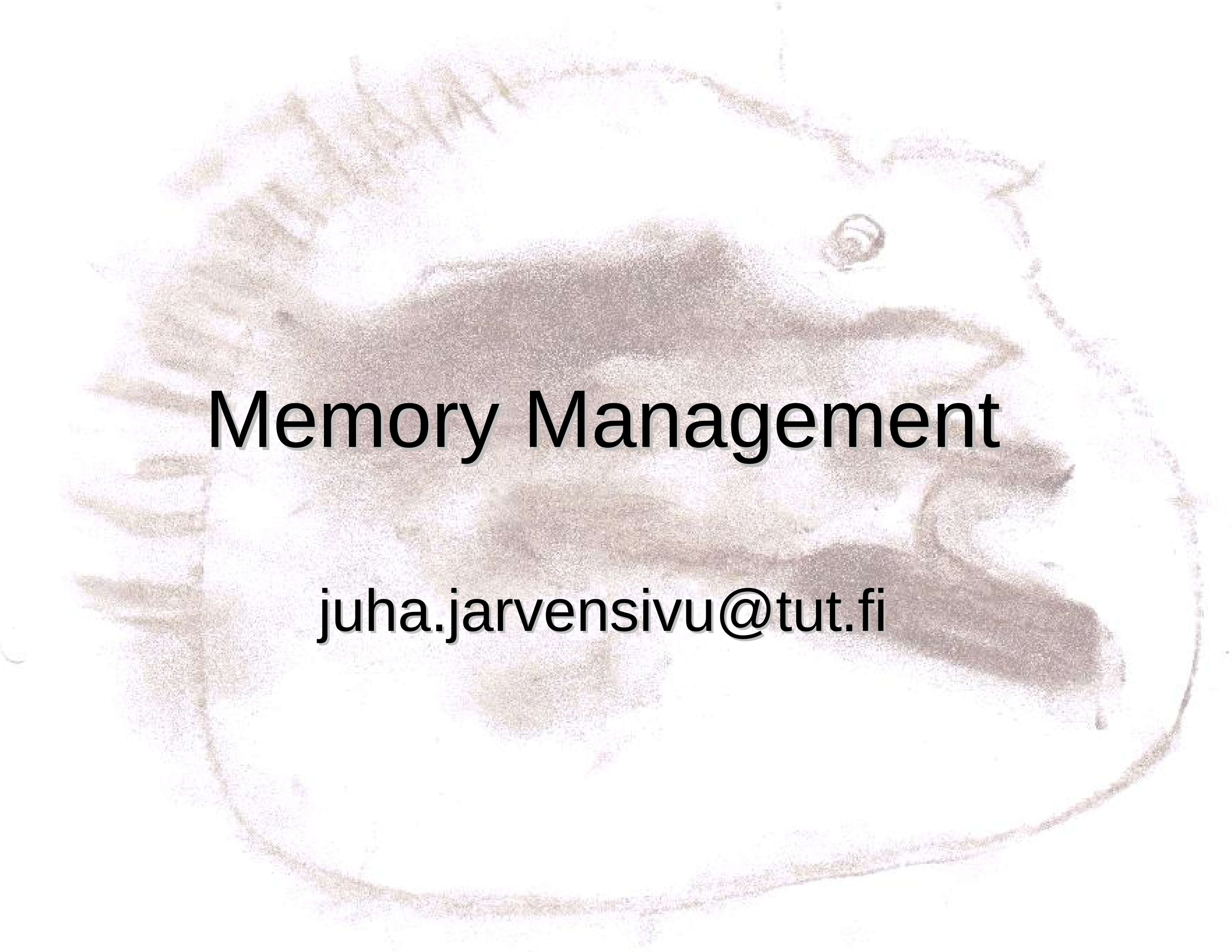


Memory Management

juha.jarvensivu@tut.fi

Content and goals

- *Basics of memory usage in mobile devices*
  - Static and dynamic allocation
  - Managing memory organization
- Memory management in Mobile Java
- Memory management in Symbian OS
- Summary

Content and goals

- Basics of memory usage in mobile devices
  - *Static and dynamic allocation*
  - Managing memory organization
- Memory management in Mobile Java
- Memory management in Symbian OS
- Summary

Static allocation

- Simplest case (in practice usually allocated in heap, but without programmer interventions)
- Variable is statically allocated to a certain location in the memory

```
int x;

int * pointer_to_static()
{
  static int y;
  return & y;
}
```

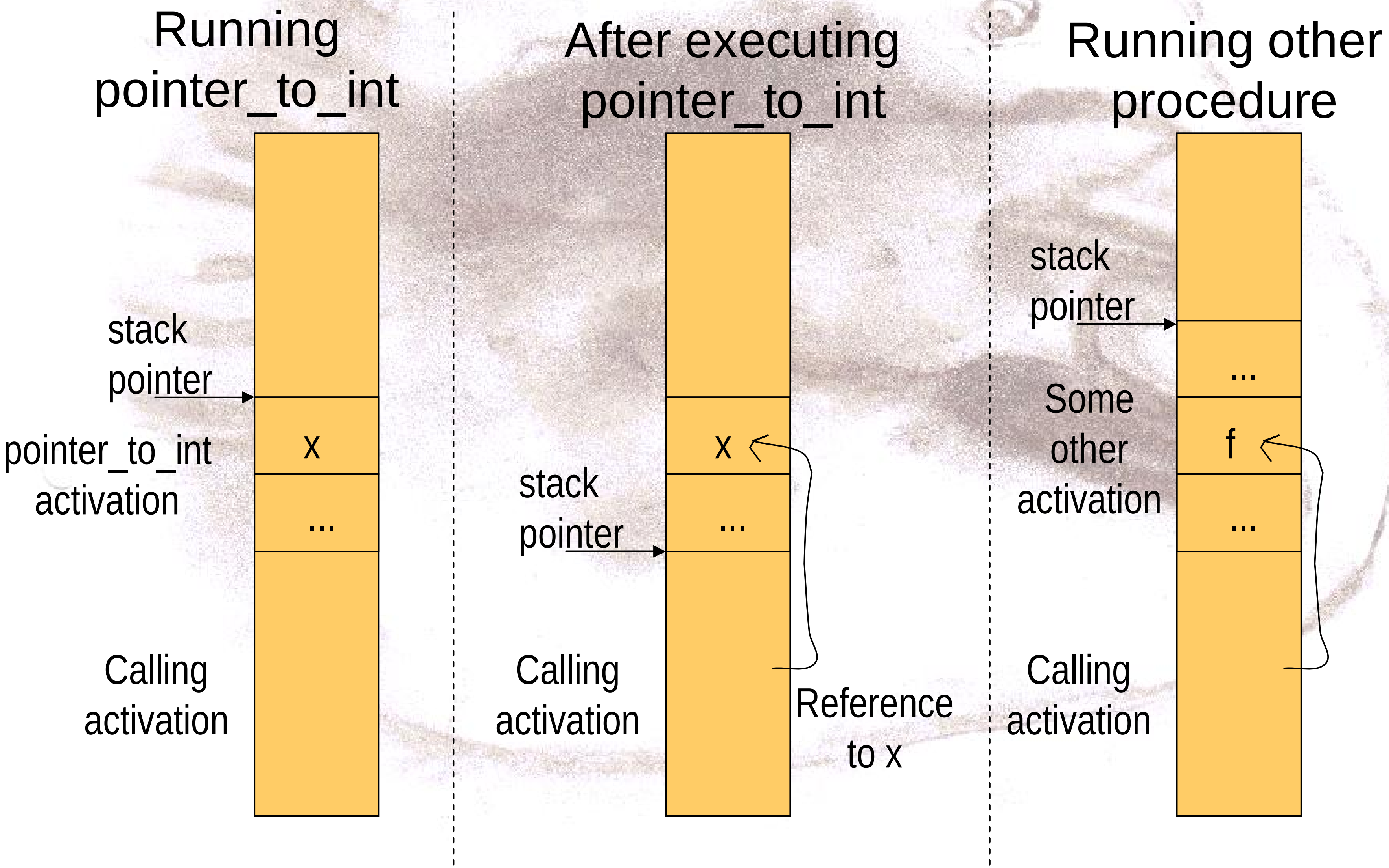
- Restrictions regarding e.g. information hiding, etc.
- String literals

Stack

- Home for transient objects
  - Allocation and deallocation is automatic
  - No need to make an explicit operating system call for memory
- References and sharing of data problematic

```
// THIS IS NEGATIVE EXAMPLE
 int * pointer_to_int()
 {
   int y;
   return & y;
 }
```

Example

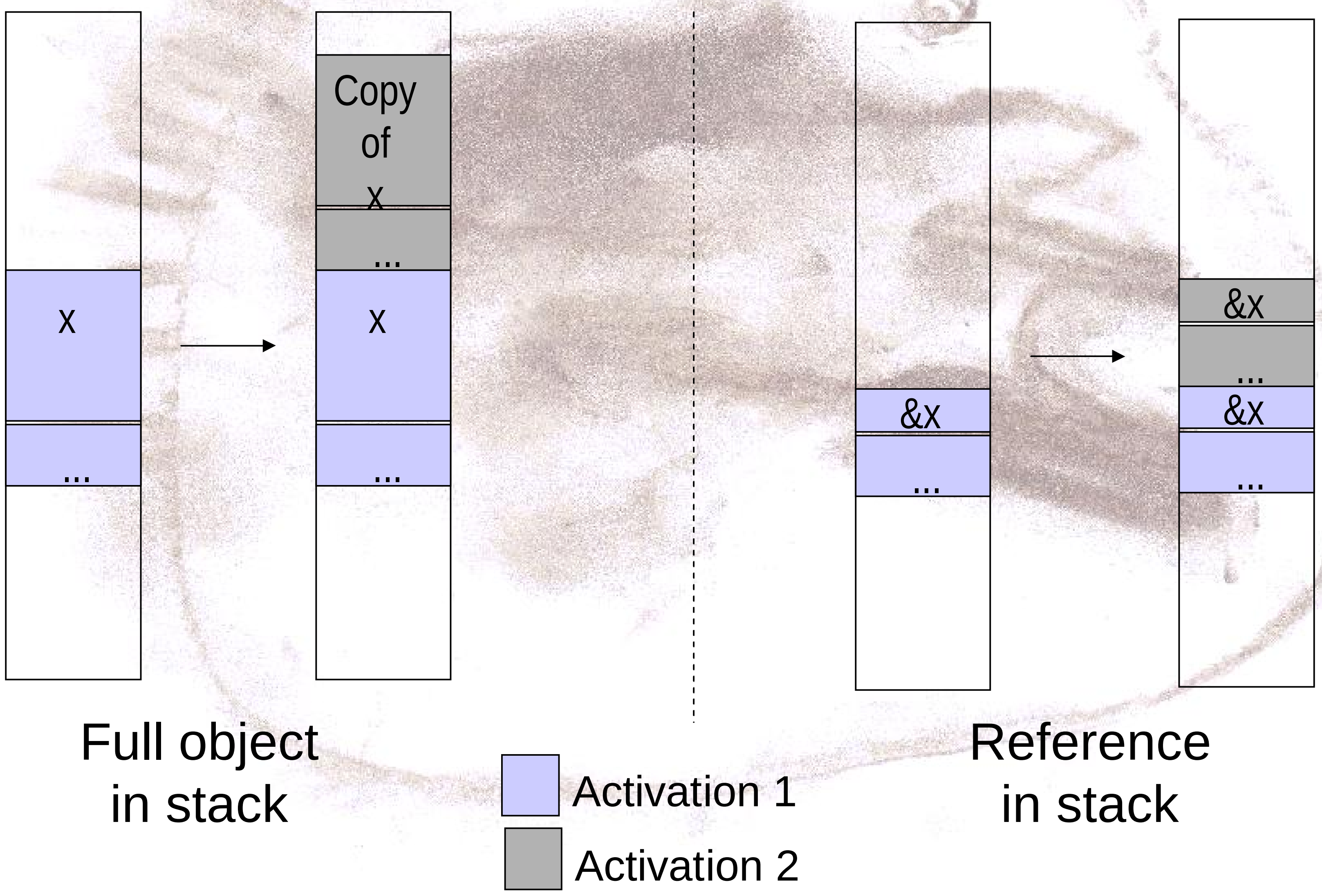

Running pointer_to_int
stack pointer
pointer_to_int activation
x
...
Calling activation
After executing pointer_to_int
x
stack pointer
...
Calling activation
Reference to x
Running other procedure
stack pointer
...
Some other activation
f
...
Calling activation

Heap

- Home for long-living objects
  - Sharing is less problematic than with stack-based variables, but errors can still occur
  - Large or global objects/data/variables that are needed in all phases of the program
- Reference passing commonly advocated in mobile setting
- Management of creation and deletion required

```
int * pointer_to_int()
{
  return new int(0);
}
```

Example


X
...
Copy
of
X
...
X
...
&X
...
&X
...
&X
...
Full object
in stack
Activation 1
Activation 2
Reference
in stack

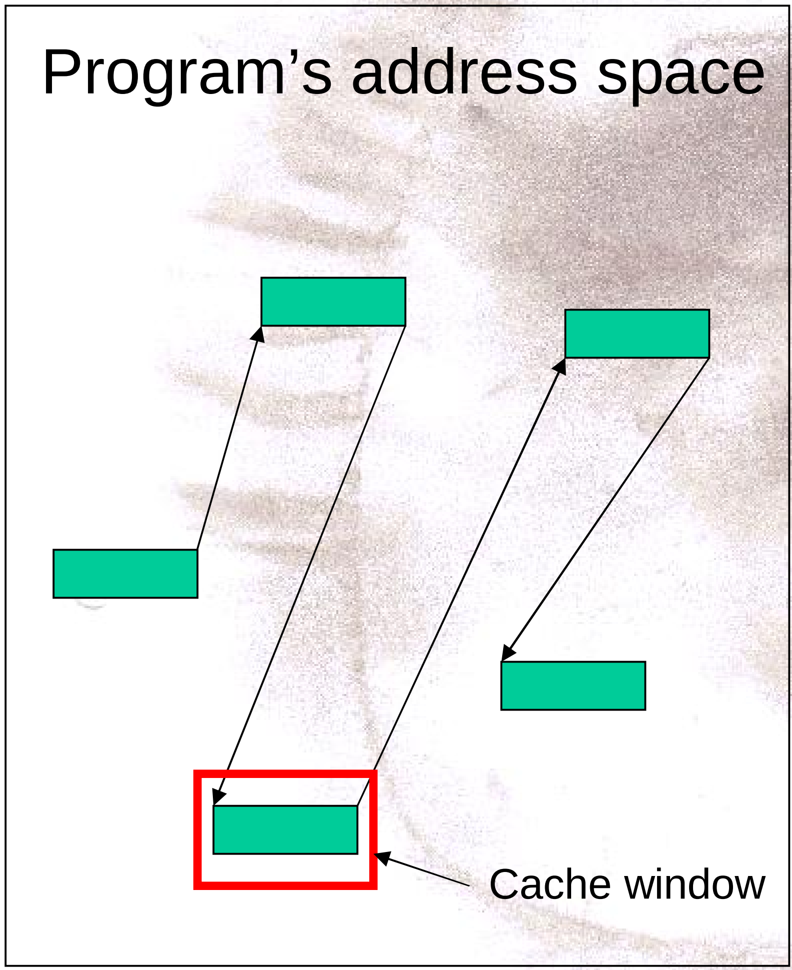
Content and goals

- Basics of memory usage in mobile devices
  - Static and dynamic allocation
  - *Managing memory organization*
- Memory management in Mobile Java
- Memory management in Symbian OS
- Summary

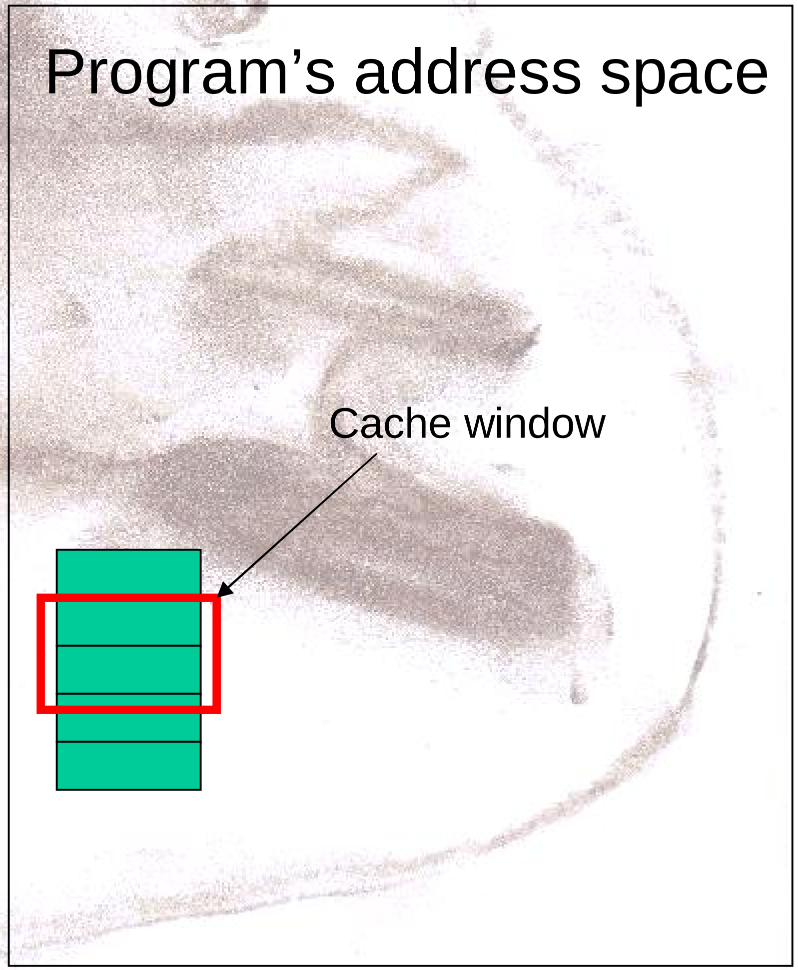
Managing memory organization

- Principle: Use the simplest data structure that offers the necessary operations
  - Consider linear data structures
- Consider other means of benefitting from memory layout (some basic principles to follow)
- Consider packing

Non-linear and linear data structure



List-based data structure



Linear data structure

Benefits of linear data structures

- Less fragmentation
- Less searching overhead
- Design-time management
- Cache improvement
- Monitoring
- Index use less memory than reference

Basic Principles

- Allocate all memory at the beginning of a program
- Allocate memory for several items even if you only need one
- Use standard allocation sizes
- Reuse objects (pool of free objects)
- Release early, allocate late
- Use permanent storage or ROM when applicable
- Avoid recursion

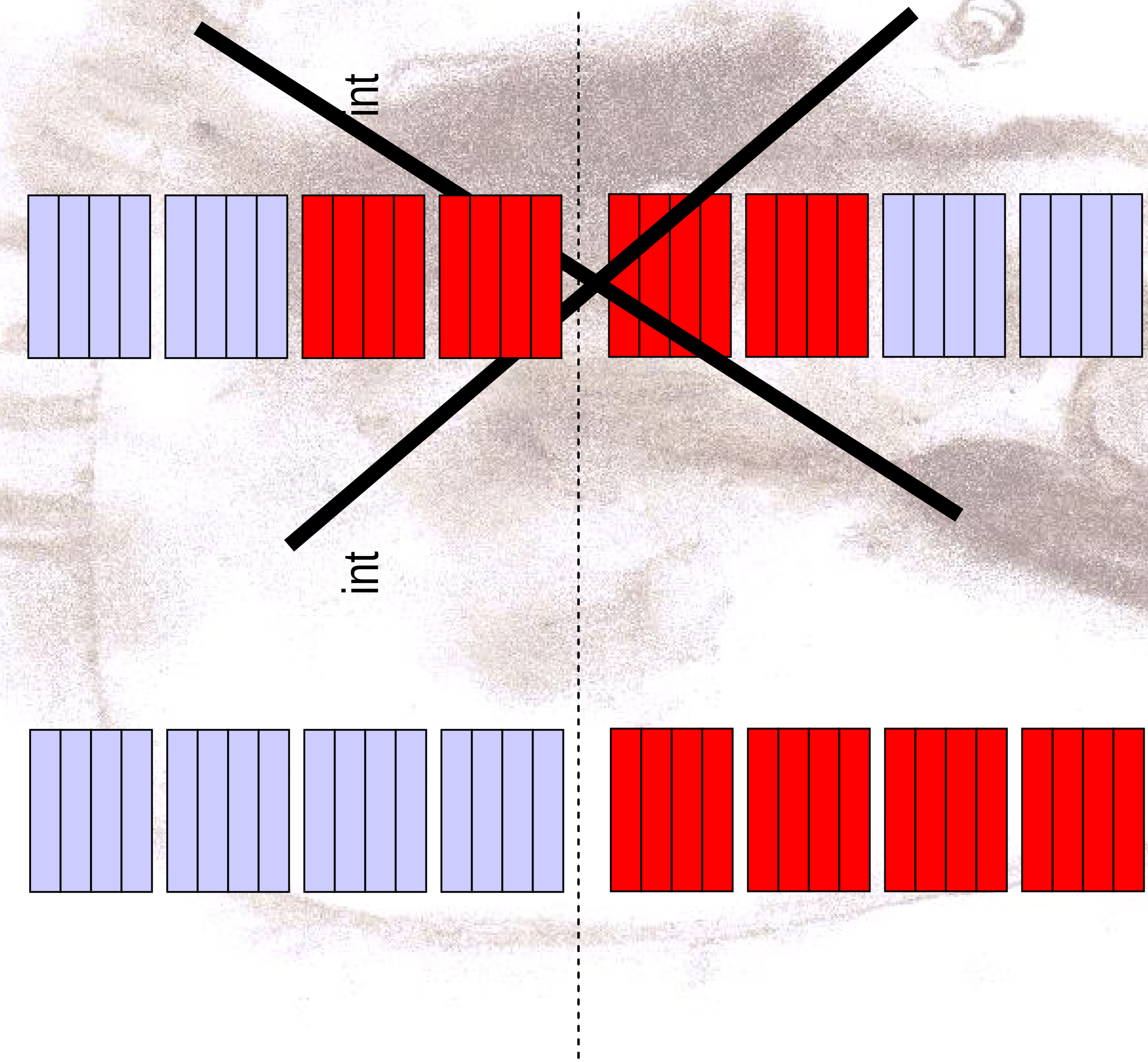
Consider packing

- Use compression with care
- Use efficient resource storage format
- Consider word alignment

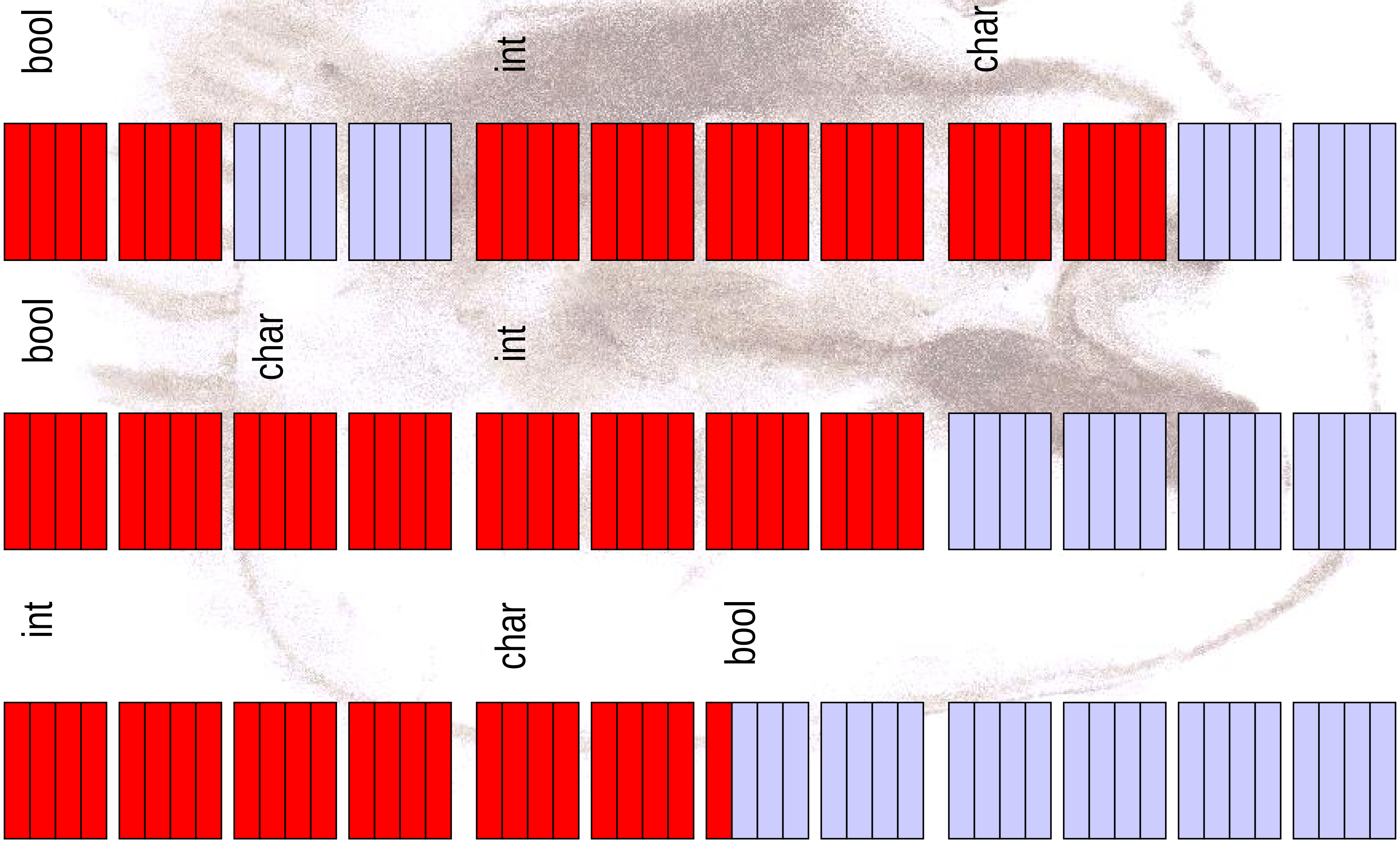
```
struct S {
  char b; // boolean
  int i;
  char c;
}
```

```
struct S {
  char b; // boolean
  char c;
  int i;
}
```

Word alignment



Packing



Content and goals

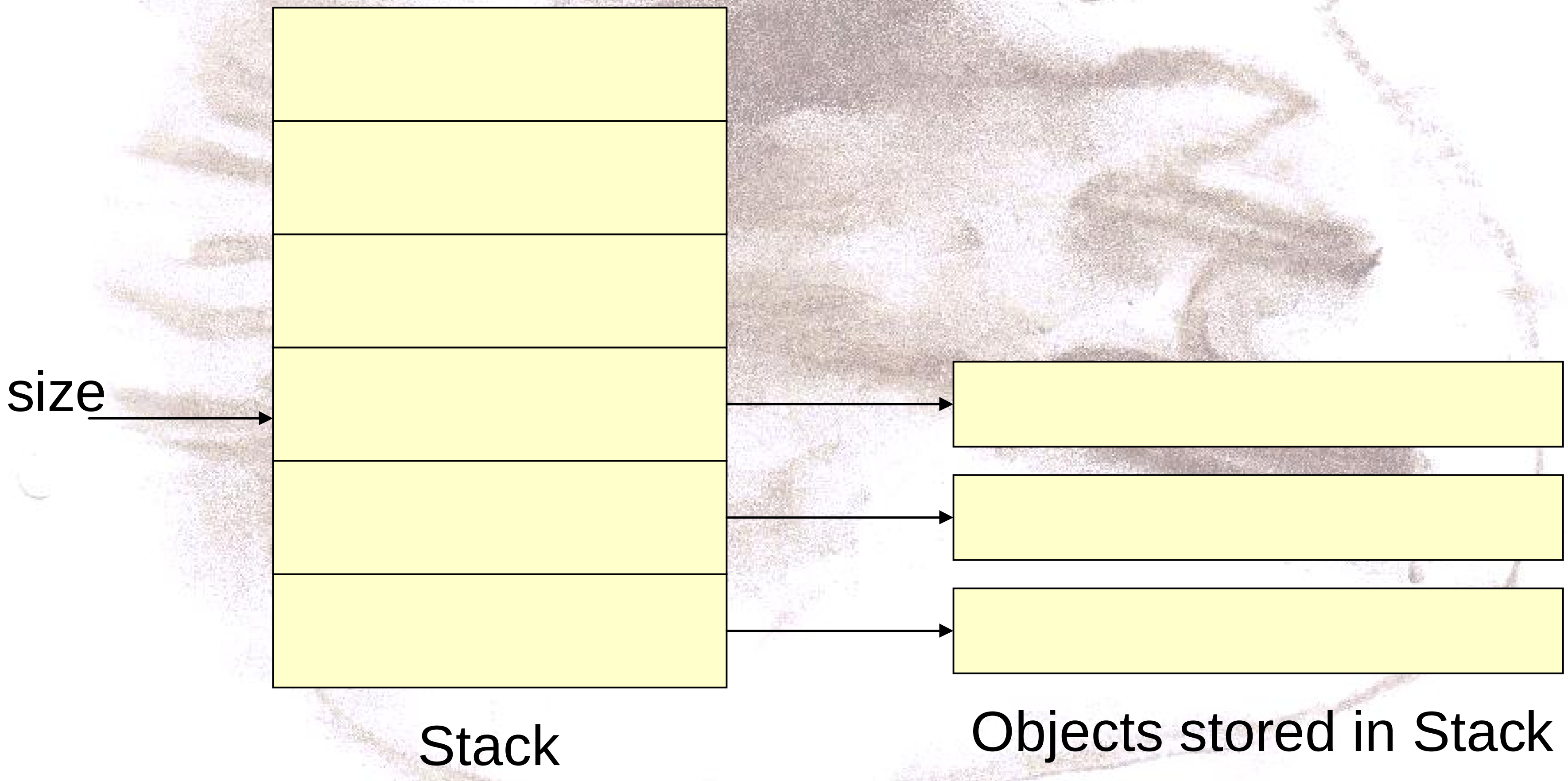
- Basics of memory usage in mobile devices
  - Static and dynamic allocation
  - Managing memory organization
- *Memory management in Mobile Java*
- Memory management in Symbian OS
- Summary

Motivation

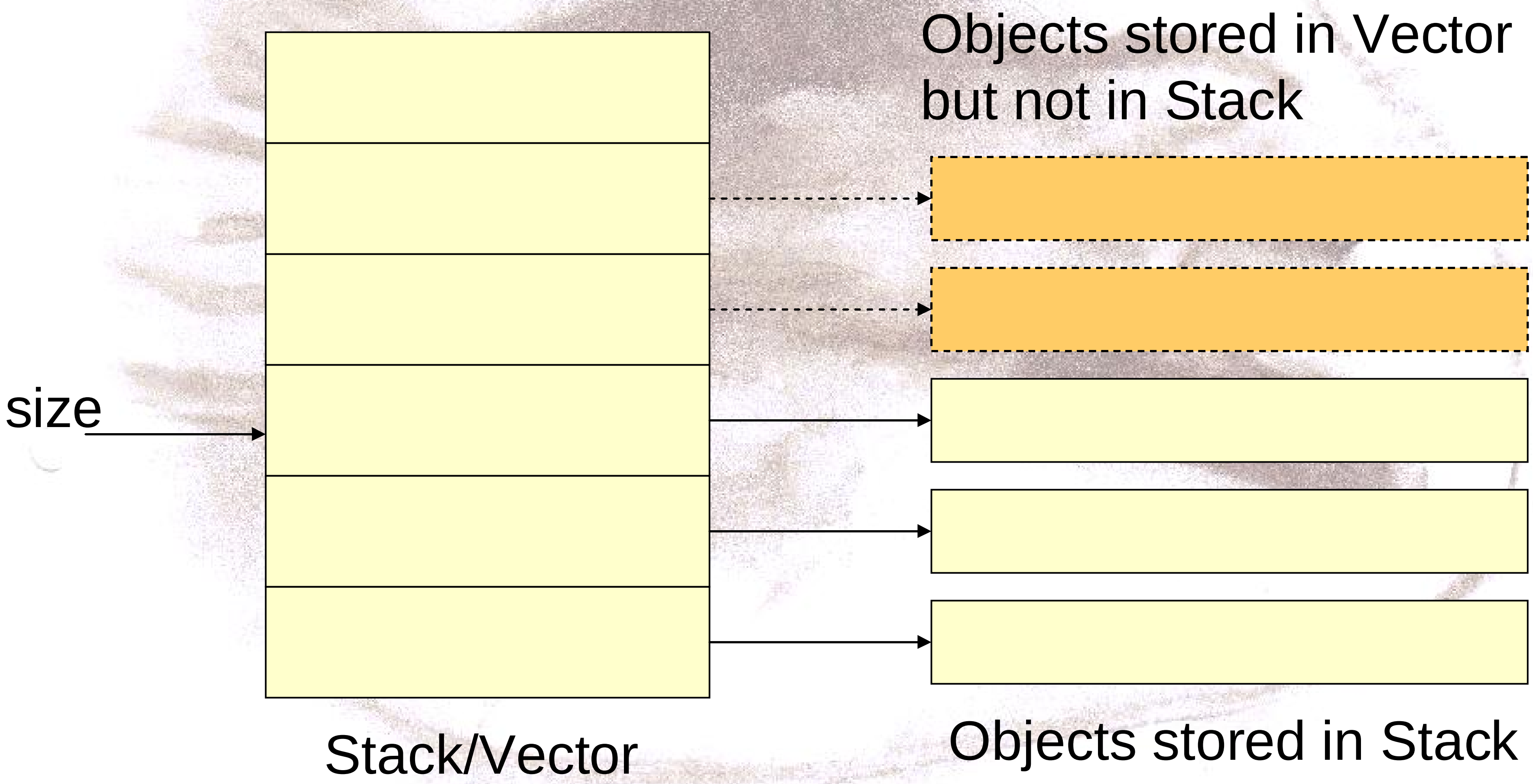
```
public void push(Object e) {
    ensureCapasity(); // Check slots count
    elements[size++] = e;
}

public Object pop() {
    if (size == 0) throw new EmptyStackException();
    return elements[--size];
}
```

- Ok?

Object stack



Leaking Abstraction



Upgrade

```
public Object pop() {
    if (size == 0)
        throw new EmptyStackException();
    Object result = elements[--size];
    elements[size] = null;
    return result;
}
```

Rules of Thumb

- Avoid small classes
- Avoid dependencies
- Select size when relevant and manage vector/string usage
- Consider using array vs. using vector
- Use stringBuffer when possible
- Manage class and object structure
- Generate less garbage
- Consider obfuscation
- Handle array initialization

Example 1

```
static final int SIZE = 2000;

private void arrayImp() {
   numbers = new int[SIZE];
   for (int i = 0; i < SIZE; i++) { numbers[i] = i; }
}

private void vectorImp() {
   numberV = new Vector(SIZE);
   for (int i = 0; i < SIZE; i++) { numberV.addElement(new Integer(i)); }
}

private void vectorImpSimple() {
   numberV2 = new Vector(); // Default size
   for (int i = 0; i < SIZE; i++) { numberV2.addElement(new Integer(i)); }
}
```

Results

- ArrayImp (minimal overhead)
  - Bytes: 8016
  - Objects: 1
- VectorImp (integers wrapped to objects)
  - Bytes: 40000
  - Objects: 2002
- VectorImpSimple (failures in guessing the size)
  - Bytes: 52000
  - Objects: 2010

[Hartikainen: Java application and library memory consumption, TUT, 2005]

Example 2

```
static final int AMOUNT = 100;

public void useString() {
    String s = “”;
    for(int i = 0; i < AMOUNT; i++) {
        s = s + “a”;
    }
}

public void useStringBuffer() {
    String s = “”;
    StringBuffer sb = new StringBuffer(AMOUNT);
    for(int i = 0; i < AMOUNT; i++) {
        sb = sb.append(“a”);
    }
    s = sb.toString();
}
```

Results

- UseString (simplest)
  - Bytes: 39000
  - Objects: 450
- UseStringBuffer (optimized)
  - Bytes: 304
  - Objects: 5

[Hartikainen: Java application and library memory consumption, TUT, 2005]

Content and goals

- Basics of memory usage in mobile devices
  - Static and dynamic allocation
  - Managing memory organization
- Memory management in Mobile Java
- *Memory management in Symbian OS*
- Summary

Naming Conventions

- Class names start with C
- Kernel class names start with D
- Type names start with T
- Mixin class names start with M
- Enumerated class names start with E
- Resource names start with R
- Method names start with a capital letter
- Names of methods that can throw an exception end with L (or LC)
- Simple getters and setters reflect the name of the variable
- Instance variable names begin with i
- Argument names begin with a
- Constant names begin with K
- Automatic variable names begin with lower-case letters

Descriptors

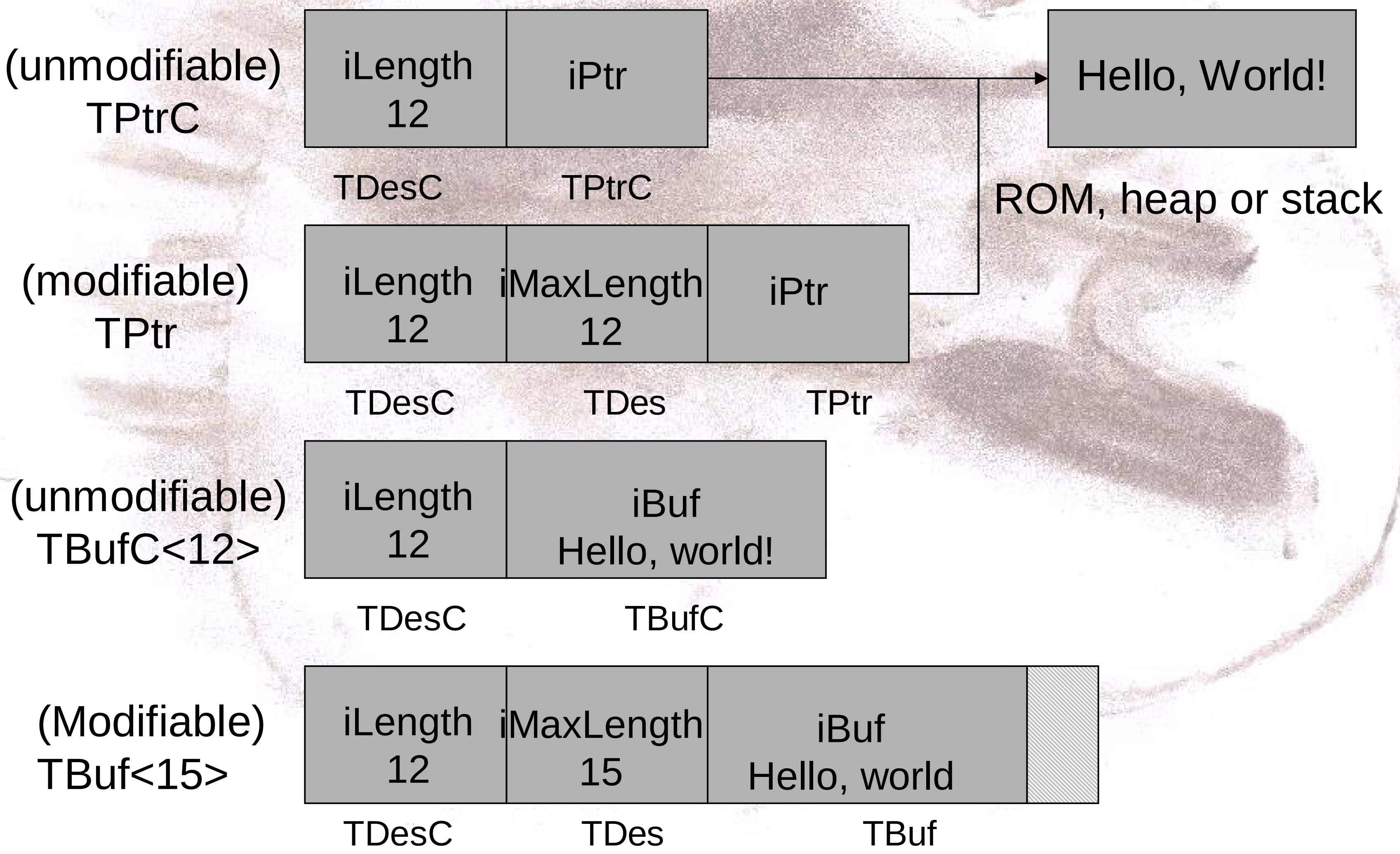
- Symbian way of using strings

```
_L("Hello"); (depreciated except in demos and debugging)
 _LIT(KHelloRom, "Hello");
 // String in program binary.

 TBufC<5> HelloStack(KHelloRom);   // Data in thread stack.

 HBufC* helloHeap = KHelloRom.AllocLC();  // Data in heap.
```

- Guards against overflows

```
char userid[8]; // Vanilla C++
strcpy(userid, "santa.claus@northpole.org");

TBuf<8> userid; // Symbian
userid = _L("santa.claus@northpole.org");
```

Some memory layouts



Using Descriptors

- Use descriptors rather than degenerate to Ttext* format
- Use TDesC& for arguments
  - Light-weight
  - Safe (no accidential modifiction)
  - Any descriptor can be passed
- Use **new** only with HBufC
  - Reserve others from stack
- Type casting is possible
  - HBufC::Des
  - TPtr, TDesC::Alloc
  - HBufC *

Exceptions

```
TRAPD(error, BehaveL()); // try

// Exception handler
if (error != KErrNone)
   { // catch
   if (error == KErrNotSupported) {...}
   if (error == KErrUnknown) {...}
   }

User::Leave(KOutOfMemory); // throw
```

Exceptions and Allocation

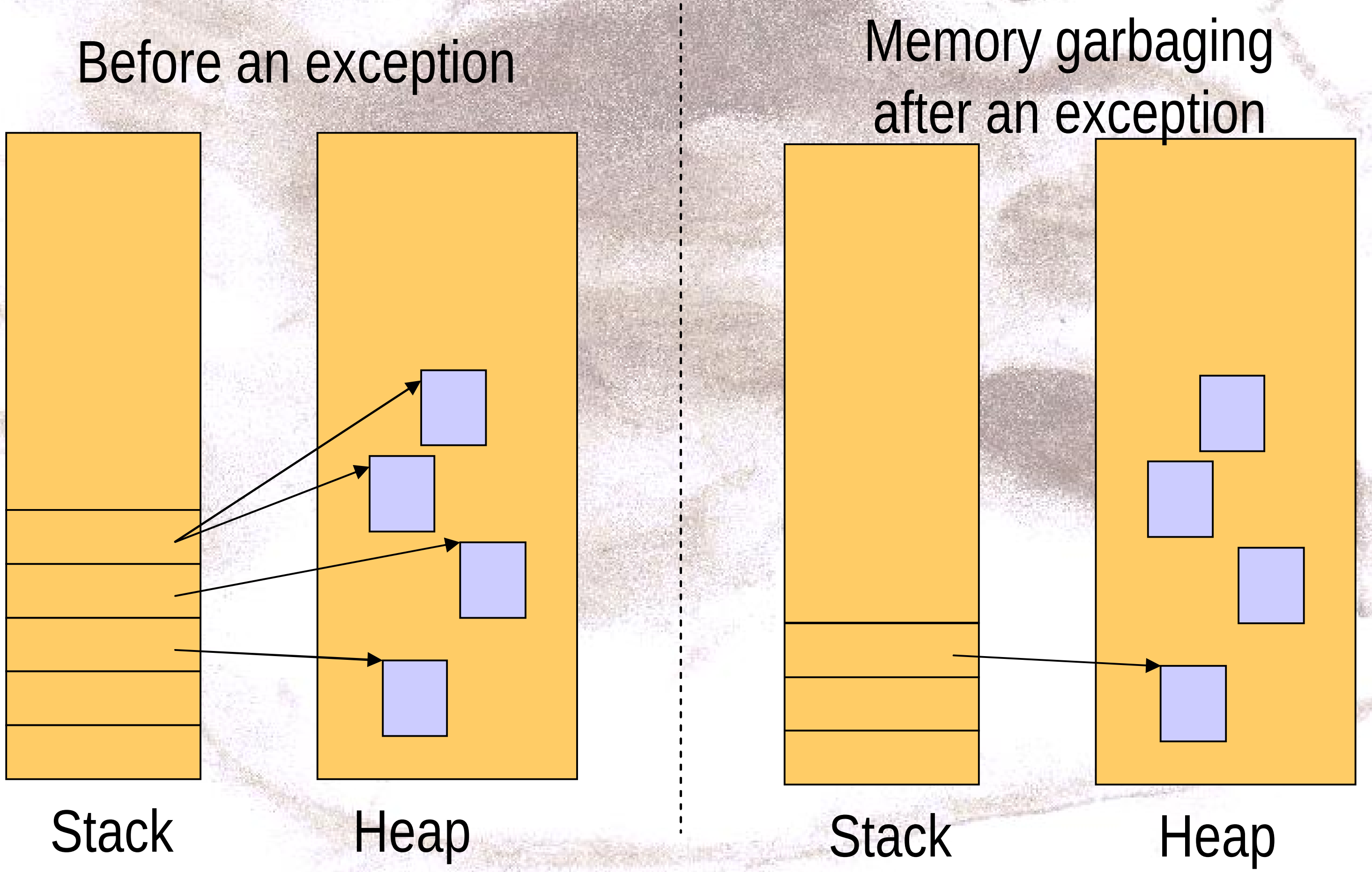
- All memory allocations use an overridden version of **new** operator

```
c = new (ELeave) CMyClass();
```

- Corresponding method

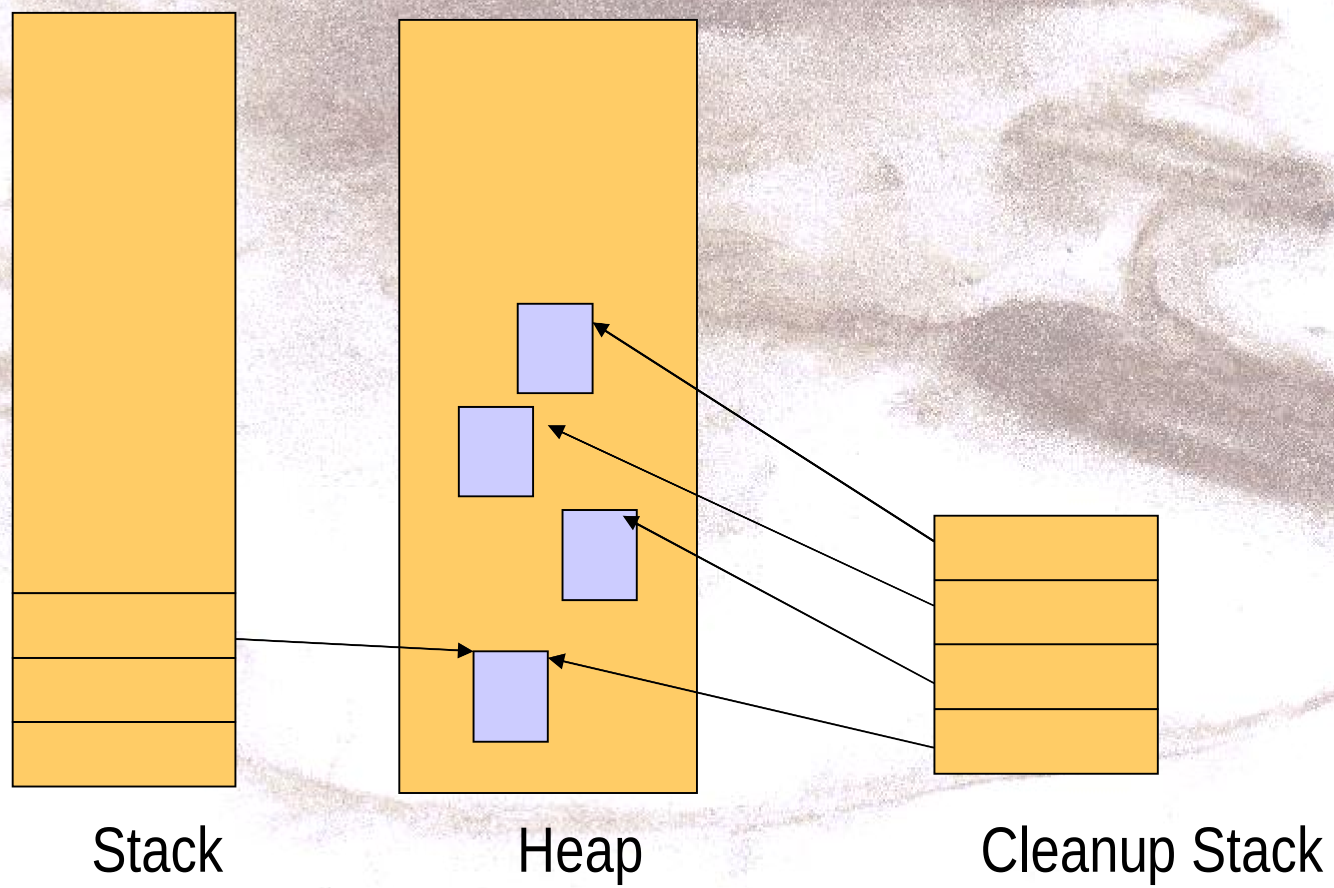
```
c = new CMyClass();
if (!c) User::Leave(KOutOfMemory);
return c;
```

Problem: What happens to automatic heap-based variables in an exception?



Cleanup Stack – An Auxiliary Data Structure

Cleanup stack enables deallocation during an exception



Using Cleanup Stack

- A programmer responsiblity
- Only for automatic variables, never for others

```
CMyClass * c = new CMyClass();
CleanupStack::PushL(c)
c->DoSomethingL(); //... c is used
CleanupStack::Pop(); // c
delete c;
c = 0;
```

- Classes derived from CBase get their destructor called, for others only memory is deallocated
- Also other actions (e.g. CleanupStack::ClosePushL(file);)

Two-Phase Construction

- Cleanup stack cannot help in the creation of objects
- Therefore, actual constructor should never fail, and problematic aspects should be executed only after a reference to the object has been pushed to the cleanup stack

```
CData * id = new (ELeave) CData(256);
CleanupStack::PushL(id);
id->ConstructL();
```

Shorthands

```
CItem::NewL() {
  CItem * self = new (ELeave) CItem;
  CleanupStack::PushL(self);
  self->ConstructL();
  CleanupStack::Pop(); // self
  return self;
}

CItem::NewLC() {
  CItem * self = new (ELeave) CItem;
  CleanupStack::PushL(self);
  self->ConstructL();
return self;
}
```

Content and goals

- Basics of memory usage in mobile devices
  - Static and dynamic allocation
  - Managing memory organization
- Memory management in Mobile Java
- Memory management in Symbian OS
- *Summary*

Summary

- Memory related considerations are a practical necessity
  - Even virtual machines require programmer to consider allocation of variables and the use of data structures
- Design idioms and patterns have been introduced that give general guidelines
  - Preallocation and static allocation simplify memory management
  - Linear data structures offer several benefits
  - Data packing as the last resort
- Mobile development platforms assume that the developers are aware of the most common pitfalls