

Lecture 25 – Memory Management

J. Zarnett

jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

March 14, 2015

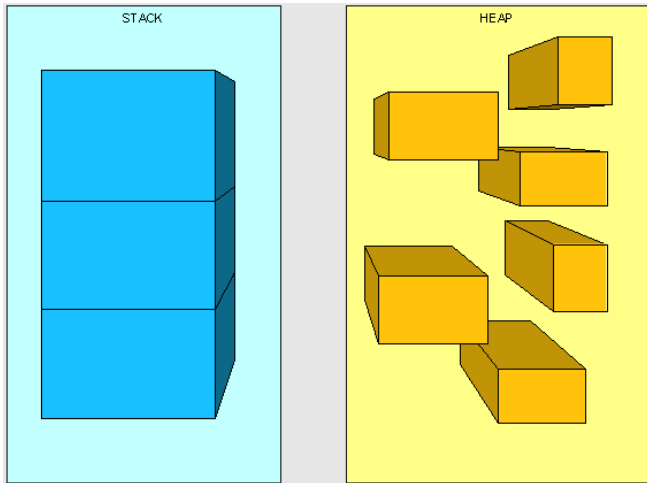
Acknowledgments: W.D. Bishop

We often talk of storing a variable in memory, but this glosses over how exactly memory works, and this is a non-trivial topic.

A program has two areas of memory: the **stack** and **heap**.

Both are simply designated areas of memory related to the running program, but they are modelled and used in different ways.

Stack vs. Heap: Visually



Source: http://www.c-sharpcorner.com/UploadFile/rmcochran/csharp_memory01122006130034PM/csharp_memory.aspx?ArticleID=9adb0e3c-b3f6-40b5-98b5-413b6d348b91

The stack keeps track of what is executing in our code.

Imagine the stack as a series of boxes stacked one on top of another.

Every time we call a function, we put another box on the pile.

This box, actually called a **frame**, contains the parameters, local variables, and the return value of the function.

We can only access the box currently on top of the pile.

Value types (like `int`, `double`, and `struct`) are allocated on the stack.

Imagine we are in a method `method1()`.

A statement like `int i = 4;` allocates the variable `i` on the stack.

It is of `int` size and appears on the top of the stack.

It is then available within `method1`.

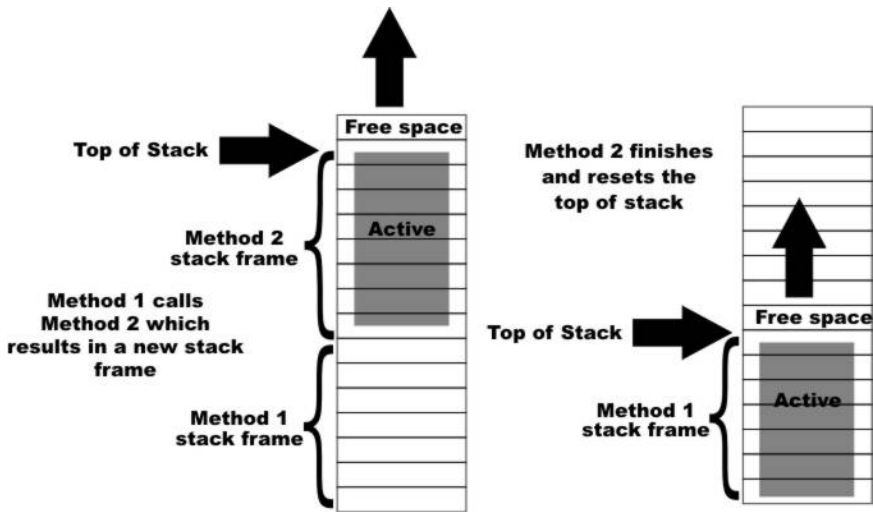
If the next statement is `int j = 0;` then another variable is allocated at the address immediately above the location for `i`.

These variables exist only as long as `method1()` is running.

When a function returns (finishes execution), the box on top of the stack is thrown away and we can use the stuff in the box below.

Thus, stack data is discarded aggressively (i.e., as soon as possible).

You will learn more about how the stack works and is structured in ECE 254 (Operating Systems).



Source: <http://www.i-programmer.info/ebooks/deep-c/363>

Stack space is limited in practice. Recall that recursion can result in a Stack Overflow error: this is what happens when we run out of space.

This was the result of infinite recursion: calling the function too many times (stacking too many boxes).

In normal operations, however, we will not run out of stack space.

But what if we allocate a reference type...?

Reference types are allocated on the heap.

If a reference type is allocated (the `new` keyword is used), the object instantiated is allocated on the heap.

This applies also to strings, arrays, and other classes.

The internals of the heap are beyond the scope of the course.
But you will examine it later in ECE 250.

Discarding Objects in the Heap

We saw we don't have to de-allocate a variable allocated on the stack.
The end of the function takes care of that for us.

Heap objects, however, can outlive the functions that created them.

Heap space is also limited by the amount of memory available on the machine. This is typically much larger than the space for the stack.

Still, we need a way to discard things.

Unused objects in memory don't have any value, and they prevent other applications from using that memory.

Recall from earlier the concept of variable scope.

If a reference to an object goes out of scope, that reference can no longer be used to access the object to which it refers.

If there are no valid references to an object, that object is no longer accessible and cannot be used for anything.

Such an inaccessible object is considered “garbage”.

The C# approach to cleaning up garbage is called **Garbage Collection**.

Garbage objects are not discarded aggressively.

- They will be cleaned automatically by the garbage collector.

- The garbage collector runs when the system chooses.

- This may “waste” some memory, but can improve performance.

There is no simple way of figuring out when an object on the heap is not needed anymore, so the process of garbage collection is complex.

How does garbage collection work?

- Memory is allocated whenever an object is constructed
- Once the scope of an object ends, the object becomes “garbage”
- Programmers do not indicate when an object is no longer needed
- The memory associated with garbage objects is made available to the system when garbage collection is performed

The system decides when garbage collection is performed; the programmer does not indicate this.

In languages like C++, developers also do memory deallocation:
The language and run-time do not implement garbage collection.
This can easily lead to **memory leaks**.

A memory leak is when some area of memory remains allocated even though it is no longer needed.

Over time, a memory leak can reduce the performance of the computer by reducing the amount of available memory.

If available memory is exhausted, this may lead to a crash.

Garbage collection has numerous advantages:

- Simplifies the design of complex applications
- Enhances code quality by eliminating memory leaks
- Enhances developer productivity
- Permits program to optimize memory utilization through compaction

It also has some disadvantages:

- Limits developer control over the de-allocation of memory
- Introduces overhead to monitor when objects become garbage
- Runs at unpredictable times (a problem in real-time systems)
- Often temporarily halts the program executing while it cleans up (also a problem in real-time systems)

C# uses references to keep track of heap allocated objects.

C# automatically manages memory which means that reference type objects can be relocated (moved in memory) dynamically.

Objects are allocated in the first available memory address where an appropriately-sized block of memory is found.

Example: the first address may have space for an `int[]` array of capacity 10, but the request is to allocate an array of capacity 20.

Over time, some objects go out of scope and become garbage.

The garbage collector frees up memory containing garbage and it moves objects to make larger contiguous blocks of free memory.

To allocate a new object, a block of memory of the size of that object will be needed. If none is found, that object cannot be allocated.

References have a memory address associated with them.

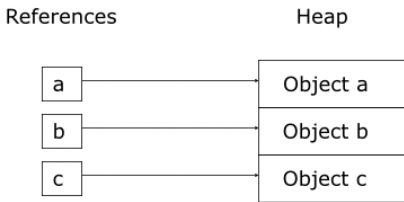
When an object is relocated, the value is changed to indicate the new memory address of that object.

Heap Memory Management Example

In the following code, three reference objects are instantiated:

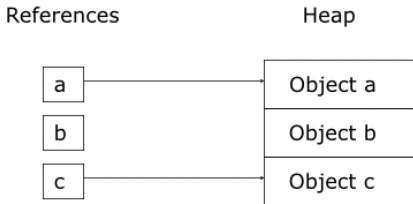
```
Object a = new Object( );  
Object b = new Object( );  
Object c = new Object( );
```

There are three references (a, b, c) that refer to three distinct locations in the heap.



Heap Memory Management Example

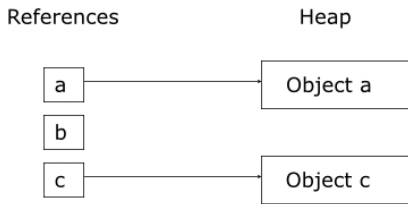
Now: `b = null;`



Object b has gone out of scope and is no longer reachable.
It is garbage and may be collected when the garbage collector runs.

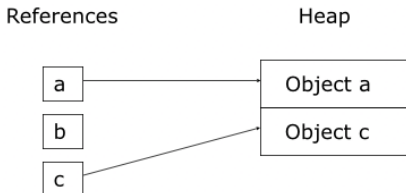
Heap Memory Management Example

When garbage collection runs, Object b is no longer there; its location is now free memory.



Heap Memory Management Example

When garbage collection runs, it may “compact” the objects in memory to reduce the spaces between allocated objects.



This also produces larger contiguous blocks of free memory.

Why does the garbage collection process compact memory?

To avoid the situation where there is enough available memory free in the system, but no single contiguous block to hold an allocation.

If there are two blocks that can hold `int[]` arrays of capacity 10, we could allocate an array up to a capacity of 20 if they are combined.