Chapter 4. Boost.Pool

Boost.Pool is a library that contains a few classes to manage memory. While C++ programs usually use `new` to allocate memory dynamically, the details of how memory is provided depends on the implementation of the standard library and the operating system. With Boost.Pool you can, for example, accelerate memory management to provide memory to your program faster.

Boost.Pool doesn’t change the behavior of `new` or of the operating system. Boost.Pool works because the managed memory is requested from the operating system first – for example using `new`. From the outside, your program has already allocated the memory, but internally, the memory isn’t required yet and is handed over to Boost.Pool to manage it.

Boost.Pool partitions memory segments with the same size. Every time you request memory from Boost.Pool, the library accesses the next free segment and assigns memory from that segment to you. The entire segment is then marked as used, no matter how many bytes you actually need from that segment.

This memory management concept is called *simple segregated storage*. This is the only concept supported by Boost.Pool. It is especially useful if many objects of the same size have to be created and destroyed frequently. In this case the required memory can be provided and released quickly.

Boost.Pool provides the class `boost::simple_segregated_storage` to create and manage segregated memory. `boost::simple_segregated_storage` is a low-level class that you usually will not use in your programs directly. It is only used in Example 4.1 to illustrate simple segregated storage. All other classes from Boost.Pool are internally based on `boost::simple_segregated_storage`.

Example 4.1. Using `boost::simple_segregated_storage`

```
#include <boost/pool/simple_segregated_storage.hpp>
#include <vector>
#include <cstddef>

int main()
{
  boost::simple_segregated_storage<std::size_t> storage;
  std::vector<char> v(1024);
  storage.add_block(&v.front(), v.size(), 256);

  int *i = static_cast<int*>(storage.malloc());
  *i = 1;

  int *j = static_cast<int*>(storage.malloc_n(1, 512));
  j[10] = 2;

  storage.free(i);
  storage.free_n(j, 1, 512);
}
```

The header file `boost/pool/simple_segregated_storage.hpp` must be included to use the class template `boost::simple_segregated_storage`. Example 4.1 passes `std::size_t` as the template parameter. This parameter specifies which type should be used for numbers passed to member functions of boost::simple_segregated_storage to refer, for example, to the size of a segment. The practical relevance of this template parameter is rather low.

More interesting are the member functions called on boost::simple_segregated_storage. First, add_block() is called to pass a memory block with 1024 bytes to **storage**. The memory is provided by the vector **v**. The third parameter passed to add_block() specifies that the memory block should be partitioned in segments with 256 bytes each. Because the total size of the memory block is 1024 bytes, the memory managed by **storage** consists of four segments.

The calls to malloc() and malloc_n() request memory from **storage**. While malloc() returns a pointer to a free segment, malloc_n() returns a pointer to one or more contiguous segments that provide as many bytes in one block as requested. Example 4.1 requests a block with 512 bytes with malloc_n(). This call consumes two segments, since each segment is 256 bytes. After the calls to malloc() and malloc_n(), **storage** has only one unused segment left.

At the end of the example, all segments are released with free() and free_n(). After these two calls, all segments are available and could be requested again with malloc() or malloc_n().

You usually don't use boost::simple_segregated_storage directly. Boost.Pool provides other classes that allocate memory automatically without requiring you to allocate memory yourself and pass it to boost::simple_segregated_storage.

Example 4.2. Using boost::object_pool

```
#include <boost/pool/object_pool.hpp>

int main()
{
  boost::object_pool<int> pool;

  int *i = pool.malloc();
  *i = 1;

  int *j = pool.construct(2);

  pool.destroy(i);
  pool.destroy(j);
}
```

Example 4.2 uses the class boost::object_pool, which is defined in boost/pool/object_pool.hpp. Unlike boost::simple_segregated_storage, boost::object_pool knows the type of the objects that will be stored in memory. **pool** in Example 4.2 is simple segregated storage for int values. The memory managed by **pool** consists of segments, each of which is the size of an int – 4 bytes for example.

Another difference is that you don't need to provide memory to boost::object_pool. boost::object_pool allocates memory automatically. In Example 4.2, the call to malloc() makes **pool** allocate a memory block with space for 32 int values. malloc() returns a pointer to the first of these 32 segments that an int value can fit into exactly.

Please note that malloc() returns a pointer of type int*. Unlike boost::simple_segregated_storage in Example 4.1, no cast operator is required.

construct() is similar to malloc() but initializes an object via a call to the constructor. In Example 4.2, **j** refers to an int object initialized with the value 2.

Please note that **pool** can return a free segment from the pool of 32 segments when construct() is called. The call to construct() does not make Example 4.2 request memory from the operating system.

The last member function called in Example 4.2 is destroy(), which releases an int object.

Example 4.3. Changing the segment size with boost::object_pool

```
#include <boost/pool/object_pool.hpp>
#include <iostream>

int main()
{
  boost::object_pool<int> pool{32, 0};
  pool.construct();
  std::cout << pool.get_next_size() << '\n';
  pool.set_next_size(8);
}
```

You can pass two parameters to the constructor of boost::object_pool. The first parameter sets the size of the memory block that boost::object_pool will allocate when the first segment is requested with a call to malloc() or construct(). The second parameter sets the maximum size of the memory block to allocate.

If malloc() or construct() are called so often that all segments in a memory block are used, the next call to one of these member functions will cause boost::object_pool to allocate a new memory block, which will be twice as big as the previous one. The size will double each time a new memory block is allocated by boost::object_pool. boost::object_pool can manage an arbitrary number of memory blocks, but their sizes will grow exponentially. The second constructor parameter lets you limit the growth.

The default constructor of boost::object_pool does the same as what the call to the constructor in Example 4.3 does. The first parameter sets the size of the memory block to 32 int values. The second parameter specifies that there is no maximum size. If 0 is passed, boost::object_pool can double the size of the memory block indefinitely.

The call to construct() in Example 4.3 makes **pool** allocate a memory block of 32 int values. **pool** can serve up to 32 calls to malloc() or construct() without requesting memory from the operating system. If more memory is required, the next memory block to allocate will have space for 64 int values.

get_next_size() returns the size of the next memory block to allocate. set_next_size() lets you set the size of the next memory block. In Example 4.3 get_next_size() returns 64. The call to set_next_size() changes the size of the next memory block to allocate from 64 to 8 int values. With `set_next_size()` the size of the next memory block can be changed directly. If you only want to set a maximum size, pass it via the second parameter to the constructor.

With `boost::singleton_pool`, Boost.Pool provides a class between `boost::simple_segregated_storage` and `boost::object_pool` (see Example 4.4).

Example 4.4. Using `boost::singleton_pool`

```
#include <boost/pool/singleton_pool.hpp>

struct int_pool {};
typedef boost::singleton_pool<int_pool, sizeof(int)> singleton_int_pool;

int main()
{
  int *i = static_cast<int*>(singleton_int_pool::malloc());
  *i = 1;

  int *j = static_cast<int*>(singleton_int_pool::ordered_malloc(10));
  j[9] = 2;

  singleton_int_pool::release_memory();
  singleton_int_pool::purge_memory();
}
```

`boost::singleton_pool` is defined in `boost/pool/singleton_pool.hpp`. This class is similar to `boost::simple_segregated_storage` since it also expects the segment size as a template parameter but not the type of the objects to store. That’s why member functions such as `ordered_malloc()` and `malloc()`return a pointer of type `void*`, which must be cast explicitly.

This class is also similar to `boost::object_pool` because it allocates memory automatically. The size of the next memory block and an optional maximum size are passed as template parameters. Here `boost::singleton_pool` differs from `boost::object_pool`: you can’t change the size of the next memory block in `boost::singleton_pool` at run time.

You can create multiple objects with `boost::singleton_pool` if you want to manage several memory pools. The first template parameter passed to `boost::singleton_pool` is a *tag*. The tag is an arbitrary type that serves as a name for the memory pool. Example 4.4 uses the structure `int_pool` as a tag to highlight that `singleton_int_pool` is a pool that manages `int` values. Thanks to tags, multiple singletons can manage different memory pools, even if the second template parameter for the size is the same. The tag has no purpose other than creating separate instances of `boost::singleton_pool`.

`boost::singleton_pool` provides two member functions to release memory: `release_memory()` releases all memory blocks that aren’t used at the moment, and `purge_memory()` releases all memory blocks – including those currently being used. The call to `purge_memory()` resets `boost::singleton_pool`.

`release_memory()` and `purge_memory()` return memory to the operating system. To return memory to `boost::singleton_pool` instead of the operating system, call member functions such as `free()` or `ordered_free()`.

`boost::object_pool` and `boost::singleton_pool` allow you to request memory explicitly. You do this by calling member functions such as `malloc()` or `construct()`. Boost.Pool also provides the class `boost::pool_allocator`, which you can pass as an allocator to containers (see Example 4.5).

Example 4.5. Using `boost::pool_allocator`

```
#include <boost/pool/pool_alloc.hpp>
#include <vector>

int main()
{
  std::vector<int, boost::pool_allocator<int>> v;
  for (int i = 0; i < 1000; ++i)
    v.push_back(i);

  v.clear();
  boost::singleton_pool<boost::pool_allocator_tag, sizeof(int)>::
    purge_memory();
}
```

`boost::pool_allocator` is defined in `boost/pool/pool_alloc.hpp`. The class is an allocator that is usually passed as a second template parameter to containers from the standard library. The allocator provides memory required by the container.

`boost::pool_allocator` is based on `boost::singleton_pool`. To release memory, you have to use a tag to access `boost::singleton_pool` and call `purge_memory()` or `release_memory()`. Example 4.5 uses the tag `boost::pool_allocator_tag`. This tag is defined by Boost.Pool and is used by `boost::pool_allocator` for the internal `boost::singleton_pool`.

When Example 4.5 calls `push_back()` the first time, **v** accesses the allocator to get the requested memory. Because the allocator `boost::pool_allocator` is used, a memory block with space for 32 `int` values is allocated. **v** receives the pointer to the first segment in that memory block that has the size of an `int`. With every subsequent call to `push_back()`, another segment is used from the memory block until the allocator detects that a bigger memory block is required.

Please note that you should call `clear()` on a container before you release memory with `purge_memory()` (see Example 4.5). A call to `purge_memory()` releases memory but doesn't notify the container that it doesn't own the memory anymore. A call to `release_memory()` is less dangerous because it only releases memory blocks that aren't in use.

Boost.Pool also provides an allocator called `boost::fast_pool_allocator` (see Example 4.6).

Example 4.6. Using `boost::fast_pool_allocator`

```
#define BOOST_POOL_NO_MT
#include <boost/pool/pool_alloc.hpp>
#include <list>

int main()
{
  typedef boost::fast_pool_allocator<int,
    boost::default_user_allocator_new_delete,
```

```
    boost::details::pool::default_mutex,
    64, 128> allocator;

  std::list<int, allocator> l;
  for (int i = 0; i < 1000; ++i)
    l.push_back(i);

  l.clear();
  boost::singleton_pool<boost::fast_pool_allocator_tag, sizeof(int)>::
    purge_memory();
}
```

Both allocators are used in the same way, but `boost::pool_allocator` should be preferred if you are requesting contiguous segments. `boost::fast_pool_allocator` can be used if segments are requested one by one. Grossly simplified: You use `boost::pool_allocator` for `std::vector` and `boost::fast_pool_allocator` for `std::list`.

Example 4.6 illustrates which template parameters can be passed to `boost::fast_pool_allocator`. `boost::pool_allocator` accepts the same parameters.

`boost::default_user_allocator_new_delete` is a class that allocates memory blocks with `new` and releases them with `delete[]`. You can also use `boost::default_user_allocator_malloc_free`, which calls `malloc()` and `free()`.

`boost::details::pool::default_mutex` is a type definition that is set to `boost::mutex` or `boost::details::pool::null_mutex`. `boost::mutex` is the default type that supports multiple threads requesting memory from the allocator. If the macro `BOOST_POOL_NO_MT` is defined as in Example 4.6, multithreading support for Boost.Pool is disabled. The allocator in Example 4.6 uses a null mutex.

The last two parameters passed to `boost::fast_pool_allocator` in Example 4.6 set the size of the first memory block and the maximum size of memory blocks to allocate.