Boost.Pool

Stephen Cleary

Table of Contents

Boost Pool Library

Overview

Documentation Naming and Formatting Conventions

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 made inscope by `#include <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.

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.

Memory block, allocated (not by process)

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

Memory block, allocated (in use by process)

Table 2.

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

Dynamic memory allocation is often inefficient

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).

Boost Pool Interfaces - What interfaces are provided and when to use each one.

Introduction

There are several interfaces provided which allow users great flexibility in how they want to use Pools. Review the concepts document to get the basic understanding of how Pools work.

Terminology and Tradeoffs

Object Usage vs. Singleton Usage

Object Usage is the method where each Pool is an object that may be created and destroyed. Destroying a Pool implicitly frees all chunks that have been allocated from it.

Singleton Usage is the method where each Pool is an object with static duration; that is, it will not be destroyed until program exit. Pool objects with Singleton Usage may be shared; thus, Singleton Usage implies thread-safety as well. System memory allocated by Pool objects with Singleton Usage may be freed through release_memory or purge_memory.

Out-of-Memory Conditions: Exceptions vs. Null Return

Some Pool interfaces throw exceptions when out-of-memory; others will return 0. In general, unless mandated by the Standard, Pool interfaces will always prefer to return 0 instead of throw an exception.

Pool Interfaces

pool

The pool interface is a simple Object Usage interface with Null Return.

Example:

```
void func()
{
  boost::pool<> p(sizeof(int));
  for (int i = 0; i < 10000; ++i)
  {
    int * const t = p.malloc();
    ... / Do something with t; don't take the time to free() it
  }
} / on function exit, p is destroyed, and all malloc()'ed ints are implicitly freed
```

object_pool

The object_pool interface is an Object Usage interface with Null Return, but is aware of the type of the object for which it is allocating chunks. On destruction, any chunks that have been allocated from that object_pool will have their destructors called.

Example:

```
struct X { ... }; / has destructor with side-effects

void func()
{
  boost::object_pool<X> p;
  for (int i = 0; i < 10000; ++i)
  {
    X * const t = p.malloc();
    ... / Do something with t; don't take the time to free() it
  }
} // on function exit, p is destroyed, and all destructors for the X objects are called
```

singleton_pool

The singleton_pool interface is a Singleton Usage interface with Null Return. It's just the same as the pool interface but with Singleton Usage instead.

Example:

```
struct MyPoolTag { };

typedef boost::singleton_pool<MyPoolTag, sizeof(int)> my_pool;
void func()
{
  for (int i = 0; i < 10000; ++i)
  {
    int * const t = my_pool::malloc();
    ... / Do something with t; don't take the time to free() it
  }
  / Explicitly free all malloc()'ed int's
  my_pool::purge_memory();
}
```

pool_alloc

The pool_alloc interface is a Singleton Usage interface with Exceptions. It is built on the singleton_pool interface, and provides a Standard Allocator-compliant class (for use in containers, etc.).

Example:

```
void func()
{
  std::vector<int, boost::pool_allocator<int> > v;
  for (int i = 0; i < 10000; ++i)
    v.push_back(13);
} / Exiting the function does NOT free the system memory allocated by the pool allocator
  / You must call
  /  boost::singleton_pool<boost::pool_allocator_tag, sizeof(int)>::release_memory()
  / in order to force that
```

Future Directions

Another pool interface will be written: a base class for per-class pool allocation. This "pool_base" interface will be Singleton Usage with Exceptions, and built on the singleton_pool interface.

Simple_segregated_storage - Not for the faint of heart; embedded programmers only!

Introduction

simple_segregated_storage.hpp provides a template class simple_segregated_storage that controls access to a free list of memory chunks. Please note that this is a very simple class, with preconditions on almost all its functions. It is intended to be the fastest and smallest possible quick memory allocator for example, something to use in embedded systems. This class delegates many difficult preconditions to the user (especially alignment issues). For more general usage, see the other pool interfaces.

Synopsis

```
template <typename SizeType = std::size_t>
class simple_segregated_storage
{
  private:
    simple_segregated_storage(const simple_segregated_storage &);
    void operator=(const simple_segregated_storage &);

  public:
    typedef SizeType size_type;

    simple_segregated_storage();
    ~simple_segregated_storage();

    static void * segregate(void * block,
        size_type nsz, size_type npartition_sz,
        void * end = 0);
    void add_block(void * block,
        size_type nsz, size_type npartition_sz);
    void add_ordered_block(void * block,
        size_type nsz, size_type npartition_sz);

    bool empty() const;

    void * malloc();
    void free(void * chunk);
    void ordered_free(void * chunk);
    void * malloc_n(size_type n, size_type partition_sz);
    void free_n(void * chunks, size_type n,
        size_type partition_sz);
    void ordered_free_n(void * chunks, size_type n,
        size_type partition_sz);
};
```

Semantics

An object of type `simple_segregated_storage<SizeType>` is empty if its free list is empty. If it is not empty, then it is ordered if its free list is ordered. A free list is ordered if repeated calls to `malloc()` will result in a constantly-increasing sequence of values, as determined by `std::less<void *>`. A member function is order-preserving if the free list maintains its order orientation (that is, an ordered free list is still ordered after the member function call).

Table 11. Symbol Table

Symbol	Meaning
Store	simple_segregated_storage<SizeType>
t	value of type Store
u	value of type const Store
block, chunk, end	values of type void *
partition_sz, sz, n	values of type Store::size_type

Table 12. Template Parameters

Parameter	Default	Requirements
SizeType	std::size_t	An unsigned integral type

Table 13. Typedefs [[Symbol] [Type]]

size_type	SizeType

Table 14. Constructors, Destructors, and State

Expression	Return Type	Post-Condition	Notes
Store()	not used	empty()	Constructs a new Store
(&t)->~Store()	not used		Destructs the Store
u.empty()	bool		Returns true if u is empty. Order-preserving.

Table 15. Segregation

Expression	Return Type	Pre-Condition	Post-Condition	Semantic Equivalence	Notes
Store::segregate(block, sz, partition_sz, end)	void *	partition_sz >= sizeof(void *) partition_sz = sizeof(void *) * i, for some integer i sz >= partition_sz block is properly aligned for an array of objects of size partition_sz block is properly aligned for an array of void *			Interleaves a free list through the memory block specified by block of size sz bytes, partitioning it into as many partition_sz-sized chunks as possible. The last chunk is set to point to end, and a pointer to the first chunck is returned (this is always equal to block). This interleaved free list is ordered. O(sz).
Store::segregate(block, sz, partition_sz)	void *	Same as above		Store::segregate(block, sz, partition_sz, 0)	
t.add_block(block, sz, partition_sz)	void	Same as above	!t.empty()		Segregates the memory block specified by block of size sz bytes into partition_sz-sized chunks, and adds that free list to its own. If t was empty before this call, then it is ordered after this call. O(sz).
t.add_ordered_block(block, sz, partition_sz)	void	Same as above	!t.empty()		Segregates the memory block specified by block of size sz bytes into partition_sz-sized chunks, and merges that free list into its own. Order-preserving. O(sz).

Table 16. Allocation and Deallocation

Expression	Return Type	Pre-Condition	Post-Condition	Semantic Equivalence	Notes
t.malloc()	void *	!t.empty()			Takes the first available chunk from the free list and returns it. Order-preserving. O(1).
t.free(chunk)	void	chunk was previously returned from a call to t.malloc()	!t.empty()		Places chunk back on the free list. Note that chunk may not be 0. O(1).
t.ordered_free(chunk)	void	Same as above	!t.empty()		Places chunk back on the free list. Note that chunk may not be 0. Order-preserving. O(N) with respect to the size of the free list.
t.malloc_n(n, partition_sz)	void *				Attempts to find a contiguous sequence of n partition_sz-sized chunks. If found, removes them all from the free list and returns a pointer to the first. If not found, returns 0. It is strongly recommended (but not required) that the free list be ordered, as this algorithm will fail to find a contiguous sequence unless it is contiguous in the free list as well. Order-preserving. O(N) with respect to the size of the free list.
t.free_n(chunk, n, partition_sz)	void	chunk was previously returned from a call to t.malloc_n(n, partition_sz)	!t.empty()	t.add_block(chunk, n * partition_sz, partition_sz)	Assumes that chunk actually refers to a block of chunks spanning n * partition_sz bytes; segregates and adds in that block. Note that chunk may not be 0. O(n).
t.ordered_free_n(chunk, n, partition_sz)	void	same as above	same as above	t.add_ordered_block(chunk, n * partition_sz, partition_sz)	Same as above, except it merges in the free list. Order-preserving. O(N + n) where N is the size of the free list.

Boost.Pool Reference

Header <boost/pool/detail/ct_gcd_lcm.hpp>

Compile-Time GCD and LCM.

Provides two compile-time algorithms: greatest common divisor and least common multiple.

Selected Quotation from the C++ Standard

5.19/1: Expressions: Constant Expressions: ". . . An integral constant expression can involve only literals (2.13), enumerators, const variables or static data members of integral or enumeration types initialized with constant expressions (8.5), non-type template parameters of integral or enumeration types, and sizeof expressions. Floating literals (2.13.3) can appear only if they are cast to integral or enumeration types. Only type conversions to integral or enumeration types can be used. In particular, except in sizeof expressions, functions, class objects, pointers, or references shall not be used, and assignment, increment, decrement, function-call, or comma operators shall not be used."

Header <boost/pool/detail/gcd_lcm.hpp>

GCD and LCM two generic integer algorithms: greatest common divisor and least common multiple..

Header <boost/pool/detail/guard.hpp>

Extremely Light-Weight guard class.

Auto-lock/unlock-er detail/guard.hpp provides a type guard<Mutex> that allows scoped access to the Mutex's locking and unlocking operations. It is used to ensure that a Mutex is unlocked, even if an exception is thrown.

Header <boost/pool/detail/mutex.hpp>

Extremely Light-Weight wrapper classes for OS thread synchronization.

detail/mutex.hpp provides several mutex types that provide a consistent interface for OS-supplied mutex types. These are all thread-level mutexes; interprocess mutexes are not supported.

Configuration
This header file will try to guess what kind of system it is on. It will auto-configure itself for Win32 or POSIX+pthread systems. To stub out all mutex code, bypassing the auto-configuration, #define BOOST_NO_MT before any inclusion of this header. To prevent ODR violations, this should be defined in every translation unit in your project, including any library files.

Note: Each mutex is always either owned or unowned. If owned, then it is owned by a particular thread. To "lock" a mutex means to wait until the mutex is unowned, and then make it owned by the current thread. To "unlock" a mutex means to release ownership from the current thread (note that the current thread must own the mutex to release that ownership!). As a special case, the null_mutex never waits.

```
BOOST_MUTEX_HELPER_NONE
BOOST_MUTEX_HELPER_WIN32
BOOST_MUTEX_HELPER_PTHREAD
BOOST_NO_MT
BOOST_MUTEX_HELPER
```

Macro BOOST_MUTEX_HELPER_NONE

BOOST_MUTEX_HELPER_NONE

Synopsis

```
// In header: <boost/pool/detail/mutex.hpp>

BOOST_MUTEX_HELPER_NONE
```

Macro BOOST_MUTEX_HELPER_WIN32

BOOST_MUTEX_HELPER_WIN32

Synopsis

```
// In header: <boost/pool/detail/mutex.hpp>

BOOST_MUTEX_HELPER_WIN32
```

Macro BOOST_MUTEX_HELPER_PTHREAD

BOOST_MUTEX_HELPER_PTHREAD

Synopsis

```
// In header: <boost/pool/detail/mutex.hpp>

BOOST_MUTEX_HELPER_PTHREAD
```

Macro BOOST_NO_MT

BOOST_NO_MT

Synopsis

```
// In header: <boost/pool/detail/mutex.hpp>

BOOST_NO_MT
```

Macro BOOST_MUTEX_HELPER

BOOST_MUTEX_HELPER

Synopsis

```
// In header: <boost/pool/detail/mutex.hpp>

BOOST_MUTEX_HELPER
```

Header <boost/pool/detail/singleton.hpp>

Access to a Singleton of a class type. detail/singleton.hpp provides a way to access a Singleton of a class type. This is not a general Singleton solution! It is restricted in that the class type must have a default constructor.

Guarantees
The singleton instance is guaranteed to be constructed before main() begins, and destructed after main() ends. Furthermore, it is guaranteed to be constructed before the first call to singleton_default<T>::instance() is complete (even if called before main() begins). Thus, if there are not multiple threads running except within main(), and if all access to the singleton is restricted by mutexes, then this guarantee allows a thread-safe singleton.

The following helper classes are placeholders for a generic "singleton" class. The classes below support usage of singletons, including use in program startup/shutdown code, AS LONG AS there is only one thread running before main() begins, and only one thread running after main() exits.

This class is also limited in that it can only provide singleton usage for classes with default constructors.

The design of this class is somewhat twisted, but can be followed by the calling inheritance. Let us assume that there is some user code that calls "singleton_default<T>::instance()". The following (convoluted) sequence ensures that the same function will be called before main(): instance() contains a call to create_object.do_nothing() Thus, object_creator is implicitly instantiated, and create_object must exist. Since create_object is a static member, its constructor must be called before main(). The constructor contains a call to instance(), thus ensuring that instance() will be called before main(). The first time instance() is called (i.e., before main()) is the latest point in program execution where the object of type T can be created. Thus, any call to instance() will auto-magically result in a call to instance() before main(), unless already present. Furthermore, since the instance() function contains the object, instead of the singleton_default class containing a static instance of the object, that object is guaranteed to be constructed (at the latest) in the first call to instance(). This permits calls to instance() from static code, even if that code is called before the file-scope objects in this file have been initialized.

Header <boost/pool/object_pool.hpp>

provides a template type that can be used for fast and efficient memory allocation. It also provides automatic destruction of non-deallocated objects.
For information on other pool-based interfaces, see the other pool interfaces.

UserAllocator
Defines the allocator that the underlying Pool will use to allocate memory from the system. See User Allocators for details.

(&t)->~ObjectPool() Destructs the ObjectPool.
~ElementType() is called for each allocated ElementType that has not been deallocated. O(N).

Extensions to Public Interface

Whenever an object of type ObjectPool needs memory from the system, it will request it from its UserAllocator template parameter. The amount requested is determined using a doubling algorithm; that is, each time more system memory is allocated, the amount of system memory requested is doubled. Users may control the doubling algorithm by using the following extensions.

Additional constructor parameter

Users may pass an additional constructor parameter to ObjectPool. This parameter is of type size_type, and is the number of chunks to request from the system the first time that object needs to allocate system memory. The default is 32. This parameter may not be 0.

```
namespace boost {
  template<typename T, typename UserAllocator> class object_pool;
}
```

Class template object_pool

boost::object_pool

Synopsis

```
// In header: <boost/pool/object_pool.hpp>

template<typename T, typename UserAllocator>
class object_pool : protected boost::pool< UserAllocator > {
public:
  // types
  typedef T                                     element_type;     // ElementType.
  typedef UserAllocator                         user_allocator;   // User allocator.
  typedef pool< UserAllocator >::size_type      size_type;        // pool<UserAllocat↵
or>::size_type
  typedef pool< UserAllocator >::difference_type difference_type;  // pool<UserAllocator>::dif↵
ference_type

  // construct/copy/destruct
  object_pool(const size_type = 32, const size_type = 0);
  ~object_pool();

  // protected member functions
  pool< UserAllocator > & store();
  const pool< UserAllocator > & store() const;

  // protected static functions
  static void *& nextof(void *const);

  // public member functions
  element_type *malloc BOOST_PREVENT_MACRO_SUBSTITUTION();
  void free BOOST_PREVENT_MACRO_SUBSTITUTION(element_type *const);
  bool is_from(element_type *const) const;
  element_type * construct();
  void destroy(element_type *const);
  size_type get_next_size() const;
  void set_next_size(const size_type);
};
```

Description

`object_pool` public construct/copy/destruct

1. ```
   object_pool(const size_type next_size = 32, const size_type max_size = 0);
   ```

   Constructs a new (empty by default) ObjectPool.

   Parameters:
   - `max_size` maximum size of block.
   - `next_size` number of chunks to request from the system the next time that object needs to allocate system memory (default 32).

   Requires: next_size != 0.

2. ```
   ~object_pool();
   ```

`object_pool` protected member functions

1. ```
   pool< UserAllocator > & store();
   ```
   ```

2.
```
const pool< UserAllocator > & store() const;
```

`object_pool` protected static functions

1.
```
static void *& nextof(void *const ptr);
```

Returns: dereferenced ptr (for the sake of code readability :)

`object_pool` public member functions

1.
```
element_type *malloc BOOST_PREVENT_MACRO_SUBSTITUTION();
```

Allocates memory that can hold one object of type ElementType. If out of memory, returns 0. Amortized O(1).

2.
```
void free BOOST_PREVENT_MACRO_SUBSTITUTION(element_type *const chunk);
```

De-Allocates memory that holds a chunk of type ElementType. Note that p may not be 0.
Note that the destructor for p is not called. O(N).

3.
```
bool is_from(element_type *const chunk) const;
```

Returns: true if p was allocated from u or may be returned as the result of a future allocation from u. Returns false if p was allocated from some other pool or may be returned as the result of a future allocation from some other pool. Otherwise, the return value is meaningless. Note that this function may not be used to reliably test random pointer values!

4.
```
element_type * construct();
```

Constructs a new

5.
```
void destroy(element_type *const chunk);
```

destroy a chunk. (== p->~ElementType(); t.free(p);)
Requires: p must have been previously allocated from t.

6.
```
size_type get_next_size() const;
```

7.
```
void set_next_size(const size_type x);
```

Header <boost/pool/pool.hpp>

Fast memory allocator.

Fast memory allocator that guarantees proper alignment of all allocated chunks. Provides two UserAllocator classes and a template class pool, which extends and generalizes the framework provided by the simple segregated storage solution. For information on other pool-based interfaces, see the other pool interfaces.

```
namespace boost {
  struct default_user_allocator_new_delete;
  struct default_user_allocator_malloc_free;

  template<typename UserAllocator> class pool;
}
```

Struct default_user_allocator_new_delete

boost::default_user_allocator_new_delete

Synopsis

```
// In header: <boost/pool/pool.hpp>


struct default_user_allocator_new_delete {
  // types
  typedef std::size_t size_type;
  typedef std::ptrdiff_t difference_type;  // A signed integral type that can represent the dif↵
ference of any two pointers.

  // public static functions
  static char *malloc BOOST_PREVENT_MACRO_SUBSTITUTION(const size_type);
  static void free BOOST_PREVENT_MACRO_SUBSTITUTION(char *const);
};
```

Description

`default_user_allocator_new_delete` public types

1. typedef std::size_t size_type;

   An unsigned integral type that can represent the size of the largest object to be allocated.

`default_user_allocator_new_delete` public static functions

1. ```
   static char *malloc BOOST_PREVENT_MACRO_SUBSTITUTION(const size_type bytes);
   ```

   Attempts to allocate n bytes from the system. Returns 0 if out-of-memory

2. ```
   static void free BOOST_PREVENT_MACRO_SUBSTITUTION(char *const block);
   ```

   Attempts to de-allocate block.
   Requires: Block must have been previously returned from a call to UserAllocator::malloc.

Struct default_user_allocator_malloc_free

boost::default_user_allocator_malloc_free

Synopsis

```
// In header: <boost/pool/pool.hpp>


struct default_user_allocator_malloc_free {
  // types
  typedef std::size_t size_type;
  typedef std::ptrdiff_t difference_type;  // A signed integral type that can represent the dif↵
ference of any two pointers.

  // public static functions
  static char *malloc BOOST_PREVENT_MACRO_SUBSTITUTION(const size_type);
  static void free BOOST_PREVENT_MACRO_SUBSTITUTION(char *const);
};
```

Description

`default_user_allocator_malloc_free` public types

1. typedef std::size_t size_type;

   An unsigned integral type that can represent the size of the largest object to be allocated.

`default_user_allocator_malloc_free` public static functions

1.
```
static char *malloc BOOST_PREVENT_MACRO_SUBSTITUTION(const size_type bytes);
```

2.
```
static void free BOOST_PREVENT_MACRO_SUBSTITUTION(char *const block);
```

Class template pool

boost::pool

Synopsis

```
// In header: <boost/pool/pool.hpp>

template<typename UserAllocator>
class pool :
  protected boost::simple_segregated_storage< UserAllocator::size_type >
{
public:
  // types
  typedef UserAllocator                  user_allocator;   // User allocator.
  typedef UserAllocator::size_type       size_type;        // An unsigned integral type that ↵
can represent the size of the largest object to be allocated.
  typedef UserAllocator::difference_type difference_type;  // A signed integral type that can ↵
represent the difference of any two pointers.

  // member classes/structs/unions

  // A fast memory allocator that guarantees proper alignment of all allocated
  // chunks.

  class pool {
  };

  // construct/copy/destruct
  pool(const size_type, const size_type = 32, const size_type = 0);
  ~pool();

  // private member functions
   BOOST_STATIC_CONSTANT(unsigned,
                         min_alloc_size = (::boost::de↵
tails::pool::ct_lcm< sizeof(void *), sizeof(size_type)>::value));
  void * malloc_need_resize();
  void * ordered_malloc_need_resize();

  // protected member functions
  simple_segregated_storage< size_type > & store();
  const simple_segregated_storage< size_type > & store() const;
  details::PODptr< size_type > find_POD(void *const) const;
  size_type alloc_size() const;

  // protected static functions
  static bool is_from(void *const, char *const, const size_type);
  static void *& nextof(void *const);

  // public member functions
  bool release_memory();
  bool purge_memory();
  size_type get_next_size() const;
  void set_next_size(const size_type);
  size_type get_max_size() const;
  void set_max_size(const size_type);
  size_type get_requested_size() const;
  void *malloc BOOST_PREVENT_MACRO_SUBSTITUTION();
  void * ordered_malloc();
  void * ordered_malloc(size_type);
```

```
  void free BOOST_PREVENT_MACRO_SUBSTITUTION(void *const);
  void ordered_free(void *const);
  void free BOOST_PREVENT_MACRO_SUBSTITUTION(void *const, const size_type);
  void ordered_free(void *const, const size_type);
  bool is_from(void *const) const;
};
```

Description

`pool` public construct/copy/destruct

1.
```
pool(const size_type nrequested_size, const size_type nnext_size = 32,
     const size_type nmax_size = 0);
```

Constructs a new empty Pool that can be used to allocate chunks of size RequestedSize. is the number of chunks to request from the system the first time that object needs to allocate system memory. The default is 32. This parameter may not be 0.

Parameters:	`nmax_size`	is the maximum size of ?
	`nnext_size`	parameter is of type size_type,
	`nrequested_size`	Requested chunk size

2.
```
~pool();
```

Destructs the Pool, freeing its list of memory blocks.

`pool` private member functions

1.
```
 BOOST_STATIC_CONSTANT(unsigned,
                       min_alloc_size = (::boost::details::pool::ct_lcm< sizeof(void *),
sizeof(size_type)>::value));
```

2.
```
void * malloc_need_resize();
```

No memory in any of our storages; make a new storage, Allocates chunk in newly malloc aftert resize.

Returns:	0 if out-of-memory. Called if malloc/ordered_malloc needs to resize the free list.
Returns:	pointer to chunk.

3.
```
void * ordered_malloc_need_resize();
```

Called if malloc needs to resize the free list.

No memory in any of our storages; make a new storage,

Returns:	pointer to new chunk.

`pool` protected member functions

1.
```
simple_segregated_storage< size_type > & store();
```

Returns: pointer to store.

2.
```
const simple_segregated_storage< size_type > & store() const;
```

Returns: pointer to store.

3.
```
details::PODptr< size_type > find_POD(void *const chunk) const;
```

finds which POD in the list 'chunk' was allocated from.

find which PODptr storage memory that this chunk is from.
Returns: the PODptr that holds this chunk.

4.
```
size_type alloc_size() const;
```

Calculated size of the memory chunks that will be allocated by this Pool. For alignment reasons, this is defined to be lcm(requested_size, sizeof(void *), sizeof(size_type)).
Returns: allocated size.

`pool` protected static functions

1.
```
static bool is_from(void *const chunk, char *const i,
                    const size_type sizeof_i);
```

as the result of a future allocation. Returns false if chunk was allocated from some other pool, or may be returned as the result of a future allocation from some other pool. Otherwise, the return value is meaningless. Note that this function may not be used to reliably test random pointer values.
Parameters:
- `chunk` chunk to check if is from this pool.
- `i` memory chunk at i with element sizeof_i.
- `sizeof_i` element size (size of the chunk area of that block, not the total size of that block).

Returns: true if chunk was allocated or may be returned.

2.
```
static void *& nextof(void *const ptr);
```

Returns: Pointer dereferenced. (Provided and used for the sake of code readability :)

`pool` public member functions

1.
```
bool release_memory();
```

pool must be ordered. Frees every memory block that doesn't have any allocated chunks.
Returns: true if at least one memory block was freed.

2.
```
bool purge_memory();
```

pool must be ordered. Frees every memory block. This function invalidates any pointers previously returned by allocation functions of t.
Returns: true if at least one memory block was freed.

3.
```
size_type get_next_size() const;
```

Number of chunks to request from the system the next time that object needs to allocate system memory. This value should never be 0.
Returns: next_size;

4.
```
void set_next_size(const size_type nnext_size);
```

Set number of chunks to request from the system the next time that object needs to allocate system memory. This value should never be set to 0.

Returns: nnext_size.

5.
```
size_type get_max_size() const;
```

Returns: max_size.

6.
```
void set_max_size(const size_type nmax_size);
```

Set max_size.

7.
```
size_type get_requested_size() const;
```

Returns: the requested size passed into the constructor. (This value will not change during the lifetime of a Pool object).

8.
```
void *malloc BOOST_PREVENT_MACRO_SUBSTITUTION();
```

Allocates a chunk of memory. Searches in the list of memory blocks for a block that has a free chunk, and returns that free chunk if found. Otherwise, creates a new memory block, adds its free list to pool's free list,
Returns: a free chunk from that block. If a new memory block cannot be allocated, returns 0. Amortized O(1).

9.
```
void * ordered_malloc();
```

Same as malloc, only merges the free lists, to preserve order. Amortized O(1). If a new memory block cannot be allocated, returns 0. Amortized O(1).
Returns: a free chunk from that block.

10.
```
void * ordered_malloc(size_type n);
```

Gets address of a chunk n, allocating new memory if not already available.
Returns: Address of chunk n if allocated ok.

0 if not enough memory for n chunks.

11.
```
void free BOOST_PREVENT_MACRO_SUBSTITUTION(void *const chunk);
```

If a new memory block cannot be allocated, returns 0. Amortized O(1).

Same as malloc, only allocates enough contiguous chunks to cover n * requested_size bytes. Amortized O(n).

Deallocates a chunk of memory. Note that chunk may not be 0. O(1). chunk must have been previously returned by t.malloc() or t.ordered_malloc(). Assumes that chunk actually refers to a block of chunks spanning n * partition_sz bytes. deallocates each chunk in that block. Note that chunk may not be 0. O(n).
Returns: a free chunk from that block.

12.
```
void ordered_free(void *const chunk);
```

Same as above, but is order-preserving. Note that chunk may not be 0. O(N) with respect to the size of the free list. chunk must have been previously returned by t.malloc() or t.ordered_malloc().

13.
```
void free BOOST_PREVENT_MACRO_SUBSTITUTION(void *const chunks,
                                           const size_type n);
```

Assumes that chunk actually refers to a block of chunks. chunk must have been previously returned by t.ordered_malloc(n) spanning n * partition_sz bytes. Deallocates each chunk in that block. Note that chunk may not be 0. O(n).

14.
```
void ordered_free(void *const chunks, const size_type n);
```

Assumes that chunk actually refers to a block of chunks spanning n * partition_sz bytes; deallocates each chunk in that block. Note that chunk may not be 0. Order-preserving. O(N + n) where N is the size of the free list.

chunk must have been previously returned by t.malloc() or t.ordered_malloc().

15.
```
bool is_from(void *const chunk) const;
```

Returns: Returns true if chunk was allocated from u or may be returned as the result of a future allocation from u. Returns false if chunk was allocated from some other pool or may be returned as the result of a future allocation from some other pool. Otherwise, the return value is meaningless. Note that this function may not be used to reliably test random pointer values.

Class pool

boost::pool::pool — A fast memory allocator that guarantees proper alignment of all allocated chunks.

Synopsis

```
// In header: <boost/pool/pool.hpp>



// A fast memory allocator that guarantees proper alignment of all allocated
// chunks.

class pool {
};
```

Description

Whenever an object of type pool needs memory from the system, it will request it from its UserAllocator template parameter. The amount requested is determined using a doubling algorithm; that is, each time more system memory is allocated, the amount of system memory requested is doubled.

Users may control the doubling algorithm by using the following extensions.

Users may pass an additional constructor parameter to pool. This parameter is of type size_type, and is the number of chunks to request from the system the first time that object needs to allocate system memory. The default is 32. This parameter may not be 0.

Header <boost/pool/pool_alloc.hpp>

Standard Pool allocators.

provides two template types that can be used for fast and efficient memory allocation. These types both satisfy the Standard Allocator requirements [20.1.5] and the additional requirements in [20.1.5/4], so they can be used with Standard or user-supplied containers. For information on other pool-based interfaces, see the other pool interfaces.

Both of the pool allocators above satisfy all Standard Allocator requirements, as laid out in the Standard [20.1.5]. They also both satisfy the additional requirements found in [20.1.5/4]; this permits their usage with any Standard-compliant container.

In addition, the fast_pool_allocator also provides an additional allocation and an additional deallocation function:

Symbol Table
Symbol Meaning
PoolAlloc fast_pool_allocator<T, UserAllocator>
p value of type T *
Additional allocation/deallocation functions (fast_pool_allocator only)
Expression Return Type Semantic Equivalence
PoolAlloc::allocate() T * PoolAlloc::allocate(1)
PoolAlloc::deallocate(p) void PoolAlloc::deallocate(p, 1)

The typedef user_allocator publishes the value of the UserAllocator template parameter.

Notes

If the allocation functions run out of memory, they will throw std::bad_alloc.

The underlying Pool type used by the allocators is accessible through the Singleton Pool Interface. The identifying tag used for pool_allocator is pool_allocator_tag, and the tag used for fast_pool_allocator is fast_pool_allocator_tag. All template parameters of the allocators (including implementation-specific ones) determine the type of the underlying Pool, with the exception of the first parameter T, whose size is used instead.

Since the size of T is used to determine the type of the underlying Pool, each allocator for different types of the same size will share the same underlying pool. The tag class prevents pools from being shared between pool_allocator and fast_pool_allocator. For example, on a system where sizeof(int) == sizeof(void *), pool_allocator<int> and pool_allocator<void *> will both allocate/deallocate from/to the same pool.

If there is only one thread running before main() starts and after main() ends, then both allocators are completely thread-safe.

Compiler and STL Notes

A number of common STL libraries contain bugs in their using of allocators. Specifically, they pass null pointers to the deallocate function, which is explicitly forbidden by the Standard [20.1.5 Table 32]. PoolAlloc will work around these libraries if it detects them; currently, workarounds are in place for:

Borland C++ (Builder and command-line compiler) with default (RogueWave) library, ver. 5 and earlier
STLport (with any compiler), ver. 4.0 and earlier

```
namespace boost {
  struct pool_allocator_tag;

  template<typename T, typename UserAllocator, typename Mutex,
           unsigned NextSize, unsigned MaxSize>
    class pool_allocator;

  template<typename UserAllocator, typename Mutex, unsigned NextSize,
           unsigned MaxSize>
    class pool_allocator<void, UserAllocator, Mutex, NextSize, MaxSize>;

  struct fast_pool_allocator_tag;

  template<typename T, typename UserAllocator, typename Mutex,
           unsigned NextSize, unsigned MaxSize>
    class fast_pool_allocator;

  template<typename UserAllocator, typename Mutex, unsigned NextSize,
           unsigned MaxSize>
    class fast_pool_allocator<void, UserAllocator, Mutex, NextSize, MaxSize>;
}
```

Struct pool_allocator_tag

boost::pool_allocator_tag

Synopsis

```
// In header: <boost/pool/pool_alloc.hpp>


struct pool_allocator_tag {
};
```

Class template pool_allocator

boost::pool_allocator

Synopsis

```
// In header: <boost/pool/pool_alloc.hpp>

template<typename T, typename UserAllocator, typename Mutex,
         unsigned NextSize, unsigned MaxSize>
class pool_allocator {
public:
  // types
  typedef T                                    value_type;       // Allocate a pool of memory.
  typedef UserAllocator                          user_allocator;
  typedef Mutex mutex;
  typedef value_type *                           pointer;
  typedef const value_type *                     const_pointer;
  typedef value_type &                           reference;
  typedef const value_type &                     const_reference;
  typedef pool< UserAllocator >::size_type       size_type;
  typedef pool< UserAllocator >::difference_type difference_type;

  // member classes/structs/unions
  template<typename U>
  struct rebind {
    // types
    typedef pool_allocator< U, UserAllocator, Mutex, NextSize, MaxSize > other;
  };

  // construct/copy/destruct
  pool_allocator();
  template<typename U>
    pool_allocator(const pool_allocator< U, UserAllocator, Mutex, NextSize, MaxSize > &);

  // public member functions
   BOOST_STATIC_CONSTANT(unsigned, next_size = NextSize);
  bool operator==(const pool_allocator &) const;
  bool operator!=(const pool_allocator &) const;

  // public static functions
  static pointer address(reference);
  static const_pointer address(const_reference);
  static size_type max_size();
  static void construct(const pointer, const value_type &);
  static void destroy(const pointer);
  static pointer allocate(const size_type);
  static pointer allocate(const size_type, const void *);
  static void deallocate(const pointer, const size_type);
};
```

Description

`pool_allocator` public types

1. typedef Mutex mutex;

   typedef mutex publishs the value of the template parameter Mutex.

`pool_allocator` public construct/copy/destruct

1.
```
pool_allocator();
```

Construction of default singleton_pool IFF an instance of this allocator is constructed during global initialization.

2.
```
template<typename U>
  pool_allocator(const pool_allocator< U, UserAllocator, Mutex, NextSize, MaxSize > &);
```

Construction of singleton_pool using template U.

`pool_allocator` public member functions

1.
```
 BOOST_STATIC_CONSTANT(unsigned, next_size = NextSize);
```

> BOOST_STATIC_CONSTANT static const value next_size publishes the values of the template parameter NextSize.

2.
```
bool operator==(const pool_allocator &) const;
```

3.
```
bool operator!=(const pool_allocator &) const;
```

`pool_allocator` public static functions

1.
```
static pointer address(reference r);
```

2.
```
static const_pointer address(const_reference s);
```

3.
```
static size_type max_size();
```

4.
```
static void construct(const pointer ptr, const value_type & t);
```

5.
```
static void destroy(const pointer ptr);
```

6.
```
static pointer allocate(const size_type n);
```

7.
```
static pointer allocate(const size_type n, const void * const);
```

8.
```
static void deallocate(const pointer ptr, const size_type n);
```

Struct template rebind

boost::pool_allocator::rebind

Synopsis

```
// In header: <boost/pool/pool_alloc.hpp>


template<typename U>
struct rebind {
  // types
  typedef pool_allocator< U, UserAllocator, Mutex, NextSize, MaxSize > other;
};
```

Description

Specializations

- Class template pool_allocator<void, UserAllocator, Mutex, NextSize, MaxSize>

Class template pool_allocator<void, UserAllocator, Mutex, NextSize, MaxSize>

boost::pool_allocator<void, UserAllocator, Mutex, NextSize, MaxSize>

Synopsis

```
// In header: <boost/pool/pool_alloc.hpp>

template<typename UserAllocator, typename Mutex, unsigned NextSize,
         unsigned MaxSize>
class pool_allocator<void, UserAllocator, Mutex, NextSize, MaxSize> {
public:
  // types
  typedef void *       pointer;        // Pool memory allocator.
  typedef const void * const_pointer;
  typedef void         value_type;

  // member classes/structs/unions
  template<typename U>
  struct rebind {
    // types
    typedef pool_allocator< U, UserAllocator, Mutex, NextSize, MaxSize > other;
  };
};
```

Description

Struct template rebind

boost::pool_allocator<void, UserAllocator, Mutex, NextSize, MaxSize>::rebind

Synopsis

```
// In header: <boost/pool/pool_alloc.hpp>


template<typename U>
struct rebind {
  // types
  typedef pool_allocator< U, UserAllocator, Mutex, NextSize, MaxSize > other;
};
```

Struct fast_pool_allocator_tag

boost::fast_pool_allocator_tag

Synopsis

```
// In header: <boost/pool/pool_alloc.hpp>


struct fast_pool_allocator_tag {
};
```

Class template fast_pool_allocator

boost::fast_pool_allocator

Synopsis

```
// In header: <boost/pool/pool_alloc.hpp>

template<typename T, typename UserAllocator, typename Mutex,
         unsigned NextSize, unsigned MaxSize>
class fast_pool_allocator {
public:
  // types
  typedef T value_type;
  typedef UserAllocator                          user_allocator;
  typedef Mutex                                  mutex;
  typedef value_type *                           pointer;
  typedef const value_type *                     const_pointer;
  typedef value_type &                           reference;
  typedef const value_type &                     const_reference;
  typedef pool< UserAllocator >::size_type       size_type;
  typedef pool< UserAllocator >::difference_type difference_type;

  // member classes/structs/unions
  template<typename U>
  struct rebind {
    // types
    typedef fast_pool_allocator< U, UserAllocator, Mutex, NextSize, MaxSize > other;
  };

  // construct/copy/destruct
  fast_pool_allocator();
  template<typename U>
   fast_pool_allocator(const fast_pool_allocator< U, UserAllocator, Mutex, NextSize, MaxSize > &);

  // public member functions
   BOOST_STATIC_CONSTANT(unsigned, next_size = NextSize);
  void construct(const pointer, const value_type &);
  void destroy(const pointer);
  bool operator==(const fast_pool_allocator &) const;
  bool operator!=(const fast_pool_allocator &) const;

  // public static functions
  static pointer address(reference);
  static const_pointer address(const_reference);
  static size_type max_size();
  static pointer allocate(const size_type);
  static pointer allocate(const size_type, const void *);
  static pointer allocate();
  static void deallocate(const pointer, const size_type);
  static void deallocate(const pointer);
};
```

Description

`fast_pool_allocator` public types

1. typedef T value_type;

   /details pool_allocator is a more general-purpose solution, geared towards efficiently servicing requests for any number of contiguous chunks. fast_pool_allocator is also a general-purpose solution, but is geared towards efficiently servicing requests for one chunk at a time; it will work for contiguous chunks, but not as well as pool_allocator. If you are seriously concerned about per-

formance, use fast_pool_allocator when dealing with containers such as std::list, and use pool_allocator when dealing with containers such as std::vector.

`fast_pool_allocator` public construct/copy/destruct

1.
```
fast_pool_allocator();
```

2.
```
template<typename U>
  fast_pool_allocator(const fast_pool_allocator< U, UserAllocator, Mutex, NextSize, MaxSize
> &);
```

`fast_pool_allocator` public member functions

1.
```
 BOOST_STATIC_CONSTANT(unsigned, next_size = NextSize);
```

2.
```
void construct(const pointer ptr, const value_type & t);
```

3.
```
void destroy(const pointer ptr);
```

4.
```
bool operator==(const fast_pool_allocator &) const;
```

5.
```
bool operator!=(const fast_pool_allocator &) const;
```

`fast_pool_allocator` public static functions

1.
```
static pointer address(reference r);
```

2.
```
static const_pointer address(const_reference s);
```

3.
```
static size_type max_size();
```

4.
```
static pointer allocate(const size_type n);
```

5.
```
static pointer allocate(const size_type n, const void * const);
```

6.
```
static pointer allocate();
```

7.
```
static void deallocate(const pointer ptr, const size_type n);
```

8.
```
static void deallocate(const pointer ptr);
```

Struct template rebind

boost::fast_pool_allocator::rebind

Synopsis

```
// In header: <boost/pool/pool_alloc.hpp>


template<typename U>
struct rebind {
  // types
  typedef fast_pool_allocator< U, UserAllocator, Mutex, NextSize, MaxSize > other;
};
```

Specializations

- Class template fast_pool_allocator<void, UserAllocator, Mutex, NextSize, MaxSize>

Class template fast_pool_allocator<void, UserAllocator, Mutex, NextSize, MaxSize>

boost::fast_pool_allocator<void, UserAllocator, Mutex, NextSize, MaxSize>

Synopsis

```
// In header: <boost/pool/pool_alloc.hpp>

template<typename UserAllocator, typename Mutex, unsigned NextSize,
         unsigned MaxSize>
class fast_pool_allocator<void, UserAllocator, Mutex, NextSize, MaxSize> {
public:
  // types
  typedef void *       pointer;
  typedef const void * const_pointer;
  typedef void         value_type;

  // member classes/structs/unions
  template<typename U>
  struct rebind {
    // types
    typedef fast_pool_allocator< U, UserAllocator, Mutex, NextSize, MaxSize > other;
  };
};
```

Description

Struct template rebind

boost::fast_pool_allocator<void, UserAllocator, Mutex, NextSize, MaxSize>::rebind

Synopsis

```
// In header: <boost/pool/pool_alloc.hpp>


template<typename U>
struct rebind {
  // types
  typedef fast_pool_allocator< U, UserAllocator, Mutex, NextSize, MaxSize > other;
};
```

Header <boost/pool/poolfwd.hpp>

Forward declarations of all public (non-implemention) classes.

Header <boost/pool/simple_segregated_storage.hpp>

Simple Segregated Storage.

Simple Segregated Storage Implementation. 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.

```
namespace boost {
  template<typename SizeType> class simple_segregated_storage;
}
```

Class template simple_segregated_storage

boost::simple_segregated_storage

Synopsis

```
// In header: <boost/pool/simple_segregated_storage.hpp>

template<typename SizeType>
class simple_segregated_storage {
public:
  // types
  typedef SizeType size_type;

  // construct/copy/destruct
  simple_segregated_storage(const simple_segregated_storage &);
  simple_segregated_storage();
  simple_segregated_storage& operator=(const simple_segregated_storage &);

  // private static functions
  static void * try_malloc_n(void *&, size_type, size_type);

  // protected member functions
  void * find_prev(void *);

  // protected static functions
  static void *& nextof(void *const);

  // public member functions
  void add_block(void *const, const size_type, const size_type);
  void add_ordered_block(void *const, const size_type, const size_type);
  bool empty() const;
  void *malloc BOOST_PREVENT_MACRO_SUBSTITUTION();
  void free BOOST_PREVENT_MACRO_SUBSTITUTION(void *const);
  void ordered_free(void *const);
  void * malloc_n(size_type, size_type);
  void free_n(void *const, const size_type, const size_type);
  void ordered_free_n(void *const, const size_type, const size_type);

  // public static functions
  static void * segregate(void *, size_type, size_type, void * = 0);
};
```

Description

`simple_segregated_storage` public types

1. typedef SizeType size_type;

   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.

`simple_segregated_storage` public construct/copy/destruct

1. ```
   simple_segregated_storage(const simple_segregated_storage &);
   ```

2. ```
   simple_segregated_storage();
   ```

   Construct empty storage area.

Postconditions: empty()

3.
```
simple_segregated_storage& operator=(const simple_segregated_storage &);
```

`simple_segregated_storage` private static functions

1.
```
static void *
try_malloc_n(void *& start, size_type n, size_type partition_size);
```

The following function attempts to find n contiguous chunks of size partition_size in the free list, starting at start. If it succeds, it returns the last chunk in that contiguous sequence, so that the sequence is known by [start, {retval}] If it fails, it does do either because it's at the end of the free list or hits a non-contiguous chunk. In either case, it will return 0, and set start to the last considered chunk. You are at the end of the free list if nextof(start) == 0. Otherwise, start points to the last chunk in the contiguous sequence, and nextof(start) points to the first chunk in the next contiguous sequence (assuming an ordered free list).

`simple_segregated_storage` protected member functions

1.
```
void * find_prev(void * ptr);
```

Note that this function finds the location previous to where ptr would go if it was in the free list. It does not find the entry in the free list before ptr (unless ptr is already in the free list). Specifically, find_prev(0) will return 0, not the last entry in the free list.

`simple_segregated_storage` protected static functions

1.
```
static void *& nextof(void *const ptr);
```

The return value is just *ptr cast to the appropriate type. ptr must not be 0. (For the sake of code readability :)

As an example, let us assume that we want to truncate the free list after the first chunk. That is, we want to set *first to 0; this will result in a free list with only one entry. The normal way to do this is to first cast first to a pointer to a pointer to void, and then dereference and assign (*static_cast<void **>(first) = 0;). This can be done more easily through the use of this convenience function (nextof(first) = 0;).
Returns: dereferenced pointer.

`simple_segregated_storage` public member functions

1.
```
void add_block(void *const block, const size_type nsz,
               const size_type npartition_sz);
```

Add block Segregate this block and merge its free list into the free list referred to by "first".
Requires: Same as segregate.
Postconditions: !empty()

2.
```
void add_ordered_block(void *const block, const size_type nsz,
                       const size_type npartition_sz);
```

add block (ordered into list) This (slower) version of add_block segregates the block and merges its free list into our free list in the proper order.

3.
```
bool empty() const;
```

Returns: true if simple_segregated_storage is empty.

4.
```
void *malloc BOOST_PREVENT_MACRO_SUBSTITUTION();
```

Create a chunk
Requires: !empty() Increment the "first" pointer to point to the next chunk.

5.
```
void free BOOST_PREVENT_MACRO_SUBSTITUTION(void *const chunk);
```

Freen a chunk.
Requires: chunk was previously returned from a malloc() referring to the same free list.
Postconditions: !empty()

6.
```
void ordered_free(void *const chunk);
```

This (slower) implementation of 'free' places the memory back in the list in its proper order.
Requires: chunk was previously returned from a malloc() referring to the same free list
Postconditions: !empty().

7.
```
void * malloc_n(size_type n, size_type partition_size);
```

8.
```
void free_n(void *const chunks, const size_type n,
            const size_type partition_size);
```

Free N chunks. values for n and partition_size.
Requires: chunks was previously allocated from *this with the same values for n and partition_size.
Requires: chunks was previously allocated from *this with the same
Postconditions: !empty() Note: if you're allocating/deallocating n a lot, you should be using an ordered pool.

9.
```
void ordered_free_n(void *const chunks, const size_type n,
                    const size_type partition_size);
```

Free n chunkcs from order list.

Requires: chunks was previously allocated from *this with the same values for n and partition_size.

`simple_segregated_storage` public static functions

1.
```
static void *
segregate(void * block, size_type nsz, size_type npartition_sz,
          void * end = 0);
```

Get pointer to last valid chunk, preventing overflow on size calculations The division followed by the multiplication just makes sure that old == block + partition_sz * i, for some integer i, even if the block size (sz) is not a multiple of the partition size.

Header <boost/pool/singleton_pool.hpp>

Singleton_pool class allows other pool interfaces for types of the same size to share the same pool.

/details singleton_pool.hpp provides a template class singleton_pool, which provides access to a pool as a singleton object.
For information on other pool-based interfaces, see the other pool interfaces.

Notes
The underlying pool p referenced by the static functions in singleton_pool is actually declared in a way that it is:

1 Thread-safe if there is only one thread running before main() begins and after main() ends -- all of the static functions of singleton_pool synchronize their access to p.

2 Guaranteed to be constructed before it is used -- thus, the simple static object in the synopsis above would actually be an incorrect implementation. The actual implementation to guarantee this is considerably more complicated.

3 Note too that a different underlying pool p exists for each different set of template parameters, including implementation-specific ones.

```
namespace boost {
  template<typename Tag, unsigned RequestedSize, typename UserAllocator,
           typename Mutex, unsigned NextSize, unsigned MaxSize>
    struct singleton_pool;
}
```

Struct template singleton_pool

boost::singleton_pool

Synopsis

```
// In header: <boost/pool/singleton_pool.hpp>

template<typename Tag, unsigned RequestedSize, typename UserAllocator,
         typename Mutex, unsigned NextSize, unsigned MaxSize>
struct singleton_pool {
  // types
  typedef Tag tag;
  typedef Mutex                                  mutex;
  typedef UserAllocator                          user_allocator;
  typedef pool< UserAllocator >::size_type       size_type;
  typedef pool< UserAllocator >::difference_type difference_type;

  // member classes/structs/unions

  struct pool_type {
    // construct/copy/destruct
    pool_type();
    pool< UserAllocator > p;
  };

  // construct/copy/destruct
  singleton_pool();

  // public member functions
   BOOST_STATIC_CONSTANT(unsigned, requested_size = RequestedSize);
   BOOST_STATIC_CONSTANT(unsigned, next_size = NextSize);

  // public static functions
  static void *malloc BOOST_PREVENT_MACRO_SUBSTITUTION();
  static void * ordered_malloc();
  static void * ordered_malloc(const size_type);
  static bool is_from(void *const);
  static void free BOOST_PREVENT_MACRO_SUBSTITUTION(void *const);
  static void ordered_free(void *const);
  static void free
  BOOST_PREVENT_MACRO_SUBSTITUTION(void *const, const size_type);
  static void ordered_free(void *const, const size_type);
  static bool release_memory();
  static bool purge_memory();
};
```

Description

`singleton_pool` public types

1. typedef Tag tag;

   The Tag template parameter allows different unbounded sets of singleton pools to exist. For example, the pool allocators use two tag classes to ensure that the two different allocator types never share the same underlying singleton pool. Tag is never actually used by singleton_pool.

`singleton_pool` public construct/copy/destruct

1. ```
   singleton_pool();
   ```
   ```

`singleton_pool` public member functions

1.
```
BOOST_STATIC_CONSTANT(unsigned, requested_size = RequestedSize);
```

2.
```
BOOST_STATIC_CONSTANT(unsigned, next_size = NextSize);
```

`singleton_pool` public static functions

1.
```
static void *malloc BOOST_PREVENT_MACRO_SUBSTITUTION();
```

Equivalent to SingletonPool::p.malloc(); synchronized.

2.
```
static void * ordered_malloc();
```

Equivalent to SingletonPool::p.ordered_malloc(); synchronized.

3.
```
static void * ordered_malloc(const size_type n);
```

Equivalent to SingletonPool::p.ordered_malloc(n); synchronized.

4.
```
static bool is_from(void *const ptr);
```

Equivalent to SingletonPool::p.is_from(chunk); synchronized.
Returns: true if chunk is from SingletonPool::is_from(chunk)

5.
```
static void free BOOST_PREVENT_MACRO_SUBSTITUTION(void *const ptr);
```

Equivalent to SingletonPool::p.free(chunk); synchronized.

6.
```
static void ordered_free(void *const ptr);
```

Equivalent to SingletonPool::p.ordered_free(chunk); synchronized.

7.
```
static void free
BOOST_PREVENT_MACRO_SUBSTITUTION(void *const ptr, const size_type n);
```

Equivalent to SingletonPool::p.free(chunk, n); synchronized.

8.
```
static void ordered_free(void *const ptr, const size_type n);
```

Equivalent to SingletonPool::p.ordered_free(chunk, n); synchronized.

9.
```
static bool release_memory();
```

Equivalent to SingletonPool::p.release_memory(); synchronized.

10.
```
static bool purge_memory();
```

Equivalent to SingletonPool::p.purge_memory(); synchronized.

Struct pool_type

boost::singleton_pool::pool_type

Synopsis

```
// In header: <boost/pool/singleton_pool.hpp>



struct pool_type {
  // construct/copy/destruct
  pool_type();
  pool< UserAllocator > p;
};
```

Description

`pool_type` public construct/copy/destruct

1. ```
   pool_type();
   ```

   Pool allocation

   Requires: NextSize value should never be set to 0.

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.