C++ Embedded and Allocation(allocator)

sha@ase.au.dk
V0.9

Introduction

In this exercise we are going to different aspects of running C++ in an Embedded context. This includes looking at the difference between C and C++, as well dicussing which language constructs have a performance impact ending with creating out own *small object allocator*, that can be used with STL.

Exercise 1 Migrating from C to C++

Consider the two languages *C* and *C++*. From a historic point of view, the *C* language has always been and to some extent still is the best supported language on different embedded cpu architectures. The assumption for this exercise is therefore, that a migration from *C* to *C++* is to take place at some firm. You are the leading architect that has been part of the process in which it was chosen to migrate to *C++*.

Exercise 1.1 C versus C++

Since you have superior knowledge of *C++* as opposed to your colleagues, you are to give a talk on the benefits of migrating to said language. To ensure a proper setting for this the number of topics should not be more than 5.

This exercise is therefore to be a written text on the 5 topics.

- Why you chose exactly these
- What they contribute to design/implementation wise

Exercise 1.2 Language constructs

As in any given situation *nothing is perfect* neither is *C++*. As an extension to the above talk you are to give another on different aspects of the language one should be wary about using or that one should not using at all in an embedded context.

In this exercise, pick 3 topics and argue as to how you want them handled. This includes how they are used, and, if applicable, examples on how they can be misused. It could also be that their inherent nature makes for an unsatisfactory generated binary. Other reasons not mentioned here could very well be just as important...

Exercise 2 My Allocator

In this exercise the task is to create an allocator. Generally speaking the allocator in the standard library does a mighty fine job, however in certain circumstances it can be suboptimal thus enabling us to create a better implementation.

Containers such as `list`, `set` and `map` use an allocator in a certain fashion, they in fact allocate a large number of elements of the same size[1]. In contrast the general allocator was and is designed to create objects of various sizes. To do so the heap manager must know the size of

[1] Do note that the element size is *not* the same as the size of the template parameter they were instantiated with.

the different memory blocks resulting in an overhead of usual 4 bytes. In the scenario where the individual blocks are large the overhead is insignificant, however for small blocks this overhead may become unacceptable.

Functional wise the allocator from the standard must also find a block of free memory to use. However in our setup (*list* and friends...) we already know that the blocks to allocate have but one size. This means that we can optimize our allocator for one size only and thus hope that our implementation is an improvement.

Exercise 2.1 Pre-design/implementation

Consider these questions for an embedded scenario.

- What is memory fragmentation?
  - why is this particulary a problem for embedded devices?
  - in contrast, how do we handle it in normal desktop applications?
- By creating our own allocator, what may we achieve in an embedded context? and at the expense of what(probably)?

Exercise 2.2 Small Heap Object

In this exercise you are to create the template class `SmallObjectHeap`. Inspect the header file `MyAllocator/SmallObjectHeap.hpp`. In it you will find the the class `Page` that performs some important pointer arithmetic. The template class you are to implement is to be implemented after the class `Page` outside the `namespace details`.

Exercise 2.2.1 Basics

The template class `SmallObjectHeap` is to be parameterized by a number corresponding to the size of the element to be allocated. The type should be `std::size_t`.

Since all our objects are to be stored in *pages* of type `Page` we will need a container to handle them all. Add a suitable one for our `Pages`.

Furthermore add a suitable default constructor as well as suitable destructor. In particular, at this point, what should the destructor do?

Exercise 2.2.2 Meyers-Singleton

The class must be a singleton, use Meyers singleton[2] approach, however there are certain things one has to do concerning certain member methods for a singleton class. What are they and do remember to do it.

Exercise 2.2.3 Implement the `allocate()` method

The task of the `allocate()` method is to search through the individual `Page` objects until a free spot is found. If one is not found, a new `Page` is allocated and inserted into our container.

Exercise 2.2.4 Implement the `deallocate()` method

This method determines from which `Page` object the element in question was allocated on and frees it.

[2] Google will help, if you are uncertain as to what *Meyers singleton* means. If you remember go back and inspect the singleton you used in I3ISU.

Exercise 2.3 Small Object Allocator

In this exercise part, we are going to create or rather complete the allocator found in the file `MyAllocator/SmallObjectAllocator.hpp`.

Exercise 2.3.1 Construction ande Destruction

Currently nothing happens in the methods `construct()` and `destroy()`, however the rest works as expected using the global `new()` and `delete()` operations. This means that after you have added the missing implementation for the two mentioned methods, you will be able to try the allocator.

Exercise 2.3.2 Using our new small object heap class

In order to use the new *class SmallObjectHeap*, alterations in methods `allocate()` and `deallocate()` are necessary. Reimplement these using the functionality found in *class SmallObjectHeap*.

Do note that the parameter `n` in the method `allocate()` only may be 1.

AND do verify that it works...

Exercise 2.4 Optimization

As it is, our implementation works albeit not necessarily very fast. We find that inspecting the operations that are performed in methods `allocate()` and `deallocate()` on single `Page` objects are rather fast, however finding the correct `Page` object takes a long time.

The task in this exercise is therefore to alter `allocate()` and `deallocate()` such that they cache the last used `Page` object in both of these methods.

Exercise 2.5 Reflection

From your knowledge of embedded systems, how do you believe that this implementation should be altered to function well in an embedded context.

Discuss the chosen points.