School of Electrical Engineering and Computer Science

THE UNIVERSITY OF NEWCASTLE AUSTRALIA

SENG2200/6220 Programming Languages & Paradigms

Topic 2

C++ vs Java - Part 1 Encapsulation, Information Hiding, and Memory Management

Dr Nan Li
Office: ES222
Phone: 4921 6503
Nan.Li@newcastle.edu.au

Topic 2 Overview

Easy stuff first –

Compilation is quite different

Encapsulation in C++ and Java are very similar

Information Hiding also

but

Memory Management is VERY different

- Memory structure of a C program
- Why is static called *static*?
- Scoping support for Linked Structures
- Lifetime support for Linked Structures
- Destructor Methods vs Garbage Collection

Compilation Comparisons

Java

- Single file compilation
- Import of required classes
- Java files compiled to byte code Class files
- Any class with a main() method may be the entry point
- JVM runs the byte code class files as an interpreter
- Where does BlueJ fit in?

C++

- Separation of specification and implementation
- Separate file compilation for each class
- Inclusion of required classes
- Implementation files contain compiled but unlinked code
- Linking required classes into a single executable file with a single entry point
- Compiled program is loaded by the o/s and executed directly on the hardware

Encapsulation

Both languages use the standard O-O Class abstraction to encapsulate

- Instance or attribute data
- Functionality via methods

Syntax structures are different

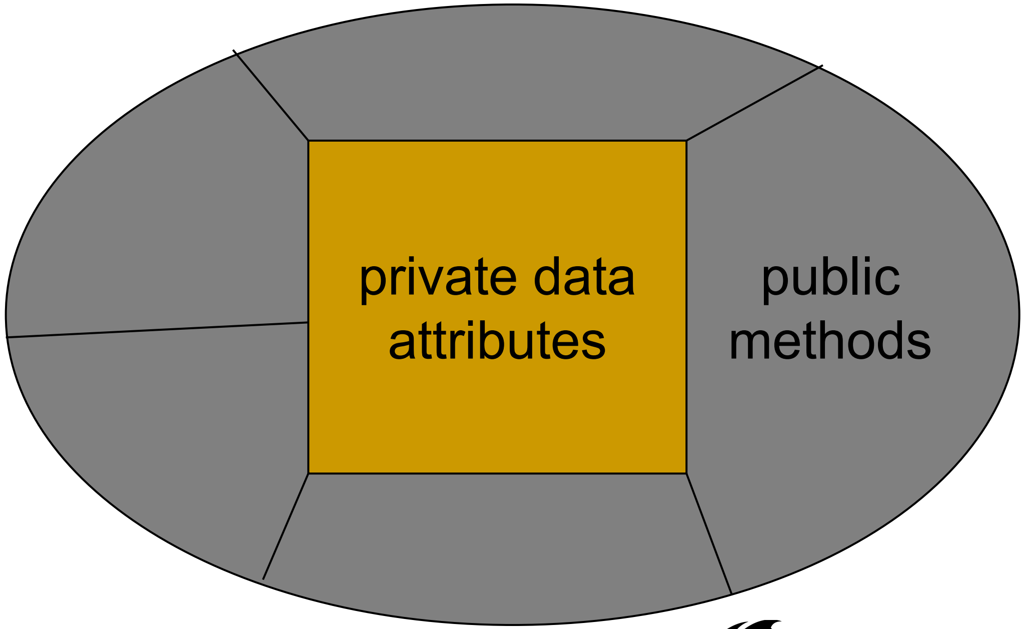
- Java has a single class file containing data and methods
- C++ uses the C-style header file structure for attribute data and method specification
- C++ uses a standard .cc (.cpp) file to hold method implementations for a class

Information Hiding

Both languages use the standard O-O Class abstraction to hide private attribute data and present a functionality interface of public methods

Private data cannot be accessed outside the class, being hidden behind the public methods

The public methods are the only feature of the class that can be accessed from outside, ie they are a public interface to the class



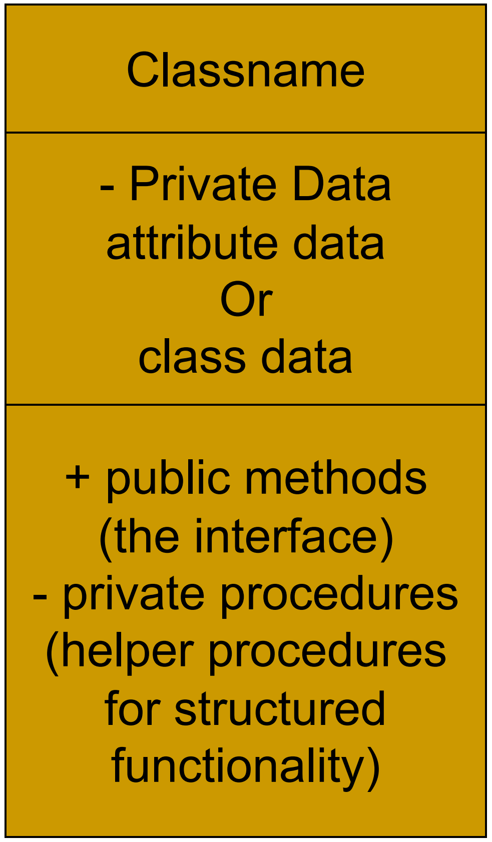
Information Hiding & UML

Data will always be expected to be private (or perhaps protected once we deal with inheritance)

- What is the only data (information) that might possibly be public?
- Class constants - possibly - but these will be better as class (static) items even if they remain public

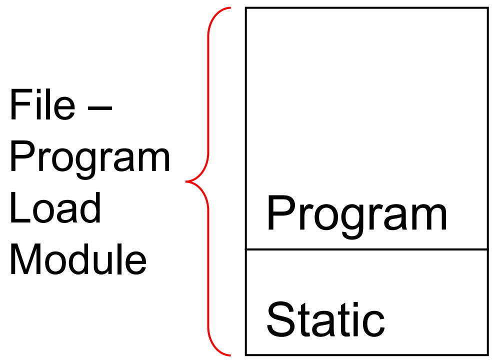
Methods are the only things that allow an object to change its state (values of its attributes - concept of closure) and so are public

Helper procedures (not really methods) allow the public methods to be designed using structured programming techniques, and so are private.


Classname
- Private Data attribute data Or class data
+ public methods (the interface)
- private procedures (helper procedures for structured functionality)

A C Program

A short trip back into history to explore Static, Automatic, and Heap data.



Program

- instructions bound for the CPU Fetch Execute Cycle

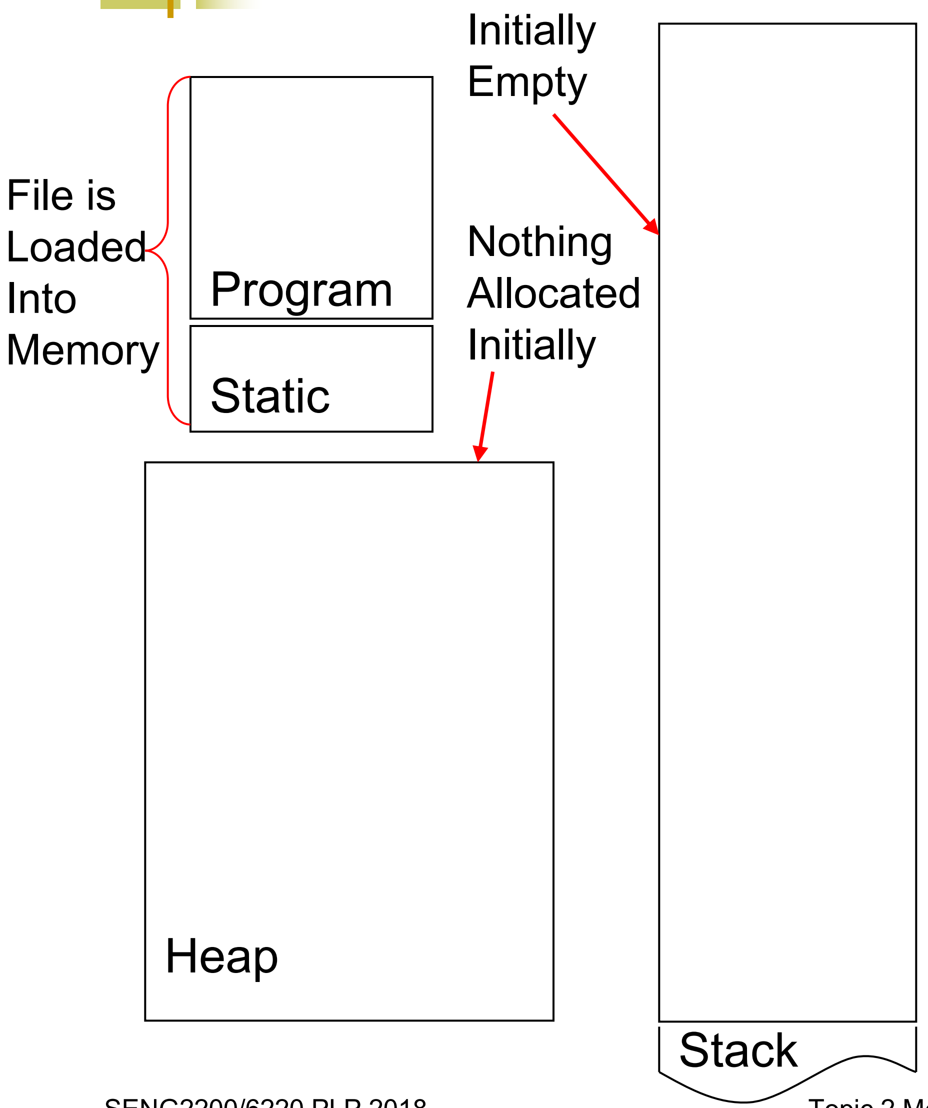
Static

- memory allocated at (as part of) program initiation and loaded directly into a fixed area of RAM (hence the term static)

These are loaded into memory, but for program initiation and execution, there needs extra run-time support by means of a stack and dynamically allocated heap.

THE UNIVERSITY OF NEWCASTLE AUSTRALIA

Basic Memory Management



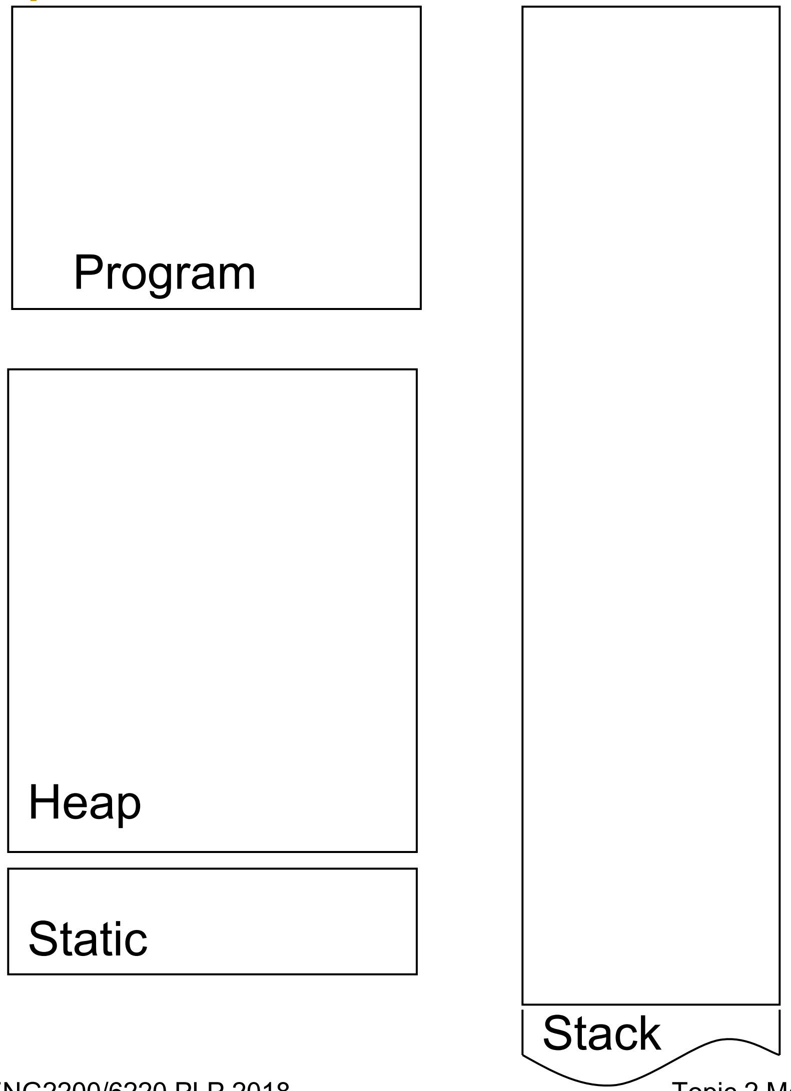
Program
- instructions bound for the CPU Fetch Execute Cycle

Static
- memory allocated at as part of program initiation

Heap
- memory allocated and de-allocated under program or environment control

Stack
- memory area allocated as last allocated first released, (LIFO) generally supporting procedure/method calls

Basic Memory Management



Program

- Loaded into memory

Static

- Loaded into memory

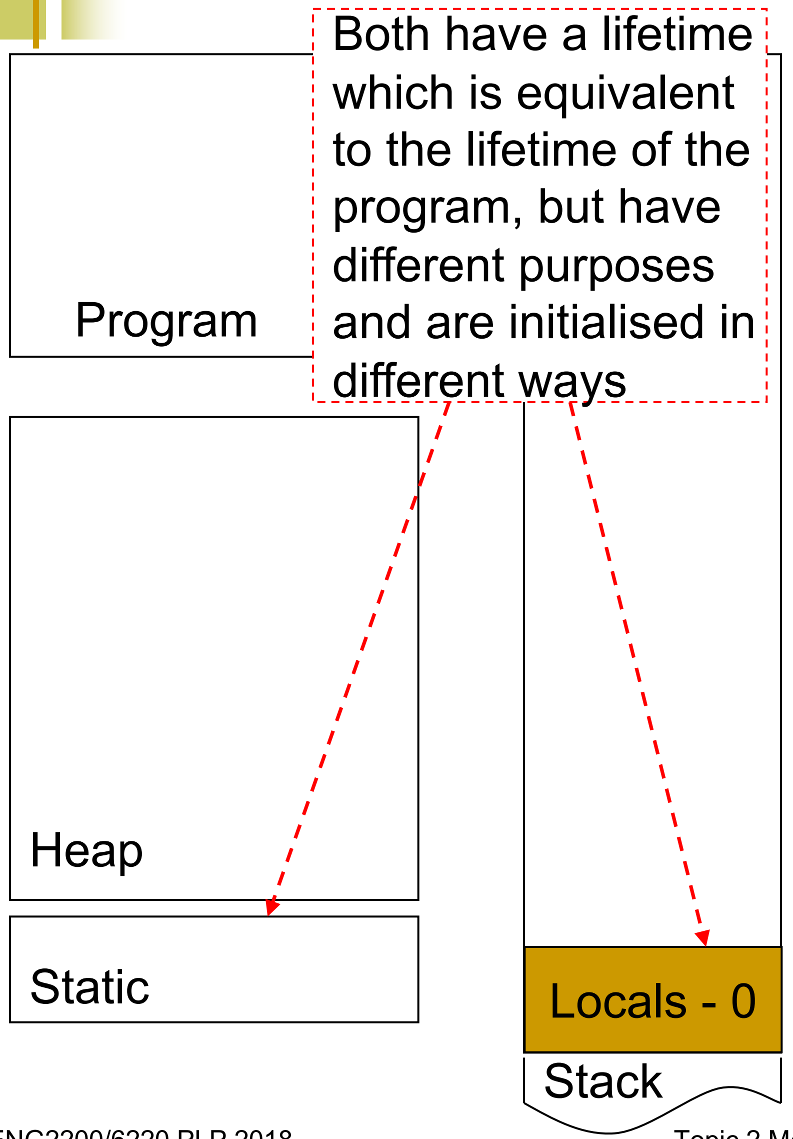
Heap

- Memory area set aside with allocation and de-allocation support, effectively on a byte or word granularity.

Stack

- Memory area set aside initially empty.

C Program Initiation



Program
Both have a lifetime which is equivalent to the lifetime of the program, but have different purposes and are initialised in different ways
Heap
Static
Locals - 0
Stack

Program & Static

- Loaded into memory

Heap and Stack

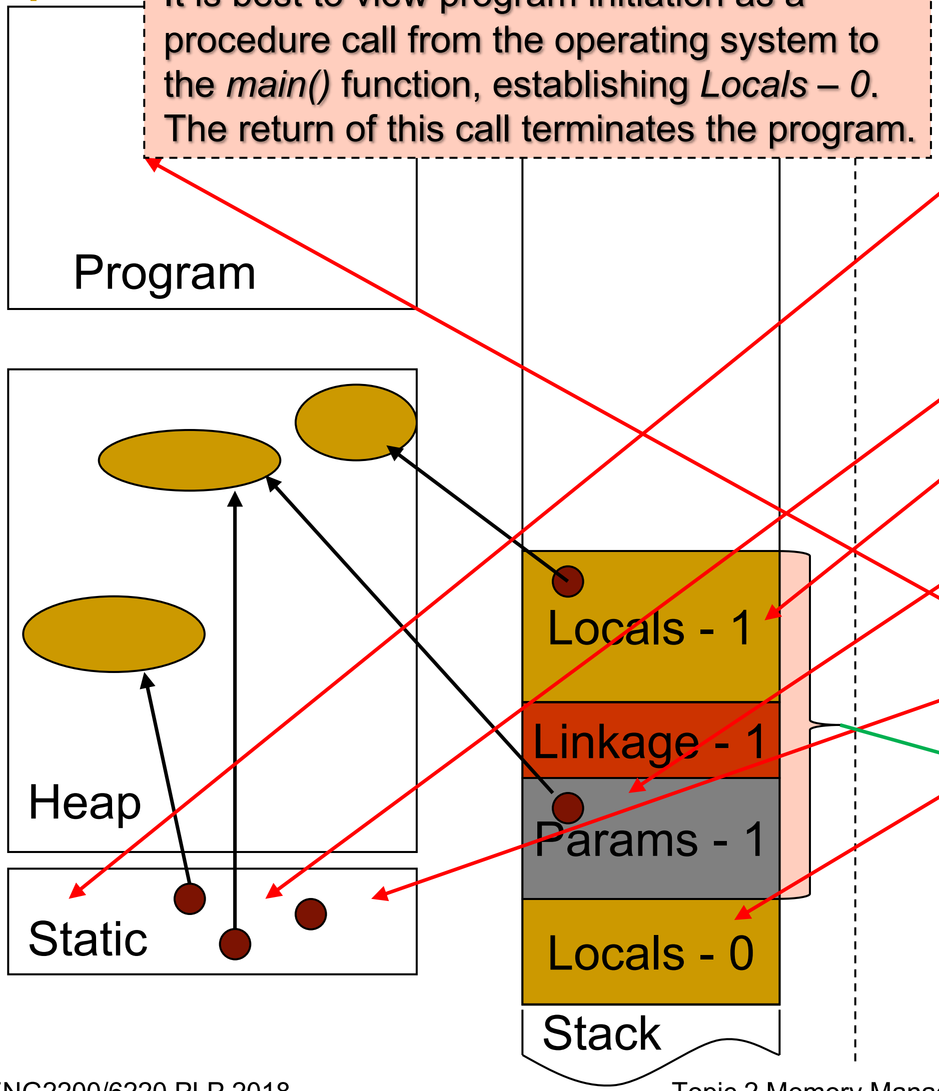
- Empty

Initiation is best seen as a procedure call from the operating system, once the program completes this “call” returns

This leads to an interesting lifetime relationship between static memory and the main program local variables that are at the base of the C program’s runtime stack

C - Static & Automatic Data

It is best to view program initiation as a procedure call from the operating system to the *main()* function, establishing *Locals – 0*. The return of this call terminates the program.



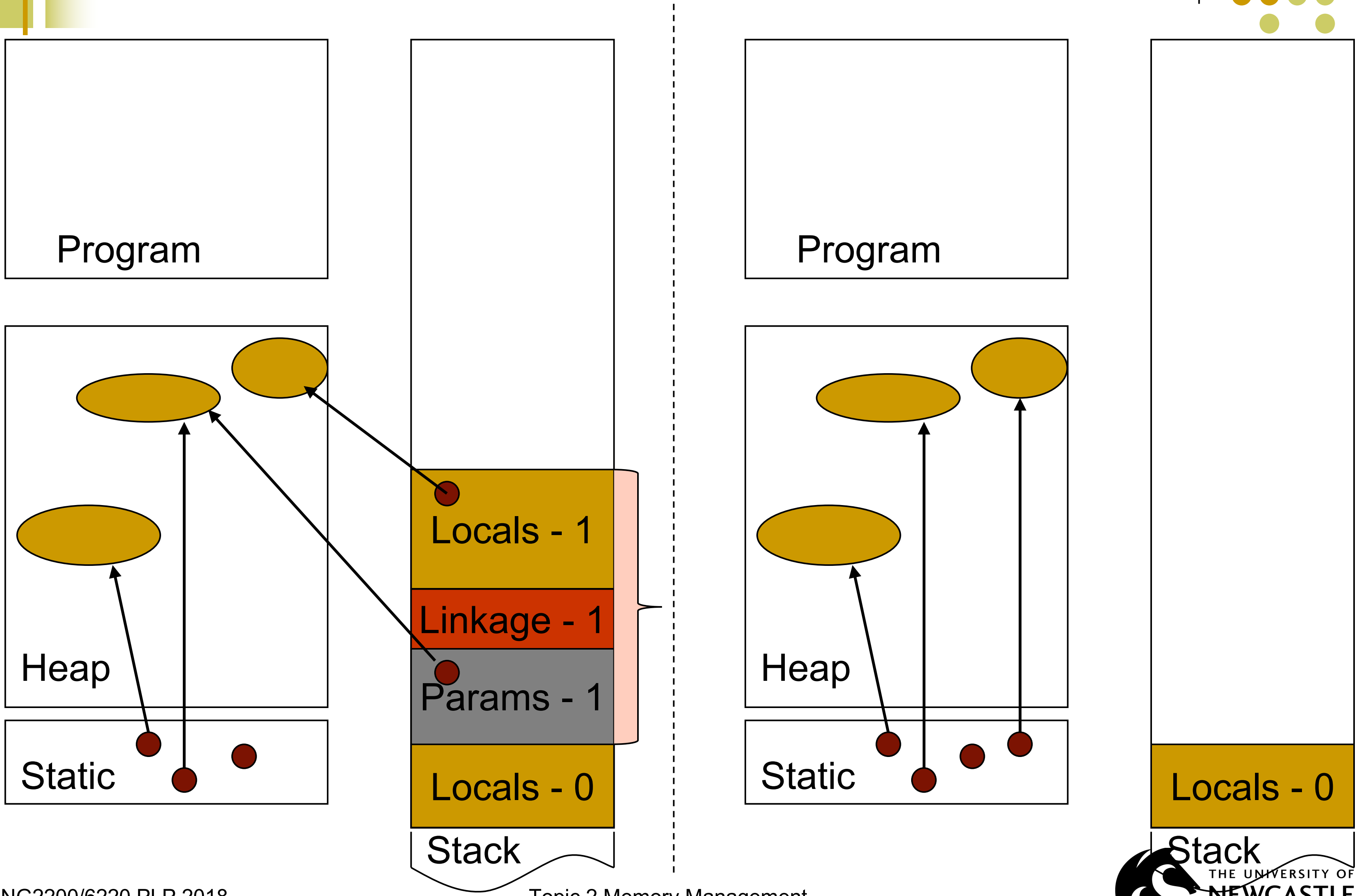
Three important aspects placement, lifetime & visibility

```
int a,b,c;
int proc1 (int d, int e) {
      static int f;
      int g, h;
      ……
}
main() {
      static int i,j,k;
      int l, m, n;
      ……
}
```

The Call Frame

Params: Pushed by Caller
Linkage: Return Address and Frame Pointer
Locals: Declared & initialised by Callee

C Programs and The Heap



C, C++ & Java Static Data

In C, we have data that is shared across procedure calls

C++ extends this to include data that is shared by all objects in a class – i.e. data that belongs to the class rather to any particular instance of the class.

Java basically removes the C-style use of static data because everything is an object in Java.

C++ and Java then both extend static to include methods that are shared by all objects of the class, and can even run when there are no objects instantiated for the class. These methods can therefore only access static data, or parameters passed to them.

So, what is special about Static data?

1) In C: It gets allocated (and initialised) at load time with a lifetime of the whole program execution.

Visibility: Globally declared data is visible to all functions of the program. BUT - visibility can be restricted to a single procedure if it is declared there, effectively giving the procedure some data it can remember from one call to the next (ie share the data amongst all the procedure calls). The data item is shared across time by all the procedure calls.

In Java: It serves a similar purpose but instead of just being applied to procedure calls, it is made to apply to objects – so static data can be shared by all the objects that currently exist for a class (and has a meaningful value even when there are no objects of the class instantiated). The data is shared across time and memory by all the objects of a class. As Java forces the use of classes, every piece of data must belong to a class, and this is therefore the case for static data as well as instance/attribute data.

In C++: Static data can be used in both the C fashion and the Java fashion, but is mostly used in the Java fashion now.

Static – applied to methods

C++ and Java:

- Further extend the meaning of static to include methods that can use this shared data

Consequently:

- Static methods can run when there aren't any objects of the class to run them
- Static methods can ONLY access static data

Java: public static void main()

Finally we have a definitive explanation of what this really means

public: can be called from outside the class where it is declared

void: has no return value

main(): its "well known" name

static: can be run when there are no objects instantiated yet – and so is the perfect way to start off a program, establish the required interfaces and data structures, and then start the processing …

Back to static data: C++ "const" and Java "final"

C++ uses the keyword const in several different ways

- However the basic tenet of const is that the value stored cannot be changed
- Used with parameters as well as "constant" data items
- Also used with values returned from functions

Java uses the keyword final in most of the ways that C++ uses const

public static final …..?

- public/private is a local or global constant (visibility wrt the class)
- static shares one copy between all objects
- final stops any effort to alter the value
- Accessed via class name
- Orthogonal use of static and final for constant values

Review of Static and Automatic

Returning to our example (from slide 11 – slightly altered)

```
int a = 0, b = 1, c = 2;
int proc1 (int d, int e) {
    static int f = 0;
    int g, h;
    ……
}
main() {
    static int i = 0, j = 1,k = 2;
    int l, m, n;
    ……
}
```

a, b, c: Global so default to static, initialised at load time, lifetime and visibility are whole program

d, e: Parameters, default to automatic, local to procedure but initialised at call time, lifetime and visibility are a single procedure call

f: Local to procedure so explicitly static, initalised once at load time, lifetime is whole program, visibility is any call to this procedure

g, h: Local to procedure so default to automatic, created at call time, initialised within procedure, lifetime is a single procedure call

i, j, k: Local to main() but explicitly static, created and initialised at load time, lifetime is whole program, visibility is main().

l, m, n: Local to main(), created at program initiation (the o/s call to main()), lifetime is whole program, initialised within main(), visibility is main().

Class Data & Class Methods

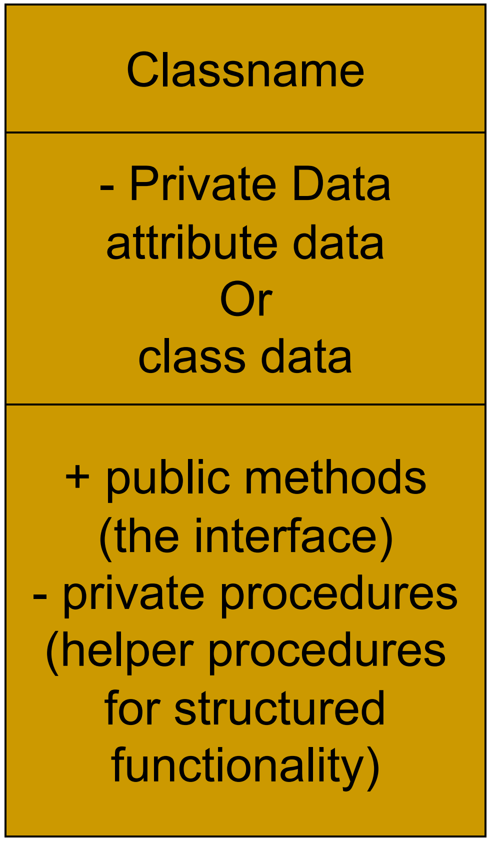
Class data is shared between all instantiated objects of the class

- The data must have a legitimate (usually initial) value when there are no objects instantiated
- So this is why it is placed in the static memory area - giving rise to the term “static data” in C++ and Java in place of the usual O-O term “class data”
- Any object can alter a class data item value

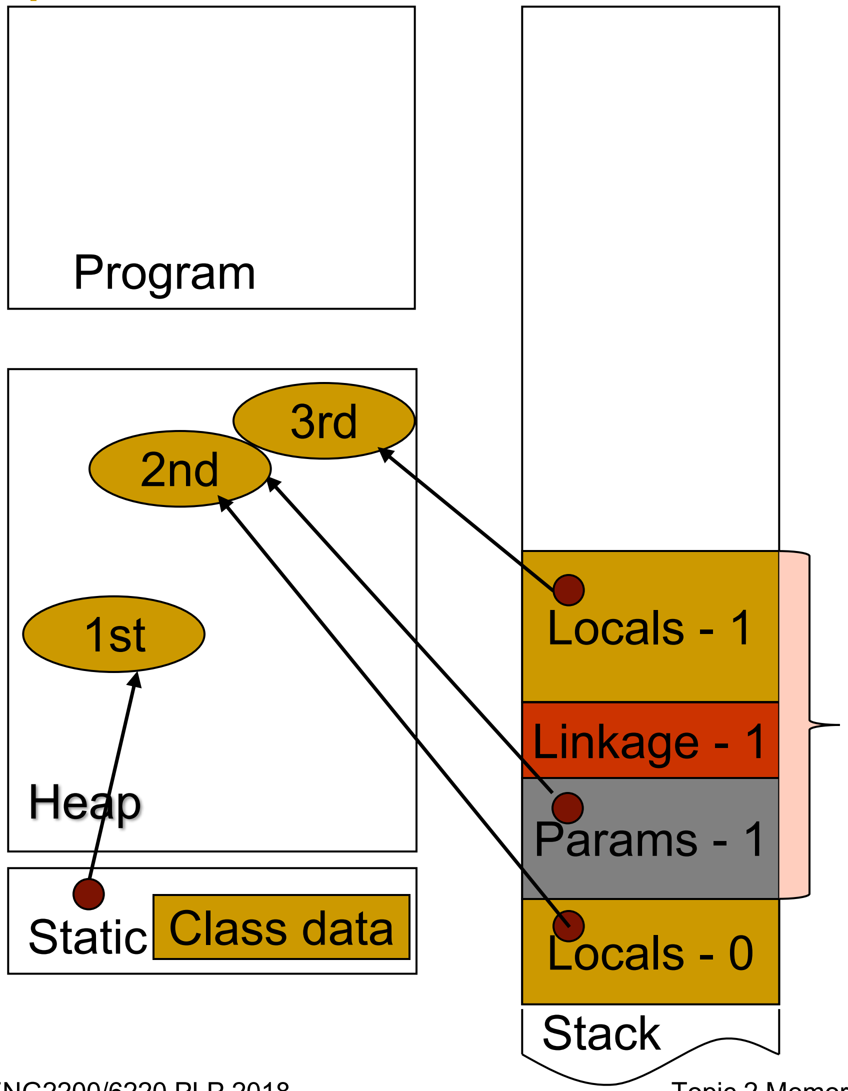
Class methods can therefore only access class data (because they can be called when there are no objects instantiated).

- In C++ and Java the term static carries through to these methods

Class data and class methods are usually used to audit the objects of a class.


Classname
- Private Data attribute data Or class data
+ public methods (the interface)
- private procedures (helper procedures for structured functionality)

Class Data & Static Memory

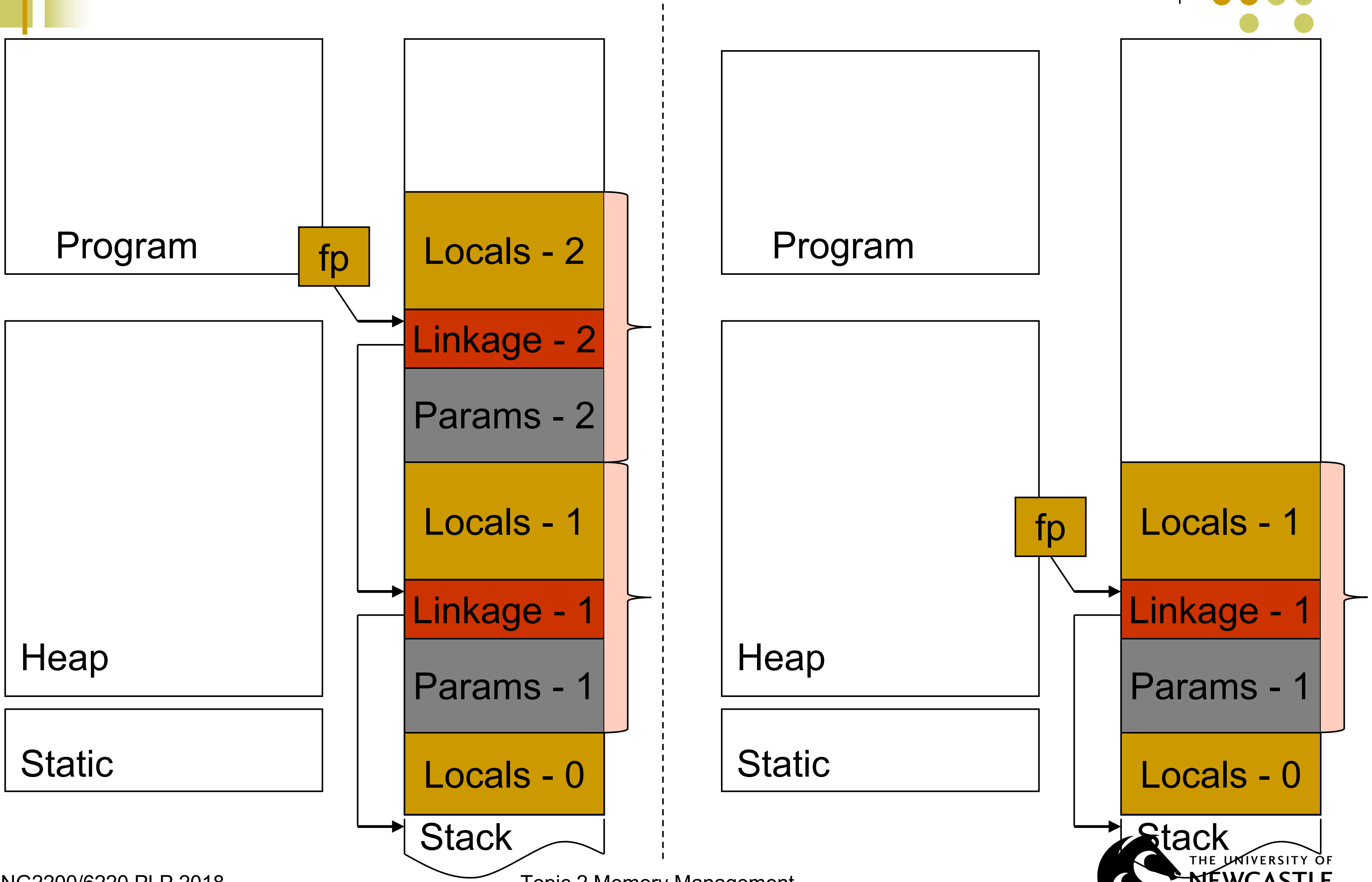


In Java and C++ class data is allocated to the static memory area, and these data are then shared between all the instantiated objects of the class

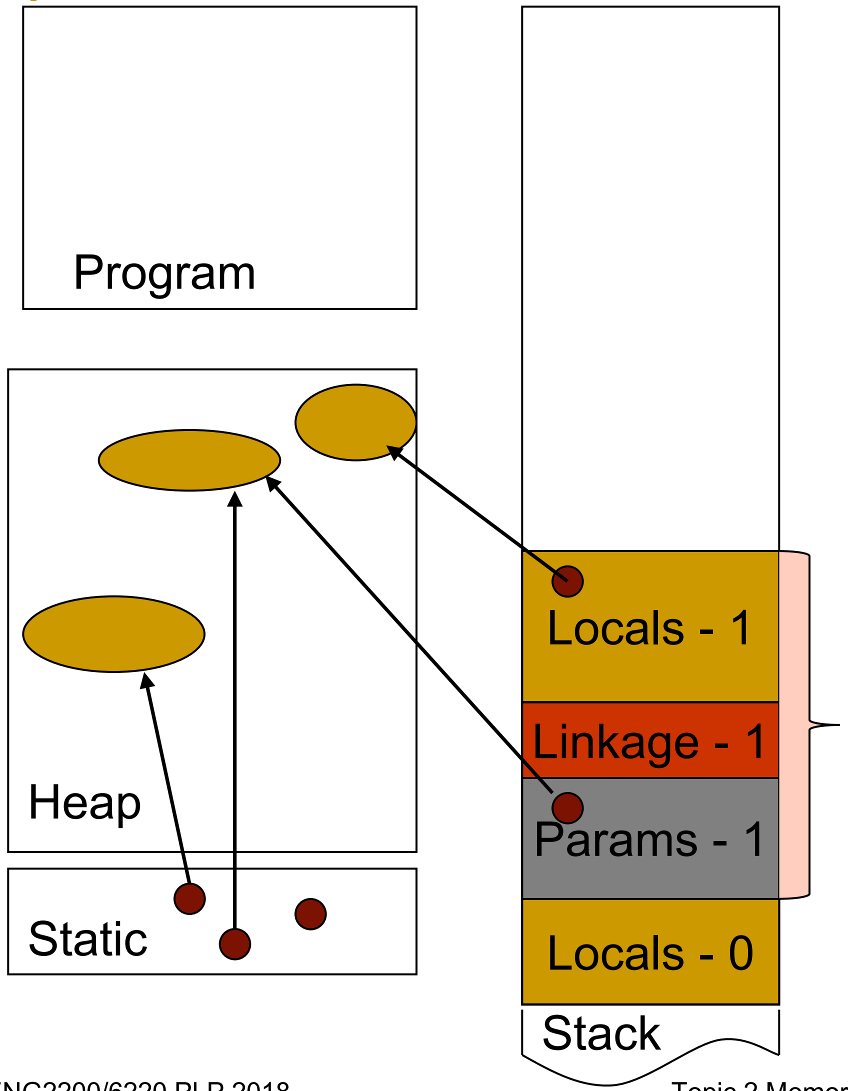
The class methods that operate on the class data also carry the similar designation of static

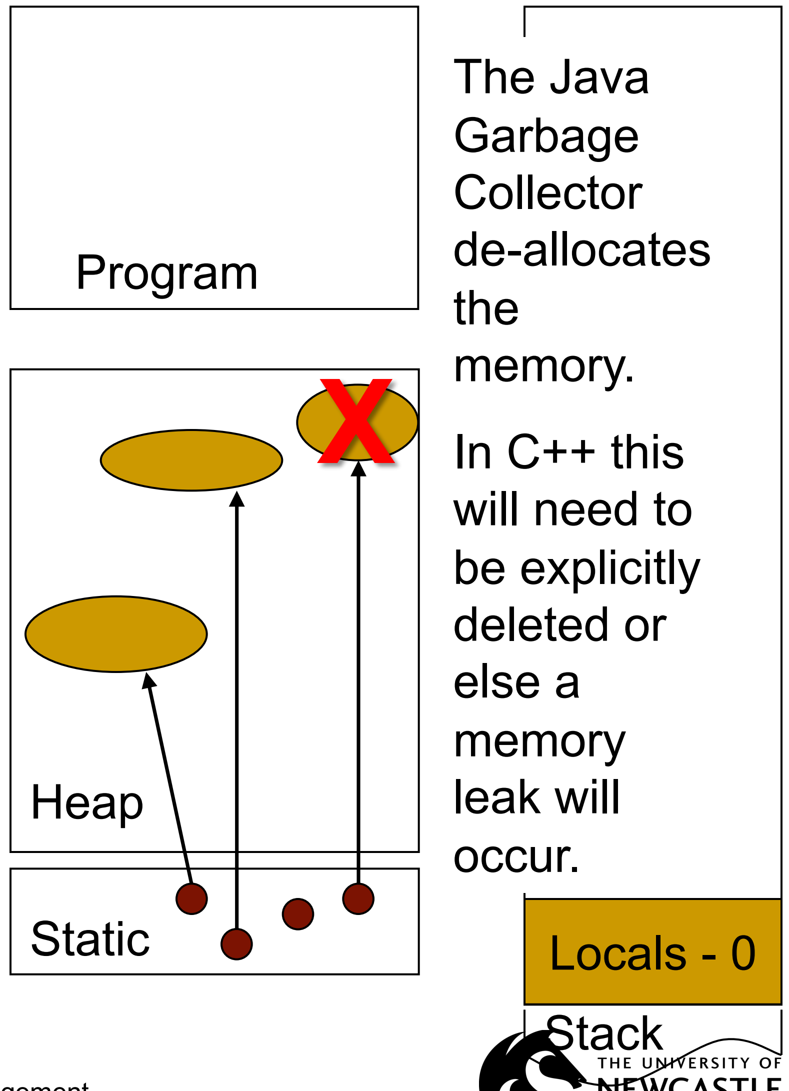
A method (non-static) which accesses class data, accesses exactly the same memory location(s) irrespective of which object is executing the method

Procedure/Method Calling



Java vs C++ Heap Memory



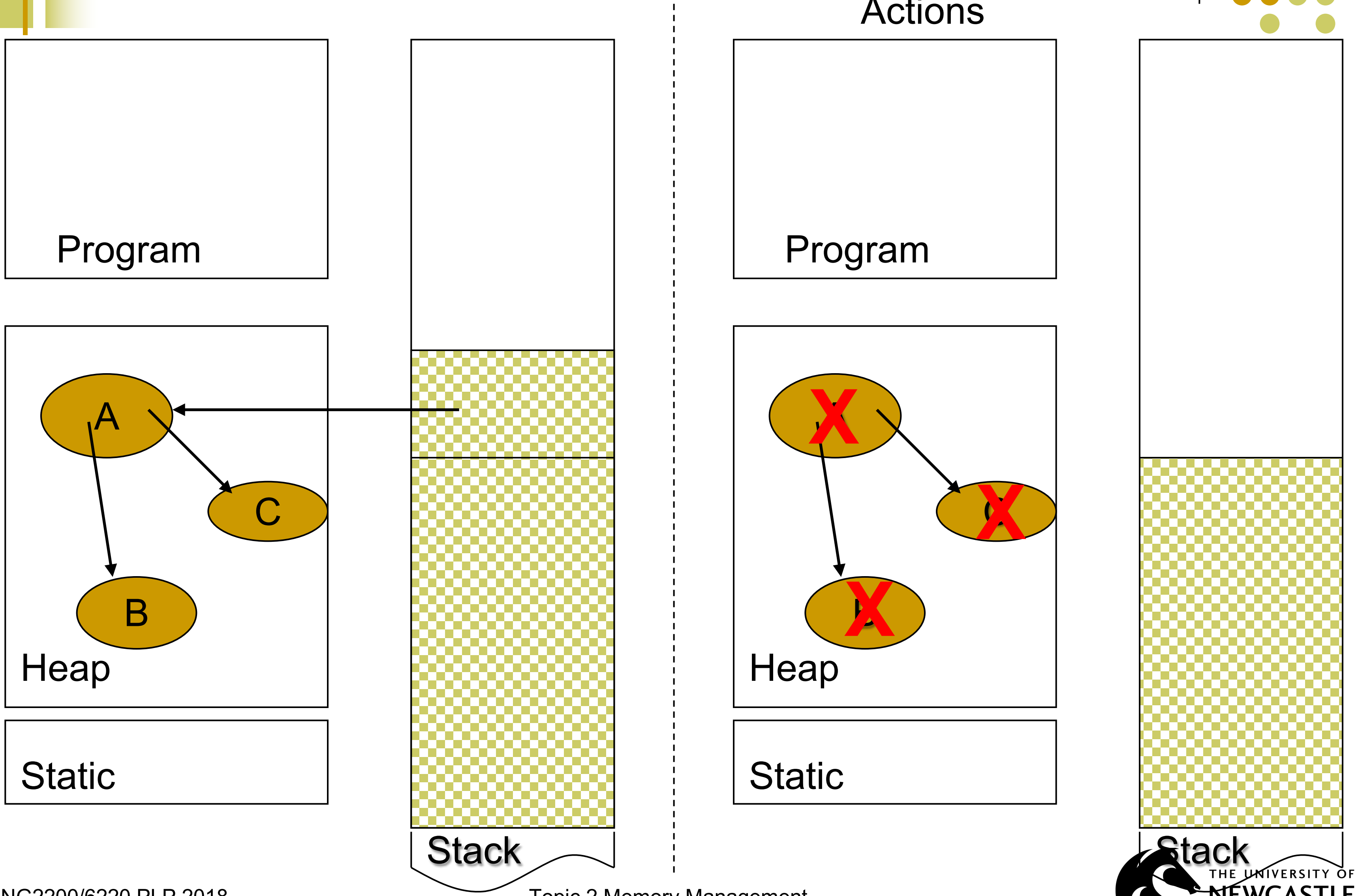


The Java Garbage Collector de-allocates the memory.

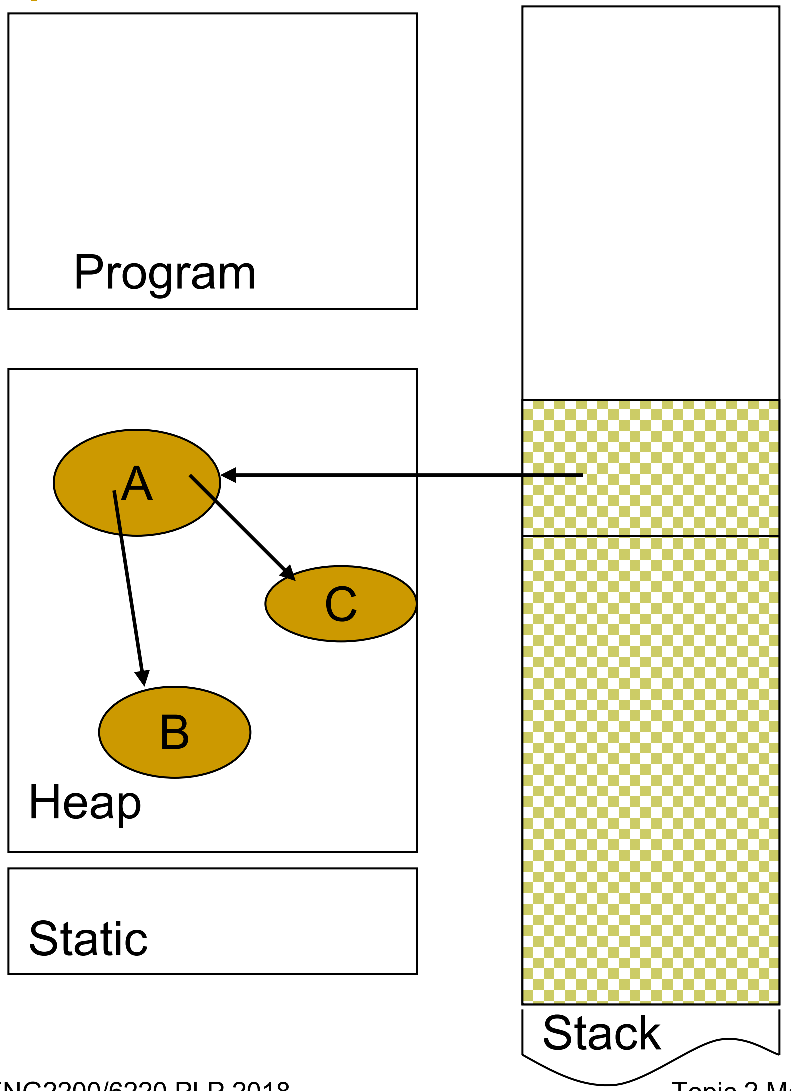
In C++ this will need to be explicitly deleted or else a memory leak will occur.

Java Heap Retrieval

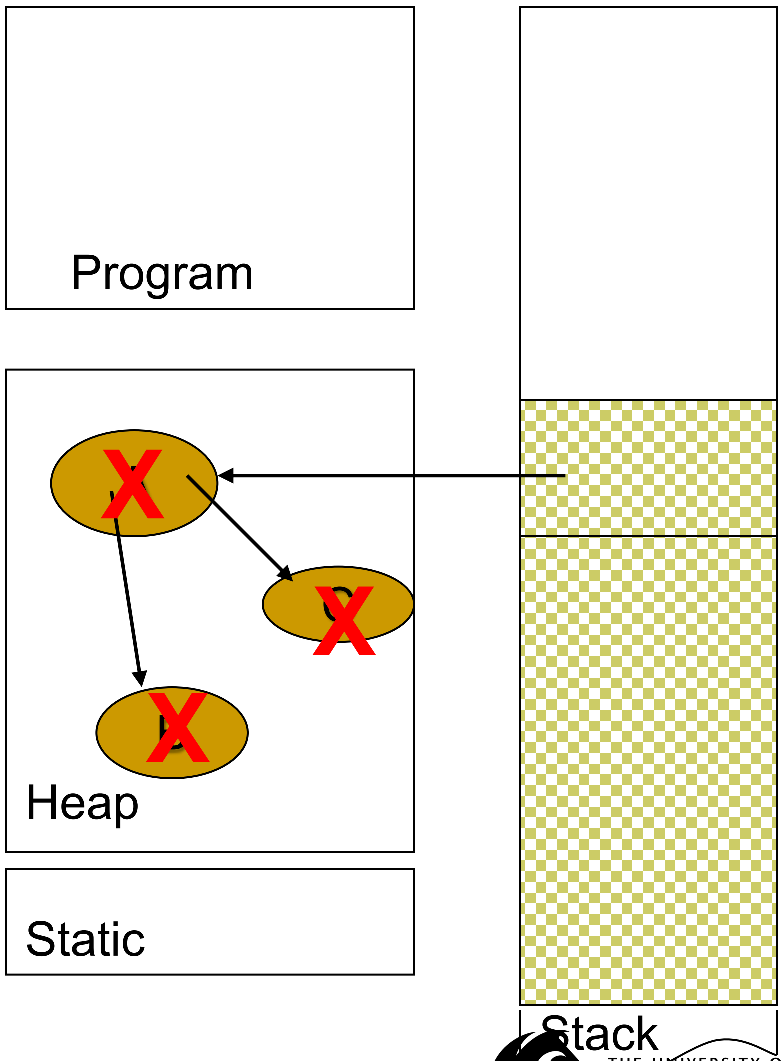
Garbage Collector Actions



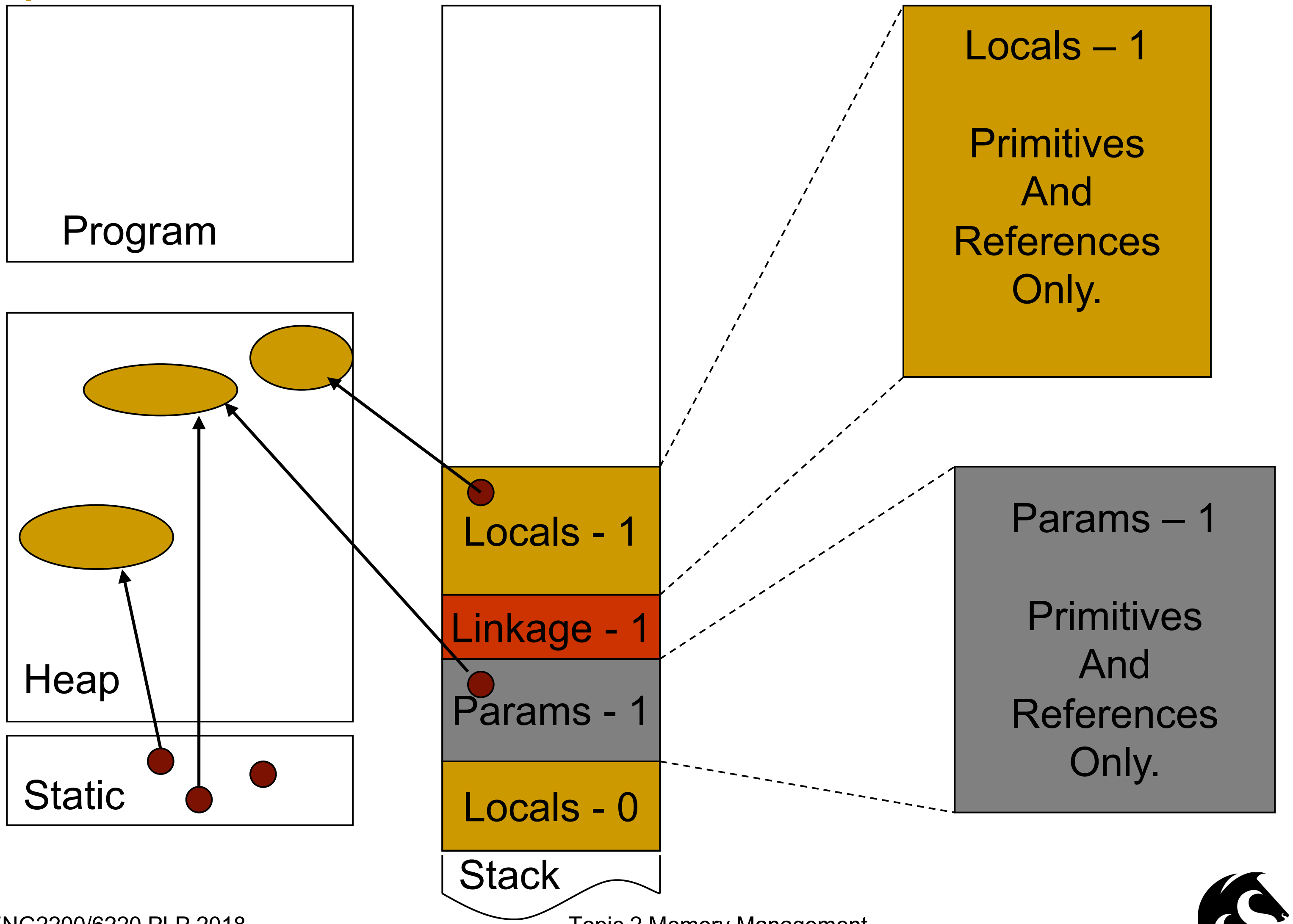
C++ Heap Retrieval



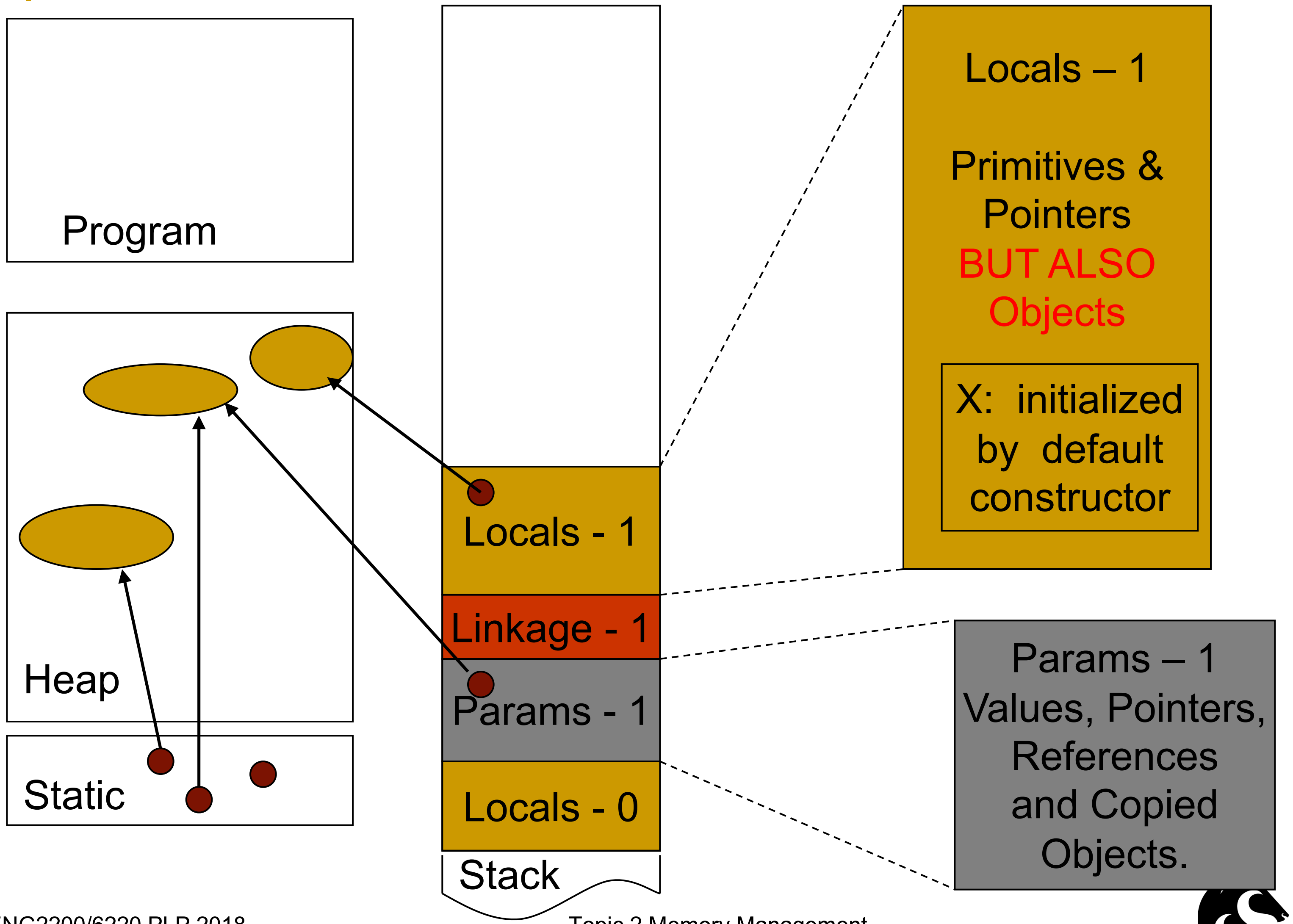
C++ Destructor Method Actions



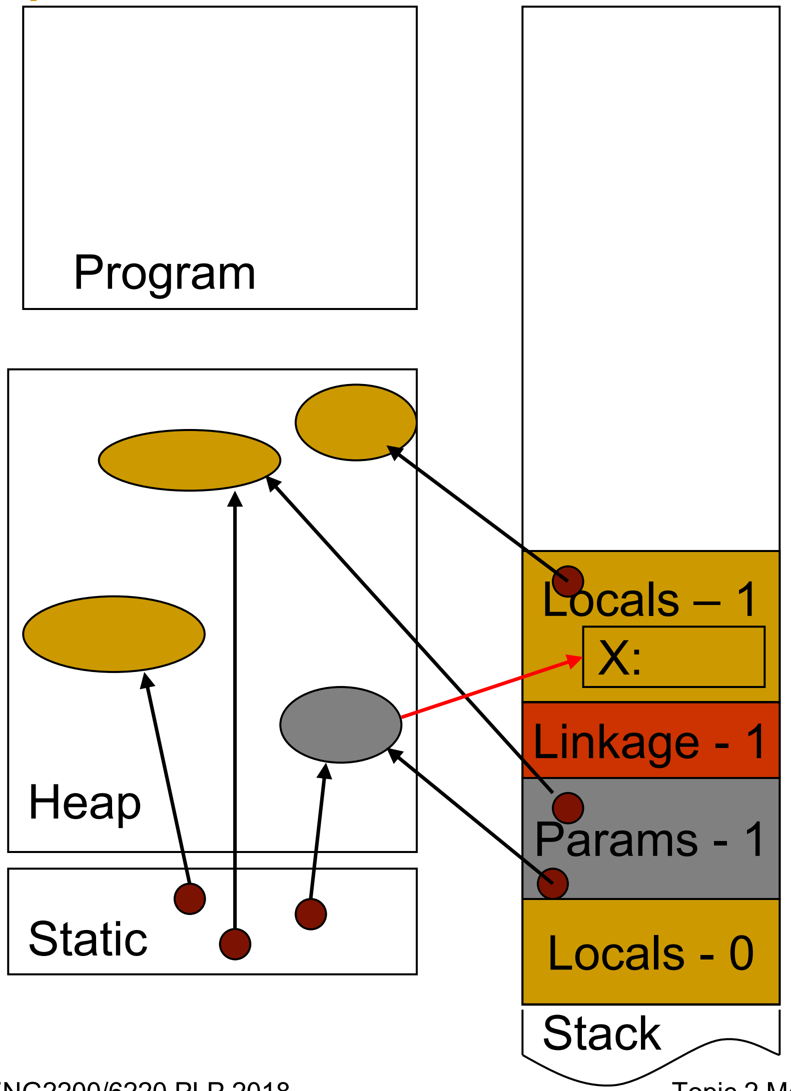
Java Stack Memory



C++ Stack Memory



C++ Stack Memory Objects



<u>Implications of object data residing on the Stack in C++</u>

What happens to them when the method returns?

What about their internal pointers?

What about returning pointers to them?

What about storing a pointer to them back into a pass-by-ref or pass-by-pointer parameter?

Program control of the heap implies complete programmer control of all data, otherwise (big) problems will occur.

Java Garbage Collection

Java is one of a group of languages commonly known as managed languages – part of this is automatic deletion of objects

Used for returning de-referenced objects to the usable heap storage

- Aims to find objects that would be memory leaks in a language like C++ (if they were not explicitly deleted)

Can use significant time

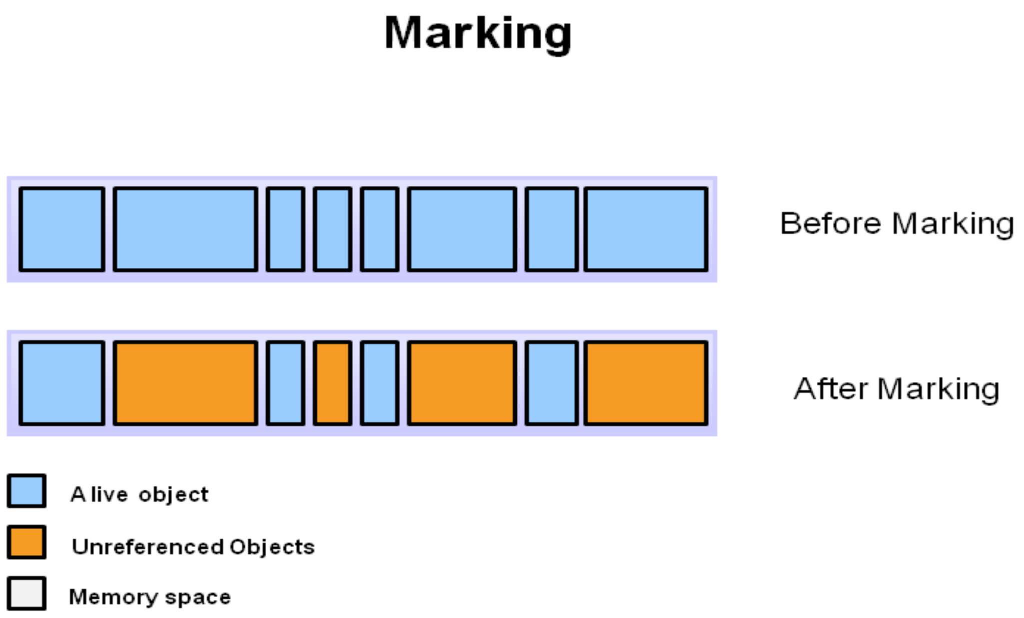
- A full garbage collection on a large application might take as much as 2 seconds on a modern processor
- Java therefore needs to have an efficient mechanism to alleviate this, and special provisions are needed for real-time systems
- Most applications are not sensitive to this

Reference counts are not enough

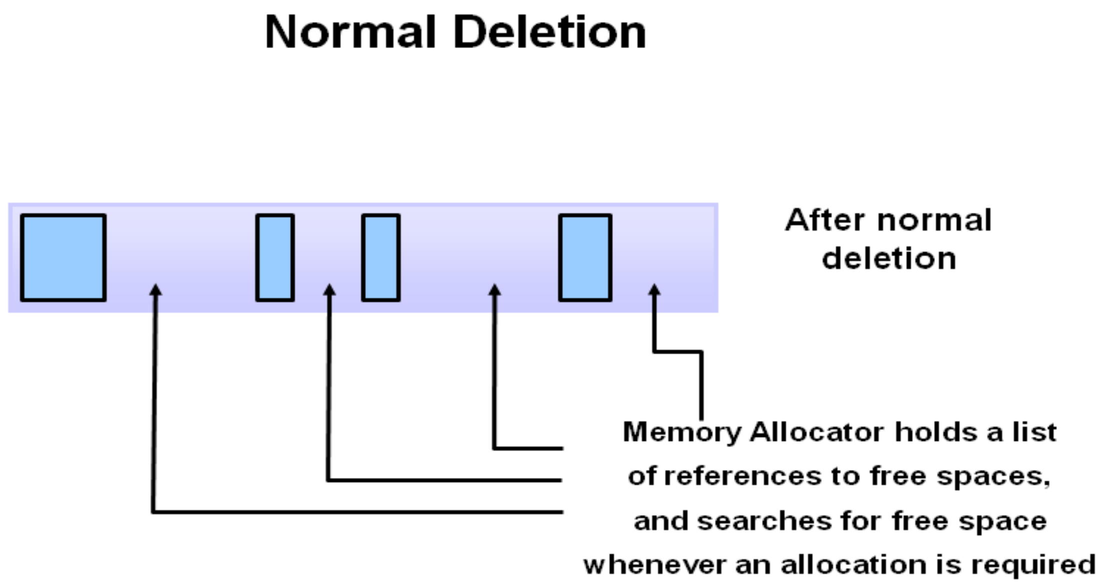
- It is possible to have circular references

Lots of research goes into efficient garbage collection

Mark as a Prelude to Sweep

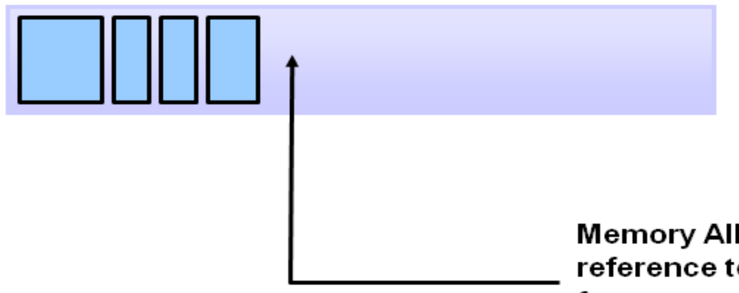


Sweep to Delete


Normal Deletion
After normal deletion
Memory Allocator holds a list of references to free spaces, and searches for free space whenever an allocation is required

Compaction

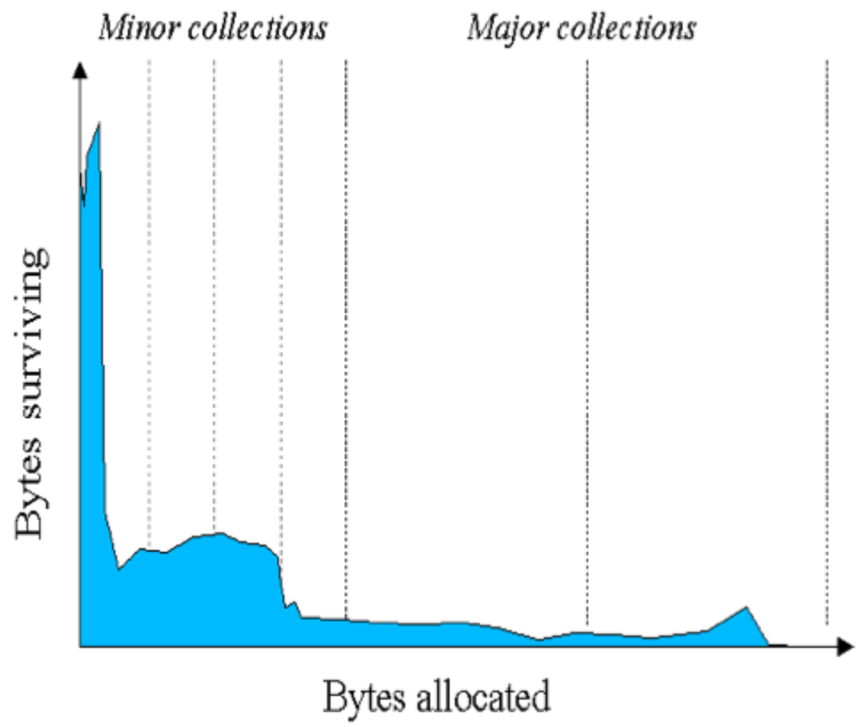
Deletion with Compacting



After normal Deletion with compacting

Memory Allocator holds the reference to the beginning of free space, and allocated memory sequentially then on.

Typical Lifetime Distribution



Time (t) since allocation

The lifetime of most objects is quite short.

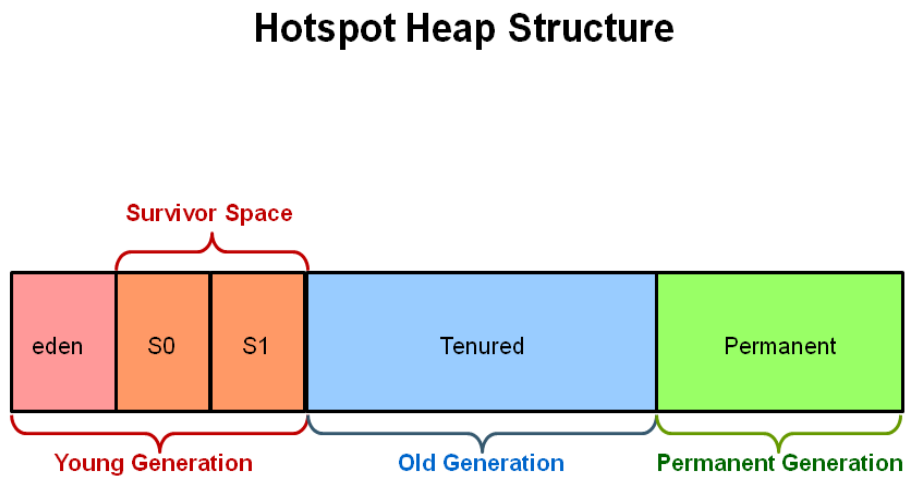
Garbage collection that treats all memory as equivalent will be inefficient.

Garbage collection of recently created objects is most likely to find the most objects ready for deletion.

Once an object has a reasonably long lifetime it is less likely to require deletion.

Efficiency depends very much on the execution profile of a particular program.

The Java Solution – Generational Garbage Collection



The Young Generation can be collected often as it will take a short(er) amount of time. When its collection does not retrieve enough memory or when the tenured area is full, then a FULL collection of the Old and Permanent Generations can be done (longer time needed, but done less often).

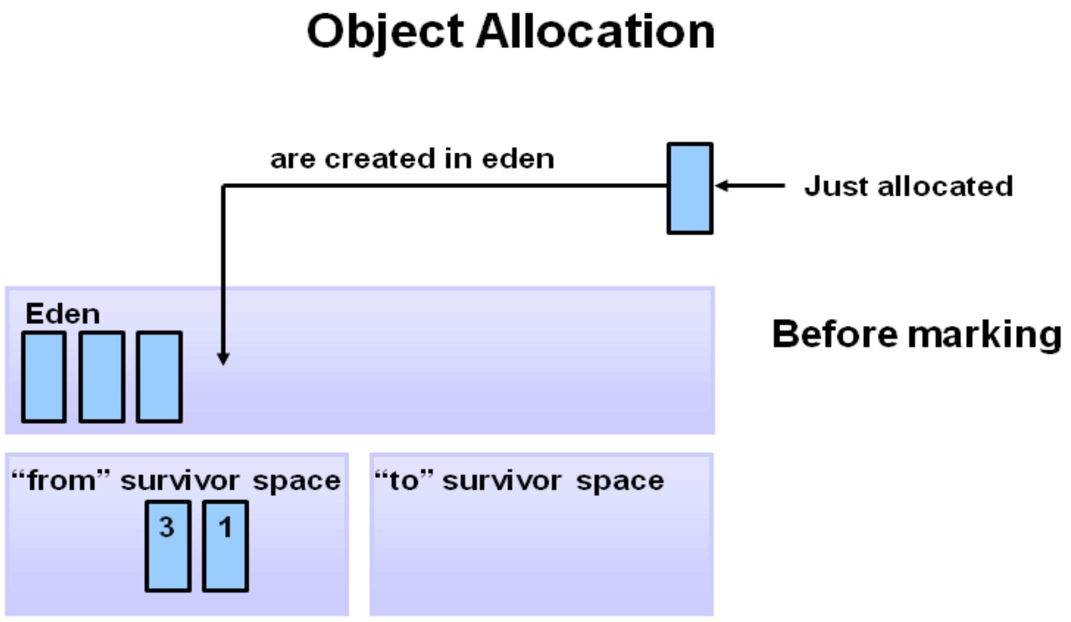
Objects which survive a sweep by the garbage collector eventually get promoted to a status that ensures that they get checked less often.

Promotion parameters can be set for any particular application

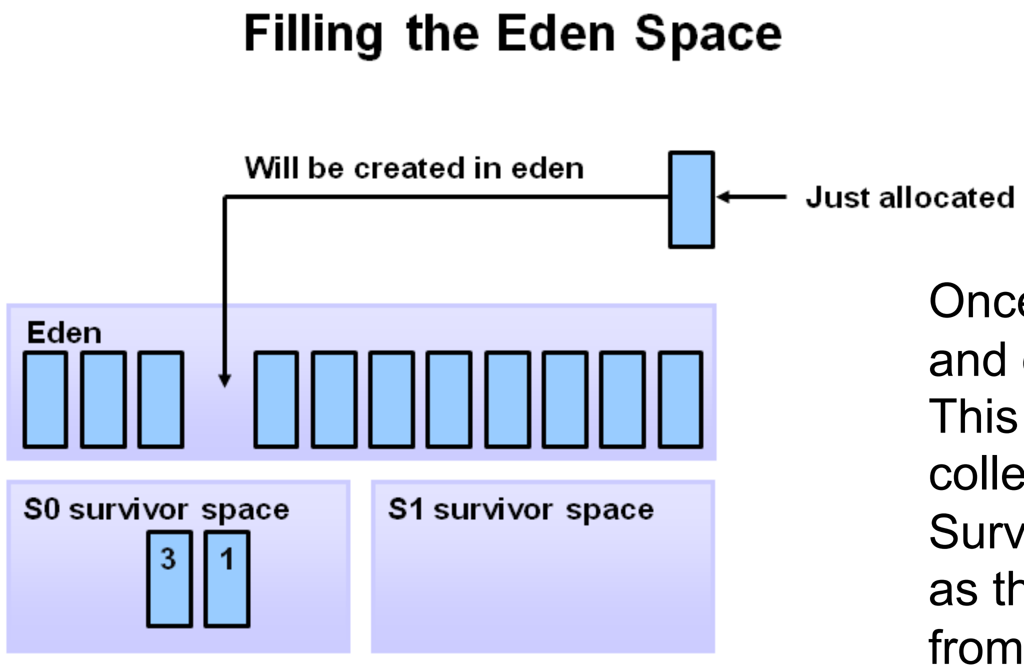
Permanent generation holds class meta-data (Metaspace Java 8)

THE UNIVERSITY OF NEWCASTLE AUSTRALIA

Eden – Where all is newly created



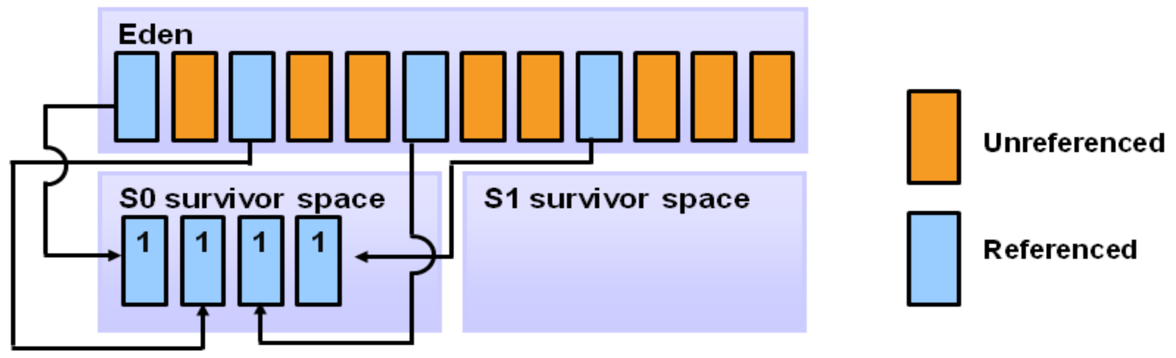
Eden – Where all is newly created



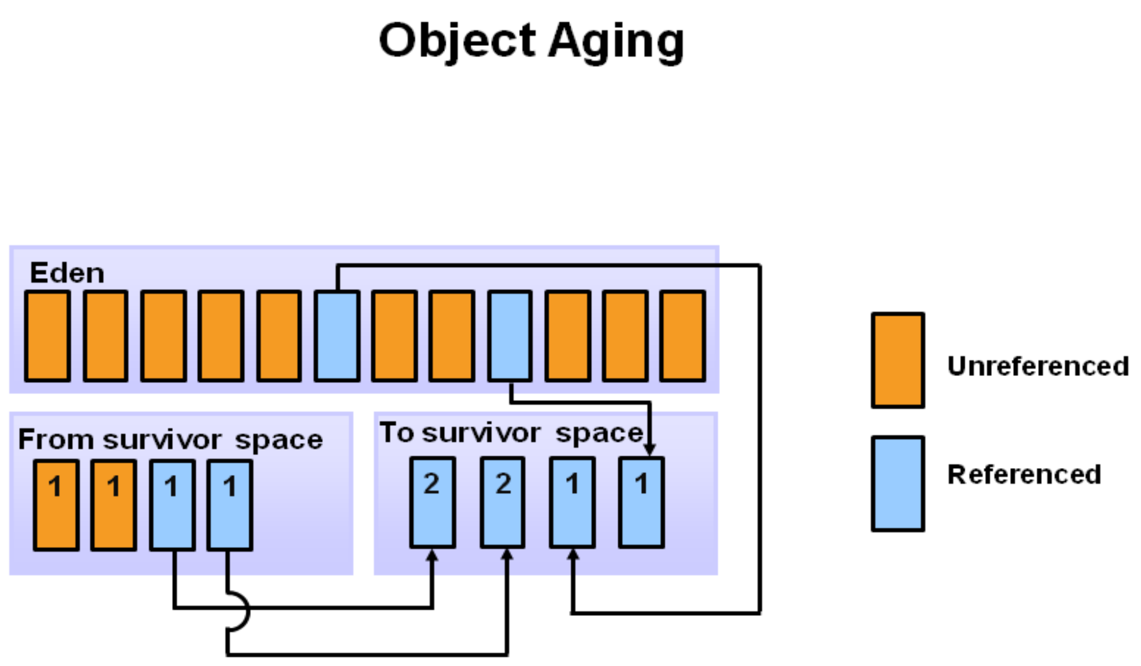
Once Eden is full, a sweep and delete is required
This is called a minor garbage collection.
Surviving objects are promoted, as they survive further sweeps from within the survivor space, objects are eventually promoted to the "Older Generation" or "Tenured" status.

Sweep – Identify Candidates

Copying Referenced Objects

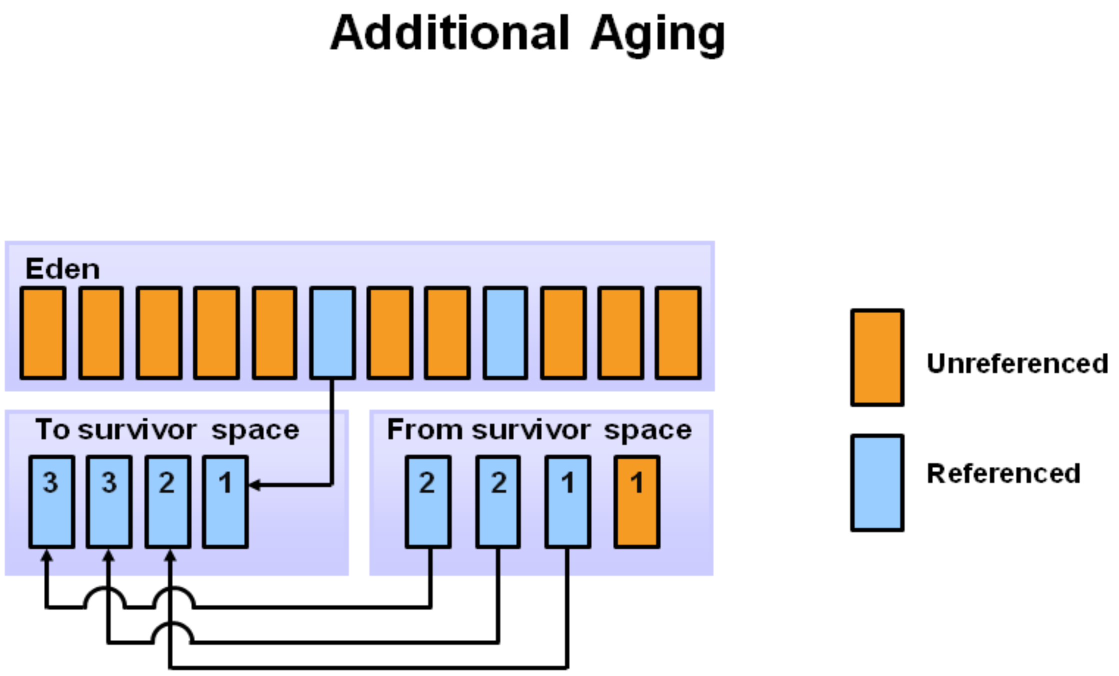


Toggling the Survivor Areas (1)



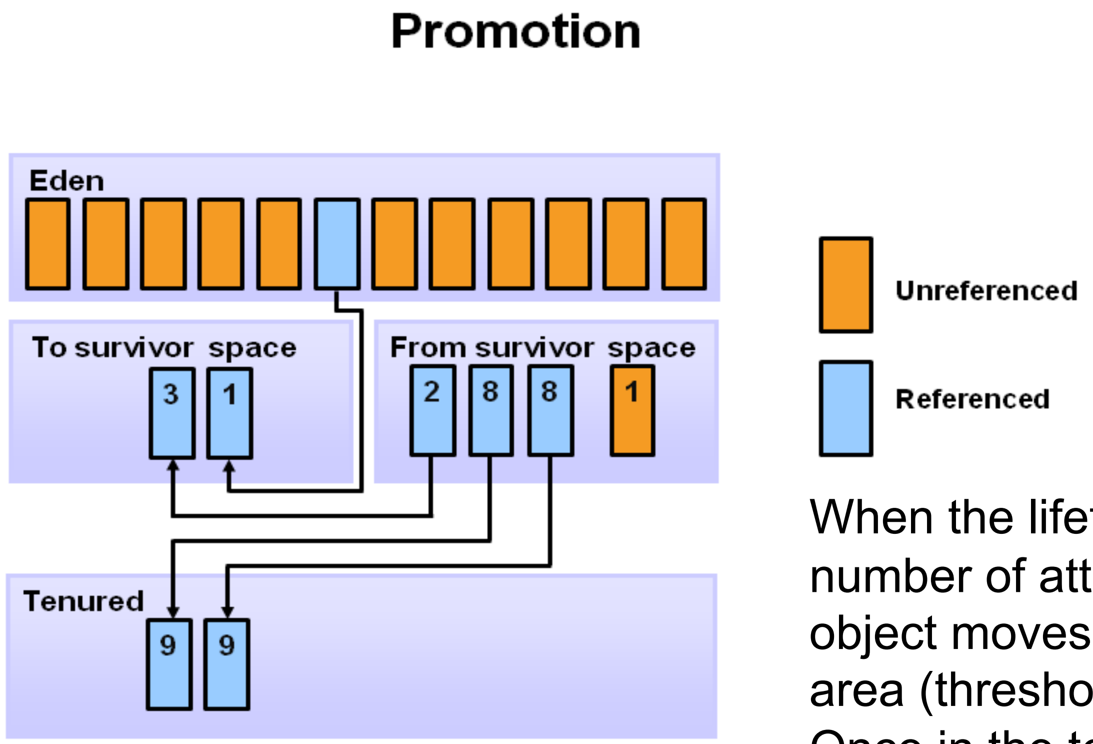
Note that a survival count (counting the number of times the object has survived a sweep) is kept during the toggling process. Both the *Eden* area and the *From Survivor Space* are checked and moved to the *To Survivor Space* with the correct count.

Toggling the Survivor Areas (2)



Note that a survival count (counting the number of times the object has survived a sweep) is kept during the toggling process. Both the *Eden* area and the *From Survivor Space* are checked and moved to the *To Survivor Space* with the correct count.

Promotion to Tenured Space

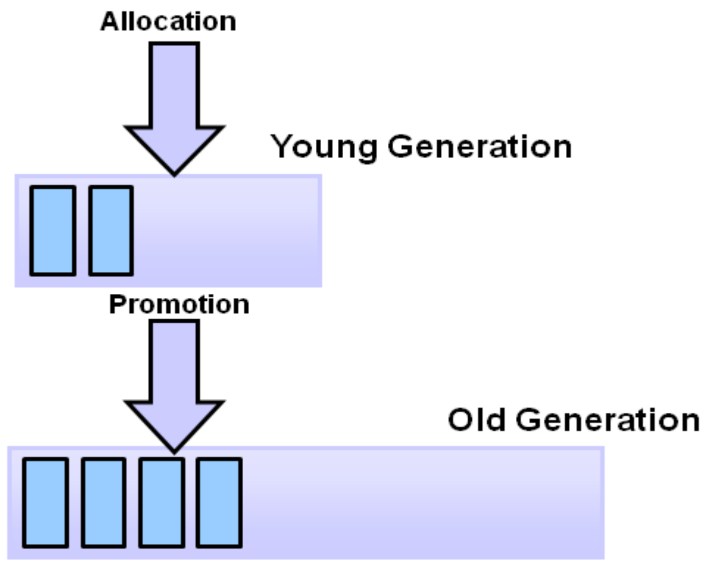


When the lifetime exceeds a certain number of attempts to delete, the object moves to the older generation area (threshold is 8 for this example). Once in the tenured space the object is checked for deletion far less often, that is, only when a full garbage collection is done.

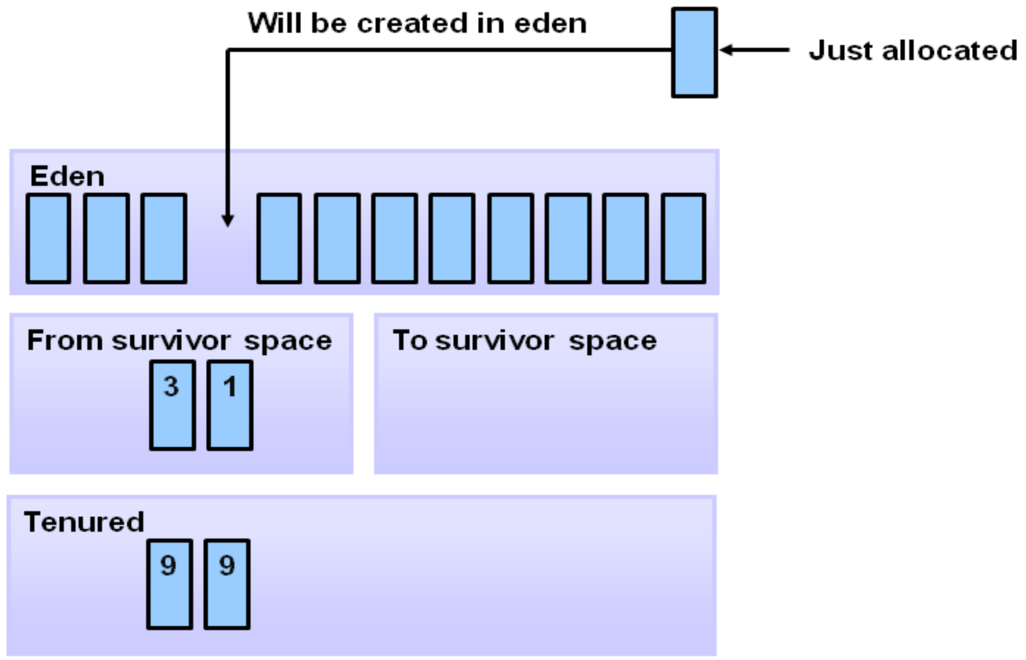
Summary of Promotion

Eden, S0, S1

Tenured



GC Process Summary



The C++ Answer – Smart Pointers in C++11

Developed and extensively tested by the Boost Group (boost.org) – allows objects to be managed

- A special set of classes invoked through **#include <memory>**
- Only brought into C++ (in the 2011 standard) after years of testing
- A set of template classes making extensive use of operator overloading, also supporting inheritance relationships

Provides consistent ownership of heap objects and automatic deletion of heap objects when references to them go out of scope, (otherwise leaving them prone to memory leaks or dangling references)

Smart pointers are objects in their own right, but can be created with their managed object to cut down on memory allocation overheads

Smart Pointers in C++11

`shared_ptr`

- Implements shared ownership – the general case
- For regular placement of objects into (potentially multiple) containers and regular associations between classes

`weak_ptr`

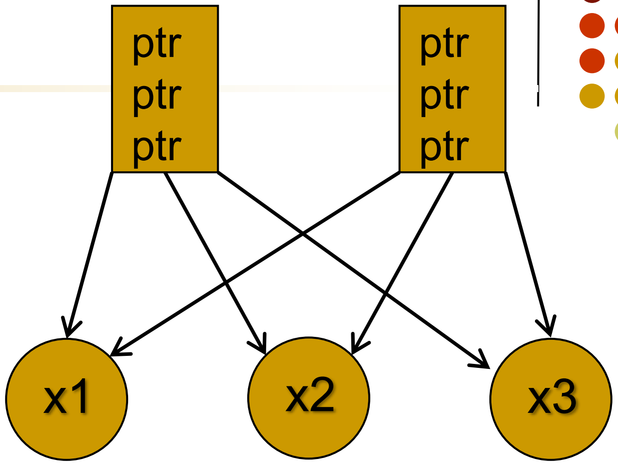
- Supports shared_ptr's in handling circular references
- Does not own the object at all – simply *observes* the object

`unique_ptr`

- Implements unique ownership – only one smart pointer may own an object at any time
- Provides safe deletion and transferability in cases of single ownership of objects

shared_ptr

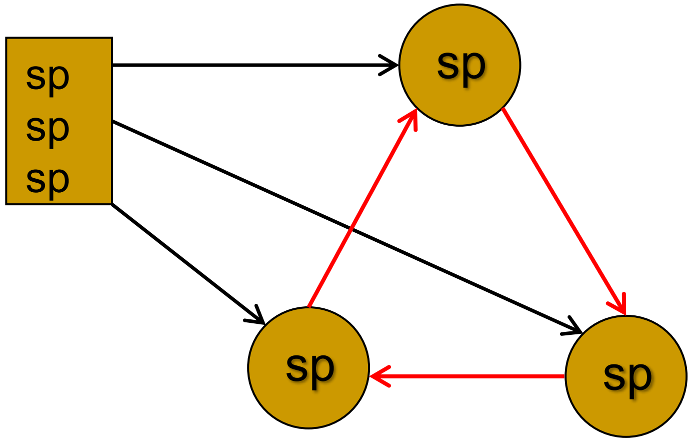
With complicated relationships between objects, it can be difficult to decide when a heap object needs to be deleted.



Containers of pointers

Smart pointers (shared_ptr's) keep a reference count that allows an object to be deleted when its reference count becomes zero (ie no other object making use of it → no-one owns it).

However circular references to objects will leave the objects alive as a closed system, even though none of them can be referenced from outside (a memory leak) and so they need extra support



Container of smart pointers each with a smart pointer

shared_ptr(s) plus weak_ptr(s)

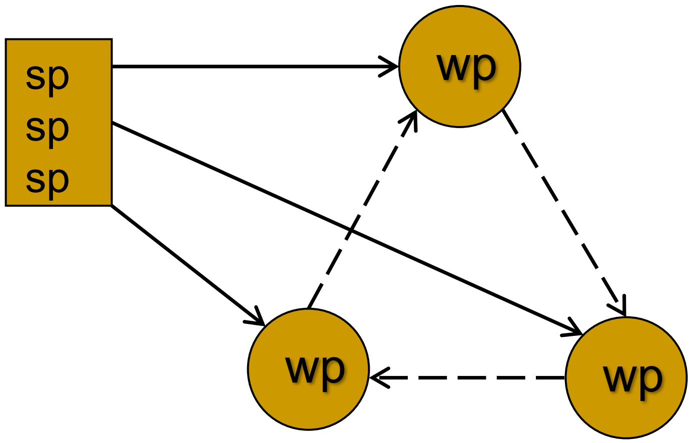
A set of template classes making heavy use of operator overloading.

A **`weak_ptr`** references an object but cannot help to keep that object alive – effectively being just an observer.

They have a highly restricted set of methods/operators available.

When the **`shared_ptr`** references no longer point at the objects the ring of objects will be deleted.

However the **`weak_ptr`** itself remains in place and can be queried as to whether the object is still alive or not.



Container of smart pointers referencing a ring of objects each with a weak pointer

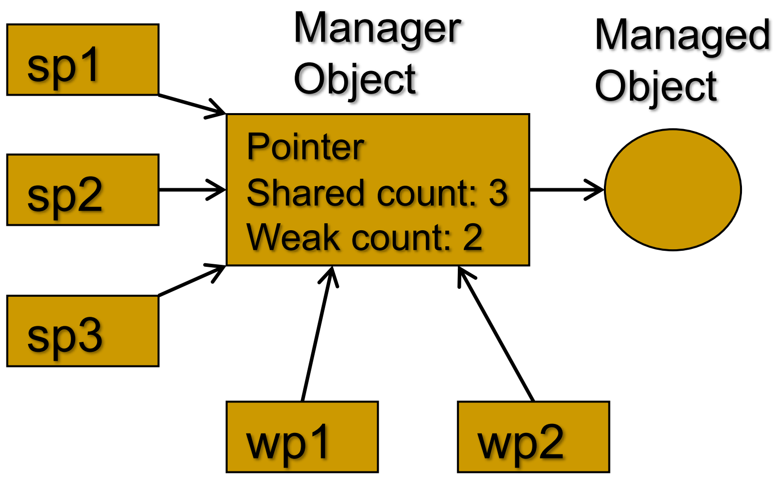
How they work

The managed object is created on the heap and given to the constructor for the first shared_ptr object which creates the Manager Object.

The Manager Object holds the only native (raw) pointer to the Managed Object.

Subsequent assignment or copying of the sp1 into either sp2 or sp3 will simply increase the shared count and refer to the same Manager Object.

weak_ptr objects are only created by assignment or copy of either a shared_ptr or another weak_ptr.



Shared count of 0 triggers deletion of the Managed Object but the Manager stays alive so that any remaining weak_ptr can be asked if it the object is alive.

The manager object is deleted when both counts are 0.

shared_ptr operations

Just how well do you understand operator overloading in C++?

Extensive use of operator overloading within templates

- The programmer has access to a raw pointer to the managed object (via the get() method) but should "never" be needed.

Assignment and Copying

- `=` operator and a copy constructor exist for shared_ptr
- A weak_ptr can only return a new shared_ptr via `lock()`.

Dereferencing

- Operators such as sp1`->`do_something(…) and (`*`sp2) exist

Comparison and Testing

- `==` and `!=` and `<` actually compare the internal raw pointers, and shared_ptr provides a conversion to `bool` type so that existence of the managed object can be checked.

Casting is supported, as are base class pointers via inheritance.

Restrictions on shared_ptr and weak_ptr

1. Can only be used on objects created on the heap with `new` and that can be deleted with `delete`.
   - Trying to delete objects on the stack will give a run-time error.
2. There must be only one manager object for each managed object – when an object is first created it should be immediately given to a shared_ptr to be managed.
3. Avoid using raw pointers and smart pointers to refer to the same objects or else there is serious risk of problems with dangling pointers and double deletions.
4. Special consideration is needed for a `this` pointer, by constructing the object using a weak_ptr to *itself*.

These still rely on good behaviour by the programmer.

unique_ptr

Enforces exclusive ownership of the managed object.

- Copy Construction and Copy Assignment of the unique_ptr are not allowed – consequently a unique_ptr must only ever be passed to a function by reference.
- unique_ptr implements *Move Semantics*, via a move constructor and move assignment function, which transfer ownership from the original owner to the new owner.
  - E.g. with unique_ptrs p1 and p2 and p1 owning the object
  - p2 = p1; // gives a compile error - copy asgn
  - p2 = std::move(p1) // p2 owns the object, p1 owns nothing

Smart Pointers vs Garbage Collection

Smart Pointers in C++11 still require proper use by the programmer and can be subverted by *bad* programming.

- BUT STILL - A big step forward for C++11 reliability.
- Bad Programming? Something that works but doesn't quite do what it should *ALL* The Time.
- Corporate programming standards become essential as team programmers trust each other more and more.

Cascaded deletions can still result in varied running time.

Garbage Collection in Java is significantly complex and overall performance depends on the runtime and memory referencing behaviour at the individual program run level.

- Doesn't fix everything – if an object holds a resource (eg. A file lock), then we have all the same problems that C++ has with memory (finalise method?).

References

http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html

- Oracle Corporation [accessed on March 7, 2018].

www.umich.edu/~eecs381/handouts/C++11_smart_ptrs.pdf

- David Kieras, EECS, University of Michigan [accessed on March 7, 2018].