



Dynamic memory allocation

- Dynamic memory allocation is very useful, and is part of the philosophy of using C++ effectively, but is fraught with dangers
- The standard `new()` and `delete()` functions, though widely used, are highly unsafe!
- We need to make dynamic memory allocation safer while maintaining as much transparency as possible for the programmer.

Types of allocators

- There are two main allocation schemes:-
 - Heaps
 - A single area of memory provides blocks of varying sizes, as requested by the user on each occasion
 - Pools (sometimes called Partitions)
 - A single pool holds a number of blocks which are all the same size, specified when the pool is created
 - A user can request only one block in a single allocation request
 - To allow a reasonable match with actual user requirements, there may be several pools available, each with a different block size

The C++ allocation scheme

- The C standard library provides a heap allocation scheme
 - malloc() is most often used to request memory
 - calloc() and realloc() also exist
 - free() is used to free memory back to the heap
- C++ provides:
 - new(), which typically calls malloc() by default
 - delete(), which typically calls free() by default
- As with other C++ functions, new() and delete() can be overridden

Allocation problems

- Shortage of memory
- Memory leaks
- Allocator performance
- Allocator robustness
- Memory fragmentation

Shortage of memory (exhaustion)

- Unless caused by memory leaks (next slide), exhaustion is usually a preventable design problem
 - The design must be altered:
 - to provide more memory or
 - to need less memory
- Sometimes, though, the design calls for “graceful degradation”
 - Exhaustion is detected and dealt with at run time
 - An example would be a telephone exchange in which the number of calls which can be simultaneously processed depends upon how much physical memory is actually fitted

Memory leaks

- A memory leak is a simple coding error
 - Some allocated memory should be freed after use, but is not
- Finding a memory leak can be very difficult but fixing it is easy

Allocator performance – Heaps

- Heap processing is slow
 - On allocation, the following must be done:-
 - Find a block big enough to meet the request
 - Extract the appropriate number of bytes
 - Re-link the remaining fragment into the free-list
 - On freeing, the following must be done:-
 - Find the free blocks (if any) which lie physically adjacent in memory to the block now being freed
 - Merge the adjacent blocks into one
 - Adjust the links in the free-list to link in the merged block
- Performance is not just slow; it is also variable
 - The time taken is, for practical purposes, indeterminate

Allocator performance – Pools

- Pool processing can be fast:-
 - A pool can be most simply implemented as a singly-linked list of free blocks
 - A block is allocated by removing it from the head of the list
 - A block is freed by re-linking it to the head of the list (LIFO)
 - Suitable locking (mutual exclusion) must also be provided but, even so, the operations above are typically fast enough to be used even by Interrupt Service Routines (ISRs)
 - For use with ISRs, locking must be done by disabling interrupts, not by using mutexes

Allocator robustness

- Warning!
 - The standard library functions (malloc, etc.), as supplied with most compilers are inherently unsafe
 - They are not re-entrant
 - They provide no locking (mutual exclusion)
 - The default C++ operators, new() and delete(), rely on the above, and so are also unsafe
 - If the above functions are required, always use the standard library provided with the RTOS in place of the one supplied with the compiler
 - Better still, use pools to avoid fragmentation (next slide)

Memory fragmentation

- All heap allocation schemes have a big problem
 - As blocks of variable sizes are allocated and freed, in no particular order, the free memory becomes fragmented into ever-smaller (though more numerous) discontinuous blocks
 - Eventually the success of a request cannot be predicted
 - Even if there is sufficient total memory to meet the demand, there may or may not be a single block large enough to fulfil the request
 - Fragmentation gets worse over time
 - This is a big problem for embedded systems, which must often run, without reset, for very long periods

Preventing fragmentation

- Always use pools, never heaps
 - Exception: memory which is never intended to be freed can be taken from a heap
- Override the global `new()` and `delete()` operators
- Consider per-class `new()` and `delete()` operators
 - Quicker, as well as safer