Boost.Pool

Stephen Cleary

Table of Contents

Boost Pool Library

Overview

How to Use This Documentation

This documentation makes use of the following naming and formatting conventions.

- Code is in `fixed width font` and is syntax-highlighted in color.
- Replaceable text that you will need to supply is in *`italics`*.
- Free functions are rendered in the `code font` followed by `()`, as in `free_function()`.
- If a name refers to a class template, it is specified like this: `class_template<>`; that is, it is in code font and its name is followed by <> to indicate that it is a class template.
- If a name refers to a function-like macro, it is specified like this: `MACRO()`; that is, it is uppercase in code font and its name is followed by `()` to indicate that it is a function-like macro. Object-like macros appear without the trailing `()`.
- Names that refer to *concepts* in the generic programming sense are specified in CamelCase.

Note

In addition, notes such as this one specify non-essential information that provides additional background or rationale.

Finally, you can mentally add the following to any code fragments in this document:

```
// Include all of Pool files
#include <boost/pool.hpp>
```

Introduction

What is Pool?

Pool allocation is a memory allocation scheme that is very fast, but limited in its usage. For more information on pool allocation (also called *simple segregated storage*, see the concepts document.

Why should I use Pool?

Using Pools gives you more control over how memory is used in your program. For example, you could have a situation where you want to allocate a bunch of small objects at one point, and then reach a point in your program where none of them are needed any more. Using pool interfaces, you can choose to run their destructors or just drop them off into oblivion; the pool interface will guarantee that there are no system memory leaks.

When should I use Pool?

Pools are generally used when there is a lot of allocation and deallocation of small objects. Another common usage is the situation above, where many objects may be dropped out of memory.

In general, use Pools when you need a more efficient way to do unusual memory control.

How do I use Pool?

See the pool interfaces document, which covers the different Pool interfaces supplied by this library.

Library Structure and Dependencies

Forward declarations of all the exposed symbols for this library are in the header <boost/pool/poolfwd.hpp>.

The library may use macros, which will be prefixed with `BOOST_POOL_`. The exception to this rule are the include file guards, which (for file `xxx.hpp`) is `BOOST_xxx_HPP`.

All exposed symbols defined by the library will be in namespace boost. All symbols used only by the implementation will be in namespace boost::details::pool.

Every header used only by the implementation is in the subdirectory `/detail/`.

Any header in the library may include any other header in the library or any system-supplied header at its discretion.

Installation

The Boost Pool library is a header-only library. That means there is no .lib, .dll, or .so to build; just add the Boost directory to your compiler's include file path, and you should be good to go!

Building the Test Programs

The subdirectory *build* contains subdirectories for several different platforms. These subdirectories contain all necessary work-around code for that platform, as well as makefiles or IDE project files as appropriate.

Read the `readme.txt` in the proper subdirectory, if it exists.

The standard makefile targets are *all*, *clean* (which deletes any intermediate files), and *veryclean* (which deletes any intermediate files and executables). All intermediate and executable files are built in the same directory as the makefile/project file. If there is a project file supplied instead of a makefile, *clean* and *veryclean* shell scripts/batch files will be provided.

Documentation Map

Overview of Pooling

Pool Concepts - Basic ideas behind pooling

Dynamic memory allocation has been a fundamental part of most computer systems since roughly 1960... 1

Everyone uses dynamic memory allocation. If you have ever called malloc or new, then you have used dynamic memory allocation. Most programmers have a tendency to treat the heap as a "magic bag": we ask it for memory, and it magically creates some for us. Sometimes we run into problems because the heap is not magic.

The heap is limited. Even on large systems (i.e., not embedded) with huge amounts of virtual memory available, there is a limit. Everyone is aware of the physical limit, but there is a more subtle, 'virtual' limit, that limit at which your program (or the entire system) slows down due to the use of virtual memory. This virtual limit is much closer to your program than the physical limit, especially if you are running on a multitasking system. Therefore, when running on a large system, it is considered *nice* to make your program use as few resources as necessary, and release them as soon as possible. When using an embedded system, programmers usually have no memory to waste.

The heap is complicated. It has to satisfy any type of memory request, for any size, and do it fast. The common approaches to memory management have to do with splitting the memory up into portions, and keeping them ordered by size in some sort of a tree or list structure. Add in other factors, such as locality and estimating lifetime, and heaps quickly become very complicated. So complicated, in fact, that there is no known *perfect* answer to the problem of how to do dynamic memory allocation. The diagrams below illustrate how most common memory managers work: for each chunk of memory, it uses part of that memory to maintain its internal tree or list structure. Even when a chunk is malloc'ed out to a program, the memory manager must *save* some information in it - usually just its size. Then, when the block is free'd, the memory manager can easily tell how large it is.

Table 1.
****
Memory block, not allocated
****

Memory not belonging to process
Memory used internally by memory allocator algorithm (usually 8-12 bytes)
Unused memory
Memory not belonging to process

Table 2.
****
Memory block, allocated (used by program)
****

Memory not belonging to process
Memory used internally by memory allocator algorithm (usually 4 bytes)
Memory usable by program
Memory not belonging to process

Because of the complication of dynamic memory allocation, it is often inefficient in terms of time and/or space. Most memory allocation algorithms store some form of information with each memory block, either the block size or some relational information, such as its position in the internal tree or list structure. It is common for such *header fields* to take up one machine word in a block that is being used by the program. The obvious problem, then, is when small objects are dynamically allocated. For example, if ints were dynamically allocated, then automatically the algorithm will reserve space for the header fields as well, and we end up with a 50% waste of memory. Of course, this is a worst-case scenario. However, more modern programs are making use of small objects on the heap; and that is making this problem more and more apparent. Wilson et. al. state that an average-case memory overhead is about ten to twenty percent2. This memory overhead will grow higher as more programs use more smaller objects. It is this memory overhead that brings programs closer to the virtual limit.

In larger systems, the memory overhead is not as big of a problem (compared to the amount of time it would take to work around it), and thus is often ignored. However, there are situations where many allocations and/or deallocations of smaller objects are taking place as part of a time-critical algorithm, and in these situations, the system-supplied memory allocator is often too slow.

Simple segregated storage addresses both of these issues. Almost all memory overhead is done away with, and all allocations can take place in a small amount of (amortized) constant time. However, this is done at the loss of generality; simple segregated storage only can allocate memory chunks of a single size.

Simple Segregated Storage

Simple Segregated Storage is the basic idea behind the Boost Pool library. Simple Segregated Storage is the simplest, and probably the fastest, memory allocation/deallocation algorithm. It begins by partitioning a memory block into fixed-size chunks. Where the block comes from is not important until implementation time. A Pool is some object that uses Simple Segregated Storage in this fashion. To illustrate:

Table 3. Memory block, split into chunks

Memory not belonging to process
Chunk 0
Chunk 1
Chunk 2
Chunk 3
Memory not belonging to process

Each of the chunks in any given block are always the same size. This is the fundamental restriction of Simple Segregated Storage: you cannot ask for chunks of different sizes. For example, you cannot ask a Pool of integers for a character, or a Pool of characters for an integer (assuming that characters and integers are different sizes).

Simple Segregated Storage works by interleaving a free list within the unused chunks. For example:

Table 4. Memory block, with no chunks allocated

Memory not belonging to process
Chunk 0; points to Chunk 1
Chunk 1; points to Chunk 2
Chunk 2; points to Chunk 3
Chunk 3; end-of-list
Memory not belonging to process

Table 5. Memory block, with two chunks allocated

Memory not belonging to process
Chunk 0; points to Chunk 2
Chunk 1 (in use by process)
Chunk 2; end-of-list
Chunk 3 (in use by process)
Memory not belonging to process

By interleaving the free list inside the chunks, each Simple Segregated Storage only has the overhead of a single pointer (the pointer to the first element in the list). It has no memory overhead for chunks that are in use by the process.

Simple Segregated Storage is also extremely fast. In the simplest case, memory allocation is merely removing the first chunk from the free list, a O(1) operation. In the case where the free list is empty, another block may have to be acquired and partitioned, which would result in an amortized O(1) time. Memory deallocation may be as simple as adding that chunk to the front of the free list, a O(1) operation. However, more complicated uses of Simple Segregated Storage may require a sorted free list, which makes deallocation O(N).

Simple Segregated Storage gives faster execution and less memory overhead than a system-supplied allocator, but at the loss of generality. A good place to use a Pool is in situations where many (noncontiguous) small objects may be allocated on the heap, or if allocation and deallocation of the same-sized objects happens repeatedly.

References

1. Doug Lea, A Memory Allocator. See http://gee.cs.oswego.edu/dl/html/malloc.html

2. Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles, *Dynamic Storage Allocation: A Survey and Critical Review* in International Workshop on Memory Management, September 1995, pg. 28, 36. See ftp://ftp.cs.utexas.edu/pub/garbage/allocsrv.ps

Other Implementations

Pool allocators are found in many programming languages, and in many variations. The beginnings of many implementations may be found in common programming literature; some of these are given below. Note that none of these are complete implementations of a Pool; most of these leave some aspects of a Pool as a user exercise. However, in each case, even though some aspects are missing, these examples use the same underlying concept of a Simple Segregated Storage described in this document.

1. *The C++ Programming Language*, 3rd ed., by Bjarne Stroustrup, Section 19.4.2. Missing aspects:
   - Not portable
   - Cannot handle allocations of arbitrary numbers of objects (this was left as an exercise)
   - Not thread-safe
   - Suffers from the static initialization problem
2. *MicroC/OS-II: The Real-Time Kernel*, by Jean J. Labrosse, Chapter 7 and Appendix B.04.
   - An example of the Simple Segregated Storage scheme at work in the internals of an actual OS.
   - Missing aspects:
     - Not portable (though this is OK, since it's part of its own OS)
     - Cannot handle allocations of arbitrary numbers of blocks (which is also OK, since this feature is not needed)
     - Requires non-intuitive user code to create and destroy the Pool
3. *Efficient C++: Performance Programming Techniques*, by Dov Bulka and David Mayhew, Chapters 6 and 7.
   - This is a good example of iteratively developing a Pool solution;
   - however, their premise (that the system-supplied allocation mechanism is hopelessly inefficient) is flawed on every system I've tested on.
   - Run their timings on your system before you accept their conclusions.
   - Missing aspect: Requires non-intuitive user code to create and destroy the Pool
4. *Advanced C++: Programming Styles and Idioms*, by James O. Coplien, Section 3.6.
   - Has examples of both static and dynamic pooling, but missing aspects:
   - Not thread-safe
   - The static pooling example is not portable

Guaranteeing Alignment - How we guarantee alignment portably.

Terminology

Review the *concepts* if you are not already familiar with it. Remember that block is a contiguous section of memory, which is partitioned or segregated into fixed-size chunks. These chunks are what are allocated and deallocated by the user.

Overview

Each Pool has a single free list that can extend over a number of memory blocks. Thus, Pool also has a linked list of allocated memory blocks. Each memory block, by default, is allocated using new[], and all memory blocks are freed on destruction. It is the use of new[] that allows us to guarantee alignment.

Proof of Concept: Guaranteeing Alignment

Each block of memory is allocated as a POD type (specifically, an array of characters) through operator new[]. Let POD_size be the number of characters allocated.

Predicate 1: Arrays may not have padding

This follows from the following quote:

[5.3.3/2] (Expressions::Unary expressions::Sizeof) ... *When applied to an array, the result is the total number of bytes in the array. This implies that the size of an array of n elements is n times the size of an element.*

Therefore, arrays cannot contain padding, though the elements within the arrays may contain padding.

Predicate 2: Any block of memory allocated as an array of characters through operator new[

(hereafter referred to as the block) is properly aligned for any object of that size or smaller]

This follows from:

- [3.7.3.1/2] (Basic concepts::Storage duration::Dynamic storage duration::Allocation functions) ... *The pointer returned shall be suitably aligned so that it can be converted to a pointer of any complete object type and then used to access the object or array in the storage allocated* ...
- [5.3.4/10] (Expressions::Unary expressions::New) “... For arrays of char and unsigned char, the difference between the result of the new-expression and the address returned by the allocation function shall be an integral multiple of the most stringent alignment requirement (3.9) of any object type whose size is no greater than the size of the array being created. [Note: Because allocation functions are assumed to return pointers to storage that is appropriately aligned for objects of any type, this constraint on array allocation overhead permits the common idiom of allocating character arrays into which objects of other types will later be placed. "”

Consider: imaginary object type Element of a size which is a multiple of some actual object size; assume sizeof(Element) > POD_size

Note that an object of that size can exist. One object of that size is an array of the "actual" objects.

Note that the block is properly aligned for an Element. This directly follows from Predicate 2.

Corollary 1: The block is properly aligned for an array of Elements

This follows from Predicates 1 and 2, and the following quote:

[3.9/9] (Basic concepts::Types) “An object type is a (possibly cv-qualified) type that is not a function type, not a reference type, and not a void type."”

(Specifically, array types are object types.)

Corollary 2: For any pointer p and integer i, if p is properly aligned for the type it points to, then p + i (when well-defined) is properly aligned for that type; in other words, if an array is properly aligned, then each element in that array is properly aligned

There are no quotes from the Standard to directly support this argument, but it fits the common conception of the meaning of “alignment"”.

Note that the conditions for p + i being well-defined are outlined in [5.7/5]. We do not quote that here, but only make note that it is well-defined if p and p + i both point into or one past the same array.

Let: sizeof(Element) be the least common multiple of sizes of several actual objects (T1, T2, T3, ...)

Let: block be a pointer to the memory block, pe be (Element *) block, and pn be (Tn *) block

Corollary 3: For each integer i, such that pe + i is well-defined, then for each n, there exists some integer jn such that pn + jn is well-defined and refers to the same memory address as pe + i

This follows naturally, since the memory block is an array of Elements, and for each n, sizeof(Element) % sizeof(Tn) == 0; thus, the boundary of each element in the array of Elements is also a boundary of each element in each array of Tn.

Theorem: For each integer i, such that pe + i is well-defined, that address (pe + i) is properly aligned for each type Tn

Since pe + i is well-defined, then by Corollary 3, pn + jn is well-defined. It is properly aligned from Predicate 2 and Corollaries 1 and 2.

Use of the Theorem

The proof above covers alignment requirements for cutting chunks out of a block. The implementation uses actual object sizes of:

- The requested object size (requested_size); this is the size of chunks requested by the user
- void * (pointer to void); this is because we interleave our free list through the chunks
- size_type; this is because we store the size of the next block within each memory block

Each block also contains a pointer to the next block; but that is stored as a pointer to void and cast when necessary, to simplify alignment requirements to the three types above.

Therefore, alloc_size is defined to be the lcm of the sizes of the three types above.

A Look at the Memory Block

Each memory block consists of three main sections . The first section is the part that chunks are cut out of, and contains the interleaved free list. The second section is the pointer to the next block, and the third section is the size of the next block.

Each of these sections may contain padding as necessary to guarantee alignment for each of the next sections. The size of the first section is `number_of_chunks * lcm(requested_size, sizeof(void *), sizeof(size_type));` the size of the second section is `lcm(sizeof(void *), sizeof(size_type);` and the size of the third section is `sizeof(size_type)`.

Here's an example memory block, where `requested_size == sizeof(void *) == sizeof(size_type) == 4`:

Table 6. Memory block containing 4 chunks, showing overlying array structures; FLP = Interleaved Free List Pointer

Sections	size_type alignment	void * alignment	requested_size alignment
Memory not belonging to process			
Chunks section (16 bytes)	(4 bytes)	FLP for Chunk 1 (4 bytes)	Chunk 1 (4 bytes)
(4 bytes)	FLP for Chunk 2 (4 bytes)	Chunk 2 (4 bytes)	
(4 bytes)	FLP for Chunk 3 (4 bytes)	Chunk 3 (4 bytes)	
(4 bytes)	FLP for Chunk 4 (4 bytes)	Chunk 4 (4 bytes)	
Pointer to next Block (4 bytes)	(4 bytes)	Pointer to next Block (4 bytes)	
Size of next Block (4 bytes)	Size of next Block (4 bytes)		
Memory not belonging to process			

To show a visual example of possible padding, here's an example memory block where requested_size `= 8 and sizeof(void *)` = sizeof(size_type) == 4:

Table 7. Memory block containing 4 chunks, showing overlying array structures; FLP = Interleaved Free List Pointer

Sections	size_type alignment	void * alignment	requested_size alignment
Memory not belonging to process			
Chunks section (32 bytes)	(4 bytes)	FLP for Chunk 1 (4 bytes)	Chunk 1 (8 bytes)
(4 bytes)	(4 bytes)		
(4 bytes)	FLP for Chunk 2 (4 bytes)	Chunk 2 (8 bytes)	
(4 bytes)	(4 bytes)		
(4 bytes)	FLP for Chunk 3 (4 bytes)	Chunk 3 (8 bytes)	
(4 bytes)	(4 bytes)		
(4 bytes)	FLP for Chunk 4 (4 bytes)	Chunk 4 (8 bytes)	
(4 bytes)	(4 bytes)		
Pointer to next Block (4 bytes)	(4 bytes)	Pointer to next Block (4 bytes)	
Size of next Block (4 bytes)	Size of next Block (4 bytes)		
Memory not belonging to process			

Finally, here is a convoluted example where the requested_size is 7, `sizeof(void *) == 3, and sizeof(size_type) == 5`, showing how the least common multiple guarantees alignment requirements even in the oddest of circumstances:

Table 8. Memory block containing 2 chunks, showing overlying array structures

Sections	size_type alignment	void * alignment	requested_size alignment
Memory not belonging to process			
Chunks section (210 bytes)	(5 bytes)	Interleaved free list pointer for Chunk 1 (15 bytes; 3 used)	Chunk 1 (105 bytes; 7 used)
(5 bytes)			
(5 bytes)			
(5 bytes)	(15 bytes)		
(5 bytes)			
(5 bytes)			
(5 bytes)	(15 bytes)		
(5 bytes)			
(5 bytes)			
(5 bytes)	(15 bytes)		
(5 bytes)			
(5 bytes)			
(5 bytes)	(15 bytes)		
(5 bytes)			
(5 bytes)			
(5 bytes)	(15 bytes)		
(5 bytes)			
(5 bytes)			
(5 bytes)	(15 bytes)		
(5 bytes)			
(5 bytes)			
(5 bytes)	Interleaved free list pointer for Chunk 2 (15 bytes; 3 used)	Chunk 2 (105 bytes; 7 used)	
(5 bytes)			
(5 bytes)			

Sections	size_type alignment	void * alignment	requested_size alignment
(5 bytes)	(15 bytes)		
(5 bytes)			
(5 bytes)			
(5 bytes)	(15 bytes)		
(5 bytes)			
(5 bytes)			
(5 bytes)	(15 bytes)		
(5 bytes)			
(5 bytes)			
(5 bytes)	(15 bytes)		
(5 bytes)			
(5 bytes)			
(5 bytes)	(15 bytes)		
(5 bytes)			
(5 bytes)			
(5 bytes)	(15 bytes)		
(5 bytes)			
(5 bytes)			
Pointer to next Block (15 bytes; 3 used)	(5 bytes)	Pointer to next Block (15 bytes; 3 used)	
(5 bytes)			
(5 bytes)			
Size of next Block (5 bytes; 5 used)	Size of next Block (5 bytes; 5 used)		
Memory not belonging to process			

How Contiguous Chunks are Handled

The theorem above guarantees all alignment requirements for allocating chunks and also implementation details such as the interleaved free list. However, it does so by adding padding when necessary; therefore, we have to treat allocations of contiguous chunks in a different way.

Using array arguments similar to the above, we can translate any request for contiguous memory for n objects of requested_size into a request for m contiguous chunks. m is simply `ceil(n * requested_size / alloc_size)`, where `alloc_size` is the actual size of the chunks.

To illustrate:

Here's an example memory block, where `requested_size == 1` and `sizeof(void *) == sizeof(size_type) == 4`:

Table 9. Memory block containing 4 chunks; requested_size is 1

Sections	size_type alignment	void * alignment	requested_size alignment
Memory not belonging to process			
Chunks section (16 bytes)	(4 bytes)	FLP to Chunk 2 (4 bytes)	Chunk 1 (4 bytes)
(4 bytes)	FLP to Chunk 3 (4 bytes)	Chunk 2 (4 bytes)	
(4 bytes)	FLP to Chunk 4 (4 bytes)	Chunk 3 (4 bytes)	
(4 bytes)	FLP to end-of-list (4 bytes)	Chunk 4 (4 bytes)	
Pointer to next Block (4 bytes)	(4 bytes)	Ptr to end-of-list (4 bytes)	
Size of next Block (4 bytes)	0 (4 bytes)		
Memory not belonging to process			

Table 10. After user requests 7 contiguous elements of requested_size

Sections	size_type alignment	void * alignment	`requested_size` alignment
Memory not belonging to process			
Chunks section (16 bytes)	(4 bytes)	(4 bytes)	4 bytes in use by program
(4 bytes)	(4 bytes)	3 bytes in use by program (1 byte unused)	
(4 bytes)	FLP to Chunk 4 (4 bytes)	Chunk 3 (4 bytes)	
(4 bytes)	FLP to end-of-list (4 bytes)	Chunk 4 (4 bytes)	
Pointer to next Block (4 bytes)	(4 bytes)	Ptr to end-of-list (4 bytes)	
Size of next Block (4 bytes)	0 (4 bytes)		
Memory not belonging to process			

Then, when the user deallocates the contiguous memory, we can split it up into chunks again.

Note that the implementation provided for allocating contiguous chunks uses a linear instead of quadratic algorithm. This means that it may not find contiguous free chunks if the free list is not ordered. Thus, it is recommended to always use an ordered free list when dealing with contiguous allocation of chunks. (In the example above, if Chunk 1 pointed to Chunk 3 pointed to Chunk 2

pointed to Chunk 4, instead of being in order, the contiguous allocation algorithm would have failed to find any of the contiguous chunks).

- interfaces.html - What interfaces are provided and when to use each one.
- Pool Exposed Interfaces
- interfaces/simple_segregated_storage.html - Not for the faint of heart; embedded programmers only.
- interfaces/pool.html - The basic pool interface.
- interfaces/singleton_pool.html - The basic pool interface as a thread-safe singleton.
- interfaces/object_pool.html - A type-oriented (instead of size-oriented) pool interface.
- interfaces/pool_alloc.html - A Standard Allocator pool interface based on singleton_pool.
- interfaces/user_allocator.html - OK, not a pool interface, but it describes how the user can control how Pools allocate system memory.
- Pool Implementation Details and Extensions
- Interface Implementations and Extensions
- implementation/simple_segregated_storage.html
- implementation/pool.html
- implementation/singleton_pool.html
- implementation/object_pool.html
- implementation/pool_alloc.html
- Components Used Only by the Implementation
- implementation/ct_gcd_lcm.html - Compile-time GCD and LCM.
- implementation/for.html - Description of an m4 component.
- implementation/gcd_lcm.html - Run-time GCD and LCM.
- implementation/guard.html - Auto lock/unlock for mutex.
- implementation/mutex.html - Platform-dependent mutex type.
- implementation/pool_construct.html - The system for supporting more constructor arguments in object_pool.
- implementation/singleton.html - Singleton that avoids static initialization problem.

Appendices

Appendix A: History

Version 1.0.0, January 1, 2000 *First release*

Version 2.0.0, January 11, 2011 *Documentation and testing revision*

Features:

- Converted documentation using Quickbook, Doxygen, for html and pdf, based on Stephen Cleary's html version, Revised 05 December, 2006.

Appendix B: Rationale

TODO.

Appendix C: Implementation Notes

TODO.

Appendix D: FAQ

Why should I use Pool?

Using Pools gives you more control over how memory is used in your program. For example, you could have a situation where you want to allocate a bunch of small objects at one point, and then reach a point in your program where none of them are needed any more. Using pool interfaces, you can choose to run their destructors or just drop them off into oblivion; the pool interface will guarantee that there are no system memory leaks.

When should I use Pool?

Pools are generally used when there is a lot of allocation and deallocation of small objects. Another common usage is the situation above, where many objects may be dropped out of memory.

In general, use Pools when you need a more efficient way to do unusual memory control.

Appendix E: Acknowledgements

Many, many thanks to the Boost peers, notably Jeff Garland, Beman Dawes, Ed Brey, Gary Powell, Peter Dimov, and Jens Maurer for providing helpful suggestions!

Appendix F: Tests

See folder `boost/libs/pool/test/`.

Appendix G: Tickets

Report and view bugs and features by adding a ticket at Boost.Trac.

Appendix H: Future plans

For later releases

Another pool interface will be written: a base class for per-class pool allocation.