Boost.Interprocess

Ion Gaztanaga

Copyright © 2005 - 2010 Ion Gaztanaga

Distributed under the Boost Software License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)

Table of Contents

Introduction

Boost.Interprocess simplifies the use of common interprocess communication and synchronization mechanisms and offers a wide range of them:

- Shared memory.

- Memory-mapped files.

- Semaphores, mutexes, condition variables and upgradable mutex types to place them in shared memory and memory mapped files.

- Named versions of those synchronization objects, similar to UNIX/Windows sem_open/CreateSemaphore API.

- File locking.

- Relative pointers.

- Message queues.

Boost.Interprocess also offers higher-level interprocess mechanisms to allocate dynamically portions of a shared memory or a memory mapped file (in general, to allocate portions of a fixed size memory segment). Using these mechanisms, **Boost.Interprocess** offers useful tools to construct C++ objects, including STL-like containers, in shared memory and memory mapped files:

- Dynamic creation of anonymous and named objects in a shared memory or memory mapped file.

- STL-like containers compatible with shared memory/memory-mapped files.

- STL-like allocators ready for shared memory/memory-mapped files implementing several memory allocation patterns (like pooling).

Building Boost.Interprocess

There is no need to compile **Boost.Interprocess**, since it's a header only library. Just include your Boost header directory in your compiler include path.

Boost.Interprocess depends on **Boost.DateTime**, which needs separate compilation. However, the subset used by **Boost.Interprocess** does not need any separate compilation so the user can define `BOOST_DATE_TIME_NO_LIB` to avoid Boost from trying to automatically link the **Boost.DateTime**.

In POSIX systems, **Boost.Interprocess** uses pthread system calls to implement classes like mutexes, condition variables, etc... In some operating systems, these POSIX calls are implemented in separate libraries that are not automatically linked by the compiler. For example, in some Linux systems POSIX pthread functions are implemented in `librt.a` library, so you might need to add that library when linking an executable or shared library that uses **Boost.Interprocess**. If you obtain linking errors related to those pthread functions, please revise your system's documentation to know which library implements them.

Tested compilers

Boost.Interprocess has been tested in the following compilers/platforms:

- Visual 7.1 Windows XP

- Visual 8.0 Windows XP

- GCC 4.1.1 MinGW

- GCC 3.4.4 Cygwin

- Intel 9.1 Windows XP

- GCC 4.1.2 Linux

- GCC 3.4.3 Solaris 11

- GCC 4.0 MacOs 10.4.1

Quick Guide for the Impatient

Using shared memory as a pool of unnamed memory blocks

You can just allocate a portion of a shared memory segment, copy the message to that buffer, send the offset of that portion of shared memory to another process, and you are done. Let's see the example:

```cpp
#include <boost/interprocess/managed_shared_memory.hpp>
#include <cstdlib> //std::system
#include <sstream>

int main (int argc, char *argv[])
{
   using namespace boost::interprocess;
   if(argc == 1){  //Parent process
      //Remove shared memory on construction and destruction
      struct shm_remove
      {
         shm_remove() {  shared_memory_object::remove("MySharedMemory"); }
         ~shm_remove(){  shared_memory_object::remove("MySharedMemory"); }
      } remover;

      //Create a managed shared memory segment
      managed_shared_memory segment(create_only, "MySharedMemory", 65536);

      //Allocate a portion of the segment (raw memory)
      std::size_t free_memory = segment.get_free_memory();
      void * shptr = segment.allocate(1024/*bytes to allocate*/);

      //Check invariant
      if(free_memory <= segment.get_free_memory())
         return 1;

      //An handle from the base address can identify any byte of the shared
      //memory segment even if it is mapped in different base addresses
      managed_shared_memory::handle_t handle = segment.get_handle_from_address(shptr);
      std::stringstream s;
      s << argv[0] << " " << handle;
      s << std::ends;
      //Launch child process
      if(0 != std::system(s.str().c_str()))
         return 1;
      //Check memory has been freed
      if(free_memory != segment.get_free_memory())
         return 1;
   }
   else{
      //Open managed segment
      managed_shared_memory segment(open_only, "MySharedMemory");

      //An handle from the base address can identify any byte of the shared
      //memory segment even if it is mapped in different base addresses
      managed_shared_memory::handle_t handle = 0;

      //Obtain handle value
      std::stringstream s; s << argv[1]; s >> handle;

      //Get buffer local address from handle
      void *msg = segment.get_address_from_handle(handle);
```

```
        //Deallocate previously allocated memory
        segment.deallocate(msg);
    }
    return 0;
}
```

Creating named shared memory objects

You want to create objects in a shared memory segment, giving a string name to them so that any other process can find, use and delete them from the segment when the objects are not needed anymore. Example:

```cpp
#include <boost/interprocess/managed_shared_memory.hpp>
#include <cstdlib> //std::system
#include <cstddef>
#include <cassert>
#include <utility>

int main(int argc, char *argv[])
{
   using namespace boost::interprocess;
   typedef std::pair<double, int> MyType;

   if(argc == 1){  //Parent process
      //Remove shared memory on construction and destruction
      struct shm_remove
      {
         shm_remove() { shared_memory_object::remove("MySharedMemory"); }
         ~shm_remove(){ shared_memory_object::remove("MySharedMemory"); }
      } remover;

      //Construct managed shared memory
      managed_shared_memory segment(create_only, "MySharedMemory", 65536);

      //Create an object of MyType initialized to {0.0, 0}
      MyType *instance = segment.construct<MyType>
         ("MyType instance")  //name of the object
         (0.0, 0);            //ctor first argument

      //Create an array of 10 elements of MyType initialized to {0.0, 0}
      MyType *array = segment.construct<MyType>
         ("MyType array")     //name of the object
         [10]                 //number of elements
         (0.0, 0);            //Same two ctor arguments for all objects

      //Create an array of 3 elements of MyType initializing each one
      //to a different value {0.0, 0}, {1.0, 1}, {2.0, 2}...
      float float_initializer[3] = { 0.0, 1.0, 2.0 };
      int   int_initializer[3]   = { 0, 1, 2 };

      MyType *array_it = segment.construct_it<MyType>
         ("MyType array from it")  //name of the object
         [3]                       //number of elements
         ( &float_initializer[0]   //Iterator for the 1st ctor argument
         , &int_initializer[0]);   //Iterator for the 2nd ctor argument

      //Launch child process
      std::string s(argv[0]); s += " child ";
      if(0 != std::system(s.c_str()))
         return 1;
```

```cpp
      //Check child has destroyed all objects
      if(segment.find<MyType>("MyType array").first ||
         segment.find<MyType>("MyType instance").first ||
         segment.find<MyType>("MyType array from it").first)
         return 1;
   }
   else{
      //Open managed shared memory
      managed_shared_memory segment(open_only, "MySharedMemory");

      std::pair<MyType*, std::size_t> res;

      //Find the array
      res = segment.find<MyType> ("MyType array");
      //Length should be 10
      if(res.second != 10) return 1;

      //Find the object
      res = segment.find<MyType> ("MyType instance");
      //Length should be 1
      if(res.second != 1) return 1;

      //Find the array constructed from iterators
      res = segment.find<MyType> ("MyType array from it");
      //Length should be 3
      if(res.second != 3) return 1;

      //We're done, delete all the objects
      segment.destroy<MyType>("MyType array");
      segment.destroy<MyType>("MyType instance");
      segment.destroy<MyType>("MyType array from it");
   }
   return 0;
}
```

Using an offset smart pointer for shared memory

Boost.Interprocess offers offset_ptr smart pointer family as an offset pointer that stores the distance between the address of the offset pointer itself and the address of the pointed object. When offset_ptr is placed in a shared memory segment, it can point safely objects stored in the same shared memory segment, even if the segment is mapped in different base addresses in different processes.

This allows placing objects with pointer members in shared memory. For example, if we want to create a linked list in shared memory:

```cpp
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/offset_ptr.hpp>

using namespace boost::interprocess;

//Shared memory linked list node
struct list_node
{
   offset_ptr<list_node> next;
   int                   value;
};

int main ()
{
   //Remove shared memory on construction and destruction
   struct shm_remove
   {
      shm_remove() { shared_memory_object::remove("MySharedMemory"); }
      ~shm_remove(){ shared_memory_object::remove("MySharedMemory"); }
   } remover;

   //Create shared memory
   managed_shared_memory segment(create_only,
                                 "MySharedMemory",  //segment name
                                 65536);

   //Create linked list with 10 nodes in shared memory
   offset_ptr<list_node> prev = 0, current, first;

   int i;
   for(i = 0; i < 10; ++i, prev = current){
      current = static_cast<list_node*>(segment.allocate(sizeof(list_node)));
      current->value = i;
      current->next  = 0;

      if(!prev)
         first = current;
      else
         prev->next = current;
   }

   //Communicate list to other processes
   //. . .
   //When done, destroy list
   for(current = first; current; /**/){
      prev = current;
      current = current->next;
      segment.deallocate(prev.get());
   }
   return 0;
}
```

To help with basic data structures, **Boost.Interprocess** offers containers like vector, list, map, so you can avoid these manual data structures just like with standard containers.

Creating vectors in shared memory

Boost.Interprocess allows creating complex objects in shared memory and memory mapped files. For example, we can construct STL-like containers in shared memory. To do this, we just need to create a special (managed) shared memory segment, declare a **Boost.Interprocess** allocator and construct the vector in shared memory just if it was any other object.

The class that allows this complex structures in shared memory is called `boost::interprocess::managed_shared_memory` and it's easy to use. Just execute this example without arguments:

```cpp
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/containers/vector.hpp>
#include <boost/interprocess/allocators/allocator.hpp>
#include <string>
#include <cstdlib> //std::system

using namespace boost::interprocess;

//Define an STL compatible allocator of ints that allocates from the managed_shared_memory.
//This allocator will allow placing containers in the segment
typedef allocator<int, managed_shared_memory::segment_manager>  ShmemAllocator;

//Alias a vector that uses the previous STL-like allocator so that allocates
//its values from the segment
typedef vector<int, ShmemAllocator> MyVector;

//Main function. For parent process argc == 1, for child process argc == 2
int main(int argc, char *argv[])
{
   if(argc == 1){ //Parent process
      //Remove shared memory on construction and destruction
      struct shm_remove
      {
          shm_remove() { shared_memory_object::remove("MySharedMemory"); }
          ~shm_remove(){ shared_memory_object::remove("MySharedMemory"); }
      } remover;

      //Create a new segment with given name and size
      managed_shared_memory segment(create_only, "MySharedMemory", 65536);

      //Initialize shared memory STL-compatible allocator
      const ShmemAllocator alloc_inst (segment.get_segment_manager());

      //Construct a vector named "MyVector" in shared memory with argument alloc_inst
      MyVector *myvector = segment.construct<MyVector>("MyVector")(alloc_inst);

      for(int i = 0; i < 100; ++i)  //Insert data in the vector
         myvector->push_back(i);

      //Launch child process
      std::string s(argv[0]); s += " child ";
      if(0 != std::system(s.c_str()))
         return 1;

      //Check child has destroyed the vector
      if(segment.find<MyVector>("MyVector").first)
         return 1;
   }
   else{ //Child process
      //Open the managed segment
      managed_shared_memory segment(open_only, "MySharedMemory");

      //Find the vector using the c-string name
      MyVector *myvector = segment.find<MyVector>("MyVector").first;

      //Use vector in reverse order
      std::sort(myvector->rbegin(), myvector->rend());

      //When done, destroy the vector from the segment
```

```
      segment.destroy<MyVector>("MyVector");
   }

   return 0;
};
```

The parent process will create an special shared memory class that allows easy construction of many complex data structures associated with a name. The parent process executes the same program with an additional argument so the child process opens the shared memory and uses the vector and erases it.

Creating maps in shared memory

Just like a vector, **Boost.Interprocess** allows creating maps in shared memory and memory mapped files. The only difference is that like standard associative containers, **Boost.Interprocess**'s map needs also the comparison functor when an allocator is passed in the constructor:

```cpp
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/containers/map.hpp>
#include <boost/interprocess/allocators/allocator.hpp>
#include <functional>
#include <utility>

int main ()
{
   using namespace boost::interprocess;

   //Remove shared memory on construction and destruction
   struct shm_remove
   {
      shm_remove() { shared_memory_object::remove("MySharedMemory"); }
      ~shm_remove(){ shared_memory_object::remove("MySharedMemory"); }
   } remover;

   //Shared memory front-end that is able to construct objects
   //associated with a c-string. Erase previous shared memory with the name
   //to be used and create the memory segment at the specified address and initialize resources
   managed_shared_memory segment
      (create_only
      ,"MySharedMemory" //segment name
      ,65536);          //segment size in bytes

   //Note that map<Key, MappedType>'s value_type is std::pair<const Key, MappedType>,
   //so the allocator must allocate that pair.
   typedef int    KeyType;
   typedef float  MappedType;
   typedef std::pair<const int, float> ValueType;

   //Alias an STL compatible allocator of for the map.
   //This allocator will allow to place containers
   //in managed shared memory segments
   typedef allocator<ValueType, managed_shared_memory::segment_manager>
      ShmemAllocator;

   //Alias a map of ints that uses the previous STL-like allocator.
   //Note that the third parameter argument is the ordering function
   //of the map, just like with std::map, used to compare the keys.
   typedef map<KeyType, MappedType, std::less<KeyType>, ShmemAllocator> MyMap;

   //Initialize the shared memory STL-compatible allocator
   ShmemAllocator alloc_inst (segment.get_segment_manager());
```

```
    //Construct a shared memory map.
    //Note that the first parameter is the comparison function,
    //and the second one the allocator.
    //This the same signature as std::map's constructor taking an allocator
    MyMap *mymap =
        segment.construct<MyMap>("MyMap")      //object name
                                 (std::less<int>() //first  ctor parameter
                                 ,alloc_inst);     //second ctor parameter

    //Insert data in the map
    for(int i = 0; i < 100; ++i){
        mymap->insert(std::pair<const int, float>(i, (float)i));
    }
    return 0;
}
```

For a more advanced example including containers of containers, see the section Containers of containers.

Some basic explanations

Processes And Threads

Boost.Interprocess does not work only with processes but also with threads. **Boost.Interprocess** synchronization mechanisms can synchronize threads from different processes, but also threads from the same process.

Sharing information between processes

In the traditional programming model an operating system has multiple processes running and each process has its own address space. To share information between processes we have several alternatives:

- Two processes share information using a **file**. To access to the data, each process uses the usual file read/write mechanisms. When updating/reading a file shared between processes, we need some sort of synchronization, to protect readers from writers.

- Two processes share information that resides in the **kernel** of the operating system. This is the case, for example, of traditional message queues. The synchronization is guaranteed by the operating system kernel.

- Two processes can share a **memory** region. This is the case of classical shared memory or memory mapped files. Once the processes set up the memory region, the processes can read/write the data like any other memory segment without calling the operating system's kernel. This also requires some kind of manual synchronization between processes.

Persistence Of Interprocess Mechanisms

One of the biggest issues with interprocess communication mechanisms is the lifetime of the interprocess communication mechanism. It's important to know when an interprocess communication mechanism disappears from the system. In **Boost.Interprocess**, we can have 3 types of persistence:

- **Process-persistence**: The mechanism lasts until all the processes that have opened the mechanism close it, exit or crash.

- **Kernel-persistence**: The mechanism exists until the kernel of the operating system reboots or the mechanism is explicitly deleted.

- **Filesystem-persistence**: The mechanism exists until the mechanism is explicitly deleted.

Some native POSIX and Windows IPC mechanisms have different persistence so it's difficult to achieve portability between Windows and POSIX native mechanisms. **Boost.Interprocess** classes have the following persistence:

Table 1. Boost.Interprocess Persistence Table

Mechanism	Persistence
Shared memory	Kernel or Filesystem
Memory mapped file	Filesystem
Process-shared mutex types	Process
Process-shared semaphore	Process
Process-shared condition	Process
File lock	Process
Message queue	Kernel or Filesystem
Named mutex	Kernel or Filesystem
Named semaphore	Kernel or Filesystem
Named condition	Kernel or Filesystem

As you can see, **Boost.Interprocess** defines some mechanisms with "Kernel or Filesystem" persistence. This is because POSIX allows this possibility to native interprocess communication implementations. One could, for example, implement shared memory using memory mapped files and obtain filesystem persistence (for example, there is no proper known way to emulate kernel persistence with a user library for Windows shared memory using native shared memory, or process persistence for POSIX shared memory, so the only portable way is to define "Kernel or Filesystem" persistence).

Names Of Interprocess Mechanisms

Some interprocess mechanisms are anonymous objects created in shared memory or memory-mapped files but other interprocess mechanisms need a name or identifier so that two unrelated processes can use the same interprocess mechanism object. Examples of this are shared memory, named mutexes and named semaphores (for example, native windows CreateMutex/CreateSemaphore API family).

The name used to identify an interprocess mechanism is not portable, even between UNIX systems. For this reason, **Boost.Interprocess** limits this name to a C++ variable identifier or keyword:

• Starts with a letter, lowercase or uppercase, such as a letter from a to z or from A to Z. Examples: *Sharedmemory, sharedmemory, sHaReDmEmOrY...*

• Can include letters, underscore, or digits. Examples: *shm1, shm2and3, ShM3plus4...*

Constructors, destructors and lifetime of Interprocess named resources

Named **Boost.Interprocess** resources (shared memory, memory mapped files, named mutexes/conditions/semaphores) have kernel or filesystem persistency. This means that even if all processes that have opened those resources end, the resource will still be accessible to be opened again and the resource can only be destructed via an explicit to their static member `remove` function. This behavior can be easily understood, since it's the same mechanism used by functions controlling file opening/creation/erasure:

Table 2. Boost.Interprocess-Filesystem Analogy

Named Interprocess resource	Corresponding std file	Corresponding POSIX operation
Constructor	std::fstream constructor	open
Destructor	std::fstream destructor	close
Member `remove`	None. `std::remove`	unlink

Now the correspondence between POSIX and Boost.Interprocess regarding shared memory and named semaphores:

Table 3. Boost.Interprocess-POSIX shared memory

`shared_memory_object` operation	POSIX operation
Constructor	shm_open
Destructor	close
Member `remove`	shm_unlink

Table 4. Boost.Interprocess-POSIX named semaphore

`named_semaphore` operation	POSIX operation
Constructor	sem_open
Destructor	close
Member `remove`	sem_unlink

The most important property is that **destructors of named resources don't remove the resource from the system**, they only liberate resources allocated by the system for use by the process for the named resource. **To remove the resource from the system the programmer must use `remove`**.

Permissions

Named resources offered by **Boost.Interprocess** must cope with platform-dependant permission issues also present when creating files. If a programmer wants to shared shared memory, memory mapped files or named synchronization mechanisms (mutexes, semaphores, etc...) between users, it's necessary to specify those permissions. Sadly, traditional UNIX and Windows permissions are very different and **Boost.Interprocess** does not try to standardize permissions, but does not ignore them.

All named resource creation functions take an optional permissions object that can be configured with platform-dependant permissions.

Since each mechanism can be emulated through diferent mechanisms (a semaphore might be implement using mapped files or native semaphores) permissions types could vary when the implementation of a named resource changes (eg.: in Windows mutexes require synchronize permissions, but that's not the case of files). To avoid this, **Boost.Interprocess** relies on file-like permissions, requiring file read-write-delete permissions to open named synchronization mechanisms (mutex, semaphores, etc.) and appropiate read or read-write-delete permissions for shared memory. This approach has two advantages: it's similar to the UNIX philosophy and the programmer does not need to know how the named resource is implemented.

Sharing memory between processes

Shared memory

What is shared memory?

Shared memory is the fastest interprocess communication mechanism. The operating system maps a memory segment in the address space of several processes, so that several processes can read and write in that memory segment without calling operating system functions. However, we need some kind of synchronization between processes that read and write shared memory.

Consider what happens when a server process wants to send an HTML file to a client process that resides in the same machine using network mechanisms:

- The server must read the file to memory and pass it to the network functions, that copy that memory to the OS's internal memory.

- The client uses the network functions to copy the data from the OS's internal memory to its own memory.

As we can see, there are two copies, one from memory to the network and another one from the network to memory. And those copies are made using operating system calls that normally are expensive. Shared memory avoids this overhead, but we need to synchronize both processes:

- The server maps a shared memory in its address space and also gets access to a synchronization mechanism. The server obtains exclusive access to the memory using the synchronization mechanism and copies the file to memory.

- The client maps the shared memory in its address space. Waits until the server releases the exclusive access and uses the data.

Using shared memory, we can avoid two data copies, but we have to synchronize the access to the shared memory segment.

Creating memory segments that can be shared between processes

To use shared memory, we have to perform 2 basic steps:

- Request to the operating system a memory segment that can be shared between processes. The user can create/destroy/open this memory using a **shared memory object**: *An object that represents memory that can be mapped concurrently into the address space of more than one process..*

- Associate a part of that memory or the whole memory with the address space of the calling process. The operating system looks for a big enough memory address range in the calling process' address space and marks that address range as an special range. Changes in that address range are automatically seen by other process that also have mapped the same shared memory object.

Once the two steps have been successfully completed, the process can start writing to and reading from the address space to send to and receive data from other processes. Now, let's see how can we do this using **Boost.Interprocess**:

Header

To manage shared memory, you just need to include the following header:

```
#include <boost/interprocess/shared_memory_object.hpp>
```

Creating shared memory segments

As we've mentioned we have to use the `shared_memory_object` class to create, open and destroy shared memory segments that can be mapped by several processes. We can specify the access mode of that shared memory object (read only or read-write), just as if it was a file:

- Create a shared memory segment. Throws if already created:

```
using boost::interprocess;
shared_memory_object shm_obj
   (create_only                  //only create
   ,"shared_memory"              //name
   ,read_write                   //read-write mode
   );
```

• To open or create a shared memory segment:

```
using boost::interprocess;
shared_memory_object shm_obj
   (open_or_create               //open or create
   ,"shared_memory"              //name
   ,read_only                    //read-only mode
   );
```

• To only open a shared memory segment. Throws if does not exist:

```
using boost::interprocess;
shared_memory_object shm_obj
   (open_only                    //only open
   ,"shared_memory"              //name
   ,read_write                   //read-write mode
   );
```

When a shared memory object is created, its size is 0. To set the size of the shared memory, the user must use the `truncate` function call, in a shared memory that has been opened with read-write attributes:

```
shm_obj.truncate(10000);
```

As shared memory has kernel or filesystem persistence, the user must explicitly destroy it. The `remove` operation might fail returning false if the shared memory does not exist, the file is open or the file is still memory mapped by other processes:

```
using boost::interprocess;
shared_memory_object::remove("shared_memory");
```

For more details regarding `shared_memory_object` see the `boost::interprocess::shared_memory_object` class reference.

Mapping Shared Memory Segments

Once created or opened, a process just has to map the shared memory object in the process' address space. The user can map the whole shared memory or just part of it. The mapping process is done using the `mapped_region` class. The class represents a memory region that has been mapped from a shared memory or from other devices that have also mapping capabilities (for example, files). A `mapped_region` can be created from any `memory_mappable` object and as you might imagine, `shared_memory_object` is a `memory_mappable` object:

```
using boost::interprocess;
std::size_t ShmSize = ...

//Map the second half of the memory
mapped_region region
   ( shm                      //Memory-mappable object
   , read_write               //Access mode
   , ShmSize/2                //Offset from the beginning of shm
   , ShmSize-ShmSize/2        //Length of the region
   );

//Get the address of the region
region.get_address();

//Get the size of the region
region.get_size();
```

The user can specify the offset from the mappable object where the mapped region should start and the size of the mapped region. If no offset or size is specified, the whole mappable object (in this case, shared memory) is mapped. If the offset is specified, but not the size, the mapped region covers from the offset until the end of the mappable object.

For more details regarding `mapped_region` see the `boost::interprocess::mapped_region` class reference.

A Simple Example

Let's see a simple example of shared memory use. A server process creates a shared memory object, maps it and initializes all the bytes to a value. After that, a client process opens the shared memory, maps it, and checks that the data is correctly initialized:

```cpp
#include <boost/interprocess/shared_memory_object.hpp>
#include <boost/interprocess/mapped_region.hpp>
#include <cstring>
#include <cstdlib>
#include <string>

int main(int argc, char *argv[])
{
   using namespace boost::interprocess;

   if(argc == 1){  //Parent process
      //Remove shared memory on construction and destruction
      struct shm_remove
      {
         shm_remove() { shared_memory_object::remove("MySharedMemory"); }
         ~shm_remove(){ shared_memory_object::remove("MySharedMemory"); }
      } remover;

      //Create a shared memory object.
      shared_memory_object shm (create_only, "MySharedMemory", read_write);

      //Set size
      shm.truncate(1000);

      //Map the whole shared memory in this process
      mapped_region region(shm, read_write);

      //Write all the memory to 1
      std::memset(region.get_address(), 1, region.get_size());

      //Launch child process
      std::string s(argv[0]); s += " child ";
      if(0 != std::system(s.c_str()))
         return 1;
   }
   else{
      //Open already created shared memory object.
      shared_memory_object shm (open_only, "MySharedMemory", read_only);

      //Map the whole shared memory in this process
      mapped_region region(shm, read_only);

      //Check that memory was initialized to 1
      char *mem = static_cast<char*>(region.get_address());
      for(std::size_t i = 0; i < region.get_size(); ++i)
         if(*mem++ != 1)
            return 1;   //Error checking memory
   }
   return 0;
}
```

Emulation for systems without shared memory objects

Boost.Interprocess provides portable shared memory in terms of POSIX semantics. Some operating systems don't support shared memory as defined by POSIX:

• Windows operating systems provide shared memory using memory backed by the paging file but the lifetime semantics are different from the ones defined by POSIX (see Native windows shared memory section for more information).

• Some UNIX systems don't fully support POSIX shared memory objects at all.

In those platforms, shared memory is emulated with mapped files created in a "boost_interprocess" folder created in a temporary files directory. In Windows platforms, if "Common AppData" key is present in the registry, "boost_interprocess" folder is created in that directory (in XP usually "C:\Documents and Settings\All Users\Application Data" and in Vista "C:\ProgramData"). For Windows platforms without that registry key and Unix systems, shared memory is created in the system temporary files directory ("/tmp" or similar).

Because of this emulation, shared memory has filesystem lifetime in some of those systems.

Removing shared memory

`shared_memory_object` provides a static `remove` function to remove a shared memory objects.

This function **can** fail if the shared memory objects does not exist or it's opened by another process. Note that this function is similar to the standard C `int remove(const char *path)` function. In UNIX systems, `shared_memory_object::remove` calls `shm_unlink`:

- The function will remove the name of the shared memory object named by the string pointed to by name.

- If one or more references to the shared memory object exist when is unlinked, the name will be removed before the function returns, but the removal of the memory object contents will be postponed until all open and map references to the shared memory object have been removed.

- Even if the object continues to exist after the last function call, reuse of the name will subsequently cause the creation of a `boost::interprocess::shared_memory_object` instance to behave as if no shared memory object of this name exists (that is, trying to open an object with that name will fail and an object of the same name can be created again).

In Windows operating systems, current version supports an usually acceptable emulation of the UNIX unlink behaviour: the file is renamed with a random name and marked as *to be deleted when the last open handle is closed*.

Anonymous shared memory for UNIX systems

Creating a shared memory segment and mapping it can be a bit tedious when several processes are involved. When processes are related via `fork()` operating system call in UNIX systems a simpler method is available using anonymous shared memory.

This feature has been implemented in UNIX systems mapping the device `\dev\zero` or just using the `MAP_ANONYMOUS` in a POSIX conformant `mmap` system call.

This feature is wrapped in **Boost.Interprocess** using the `anonymous_shared_memory()` function, which returns a `mapped_region` object holding an anonymous shared memory segment that can be shared by related processes.

Here is an example:

```cpp
#include <boost/interprocess/anonymous_shared_memory.hpp>
#include <boost/interprocess/mapped_region.hpp>
#include <iostream>
#include <cstring>

int main ()
{
   using namespace boost::interprocess;
   try{
      //Create an anonymous shared memory segment with size 1000
      mapped_region region(anonymous_shared_memory(1000));

      //Write all the memory to 1
      std::memset(region.get_address(), 1, region.get_size());

      //The segment is unmapped when "region" goes out of scope
   }
   catch(interprocess_exception &ex){
      std::cout << ex.what() << std::endl;
      return 1;
   }
   return 0;
}
```

Once the segment is created, a `fork()` call can be used so that `region` is used to communicate two related processes.

Native windows shared memory

Windows operating system also offers shared memory, but the lifetime of this shared memory is very different to kernel or filesystem lifetime. The shared memory is created backed by the pagefile and it's automatically destroyed when the last process attached to the shared memory is destroyed.

Because of this reason, there is no effective way to simulate kernel or filesystem persistence using native windows shared memory and **Boost.Interprocess** emulates shared memory using memory mapped files. This assures portability between POSIX and Windows operating systems.

However, accessing native windows shared memory is a common request of **Boost.Interprocess** users because they want to access to shared memory created with other process that don't use **Boost.Interprocess**. In order to manage the native windows shared memory **Boost.Interprocess** offers the `windows_shared_memory` class.

Windows shared memory creation is a bit different from portable shared memory creation: the size of the segment must be specified when creating the object and can't be specified through `truncate` like with the shared memory object.

Take in care that when the last process attached to a shared memory is destroyed **the shared memory is destroyed** so there is **no persistency** with native windows shared memory. Native windows shared memory has also another limitation: a process can open and map the whole shared memory created by another process but it can't know which is the size of that memory. This limitation is imposed by the Windows API so the user must somehow transmit the size of the segment to processes opening the segment.

Sharing memory between services and user applications is also different. To share memory between services and user applications the name of the shared memory must start with the global namespace prefix `"Global\\"`. This global namespace enables processes on multiple client sessions to communicate with a service application. The server component can create the shared memory in the global namespace. Then a client session can use the "Global" prefix to open that memory.

The creation of a shared memory object in the global namespace from a session other than session zero is a privileged operation.

Let's repeat the same example presented for the portable shared memory object: A server process creates a shared memory object, maps it and initializes all the bytes to a value. After that, a client process opens the shared memory, maps it, and checks that the data is correctly initialized. Take in care that **if the server exits before the client connects to the shared memory the client connection will fail**, because the shared memory segment is destroyed when no proces is attached to the memory.

This is the server process:

```cpp
#include <boost/interprocess/windows_shared_memory.hpp>
#include <boost/interprocess/mapped_region.hpp>
#include <cstring>
#include <cstdlib>
#include <string>

int main(int argc, char *argv[])
{
   using namespace boost::interprocess;

   if(argc == 1){  //Parent process
      //Create a native windows shared memory object.
      windows_shared_memory shm (create_only, "MySharedMemory", read_write, 1000);

      //Map the whole shared memory in this process
      mapped_region region(shm, read_write);

      //Write all the memory to 1
      std::memset(region.get_address(), 1, region.get_size());

      //Launch child process
      std::string s(argv[0]); s += " child ";
      if(0 != std::system(s.c_str()))
         return 1;
      //windows_shared_memory is destroyed when the last attached process dies...
   }
   else{
      //Open already created shared memory object.
      windows_shared_memory shm (open_only, "MySharedMemory", read_only);

      //Map the whole shared memory in this process
      mapped_region region(shm, read_only);

      //Check that memory was initialized to 1
      char *mem = static_cast<char*>(region.get_address());
      for(std::size_t i = 0; i < region.get_size(); ++i)
         if(*mem++ != 1)
            return 1;   //Error checking memory
      return 0;
   }
   return 0;
}
```

As we can see, native windows shared memory needs synchronization to make sure that the shared memory won't be destroyed before the client is launched.

XSI shared memory

In many UNIX systems, the OS offers another shared memory memory mechanism, XSI (X/Open System Interfaces) shared memory segments, also known as "System V" shared memory. This shared memory mechanism is quite popular and portable, and it's not based in file-mapping semantics, but it uses special functions (shmget, shmat, shmdt, shmctl...).

Unlike POSIX shared memory segments, XSI shared memory segments are not identified by names but by 'keys' usually created with ftok. XSI shared memory segments have kernel lifetime and must be explicitly removed. XSI shared memory does not support copy-on-write and partial shared memory mapping but it supports anonymous shared memory.

Boost.Interprocess offers simple (xsi_shared_memory) and managed (managed_xsi_shared_memory) shared memory classes to ease the use of XSI shared memory. It also wraps key creation with the simple xsi_key class.

Let's repeat the same example presented for the portable shared memory object: A server process creates a shared memory object, maps it and initializes all the bytes to a value. After that, a client process opens the shared memory, maps it, and checks that the data is correctly initialized.

This is the server process:

```cpp
#include <boost/interprocess/xsi_shared_memory.hpp>
#include <boost/interprocess/mapped_region.hpp>
#include <cstring>
#include <cstdlib>
#include <string>

using namespace boost::interprocess;

void remove_old_shared_memory(const xsi_key &key)
{
   try{
      xsi_shared_memory xsi(open_only, key);
      xsi_shared_memory::remove(xsi.get_shmid());
   }
   catch(interprocess_exception &e){
      if(e.get_error_code() != not_found_error)
         throw;
   }
}

int main(int argc, char *argv[])
{
   if(argc == 1){  //Parent process
      //Build XSI key (ftok based)
      xsi_key key(argv[0], 1);

      remove_old_shared_memory(key);

      //Create a shared memory object.
      xsi_shared_memory shm (create_only, key, 1000);

      //Remove shared memory on destruction
      struct shm_remove
      {
         int shmid_;
         shm_remove(int shmid) : shmid_(shmid){}
         ~shm_remove(){ xsi_shared_memory::remove(shmid_); }
      } remover(shm.get_shmid());

      //Map the whole shared memory in this process
      mapped_region region(shm, read_write);

      //Write all the memory to 1
      std::memset(region.get_address(), 1, region.get_size());

      //Launch child process
      std::string s(argv[0]); s += " child ";
      if(0 != std::system(s.c_str()))
         return 1;
   }
   else{
      //Build XSI key (ftok based)
      xsi_key key(argv[0], 1);

      //Create a shared memory object.
      xsi_shared_memory shm (open_only, key);
```

```
        //Map the whole shared memory in this process
        mapped_region region(shm, read_only);

        //Check that memory was initialized to 1
        char *mem = static_cast<char*>(region.get_address());
        for(std::size_t i = 0; i < region.get_size(); ++i)
            if(*mem++ != 1)
                return 1;   //Error checking memory
    }
    return 0;
}
```

As we can see, native windows shared memory needs synchronization to make sure that the shared memory won't be destroyed before the client is launched.

Memory Mapped Files

What is a memory mapped file?

File mapping is the association of a file's contents with a portion of the address space of a process. The system creates a file mapping to associate the file and the address space of the process. A mapped region is the portion of address space that the process uses to access the file's contents. A single file mapping can have several mapped regions, so that the user can associate parts of the file with the address space of the process without mapping the entire file in the address space, since the file can be bigger than the whole address space of the process (a 9GB DVD image file in a usual 32 bit systems). Processes read from and write to the file using pointers, just like with dynamic memory. File mapping has the following advantages:

• Uniform resource use. Files and memory can be treated using the same functions.

• Automatic file data synchronization and cache from the OS.

• Reuse of C++ utilities (STL containers, algorithms) in files.

• Shared memory between two or more applications.

• Allows efficient work with a large files, without mapping the whole file into memory

• If several processes use the same file mapping to create mapped regions of a file, each process' views contain identical copies of the file on disk.

File mapping is not only used for interprocess communication, it can be used also to simplify file usage, so the user does not need to use file-management functions to write the file. The user just writes data to the process memory, and the operating systems dumps the data to the file.

When two processes map the same file in memory, the memory that one process writes is seen by another process, so memory mapped files can be used as an interprocess communication mechanism. We can say that memory-mapped files offer the same interprocess communication services as shared memory with the addition of filesystem persistence. However, as the operating system has to synchronize the file contents with the memory contents, memory-mapped files are not as fast as shared memory.

Using mapped files

To use memory-mapped files, we have to perform 2 basic steps:

• Create a mappable object that represent an already created file of the filesystem. This object will be used to create multiple mapped regions of the the file.

• Associate the whole file or parts of the file with the address space of the calling process. The operating system looks for a big enough memory address range in the calling process' address space and marks that address range as an special range. Changes in that address range are automatically seen by other process that also have mapped the same file and those changes are also transferred to the disk automatically.

Once the two steps have been successfully completed, the process can start writing to and reading from the address space to send to and receive data from other processes and synchronize the file's contents with the changes made to the mapped region. Now, let's see how can we do this using **Boost.Interprocess**:

Header

To manage mapped files, you just need to include the following header:

```
#include <boost/interprocess/file_mapping.hpp>
```

Creating a file mapping

First, we have to link a file's contents with the process' address space. To do this, we have to create a mappable object that represents that file. This is achieved in **Boost.Interprocess** creating a `file_mapping` object:

```
using boost::interprocess;
file_mapping m_file
   ("/usr/home/file"        //filename
   ,read_write              //read-write mode
   );
```

Now we can use the newly created object to create mapped regions. For more details regarding this class see the `boost::interprocess::file_mapping` class reference.

Mapping File's Contents In Memory

After creating a file mapping, a process just has to map the shared memory in the process' address space. The user can map the whole shared memory or just part of it. The mapping process is done using the `mapped_region` class. as we have said before The class represents a memory region that has been mapped from a shared memory or from other devices that have also mapping capabilities:

```
using boost::interprocess;
std::size_t FileSize = ...

//Map the second half of the file
mapped_region region
   ( m_file                 //Memory-mappable object
   , read_write             //Access mode
   , FileSize/2             //Offset from the beginning of shm
   , FileSize-FileSize/2    //Length of the region
   );

//Get the address of the region
region.get_address();

//Get the size of the region
region.get_size();
```

The user can specify the offset from the file where the mapped region should start and the size of the mapped region. If no offset or size is specified, the whole file is mapped. If the offset is specified, but not the size, the mapped region covers from the offset until the end of the file.

If several processes map the same file, and a process modifies a memory range from a mapped region that is also mapped by other process, the changes are inmedially visible to other processes. However, the file contents on disk are not updated immediately, since that would hurt performance (writing to disk is several times slower than writing to memory). If the user wants to make sure that file's contents have been updated, it can flush a range from the view to disk. When the function returns, the flushing process has startd but there is not guarantee that all data has been written to disk:

```
//Flush the whole region
region.flush();

//Flush from an offset until the end of the region
region.flush(offset);

//Flush a memory range starting on an offset
region.flush(offset, size);
```

Remember that the offset is **not** an offset on the file, but an offset in the mapped region. If a region covers the second half of a file and flushes the whole region, only the half of the file is guaranteed to have been flushed.

For more details regarding `mapped_region` see the `boost::interprocess::mapped_region` class reference.

A Simple Example

Let's reproduce the same example described in the shared memory section, using memory mapped files. A server process creates a shared memory segment, maps it and initializes all the bytes to a value. After that, a client process opens the shared memory, maps it, and checks that the data is correctly initialized::

```cpp
#include <boost/interprocess/file_mapping.hpp>
#include <boost/interprocess/mapped_region.hpp>
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <cstring>
#include <cstddef>
#include <cstdlib>

int main(int argc, char *argv[])
{
   using namespace boost::interprocess;

   //Define file names
   const char *FileName  = "file.bin";
   const std::size_t FileSize = 10000;

   if(argc == 1){ //Parent process executes this
      {  //Create a file
         file_mapping::remove(FileName);
         std::filebuf fbuf;
         fbuf.open(FileName, std::ios_base::in | std::ios_base::out
                            | std::ios_base::trunc | std::ios_base::binary);
         //Set the size
         fbuf.pubseekoff(FileSize-1, std::ios_base::beg);
         fbuf.sputc(0);
      }

      //Remove on exit
      struct file_remove
      {
         file_remove(const char *FileName)
            : FileName_(FileName) {}
         ~file_remove(){ file_mapping::remove(FileName_); }
         const char *FileName_;
      } remover(FileName);

      //Create a file mapping
      file_mapping m_file(FileName, read_write);
```

```
      //Map the whole file with read-write permissions in this process
      mapped_region region(m_file, read_write);

      //Get the address of the mapped region
      void * addr       = region.get_address();
      std::size_t size  = region.get_size();

      //Write all the memory to 1
      std::memset(addr, 1, size);

      //Launch child process
      std::string s(argv[0]); s += " child ";
      if(0 != std::system(s.c_str()))
         return 1;
   }
   else{  //Child process executes this
      {  //Open the file mapping and map it as read-only
         file_mapping m_file(FileName, read_only);

         mapped_region region(m_file, read_only);

         //Get the address of the mapped region
         void * addr       = region.get_address();
         std::size_t size  = region.get_size();

         //Check that memory was initialized to 1
         const char *mem = static_cast<char*>(addr);
         for(std::size_t i = 0; i < size; ++i)
            if(*mem++ != 1)
               return 1;   //Error checking memory
      }
      {  //Now test it reading the file
         std::filebuf fbuf;
         fbuf.open(FileName, std::ios_base::in | std::ios_base::binary);

         //Read it to memory
         std::vector<char> vect(FileSize, 0);
         fbuf.sgetn(&vect[0], std::streamsize(vect.size()));

         //Check that memory was initialized to 1
         const char *mem = static_cast<char*>(&vect[0]);
         for(std::size_t i = 0; i < FileSize; ++i)
            if(*mem++ != 1)
               return 1;   //Error checking memory
      }
   }

   return 0;
}
```

More About Mapped Regions

One Class To Rule Them All

As we have seen, both `shared_memory_object` and `file_mapping` objects can be used to create `mapped_region` objects. A mapped region created from a shared memory object or a file mapping are the same class and this has many advantages.

One can, for example, mix in STL containers mapped regions from shared memory and memory mapped files. The libraries that only depend on mapped regions can be used to work with shared memory or memory mapped files without recompiling them.

Mapping Address In Several Processes

In the example we have seen, the file or shared memory contents are mapped to the address space of the process, but the address was chosen by the operating system.

If several processes map the same file/shared memory, the mapping address will be surely different in each process. Since each process might have used its address space in a different way (allocation of more or less dynamic memory, for example), there is no guarantee that the file/shared memory is going to be mapped in the same address.

If two processes map the same object in different addresses, this invalids the use of pointers in that memory, since the pointer (which is an absolute address) would only make sense for the process that wrote it. The solution for this is to use offsets (distance) between objects instead of pointers: If two objects are placed in the same shared memory segment by one process, **the address of each object will be different** in another process but **the distance between them (in bytes) will be the same**.

So the first advice when mapping shared memory and memory mapped files is to avoid using raw pointers, unless you know what you are doing. Use offsets between data or relative pointers to obtain pointer functionality when an object placed in a mapped region wants to point to an object placed in the same mapped region. **Boost.Interprocess** offers a smart pointer called `boost::interprocess::offset_ptr` that can be safely placed in shared memory and that can be used to point to another object placed in the same shared memory / memory mapped file.

Fixed Address Mapping

The use of relative pointers is less efficient than using raw pointers, so if a user can succeed mapping the same file or shared memory object in the same address in two processes, using raw pointers can be a good idea.

To map an object in a fixed address, the user can specify that address in the `mapped region`'s constructor:

```
mapped_region region ( shm                    //Map shared memory
                     , read_write             //Map it as read-write
                     , 0                       //Map from offset 0
                     , 0                       //Map until the end
                     , (void*)0x3F000000       //Map it exactly there
                     );
```

However, the user can't map the region in any address, even if the address is not being used. The offset parameter that marks the start of the mapping region is also limited. These limitations are explained in the next section.

Mapping Offset And Address Limitations

As mentioned, the user can't map the memory mappable object at any address and it can specify the offset of the mappable object that is equivalent to the start of the mapping region to an arbitrary value. Most operating systems limit the mapping address and the offset of the mappable object to a multiple of a value called **page size**. This is due to the fact that the **operating system performs mapping operations over whole pages**.

If fixed mapping address is used, *offset* and *address* parameters should be multiples of that value. This value is, typically, 4KB or 8KB for 32 bit operating systems.

```
//These might fail because the offset is not a multiple of the page size
//and we are using fixed address mapping
mapped_region region1( shm                  //Map shared memory
                     , read_write           //Map it as read-write
                     , 1                    //Map from offset 1
                     , 1                    //Map 1 byte
                     , (void*)0x3F000000    //Aligned mapping address
                     );

//These might fail because the address is not a multiple of the page size
mapped_region region2( shm                  //Map shared memory
                     , read_write           //Map it as read-write
                     , 0                    //Map from offset 0
                     , 1                    //Map 1 byte
                     , (void*)0x3F000001    //Not aligned mapping address
                     );
```

Since the operating system performs mapping operations over whole pages, specifying a mapping *size* or *offset* that are not multiple of the page size will waste more resources than necessary. If the user specifies the following 1 byte mapping:

```
//Map one byte of the shared memory object.
//A whole memory page will be used for this.
mapped_region region ( shm                  //Map shared memory
                     , read_write           //Map it as read-write
                     , 0                    //Map from offset 0
                     , 1                    //Map 1 byte
                     );
```

The operating system will reserve a whole page that will not be reused by any other mapping so we are going to waste **(page size - 1)** bytes. If we want to use efficiently operating system resources, we should create regions whose size is a multiple of **page size** bytes. If the user specifies the following two mapped regions for a file with which has `2*page_size` bytes:

```
//Map the first quarter of the file
//This will use a whole page
mapped_region region1( shm                  //Map shared memory
                     , read_write           //Map it as read-write
                     , 0                    //Map from offset 0
                     , page_size/2          //Map page_size/2 bytes
                     );

//Map the rest of the file
//This will use a 2 pages
mapped_region region2( shm                  //Map shared memory
                     , read_write           //Map it as read-write
                     , page_size/2          //Map from offset 0
                     , 3*page_size/2        //Map the rest of the shared memory
                     );
```

In this example, a half of the page is wasted in the first mapping and another half is wasted in the second because the offset is not a multiple of the page size. The mapping with the minimum resource usage would be to map whole pages:

```
//Map the whole first half: uses 1 page
mapped_region region1( shm                 //Map shared memory
                     , read_write          //Map it as read-write
                     , 0                    //Map from offset 0
                     , page_size           //Map a full page_size
                     );

//Map the second half: uses 1 page
mapped_region region2( shm                 //Map shared memory
                     , read_write          //Map it as read-write
                     , page_size           //Map from offset 0
                     , page_size           //Map the rest
                     );
```

How can we obtain the **page size**? The `mapped_region` class has a static function that returns that value:

```
//Obtain the page size of the system
std::size_t page_size = mapped_region::get_page_size();
```

The operating system might also limit the number of mapped memory regions per process or per system.

Limitations When Constructing Objects In Mapped Regions

When two processes create a mapped region of the same mappable object, two processes can communicate writing and reading that memory. A process could construct a C++ object in that memory so that the second process can use it. However, a mapped region shared by multiple processes, can't hold any C++ object, because not every class is ready to be a process-shared object, specially, if the mapped region is mapped in different address in each process.

Offset pointers instead of raw pointers

When placing objects in a mapped region and mapping that region in different address in every process, raw pointers are a problem since they are only valid for the process that placed them there. To solve this, **Boost.Interprocess** offers a special smart pointer that can be used instead of a raw pointer. So user classes containing raw pointers (or Boost smart pointers, that internally own a raw pointer) can't be safely placed in a process shared mapped region. These pointers must be replaced with offset pointers, and these pointers must point only to objects placed in the same mapped region if you want to use these shared objects from different processes.

Of course, a pointer placed in a mapped region shared between processes should only point to an object of that mapped region. Otherwise, the pointer would point to an address that it's only valid one process and other processes may crash when accessing to that address.

References forbidden

References suffer from the same problem as pointers (mainly because they are implemented as pointers). However, it is not possible to create a fully workable smart reference currently in C++ (for example, `operator .()` can't be overloaded). Because of this, if the user wants to put an object in shared memory, the object can't have any (smart or not) reference as a member.

References will only work if the mapped region is mapped in the same base address in all processes sharing a memory segment. Like pointers, a reference placed in a mapped region should only point to an object of that mapped region.

Virtuality forbidden

The virtual table pointer and the virtual table are in the address space of the process that constructs the object, so if we place a class with a virtual function or virtual base class, the virtual pointer placed in shared memory will be invalid for other processes and they will crash.

This problem is very difficult to solve, since each process needs a different virtual table pointer and the object that contains that pointer is shared across many processes. Even if we map the mapped region in the same address in every process, the virtual table

can be in a different address in every process. To enable virtual functions for objects shared between processes, deep compiler changes are needed and virtual functions would suffer a performance hit. That's why **Boost.Interprocess** does not have any plan to support virtual function and virtual inheritance in mapped regions shared between processes.

Be careful with static class members

Static members of classes are global objects shared by all instances of the class. Because of this, static members are implemented as global variables in processes.

When constructing a class with static members, each process has its own copy of the static member, so updating a static member in one process does not change the value of the static member the another process. So be careful with these classes. Static members are not dangerous if they are just constant variables initialized when the process starts, but they don't change at all (for example, when used like enums) and their value is the same for all processes.

Mapping Address Independent Pointer: offset_ptr

When creating shared memory and memory mapped files to communicate two processes the memory segment can be mapped in a different address in each process:

```cpp
#include<boost/interprocess/shared_memory_object.hpp>

// ...

using boost::interprocess;

//Open a shared memory segment
shared_memory_object shm_obj
   (open_only                  //open or create
   ,"shared_memory"            //name
   ,read_only   //read-only mode
   );

//Map the whole shared memory
mapped_region region
   ( shm                       //Memory-mappable object
   , read_write                //Access mode
   );

//This address can be different in each process
void *addr = region.get_address();
```

This makes the creation of complex objects in mapped regions difficult: a C++ class instance placed in a mapped region might have a pointer pointing to another object also placed in the mapped region. Since the pointer stores an absolute address, that address is only valid for the process that placed the object there unless all processes map the mapped region in the same address.

To be able to simulate pointers in mapped regions, users must use **offsets** (distance between objects) instead of absolute addresses. The offset between two objects in a mapped region is the same for any process that maps the mapped region, even if that region is placed in different base addresses. To facilitate the use of offsets, **Boost.Interprocess** offers offset_ptr.

offset_ptr wraps all the background operations needed to offer a pointer-like interface. The class interface is inspired in Boost Smart Pointers and this smart pointer stores the offset (distance in bytes) between the pointee's address and it's own this pointer. Imagine a structure in a common 32 bit processor:

```
struct structure
{
   int              integer1;   //The compiler places this at offset 0 in the structure
   offset_ptr<int>  ptr;        //The compiler places this at offset 4 in the structure
   int              integer2;   //The compiler places this at offset 8 in the structure
};

//...

structure s;

//Assign the address of "integer1" to "ptr".
//"ptr" will store internally "-4":
//    (char*)&s.integer1 - (char*)&s.ptr;
s.ptr = &s.integer1;

//Assign the address of "integer2" to "ptr".
//"ptr" will store internally "4":
//    (char*)&s.integer2 - (char*)&s.ptr;
s.ptr = &s.integer2;
```

One of the big problems of offset_ptr is the representation of the null pointer. The null pointer can't be safely represented like an offset, since the absolute address 0 is always outside of the mapped region. Due to the fact that the segment can be mapped in a different base address in each process the distance between the address 0 and offset_ptr is different for every process.

Some implementations choose the offset 0 (that is, an offset_ptr pointing to itself) as the null pointer pointer representation but this is not valid for many use cases since many times structures like linked lists or nodes from STL containers point to themselves (the end node in an empty container, for example) and 0 offset value is needed. An alternative is to store, in addition to the offset, a boolean to indicate if the pointer is null. However, this increments the size of the pointer and hurts performance.

In consequence, offset_ptr defines offset 1 as the null pointer, meaning that this class **can't** point to the byte after its own *this* pointer:

```
using namespace boost::interprocess;

offset_ptr<char> ptr;

//Pointing to the next byte of it's own address
//marks the smart pointer as null.
ptr = (char*)&ptr + 1;

//ptr is equal to null
assert(!ptr);

//This is the same as assigning the null value...
ptr = 0;

//ptr is also equal to null
assert(!ptr);
```

In practice, this limitation is not important, since a user almost never wants to point to this address.

offset_ptr offers all pointer-like operations and random_access_iterator typedefs, so it can be used in STL algorithms requiring random access iterators and detected via traits. For more information about the members and operations of the class, see offset_ptr reference.

Synchronization mechanisms

Synchronization mechanisms overview

As mentioned before, the ability to shared memory between processes through memory mapped files or shared memory objects is not very useful if the access to that memory can't be effectively synchronized. This is the same problem that happens with thread-synchronization mechanisms, where heap memory and global variables are shared between threads, but the access to these resources needs to be synchronized typically through mutex and condition variables. **Boost.Threads** implements these synchronization utilities between threads inside the same process. **Boost.Interprocess** implements similar mechanisms to synchronize threads from different processes.

Named And Anonymous Synchronization Mechanisms

Boost.Interprocess presents two types of synchronization objects:

- **Named utilities**: When two processes want to create an object of such type, both processes must *create* or *open* an object using the same name. This is similar to creating or opening files: a process creates a file with using a `fstream` with the name *filename* and another process opens that file using another `fstream` with the same *filename* argument. **Each process uses a different object to access to the resource, but both processes are using the same underlying resource**.

- **Anonymous utilities**: Since these utilities have no name, two processes must share **the same object** through shared memory or memory mapped files. This is similar to traditional thread synchronization objects: **Both processes share the same object**. Unlike thread synchronization, where global variables and heap memory is shared between threads of the same process, sharing objects between two threads from different process can be only possible through mapped regions that map the same mappable resource (for example, shared memory or memory mapped files).

Each type has it's own advantages and disadvantages:

- Named utilities are easier to handle for simple synchronization tasks, since both process don't have to create a shared memory region and construct the synchronization mechanism there.

- Anonymous utilities can be serialized to disk when using memory mapped objects obtaining automatic persistence of synchronization utilities. One could construct a synchronization utility in a memory mapped file, reboot the system, map the file again, and use the synchronization utility again without any problem. This can't be achieved with named synchronization utilities.

The main interface difference between named and anonymous utilities are the constructors. Usually anonymous utilities have only one constructor, whereas the named utilities have several constructors whose first argument is a special type that requests creation, opening or opening or creation of the underlying resource:

```cpp
using namespace boost::interprocess;

//Create the synchronization utility. If it previously
//exists, throws an error
NamedUtility(create_only, ...)

//Open the synchronization utility. If it does not previously
//exist, it's created.
NamedUtility(open_or_create, ...)

//Open the synchronization utility. If it does not previously
//exist, throws an error.
NamedUtility(open_only, ...)
```

On the other hand the anonymous synchronization utility can only be created and the processes must synchronize using other mechanisms who creates the utility:

```
using namespace boost::interprocess;

//Create the synchronization utility.
AnonymousUtility(...)
```

Types Of Synchronization Mechanisms

Apart from its named/anonymous nature, **Boost.Interprocess** presents the following synchronization utilities:

- Mutexes (named and anonymous)

- Condition variables (named and anonymous)

- Semaphores (named and anonymous)

- Upgradable mutexes

- File locks

Mutexes

What's A Mutex?

Mutex stands for **mut**ual **ex**clusion and it's the most basic form of synchronization between processes. Mutexes guarantee that only one thread can lock a given mutex. If a code section is surrounded by a mutex locking and unlocking, it's guaranteed that only a thread at a time executes that section of code. When that thread **unlocks** the mutex, other threads can enter to that code region:

```
//The mutex has been previously constructed

lock_the_mutex();

//This code will be executed only by one thread
//at a time.

unlock_the_mutex();
```

A mutex can also be **recursive** or **non-recursive**:

- Recursive mutexes can be locked several times by the same thread. To fully unlock the mutex, the thread has to unlock the mutex the same times it has locked it.

- Non-recursive mutexes can't be locked several times by the same thread. If a mutex is locked twice by a thread, the result is undefined, it might throw an error or the thread could be blocked forever.

Mutex Operations

All the mutex types from **Boost.Interprocess** implement the following operations:

void lock()

Effects: The calling thread tries to obtain ownership of the mutex, and if another thread has ownership of the mutex, it waits until it can obtain the ownership. If a thread takes ownership of the mutex the mutex must be unlocked by the same thread. If the mutex supports recursive locking, the mutex must be unlocked the same number of times it is locked.

Throws: interprocess_exception on error.

> ### *bool try_lock()*

Effects: The calling thread tries to obtain ownership of the mutex, and if another thread has ownership of the mutex returns immediately. If the mutex supports recursive locking, the mutex must be unlocked the same number of times it is locked.

Returns: If the thread acquires ownership of the mutex, returns true, if the another thread has ownership of the mutex, returns false.

Throws: interprocess_exception on error.

> ### *bool timed_lock(const boost::posix_time::ptime &abs_time)*

Effects: The calling thread will try to obtain exclusive ownership of the mutex if it can do so in until the specified time is reached. If the mutex supports recursive locking, the mutex must be unlocked the same number of times it is locked.

Returns: If the thread acquires ownership of the mutex, returns true, if the timeout expires returns false.

Throws: interprocess_exception on error.

> ### *void unlock()*

Precondition: The thread must have exclusive ownership of the mutex.

Effects: The calling thread releases the exclusive ownership of the mutex. If the mutex supports recursive locking, the mutex must be unlocked the same number of times it is locked.

Throws: An exception derived from **interprocess_exception** on error.

Boost.Interprocess Mutex Types And Headers

Boost.Interprocess offers the following mutex types:

```
#include <boost/interprocess/sync/interprocess_mutex.hpp>
```

- interprocess_mutex: A non-recursive, anonymous mutex that can be placed in shared memory or memory mapped files.

```
#include <boost/interprocess/sync/interprocess_recursive_mutex.hpp>
```

- interprocess_recursive_mutex: A recursive, anonymous mutex that can be placed in shared memory or memory mapped files.

```
#include <boost/interprocess/sync/named_mutex.hpp>
```

- named_mutex: A non-recursive, named mutex.

```
#include <boost/interprocess/sync/named_recursive_mutex.hpp>
```

- named_recursive_mutex: A recursive, named mutex.

Scoped lock

It's very important to unlock a mutex after the process has read or written the data. This can be difficult when dealing with exceptions, so usually mutexes are used with a scoped lock, a class that can guarantee that a mutex will always be unlocked even when an exception occurs. To use a scoped lock just include:

```
#include <boost/interprocess/sync/scoped_lock.hpp>
```

Basically, a scoped lock calls **unlock()** in its destructor, and a mutex is always unlocked when an exception occurs. Scoped lock has many constructors to lock, try_lock, timed_lock a mutex or not to lock it at all.

```
using namespace boost::interprocess;

//Let's create any mutex type:
MutexType mutex;

{
   //This will lock the mutex
   scoped_lock<MutexType> lock(mutex);

   //Some code

   //The mutex will be unlocked here
}

{
   //This will try_lock the mutex
   scoped_lock<MutexType> lock(mutex, try_to_lock);

   //Check if the mutex has been successfully locked
   if(lock){
      //Some code
   }

   //If the mutex was locked it will be unlocked
}

{
   boost::posix_time::ptime abs_time = ...

   //This will timed_lock the mutex
   scoped_lock<MutexType> lock(mutex, abs_time);

   //Check if the mutex has been successfully locked
   if(lock){
      //Some code
   }

   //If the mutex was locked it will be unlocked
}
```

For more information, check the scoped_lock's reference.

Anonymous mutex example

Imagine that two processes need to write traces to a cyclic buffer built in shared memory. Each process needs to obtain exclusive access to the cyclic buffer, write the trace and continue.

To protect the cyclic buffer, we can store a process shared mutex in the cyclic buffer. Each process will lock the mutex before writing the data and will write a flag when ends writing the traces (doc_anonymous_mutex_shared_data.hpp header):

```cpp
#include <boost/interprocess/sync/interprocess_mutex.hpp>

struct shared_memory_log
{
   enum { NumItems = 100 };
   enum { LineSize = 100 };

   shared_memory_log()
      :  current_line(0)
      ,  end_a(false)
      ,  end_b(false)
   {}

   //Mutex to protect access to the queue
   boost::interprocess::interprocess_mutex mutex;

   //Items to fill
   char    items[NumItems][LineSize];
   int     current_line;
   bool    end_a;
   bool    end_b;
};
```

This is the process main process. Creates the shared memory, constructs the cyclic buffer and start writing traces:

```cpp
#include <boost/interprocess/shared_memory_object.hpp>
#include <boost/interprocess/mapped_region.hpp>
#include <boost/interprocess/sync/scoped_lock.hpp>
#include "doc_anonymous_mutex_shared_data.hpp"
#include <iostream>
#include <cstdio>

using namespace boost::interprocess;

int main ()
{
   try{
      //Remove shared memory on construction and destruction
      struct shm_remove
      {
         shm_remove() { shared_memory_object::remove("MySharedMemory"); }
         ~shm_remove(){ shared_memory_object::remove("MySharedMemory"); }
      } remover;

      //Create a shared memory object.
      shared_memory_object shm
         (create_only                //only create
         ,"MySharedMemory"           //name
         ,read_write   //read-write mode
         );

      //Set size
      shm.truncate(sizeof(shared_memory_log));

      //Map the whole shared memory in this process
      mapped_region region
         (shm                        //What to map
         ,read_write   //Map it as read-write
         );

      //Get the address of the mapped region
      void * addr       = region.get_address();

      //Construct the shared structure in memory
      shared_memory_log * data = new (addr) shared_memory_log;

      //Write some logs
      for(int i = 0; i < shared_memory_log::NumItems; ++i){
         //Lock the mutex
         scoped_lock<interprocess_mutex> lock(data->mutex);
         std::sprintf(data->items[(data->current_line++) % shared_memory_log::NumItems]
                  ,"%s_%d", "process_a", i);
         if(i == (shared_memory_log::NumItems-1))
            data->end_a = true;
         //Mutex is released here
      }

      //Wait until the other process ends
      while(1){
         scoped_lock<interprocess_mutex> lock(data->mutex);
         if(data->end_b)
            break;
      }
   }
```

```
    catch(interprocess_exception &ex){
        std::cout << ex.what() << std::endl;
        return 1;
    }
    return 0;
}
```

The second process opens the shared memory, obtains access to the cyclic buffer and starts writing traces:

```
#include <boost/interprocess/shared_memory_object.hpp>
#include <boost/interprocess/mapped_region.hpp>
#include <boost/interprocess/sync/scoped_lock.hpp>
#include "doc_anonymous_mutex_shared_data.hpp"
#include <iostream>
#include <cstdio>

using namespace boost::interprocess;

int main ()
{
    //Remove shared memory on destruction
    struct shm_remove
    {
        ~shm_remove(){ shared_memory_object::remove("MySharedMemory"); }
    } remover;

    //Open the shared memory object.
    shared_memory_object shm
        (open_only                  //only create
        ,"MySharedMemory"           //name
        ,read_write  //read-write mode
        );

    //Map the whole shared memory in this process
    mapped_region region
        (shm                        //What to map
        ,read_write //Map it as read-write
        );

    //Get the address of the mapped region
    void * addr       = region.get_address();

    //Construct the shared structure in memory
    shared_memory_log * data = static_cast<shared_memory_log*>(addr);

    //Write some logs
    for(int i = 0; i < 100; ++i){
        //Lock the mutex
        scoped_lock<interprocess_mutex> lock(data->mutex);
        std::sprintf(data->items[(data->current_line++) % shared_memory_log::NumItems]
                ,"%s_%d", "process_a", i);
        if(i == (shared_memory_log::NumItems-1))
            data->end_b = true;
        //Mutex is released here
    }

    //Wait until the other process ends
    while(1){
```

```
        scoped_lock<interprocess_mutex> lock(data->mutex);
        if(data->end_a)
            break;
    }
    return 0;
}
```

As we can see, a mutex is useful to protect data but not to notify an event to another process. For this, we need a condition variable, as we will see in the next section.

Named mutex example

Now imagine that two processes want to write a trace to a file. First they write their name, and after that they write the message. Since the operating system can interrupt a process in any moment we can mix parts of the messages of both processes, so we need a way to write the whole message to the file atomically. To achieve this, we can use a named mutex so that each process locks the mutex before writing:

```cpp
#include <boost/interprocess/sync/scoped_lock.hpp>
#include <boost/interprocess/sync/named_mutex.hpp>
#include <fstream>
#include <iostream>
#include <cstdio>


int main ()
{
   using namespace boost::interprocess;
   try{
      struct file_remove
      {
         file_remove() { std::remove("file_name"); }
         ~file_remove(){ std::remove("file_name"); }
      } file_remover;
      struct mutex_remove
      {
         mutex_remove() { named_mutex::remove("fstream_named_mutex"); }
         ~mutex_remove(){ named_mutex::remove("fstream_named_mutex"); }
      } remover;

      //Open or create the named mutex
      named_mutex mutex(open_or_create, "fstream_named_mutex");

      std::ofstream file("file_name");

      for(int i = 0; i < 10; ++i){

         //Do some operations...

         //Write to file atomically
         scoped_lock<named_mutex> lock(mutex);
         file << "Process name, ";
         file << "This is iteration #" << i;
         file << std::endl;
      }
   }
   catch(interprocess_exception &ex){
      std::cout << ex.what() << std::endl;
      return 1;
   }
   return 0;
}
```

Conditions

What's A Condition Variable?

In the previous example, a mutex is used to *lock* but we can't use it to *wait* efficiently until the condition to continue is met. A condition variable can do two things:

- **wait**: The thread is blocked until some other thread notifies that it can continue because the condition that lead to waiting has disappeared.

- **notify**: The thread sends a signal to one blocked thread or to all blocked threads to tell them that they the condition that provoked their wait has disappeared.

Waiting in a condition variable is always associated with a mutex. The mutex must be locked prior to waiting on the condition. When waiting on the condition variable, the thread unlocks the mutex and waits **atomically**.

When the thread returns from a wait function (because of a signal or a timeout, for example) the mutex object is again locked.

Boost.Interprocess Condition Types And Headers

Boost.Interprocess offers the following condition types:

```
#include <boost/interprocess/sync/interprocess_condition.hpp>
```

- interprocess_condition: An anonymous condition variable that can be placed in shared memory or memory mapped files to be used with boost::interprocess::interprocess_mutex.

```
#include <boost/interprocess/sync/named_condition.hpp>
```

- named_condition: A named condition variable to be used with named_mutex.

Named conditions are similar to anonymous conditions, but they are used in combination with named mutexes. Several times, we don't want to store synchronization objects with the synchronized data:

- We want to change the synchronization method (from interprocess to intra-process, or without any synchronization) using the same data. Storing the process-shared anonymous synchronization with the synchronized data would forbid this.

- We want to send the synchronized data through the network or any other communication method. Sending the process-shared synchronization objects wouldn't have any sense.

Anonymous condition example

Imagine that a process that writes a trace to a simple shared memory buffer that another process prints one by one. The first process writes the trace and waits until the other process prints the data. To achieve this, we can use two condition variables: the first one is used to block the sender until the second process prints the message and the second one to block the receiver until the buffer has a trace to print.

The shared memory trace buffer (doc_anonymous_condition_shared_data.hpp):

```
#include <boost/interprocess/sync/interprocess_mutex.hpp>
#include <boost/interprocess/sync/interprocess_condition.hpp>

struct trace_queue
{
   enum { LineSize = 100 };

   trace_queue()
      :  message_in(false)
   {}

   //Mutex to protect access to the queue
   boost::interprocess::interprocess_mutex      mutex;

   //Condition to wait when the queue is empty
   boost::interprocess::interprocess_condition  cond_empty;

   //Condition to wait when the queue is full
   boost::interprocess::interprocess_condition  cond_full;

   //Items to fill
   char    items[LineSize];

   //Is there any message
   bool message_in;
};
```

This is the process main process. Creates the shared memory, places there the buffer and starts writing messages one by one until it writes "last message" to indicate that there are no more messages to print:

```
trace_queue()
```

```cpp
#include <boost/interprocess/shared_memory_object.hpp>
#include <boost/interprocess/mapped_region.hpp>
#include <boost/interprocess/sync/scoped_lock.hpp>
#include <iostream>
#include <cstdio>
#include "doc_anonymous_condition_shared_data.hpp"

using namespace boost::interprocess;

int main ()
{

   //Erase previous shared memory and schedule erasure on exit
   struct shm_remove
   {
      shm_remove() { shared_memory_object::remove("MySharedMemory"); }
      ~shm_remove(){ shared_memory_object::remove("MySharedMemory"); }
   } remover;

   //Create a shared memory object.
   shared_memory_object shm
      (create_only              //only create
      ,"MySharedMemory"          //name
      ,read_write               //read-write mode
      );
   try{
      //Set size
      shm.truncate(sizeof(trace_queue));

      //Map the whole shared memory in this process
      mapped_region region
         (shm                        //What to map
         ,read_write //Map it as read-write
         );

      //Get the address of the mapped region
      void * addr       = region.get_address();

      //Construct the shared structure in memory
      trace_queue * data = new (addr) trace_queue;

      const int NumMsg = 100;

      for(int i = 0; i < NumMsg; ++i){
         scoped_lock<interprocess_mutex> lock(data->mutex);
         if(data->message_in){
            data->cond_full.wait(lock);
         }
         if(i == (NumMsg-1))
            std::sprintf(data->items, "%s", "last message");
         else
            std::sprintf(data->items, "%s_%d", "my_trace", i);

         //Notify to the other process that there is a message
         data->cond_empty.notify_one();

         //Mark message buffer as full
         data->message_in = true;
      }
   }
   catch(interprocess_exception &ex){
```

```
        std::cout << ex.what() << std::endl;
        return 1;
    }

    return 0;
}
```

The second process opens the shared memory and prints each message until the "last message" message is received:

```cpp
#include <boost/interprocess/shared_memory_object.hpp>
#include <boost/interprocess/mapped_region.hpp>
#include <boost/interprocess/sync/scoped_lock.hpp>
#include <iostream>
#include <cstring>
#include "doc_anonymous_condition_shared_data.hpp"

using namespace boost::interprocess;

int main ()
{
    //Create a shared memory object.
    shared_memory_object shm
        (open_only                  //only create
        ,"MySharedMemory"            //name
        ,read_write                  //read-write mode
        );

    try{
        //Map the whole shared memory in this process
        mapped_region region
            (shm                     //What to map
            ,read_write //Map it as read-write
            );

        //Get the address of the mapped region
        void * addr       = region.get_address();

        //Obtain a pointer to the shared structure
        trace_queue * data = static_cast<trace_queue*>(addr);

        //Print messages until the other process marks the end
        bool end_loop = false;
        do{
            scoped_lock<interprocess_mutex> lock(data->mutex);
            if(!data->message_in){
                data->cond_empty.wait(lock);
            }
            if(std::strcmp(data->items, "last message") == 0){
                end_loop = true;
            }
            else{
                //Print the message
                std::cout << data->items << std::endl;
                //Notify the other process that the buffer is empty
                data->message_in = false;
                data->cond_full.notify_one();
            }
        }
        while(!end_loop);
    }
    catch(interprocess_exception &ex){
```

```
        std::cout << ex.what() << std::endl;
        return 1;
    }

    return 0;
}
```

With condition variables, a process can block if it can't continue the work, and when the conditions to continue are met another process can wake it.

Semaphores

What's A Semaphore?

A semaphore is a synchronization mechanism between processes based in an internal count that offers two basic operations:

- **Wait**: Tests the value of the semaphore count, and waits if the value is less than or equal than 0. Otherwise, decrements the semaphore count.

- **Post**: Increments the semaphore count. If any process is blocked, one of those processes is awoken.

If the initial semaphore count is initialized to 1, a **Wait** operation is equivalent to a mutex locking and **Post** is equivalent to a mutex unlocking. This type of semaphore is known as a **binary semaphore**.

Although semaphores can be used like mutexes, they have a unique feature: unlike mutexes, a **Post** operation need not be executed by the same thread/process that executed the **Wait** operation.

Boost.Interprocess Semaphore Types And Headers

Boost.Interprocess offers the following semaphore types:

```
#include <boost/interprocess/sync/interprocess_semaphore.hpp>
```

- `interprocess_semaphore`: An anonymous semaphore that can be placed in shared memory or memory mapped files.

```
#include <boost/interprocess/sync/named_semaphore.hpp>
```

- `named_semaphore`: A named semaphore.

Anonymous semaphore example

We will implement an integer array in shared memory that will be used to transfer data from one process to another process. The first process will write some integers to the array and the process will block if the array is full.

The second process will copy the transmitted data to its own buffer, blocking if there is no new data in the buffer.

This is the shared integer array (doc_anonymous_semaphore_shared_data.hpp):

```cpp
#include <boost/interprocess/sync/interprocess_semaphore.hpp>

struct shared_memory_buffer
{
   enum { NumItems = 10 };

   shared_memory_buffer()
      : mutex(1), nempty(NumItems), nstored(0)
   {}

   //Semaphores to protect and synchronize access
   boost::interprocess::interprocess_semaphore
      mutex, nempty, nstored;

   //Items to fill
   int items[NumItems];
};
```

This is the process main process. Creates the shared memory, places there the integer array and starts integers one by one, blocking if the array is full:

```cpp
#include <boost/interprocess/shared_memory_object.hpp>
#include <boost/interprocess/mapped_region.hpp>
#include <iostream>
#include "doc_anonymous_semaphore_shared_data.hpp"

using namespace boost::interprocess;

int main ()
{
   //Remove shared memory on construction and destruction
   struct shm_remove
   {
      shm_remove() { shared_memory_object::remove("MySharedMemory"); }
      ~shm_remove(){ shared_memory_object::remove("MySharedMemory"); }
   } remover;

   //Create a shared memory object.
   shared_memory_object shm
      (create_only                 //only create
      ,"MySharedMemory"            //name
      ,read_write  //read-write mode
      );

   //Set size
   shm.truncate(sizeof(shared_memory_buffer));

   //Map the whole shared memory in this process
   mapped_region region
      (shm                        //What to map
      ,read_write //Map it as read-write
      );

   //Get the address of the mapped region
   void * addr       = region.get_address();

   //Construct the shared structure in memory
   shared_memory_buffer * data = new (addr) shared_memory_buffer;

   const int NumMsg = 100;

   //Insert data in the array
```

```
   for(int i = 0; i < NumMsg; ++i){
      data->nempty.wait();
      data->mutex.wait();
      data->items[i % shared_memory_buffer::NumItems] = i;
      data->mutex.post();
      data->nstored.post();
   }

   return 0;
}
```

The second process opens the shared memory and copies the received integers to it's own buffer:

```
#include <boost/interprocess/shared_memory_object.hpp>
#include <boost/interprocess/mapped_region.hpp>
#include <iostream>
#include "doc_anonymous_semaphore_shared_data.hpp"

using namespace boost::interprocess;

int main ()
{
   //Remove shared memory on destruction
   struct shm_remove
   {
      ~shm_remove(){ shared_memory_object::remove("MySharedMemory"); }
   } remover;

   //Create a shared memory object.
   shared_memory_object shm
      (open_only                    //only create
      ,"MySharedMemory"             //name
      ,read_write  //read-write mode
      );

   //Map the whole shared memory in this process
   mapped_region region
      (shm                     //What to map
      ,read_write //Map it as read-write
      );

   //Get the address of the mapped region
   void * addr       = region.get_address();

   //Obtain the shared structure
   shared_memory_buffer * data = static_cast<shared_memory_buffer*>(addr);

   const int NumMsg = 100;

   int extracted_data [NumMsg];

   //Extract the data
   for(int i = 0; i < NumMsg; ++i){
      data->nstored.wait();
      data->mutex.wait();
```

```
            extracted_data[i] = data->items[i % shared_memory_buffer::NumItems];
            data->mutex.post();
            data->nempty.post();
        }
        return 0;
    }
```

The same interprocess communication can be achieved with a condition variables and mutexes, but for several synchronization patterns, a semaphore is more efficient than a mutex/condition combination.

Upgradable Mutexes

What's An Upgradable Mutex?

An upgradable mutex is a special mutex that offers more locking possibilities than a normal mutex. Sometimes, we can distinguish between **reading** the data and **modifying** the data. If just some threads need to modify the data, and a plain mutex is used to protect the data from concurrent access, concurrency is pretty limited: two threads that only read the data will be serialized instead of being executed concurrently.

If we allow concurrent access to threads that just read the data but we avoid concurrent access between threads that read and modify or between threads that modify, we can increase performance. This is specially true in applications where data reading is more common than data modification and the synchronized data reading code needs some time to execute. With an upgradable mutex we can acquire 3 lock types:

- **Exclusive lock**: Similar to a plain mutex. If a thread acquires an exclusive lock, no other thread can acquire any lock (exclusive or other) until the exclusive lock is released. If any thread has a sharable or upgradable lock a thread trying to acquire an exclusive lock will block. This lock will be acquired by threads that will modify the data.

- **Sharable lock**: If a thread acquires a sharable lock, other threads can acquire a sharable lock or an upgradable lock. If any thread has acquired the exclusive lock a thread trying to acquire a sharable lock will block. This locking is executed by threads that just need to read the data.

- **Upgradable lock**: Acquiring an upgradable lock is similar to acquiring a **privileged sharable lock**. If a thread acquires an upgradable lock, other threads can acquire a sharable lock. If any thread has acquired the exclusive or upgradable lock a thread trying to acquire an upgradable lock will block. A thread that has acquired an upgradable lock, is guaranteed to be able to acquire atomically an exclusive lock when other threads that have acquired a sharable lock release it. This is used for a thread that **maybe** needs to modify the data, but usually just needs to read the data. This thread acquires the upgradable lock and other threads can acquire the sharable lock. If the upgradable thread reads the data and it has to modify it, the thread can be promoted to acquire the exclusive lock: when all sharable threads have released the sharable lock, the upgradable lock is atomically promoted to an exclusive lock. The newly promoted thread can modify the data and it can be sure that no other thread has modified it while doing the transition. **Only 1 thread can acquire the upgradable (privileged reader) lock**.

To sum up:

Table 5. Locking Possibilities

If a thread has acquired the...	Other threads can acquire...
Sharable lock	many sharable locks and 1 upgradable lock
Upgradable lock	many sharable locks
Exclusive lock	no locks

A thread that has acquired a lock can try to acquire another lock type atomically. All lock transitions are not guaranteed to succeed. Even if a transition is guaranteed to succeed, some transitions will block the thread waiting until other threads release the sharable

locks. **Atomically** means that no other thread will acquire an Upgradable or Exclusive lock in the transition, **so data is guaranteed to remain unchanged**:

Table 6. Transition Possibilities

If a thread has acquired the...	It can atomically release the previous lock and...
Sharable lock	try to obtain (not guaranteed) immediately the Exclusive lock if no other thread has exclusive or upgrable lock
Sharable lock	try to obtain (not guaranteed) immediately the Upgradable lock if no other thread has exclusive or upgrable lock
Upgradable lock	obtain the Exclusive lock when all sharable locks are released
Upgradable lock	obtain the Sharable lock immediately
Exclusive lock	obtain the Upgradable lock immediately
Exclusive lock	obtain the Sharable lock immediately

As we can see, an upgradable mutex is a powerful synchronization utility that can improve the concurrency. However, if most of the time we have to modify the data, or the synchronized code section is very short, it's more efficient to use a plain mutex, since it has less overhead. Upgradable lock shines when the synchronized code section is bigger and there are more readers than modifiers.

Upgradable Mutex Operations

All the upgradable mutex types from **Boost.Interprocess** implement the following operations:

Exclusive Locking

> *void lock()*

Effects: The calling thread tries to obtain exclusive ownership of the mutex, and if another thread has exclusive, sharable or upgradable ownership of the mutex, it waits until it can obtain the ownership.

Throws: interprocess_exception on error.

> *bool try_lock()*

Effects: The calling thread tries to acquire exclusive ownership of the mutex without waiting. If no other thread has exclusive, sharable or upgradable ownership of the mutex this succeeds.

Returns: If it can acquire exclusive ownership immediately returns true. If it has to wait, returns false.

Throws: interprocess_exception on error.

> *bool timed_lock(const boost::posix_time::ptime &abs_time)*

Effects: The calling thread tries to acquire exclusive ownership of the mutex waiting if necessary until no other thread has has exclusive, sharable or upgradable ownership of the mutex or abs_time is reached.

Returns: If acquires exclusive ownership, returns true. Otherwise returns false.

Throws: interprocess_exception on error.

> *void unlock()*

Precondition: The thread must have exclusive ownership of the mutex.

Effects: The calling thread releases the exclusive ownership of the mutex.

Throws: An exception derived from **interprocess_exception** on error.

Sharable Locking

> *void lock_sharable()*

Effects: The calling thread tries to obtain sharable ownership of the mutex, and if another thread has exclusive or upgradable ownership of the mutex, waits until it can obtain the ownership.

Throws: interprocess_exception on error.

> *bool try_lock_sharable()*

Effects: The calling thread tries to acquire sharable ownership of the mutex without waiting. If no other thread has has exclusive or upgradable ownership of the mutex this succeeds.

Returns: If it can acquire sharable ownership immediately returns true. If it has to wait, returns false.

Throws: interprocess_exception on error.

> *bool timed_lock_sharable(const boost::posix_time::ptime &abs_time)*

Effects: The calling thread tries to acquire sharable ownership of the mutex waiting if necessary until no other thread has has exclusive or upgradable ownership of the mutex or abs_time is reached.

Returns: If acquires sharable ownership, returns true. Otherwise returns false.

Throws: interprocess_exception on error.

> *void unlock_sharable()*

Precondition: The thread must have sharable ownership of the mutex.

Effects: The calling thread releases the sharable ownership of the mutex.

Throws: An exception derived from **interprocess_exception** on error.

Upgradable Locking

> *void lock_upgradable()*

Effects: The calling thread tries to obtain upgradable ownership of the mutex, and if another thread has exclusive or upgradable ownership of the mutex, waits until it can obtain the ownership.

Throws: interprocess_exception on error.

> *bool try_lock_upgradable()*

Effects: The calling thread tries to acquire upgradable ownership of the mutex without waiting. If no other thread has has exclusive or upgradable ownership of the mutex this succeeds.

Returns: If it can acquire upgradable ownership immediately returns true. If it has to wait, returns false.

Throws: interprocess_exception on error.

> *bool timed_lock_upgradable(const boost::posix_time::ptime &abs_time)*

Effects: The calling thread tries to acquire upgradable ownership of the mutex waiting if necessary until no other thread has has exclusive or upgradable ownership of the mutex or abs_time is reached.

Returns: If acquires upgradable ownership, returns true. Otherwise returns false.

Throws: interprocess_exception on error.

> *void unlock_upgradable()*

Precondition: The thread must have upgradable ownership of the mutex.

Effects: The calling thread releases the upgradable ownership of the mutex.

Throws: An exception derived from **interprocess_exception** on error.

Demotions

> *void unlock_and_lock_upgradable()*

Precondition: The thread must have exclusive ownership of the mutex.

Effects: The thread atomically releases exclusive ownership and acquires upgradable ownership. This operation is non-blocking.

Throws: An exception derived from **interprocess_exception** on error.

> *void unlock_and_lock_sharable()*

Precondition: The thread must have exclusive ownership of the mutex.

Effects: The thread atomically releases exclusive ownership and acquires sharable ownership. This operation is non-blocking.

Throws: An exception derived from **interprocess_exception** on error.

> *void unlock_upgradable_and_lock_sharable()*

Precondition: The thread must have upgradable ownership of the mutex.

Effects: The thread atomically releases upgradable ownership and acquires sharable ownership. This operation is non-blocking.

Throws: An exception derived from **interprocess_exception** on error.

Promotions

> *void unlock_upgradable_and_lock()*

Precondition: The thread must have upgradable ownership of the mutex.

Effects: The thread atomically releases upgradable ownership and acquires exclusive ownership. This operation will block until all threads with sharable ownership release it.

Throws: An exception derived from **interprocess_exception** on error.

> *bool try_unlock_upgradable_and_lock()*

Precondition: The thread must have upgradable ownership of the mutex.

Effects: The thread atomically releases upgradable ownership and tries to acquire exclusive ownership. This operation will fail if there are threads with sharable ownership, but it will maintain upgradable ownership.

Returns: If acquires exclusive ownership, returns true. Otherwise returns false.

Throws: An exception derived from **interprocess_exception** on error.

> *bool timed_unlock_upgradable_and_lock(const boost::posix_time::ptime &abs_time)*

Precondition: The thread must have upgradable ownership of the mutex.

Effects: The thread atomically releases upgradable ownership and tries to acquire exclusive ownership, waiting if necessary until abs_time. This operation will fail if there are threads with sharable ownership or timeout reaches, but it will maintain upgradable ownership.

Returns: If acquires exclusive ownership, returns true. Otherwise returns false.

Throws: An exception derived from **interprocess_exception** on error.

> *bool try_unlock_sharable_and_lock()*

Precondition: The thread must have sharable ownership of the mutex.

Effects: The thread atomically releases sharable ownership and tries to acquire exclusive ownership. This operation will fail if there are threads with sharable or upgradable ownership, but it will maintain sharable ownership.

Returns: If acquires exclusive ownership, returns true. Otherwise returns false.

Throws: An exception derived from **interprocess_exception** on error.

---

bool try_unlock_sharable_and_lock_upgradable()

Precondition: The thread must have sharable ownership of the mutex.

Effects: The thread atomically releases sharable ownership and tries to acquire upgradable ownership. This operation will fail if there are threads with sharable or upgradable ownership, but it will maintain sharable ownership.

Returns: If acquires upgradable ownership, returns true. Otherwise returns false.

Throws: An exception derived from **interprocess_exception** on error.

Boost.Interprocess Upgradable Mutex Types And Headers

Boost.Interprocess offers the following upgradable mutex types:

```
#include <boost/interprocess/sync/interprocess_upgradable_mutex.hpp>
```

- `interprocess_upgradable_mutex`: A non-recursive, anonymous upgradable mutex that can be placed in shared memory or memory mapped files.

```
#include <boost/interprocess/sync/named_upgradable_mutex.hpp>
```

- `named_upgradable_mutex`: A non-recursive, named upgradable mutex.

Sharable Lock And Upgradable Lock

As with plain mutexes, it's important to release the acquired lock even in the presence of exceptions. **Boost.Interprocess** mutexes are best used with the `scoped_lock` utility, and this class only offers exclusive locking.

As we have sharable locking and upgradable locking with upgradable mutexes, we have two new utilities: `sharable_lock` and `upgradable_lock`. Both classes are similar to `scoped_lock` but `sharable_lock` acquires the sharable lock in the constructor and `upgradable_lock` acquires the upgradable lock in the constructor.

These two utilities can be use with any synchronization object that offers the needed operations. For example, a user defined mutex type with no upgradable locking features can use `sharable_lock` if the synchronization object offers **lock_sharable()** and **unlock_sharable()** operations:

Sharable Lock And Upgradable Lock Headers

```
#include <boost/interprocess/sync/sharable_lock.hpp>
```

```
#include <boost/interprocess/sync/upgradable_lock.hpp>
```

`sharable_lock` calls **unlock_sharable()** in its destructor, and `upgradable_lock` calls **unlock_upgradable()** in its destructor, so the upgradable mutex is always unlocked when an exception occurs. Scoped lock has many constructors to lock, try_lock, timed_lock a mutex or not to lock it at all.

```cpp
using namespace boost::interprocess;

//Let's create any mutex type:
MutexType mutex;

{
   //This will call lock_sharable()
   sharable_lock<MutexType> lock(mutex);

   //Some code

   //The mutex will be unlocked here
}

{
   //This won't lock the mutex()
   sharable_lock<MutexType> lock(mutex, defer_lock);

   //Lock it on demand. This will call lock_sharable()
   lock.lock();

   //Some code

   //The mutex will be unlocked here
}

{
   //This will call try_lock_sharable()
   sharable_lock<MutexType> lock(mutex, try_to_lock);

   //Check if the mutex has been successfully locked
   if(lock){
      //Some code
   }
   //If the mutex was locked it will be unlocked
}

{
   boost::posix_time::ptime abs_time = ...

   //This will call timed_lock_sharable()
   scoped_lock<MutexType> lock(mutex, abs_time);

   //Check if the mutex has been successfully locked
   if(lock){
      //Some code
   }
   //If the mutex was locked it will be unlocked
}

{
   //This will call lock_upgradable()
   upgradable_lock<MutexType> lock(mutex);

   //Some code

   //The mutex will be unlocked here
}

{
   //This won't lock the mutex()
   upgradable_lock<MutexType> lock(mutex, defer_lock);
```

```
   //Lock it on demand. This will call lock_upgradable()
   lock.lock();

   //Some code

   //The mutex will be unlocked here
}

{
   //This will call try_lock_upgradable()
   upgradable_lock<MutexType> lock(mutex, try_to_lock);

   //Check if the mutex has been successfully locked
   if(lock){
      //Some code
   }
   //If the mutex was locked it will be unlocked
}

{
   boost::posix_time::ptime abs_time = ...

   //This will call timed_lock_upgradable()
   scoped_lock<MutexType> lock(mutex, abs_time);

   //Check if the mutex has been successfully locked
   if(lock){
      //Some code
   }
   //If the mutex was locked it will be unlocked
}
```

upgradable_lock and sharable_lock offer more features and operations, see their reference for more informations

Lock Transfers Through Move Semantics

Interprocess uses its own move semantics emulation code for compilers that don't support rvalues references. This is a temporary solution until a Boost move semantics library is accepted.

Scoped locks and similar utilities offer simple resource management possibilities, but with advanced mutex types like upgradable mutexes, there are operations where an acquired lock type is released and another lock type is acquired atomically. This is implemented by upgradable mutex operations like unlock_and_lock_sharable().

These operations can be managed more effectively using **lock transfer operations**. A lock transfer operations explicitly indicates that a mutex owned by a lock is transferred to another lock executing atomic unlocking plus locking operations.

Simple Lock Transfer

Imagine that a thread modifies some data in the beginning but after that, it has to just read it in a long time. The code can acquire the exclusive lock, modify the data and atomically release the exclusive lock and acquire the sharable lock. With these sequence we guarantee that no other thread can modify the data in the transition and that more readers can acquire sharable lock, increasing concurrency. Without lock transfer operations, this would be coded like this:

```
using boost::interprocess;
interprocess_upgradable_mutex mutex;

//Acquire exclusive lock
mutex.lock();

//Modify data

//Atomically release exclusive lock and acquire sharable lock.
//More threads can acquire the sharable lock and read the data.
mutex.unlock_and_lock_sharable();

//Read data

//Explicit unlocking
mutex.unlock_sharable();
```

This can be simple, but in the presence of exceptions, it's complicated to know what type of lock the mutex had when the exception was thrown and what function we should call to unlock it:

```
try{
    //Mutex operations
}
catch(...){
    //What should we call? "unlock()" or "unlock_sharable()"
    //Is the mutex locked?
}
```

We can use **lock transfer** to simplify all this management:

```
using boost::interprocess;
interprocess_upgradable_mutex mutex;

//Acquire exclusive lock
scoped_lock s_lock(mutex);

//Modify data

//Atomically release exclusive lock and acquire sharable lock.
//More threads can acquire the sharable lock and read the data.
sharable_lock(move(s_lock));

//Read data

//The lock is automatically unlocked calling the appropriate unlock
//function even in the presence of exceptions.
//If the mutex was not locked, no function is called.
```

As we can see, even if an exception is thrown at any moment, the mutex will be automatically unlocked calling the appropriate un-lock() or unlock_sharable() method.

Lock Transfer Summary

There are many lock transfer operations that we can classify according to the operations presented in the upgradable mutex operations:

- **Guaranteed to succeed and non-blocking:** Any transition from a more restrictive lock to a less restrictive one. Scoped -> Upgradable, Scoped -> Sharable, Upgradable -> Sharable.

- **Not guaranteed to succeed:** The operation might succeed if no one has acquired the upgradable or exclusive lock: Sharable -> Exclusive. This operation is a try operation.

- **Guaranteed to succeed if using an infinite waiting:** Any transition that will succeed but needs to wait until all Sharable locks are released: Upgradable -> Scoped. Since this is a blocking operation, we can also choose not to wait infinitely and just try or wait until a timeout is reached.

Transfers To Scoped Lock

Transfers to `scoped_lock` are guaranteed to succeed only from an `upgradable_lock` and only if a blocking operation is requested, due to the fact that this operation needs to wait until all sharable locks are released. The user can also use "try" or "timed" transfer to avoid infinite locking, but succeed is not guaranteed.

A conversion from a `sharable_lock` is never guaranteed and thus, only a try operation is permitted:

```
//Conversions to scoped_lock
{
   upgradable_lock<Mutex>  u_lock(mut);
   //This calls unlock_upgradable_and_lock()
   scoped_lock<Mutex>      e_lock(move(u_lock));
}
{
   upgradable_lock<Mutex>  u_lock(mut);
   //This calls try_unlock_upgradable_and_lock()
   scoped_lock<Mutex>      e_lock(move(u_lock, try_to_lock));
}
{
   boost::posix_time::ptime t = test::delay(100);
   upgradable_lock<Mutex>  u_lock(mut);
   //This calls timed_unlock_upgradable_and_lock()
   scoped_lock<Mutex>      e_lock(move(u_lock));
}
{
   sharable_lock<Mutex>    s_lock(mut);
   //This calls try_unlock_sharable_and_lock()
   scoped_lock<Mutex>      e_lock(move(s_lock, try_to_lock));
}
```

Transfers To Upgradable Lock

A transfer to an `upgradable_lock` is guaranteed to succeed only from a `scoped_lock` since scoped locking is a more restrictive locking than an upgradable locking. This operation is also non-blocking.

A transfer from a `sharable_lock` is not guaranteed and only a "try" operation is permitted:

```
//Conversions to upgradable
{
   sharable_lock<Mutex>    s_lock(mut);
   //This calls try_unlock_sharable_and_lock_upgradable()
   upgradable_lock<Mutex>  u_lock(move(s_lock, try_to_lock));
}
{
   scoped_lock<Mutex>      e_lock(mut);
   //This calls unlock_and_lock_upgradable()
   upgradable_lock<Mutex>  u_lock(move(e_lock));
}
```

Transfers To Sharable Lock

All transfers to a `sharable_lock` are guaranteed to succeed since both `upgradable_lock` and `scoped_lock` are more restrictive than `sharable_lock`. These operations are also non-blocking:

```
//Conversions to sharable_lock
{
   upgradable_lock<Mutex>  u_lock(mut);
   //This calls unlock_upgradable_and_lock_sharable()
   sharable_lock<Mutex>    s_lock(move(u_lock));
}
{
   scoped_lock<Mutex>      e_lock(mut);
   //This calls unlock_and_lock_sharable()
   sharable_lock<Mutex>    s_lock(move(e_lock));
}
```

Transferring Unlocked Locks

In the previous examples, the mutex used in the transfer operation was previously locked:

```
   Mutex mut;

   //This calls mut.lock()
   scoped_lock<Mutex>      e_lock(mut);

   //This calls unlock_and_lock_sharable()
   sharable_lock<Mutex>    s_lock(move(e_lock));
}
```

but it's possible to execute the transfer with an unlocked source, due to explicit unlocking, a try, timed or a `defer_lock` constructor:

```
//These operations can leave the mutex unlocked!

{
   //Try might fail
   scoped_lock<Mutex>      e_lock(mut, try_to_lock);
   sharable_lock<Mutex>    s_lock(move(e_lock));
}
{
   //Timed operation might fail
   scoped_lock<Mutex>      e_lock(mut, time);
   sharable_lock<Mutex>    s_lock(move(e_lock));
}
{
   //Avoid mutex locking
   scoped_lock<Mutex>      e_lock(mut, defer_lock);
   sharable_lock<Mutex>    s_lock(move(e_lock));
}
{
   //Explicitly call unlock
   scoped_lock<Mutex>      e_lock(mut);
   e_lock.unlock();
   //Mutex was explicitly unlocked
   sharable_lock<Mutex>    s_lock(move(e_lock));
}
```

If the source mutex was not locked:

• The target lock does not execute the atomic `unlock_xxx_and_lock_xxx` operation.

• The target lock is also unlocked.

• The source lock is released() and the ownership of the mutex is transferred to the target.

```
{
   scoped_lock<Mutex>       e_lock(mut, defer_lock);
   sharable_lock<Mutex>     s_lock(move(e_lock));

   //Assertions
   assert(e_lock.mutex() == 0);
   assert(s_lock.mutex() != 0);
   assert(e_lock.owns()  == false);
}
```

Transfer Failures

When executing a lock transfer, the operation can fail:

• The executed atomic mutex unlock plus lock function might throw.

• The executed atomic function might be a "try" or "timed" function that can fail.

In the first case, the mutex ownership is not transferred and the source lock's destructor will unlock the mutex:

```
{
   scoped_lock<Mutex>       e_lock(mut, defer_lock);

   //This operations throws because
   //"unlock_and_lock_sharable()" throws!!!
   sharable_lock<Mutex>     s_lock(move(e_lock));

   //Some code ...

   //e_lock's destructor will call "unlock()"
}
```

In the second case, if an internal "try" or "timed" operation fails (returns "false") then the mutex ownership is **not** transferred, the source lock is unchanged and the target lock's state will the same as a default construction:

```
{
   sharable_lock<Mutex>     s_lock(mut);

   //Internal "try_unlock_sharable_and_lock_upgradable()" returns false
   upgradable_lock<Mutex>  u_lock(move(s_lock, try_to_lock));

   assert(s_lock.mutex() == &mut);
   assert(s_lock.owns()  == true);
   assert(u_lock.mutex() == 0);
   assert(u_lock.owns()  == false);

   //u_lock's destructor does nothing
   //s_lock's destructor calls "unlock()"
}
```

File Locks

What's A File Lock?

A file lock is an interprocess synchronization mechanism to protect concurrent writes and reads to files using a mutex *embedded* in the file. This *embedded mutex* has sharable and exclusive locking capabilities. With a file lock, an existing file can be used as a mutex without the need of creating additional synchronization objects to control concurrent file reads or writes.

Generally speaking, we can have two file locking capabilities:

- **Advisory locking:** The operating system kernel maintains a list of files that have been locked. But does not prevent writing to those files even if a process has acquired a sharable lock or does not prevent reading from the file when a process has acquired the exclusive lock. Any process can ignore an advisory lock. This means that advisory locks are for **cooperating** processes, processes that can trust each other. This is similar to a mutex protecting data in a shared memory segment: any process connected to that memory can overwrite the data but **cooperative** processes use mutexes to protect the data first acquiring the mutex lock.

- **Mandatory locking:** The OS kernel checks every read and write request to verify that the operation can be performed according to the acquired lock. Reads and writes block until the lock is released.

Boost.Interprocess implements **advisory blocking** because of portability reasons. This means that every process accessing to a file concurrently, must cooperate using file locks to synchronize the access.

In some systems file locking can be even further refined, leading to **record locking**, where a user can specify a **byte range** within the file where the lock is applied. This allows concurrent write access by several processes if they need to access a different byte range in the file. **Boost.Interprocess** does **not** offer record locking for the moment, but might offer it in the future. To use a file lock just include:

```
#include <boost/interprocess/sync/file_lock.hpp>
```

A file locking is a class that has **process lifetime**. This means that if a process holding a file lock ends or crashes, the operating system will automatically unlock it. This feature is very useful in some situations where we want to assure automatic unlocking even when the process crashes and avoid leaving blocked resources in the system. A file lock is constructed using the name of the file as an argument:

```
#include <boost/interprocess/sync/file_lock.hpp>

int main()
{
   //This throws if the file does not exist or it can't
   //open it with read-write access!
   boost::interprocess::file_lock flock("my_file");
   return 0;
}
```

File Locking Operations

File locking has normal mutex operations plus sharable locking capabilities. This means that we can have multiple readers holding the sharable lock and writers holding the exclusive lock waiting until the readers end their job.

However, file locking does **not** support upgradable locking or promotion or demotion (lock transfers), so it's more limited than an upgradable lock. These are the operations:

> *void lock()*

Effects: The calling thread tries to obtain exclusive ownership of the file lock, and if another thread has exclusive or sharable ownership of the mutex, it waits until it can obtain the ownership.

Throws: interprocess_exception on error.

> *bool try_lock()*

Effects: The calling thread tries to acquire exclusive ownership of the file lock without waiting. If no other thread has exclusive or sharable ownership of the file lock, this succeeds.

Returns: If it can acquire exclusive ownership immediately returns true. If it has to wait, returns false.

Throws: interprocess_exception on error.

> *bool timed_lock(const boost::posix_time::ptime &abs_time)*

Effects: The calling thread tries to acquire exclusive ownership of the file lock waiting if necessary until no other thread has has exclusive or sharable ownership of the file lock or abs_time is reached.

Returns: If acquires exclusive ownership, returns true. Otherwise returns false.

Throws: interprocess_exception on error.

> *void unlock()*

Precondition: The thread must have exclusive ownership of the file lock.

Effects: The calling thread releases the exclusive ownership of the file lock.

Throws: An exception derived from **interprocess_exception** on error.

> *void lock_sharable()*

Effects: The calling thread tries to obtain sharable ownership of the file lock, and if another thread has exclusive ownership of the file lock, waits until it can obtain the ownership.

Throws: interprocess_exception on error.

> *bool try_lock_sharable()*

Effects: The calling thread tries to acquire sharable ownership of the file lock without waiting. If no other thread has has exclusive ownership of the file lock, this succeeds.

Returns: If it can acquire sharable ownership immediately returns true. If it has to wait, returns false.

Throws: interprocess_exception on error.

> *bool timed_lock_sharable(const boost::posix_time::ptime &abs_time)*

Effects: The calling thread tries to acquire sharable ownership of the file lock waiting if necessary until no other thread has has exclusive ownership of the file lock or abs_time is reached.

Returns: If acquires sharable ownership, returns true. Otherwise returns false.

Throws: interprocess_exception on error.

> *void unlock_sharable()*

Precondition: The thread must have sharable ownership of the file lock.

Effects: The calling thread releases the sharable ownership of the file lock.

Throws: An exception derived from **interprocess_exception** on error.

For more file locking methods, please `file_lock reference`.

Scoped Lock and Sharable Lock With File Locking

`scoped_lock` and `sharable_lock` can be used to make file locking easier in the presence of exceptions, just like with mutexes:

```cpp
#include <boost/interprocess/sync/file_lock.hpp>
#include <boost/interprocess/sync/sharable_lock.hpp>
//...

using namespace boost::interprocess;
//This process reads the file
//      ...
//Open the file lock
file_lock f_lock("my_file");

{
   //Construct a sharable lock with the filel lock.
   //This will call "f_lock.sharable_lock()".
   sharable_lock<file_lock> sh_lock(f_lock);

   //Now read the file...

   //The sharable lock is automatically released by
   //sh_lock's destructor
}
```

```cpp
#include <boost/interprocess/sync/file_lock.hpp>
#include <boost/interprocess/sync/scoped_lock.hpp>

//...

using namespace boost::interprocess;
//This process writes the file
//      ...
//Open the file lock
file_lock f_lock("my_file");

{
   //Construct a sharable lock with the filel lock.
   //This will call "f_lock.lock()".
   scoped_lock<file_lock> e_lock(f_lock);

   //Now write the file...

   //The exclusive lock is automatically released by
   //e_lock's destructor
}
```

However, lock transfers are only allowed between same type of locks, that is, from a sharable lock to another sharable lock or from a scoped lock to another scoped lock. A transfer from a scoped lock to a sharable lock is not allowed, because `file_lock` has no lock promotion or demotion functions like `unlock_and_lock_sharable()`. This will produce a compilation error:

```
//Open the file lock
file_lock f_lock("my_file");

scoped_lock<file_lock> e_lock(f_lock);

//Compilation error, f_lock has no "unlock_and_lock_sharable()" member!
sharable_lock<file_lock> e_lock(move(f_lock));
```

Caution: Synchronization limitations

If you plan to use file locks just like named mutexes, be careful, because portable file locks have synchronization limitations, mainly because different implementations (POSIX, Windows) offer different guarantees. Interprocess file locks have the following limitations:

• It's unspecified if a `file_lock` synchronizes **two threads from the same process**.

• It's unspecified if a process can use two `file_lock` objects pointing to the same file.

The first limitation comes mainly from POSIX, since a file handle is a per-process attribute and not a per-thread attribute. This means that if a thread uses a `file_lock` object to lock a file, other threads will see the file as locked. Windows file locking mechanism, on the other hand, offer thread-synchronization guarantees so a thread trying to lock the already locked file, would block.

The second limitation comes from the fact that file locking synchronization state is tied with a single file descriptor in Windows. This means that if two `file_lock` objects are created pointing to the same file, no synchronization is guaranteed. In POSIX, when two file descriptors are used to lock a file if a descriptor is closed, all file locks set by the calling process are cleared.

To sum up, if you plan to use portable file locking in your processes, use the following restrictions:

• **For each file, use a single `file_lock` object per process.**

• **Use the same thread to lock and unlock a file.**

• If you are using a std::fstream/native file handle to write to the file while using file locks on that file, **don't close the file before releasing all the locks of the file.**

Be Careful With Iostream Writing

As we've seen file locking can be useful to synchronize two processes reading and writing to a file, but **make sure data is written to the file** before unlocking the file lock. Take in care that iostream classes do some kind of buffering, so if you want to make sure that other processes can see the data you've written, you have the following alternatives:

• Use native file functions (read()/write() in Unix systems and ReadFile/WriteFile in Windows systems) instead of iostream.

• Flush data before unlocking the file lock in writers using `fflush` if you are using standard C functions or the `flush()` member function when using C++ iostreams.

```
//...

using namespace boost::interprocess;
//This process writes the file
//     ...
//Open the file lock
fstream file("my_file")
file_lock f_lock("my_file");

{
    scoped_lock<file_lock> e_lock(f_lock);

    //Now write the file...

    //Flush data before unlocking the exclusive lock
    file.flush();
}
```

Message Queue

What's A Message Queue?

A message queue is similar to a list of messages. Threads can put messages in the queue and they can also remove messages from the queue. Each message can have also a **priority** so that higher priority messages are read before lower priority messages. Each message has some attributes:

- A priority.

- The length of the message.

- The data (if length is bigger than 0).

A thread can send a message to or receive a message from the message queue using 3 methods:

- **Blocking**: If the message queue is full when sending or the message queue is empty when receiving, the thread is blocked until there is room for a new message or there is a new message.

- **Try**: If the message queue is full when sending or the message queue is empty when receiving, the thread returns immediately with an error.

- **Timed**: If the message queue is full when sending or the message queue is empty when receiving, the thread retries the operation until succeeds (returning successful state) or a timeout is reached (returning a failure).

A message queue **just copies raw bytes between processes** and does not send objects. This means that if we want to send an object using a message queue **the object must be binary serializable**. For example, we can send integers between processes but **not** a std::string. You should use **Boost.Serialization** or use advanced **Boost.Interprocess** mechanisms to send complex data between processes.

The **Boost.Interprocess** message queue is a named interprocess communication: the message queue is created with a name and it's opened with a name, just like a file. When creating a message queue, the user must specify the maximum message size and the maximum message number that the message queue can store. These parameters will define the resources (for example the size of the shared memory used to implement the message queue if shared memory is used).

```
using boost::interprocess;
//Create a message_queue. If the queue
//exists throws an exception
message_queue mq
   (create_only           //only create
   ,"message_queue"       //name
   ,100                   //max message number
   ,100                   //max message size
   );
```

```
using boost::interprocess;
//Creates or opens a message_queue. If the queue
//does not exist creates it, otherwise opens it.
//Message number and size are ignored if the queue
//is opened
message_queue mq
   (open_or_create        //open or create
   ,"message_queue"       //name
   ,100                   //max message number
   ,100                   //max message size
   );
```

```
using boost::interprocess;
//Opens a message_queue. If the queue
//does not exist throws an exception.
message_queue mq
   (open_only             //only open
   ,"message_queue"       //name
   );
```

The message queue is explicitly removed calling the static `remove` function:

```
using boost::interprocess;
message_queue::remove("message_queue");
```

The function can fail if the message queue is still being used by any process.

Using a message queue

To use a message queue you must include the following header:

```
#include <boost/interprocess/ipc/message_queue.hpp>
```

In the following example, the first process creates the message queue, and writes an array of integers on it. The other process just reads the array and checks that the sequence number is correct. This is the first process:

```cpp
#include <boost/interprocess/ipc/message_queue.hpp>
#include <iostream>
#include <vector>

using namespace boost::interprocess;

int main ()
{
   try{
      //Erase previous message queue
      message_queue::remove("message_queue");

      //Create a message_queue.
      message_queue mq
         (create_only               //only create
         ,"message_queue"           //name
         ,100                       //max message number
         ,sizeof(int)              //max message size
         );

      //Send 100 numbers
      for(int i = 0; i < 100; ++i){
         mq.send(&i, sizeof(i), 0);
      }
   }
   catch(interprocess_exception &ex){
      std::cout << ex.what() << std::endl;
      return 1;
   }

   return 0;
}
```

This is the second process:

```cpp
#include <boost/interprocess/ipc/message_queue.hpp>
#include <iostream>
#include <vector>

using namespace boost::interprocess;

int main ()
{
   try{
      //Open a message queue.
      message_queue mq
         (open_only          //only create
         ,"message_queue"    //name
         );

      unsigned int priority;
      std::size_t recvd_size;

      //Receive 100 numbers
      for(int i = 0; i < 100; ++i){
         int number;
         mq.receive(&number, sizeof(number), recvd_size, priority);
         if(number != i || recvd_size != sizeof(number))
            return 1;
      }
   }
   catch(interprocess_exception &ex){
      message_queue::remove("message_queue");
      std::cout << ex.what() << std::endl;
      return 1;
   }
   message_queue::remove("message_queue");
   return 0;
}
```

To know more about this class and all its operations, please see the `message_queue` class reference.

Managed Memory Segments

Making Interprocess Data Communication Easy

Introduction

As we have seen, **Boost.Interprocess** offers some basic classes to create shared memory objects and file mappings and map those mappable classes to the process' address space.

However, managing those memory segments is not not easy for non-trivial tasks. A mapped region is a fixed-length memory buffer and creating and destroying objects of any type dynamically, requires a lot of work, since it would require programming a memory management algorithm to allocate portions of that segment. Many times, we also want to associate names to objects created in shared memory, so all the processes can find the object using the name.

Boost.Interprocess offers 4 managed memory segment classes:

• To manage a shared memory mapped region (**basic_managed_shared_memory** class).

• To manage a memory mapped file (**basic_managed_mapped_file**).

• To manage a heap allocated (`operator new`) memory buffer (**basic_managed_heap_memory** class).

• To manage a user provided fixed size buffer (**basic_managed_external_buffer** class).

The first two classes manage memory segments that can be shared between processes. The third is useful to create complex databases to be sent though other mechanisms like message queues to other processes. The fourth class can manage any fixed size memory buffer. The first two classes will be explained in the next two sections. **basic_managed_heap_memory** and **basic_managed_external_buffer** will be explained later.

The most important services of a managed memory segment are:

- Dynamic allocation of portions of a memory the segment.

- Construction of C++ objects in the memory segment. These objects can be anonymous or we can associate a name to them.

- Searching capabilities for named objects.

- Customization of many features: memory allocation algorithm, index types or character types.

- Atomic constructions and destructions so that if the segment is shared between two processes it's impossible to create two objects associated with the same name, simplifying synchronization.

Declaration of managed memory segment classes

All **Boost.Interprocess** managed memory segment classes are templatized classes that can be customized by the user:

```
template
        <
           class CharType,
           class MemoryAlgorithm,
           template<class IndexConfig> class IndexType
        >
class basic_managed_shared_memory / basic_managed_mapped_file /
        basic_managed_heap_memory   / basic_external_buffer;
```

These classes can be customized with the following template parameters:

- **CharType** is the type of the character that will be used to identify the created named objects (for example, **char** or **wchar_t**)

- **MemoryAlgorithm** is the memory algorithm used to allocate portions of the segment (for example, rbtree_best_fit). The internal typedefs of the memory algorithm also define:

  - The synchronization type (MemoryAlgorithm::mutex_family) to be used in all allocation operations. This allows the use of user-defined mutexes or avoiding internal locking (maybe code will be externally synchronized by the user).

  - The Pointer type (MemoryAlgorithm::void_pointer) to be used by the memory allocation algorithm or additional helper structures (like a map to maintain object/name associations). All STL compatible allocators and containers to be used with this managed memory segment will use this pointer type. The pointer type will define if the managed memory segment can be mapped between several processes. For example, if void_pointer is offset_ptr<void> we will be able to map the managed segment in different base addresses in each process. If void_pointer is void* only fixed address mapping could be used.

  - See Writing a new memory allocation algorithm for more details about memory algorithms.

- **IndexType** is the type of index that will be used to store the name-object association (for example, a map, a hash-map, or an ordered vector).

This way, we can use char or wchar_t strings to identify created C++ objects in the memory segment, we can plug new shared memory allocation algorithms, and use the index type that is best suited to our needs.

Managed Shared Memory

Common Managed Shared Memory Classes

As seen, **basic_managed_shared_memory** offers a great variety of customization. But for the average user, a common, default shared memory named object creation is needed. Because of this, **Boost.Interprocess** defines the most common managed shared memory specializations:

```
//!Defines a managed shared memory with c-strings as keys for named objects,
//!the default memory algorithm (with process-shared mutexes,
//!and offset_ptr as internal pointers) as memory allocation algorithm
//!and the default index type as the index.
//!This class allows the shared memory to be mapped in different base
//!in different processes
typedef
   basic_managed_shared_memory<char
                            ,/*Default memory algorithm defining offset_ptr<void> as void_point↵
er*/
                            ,/*Default index type*/>
   managed_shared_memory;

//!Defines a managed shared memory with wide strings as keys for named objects,
//!the default memory algorithm (with process-shared mutexes,
//!and offset_ptr as internal pointers) as memory allocation algorithm
//!and the default index type as the index.
//!This class allows the shared memory to be mapped in different base
//!in different processes
typedef
   basic_managed_shared_memory<wchar_t
                            ,/*Default memory algorithm defining offset_ptr<void> as void_point↵
er*/
                            ,/*Default index type*/>
   wmanaged_shared_memory;
```

`managed_shared_memory` allocates objects in shared memory associated with a c-string and `wmanaged_shared_memory` allocates objects in shared memory associated with a wchar_t null terminated string. Both define the pointer type as `offset_ptr<void>` so they can be used to map the shared memory at different base addresses in different processes.

If the user wants to map the shared memory in the same address in all processes and want to use raw pointers internally instead of offset pointers, **Boost.Interprocess** defines the following types:

```
//!Defines a managed shared memory with c-strings as keys for named objects,
//!the default memory algorithm (with process-shared mutexes,
//!and offset_ptr as internal pointers) as memory allocation algorithm
//!and the default index type as the index.
//!This class allows the shared memory to be mapped in different base
//!in different processes*/
typedef basic_managed_shared_memory
   <char
   ,/*Default memory algorithm defining void * as void_pointer*/
   ,/*Default index type*/>
fixed_managed_shared_memory;

//!Defines a managed shared memory with wide strings as keys for named objects,
//!the default memory algorithm (with process-shared mutexes,
//!and offset_ptr as internal pointers) as memory allocation algorithm
//!and the default index type as the index.
//!This class allows the shared memory to be mapped in different base
//!in different processes
typedef basic_managed_shared_memory
   <wchar_t
   ,/*Default memory algorithm defining void * as void_pointer*/
   ,/*Default index type*/>
wfixed_managed_shared_memory;
```

Constructing Managed Shared Memory

Managed shared memory is an advanced class that combines a shared memory object and a mapped region that covers all the shared memory object. That means that when we **create** a new managed shared memory:

• A new shared memory object is created.

• The whole shared memory object is mapped in the process' address space.

• Some helper objects are constructed (name-object index, internal synchronization objects, internal variables...) in the mapped region to implement managed memory segment features.

When we **open** a managed shared memory

• A shared memory object is opened.

• The whole shared memory object is mapped in the process' address space.

To use a managed shared memory, you must include the following header:

```
#include <boost/interprocess/managed_shared_memory.hpp>
```

```
//1.  Creates a new shared memory object
//    called "MySharedMemory".
//2.  Maps the whole object to this
//    process' address space.
//3.  Constructs some objects in shared memory
//    to implement managed features.
//!! If anything fails, throws interprocess_exception
//
managed_shared_memory segment       ( create_only
                                    , "MySharedMemory" //Shared memory object name
                                    , 65536);          //Shared memory object size in bytes
```

```
//1.  Opens a shared memory object
//    called "MySharedMemory".
//2.  Maps the whole object to this
//    process' address space.
//3.  Obtains pointers to constructed internal objects
//    to implement managed features.
//!! If anything fails, throws interprocess_exception
//
managed_shared_memory segment       (open_only,       "MySharedMemory");//Shared memory object name
```

```
//1.  If the segment was previously created
//    equivalent to "open_only".
//2.  Otherwise, equivalent to "open_only" (size is ignored)
//!! If anything fails, throws interprocess_exception
//
managed_shared_memory segment       ( open_or_create
                                    , "MySharedMemory" //Shared memory object name
                                    , 65536);          //Shared memory object size in bytes
```

When the `managed_shared_memory` object is destroyed, the shared memory object is automatically unmapped, and all the resources are freed. To remove the shared memory object from the system you must use the `shared_memory_object::remove` function. Shared memory object removing might fail if any process still has the shared memory object mapped.

The user can also map the managed shared memory in a fixed address. This option is essential when using using `fixed_managed_shared_memory`. To do this, just add the mapping address as an extra parameter:

```
fixed_managed_shared_memory segment       (open_only       ,"MyFixedAddressSharedMemory" //Shared ↵
memory object name
    ,(void*)0x30000000              //Mapping address
```

Using native windows shared memory

Windows users might also want to use native windows shared memory instead of the portable `shared_memory_object` managed memory. This is achieved through the `basic_managed_windows_shared_memory` class. To use it just include:

```
#include <boost/interprocess/managed_windows_shared_memory.hpp>
```

This class has the same interface as `basic_managed_shared_memory` but uses native windows shared memory. Note that this managed class has the same lifetime issues as the windows shared memory: when the last process attached to the windows shared memory is detached from the memory (or ends/crashes) the memory is destroyed. So there is no persistence support for windows shared memory.

To communicate between system services and user applications using `managed_windows_shared_memory`, please read the explanations given in chapter [interprocess.sharedmemorybetweenprocesses.sharedmemory.windows_shared_memory Native windows shared memory]

Using XSI (system V) shared memory

Unix users might also want to use XSI (system V) instead of the portable `shared_memory_object` managed memory. This is achieved through the `basic_managed_xsi_shared_memory` class. To use it just include:

```
#include <boost/interprocess/managed_xsi_shared_memory.hpp>
```

This class has nearly the same interface as `basic_managed_shared_memory` but uses XSI shared memory as backend.

For more information about managed shared memory capabilities, see `basic_managed_shared_memory` class reference.

Managed Mapped File

Common Managed Mapped Files

As seen, **basic_managed_mapped_file** offers a great variety of customization. But for the average user, a common, default shared memory named object creation is needed. Because of this, **Boost.Interprocess** defines the most common managed mapped file specializations:

```
//Named object creation managed memory segment
//All objects are constructed in the memory-mapped file
//   Names are c-strings,
//   Default memory management algorithm(rbtree_best_fit with no mutexes)
//   Name-object mappings are stored in the default index type (flat_map)
typedef basic_managed_mapped_file <
   char,
   rbtree_best_fit<mutex_family, offset_ptr<void> >,
   flat_map_index
   >  managed_mapped_file;

//Named object creation managed memory segment
//All objects are constructed in the memory-mapped file
//   Names are wide-strings,
//   Default memory management algorithm(rbtree_best_fit with no mutexes)
//   Name-object mappings are stored in the default index type (flat_map)
typedef basic_managed_mapped_file<
   wchar_t,
   rbtree_best_fit<mutex_family, offset_ptr<void> >,
   flat_map_index
   >  wmanaged_mapped_file;
```

`managed_mapped_file` allocates objects in a memory mapped files associated with a c-string and `wmanaged_mapped_file` allocates objects in a memory mapped file associated with a wchar_t null terminated string. Both define the pointer type as `offset_ptr<void>` so they can be used to map the file at different base addresses in different processes.

Constructing Managed Mapped Files

Managed mapped file is an advanced class that combines a file and a mapped region that covers all the file. That means that when we **create** a new managed mapped file:

• A new file is created.

• The whole file is mapped in the process' address space.

69

- Some helper objects are constructed (name-object index, internal synchronization objects, internal variables...) in the mapped region to implement managed memory segment features.

When we **open** a managed mapped file

- A file is opened.

- The whole file is mapped in the process' address space.

To use a managed mapped file, you must include the following header:

```
#include <boost/interprocess/managed_mapped_file.hpp>
```

```
//1.  Creates a new file
//    called "MyMappedFile".
//2.  Maps the whole file to this
//    process' address space.
//3.  Constructs some objects in the memory mapped
//    file to implement managed features.
//!! If anything fails, throws interprocess_exception
//
managed_mapped_file mfile        (create_only,      "MyMappedFile",   //Mapped file name       ↵
65536);          //Mapped file size
```

```
//1.  Opens a file
//    called "MyMappedFile".
//2.  Maps the whole file to this
//    process' address space.
//3.  Obtains pointers to constructed internal objects
//    to implement managed features.
//!! If anything fails, throws interprocess_exception
//
managed_mapped_file mfile        (open_only,      "MyMappedFile"); //Mapped file name[c++]

//1.  If the file was previously created
//    equivalent to "open_only".
//2.  Otherwise, equivalent to "open_only" (size is ignored)
//
//!! If anything fails, throws interprocess_exception
//
managed_mapped_file mfile        (open_or_create,      "MyMappedFile",   //Mapped file name       ↵
65536);          //Mapped file size
```

When the `managed_mapped_file` object is destroyed, the file is automatically unmapped, and all the resources are freed. To remove the file from the filesystem you could use standard C `std::remove` or **Boost.Filesystem**'s `remove()` functions, but file removing might fail if any process still has the file mapped in memory or the file is open by any process.

To obtain a more portable behaviour, use `file_mapping::remove(const char *)` operation, which will remove the file even if it's being mapped. However, removal will fail in some OS systems if the file (eg. by C++ file streams) and no delete share permission was granted to the file. But in most common cases `file_mapping::remove` is portable enough.

For more information about managed mapped file capabilities, see `basic_managed_mapped_file` class reference.

Managed Memory Segment Features

The following features are common to all managed memory segment classes, but we will use managed shared memory in our examples. We can do the same with memory mapped files or other managed memory segment classes.

Allocating fragments of a managed memory segment

If a basic raw-byte allocation is needed from a managed memory segment, (for example, a managed shared memory), to implement top-level interprocess communications, this class offers **allocate** and **deallocate** functions. The allocation function comes with throwing and no throwing versions. Throwing version throws boost::interprocess::bad_alloc (which derives from `std::bad_alloc`) if there is no more memory and the non-throwing version returns 0 pointer.

```cpp
#include <boost/interprocess/managed_shared_memory.hpp>

int main()
{
   using namespace boost::interprocess;

   //Remove shared memory on construction and destruction
   struct shm_remove
   {
      shm_remove() { shared_memory_object::remove("MySharedMemory"); }
      ~shm_remove(){ shared_memory_object::remove("MySharedMemory"); }
   } remover;

   //Managed memory segment that allocates portions of a shared memory
   //segment with the default management algorithm
   managed_shared_memory managed_shm(create_only,"MySharedMemory", 65536);

   //Allocate 100 bytes of memory from segment, throwing version
   void *ptr = managed_shm.allocate(100);

   //Deallocate it
   managed_shm.deallocate(ptr);

   //Non throwing version
   ptr = managed_shm.allocate(100, std::nothrow);

   //Deallocate it
   managed_shm.deallocate(ptr);
   return 0;
}
```

Obtaining handles to identify data

The class also offers conversions between absolute addresses that belong to a managed memory segment and a handle that can be passed using any interprocess mechanism. That handle can be transformed again to an absolute address using a managed memory segment that also contains that object. Handles can be used as keys between processes to identify allocated portions of a managed memory segment or objects constructed in the managed segment.

```cpp
//Process A obtains the offset of the address
managed_shared_memory::handle handle =
   segment.get_handle_from_address(processA_address);

//Process A sends this address using any mechanism to process B

//Process B obtains the handle and transforms it again to an address
managed_shared_memory::handle handle = ...
void * processB_address = segment.get_address_from_handle(handle);
```

Object construction function family

When constructing objects in a managed memory segment (managed shared memory, managed mapped files...) associated with a name, the user has a varied object construction family to "construct" or to "construct if not found". **Boost.Interprocess** can construct

a single object or an array of objects. The array can be constructed with the same parameters for all objects or we can define each parameter from a list of iterators:

```
//!Allocates and constructs an object of type MyType (throwing version)
MyType *ptr = managed_memory_segment.construct<MyType>("Name") (par1, par2...);

//!Allocates and constructs an array of objects of type MyType (throwing version)
//!Each object receives the same parameters (par1, par2, ...)
MyType *ptr = managed_memory_segment.construct<MyType>("Name")[count](par1, par2...);

//!Tries to find a previously created object. If not present, allocates
//!and constructs an object of type MyType (throwing version)
MyType *ptr = managed_memory_segment.find_or_construct<MyType>("Name") (par1, par2...);

//!Tries to find a previously created object. If not present, allocates and
//!constructs an array of objects of type MyType (throwing version). Each object
//!receives the same parameters (par1, par2, ...)
MyType *ptr = managed_memory_segment.find_or_construct<MyType>("Name")[count](par1, par2...);

//!Allocates and constructs an array of objects of type MyType (throwing version)
//!Each object receives parameters returned with the expression (*it1++, *it2++,... )
MyType *ptr = managed_memory_segment.construct_it<MyType>("Name")[count](it1, it2...);

//!Tries to find a previously created object. If not present, allocates and constructs
//!an array of objects of type MyType (throwing version).  Each object receives
//!parameters returned with the expression (*it1++, *it2++,... )
MyType *ptr = managed_memory_segment.find_or_construct_it<MyType>("Name")[count](it1, it2...);

//!Tries to find a previously created object. Returns a pointer to the object and the
//!count (if it is not an array, returns 1). If not present, the returned pointer is 0
std::pair<MyType *,std::size_t> ret = managed_memory_segment.find<MyType>("Name");

//!Destroys the created object, returns false if not present
bool destroyed = managed_memory_segment.destroy<MyType>("Name");

//!Destroys the created object via pointer
managed_memory_segment.destroy_ptr(ptr);
```

All these functions have a non-throwing version, that is invoked with an additional parameter std::nothrow. For example, for simple object construction:

```
//!Allocates and constructs an object of type MyType (no throwing version)
MyType *ptr = managed_memory_segment.construct<MyType>("Name", std::nothrow) (par1, par2...);
```

Anonymous instance construction

Sometimes, the user doesn't want to create class objects associated with a name. For this purpose, **Boost.Interprocess** can create anonymous objects in a managed memory segment. All named object construction functions are available to construct anonymous objects. To allocate an anonymous objects, the user must use "boost::interprocess::anonymous_instance" name instead of a normal name:

```
MyType *ptr = managed_memory_segment.construct<MyType>(anonymous_instance) (par1, par2...);

//Other construct variants can also be used (including non-throwing ones)
...

//We can only destroy the anonymous object via pointer
managed_memory_segment.destroy_ptr(ptr);
```

Find functions have no sense here, since anonymous objects have no name. We can only destroy the anonymous object via pointer.

Unique instance construction

Sometimes, the user wants to emulate a singleton in a managed memory segment. Obviously, as the managed memory segment is constructed at run-time, the user must construct and destroy this object explicitly. But how can the user be sure that the object is the only object of its type in the managed memory segment? This can be emulated using a named object and checking if it is present before trying to create one, but all processes must agree in the object's name, that can also conflict with other existing names.

To solve this, **Boost.Interprocess** offers a "unique object" creation in a managed memory segment. Only one instance of a class can be created in a managed memory segment using this "unique object" service (you can create more named objects of this class, though) so it makes easier the emulation of singleton-like objects across processes, for example, to design pooled, shared memory allocators. The object can be searched using the type of the class as a key.

```cpp
// Construct
MyType *ptr = managed_memory_segment.construct<MyType>(unique_instance) (par1, par2...);

// Find it
std::pair<MyType *,std::size_t> ret = managed_memory_segment.find<MyType>(unique_instance);

// Destroy it
managed_memory_segment.destroy<MyType>(unique_instance);

// Other construct and find variants can also be used (including non-throwing ones)
//...
```

```cpp
// We can also destroy the unique object via pointer
MyType *ptr = managed_memory_segment.construct<MyType>(unique_instance) (par1, par2...);
managed_shared_memory.destroy_ptr(ptr);
```

The find function obtains a pointer to the only object of type T that can be created using this "unique instance" mechanism.

Synchronization guarantees

One of the features of named/unique allocations/searches/destructions is that they are **atomic**. Named allocations use the recursive synchronization scheme defined by the internal `mutex_family` typedef defined of the memory allocation algorithm template parameter (`MemoryAlgorithm`). That is, the mutex type used to synchronize named/unique allocations is defined by the `MemoryAlgorithm::mutex_family::recursive_mutex_type` type. For shared memory, and memory mapped file based managed segments this recursive mutex is defined as `interprocess_recursive_mutex`.

If two processes can call:

```cpp
MyType *ptr = managed_shared_memory.find_or_construct<MyType>("Name")[count](par1, par2...);
```

at the same time, but only one process will create the object and the other will obtain a pointer to the created object.

Raw allocation using `allocate()` can be called also safely while executing named/anonymous/unique allocations, just like when programming a multithreaded application inserting an object in a mutex-protected map does not block other threads from calling new[] while the map thread is searching the place where it has to insert the new object. The synchronization does happen once the map finds the correct place and it has to allocate raw memory to construct the new value.

This means that if we are creating or searching for a lot of named objects, we only block creation/searches from other processes but we don't block another process if that process is inserting elements in a shared memory vector.

Index types for name/object mappings

As seen, managed memory segments, when creating named objects, store the name/object association in an index. The index is a map with the name of the object as a key and a pointer to the object as the mapped type. The default specializations, **managed_shared_memory** and **wmanaged_shared_memory**, use **flat_map_index** as the index type.

Each index has its own characteristics, like search-time, insertion time, deletion time, memory use, and memory allocation patterns. **Boost.Interprocess** offers 3 index types right now:

- **boost::interprocess::flat_map_index flat_map_index**: Based on boost::interprocess::flat_map, an ordered vector similar to Loki library's AssocVector class, offers great search time and minimum memory use. But the vector must be reallocated when is full, so all data must be copied to the new buffer. Ideal when insertions are mainly in initialization time and in run-time we just need searches.

- **boost::interprocess::map_index map_index**: Based on boost::interprocess::map, a managed memory ready version of std::map. Since it's a node based container, it has no reallocations, the tree must be just rebalanced sometimes. Offers equilibrated insertion/deletion/search times with more overhead per node comparing to **boost::interprocess::flat_map_index**. Ideal when searches/insertions/deletions are in random order.

- **boost::interprocess::null_index null_index**: This index is for people using a managed memory segment just for raw memory buffer allocations and they don't make use of named/unique allocations. This class is just empty and saves some space and compilation time. If you try to use named object creation with a managed memory segment using this index, you will get a compilation error.

As an example, if we want to define new managed shared memory class using **boost::interprocess::map** as the index type we just must specify [boost::interprocess::map_index map_index] as a template parameter:

```
//This managed memory segment can allocate objects with:
// -> a wchar_t string as key
// -> boost::interprocess::rbtree_best_fit with process-shared mutexes
//        as memory allocation algorithm.
// -> boost::interprocess::map<...> as the index to store name/object mappings
//
typedef boost::interprocess::basic_managed_shared_memory
        <  wchar_t
        ,  boost::interprocess::rbtree_best_fit<boost::interprocess::mutex_family, off↵
set_ptr<void> >
        ,  boost::interprocess::map_index
        >  my_managed_shared_memory;
```

Boost.Interprocess plans to offer an **unordered_map** based index as soon as this container is included in Boost. If these indexes are not enough for you, you can define your own index type. To know how to do this, go to Building custom indexes section.

Segment Manager

All **Boost.Interprocess** managed memory segment classes construct in their respective memory segments (shared memory, memory mapped files, heap memory...) some structures to implement the memory management algorithm, named allocations, synchronization objects... All these objects are encapsulated in a single object called **segment manager**. A managed memory mapped file and a managed shared memory use the same **segment manager** to implement all managed memory segment features, due to the fact that a **segment manager** is a class that manages a fixed size memory buffer. Since both shared memory or memory mapped files are accessed though a mapped region, and a mapped region is a fixed size memory buffer, a single **segment manager** class can manage several managed memory segment types.

Some **Boost.Interprocess** classes require a pointer to the segment manager in their constructors, and the segment manager can be obtained from any managed memory segment using `get_segment_manager` member:

```
managed_shared_memory::segment_manager *seg_manager =
    managed_shm.get_segment_manager();
```

Obtaining information about a constructed object

Once an object is constructed using `construct<>` function family, the programmer can obtain information about the object using a pointer to the object. The programmer can obtain the following information:

- Name of the object: If it's a named instance, the name used in the construction function is returned, otherwise 0 is returned.

- Length of the object: Returns the number of elements of the object (1 if it's a single value, >=1 if it's an array).

- The type of construction: Whether the object was constructed using a named, unique or anonymous construction.

Here is an example showing this functionality:

```cpp
#include <boost/interprocess/managed_shared_memory.hpp>
#include <cassert>
#include <cstring>

class my_class
{
   //...
};

int main()
{
   using namespace boost::interprocess;

   //Remove shared memory on construction and destruction
   struct shm_remove
   {
      shm_remove() { shared_memory_object::remove("MySharedMemory"); }
      ~shm_remove(){ shared_memory_object::remove("MySharedMemory"); }
   } remover;

   managed_shared_memory managed_shm(create_only, "MySharedMemory", 10000*sizeof(std::size_t));

   //Construct objects
   my_class *named_object  = managed_shm.construct<my_class>("Object name")[1]();
   my_class *unique_object = managed_shm.construct<my_class>(unique_instance)[2]();
   my_class *anon_object   = managed_shm.construct<my_class>(anonymous_instance)[3]();

   //Now test "get_instance_name" function.
   assert(0 == std::strcmp(managed_shared_memory::get_instance_name(named_object), "Object name"));
   assert(0 == managed_shared_memory::get_instance_name(unique_object));
   assert(0 == managed_shared_memory::get_instance_name(anon_object));

   //Now test "get_instance_type" function.
   assert(named_type     == managed_shared_memory::get_instance_type(named_object));
   assert(unique_type    == managed_shared_memory::get_instance_type(unique_object));
   assert(anonymous_type == managed_shared_memory::get_instance_type(anon_object));

   //Now test "get_instance_length" function.
   assert(1 == managed_shared_memory::get_instance_length(named_object));
   assert(2 == managed_shared_memory::get_instance_length(unique_object));
   assert(3 == managed_shared_memory::get_instance_length(anon_object));

   managed_shm.destroy_ptr(named_object);
   managed_shm.destroy_ptr(unique_object);
   managed_shm.destroy_ptr(anon_object);
   return 0;
}
```

Executing an object function atomically

Sometimes the programmer must execute some code, and needs to execute it with the guarantee that no other process or thread will create or destroy any named, unique or anonymous object while executing the functor. A user might want to create several named objects and initialize them, but those objects should be available for the rest of processes at once.

To achieve this, the programmer can use the `atomic_func()` function offered by managed classes:

```
//This object function will create several named objects
create_several_objects_func func(/**/);

//While executing the function, no other process will be
//able to create or destroy objects
managed_memory.atomic_func(func);
```

Note that `atomic_func` does not prevent other processes from allocating raw memory or executing member functions for already constructed objects (e.g.: another process might be pushing elements into a vector placed in the segment). The atomic function only blocks named, unique and anonymous creation, search and destruction (concurrent calls to `construct<>`, `find<>`, `find_or_construct<>`, `destroy<>`...) from other processes.

Managed Memory Segment Advanced Features

Obtaining information about the managed segment

These functions are available to obtain information about the managed memory segments:

Obtain the size of the memory segment:

```
managed_shm.get_size();
```

Obtain the number of free bytes of the segment:

```
managed_shm.get_free_memory();
```

Clear to zero the free memory:

```
managed_shm.zero_free_memory();
```

Know if all memory has been deallocated, false otherwise:

```
managed_shm.all_memory_deallocated();
```

Test internal structures of the managed segment. Returns true if no errors are detected:

```
managed_shm.check_sanity();
```

Obtain the number of named and unique objects allocated in the segment:

```
managed_shm.get_num_named_objects();
managed_shm.get_num_unique_objects();
```

Growing managed segments

Once a managed segment is created the managed segment can't be grown. The limitation is not easily solvable: every process attached to the managed segment would need to be stopped, notified of the new size, they would need to remap the managed segment and continue working. Nearly impossible to achieve with a user-level library without the help of the operating system kernel.

On the other hand, **Boost.Interprocess** offers off-line segment growing. What does this mean? That the segment can be grown if no process has mapped the managed segment. If the application can find a moment where no process is attached it can grow or shrink to fit the managed segment.

Here we have an example showing how to grow and shrink to fit managed_shared_memory:

```cpp
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/managed_mapped_file.hpp>
#include <cassert>

class MyClass
{
   //...
};

int main()
{
   using namespace boost::interprocess;
   //Remove shared memory on construction and destruction
   struct shm_remove
   {
      shm_remove() { shared_memory_object::remove("MySharedMemory"); }
      ~shm_remove(){ shared_memory_object::remove("MySharedMemory"); }
   } remover;

   {
      //Create a managed shared memory
      managed_shared_memory shm(create_only, "MySharedMemory", 1000);

      //Check size
      assert(shm.get_size() == 1000);
      //Construct a named object
      MyClass *myclass = shm.construct<MyClass>("MyClass")();
      //The managed segment is unmapped here
   }
   {
      //Now that the segment is not mapped grow it adding extra 500 bytes
      managed_shared_memory::grow("MySharedMemory", 500);

      //Map it again
      managed_shared_memory shm(open_only, "MySharedMemory");
      //Check size
      assert(shm.get_size() == 1500);
      //Check "MyClass" is still there
      MyClass *myclass = shm.find<MyClass>("MyClass").first;
      assert(myclass != 0);
      //The managed segment is unmapped here
   }
   {
      //Now minimize the size of the segment
      managed_shared_memory::shrink_to_fit("MySharedMemory");

      //Map it again
      managed_shared_memory shm(open_only, "MySharedMemory");
      //Check size
      assert(shm.get_size() < 1000);
      //Check "MyClass" is still there
      MyClass *myclass = shm.find<MyClass>("MyClass").first;
      assert(myclass != 0);
      //The managed segment is unmapped here
   }
   return 0;
}
```

managed_mapped_file also offers a similar function to grow or shrink_to_fit the managed file. Please, remember that **no process should be modifying the file/shared memory while the growing/shrinking process is performed**. Otherwise, the managed segment will be corrupted.

Advanced index functions

As mentioned, the managed segment stores the information about named and unique objects in two indexes. Depending on the type of those indexes, the index must reallocate some auxiliary structures when new named or unique allocations are made. For some indexes, if the user knows how many named or unique objects are going to be created it's possible to preallocate some structures to obtain much better performance. (If the index is an ordered vector it can preallocate memory to avoid reallocations. If the index is a hash structure it can preallocate the bucket array).

The following functions reserve memory to make the subsequent allocation of named or unique objects more efficient. These functions are only useful for pseudo-intrusive or non-node indexes (like `flat_map_index`, `iunordered_set_index`). These functions have no effect with the default index (`iset_index`) or other indexes (`map_index`):

```
managed_shm.reserve_named_objects(1000);
managed_shm.reserve_unique_objects(1000);
```

```
managed_shm.reserve_named_objects(1000);
managed_shm.reserve_unique_objects(1000);
```

Managed memory segments also offer the possibility to iterate through constructed named and unique objects for debugging purposes. **Caution: this iteration is not thread-safe** so the user should make sure that no other thread is manipulating named or unique indexes (creating, erasing, reserving...) in the segment. Other operations not involving indexes can be concurrently executed (raw memory allocation/deallocations, for example).

The following functions return constant iterators to the range of named and unique objects stored in the managed segment. Depending on the index type, iterators might be invalidated after a named or unique creation/erasure/reserve operation:

```
typedef managed_shared_memory::const_named_iterator const_named_it;
const_named_it named_beg = managed_shm.named_begin();
const_named_it named_end = managed_shm.named_end();

typedef managed_shared_memory::const_unique_iterator const_unique_it;
const_unique_it unique_beg = managed_shm.unique_begin();
const_unique_it unique_end = managed_shm.unique_end();

for(; named_beg != named_end; ++named_beg){
   //A pointer to the name of the named object
   const managed_shared_memory::char_type *name = named_beg->name();
   //The length of the name
   std::size_t name_len = named_beg->name_length();
   //A constant void pointer to the named object
   const void *value = named_beg->value();
}

for(; unique_beg != unique_end; ++unique_beg){
   //The typeid(T).name() of the unique object
   const char *typeid_name = unique_beg->name();
   //The length of the name
   std::size_t name_len = unique_beg->name_length();
   //A constant void pointer to the unique object
   const void *value = unique_beg->value();
}
```

Allocating aligned memory portions

Sometimes it's interesting to be able to allocate aligned fragments of memory because of some hardware or software restrictions. Sometimes, having aligned memory is a feature that can be used to improve several memory algorithms.

This allocation is similar to the previously shown raw memory allocation but it takes an additional parameter specifying the alignment. There is a restriction for the alignment: **the alignment must be power of two**.

If a user wants to allocate many aligned blocks (for example aligned to 128 bytes), the size that minimizes the memory waste is a value that's is nearly a multiple of that alignment (for example 2*128 - some bytes). The reason for this is that every memory allocation usually needs some additional metadata in the first bytes of the allocated buffer. If the user can know the value of "some bytes" and if the first bytes of a free block of memory are used to fulfill the aligned allocation, the rest of the block can be left also aligned and ready for the next aligned allocation. Note that requesting **a size multiple of the alignment is not optimal** because lefts the next block of memory unaligned due to the needed metadata.

Once the programmer knows the size of the payload of every memory allocation, he can request a size that will be optimal to allocate aligned chunks of memory maximizing both the size of the request **and** the possibilities of future aligned allocations. This information is stored in the PayloadPerAllocation constant of managed memory segments.

Here is a small example showing how aligned allocation is used:

```cpp
#include <boost/interprocess/managed_shared_memory.hpp>
#include <cassert>

int main()
{
   using namespace boost::interprocess;

   //Remove shared memory on construction and destruction
   struct shm_remove
   {
      shm_remove() { shared_memory_object::remove("MySharedMemory"); }
      ~shm_remove(){ shared_memory_object::remove("MySharedMemory"); }
   } remover;

   //Managed memory segment that allocates portions of a shared memory
   //segment with the default management algorithm
   managed_shared_memory managed_shm(create_only, "MySharedMemory", 65536);

   const std::size_t Alignment = 128;

   //Allocate 100 bytes aligned to Alignment from segment, throwing version
   void *ptr = managed_shm.allocate_aligned(100, Alignment);

   //Check alignment
   assert((static_cast<char*>(ptr)-static_cast<char*>(0)) % Alignment == 0);

   //Deallocate it
   managed_shm.deallocate(ptr);

   //Non throwing version
   ptr = managed_shm.allocate_aligned(100, Alignment, std::nothrow);

   //Check alignment
   assert((static_cast<char*>(ptr)-static_cast<char*>(0)) % Alignment == 0);

   //Deallocate it
   managed_shm.deallocate(ptr);

   //If we want to efficiently allocate aligned blocks of memory
   //use managed_shared_memory::PayloadPerAllocation value
   assert(Alignment > managed_shared_memory::PayloadPerAllocation);

   //This allocation will maximize the size of the aligned memory
   //and will increase the possibility of finding more aligned memory
   ptr = managed_shm.allocate_aligned
      (3*Alignment - managed_shared_memory::PayloadPerAllocation, Alignment);
```

```
    //Check alignment
    assert((static_cast<char*>(ptr)-static_cast<char*>(0)) % Alignment == 0);

    //Deallocate it
    managed_shm.deallocate(ptr);

    return 0;
}
```

Multiple allocation functions

If an application needs to allocate a lot of memory buffers but it needs to deallocate them independently, the application is normally forced to loop calling `allocate()`. Managed memory segments offer an alternative function to pack several allocations in a single call obtaining memory buffers that:

• are packed contiguously in memory (which improves locality)

• can be independently deallocated.

This allocation method is much faster than calling `allocate()` in a loop. The downside is that the segment must provide a contiguous memory segment big enough to hold all the allocations. Managed memory segments offer this functionality through `allocate_many()` functions. There are 2 types of `allocate_many` functions:

• Allocation of N buffers of memory with the same size.

• Allocation ot N buffers of memory, each one of different size.

```
//!Allocates n_elements of elem_size bytes.
multiallocation_iterator allocate_many(std::size_t elem_size, std::size_t min_ele↵
ments, std::size_t preferred_elements, std::size_t &received_elements);

//!Allocates n_elements, each one of elem_sizes[i] bytes.
multiallocation_iterator allocate_many(const std::size_t *elem_sizes, std::size_t n_elements);

//!Allocates n_elements of elem_size bytes. No throwing version.
multiallocation_iterator allocate_many(std::size_t elem_size, std::size_t min_ele↵
ments, std::size_t preferred_elements, std::size_t &received_elements, std::nothrow_t nothrow);

//!Allocates n_elements, each one of elem_sizes[i] bytes. No throwing version.
multiallocation_iterator allocate_many(const std::size_t *elem_sizes, std::size_t n_ele↵
ments, std::nothrow_t nothrow);
```

All functions return a `multiallocation iterator` that can be used to obtain pointers to memory the user can overwrite. A `multiallocation_iterator`:

• Becomes invalidated if the memory is pointing to is deallocated or the next iterators (which previously were reachable with `oper-ator++`) become invalid.

• Returned from `allocate_many` can be checked in a boolean expression to know if the allocation has been successful.

• A default constructed `multiallocation iterator` indicates both an invalid iterator and the "end" iterator.

• Dereferencing an iterator (`operator *()`) returns a `char &` referencing the first byte user can overwrite in the memory buffer.

• The iterator category depends on the memory allocation algorithm, but it's at least a forward iterator.

Here is a small example showing all this functionality:

```cpp
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/detail/move.hpp> //boost::interprocess::move
#include <cassert>//assert
#include <cstring>//std::memset
#include <new>      //std::nothrow
#include <vector> //std::vector

int main()
{
   using namespace boost::interprocess;
   typedef managed_shared_memory::multiallocation_chain multiallocation_chain;

   //Remove shared memory on construction and destruction
   struct shm_remove
   {
      shm_remove() { shared_memory_object::remove("MySharedMemory"); }
      ~shm_remove(){ shared_memory_object::remove("MySharedMemory"); }
   } remover;

   managed_shared_memory managed_shm(create_only,"MySharedMemory", 65536);

   //Allocate 16 elements of 100 bytes in a single call. Non-throwing version.
   multiallocation_chain chain(managed_shm.allocate_many(100, 16, std::nothrow));

   //Check if the memory allocation was successful
   if(chain.empty()) return 1;

   //Allocated buffers
   std::vector<void*> allocated_buffers;

   //Initialize our data
   while(!chain.empty()){
      void *buf = chain.front();
      chain.pop_front();
      allocated_buffers.push_back(buf);
      //The iterator must be incremented before overwriting memory
      //because otherwise, the iterator is invalidated.
      std::memset(buf, 0, 100);
   }

   //Now deallocate
   while(!allocated_buffers.empty()){
      managed_shm.deallocate(allocated_buffers.back());
      allocated_buffers.pop_back();
   }

   //Allocate 10 buffers of different sizes in a single call. Throwing version
   std::size_t sizes[10];
   for(std::size_t i = 0; i < 10; ++i)
      sizes[i] = i*3;

   chain = managed_shm.allocate_many(sizes, 10);
   managed_shm.deallocate_many(boost::interprocess::move(chain));
   return 0;
}
```

Allocating N buffers of the same size improves the performance of pools and node containers (for example STL-like lists): when inserting a range of forward iterators in a STL-like list, the insertion function can detect the number of needed elements and allocate in a single call. The nodes still can be deallocated.

Allocating N buffers of different sizes can be used to speed up allocation in cases where several objects must always be allocated at the same time but deallocated at different times. For example, a class might perform several initial allocations (some header data for

a network packet, for example) in its constructor but also allocations of buffers that might be reallocated in the future (the data to be sent through the network). Instead of allocating all the data independently, the constructor might use `allocate_many()` to speed up the initialization, but it still can deallocate and expand the memory of the variable size element.

In general, `allocate_many` is useful with large values of N. Overuse of `allocate_many` can increase the effective memory usage, because it can't reuse existing non-contiguous memory fragments that might be available for some of the elements.

Expand in place memory allocation

When programming some data structures such as vectors, memory reallocation becomes an important tool to improve performance. Managed memory segments offer an advanced reallocation function that offers:

- Forward expansion: An allocated buffer can be expanded so that the end of the buffer is moved further. New data can be written between the old end and the new end.

- Backwards expansion: An allocated buffer can be expanded so that the beginning of the buffer is moved backwards. New data can be written between the new beginning and the old beginning.

- Shrinking: An allocated buffer can be shrunk so that the end of the buffer is moved backwards. The memory between the new end and the old end can be reused for future allocations.

The expansion can be combined with the allocation of a new buffer if the expansion fails obtaining a function with "expand, if fails allocate a new buffer" semantics.

Apart from this features, the function always returns the real size of the allocated buffer, because many times, due to alignment issues the allocated buffer a bit bigger than the requested size. Thus, the programmer can maximize the memory use using `allocation_command`.

Here is the declaration of the function:

```cpp
enum boost::interprocess::allocation_type
{
   //Bitwise OR (|) combinable values
   boost::interprocess::allocate_new        = ...,
   boost::interprocess::expand_fwd          = ...,
   boost::interprocess::expand_bwd          = ...,
   boost::interprocess::shrink_in_place     = ...,
   boost::interprocess::nothrow_allocation  = ...
};


template<class T>
std::pair<T *, bool>
   allocation_command( boost::interprocess::allocation_type command
                     , std::size_t limit_size
                     , std::size_t preferred_size
                     , std::size_t &received_size
                     , T *reuse_ptr = 0);
```

Preconditions for the function:

- If the parameter command contains the value `boost::interprocess::shrink_in_place` it can't contain any of these values: `boost::interprocess::expand_fwd`, `boost::interprocess::expand_bwd`.

- If the parameter command contains `boost::interprocess::expand_fwd` or `boost::interprocess::expand_bwd`, the parameter `reuse_ptr` must be non-null and returned by a previous allocation function.

- If the parameter command contains the value `boost::interprocess::shrink_in_place`, the parameter `limit_size` must be equal or greater than the parameter `preferred_size`.

- If the parameter `command` contains any of these values: `boost::interprocess::expand_fwd` or `boost::interprocess::expand_bwd`, the parameter `limit_size` must be equal or less than the parameter `preferred_size`.

Which are the effects of this function:

- If the parameter command contains the value `boost::interprocess::shrink_in_place`, the function will try to reduce the size of the memory block referenced by pointer `reuse_ptr` to the value `preferred_size` moving only the end of the block. If it's not possible, it will try to reduce the size of the memory block as much as possible as long as this results in `size(p) <= limit_size`. Success is reported only if this results in `preferred_size <= size(p)` and `size(p) <= limit_size`.

- If the parameter `command` only contains the value `boost::interprocess::expand_fwd` (with optional additional `boost::interprocess::nothrow_allocation`), the allocator will try to increase the size of the memory block referenced by pointer reuse moving only the end of the block to the value `preferred_size`. If it's not possible, it will try to increase the size of the memory block as much as possible as long as this results in `size(p) >= limit_size`. Success is reported only if this results in `limit_size <= size(p)`.

- If the parameter `command` only contains the value `boost::interprocess::expand_bwd` (with optional additional `boost::interprocess::nothrow_allocation`), the allocator will try to increase the size of the memory block referenced by pointer `reuse_ptr` only moving the start of the block to a returned new position `new_ptr`. If it's not possible, it will try to move the start of the block as much as possible as long as this results in `size(new_ptr) >= limit_size`. Success is reported only if this results in `limit_size <= size(new_ptr)`.

- If the parameter `command` only contains the value `boost::interprocess::allocate_new` (with optional additional `boost::interprocess::nothrow_allocation`), the allocator will try to allocate memory for `preferred_size` objects. If it's not possible it will try to allocate memory for at least `limit_size` objects.

- If the parameter `command` only contains a combination of `boost::interprocess::expand_fwd` and `boost::interprocess::allocate_new`, (with optional additional `boost::interprocess::nothrow_allocation`) the allocator will try first the forward expansion. If this fails, it would try a new allocation.

- If the parameter `command` only contains a combination of `boost::interprocess::expand_bwd` and `boost::interprocess::allocate_new` (with optional additional `boost::interprocess::nothrow_allocation`), the allocator will try first to obtain `preferred_size` objects using both methods if necessary. If this fails, it will try to obtain `limit_size` objects using both methods if necessary.

- If the parameter `command` only contains a combination of `boost::interprocess::expand_fwd` and `boost::interprocess::expand_bwd` (with optional additional `boost::interprocess::nothrow_allocation`), the allocator will try first forward expansion. If this fails it will try to obtain preferred_size objects using backwards expansion or a combination of forward and backwards expansion. If this fails, it will try to obtain `limit_size` objects using both methods if necessary.

- If the parameter `command` only contains a combination of allocation_new, `boost::interprocess::expand_fwd` and `boost::interprocess::expand_bwd`, (with optional additional `boost::interprocess::nothrow_allocation`) the allocator will try first forward expansion. If this fails it will try to obtain preferred_size objects using new allocation, backwards expansion or a combination of forward and backwards expansion. If this fails, it will try to obtain `limit_size` objects using the same methods.

- The allocator always writes the size or the expanded/allocated/shrunk memory block in `received_size`. On failure the allocator writes in `received_size` a possibly successful `limit_size` parameter for a new call.

Throws an exception if two conditions are met:

- The allocator is unable to allocate/expand/shrink the memory or there is an error in preconditions

- The parameter command does not contain `boost::interprocess::nothrow_allocation`.

This function returns:

- The address of the allocated memory or the new address of the expanded memory as the first member of the pair. If the parameter command contains `boost::interprocess::nothrow_allocation` the first member will be 0 if the allocation/expansion fails or there is an error in preconditions.

- The second member of the pair will be false if the memory has been allocated, true if the memory has been expanded. If the first member is 0, the second member has an undefined value.

Notes:

- If the user chooses `char` as template argument the returned buffer will be suitably aligned to hold any type.

- If the user chooses `char` as template argument and a backwards expansion is performed, although properly aligned, the returned buffer might not be suitable because the distance between the new beginning and the old beginning might not multiple of the type the user wants to construct, since due to internal restrictions the expansion can be slightly bigger than the requested bytes. **When performing backwards expansion, if you have already constructed objects in the old buffer, make sure to specify correctly the type.**

Here is a small example that shows the use of `allocation_command`:

```cpp
#include <boost/interprocess/managed_shared_memory.hpp>
#include <cassert>

int main()
{
   using namespace boost::interprocess;

   //Remove shared memory on construction and destruction
   struct shm_remove
   {
      shm_remove() { shared_memory_object::remove("MySharedMemory"); }
      ~shm_remove(){ shared_memory_object::remove("MySharedMemory"); }
   } remover;

   //Managed memory segment that allocates portions of a shared memory
   //segment with the default management algorithm
   managed_shared_memory managed_shm(create_only, "MySharedMemory", 10000*sizeof(std::size_t));

   //Allocate at least 100 bytes, 1000 bytes if possible
   std::size_t received_size, min_size = 100, preferred_size = 1000;
   std::size_t *ptr = managed_shm.allocation_command<std::size_t>
      (boost::interprocess::allocate_new, min_size, preferred_size, received_size).first;

   //Received size must be bigger than min_size
   assert(received_size >= min_size);

   //Get free memory
   std::size_t free_memory_after_allocation = managed_shm.get_free_memory();

   //Now write the data
   for(std::size_t i = 0; i < received_size; ++i) ptr[i] = i;

   //Now try to triplicate the buffer. We won't admit an expansion
   //lower to the double of the original buffer.
   //This "should" be successful since no other class is allocating
   //memory from the segment
   std::size_t expanded_size;
   std::pair<std::size_t *, bool> ret = managed_shm.allocation_command
      (boost::interprocess::expand_fwd, received_size*2, received_size*3, expanded_size, ptr);

   //Check invariants
   assert(ret.second == true);
   assert(ret.first == ptr);
   assert(expanded_size >= received_size*2);

   //Get free memory and compare
   std::size_t free_memory_after_expansion = managed_shm.get_free_memory();
```

```
   assert(free_memory_after_expansion < free_memory_after_allocation);

   //Write new values
   for(std::size_t i = received_size; i < expanded_size; ++i)  ptr[i] = i;

   //Try to shrink approximately to min_size, but the new size
   //should be smaller than min_size*2.
   //This "should" be successful since no other class is allocating
   //memory from the segment
   std::size_t shrunk_size;
   ret = managed_shm.allocation_command
      (boost::interprocess::shrink_in_place, min_size*2, min_size, shrunk_size, ptr);

   //Check invariants
   assert(ret.second == true);
   assert(ret.first == ptr);
   assert(shrunk_size <= min_size*2);
   assert(shrunk_size >= min_size);

   //Get free memory and compare
   std::size_t free_memory_after_shrinking = managed_shm.get_free_memory();
   assert(free_memory_after_shrinking > free_memory_after_expansion);

   //Deallocate the buffer
   managed_shm.deallocate(ptr);
   return 0;
}
```

`allocation_command` is a very powerful function that can lead to important performance gains. It's specially useful when programming vector-like data structures where the programmer can minimize both the number of allocation requests and the memory waste.

Opening managed shared memory and mapped files with Copy On Write or Read Only modes

When mapping a memory segment based on shared memory or files, there is an option to open them using **open_copy_on_write** option. This option is similar to `open_only` but every change the programmer does with this managed segment is kept private to this process and is not translated to the underlying device (shared memory or file).

The underlying shared memory or file is opened as read-only so several processes can share an initial managed segment and make private changes to it. If many processes open a managed segment in copy on write mode and not modified pages from the managed segment will be shared between all those processes, with considerable memory savings.

Opening managed shared memory and mapped files with **open_read_only** maps the underlying device in memory with **read-only** attributes. This means that any attempt to write that memory, either creating objects or locking any mutex might result in an page-fault error (and thus, program termination) from the OS. Read-only mode opens the underlying device (shared memory, file...) in read-only mode and can result in considerable memory savings if several processes just want to process a managed memory segment without modifying it. Read-only mode operations are limited:

• Read-only mode must be used only from managed classes. If the programmer obtains the segment manager and tries to use it directly it might result in an access violation. The reason for this is that the segment manager is placed in the underlying device and does not nothing about the mode it's been mapped in memory.

• Only const member functions from managed segments should be used.

• Additionally, the `find<>` member function avoids using internal locks and can be used to look for named and unique objects.

Here is an example that shows the use of these two open modes:

```cpp
#include <boost/interprocess/managed_mapped_file.hpp>
#include <fstream> //std::fstream
#include <iterator>//std::distance


int main()
{
   using namespace boost::interprocess;

   //Define file names
   const char *ManagedFile  = "MyManagedFile";
   const char *ManagedFile2 = "MyManagedFile2";

   //Try to erase any previous managed segment with the same name
   file_mapping::remove(ManagedFile);
   file_mapping::remove(ManagedFile2);
   remove_file_on_destroy destroyer1(ManagedFile);
   remove_file_on_destroy destroyer2(ManagedFile2);

   {
      //Create an named integer in a managed mapped file
      managed_mapped_file managed_file(create_only, ManagedFile, 65536);
      managed_file.construct<int>("MyInt")(0u);

      //Now create a copy on write version
      managed_mapped_file managed_file_cow(open_copy_on_write, ManagedFile);

      //Erase the int and create a new one
      if(!managed_file_cow.destroy<int>("MyInt"))
         throw int(0);
      managed_file_cow.construct<int>("MyInt2");

      //Check changes
      if(managed_file_cow.find<int>("MyInt").first && !managed_file_cow.find<int>("MyInt2").first)
         throw int(0);

      //Check the original is intact
      if(!managed_file.find<int>("MyInt").first && managed_file.find<int>("MyInt2").first)
         throw int(0);

      {  //Dump the modified copy on write segment to a file
         std::fstream file(ManagedFile2, std::ios_base::out | std::ios_base::binary);
         if(!file)
            throw int(0);
         file.write(static_cast<const char *>(managed_file_cow.get_address()), man↵
aged_file_cow.get_size());
      }

      //Now open the modified file and test changes
      managed_mapped_file managed_file_cow2(open_only, ManagedFile2);
      if(managed_file_cow2.find<int>("MyInt").first && !managed_file_cow2.find<int>("My↵
Int2").first)
         throw int(0);
   }
   {
      //Now create a read-only version
      managed_mapped_file managed_file_ro(open_read_only, ManagedFile);

      //Check the original is intact
      if(!managed_file_ro.find<int>("MyInt").first && managed_file_ro.find<int>("MyInt2").first)
         throw int(0);

      //Check the number of named objects using the iterators
```

```
      if(std::distance(managed_file_ro.named_begin(),  managed_file_ro.named_end())  != 1 &&
         std::distance(managed_file_ro.unique_begin(), managed_file_ro.unique_end()) != 0 )
         throw int(0);
   }
   return 0;
}
```

Managed Heap Memory And Managed External Buffer

Boost.Interprocess offers managed shared memory between processes using `managed_shared_memory` or `managed_mapped_file`. Two processes just map the same the memory mappable resource and read from and write to that object.

Many times, we don't want to use that shared memory approach and we prefer to send serialized data through network, local socket or message queues. Serialization can be done through **Boost.Serialization** or similar library. However, if two processes share the same ABI (application binary interface), we could use the same object and container construction capabilities of `managed_shared_memory` or `managed_heap_memory` to build all the information in a single buffer that will be sent, for example, though message queues. The receiver would just copy the data to a local buffer, and it could read or modify it directly without deserializing the data . This approach can be much more efficient that a complex serialization mechanism.

Applications for **Boost.Interprocess** services using non-shared memory buffers:

- Create and use STL compatible containers and allocators, in systems where dynamic memory is not recommendable.

- Build complex, easily serializable databases in a single buffer:

  - To share data between threads

  - To save and load information from/to files.

- Duplicate information (containers, allocators, etc...) just copying the contents of one buffer to another one.

- Send complex information and objects/databases using serial/inter-process/network communications.

To help with this management, **Boost.Interprocess** provides two useful classes, `basic_managed_heap_memory` and `basic_managed_external_buffer`:

Managed External Buffer: Constructing all Boost.Interprocess objects in a user provided buffer

Sometimes, the user wants to create simple objects, STL compatible containers, STL compatible strings and more, all in a single buffer. This buffer could be a big static buffer, a memory-mapped auxiliary device or any other user buffer.

This would allow an easy serialization and we-ll just need to copy the buffer to duplicate all the objects created in the original buffer, including complex objects like maps, lists.... **Boost.Interprocess** offers managed memory segment classes to handle user provided buffers that allow the same functionality as shared memory classes:

```
//Named object creation managed memory segment
//All objects are constructed in a user provided buffer
template <
            class CharType,
            class MemoryAlgorithm,
            template<class IndexConfig> class IndexType
          >
class basic_managed_external_buffer;

//Named object creation managed memory segment
//All objects are constructed in a user provided buffer
//   Names are c-strings,
//   Default memory management algorithm
//    (rbtree_best_fit with no mutexes and relative pointers)
//   Name-object mappings are stored in the default index type (flat_map)
typedef basic_managed_external_buffer <
   char,
   rbtree_best_fit<null_mutex_family, offset_ptr<void> >,
   flat_map_index
   >  managed_external_buffer;

//Named object creation managed memory segment
//All objects are constructed in a user provided buffer
//   Names are wide-strings,
//   Default memory management algorithm
//    (rbtree_best_fit with no mutexes and relative pointers)
//   Name-object mappings are stored in the default index type (flat_map)
typedef basic_managed_external_buffer<
   wchar_t,
   rbtree_best_fit<null_mutex_family, offset_ptr<void> >,
   flat_map_index
   >  wmanaged_external_buffer;
```

To use a managed external buffer, you must include the following header:

```
#include <boost/interprocess/managed_external_buffer.hpp>
```

Let's see an example of the use of managed_external_buffer:

```cpp
#include <boost/interprocess/managed_external_buffer.hpp>
#include <boost/interprocess/allocators/allocator.hpp>
#include <boost/interprocess/containers/list.hpp>
#include <cstring>
#include <boost/aligned_storage.hpp>

int main()
{
   using namespace boost::interprocess;

   //Create the static memory who will store all objects
   const int memsize = 65536;

   static boost::aligned_storage<memsize>::type static_buffer;

   //This managed memory will construct objects associated with
   //a wide string in the static buffer
   wmanaged_external_buffer objects_in_static_memory
      (create_only, &static_buffer, memsize);

   //We optimize resources to create 100 named objects in the static buffer
   objects_in_static_memory.reserve_named_objects(100);

   //Alias an integer node allocator type
   //This allocator will allocate memory inside the static buffer
   typedef allocator<int, wmanaged_external_buffer::segment_manager>
      allocator_t;

   //Alias a STL compatible list to be constructed in the static buffer
   typedef list<int, allocator_t>    MyBufferList;

   //The list must be initialized with the allocator
   //All objects created with objects_in_static_memory will
   //be stored in the static_buffer!
   MyBufferList *list = objects_in_static_memory.construct<MyBufferList>(L"MyList")
                           (objects_in_static_memory.get_segment_manager());
   //Since the allocation algorithm from wmanaged_external_buffer uses relative
   //pointers and all the pointers constructed int the static memory point
   //to objects in the same segment,  we can create another static buffer
   //from the first one and duplicate all the data.
   static boost::aligned_storage<memsize>::type static_buffer2;
   std::memcpy(&static_buffer2, &static_buffer, memsize);

   //Now open the duplicated managed memory passing the memory as argument
   wmanaged_external_buffer objects_in_static_memory2
      (open_only, &static_buffer2, memsize);

   //Check that "MyList" has been duplicated in the second buffer
   if(!objects_in_static_memory2.find<MyBufferList>(L"MyList").first)
      return 1;

   //Destroy the lists from the static buffers
   objects_in_static_memory.destroy<MyBufferList>(L"MyList");
   objects_in_static_memory2.destroy<MyBufferList>(L"MyList");
   return 0;
}
```

Boost.Interprocess STL compatible allocators can also be used to place STL compatible containers in the user segment.

basic_managed_external_buffer can be also useful to build small databases for embedded systems limiting the size of the used memory to a predefined memory chunk, instead of letting the database fragment the heap memory.

Managed Heap Memory: Boost.Interprocess machinery in heap memory

The use of heap memory (new/delete) to obtain a buffer where the user wants to store all his data is very common, so **Boost.Interprocess** provides some specialized classes that work exclusively with heap memory.

These are the classes:

```cpp
//Named object creation managed memory segment
//All objects are constructed in a single buffer allocated via new[]
template <
            class CharType,
            class MemoryAlgorithm,
            template<class IndexConfig> class IndexType
         >
class basic_managed_heap_memory;

//Named object creation managed memory segment
//All objects are constructed in a single buffer allocated via new[]
//    Names are c-strings,
//    Default memory management algorithm
//     (rbtree_best_fit with no mutexes and relative pointers)
//    Name-object mappings are stored in the default index type (flat_map)
typedef basic_managed_heap_memory <
   char,
   rbtree_best_fit<null_mutex_family>,
   flat_map_index
   >  managed_heap_memory;

//Named object creation managed memory segment
//All objects are constructed in a single buffer allocated via new[]
//    Names are wide-strings,
//    Default memory management algorithm
//     (rbtree_best_fit with no mutexes and relative pointers)
//    Name-object mappings are stored in the default index type (flat_map)
typedef basic_managed_heap_memory<
   wchar_t,
   rbtree_best_fit<null_mutex_family>,
   flat_map_index
   >  wmanaged_heap_memory;
```

To use a managed heap memory, you must include the following header:

```cpp
#include <boost/interprocess/managed_heap_memory.hpp>
```

The use is exactly the same as `basic_managed_external_buffer`, except that memory is created by the managed memory segment itself using dynamic (new/delete) memory.

basic_managed_heap_memory also offers a `grow(std::size_t extra_bytes)` function that tries to resize internal heap memory so that we have room for more objects. But **be careful**, if memory is reallocated, the old buffer will be copied into the new one so all the objects will be binary-copied to the new buffer. To be able to use this function, all pointers constructed in the heap buffer that point to objects in the heap buffer must be relative pointers (for example `offset_ptr`). Otherwise, the result is undefined. Here is an example:

```cpp
#include <boost/interprocess/containers/list.hpp>
#include <boost/interprocess/managed_heap_memory.hpp>
#include <boost/interprocess/allocators/allocator.hpp>
#include <cstddef>

using namespace boost::interprocess;
typedef list<int, allocator<int, managed_heap_memory::segment_manager> >
   MyList;

int main ()
{
   //We will create a buffer of 1000 bytes to store a list
   managed_heap_memory heap_memory(1000);

   MyList * mylist = heap_memory.construct<MyList>("MyList")
                        (heap_memory.get_segment_manager());

   //Obtain handle, that identifies the list in the buffer
   managed_heap_memory::handle_t list_handle = heap_memory.get_handle_from_address(mylist);

   //Fill list until there is no more memory in the buffer
   try{
      while(1) {
         mylist->insert(mylist->begin(), 0);
      }
   }
   catch(const bad_alloc &){
      //memory is full
   }
   //Let's obtain the size of the list
   std::size_t old_size = mylist->size();

   //To make the list bigger, let's increase the heap buffer
   //in 1000 bytes more.
   heap_memory.grow(1000);

   //If memory has been reallocated, the old pointer is invalid, so
   //use previously obtained handle to find the new pointer.
   mylist = static_cast<MyList *>
              (heap_memory.get_address_from_handle(list_handle));

   //Fill list until there is no more memory in the buffer
   try{
      while(1) {
         mylist->insert(mylist->begin(), 0);
      }
   }
   catch(const bad_alloc &){
      //memory is full
   }

   //Let's obtain the new size of the list
   std::size_t new_size = mylist->size();

   assert(new_size > old_size);
```

```
   //Destroy list
   heap_memory.destroy_ptr(mylist);

   return 0;
}
```

Differences between managed memory segments

All managed memory segments have similar capabilities (memory allocation inside the memory segment, named object construction...), but there are some remarkable differences between **managed_shared_memory**, **managed_mapped_file** and **managed_heap_memory**, **managed_external_file**.

- Default specializations of managed shared memory and mapped file use process-shared mutexes. Heap memory and external buffer have no internal synchronization by default. The cause is that the first two are thought to be shared between processes (although memory mapped files could be used just to obtain a persistent object data-base for a process) whereas the last two are thought to be used inside one process to construct a serialized named object data-base that can be sent though serial interprocess communications (like message queues, localhost network...).

- The first two create a system-global object (a shared memory object or a file) shared by several processes, whereas the last two are objects that don't create system-wide resources.

Example: Serializing a database through the message queue

To see the utility of managed heap memory and managed external buffer classes, the following example shows how a message queue can be used to serialize a whole database constructed in a memory buffer using **Boost.Interprocess**, send the database through a message queue and duplicated in another buffer:

```cpp
//This test creates a in memory data-base using Interprocess machinery and
//serializes it through a message queue. Then rebuilds the data-base in
//another buffer and checks it against the original data-base
bool test_serialize_db()
{
   //Typedef data to create a Interprocess map
   typedef std::pair<const std::size_t, std::size_t> MyPair;
   typedef std::less<std::size_t>   MyLess;
   typedef node_allocator<MyPair, managed_external_buffer::segment_manager>
      node_allocator_t;
   typedef map<std::size_t,
               std::size_t,
               std::less<std::size_t>,
               node_allocator_t>
               MyMap;

   //Some constants
   const std::size_t BufferSize  = 65536;
   const std::size_t MaxMsgSize  = 100;

   //Allocate a memory buffer to hold the destiny database using vector<char>
   std::vector<char> buffer_destiny(BufferSize, 0);

   message_queue::remove(test::get_process_id_name());
   {
      //Create the message-queues
      message_queue mq1(create_only, test::get_process_id_name(), 1, MaxMsgSize);

      //Open previously created message-queue simulating other process
      message_queue mq2(open_only, test::get_process_id_name());

      //A managed heap memory to create the origin database
      managed_heap_memory db_origin(buffer_destiny.size());

      //Construct the map in the first buffer
      MyMap *map1 = db_origin.construct<MyMap>("MyMap")
                                    (MyLess(),
                                    db_origin.get_segment_manager());
      if(!map1)
         return false;

      //Fill map1 until is full
      try{
         std::size_t i = 0;
         while(1){
            (*map1)[i] = i;
            ++i;
         }
      }
      catch(boost::interprocess::bad_alloc &){}

      //Data control data sending through the message queue
      std::size_t sent = 0;
      std::size_t recvd = 0;
      std::size_t total_recvd = 0;
      unsigned int priority;

      //Send whole first buffer through the mq1, read it
      //through mq2 to the second buffer
      while(1){
         //Send a fragment of buffer1 through mq1
         std::size_t bytes_to_send = MaxMsgSize < (db_origin.get_size() - sent) ?
                                    MaxMsgSize : (db_origin.get_size() - sent);
```

```cpp
      mq1.send( &static_cast<char*>(db_origin.get_address())[sent]
            , bytes_to_send
            , 0);
      sent += bytes_to_send;
      //Receive the fragment through mq2 to buffer_destiny
      mq2.receive( &buffer_destiny[total_recvd]
               , BufferSize - recvd
               , recvd
               , priority);
      total_recvd += recvd;

      //Check if we have received all the buffer
      if(total_recvd == BufferSize){
         break;
      }
   }

   //The buffer will contain a copy of the original database
   //so let's interpret the buffer with managed_external_buffer
   managed_external_buffer db_destiny(open_only, &buffer_destiny[0], BufferSize);

   //Let's find the map
   std::pair<MyMap *, std::size_t> ret = db_destiny.find<MyMap>("MyMap");
   MyMap *map2 = ret.first;

   //Check if we have found it
   if(!map2){
      return false;
   }

   //Check if it is a single variable (not an array)
   if(ret.second != 1){
      return false;
   }

   //Now let's compare size
   if(map1->size() != map2->size()){
      return false;
   }

   //Now let's compare all db values
   for(std::size_t i = 0, num_elements = map1->size(); i < num_elements; ++i){
      if((*map1)[i] != (*map2)[i]){
         return false;
      }
   }

   //Destroy maps from db-s
```

```
        db_origin.destroy_ptr(map1);
        db_destiny.destroy_ptr(map2);
    }
    message_queue::remove(test::get_process_id_name());
    return true;
}
```

Allocators, containers and memory allocation algorithms

Introduction to Interprocess allocators

As seen, **Boost.Interprocess** offers raw memory allocation and object construction using managed memory segments (managed shared memory, managed mapped files...) and one of the first user requests is the use of containers in managed shared memories. To achieve this, **Boost.Interprocess** makes use of managed memory segment's memory allocation algorithms to build several memory allocation schemes, including general purpose and node allocators.

Boost.Interprocess STL compatible allocators are configurable via template parameters. Allocators define their `pointer` typedef based on the `void_pointer` typedef of the segment manager passed as template argument. When this `segment_manager::void_pointer` is a relative pointer, (for example, `offset_ptr<void>`) the user can place these allocators in memory mapped in different base addresses in several processes.

Properties of Boost.Interprocess allocators

Container allocators are normally default-constructible because the are stateless. `std::allocator` and **Boost.Pool's** `boost::pool_allocator/boost::fast_pool_allocator` are examples of default-constructible allocators.

On the other hand, **Boost.Interprocess** allocators need to allocate memory from a concrete memory segment and not from a system-wide memory source (like the heap). **Boost.Interprocess** allocators are **stateful**, which means that they must be configured to tell them where the shared memory or the memory mapped file is.

This information is transmitted at compile-time and run-time: The allocators receive a template parameter defining the type of the segment manager and their constructor receive a pointer to the segment manager of the managed memory segment where the user wants to allocate the values.

Boost.Interprocess allocators have **no default-constructors** and containers must be explicitly initialized with a configured allocator:

```
//The allocators must be templatized with the segment manager type
typedef any_interprocess_allocator
   <int, managed_shared_memory::segment_manager, ...> Allocator;

//The allocator must be constructed with a pointer to the segment manager
Allocator alloc_instance (segment.get_segment_manager(), ...);

//Containers must be initialized with a configured allocator
typedef my_list<int, Allocator> MyIntList;
MyIntList mylist(alloc_inst);

//This would lead to a compilation error, because
//the allocator has no default constructor
//MyIntList mylist;
```

Boost.Interprocess allocators also have a `get_segment_manager()` function that returns the underlying segment manager that they have received in the constructor:

```
Allocator::segment_manager s = alloc_instance.get_segment_manager();
AnotherType *a = s->construct<AnotherType>(anonymous_instance)(/*Parameters*/);
```

Swapping Boost.Interprocess allocators

When swapping STL containers, there is an active discussion on what to do with the allocators. Some STL implementations, for example Dinkumware from Visual .NET 2003, perform a deep swap of the whole container through a temporary when allocators are not equal. The proposed resolution to container swapping is that allocators should be swapped in a non-throwing way.

Unfortunately, this approach is not valid with shared memory. Using heap allocators, if Group1 of node allocators share a common segregated storage, and Group2 share another common segregated storage, a simple pointer swapping is needed to swap an allocator of Group1 and another allocator of Group2. But when the user wants to swap two shared memory allocators, each one placed in a different shared memory segment, this is not possible. As generally shared memory is mapped in different addresses in each process, a pointer placed in one segment can't point to any object placed in other shared memory segment, since in each process, the distance between the segments is different. However, if both shared memory allocators are in the same segment, a non-throwing swap is possible, just like heap allocators.

Until a final resolution is achieved. **Boost.Interprocess** allocators implement a non-throwing swap function that swaps internal pointers. If an allocator placed in a shared memory segment is swapped with other placed in a different shared memory segment, the result is undefined. But a crash is quite sure.

allocator: A general purpose allocator for managed memory segments

The `allocator` class defines an allocator class that uses the managed memory segment's algorithm to allocate and deallocate memory. This is achieved through the **segment manager** of the managed memory segment. This allocator is the equivalent for managed memory segments of the standard `std::allocator`. `allocator` is templatized with the allocated type, and the segment manager.

Equality: Two `allocator` instances constructed with the same segment manager compare equal. If an instance is created using copy constructor, that instance compares equal with the original one.

Allocation thread-safety: Allocation and deallocation are implemented as calls to the segment manager's allocation function so the allocator offers the same thread-safety as the segment manager.

To use `allocator` you must include the following header:

```
#include <boost/interprocess/allocators/allocator.hpp>
```

`allocator` has the following declaration:

```
namespace boost {
namespace interprocess {

template<class T, class SegmentManager>
class allocator;

}  //namespace interprocess {
}  //namespace boost {
```

The allocator just provides the needed typedefs and forwards all allocation and deallocation requests to the segment manager passed in the constructor, just like `std::allocator` forwards the requests to `operator new[]`.

Using `allocator` is straightforward:

```cpp
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/allocators/allocator.hpp>
#include <cassert>

using namespace boost::interprocess;

int main ()
{
   //Remove shared memory on construction and destruction
   struct shm_remove
   {
      shm_remove() { shared_memory_object::remove("MySharedMemory"); }
      ~shm_remove(){ shared_memory_object::remove("MySharedMemory"); }
   } remover;

   //Create shared memory
   managed_shared_memory segment(create_only,
                                 "MySharedMemory",  //segment name
                                 65536);

   //Create an allocator that allocates ints from the managed segment
   allocator<int, managed_shared_memory::segment_manager>
      allocator_instance(segment.get_segment_manager());

   //Copy constructed allocator is equal
   allocator<int, managed_shared_memory::segment_manager>
      allocator_instance2(allocator_instance);
   assert(allocator_instance2 == allocator_instance);

   //Allocate and deallocate memory for 100 ints
   allocator_instance2.deallocate(allocator_instance.allocate(100), 100);

   return 0;
}
```

Segregated storage node allocators

Variable size memory algorithms waste some space in management information for each allocation. Sometimes, usually for small objects, this is not acceptable. Memory algorithms can also fragment the managed memory segment under some allocation and deallocation schemes, reducing their performance. When allocating many objects of the same type, a simple segregated storage becomes a fast and space-friendly allocator, as explained in the **Boost.Pool** library.

Segregate storage node allocators allocate large memory chunks from a general purpose memory allocator and divide that chunk into several nodes. No bookkeeping information is stored in the nodes to achieve minimal memory waste: free nodes are linked using a pointer constructed in the memory of the node.

Boost.Interprocess offers 3 allocators based on this segregated storage algorithm: `node_allocator`, `private_node_allocator` and `cached_node_allocator`.

To know the details of the implementation of of the segregated storage pools see the Implementation of **Boost.Interprocess** segregated storage pools section.

Additional parameters and functions of segregated storage node allocators

`node_allocator`, `private_node_allocator` and `cached_node_allocator` implement the standard allocator interface and the functions explained in the Properties of Boost.Interprocess allocators.

All these allocators are templatized by 3 parameters:

• `class T`: The type to be allocated.

- `class SegmentManager`: The type of the segment manager that will be passed in the constructor.

- `std::size_t NodesPerChunk`: The number of nodes that a memory chunk will contain. This value will define the size of the memory the pool will request to the segment manager when the pool runs out of nodes. This parameter has a default value.

These allocators also offer the `deallocate_free_chunks()` function. This function will traverse all the memory chunks of the pool and will return to the managed memory segment the free chunks of memory. If this function is not used, deallocating the free chunks does not happen until the pool is destroyed so the only way to return memory allocated by the pool to the segment before destructing the pool is calling manually this function. This function is quite time-consuming because it has quadratic complexity (O(N^2)).

node_allocator: A process-shared segregated storage

For heap-memory node allocators (like **Boost.Pool's** `boost::fast_pool_allocator` usually a global, thread-shared singleton pool is used for each node size. This is not possible if you try to share a node allocator between processes. To achieve this sharing `node_allocator` uses the segment manager's unique type allocation service (see Unique instance construction section).

In the initialization, a `node_allocator` object searches this unique object in the segment. If it is not preset, it builds one. This way, all `node_allocator` objects built inside a memory segment share a unique memory pool.

The common segregated storage is not only shared between node_allocators of the same type, but it is also shared between all node allocators that allocate objects of the same size, for example, **node_allocator<uint32>** and **node_allocator<float32>**. This saves a lot of memory but also imposes an synchronization overhead for each node allocation.

The dynamically created common segregated storage integrates a reference count so that a `node_allocator` can know if any other `node_allocator` is attached to the same common segregated storage. When the last allocator attached to the pool is destroyed, the pool is destroyed.

Equality: Two `node_allocator` instances constructed with the same segment manager compare equal. If an instance is created using copy constructor, that instance compares equal with the original one.

Allocation thread-safety: Allocation and deallocation are implemented as calls to the shared pool. The shared pool offers the same synchronization guarantees as the segment manager.

To use `node_allocator`, you must include the following header:

```
#include <boost/interprocess/allocators/node_allocator.hpp>
```

`node_allocator` has the following declaration:

```
namespace boost {
namespace interprocess {

template<class T, class SegmentManager, std::size_t NodesPerChunk = ...>
class node_allocator;

}  //namespace interprocess {
}  //namespace boost {
```

An example using `node_allocator`:

```cpp
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/allocators/node_allocator.hpp>
#include <cassert>

using namespace boost::interprocess;

int main ()
{
   //Remove shared memory on construction and destruction
   struct shm_remove
   {
      shm_remove() { shared_memory_object::remove("MySharedMemory"); }
      ~shm_remove(){ shared_memory_object::remove("MySharedMemory"); }
   } remover;

   //Create shared memory
   managed_shared_memory segment(create_only,
                                 "MySharedMemory",  //segment name
                                 65536);

   //Create a node_allocator that allocates ints from the managed segment
   //The number of chunks per segment is the default value
   typedef node_allocator<int, managed_shared_memory::segment_manager>
      node_allocator_t;
   node_allocator_t allocator_instance(segment.get_segment_manager());

   //Create another node_allocator. Since the segment manager address
   //is the same, this node_allocator will be
   //attached to the same pool so "allocator_instance2" can deallocate
   //nodes allocated by "allocator_instance"
   node_allocator_t allocator_instance2(segment.get_segment_manager());

   //Create another node_allocator using copy-constructor. This
   //node_allocator will also be attached to the same pool
   node_allocator_t allocator_instance3(allocator_instance2);

   //All allocators are equal
   assert(allocator_instance == allocator_instance2);
   assert(allocator_instance2 == allocator_instance3);

   //So memory allocated with one can be deallocated with another
   allocator_instance2.deallocate(allocator_instance.allocate(1), 1);
   allocator_instance3.deallocate(allocator_instance2.allocate(1), 1);

   //The common pool will be destroyed here, since no allocator is
   //attached to the pool
   return 0;
}
```

private_node_allocator: a private segregated storage

As said, the node_allocator shares a common segregated storage between node_allocators that allocate objects of the same size and this optimizes memory usage. However, it needs a unique/named object construction feature so that this sharing can be possible. Also imposes a synchronization overhead per node allocation because of this share. Sometimes, the unique object service is not available (for example, when building index types to implement the named allocation service itself) or the synchronization overhead is not acceptable. Many times the programmer wants to make sure that the pool is destroyed when the allocator is destroyed, to free the memory as soon as possible.

So **private_node_allocator** uses the same segregated storage as node_allocator, but each **private_node_allocator** has its own segregated storage pool. No synchronization is used when allocating nodes, so there is far less overhead for an operation that usually involves just a few pointer operations when allocating and deallocating a node.

Equality: Two `private_node_allocator` instances **never** compare equal. Memory allocated with one allocator **can't** be deallocated with another one.

Allocation thread-safety: Allocation and deallocation are **not** thread-safe.

To use `private_node_allocator`, you must include the following header:

```
#include <boost/interprocess/allocators/private_node_allocator.hpp>
```

`private_node_allocator` has the following declaration:

```
namespace boost {
namespace interprocess {

template<class T, class SegmentManager, std::size_t NodesPerChunk = ...>
class private_node_allocator;

}  //namespace interprocess {
}  //namespace boost {
```

An example using `private_node_allocator`:

```cpp
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/allocators/private_node_allocator.hpp>
#include <cassert>

using namespace boost::interprocess;

int main ()
{
   //Remove shared memory on construction and destruction
   struct shm_remove
   {
      shm_remove() { shared_memory_object::remove("MySharedMemory"); }
      ~shm_remove(){ shared_memory_object::remove("MySharedMemory"); }
   } remover;

   //Create shared memory
   managed_shared_memory segment(create_only,
                                 "MySharedMemory",  //segment name
                                 65536);

   //Create a private_node_allocator that allocates ints from the managed segment
   //The number of chunks per segment is the default value
   typedef private_node_allocator<int, managed_shared_memory::segment_manager>
      private_node_allocator_t;
   private_node_allocator_t allocator_instance(segment.get_segment_manager());

   //Create another private_node_allocator.
   private_node_allocator_t allocator_instance2(segment.get_segment_manager());

   //Although the segment manager address
   //is the same, this private_node_allocator will have its own pool so
   //"allocator_instance2" CAN'T deallocate nodes allocated by "allocator_instance".
   //"allocator_instance2" is NOT equal to "allocator_instance"
   assert(allocator_instance != allocator_instance2);

   //Create another node_allocator using copy-constructor.
   private_node_allocator_t allocator_instance3(allocator_instance2);

   //This allocator is also unequal to allocator_instance2
   assert(allocator_instance2 != allocator_instance3);

   //Pools are destroyed with the allocators
   return 0;
}
```

cached_node_allocator: caching nodes to avoid overhead

The total node sharing of `node_allocator` can impose a high overhead for some applications and the minimal synchronization overhead of `private_node_allocator` can impose a unacceptable memory waste for other applications.

To solve this, **Boost.Interprocess** offers an allocator, `cached_node_allocator`, that allocates nodes from the common pool but caches some of them privately so that following allocations have no synchronization overhead. When the cache is full, the allocator returns some cached nodes to the common pool, and those will be available to other allocators.

Equality: Two `cached_node_allocator` instances constructed with the same segment manager compare equal. If an instance is created using copy constructor, that instance compares equal with the original one.

Allocation thread-safety: Allocation and deallocation are **not** thread-safe.

To use `cached_node_allocator`, you must include the following header:

```
#include <boost/interprocess/allocators/cached_node_allocator.hpp>
```

cached_node_allocator has the following declaration:

```
namespace boost {
namespace interprocess {

template<class T, class SegmentManager, std::size_t NodesPerChunk = ...>
class cached_node_allocator;

}  //namespace interprocess {
}  //namespace boost {
```

A cached_node_allocator instance and a node_allocator instance share the same pool if both instances receive the same template parameters. This means that nodes returned to the shared pool by one of them can be reused by the other. Please note that this does not mean that both allocators compare equal, this is just information for programmers that want to maximize the use of the pool.

cached_node_allocator, offers additional functions to control the cache (the cache can be controlled per instance):

• void set_max_cached_nodes(std::size_t n): Sets the maximum cached nodes limit. If cached nodes reach the limit, some are returned to the shared pool.

• std::size_t get_max_cached_nodes() const: Returns the maximum cached nodes limit.

• void deallocate_cache(): Returns the cached nodes to the shared pool.

An example using cached_node_allocator:

```cpp
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/allocators/cached_node_allocator.hpp>
#include <cassert>

using namespace boost::interprocess;

int main ()
{
   //Remove shared memory on construction and destruction
   struct shm_remove
   {
      shm_remove() { shared_memory_object::remove("MySharedMemory"); }
      ~shm_remove(){ shared_memory_object::remove("MySharedMemory"); }
   } remover;

   //Create shared memory
   managed_shared_memory segment(create_only,
                                 "MySharedMemory",  //segment name
                                 65536);

   //Create a cached_node_allocator that allocates ints from the managed segment
   //The number of chunks per segment is the default value
   typedef cached_node_allocator<int, managed_shared_memory::segment_manager>
      cached_node_allocator_t;
   cached_node_allocator_t allocator_instance(segment.get_segment_manager());

   //The max cached nodes are configurable per instance
   allocator_instance.set_max_cached_nodes(3);

   //Create another cached_node_allocator. Since the segment manager address
   //is the same, this cached_node_allocator will be
   //attached to the same pool so "allocator_instance2" can deallocate
   //nodes allocated by "allocator_instance"
   cached_node_allocator_t allocator_instance2(segment.get_segment_manager());

   //The max cached nodes are configurable per instance
   allocator_instance2.set_max_cached_nodes(5);

   //Create another cached_node_allocator using copy-constructor. This
   //cached_node_allocator will also be attached to the same pool
   cached_node_allocator_t allocator_instance3(allocator_instance2);

   //We can clear the cache
   allocator_instance3.deallocate_cache();

   //All allocators are equal
   assert(allocator_instance == allocator_instance2);
   assert(allocator_instance2 == allocator_instance3);

   //So memory allocated with one can be deallocated with another
   allocator_instance2.deallocate(allocator_instance.allocate(1), 1);
   allocator_instance3.deallocate(allocator_instance2.allocate(1), 1);

   //The common pool will be destroyed here, since no allocator is
   //attached to the pool
   return 0;
}
```

Adaptive pool node allocators

Node allocators based on simple segregated storage algorithm are both space-efficient and fast but they have a problem: they only can grow. Every allocated node avoids any payload to store additional data and that leads to the following limitation: when a node

is deallocated, it's stored in a free list of nodes but memory is not returned to the segment manager so a deallocated node can be only reused by other containers using the same node pool.

This behaviour can be problematic if several containers use `boost::interprocess::node_allocator` to temporarily allocate a lot of objects but they end storing a few of them: the node pool will be full of nodes that won't be reused wasting memory from the segment.

Adaptive pool based allocators trade some space (the overhead can be as low as 1%) and performance (acceptable for many applications) with the ability to return free chunks of nodes to the memory segment, so that they can be used by any other container or managed object construction. To know the details of the implementation of of "adaptive pools" see the Implementation of **Boost.Intrusive** adaptive pools section.

Like with segregated storage based node allocators, Boost.Interprocess offers 3 new allocators: `adaptive_pool`, `private_adaptive_pool`, `cached_adaptive_pool`.

Additional parameters and functions of adaptive pool node allocators

`adaptive_pool`, `private_adaptive_pool` and `cached_adaptive_pool` implement the standard allocator interface and the functions explained in the Properties of Boost.Interprocess allocators.

All these allocators are templatized by 4 parameters:

- `class T`: The type to be allocated.

- `class SegmentManager`: The type of the segment manager that will be passed in the constructor.

- `std::size_t NodesPerChunk`: The number of nodes that a memory chunk will contain. This value will define the size of the memory the pool will request to the segment manager when the pool runs out of nodes. This parameter has a default value.

- `std::size_t MaxFreeChunks`: The maximum number of free chunks that the pool will hold. If this limit is reached the pool returns the chunks to the segment manager. This parameter has a default value.

These allocators also offer the `deallocate_free_chunks()` function. This function will traverse all the memory chunks of the pool and will return to the managed memory segment the free chunks of memory. This function is much faster than for segregated storage allocators, because the adaptive pool algorithm offers constant-time access to free chunks.

adaptive_pool: a process-shared adaptive pool

Just like `node_allocator` a global, process-thread pool is used for each node size. In the initialization, `adaptive_pool` searches the pool in the segment. If it is not preset, it builds one. The adaptive pool, is created using a unique name. The adaptive pool it is also shared between all node_allocators that allocate objects of the same size, for example, **adaptive_pool<uint32>** and **adaptive_pool<float32>**.

The common adaptive pool is destroyed when all the allocators attached to the pool are destroyed.

Equality: Two `adaptive_pool` instances constructed with the same segment manager compare equal. If an instance is created using copy constructor, that instance compares equal with the original one.

Allocation thread-safety: Allocation and deallocation are implemented as calls to the shared pool. The shared pool offers the same synchronization guarantees as the segment manager.

To use `adaptive_pool`, you must include the following header:

```
#include <boost/interprocess/allocators/adaptive_pool.hpp>
```

`adaptive_pool` has the following declaration:

```
namespace boost {
namespace interprocess {

template<class T, class SegmentManager, std::size_t NodesPerChunk = ..., std::size_t Max↵
FreeChunks = ...>
class adaptive_pool;

}  //namespace interprocess {
}  //namespace boost {
```

An example using `adaptive_pool`:

```
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/allocators/adaptive_pool.hpp>
#include <cassert>

using namespace boost::interprocess;

int main ()
{
   //Remove shared memory on construction and destruction
   struct shm_remove
   {
      shm_remove() { shared_memory_object::remove("MySharedMemory"); }
      ~shm_remove(){ shared_memory_object::remove("MySharedMemory"); }
   } remover;

   //Create shared memory
   managed_shared_memory segment(create_only,
                                 "MySharedMemory",  //segment name
                                 65536);

   //Create a adaptive_pool that allocates ints from the managed segment
   //The number of chunks per segment is the default value
   typedef adaptive_pool<int, managed_shared_memory::segment_manager>
      adaptive_pool_t;
   adaptive_pool_t allocator_instance(segment.get_segment_manager());

   //Create another adaptive_pool. Since the segment manager address
   //is the same, this adaptive_pool will be
   //attached to the same pool so "allocator_instance2" can deallocate
   //nodes allocated by "allocator_instance"
   adaptive_pool_t allocator_instance2(segment.get_segment_manager());

   //Create another adaptive_pool using copy-constructor. This
   //adaptive_pool will also be attached to the same pool
   adaptive_pool_t allocator_instance3(allocator_instance2);

   //All allocators are equal
   assert(allocator_instance == allocator_instance2);
   assert(allocator_instance2 == allocator_instance3);

   //So memory allocated with one can be deallocated with another
   allocator_instance2.deallocate(allocator_instance.allocate(1), 1);
```

```
   allocator_instance3.deallocate(allocator_instance2.allocate(1), 1);

   //The common pool will be destroyed here, since no allocator is
   //attached to the pool
   return 0;
}
```

private_adaptive_pool: a private adaptive pool

Just like `private_node_allocator` owns a private segregated storage pool, `private_adaptive_pool` owns its own adaptive pool. If the user wants to avoid the excessive node allocation synchronization overhead in a container `private_adaptive_pool` is a good choice.

Equality: Two `private_adaptive_pool` instances **never** compare equal. Memory allocated with one allocator **can't** be deallocated with another one.

Allocation thread-safety: Allocation and deallocation are **not** thread-safe.

To use `private_adaptive_pool`, you must include the following header:

```
#include <boost/interprocess/allocators/private_adaptive_pool.hpp>
```

`private_adaptive_pool` has the following declaration:

```
namespace boost {
namespace interprocess {

template<class T, class SegmentManager, std::size_t NodesPerChunk = ..., std::size_t Max↵
FreeChunks = ...>
class private_adaptive_pool;

}  //namespace interprocess {
}  //namespace boost {
```

An example using `private_adaptive_pool`:

```cpp
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/allocators/private_adaptive_pool.hpp>
#include <cassert>

using namespace boost::interprocess;

int main ()
{
   //Remove shared memory on construction and destruction
   struct shm_remove
   {
      shm_remove() { shared_memory_object::remove("MySharedMemory"); }
      ~shm_remove(){ shared_memory_object::remove("MySharedMemory"); }
   } remover;

   //Create shared memory
   managed_shared_memory segment(create_only,
                                 "MySharedMemory",  //segment name
                                 65536);

   //Create a private_adaptive_pool that allocates ints from the managed segment
   //The number of chunks per segment is the default value
   typedef private_adaptive_pool<int, managed_shared_memory::segment_manager>
        private_adaptive_pool_t;
   private_adaptive_pool_t allocator_instance(segment.get_segment_manager());

   //Create another private_adaptive_pool.
   private_adaptive_pool_t allocator_instance2(segment.get_segment_manager());

   //Although the segment manager address
   //is the same, this private_adaptive_pool will have its own pool so
   //"allocator_instance2" CAN'T deallocate nodes allocated by "allocator_instance".
   //"allocator_instance2" is NOT equal to "allocator_instance"
   assert(allocator_instance != allocator_instance2);

   //Create another adaptive_pool using copy-constructor.
   private_adaptive_pool_t allocator_instance3(allocator_instance2);

   //This allocator is also unequal to allocator_instance2
   assert(allocator_instance2 != allocator_instance3);

   //Pools are destroyed with the allocators
   return 0;
}
```

cached_adaptive_pool: Avoiding synchronization overhead

Adaptive pools have also a cached version. In this allocator the allocator caches some nodes to avoid the synchronization and bookkeeping overhead of the shared adaptive pool. `cached_adaptive_pool` allocates nodes from the common adaptive pool but caches some of them privately so that following allocations have no synchronization overhead. When the cache is full, the allocator returns some cached nodes to the common pool, and those will be available to other `cached_adaptive_pools` or `adaptive_pools` of the same managed segment.

Equality: Two `cached_adaptive_pool` instances constructed with the same segment manager compare equal. If an instance is created using copy constructor, that instance compares equal with the original one.

Allocation thread-safety: Allocation and deallocation are **not** thread-safe.

To use `cached_adaptive_pool`, you must include the following header:

```
#include <boost/interprocess/allocators/cached_adaptive_pool.hpp>
```

cached_adaptive_pool has the following declaration:

```
namespace boost {
namespace interprocess {

template<class T, class SegmentManager, std::size_t NodesPerChunk = ..., std::size_t MaxFreeN↵
odes = ...>
class cached_adaptive_pool;

}  //namespace interprocess {
}  //namespace boost {
```

A cached_adaptive_pool instance and an adaptive_pool instance share the same pool if both instances receive the same template parameters. This means that nodes returned to the shared pool by one of them can be reused by the other. Please note that this does not mean that both allocators compare equal, this is just information for programmers that want to maximize the use of the pool.

cached_adaptive_pool, offers additional functions to control the cache (the cache can be controlled per instance):

• void set_max_cached_nodes(std::size_t n): Sets the maximum cached nodes limit. If cached nodes reach the limit, some are returned to the shared pool.

• std::size_t get_max_cached_nodes() const: Returns the maximum cached nodes limit.

• void deallocate_cache(): Returns the cached nodes to the shared pool.

An example using cached_adaptive_pool:

```cpp
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/allocators/cached_adaptive_pool.hpp>
#include <cassert>

using namespace boost::interprocess;

int main ()
{
   //Remove shared memory on construction and destruction
   struct shm_remove
   {
      shm_remove() { shared_memory_object::remove("MySharedMemory"); }
      ~shm_remove(){ shared_memory_object::remove("MySharedMemory"); }
   } remover;

   //Create shared memory
   managed_shared_memory segment(create_only,
                                 "MySharedMemory",  //segment name
                                 65536);

   //Create a cached_adaptive_pool that allocates ints from the managed segment
   //The number of chunks per segment is the default value
   typedef cached_adaptive_pool<int, managed_shared_memory::segment_manager>
      cached_adaptive_pool_t;
   cached_adaptive_pool_t allocator_instance(segment.get_segment_manager());

   //The max cached nodes are configurable per instance
   allocator_instance.set_max_cached_nodes(3);

   //Create another cached_adaptive_pool. Since the segment manager address
   //is the same, this cached_adaptive_pool will be
   //attached to the same pool so "allocator_instance2" can deallocate
   //nodes allocated by "allocator_instance"
   cached_adaptive_pool_t allocator_instance2(segment.get_segment_manager());

   //The max cached nodes are configurable per instance
   allocator_instance2.set_max_cached_nodes(5);

   //Create another cached_adaptive_pool using copy-constructor. This
   //cached_adaptive_pool will also be attached to the same pool
   cached_adaptive_pool_t allocator_instance3(allocator_instance2);

   //We can clear the cache
   allocator_instance3.deallocate_cache();

   //All allocators are equal
   assert(allocator_instance == allocator_instance2);
   assert(allocator_instance2 == allocator_instance3);

   //So memory allocated with one can be deallocated with another
   allocator_instance2.deallocate(allocator_instance.allocate(1), 1);
```

```
    allocator_instance3.deallocate(allocator_instance2.allocate(1), 1);

    //The common pool will be destroyed here, since no allocator is
    //attached to the pool
    return 0;
}
```

Interprocess and containers in managed memory segments

Container requirements for Boost.Interprocess allocators

Boost.Interprocess STL compatible allocators offer a STL compatible allocator interface and if they define their internal **pointer** typedef as a relative pointer, they can sbe used to place STL containers in shared memory, memory mapped files or in a user defined memory segment.

However, as Scott Meyers mentions in his Effective STL book, Item 10, *"Be aware of allocator conventions and restrictions"*:

• *"the Standard explicitly allows library implementers to assume that every allocator's pointer typedef is a synonym for T*"*

• *"the Standard says that an implementation of the STL is permitted to assume that all allocator objects of the same type are equivalent and always compare equal"*

Obviously, if any STL implementation ignores pointer typedefs, no smart pointer can be used as allocator::pointer. If STL implementations assume all allocator objects of the same type compare equal, it will assume that two allocators, each one allocating from a different memory pool are equal, which is a complete disaster.

STL containers that we want to place in shared memory or memory mapped files with **Boost.Interprocess** can't make any of these assumptions, so:

• STL containers may not assume that memory allocated with an allocator can be deallocated with other allocators of the same type. All allocators objects must compare equal only if memory allocated with one object can be deallocated with the other one, and this can only tested with operator==() at run-time.

• Containers' internal pointers should be of the type allocator::pointer and containers may not assume allocator::pointer is a raw pointer.

• All objects must be constructed-destroyed via allocator::construct and allocator::destroy functions.

STL containers in managed memory segments

Unfortunately, many STL implementations use raw pointers for internal data and ignore allocator pointer typedefs and others suppose at some point that the allocator::typedef is T. **This is because in practice, there wasn't need of allocators with a pointer typedef different from T** for pooled/node memory allocators.

Until STL implementations handle allocator::pointer typedefs in a generic way, **Boost.Interprocess** offers the following classes:

• **boost:interprocess::vector** is the implementation of `std::vector` ready to be used in managed memory segments like shared memory. To use it include:

```
#include <boost/interprocess/containers/vector.hpp>
```

• **boost:interprocess::deque** is the implementation of `std::deque` ready to be used in managed memory segments like shared memory. To use it include:

```
#include <boost/interprocess/containers/deque.hpp>
```

• `list` is the implementation of `std::list` ready to be used in managed memory segments like shared memory. To use it include:

```
#include <boost/interprocess/containers/list.hpp>
```

- `slist` is the implementation of SGI's `slist` container (singly linked list) ready to be used in managed memory segments like shared memory. To use it include:

```
#include <boost/interprocess/containers/slist.hpp>
```

- `set`/ `multiset`/ `map`/ `multimap` family is the implementation of std::set/multiset/map/multimap family ready to be used in managed memory segments like shared memory. To use them include:

```
#include <boost/interprocess/containers/set.hpp>
#include <boost/interprocess/containers/map.hpp>
```

- `flat_set`/ `flat_multiset`/ `flat_map`/ `flat_multimap` classes are the adaptation and extension of Andrei Alexandrescu's famous AssocVector class from Loki library, ready for the shared memory. These classes offer the same functionality as `std::set/multiset/map/multimap` implemented with an ordered vector, which has faster lookups than the standard ordered associative containers based on red-black trees, but slower insertions. To use it include:

```
#include <boost/interprocess/containers/flat_set.hpp>
#include <boost/interprocess/containers/flat_map.hpp>
```

- `basic_string` is the implementation of `std::basic_string` ready to be used in managed memory segments like shared memory. It's implemented using a vector-like contiguous storage, so it has fast c string conversion and can be used with the vectorstream iostream formatting classes. To use it include:

```
#include <boost/interprocess/containers/string.hpp>
```

All these containers have the same default arguments as standard containers and they can be used with other, non **Boost.Interprocess** allocators (std::allocator, or boost::pool_allocator, for example).

To place any of these containers in managed memory segments, we must define the allocator template parameter with a **Boost.Interprocess** allocator so that the container allocates the values in the managed memory segment. To place the container itself in shared memory, we construct it in the managed memory segment just like any other object with **Boost.Interprocess**:

```cpp
#include <boost/interprocess/containers/vector.hpp>
#include <boost/interprocess/allocators/allocator.hpp>
#include <boost/interprocess/managed_shared_memory.hpp>

int main ()
{
   using namespace boost::interprocess;
   //Remove shared memory on construction and destruction
   struct shm_remove
   {
      shm_remove() { shared_memory_object::remove("MySharedMemory"); }
      ~shm_remove(){ shared_memory_object::remove("MySharedMemory"); }
   } remover;

   //A managed shared memory where we can construct objects
   //associated with a c-string
   managed_shared_memory segment(create_only,
                                 "MySharedMemory",  //segment name
                                 65536);

   //Alias an STL-like allocator of ints that allocates ints from the segment
   typedef allocator<int, managed_shared_memory::segment_manager>
      ShmemAllocator;

   //Alias a vector that uses the previous STL-like allocator
   typedef vector<int, ShmemAllocator> MyVector;

   int initVal[]        = {0, 1, 2, 3, 4, 5, 6 };
   const int *begVal    = initVal;
   const int *endVal    = initVal + sizeof(initVal)/sizeof(initVal[0]);

   //Initialize the STL-like allocator
   const ShmemAllocator alloc_inst (segment.get_segment_manager());

   //Construct the vector in the shared memory segment with the STL-like allocator
   //from a range of iterators
   MyVector *myvector =
      segment.construct<MyVector>
         ("MyVector")/*object name*/
         (begVal      /*first ctor parameter*/,
          endVal      /*second ctor parameter*/,
          alloc_inst /*third ctor parameter*/);

   //Use vector as your want
   std::sort(myvector->rbegin(), myvector->rend());
   // . . .
   //When done, destroy and delete vector from the segment
   segment.destroy<MyVector>("MyVector");
   return 0;
}
```

These containers also show how easy is to create/modify an existing container making possible to place it in shared memory.

Where is this being allocated?

Boost.Interprocess containers are placed in shared memory/memory mapped files, etc... using two mechanisms **at the same time**:

- **Boost.Interprocess** `construct<>`, `find_or_construct<>`... functions. These functions place a C++ object in the shared memory/memory mapped file. But this places only the object, but **not** the memory that this object may allocate dynamically.

- Shared memory allocators. These allow allocating shared memory/memory mapped file portions so that containers can allocate dynamically fragments of memory to store newly inserted elements.

This means that to place any **Boost.Interprocess** container (including **Boost.Interprocess** strings) in shared memory or memory mapped files, containers **must**:

• Define their template allocator parameter to a **Boost.Interprocess** allocator.

• Every container constructor must take the **Boost.Interprocess** allocator as parameter.

• You must use construct<>/find_or_construct<>... functions to place the container in the managed memory.

If you do the first two points but you don't use construct<> or find_or_construct<> you are creating a container placed **only** in your process but that allocates memory for contained types from shared memory/memory mapped file.

Let's see an example:

```cpp
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/containers/vector.hpp>
#include <boost/interprocess/containers/string.hpp>
#include <boost/interprocess/allocators/allocator.hpp>

int main ()
{
   using namespace boost::interprocess;
   //Typedefs
   typedef allocator<char, managed_shared_memory::segment_manager>
      CharAllocator;
   typedef basic_string<char, std::char_traits<char>, CharAllocator>
      MyShmString;
   typedef allocator<MyShmString, managed_shared_memory::segment_manager>
      StringAllocator;
   typedef vector<MyShmString, StringAllocator>
      MyShmStringVector;

   //Open shared memory
   //Remove shared memory on construction and destruction
   struct shm_remove
   {
      shm_remove() { shared_memory_object::remove("MySharedMemory"); }
      ~shm_remove(){ shared_memory_object::remove("MySharedMemory"); }
   } remover;

   managed_shared_memory shm(create_only, "MySharedMemory", 10000);

   //Create allocators
   CharAllocator     charallocator  (shm.get_segment_manager());
   StringAllocator   stringallocator(shm.get_segment_manager());

   //This string is in only in this process (the pointer pointing to the
   //buffer that will hold the text is not in shared memory).
   //But the buffer that will hold "this is my text" is allocated from
   //shared memory
   MyShmString mystring(charallocator);
   mystring = "this is my text";

   //This vector is only in this process (the pointer pointing to the
   //buffer that will hold the MyShmString-s is not in shared memory).
   //But the buffer that will hold 10 MyShmString-s is allocated from
   //shared memory using StringAllocator. Since strings use a shared
   //memory allocator (CharAllocator) the 10 buffers that hold
   //"this is my text" text are also in shared memory.
   MyShmStringVector myvector(stringallocator);
   myvector.insert(myvector.begin(), 10, mystring);

   //This vector is fully constructed in shared memory. All pointers
```

```
   //buffers are constructed in the same shared memory segment
   //This vector can be safely accessed from other processes.
   MyShmStringVector *myshmvector =
      shm.construct<MyShmStringVector>("myshmvector")(stringallocator);
   myshmvector->insert(myshmvector->begin(), 10, mystring);

   //Destroy vector. This will free all strings that the vector contains
   shm.destroy_ptr(myshmvector);
   return 0;
}
```

Move semantics in Interprocess containers

Boost.Interprocess containers support move semantics, which means that the contents of a container can be moved from a container two another one, without any copying. The contents of the source container are transferred to the target container and the source container is left in default-constructed state.

When using containers of containers, we can also use move-semantics to insert objects in the container, avoiding unnecessary copies.

To transfer the contents of a container to another one, use `boost::interprocess::move()` function, as shown in the example. For more details about functions supporting move-semantics, see the reference section of Boost.Interprocess containers:

```cpp
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/containers/vector.hpp>
#include <boost/interprocess/containers/string.hpp>
#include <boost/interprocess/allocators/allocator.hpp>
#include <cassert>

int main ()
{
   using namespace boost::interprocess;

   //Typedefs
   typedef managed_shared_memory::segment_manager     SegmentManager;
   typedef allocator<char, SegmentManager>            CharAllocator;
   typedef basic_string<char, std::char_traits<char>
                        ,CharAllocator>                MyShmString;
   typedef allocator<MyShmString, SegmentManager>     StringAllocator;
   typedef vector<MyShmString, StringAllocator>       MyShmStringVector;

   //Remove shared memory on construction and destruction
   struct shm_remove
   {
      shm_remove() { shared_memory_object::remove("MySharedMemory"); }
      ~shm_remove(){ shared_memory_object::remove("MySharedMemory"); }
   } remover;

   managed_shared_memory shm(create_only, "MySharedMemory", 10000);

   //Create allocators
   CharAllocator     charallocator  (shm.get_segment_manager());
   StringAllocator   stringallocator(shm.get_segment_manager());

   //Create a vector of strings in shared memory.
   MyShmStringVector *myshmvector =
      shm.construct<MyShmStringVector>("myshmvector")(stringallocator);

   //Insert 50 strings in shared memory. The strings will be allocated
   //only once and no string copy-constructor will be called when inserting
   //strings, leading to a great performance.
   MyShmString string_to_compare(charallocator);
   string_to_compare = "this is a long, long, long, long, long, long, string...";

   myshmvector->reserve(50);
   for(int i = 0; i < 50; ++i){
      MyShmString move_me(string_to_compare);
      //In the following line, no string copy-constructor will be called.
      //"move_me"'s contents will be transferred to the string created in
      //the vector
      myshmvector->push_back(boost::interprocess::move(move_me));

      //The source string is in default constructed state
      assert(move_me.empty());

      //The newly created string will be equal to the "move_me"'s old contents
      assert(myshmvector->back() == string_to_compare);
   }

   //Now erase a string...
   myshmvector->pop_back();

   //...And insert one in the first position.
   //No string copy-constructor or assignments will be called, but
   //move constructors and move-assignments. No memory allocation
   //function will be called in this operations!!
```

```
    myshmvector->insert(myshmvector->begin(), boost::interprocess::move(string_to_compare));

    //Destroy vector. This will free all strings that the vector contains
    shm.destroy_ptr(myshmvector);
    return 0;
}
```

Containers of containers

When creating containers of containers, each container needs an allocator. To avoid using several allocators with complex type definitions, we can take advantage of the type erasure provided by void allocators and the ability to implicitly convert void allocators in allocators that allocate other types.

Here we have an example that builds a map in shared memory. Key is a string and the mapped type is a class that stores several containers:

```cpp
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/allocators/allocator.hpp>
#include <boost/interprocess/containers/map.hpp>
#include <boost/interprocess/containers/vector.hpp>
#include <boost/interprocess/containers/string.hpp>

using namespace boost::interprocess;

//Typedefs of allocators and containers
typedef managed_shared_memory::segment_manager                segment_manager_t;
typedef allocator<void, segment_manager_t>                    void_allocator;
typedef allocator<int, segment_manager_t>                     int_allocator;
typedef vector<int, int_allocator>                            int_vector;
typedef allocator<int_vector, segment_manager_t>              int_vector_allocator;
typedef vector<int_vector, int_vector_allocator>              int_vector_vector;
typedef allocator<char, segment_manager_t>                    char_allocator;
typedef basic_string<char, std::char_traits<char>, char_allocator>  char_string;

class complex_data
{
   int                 id_;
   char_string         char_string_;
   int_vector_vector   int_vector_vector_;

   public:
   //Since void_allocator is convertible to any other allocator<T>, we can simplify
   //the initialization taking just one allocator for all inner containers.
   complex_data(int id, const char *name, const void_allocator &void_alloc)
      : id_(id), char_string_(name, void_alloc), int_vector_vector_(void_alloc)
   {}
   //Other members...
};

//Definition of the map holding a string as key and complex_data as mapped type
typedef std::pair<const char_string, complex_data>            map_value_type;
typedef std::pair<char_string, complex_data>                  movable_to_map_value_type;
typedef allocator<map_value_type, segment_manager_t>          map_value_type_allocator;
typedef map< char_string, complex_data
           , std::less<char_string>, map_value_type_allocator>  complex_map_type;

int main ()
{
   //Remove shared memory on construction and destruction
   struct shm_remove
   {
```

```
      shm_remove() { shared_memory_object::remove("MySharedMemory"); }
     ~shm_remove(){ shared_memory_object::remove("MySharedMemory"); }
   } remover;

   //Create shared memory
   managed_shared_memory segment(create_only,"MySharedMemory", 65536);

   //An allocator convertible to any allocator<T, segment_manager_t> type
   void_allocator alloc_inst (segment.get_segment_manager());

   //Construct the shared memory map and fill it
   complex_map_type *mymap = segment.construct<complex_map_type>
      //(object name), (first ctor parameter, second ctor parameter)
         ("MyMap")(std::less<char_string>(), alloc_inst);

   for(int i = 0; i < 100; ++i){
      //Both key(string) and value(complex_data) need an allocator in their constructors
      char_string  key_object(alloc_inst);
      complex_data mapped_object(i, "default_name", alloc_inst);
      map_value_type value(key_object, mapped_object);
      //Modify values and insert them in the map
      mymap->insert(value);
   }
   return 0;
}
```

Boost containers compatible with Boost.Interprocess

As mentioned, container developers might need to change their implementation to make them compatible with Boost.Interprocess, because implementation usually ignore allocators with smart pointers. Hopefully several Boost containers are compatible with **Interprocess**.

Boost unordered containers

Boost.Unordered containers are compatible with Interprocess, so programmers can store hash containers in shared memory and memory mapped files. Here is a small example storing `unordered_map` in shared memory:

```cpp
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/allocators/allocator.hpp>

#include <boost/unordered_map.hpp>      //boost::unordered_map
#include <functional>                   //std::equal_to
#include <boost/functional/hash.hpp>    //boost::hash

int main ()
{
   using namespace boost::interprocess;
   //Remove shared memory on construction and destruction
   struct shm_remove
   {
      shm_remove() { shared_memory_object::remove("MySharedMemory"); }
      ~shm_remove(){ shared_memory_object::remove("MySharedMemory"); }
   } remover;

   //Create shared memory
   managed_shared_memory segment(create_only, "MySharedMemory", 65536);

   //Note that unordered_map<Key, MappedType>'s value_type is std::pair<const Key, MappedType>,
   //so the allocator must allocate that pair.
   typedef int    KeyType;
   typedef float  MappedType;
   typedef std::pair<const int, float> ValueType;

   //Typedef the allocator
   typedef allocator<ValueType, managed_shared_memory::segment_manager> ShmemAllocator;

   //Alias an unordered_map of ints that uses the previous STL-like allocator.
   typedef boost::unordered_map
      < KeyType               , MappedType
      , boost::hash<KeyType>  ,std::equal_to<KeyType>
      , ShmemAllocator>
   MyHashMap;

   //Construct a shared memory hash map.
   //Note that the first parameter is the initial bucket count and
   //after that, the hash function, the equality function and the allocator
   MyHashMap *myhashmap = segment.construct<MyHashMap>("MyHashMap")  //object name
      ( 3, boost::hash<int>(), std::equal_to<int>()                  //
      , segment.get_allocator<ValueType>());                         //allocator instance

   //Insert data in the hash map
   for(int i = 0; i < 100; ++i){
      myhashmap->insert(ValueType(i, (float)i));
   }
   return 0;
}
```

Boost.MultiIndex containers

The widely used **Boost.MultiIndex** library is compatible with **Boost.Interprocess** so we can construct pretty good databases in shared memory. Constructing databases in shared memory is a bit tougher than in normal memory, usually because those databases contain strings and those strings need to be placed in shared memory. Shared memory strings require an allocator in their constructors so this usually makes object insertion a bit more complicated.

Here is an example that shows how to put a multi index container in shared memory:

```cpp
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/allocators/allocator.hpp>
#include <boost/interprocess/containers/string.hpp>

#include <boost/multi_index_container.hpp>
#include <boost/multi_index/member.hpp>
#include <boost/multi_index/ordered_index.hpp>

using namespace boost::interprocess;
namespace bmi = boost::multi_index;

typedef managed_shared_memory::allocator<char>::type              char_allocator;
typedef basic_string<char, std::char_traits<char>, char_allocator>shm_string;

//Data to insert in shared memory
struct employee
{
   int         id;
   int         age;
   shm_string  name;
   employee( int id_
           , int age_
           , const char *name_
           , const char_allocator &a)
      : id(id_), age(age_), name(name_, a)
   {}
};

//Tags
struct id{};
struct age{};
struct name{};

// Define a multi_index_container of employees with following indices:
//   - a unique index sorted by employee::int,
//   - a non-unique index sorted by employee::name,
//   - a non-unique index sorted by employee::age.
typedef bmi::multi_index_container<
  employee,
  bmi::indexed_by<
    bmi::ordered_unique
      <bmi::tag<id>,  BOOST_MULTI_INDEX_MEMBER(employee,int,id)>,
    bmi::ordered_non_unique<
      bmi::tag<name>,BOOST_MULTI_INDEX_MEMBER(employee,shm_string,name)>,
    bmi::ordered_non_unique
      <bmi::tag<age>, BOOST_MULTI_INDEX_MEMBER(employee,int,age)> >,
  managed_shared_memory::allocator<employee>::type
> employee_set;

int main ()
{
   //Remove shared memory on construction and destruction
   struct shm_remove
   {
      shm_remove() { shared_memory_object::remove("MySharedMemory"); }
      ~shm_remove(){ shared_memory_object::remove("MySharedMemory"); }
   } remover;

   //Create shared memory
   managed_shared_memory segment(create_only,"MySharedMemory", 65536);

   //Construct the multi_index in shared memory
   employee_set *es = segment.construct<employee_set>
```

```
     ("My MultiIndex Container")              //Container's name in shared memory
     ( employee_set::ctor_args_list()
     , segment.get_allocator<employee>());  //Ctor parameters

   //Now insert elements
   char_allocator ca(segment.get_allocator<char>());
   es->insert(employee(0,31, "Joe", ca));
   es->insert(employee(1,27, "Robert", ca));
   es->insert(employee(2,40, "John", ca));
   return 0;
}
```

Programmers can place **Boost.CircularBuffer** containers in sharecd memory provided they disable debugging facilities with defines BOOST_CB_DISABLE_DEBUG or the more general NDEBUG. The reason is that those debugging facilities are only compatible with raw pointers.

Memory allocation algorithms

simple_seq_fit: A simple shared memory management algorithm

The algorithm is a variation of sequential fit using singly linked list of free memory buffers. The algorithm is based on the article about shared memory titled *"Taming Shared Memory"* . The algorithm is as follows:

The shared memory is divided in blocks of free shared memory, each one with some control data and several bytes of memory ready to be used. The control data contains a pointer (in our case offset_ptr) to the next free block and the size of the block. The allocator consists of a singly linked list of free blocks, ordered by address. The last block, points always to the first block:

```
simple_seq_fit memory layout:

    main        extra  allocated  free_block_1     allocated   free_block_2      allocated   ↵
free_block_3
   header       header  block       ctrl     usr     block      ctrl     usr      block      ctrl↵
   usr
   _____   _____  _____   _____   _____   _____   _____   ↵
_____
   |       ||      ||       ||           ||      |       ||           ||      |       ||      ↵
  |       |
   |free|ctrl||extra||           ||next|size| mem ||           ||next|size| mem ||           ↵
||next|size| mem |
  ↵
   |_____||_____||_____||_____|_____||_____||_____|_____||_____||_____|_____|
      |                             |  |                     |  |                  |  |      ↵

      |_>_>_>_>_>_>_>_>_>_>_>_|  |_>_>_>_>_>_>_>_>_>_>_>_|   |_>_>_>_>_>_>_>_>_>_>_>_|  |      ↵

                               |                                                         |
                               |_<_<_<_<_<_<_<_<_<_<_<_<_<_<_<_<_<_<_<_<_<_<_<_<_<_<_<_<__|
```
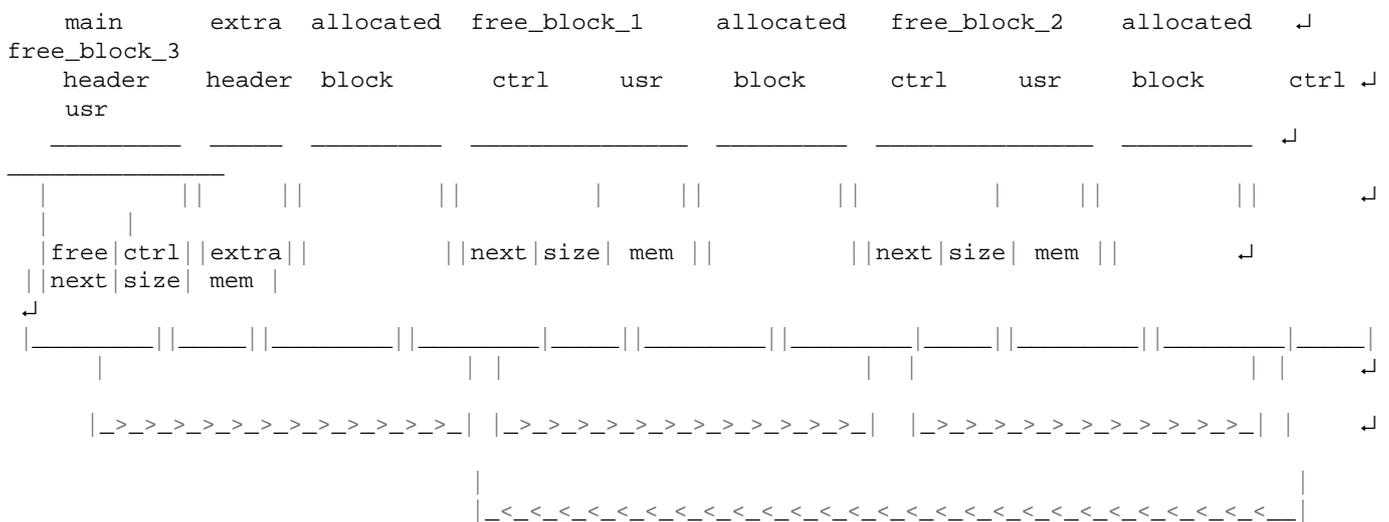
When a user requests N bytes of memory, the allocator traverses the free block list looking for a block large enough. If the "mem" part of the block has the same size as the requested memory, we erase the block from the list and return a pointer to the "mem" part of the block. If the "mem" part size is bigger than needed, we split the block in two blocks, one of the requested size and the other with remaining size. Now, we take the block with the exact size, erase it from list and give it to the user.

When the user deallocates a block, we traverse the list (remember that the list is ordered), and search its place depending on the block address. Once found, we try to merge the block with adjacent blocks if possible.

To ease implementation, the size of the free memory block is measured in multiples of "basic_size" bytes. The basic size will be the size of the control block aligned to machine most restrictive alignment.

This algorithm is a low size overhead algorithm suitable for simple allocation schemes. This algorithm should only be used when size is a major concern, because the performance of this algorithm suffers when the memory is fragmented. This algorithm has linear allocation and deallocation time, so when the number of allocations is high, the user should use a more performance-friendly algorithm.

In most 32 systems, with 8 byte alignment, "basic_size" is 8 bytes. This means that an allocation request of 1 byte leads to the creation of a 16 byte block, where 8 bytes are available to the user. The allocation of 8 bytes leads also to the same 16 byte block.

rbtree_best_fit: Best-fit logarithmic-time complexity allocation

This algorithm is an advanced algorithm using red-black trees to sort the free portions of the memory segment by size. This allows logarithmic complexity allocation. Apart from this, a doubly-linked list of all portions of memory (free and allocated) is maintained to allow constant-time access to previous and next blocks when doing merging operations.

The data used to create the red-black tree of free nodes is overwritten by the user since it's no longer used once the memory is allocated. This maintains the memory size overhead down to the doubly linked list overhead, which is pretty small (two pointers). Basically this is the scheme:

```
rbtree_best_fit memory layout:

  main                allocated block   free block                          allocated block  free block
  header
  _____   _____   _____   _____   ↵
_____
 |               ||            |   ||              |                |   ||            |   ||   ↵
  |                |         |
 |   main header  ||next|prev| mem ||next|prev|left|right|par↵
ent| mem ||next|prev| mem ||next|prev|left|right|parent| mem |
 |_____||_____|_____||_____|_____|____||_____|_____||_____|_____|____| ↵
```
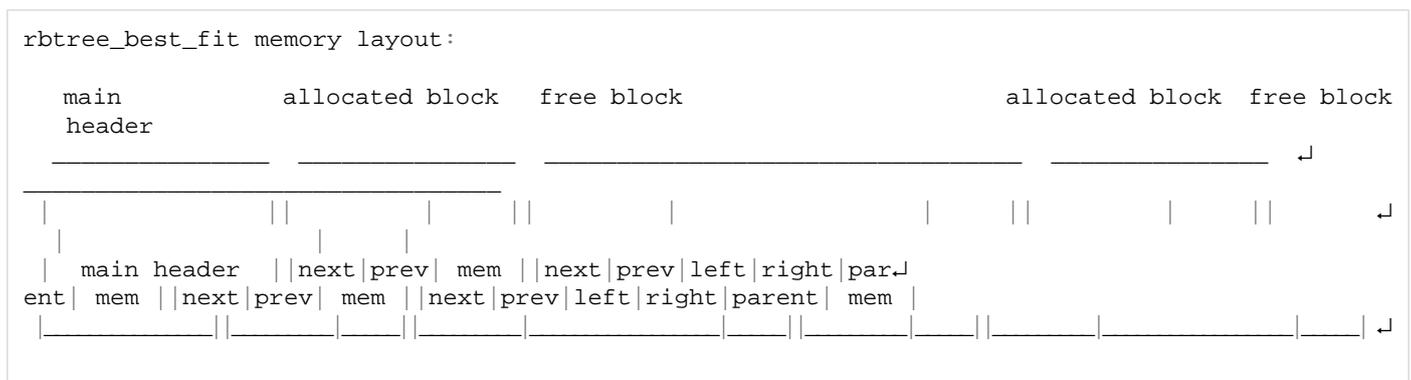
This allocation algorithm is pretty fast and scales well with big shared memory segments and big number of allocations. To form a block a minimum memory size is needed: the sum of the doubly linked list and the red-black tree control data. The size of a block is measured in multiples of the most restrictive alignment value.

In most 32 systems with 8 byte alignment the minimum size of a block is 24 byte. When a block is allocated the control data related to the red black tree is overwritten by the user (because it's only needed for free blocks).

In those systems a 1 byte allocation request means that:

• 24 bytes of memory from the segment are used to form a block.

• 16 bytes of them are usable for the user.

For really small allocations (<= 8 bytes), this algorithm wastes more memory than the simple sequential fit algorithm (8 bytes more). For allocations bigger than 8 bytes the memory overhead is exactly the same. This is the default allocation algorithm in **Boost.Interprocess** managed memory segments.

Direct iostream formatting: vectorstream and bufferstream

Shared memory, memory-mapped files and all **Boost.Interprocess** mechanisms are focused on efficiency. The reason why shared memory is used is that it's the fastest IPC mechanism available. When passing text-oriented messages through shared memory, there is need to format the message. Obviously C++ offers the iostream framework for that work.

Some programmers appreciate the iostream safety and design for memory formatting but feel that the stringstream family is far from efficient not when formatting, but when obtaining formatted data to a string, or when setting the string from which the stream will extract data. An example:

```cpp
//Some formatting elements
std::string my_text = "...";
int number;

//Data reader
std::istringstream input_processor;

//This makes a copy of the string. If not using a
//reference counted string, this is a serious overhead.
input_processor.str(my_text);

//Extract data
while(/*...*/){
    input_processor >> number;
}

//Data writer
std::ostringstream output_processor;

//Write data
while(/*...*/){
    output_processor << number;
}

//This returns a temporary string. Even with return-value
//optimization this is expensive.
my_text = input_processor.str();
```

The problem is even worse if the string is a shared-memory string, because to extract data, we must copy the data first from shared-memory to a `std::string` and then to a `std::stringstream`. To encode data in a shared memory string we should copy data from a `std::stringstream` to a `std::string` and then to the shared-memory string.

Because of this overhead, **Boost.Interprocess** offers a way to format memory-strings (in shared memory, memory mapped files or any other memory segment) that can avoid all unneeded string copy and memory allocation/deallocations, while using all iostream facilities. **Boost.Interprocess vectorstream** and **bufferstream** implement vector-based and fixed-size buffer based storage support for iostreams and all the formatting/locale hard work is done by standard `std::basic_streambuf<>` and `std::basic_iostream<>` classes.

Formatting directly in your character vector: vectorstream

The **vectorstream** class family (**basic_vectorbuf**, **basic_ivectorstream** ,**basic_ovectorstream** and **basic_vectorstream**) is an efficient way to obtain formatted reading/writing directly in a character vector. This way, if a shared-memory vector is used, data is extracted/written from/to the shared-memory vector, without additional copy/allocation. We can see the declaration of basic_vectorstream here:

```cpp
//!A basic_iostream class that holds a character vector specified by CharVector
//!template parameter as its formatting buffer. The vector must have
//!contiguous storage, like std::vector, boost::interprocess::vector or
//!boost::interprocess::basic_string
template <class CharVector, class CharTraits =
         std::char_traits<typename CharVector::value_type> >
class basic_vectorstream
: public std::basic_iostream<typename CharVector::value_type, CharTraits>

{
   public:
   typedef CharVector                                          vector_type;
   typedef typename std::basic_ios
      <typename CharVector::value_type, CharTraits>::char_type    char_type;
   typedef typename std::basic_ios<char_type, CharTraits>::int_type     int_type;
   typedef typename std::basic_ios<char_type, CharTraits>::pos_type     pos_type;
   typedef typename std::basic_ios<char_type, CharTraits>::off_type     off_type;
   typedef typename std::basic_ios<char_type, CharTraits>::traits_type  traits_type;

   //!Constructor. Throws if vector_type default constructor throws.
   basic_vectorstream(std::ios_base::openmode mode
                     = std::ios_base::in | std::ios_base::out);

   //!Constructor. Throws if vector_type(const Parameter &param) throws.
   template<class Parameter>
   basic_vectorstream(const Parameter &param, std::ios_base::openmode mode
                     = std::ios_base::in | std::ios_base::out);

   ~basic_vectorstream(){}

   //!Returns the address of the stored stream buffer.
   basic_vectorbuf<CharVector, CharTraits>* rdbuf() const;

   //!Swaps the underlying vector with the passed vector.
   //!This function resets the position in the stream.
   //!Does not throw.
   void swap_vector(vector_type &vect);

   //!Returns a const reference to the internal vector.
   //!Does not throw.
   const vector_type &vector() const;

   //!Preallocates memory from the internal vector.
   //!Resets the stream to the first position.
   //!Throws if the internals vector's memory allocation throws.
   void reserve(typename vector_type::size_type size);
};
```

The vector type is templatized, so that we can use any type of vector: **std::vector**, `boost::interprocess::vector`... But the storage must be **contiguous**, we can't use a deque. We can even use **boost::interprocess::basic_string**, since it has a vector interface and it has contiguous storage. **We can't use std::string**, because although some std::string implementation are vector-based, others can have optimizations and reference-counted implementations.

The user can obtain a const reference to the internal vector using `vector_type vector() const` function and he also can swap the internal vector with an external one calling `void swap_vector(vector_type &vect)`. The swap function resets the stream position. This functions allow efficient methods to obtain the formatted data avoiding all allocations and data copies.

Let's see an example to see how to use vectorstream:

```cpp
#include <boost/interprocess/containers/vector.hpp>
#include <boost/interprocess/containers/string.hpp>
#include <boost/interprocess/allocators/allocator.hpp>
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/streams/vectorstream.hpp>
#include <iterator>

using namespace boost::interprocess;

typedef allocator<int, managed_shared_memory::segment_manager>
   IntAllocator;
typedef allocator<char, managed_shared_memory::segment_manager>
   CharAllocator;
typedef vector<int, IntAllocator>   MyVector;
typedef basic_string
   <char, std::char_traits<char>, CharAllocator>   MyString;
typedef basic_vectorstream<MyString>                MyVectorStream;

int main ()
{
   //Remove shared memory on construction and destruction
   struct shm_remove
   {
      shm_remove() { shared_memory_object::remove("MySharedMemory"); }
      ~shm_remove(){ shared_memory_object::remove("MySharedMemory"); }
   } remover;

   managed_shared_memory segment(
      create_only,
      "MySharedMemory", //segment name
      65536);           //segment size in bytes

   //Construct shared memory vector
   MyVector *myvector =
      segment.construct<MyVector>("MyVector")
      (IntAllocator(segment.get_segment_manager()));

   //Fill vector
   myvector->reserve(100);
   for(int i = 0; i < 100; ++i){
      myvector->push_back(i);
   }

   //Create the vectorstream. To create the internal shared memory
   //basic_string we need to pass the shared memory allocator as
   //a constructor argument
   MyVectorStream myvectorstream(CharAllocator(segment.get_segment_manager()));

   //Reserve the internal string
   myvectorstream.reserve(100*5);

   //Write all vector elements as text in the internal string
   //Data will be directly written in shared memory, because
   //internal string's allocator is a shared memory allocator
   for(std::size_t i = 0, max = myvector->size(); i < max; ++i){
      myvectorstream << (*myvector)[i] << std::endl;
   }

   //Auxiliary vector to compare original data
   MyVector *myvector2 =
      segment.construct<MyVector>("MyVector2")
      (IntAllocator(segment.get_segment_manager()));
```

```
   //Avoid reallocations
   myvector2->reserve(100);

   //Extract all values from the internal
   //string directly to a shared memory vector.
   std::istream_iterator<int> it(myvectorstream), itend;
   std::copy(it, itend, std::back_inserter(*myvector2));

   //Compare vectors
   assert(std::equal(myvector->begin(), myvector->end(), myvector2->begin()));

   //Create a copy of the internal string
   MyString stringcopy (myvectorstream.vector());

   //Now we create a new empty shared memory string...
   MyString *mystring =
      segment.construct<MyString>("MyString")
      (CharAllocator(segment.get_segment_manager()));

   //...and we swap vectorstream's internal string
   //with the new one: after this statement mystring
   //will be the owner of the formatted data.
   //No reallocations, no data copies
   myvectorstream.swap_vector(*mystring);

   //Let's compare both strings
   assert(stringcopy == *mystring);

   //Done, destroy and delete vectors and string from the segment
   segment.destroy_ptr(myvector2);
   segment.destroy_ptr(myvector);
   segment.destroy_ptr(mystring);
   return 0;
}
```

Formatting directly in your character buffer: bufferstream

As seen, vectorstream offers an easy and secure way for efficient iostream formatting, but many times, we have to read or write formatted data from/to a fixed size character buffer (a static buffer, a c-string, or any other). Because of the overhead of stringstream, many developers (specially in embedded systems) choose sprintf family. The **bufferstream** classes offer iostream interface with direct formatting in a fixed size memory buffer with protection against buffer overflows. This is the interface:

```cpp
//!A basic_iostream class that uses a fixed size character buffer
//!as its formatting buffer.
template <class CharT, class CharTraits = std::char_traits<CharT> >
class basic_bufferstream
   : public std::basic_iostream<CharT, CharTraits>

{
   public:                              // Typedefs
   typedef typename std::basic_ios
      <CharT, CharTraits>::char_type          char_type;
   typedef typename std::basic_ios<char_type, CharTraits>::int_type     int_type;
   typedef typename std::basic_ios<char_type, CharTraits>::pos_type     pos_type;
   typedef typename std::basic_ios<char_type, CharTraits>::off_type     off_type;
   typedef typename std::basic_ios<char_type, CharTraits>::traits_type  traits_type;

   //!Constructor. Does not throw.
   basic_bufferstream(std::ios_base::openmode mode
                   = std::ios_base::in | std::ios_base::out);

   //!Constructor. Assigns formatting buffer. Does not throw.
   basic_bufferstream(CharT *buffer, std::size_t length,
                   std::ios_base::openmode mode
                      = std::ios_base::in | std::ios_base::out);

   //!Returns the address of the stored stream buffer.
   basic_bufferbuf<CharT, CharTraits>* rdbuf() const;

   //!Returns the pointer and size of the internal buffer.
   //!Does not throw.
   std::pair<CharT *, std::size_t> buffer() const;

   //!Sets the underlying buffer to a new value. Resets
   //!stream position. Does not throw.
   void buffer(CharT *buffer, std::size_t length);
};

//Some typedefs to simplify usage
typedef basic_bufferstream<char>     bufferstream;
typedef basic_bufferstream<wchar_t>  wbufferstream;
// ...
```

While reading from a fixed size buffer, **bufferstream** activates endbit flag if we try to read an address beyond the end of the buffer. While writing to a fixed size buffer, **bufferstream** will active the badbit flag if a buffer overflow is going to happen and disallows writing. This way, the fixed size buffer formatting through **bufferstream** is secure and efficient, and offers a good alternative to sprintf/sscanf functions. Let's see an example:

```cpp
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/streams/bufferstream.hpp>
#include <vector>
#include <iterator>
#include <cstddef>

using namespace boost::interprocess;

int main ()
{
   //Remove shared memory on construction and destruction
   struct shm_remove
   {
      shm_remove() { shared_memory_object::remove("MySharedMemory"); }
      ~shm_remove(){ shared_memory_object::remove("MySharedMemory"); }
   } remover;

   //Create shared memory
   managed_shared_memory segment(create_only,
                                 "MySharedMemory",  //segment name
                                 65536);

   //Fill data
   std::vector<int> data;
   data.reserve(100);
   for(int i = 0; i < 100; ++i){
      data.push_back(i);
   }
   const std::size_t BufferSize = 100*5;

   //Allocate a buffer in shared memory to write data
   char *my_cstring =
      segment.construct<char>("MyCString")[BufferSize](0);
   bufferstream mybufstream(my_cstring, BufferSize);

   //Now write data to the buffer
   for(int i = 0; i < 100; ++i){
      mybufstream << data[i] << std::endl;
   }

   //Check there was no overflow attempt
   assert(mybufstream.good());

   //Extract all values from the shared memory string
   //directly to a vector.
   std::vector<int> data2;
   std::istream_iterator<int> it(mybufstream), itend;
   std::copy(it, itend, std::back_inserter(data2));

   //This extraction should have ended will fail error since
   //the numbers formatted in the buffer end before the end
   //of the buffer. (Otherwise it would trigger eofbit)
   assert(mybufstream.fail());

   //Compare data
   assert(std::equal(data.begin(), data.end(), data2.begin()));

   //Clear errors and rewind
   mybufstream.clear();
   mybufstream.seekp(0, std::ios::beg);

   //Now write again the data trying to do a buffer overflow
   for(int i = 0, m = data.size()*5; i < m; ++i){
```

```
            mybufstream << data[i%5] << std::endl;
    }

    //Now make sure badbit is active
    //which means overflow attempt.
    assert(!mybufstream.good());
    assert(mybufstream.bad());
    segment.destroy_ptr(my_cstring);
    return 0;
}
```

As seen, **bufferstream** offers an efficient way to format data without any allocation and extra copies. This is very helpful in embedded systems, or formatting inside time-critical loops, where stringstream extra copies would be too expensive. Unlike sprintf/sscanf, it has protection against buffer overflows. As we know, according to the **Technical Report on C++ Performance**, it's possible to design efficient iostreams for embedded platforms, so this bufferstream class comes handy to format data to stack, static or shared memory buffers.

Ownership smart pointers

C++ users know the importance of ownership smart pointers when dealing with resources. Boost offers a wide range of such type of pointers: `intrusive_ptr<>`, `scoped_ptr<>`, `shared_ptr<>`...

When building complex shared memory/memory mapped files structures, programmers would like to use also the advantages of these smart pointers. The problem is that Boost and C++ TR1 smart pointers are not ready to be used for shared memory. The cause is that those smart pointers contain raw pointers and they use virtual functions, something that is not possible if you want to place your data in shared memory. The virtual function limitation makes even impossible to achieve the same level of functionality of Boost and TR1 with **Boost.Interprocess** smart pointers.

Interprocess ownership smart pointers are mainly "smart pointers containing smart pointers", so we can specify the pointer type they contain.

Intrusive pointer

`boost::interprocess::intrusive_ptr` is the generalization of `boost::intrusive_ptr<>` to allow non-raw pointers as intrusive pointer members. As the well-known `boost::intrusive_ptr` we must specify the pointee type but we also must also specify the pointer type to be stored in the intrusive_ptr:

```
//!The intrusive_ptr class template stores a pointer to an object
//!with an embedded reference count. intrusive_ptr is parameterized on
//!T (the type of the object pointed to) and VoidPointer(a void pointer type
//!that defines the type of pointer that intrusive_ptr will store).
//!intrusive_ptr<T, void *> defines a class with a T* member whereas
//!intrusive_ptr<T, offset_ptr<void> > defines a class with a offset_ptr<T> member.
//!Relies on unqualified calls to:
//!
//!void intrusive_ptr_add_ref(T * p);
//!void intrusive_ptr_release(T * p);
//!
//!with (p != 0)
//!
//!The object is responsible for destroying itself.
template<class T, class VoidPointer>
class intrusive_ptr;
```

So `boost::interprocess::intrusive_ptr<MyClass, void*>` is equivalent to `boost::intrusive_ptr<MyClass>`. But if we want to place the intrusive_ptr in shared memory we must specify a relative pointer type like `boost::interprocess::intrusive_ptr<MyClass, boost::interprocess::offset_ptr<void> >`

```cpp
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/smart_ptr/intrusive_ptr.hpp>

using namespace boost::interprocess;

namespace N {

//A class that has an internal reference count
class reference_counted_class
{
   private:
   //Non-copyable
   reference_counted_class(const reference_counted_class  &);
   //Non-assignable
   reference_counted_class & operator=(const reference_counted_class &);
   //A typedef to save typing
   typedef managed_shared_memory::segment_manager segment_manager;
   //This is the reference count
   unsigned int m_use_count;
   //The segment manager allows deletion from shared memory segment
   offset_ptr<segment_manager> mp_segment_manager;

   public:
   //Constructor
   reference_counted_class(segment_manager *s_mngr)
   : m_use_count(0), mp_segment_manager(s_mngr){}
   //Destructor
   ~reference_counted_class(){}

   public:
   //Returns the reference count
   unsigned int use_count() const
   {   return m_use_count;     }

   //Adds a reference
   inline friend void intrusive_ptr_add_ref(reference_counted_class * p)
   {   ++p->m_use_count; }

   //Releases a reference
   inline friend void intrusive_ptr_release(reference_counted_class * p)
   {   if(--p->m_use_count == 0)  p->mp_segment_manager->destroy_ptr(p); }
};

}  //namespace N {

//A class that has an intrusive pointer to reference_counted_class
class intrusive_ptr_owner
{
   typedef intrusive_ptr<N::reference_counted_class,
                         offset_ptr<void> > intrusive_ptr_t;
   intrusive_ptr_t m_intrusive_ptr;

   public:
   //Takes a pointer to the reference counted class
   intrusive_ptr_owner(N::reference_counted_class *ptr)
      : m_intrusive_ptr(ptr){}
};

int main()
{
   //Remove shared memory on construction and destruction
   struct shm_remove
   {
```

```
      shm_remove() { shared_memory_object::remove("MySharedMemory"); }
      ~shm_remove(){ shared_memory_object::remove("MySharedMemory"); }
   } remover;

   //Create shared memory
   managed_shared_memory shmem(create_only, "MySharedMemory", 10000);

   //Create the unique reference counted object in shared memory
   N::reference_counted_class *ref_counted =
      shmem.construct<N::reference_counted_class>
         ("ref_counted")(shmem.get_segment_manager());

   //Create an array of ten intrusive pointer owners in shared memory
   intrusive_ptr_owner *intrusive_owner_array =
      shmem.construct<intrusive_ptr_owner>
         (anonymous_instance)[10](ref_counted);

   //Now test that reference count is ten
   if(ref_counted->use_count() != 10)
      return 1;

   //Now destroy the array of intrusive pointer owners
   //This should destroy every intrusive_ptr and because of
   //that reference_counted_class will be destroyed
   shmem.destroy_ptr(intrusive_owner_array);

   //Now the reference counted object should have been destroyed
   if(shmem.find<intrusive_ptr_owner>("ref_counted").first)
      return 1;
   //Success!
   return 0;
}
```

Scoped pointer

`boost::interprocess::scoped_ptr<>` is the big brother of `boost::scoped_ptr<>`, which adds a custom deleter to specify how the pointer passed to the scoped_ptr must be destroyed. Also, the `pointer` typedef of the deleter will specify the pointer type stored by scoped_ptr.

```
//!scoped_ptr stores a pointer to a dynamically allocated object.
//!The object pointed to is guaranteed to be deleted, either on destruction
//!of the scoped_ptr, or via an explicit reset. The user can avoid this
//!deletion using release().
//!scoped_ptr is parameterized on T (the type of the object pointed to) and
//!Deleter (the functor to be executed to delete the internal pointer).
//!The internal pointer will be of the same pointer type as typename
//!Deleter::pointer type (that is, if typename Deleter::pointer is
//!offset_ptr<void>, the internal pointer will be offset_ptr<T>).
template<class T, class Deleter>
class scoped_ptr;
```

`scoped_ptr<>` comes handy to implement **rollbacks** with exceptions: if an exception is thrown or we call `return` in the scope of `scoped_ptr<>` the deleter is automatically called so that **the deleter can be considered as a rollback** function. If all goes well, we call `release()` member function to avoid rollback when the `scoped_ptr` goes out of scope.

```cpp
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/smart_ptr/scoped_ptr.hpp>

using namespace boost::interprocess;

class my_class
{};

class my_exception
{};

//A functor that destroys the shared memory object
template<class T>
class my_deleter
{
   private:
   //A typedef to save typing
   typedef managed_shared_memory::segment_manager segment_manager;
   //This my_deleter is created in the stack, not in shared memory,
   //so we can use raw pointers
   segment_manager *mp_segment_manager;

   public:
   //This typedef will specify the pointer type that
   //scoped_ptr will store
   typedef T *pointer;
   //Constructor
   my_deleter(segment_manager *s_mngr)
    : mp_segment_manager(s_mngr){}

   void operator()(pointer object_to_delete)
   {  mp_segment_manager->destroy_ptr(object_to_delete);  }
};

int main ()
{
   //Create shared memory
   //Remove shared memory on construction and destruction
   struct shm_remove
   {
      shm_remove() { shared_memory_object::remove("MySharedMemory"); }
      ~shm_remove(){ shared_memory_object::remove("MySharedMemory"); }
   } remover;

   managed_shared_memory shmem(create_only, "MySharedMemory", 10000);

   //In the first try, there will be no exceptions
   //in the second try we will throw an exception
   for(int i = 0; i < 2; ++i){
      //Create an object in shared memory
      my_class * my_object = shmem.construct<my_class>("my_object")();
      my_class * my_object2 = shmem.construct<my_class>(anonymous_instance)();
      shmem.destroy_ptr(my_object2);

      //Since the next shared memory allocation can throw
      //assign it to a scoped_ptr so that if an exception occurs
      //we destroy the object automatically
      my_deleter<my_class> d(shmem.get_segment_manager());

      try{
         scoped_ptr<my_class, my_deleter<my_class> > s_ptr(my_object, d);
         //Let's emulate a exception capable operation
         //In the second try, throw an exception
```

```
            if(i == 1){
                throw(my_exception());
            }
            //If we have passed the dangerous zone
            //we can release the scoped pointer
            //to avoid destruction
            s_ptr.release();
        }
        catch(const my_exception &){}
        //Here, scoped_ptr is destroyed
        //so it we haven't thrown an exception
        //the object should be there, otherwise, destroyed
        if(i == 0){
            //Make sure the object is alive
            if(!shmem.find<my_class>("my_object").first){
                return 1;
            }
            //Now we can use it and delete it manually
            shmem.destroy<my_class>("my_object");
        }
        else{
            //Make sure the object has been deleted
            if(shmem.find<my_class>("my_object").first){
                return 1;
            }
        }
    }
    return 0;
}
```

Shared pointer and weak pointer

Boost.Interprocess also offers the possibility of creating non-intrusive reference-counted objects in managed shared memory or mapped files.

Unlike boost::shared_ptr, due to limitations of mapped segments `boost::interprocess::shared_ptr` cannot take advantage of virtual functions to maintain the same shared pointer type while providing user-defined allocators and deleters. The allocator and the deleter are template parameters of the shared pointer.

Since the reference count and other auxiliary data needed by `shared_ptr` must be created also in the managed segment, and the deleter has to delete the object from the segment, the user must specify an allocator object and a deleter object when constructing a non-empty instance of `shared_ptr`, just like **Boost.Interprocess** containers need to pass allocators in their constructors.

Here is the declaration of `shared_ptr`:

```
template<class T, class VoidAllocator, class Deleter>
class shared_ptr;
```

- T is the type of the pointed type.

- VoidAllocator is the allocator to be used to allocate auxiliary elements such as the reference count, the deleter... The internal `pointer` typedef of the allocator will determine the type of pointer that shared_ptr will internally use, so allocators defining `pointer` as `offset_ptr<void>` will make all internal pointers used by `shared_ptr` to be also relative pointers. See `boost::interprocess::allocator` for a working allocator.

- Deleter is the function object that will be used to destroy the pointed object when the last reference to the object is destroyed. The deleter functor will take a pointer to T of the same category as the void pointer defined by `VoidAllocator::pointer`. See `boost::interprocess::deleter` for a generic deleter that erases a object from a managed segment.

With correctly specified parameters, **Boost.Interprocess** users can create objects in shared memory that hold shared pointers pointing to other objects also in shared memory, obtaining the benefits of reference counting. Let's see how to create a shared pointer in a managed shared memory:

```cpp
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/smart_ptr/shared_ptr.hpp>
#include <boost/interprocess/allocators/allocator.hpp>
#include <boost/interprocess/smart_ptr/deleter.hpp>
#include <cassert>

using namespace boost::interprocess;

//This is type of the object we want to share
class MyType
{};

typedef managed_shared_memory::segment_manager segment_manager_type;
typedef allocator<void, segment_manager_type>  void_allocator_type;
typedef deleter<MyType, segment_manager_type>  deleter_type;
typedef shared_ptr<MyType, void_allocator_type, deleter_type> my_shared_ptr;

int main ()
{
   //Remove shared memory on construction and destruction
   struct shm_remove
   {
      shm_remove() { shared_memory_object::remove("MySharedMemory"); }
      ~shm_remove(){ shared_memory_object::remove("MySharedMemory"); }
   } remover;

   managed_shared_memory segment(create_only, "MySharedMemory", 4096);

   //Create a shared pointer in shared memory
   //pointing to a newly created object in the segment
   my_shared_ptr &shared_ptr_instance =
      *segment.construct<my_shared_ptr>("shared ptr")
         //Arguments to construct the shared pointer
         ( segment.construct<MyType>("object to share")()     //object to own
         , void_allocator_type(segment.get_segment_manager())  //allocator
         , deleter_type(segment.get_segment_manager())          //deleter
         );
   assert(shared_ptr_instance.use_count() == 1);

   //Destroy "shared ptr". "object to share" will be automatically destroyed
   segment.destroy_ptr(&shared_ptr_instance);

   return 0;
}
```

`boost::interprocess::shared_ptr` is very flexible and configurable (we can specify the allocator and the deleter, for example), but as shown the creation of a shared pointer in managed segments need too much typing.

To simplify this usage, `boost::interprocess::shared_ptr` header offers a shared pointer definition helper class (`managed_shared_ptr`) and a function (`make_managed_shared_ptr`) to easily construct a shared pointer from a type allocated in a managed segment with an allocator that will allocate the reference count also in the managed segment and a deleter that will erase the object from the segment.

These utilities will use a **Boost.Interprocess** allocator (`boost::interprocess::allocator`) and deleter (`boost::interprocess::deleter`) to do their job. The definition of the previous shared pointer could be simplified to the following:

```
typedef managed_shared_ptr<MyType, managed_shared_memory>::type my_shared_ptr;
```

And the creation of a shared pointer can be simplified to this:

```
my_shared_ptr sh_ptr = make_managed_shared_ptr
   (segment.construct<MyType>("object to share")(), segment);
```

Boost.Interprocess also offers a weak pointer named weak_ptr (with its corresponding managed_weak_ptr and make_managed_weak_ptr utilities) to implement non-owning observers of an object owned by shared_ptr.

Now let's see a detailed example of the use of shared_ptr: and weak_ptr

```cpp
#include <boost/interprocess/managed_mapped_file.hpp>
#include <boost/interprocess/smart_ptr/shared_ptr.hpp>
#include <boost/interprocess/smart_ptr/weak_ptr.hpp>
#include <cassert>

using namespace boost::interprocess;

//This is type of the object we want to share
struct type_to_share
{};

//This is the type of a shared pointer to the previous type
//that will be built in the mapped file
typedef managed_shared_ptr<type_to_share, managed_mapped_file>::type shared_ptr_type;
typedef managed_weak_ptr<type_to_share, managed_mapped_file>::type   weak_ptr_type;

//This is a type holding a shared pointer
struct shared_ptr_owner
{
   shared_ptr_owner(const shared_ptr_type &other_shared_ptr)
      : shared_ptr_(other_shared_ptr)
   {}

   shared_ptr_owner(const shared_ptr_owner &other_owner)
      : shared_ptr_(other_owner.shared_ptr_)
   {}

   shared_ptr_type shared_ptr_;
   //...
};

int main ()
{
   //Define file names
   const char *MappedFile  = "MyMappedFile";

   //Destroy any previous file with the name to be used.
   struct file_remove
   {
      file_remove(const char *MappedFile)
         : MappedFile_(MappedFile) { file_mapping::remove(MappedFile_); }
      ~file_remove(){ file_mapping::remove(MappedFile_); }
      const char *MappedFile_;
   } remover(MappedFile);
   {
      managed_mapped_file file(create_only, MappedFile, 65536);

      //Construct the shared type in the file and
      //pass ownership to this local shared pointer
```

```
      shared_ptr_type local_shared_ptr = make_managed_shared_ptr
         (file.construct<type_to_share>("object to share")(), file);
      assert(local_shared_ptr.use_count() == 1);

      //Share ownership of the object between local_shared_ptr and a new "owner1"
      shared_ptr_owner *owner1 =
         file.construct<shared_ptr_owner>("owner1")(local_shared_ptr);
      assert(local_shared_ptr.use_count() == 2);

      //local_shared_ptr releases object ownership
      local_shared_ptr.reset();
      assert(local_shared_ptr.use_count() == 0);
      assert(owner1->shared_ptr_.use_count() == 1);

      //Share ownership of the object between "owner1" and a new "owner2"
      shared_ptr_owner *owner2 =
         file.construct<shared_ptr_owner>("owner2")(*owner1);
      assert(owner1->shared_ptr_.use_count() == 2);
      assert(owner2->shared_ptr_.use_count() == 2);
      assert(owner1->shared_ptr_.get() == owner2->shared_ptr_.get());

      //The mapped file is unmapped here. Objects have been flushed to disk
   }
   {
      //Reopen the mapped file and find again all owners
      managed_mapped_file file(open_only, MappedFile);

      shared_ptr_owner *owner1 = file.find<shared_ptr_owner>("owner1").first;
      shared_ptr_owner *owner2 = file.find<shared_ptr_owner>("owner2").first;
      assert(owner1 && owner2);

      //Check everything is as expected
      assert(file.find<type_to_share>("object to share").first != 0);
      assert(owner1->shared_ptr_.use_count() == 2);
      assert(owner2->shared_ptr_.use_count() == 2);
      assert(owner1->shared_ptr_.get() == owner2->shared_ptr_.get());

      //Now destroy one of the owners, the reference count drops.
      file.destroy_ptr(owner1);
      assert(owner2->shared_ptr_.use_count() == 1);

      //Create a weak pointer
      weak_ptr_type local_observer1(owner2->shared_ptr_);
      assert(local_observer1.use_count() == owner2->shared_ptr_.use_count());

      {  //Create a local shared pointer from the weak pointer
      shared_ptr_type local_shared_ptr = local_observer1.lock();
      assert(local_observer1.use_count() == owner2->shared_ptr_.use_count());
      assert(local_observer1.use_count() == 2);
      }

      //Now destroy the remaining owner. "object to share" will be destroyed
      file.destroy_ptr(owner2);
      assert(file.find<type_to_share>("object to share").first == 0);

      //Test observer
      assert(local_observer1.expired());
      assert(local_observer1.use_count() == 0);
```

```
        //The reference count will be deallocated when all weak pointers
        //disappear. After that, the file is unmapped.
    }
    return 0;
}
```

In general, using **Boost.Interprocess**' `shared_ptr` and `weak_ptr` is very similar to their counterparts boost::shared_ptr and boost::weak_ptr, but they need more template parameters and more run-time parameters in their constructors.

Just like boost::shared_ptr can be stored in a STL container, `shared_ptr` can also be stored in **Boost.Interprocess** containers.

If a programmer just uses `shared_ptr` to be able to insert objects dynamically constructed in the managed segment in a container, but does not need to share the ownership of that object with other objects `unique_ptr` is a much faster and easier to use alternative.

Unique pointer

Unique ownership smart pointers are really useful to free programmers from manual resource liberation of non-shared objects. **Boost.Interprocess**' `unique_ptr` is much like `scoped_ptr` but it's **moveable** and can be easily inserted in **Boost.Interprocess** containers. Here is the declaration of the unique pointer class:

```
template <class T, class D>
class unique_ptr;
```

• T is the type of the object pointed by `unique_ptr`.

• D is the deleter that will erase the object type of the object pointed by `unique_ptr` when the unique pointer is destroyed (and if still owns ownership of the object). If the deleter defines an internal `pointer` typedef, `unique_ptr` will use an internal pointer of the same type. So if `D::pointer` is `offset_ptr<T>` the unique pointer will store a relative pointer instead of a raw one. This allows placing `unique_ptr` in shared memory and memory-mapped files.

`unique_ptr` can release the ownership of the stored pointer so it's useful also to be used as a rollback function. One of the main properties of the class is that **is not copyable, but only moveable**. When a unique pointer is moved to another one, the ownership of the pointer is transferred from the source unique pointer to the target unique pointer. If the target unique pointer owned an object, that object is first deleted before taking ownership of the new object.

`unique_ptr` also offers auxiliary types to easily define and construct unique pointers that can be placed in managed segments and will correctly delete the owned object from the segment: `managed_unique_ptr` and `make_managed_unique_ptr` utilities.

Here we see an example of the use `unique_ptr` including creating containers of such objects:

```cpp
#include <boost/interprocess/managed_mapped_file.hpp>
#include <boost/interprocess/smart_ptr/unique_ptr.hpp>
#include <boost/interprocess/containers/vector.hpp>
#include <boost/interprocess/containers/list.hpp>
#include <boost/interprocess/allocators/allocator.hpp>
#include <cassert>

using namespace boost::interprocess;

//This is type of the object we'll allocate dynamically
struct MyType
{
   MyType(int number = 0)
      :  number_(number)
   {}
   int number_;
};

//This is the type of a unique pointer to the previous type
//that will be built in the mapped file
typedef managed_unique_ptr<MyType, managed_mapped_file>::type unique_ptr_type;

//Define containers of unique pointer. Unique pointer simplifies object management
typedef vector
   < unique_ptr_type
   , allocator<unique_ptr_type, managed_mapped_file::segment_manager>
   > unique_ptr_vector_t;

typedef list
   < unique_ptr_type
   , allocator<unique_ptr_type, managed_mapped_file::segment_manager>
   > unique_ptr_list_t;

int main ()
{
   //Define file names
   const char *MappedFile  = "MyMappedFile";

   //Destroy any previous file with the name to be used.
   struct file_remove
   {
      file_remove(const char *MappedFile)
         : MappedFile_(MappedFile) { file_mapping::remove(MappedFile_); }
      ~file_remove(){ file_mapping::remove(MappedFile_); }
      const char *MappedFile_;
   } remover(MappedFile);
   {
      managed_mapped_file file(create_only, MappedFile, 65536);

      //Construct an object in the file and
      //pass ownership to this local unique pointer
      unique_ptr_type local_unique_ptr (make_managed_unique_ptr
         (file.construct<MyType>("unique object")(), file));
      assert(local_unique_ptr.get() != 0);

      //Reset the unique pointer. The object is automatically destroyed
      local_unique_ptr.reset();
      assert(file.find<MyType>("unique object").first == 0);

      //Now create a vector of unique pointers
      unique_ptr_vector_t *unique_vector =
         file.construct<unique_ptr_vector_t>("unique vector")(file.get_segment_manager());
```

```cpp
      //Speed optimization
      unique_vector->reserve(100);

      //Now insert all values
      for(int i = 0; i < 100; ++i){
         unique_ptr_type p(make_managed_unique_ptr(file.construct<MyType>(anonymous_in↵
stance)(i), file));
         unique_vector->push_back(boost::interprocess::move(p));
         assert(unique_vector->back()->number_ == i);
      }

      //Now create a list of unique pointers
      unique_ptr_list_t *unique_list =
         file.construct<unique_ptr_list_t>("unique list")(file.get_segment_manager());

      //Pass ownership of all values to the list
      for(int i = 99; !unique_vector->empty(); --i){
         unique_list->push_front(boost::interprocess::move(unique_vector->back()));
         //The unique ptr of the vector is now empty...
         assert(unique_vector->back() == 0);
         unique_vector->pop_back();
         //...and the list has taken ownership of the value
         assert(unique_list->front() != 0);
         assert(unique_list->front()->number_ == i);
      }
      assert(unique_list->size() == 100);

      //Now destroy the empty vector.
      file.destroy_ptr(unique_vector);
      //The mapped file is unmapped here. Objects have been flushed to disk
   }
   {
      //Reopen the mapped file and find again the list
      managed_mapped_file file(open_only, MappedFile);

      unique_ptr_list_t    *unique_list =
         file.find<unique_ptr_list_t>("unique list").first;
      assert(unique_list);
      assert(unique_list->size() == 100);

      unique_ptr_list_t::const_iterator list_it = unique_list->begin();
      for(int i = 0; i < 100; ++i, ++list_it){
         assert((*list_it)->number_ == i);
      }

      //Now destroy the list. All elements will be automatically deallocated.
      file.destroy_ptr(unique_list);
   }
   return 0;
}
```

Architecture and internals

Basic guidelines

When building **Boost.Interprocess** architecture, I took some basic guidelines that can be summarized by these points:

- **Boost.Interprocess** should be portable at least in UNIX and Windows systems. That means unifying not only interfaces but also behaviour. This is why **Boost.Interprocess** has chosen kernel or filesystem persistence for shared memory and named synchronization mechanisms. Process persistence for shared memory is also desirable but it's difficult to achieve in UNIX systems.

- **Boost.Interprocess** inter-process synchronization primitives should be equal to thread synchronization primitives. **Boost.Interprocess** aims to have an interface compatible with the C++ standard thread API.

- **Boost.Interprocess** architecture should be modular, customizable but efficient. That's why **Boost.Interprocess** is based on templates and memory algorithms, index types, mutex types and other classes are templatizable.

- **Boost.Interprocess** architecture should allow the same concurrency as thread based programming. Different mutual exclusion levels are defined so that a process can concurrently allocate raw memory when expanding a shared memory vector while another process can be safely searching a named object.

- **Boost.Interprocess** containers know nothing about **Boost.Interprocess**. All specific behaviour is contained in the STL-like allocators. That allows STL vendors to slightly modify (or better said, generalize) their standard container implementations and obtain a fully std::allocator and boost::interprocess::allocator compatible container. This also make **Boost.Interprocess** containers compatible with standard algorithms.

Boost.Interprocess is built above 3 basic classes: a **memory algorithm**, a **segment manager** and a **managed memory segment**:

From the memory algorithm to the managed segment

The memory algorithm

The **memory algorithm** is an object that is placed in the first bytes of a shared memory/memory mapped file segment. The **memory algorithm** can return portions of that segment to users marking them as used and the user can return those portions to the **memory algorithm** so that the **memory algorithm** mark them as free again. There is an exception though: some bytes beyond the end of the memory algorithm object, are reserved and can't be used for this dynamic allocation. This "reserved" zone will be used to place other additional objects in a well-known place.

To sum up, a **memory algorithm** has the same mission as malloc/free of standard C library, but it just can return portions of the segment where it is placed. The layout of a memory segment would be:

```
Layout of the memory segment:

 _____  _____  _____
|             ||          ||                                                  |
|   memory    || reserved ||   The memory algorithm will return portions      |
|  algorithm  ||          ||   of the rest of the segment.                    |
|_____||_____||_____|
```
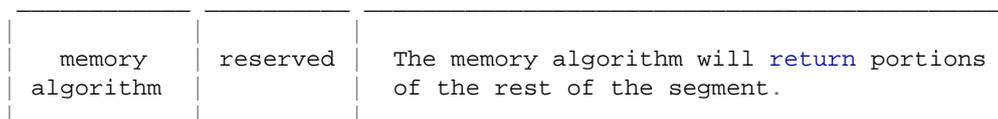
The **memory algorithm** takes care of memory synchronizations, just like malloc/free guarantees that two threads can call malloc/free at the same time. This is usually achieved placing a process-shared mutex as a member of the memory algorithm. Take in care that the memory algorithm knows **nothing** about the segment (if it is shared memory, a shared memory file, etc.). For the memory algorithm the segment is just a fixed size memory buffer.

The **memory algorithm** is also a configuration point for the rest of the **Boost.Interprocess** framework since it defines two basic types as member typedefs:

```
typedef /*implementation dependent*/ void_pointer;
typedef /*implementation dependent*/ mutex_family;
```

The `void_pointer` typedef defines the pointer type that will be used in the **Boost.Interprocess** framework (segment manager, allocators, containers). If the memory algorithm is ready to be placed in a shared memory/mapped file mapped in different base addresses, this pointer type will be defined as `offset_ptr<void>` or a similar relative pointer. If the **memory algorithm** will be used just with fixed address mapping, `void_pointer` can be defined as `void*`.

The rest of the interface of a **Boost.Interprocess memory algorithm** is described in Writing a new shared memory allocation algorithm section. As memory algorithm examples, you can see the implementations `simple_seq_fit` or `rbtree_best_fit` classes.

The segment manager

The **segment manager**, is an object also placed in the first bytes of the managed memory segment (shared memory, memory mapped file), that offers more sophisticated services built above the **memory algorithm**. How can **both** the segment manager and memory algorithm be placed in the beginning of the segment? That's because the segment manager **owns** the memory algorithm: The truth is that the memory algorithm is **embedded** in the segment manager:

```
The layout of managed memory segment:
 _____  _____
|        |        |        |
|  some  | memory | other  |<- The memory algorithm considers
|members |algorithm|members|    "other members" as reserved memory, so
|_____|_____|_____|     it does not use it for dynamic allocation.
|_____|_____
|                          |                                       |
|     segment manager      |   The memory algorithm will return portions |
|                          |   of the rest of the segment.          |
|_____|_____|
```
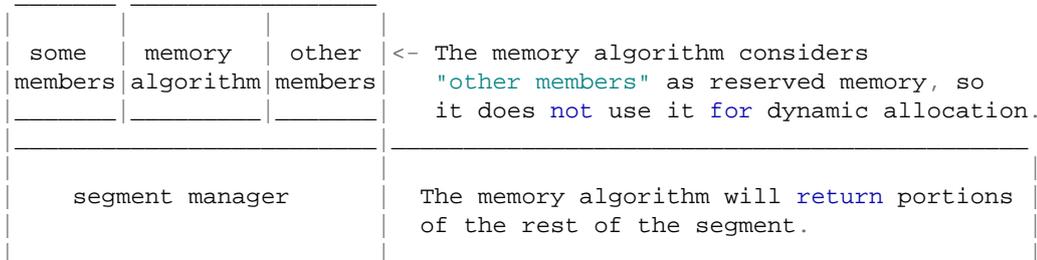
The **segment manager** initializes the memory algorithm and tells the memory manager that it should not use the memory where the rest of the **segment manager**'s member are placed for dynamic allocations. The other members of the **segment manager** are **a recursive mutex** (defined by the memory algorithm's **mutex_family::recursive_mutex** typedef member), and **two indexes (maps)**: one to implement named allocations, and another one to implement "unique instance" allocations.

- The first index is a map with a pointer to a c-string (the name of the named object) as a key and a structure with information of the dynamically allocated object (the most important being the address and the size of the object).

- The second index is used to implement "unique instances" and is basically the same as the first index, but the name of the object comes from a `typeid(T).name()` operation.

The memory needed to store [name pointer, object information] pairs in the index is allocated also via the **memory algorithm**, so we can tell that internal indexes are just like ordinary user objects built in the segment. The rest of the memory to store the name of the object, the object itself, and meta-data for destruction/deallocation is allocated using the **memory algorithm** in a single `allocate()` call.

As seen, the **segment manager** knows **nothing** about shared memory/memory mapped files. The **segment manager** itself does not allocate portions of the segment, it just asks the **memory algorithm** to allocate the needed memory from the rest of the segment. The **segment manager** is a class built above the memory algorithm that offers named object construction, unique instance constructions, and many other services.

The **segment manager** is implemented in **Boost.Interprocess** by the `segment_manager` class.

```
template<class CharType
        ,class MemoryAlgorithm
        ,template<class IndexConfig> class IndexType>
class segment_manager;
```

As seen, the segment manager is quite generic: we can specify the character type to be used to identify named objects, we can specify the memory algorithm that will control dynamically the portions of the memory segment, and we can specify also the index type that will store the [name pointer, object information] mapping. We can construct our own index types as explained in Building custom indexes section.

Boost.Interprocess managed memory segments

The **Boost.Interprocess** managed memory segments that construct the shared memory/memory mapped file, place there the segment manager and forward the user requests to the segment manager. For example, `basic_managed_shared_memory` is a **Boost.Interprocess** managed memory segment that works with shared memory. `basic_managed_mapped_file` works with memory mapped files, etc...

Basically, the interface of a **Boost.Interprocess** managed memory segment is the same as the **segment manager** but it also offers functions to "open", "create", or "open or create" shared memory/memory-mapped files segments and initialize all needed resources. Managed memory segment classes are not built in shared memory or memory mapped files, they are normal C++ classes that store a pointer to the segment manager (which is built in shared memory or memory mapped files).

Apart from this, managed memory segments offer specific functions: `managed_mapped_file` offers functions to flush memory contents to the file, `managed_heap_memory` offers functions to expand the memory, etc...

Most of the functions of **Boost.Interprocess** managed memory segments can be shared between all managed memory segments, since many times they just forward the functions to the segment manager. Because of this, in **Boost.Interprocess** all managed memory segments derive from a common class that implements memory-independent (shared memory, memory mapped files) functions: boost::interprocess::detail::basic_managed_memory_impl

Deriving from this class, **Boost.Interprocess** implements several managed memory classes, for different memory backends:

- `basic_managed_shared_memory` (for shared memory).

- `basic_managed_mapped_file` (for memory mapped files).

- `basic_managed_heap_memory` (for heap allocated memory).

- `basic_managed_external_buffer` (for user provided external buffer).

Allocators and containers

Boost.Interprocess allocators

The **Boost.Interprocess** STL-like allocators are fairly simple and follow the usual C++ allocator approach. Normally, allocators for STL containers are based above new/delete operators and above those, they implement pools, arenas and other allocation tricks.

In **Boost.Interprocess** allocators, the approach is similar, but all allocators are based on the **segment manager**. The segment manager is the only one that provides from simple memory allocation to named object creations. **Boost.Interprocess** allocators always store a pointer to the segment manager, so that they can obtain memory from the segment or share a common pool between allocators.

As you can imagine, the member pointers of the allocator are not a raw pointers, but pointer types defined by the `segment_manager::void_pointer` type. Apart from this, the `pointer` typedef of **Boost.Interprocess** allocators is also of the same type of `segment_manager::void_pointer`.

This means that if our allocation algorithm defines `void_pointer` as `offset_ptr<void>`, `boost::interprocess::allocator<T>` will store an `offset_ptr<segment_manager>` to point to the segment manager and the `boost::interprocess::allocator<T>::pointer` type will be `offset_ptr<T>`. This way, **Boost.Interprocess** allocators can be placed in the memory segment managed by the segment manager, that is, shared memory, memory mapped files, etc...

Implementation of Boost.Interprocess segregated storage pools

Segregated storage pools are simple and follow the classic segregated storage algorithm.

- The pool allocates chunks of memory using the segment manager's raw memory allocation functions.

- The chunk contains a pointer to form a singly linked list of chunks. The pool will contain a pointer to the first chunk.

- The rest of the memory of the chunk is divided in nodes of the requested size and no memory is used as payload for each node. Since the memory of a free node is not used that memory is used to place a pointer to form a singly linked list of free nodes. The pool has a pointer to the first free node.

- Allocating a node is just taking the first free node from the list. If the list is empty, a new chunk is allocated, linked in the list of chunks and the new free nodes are linked in the free node list.

- Deallocation returns the node to the free node list.

- When the pool is destroyed, the list of chunks is traversed and memory is returned to the segment manager.

The pool is implemented by the private_node_pool and shared_node_pool classes.

Implementation of Boost.Interprocess adaptive pools

Adaptive pools are a variation of segregated lists but they have a more complicated approach:

- Instead of using raw allocation, the pool allocates **aligned** chunks of memory using the segment manager. This is an **essential** feature since a node can reach its chunk information applying a simple mask to its address.

- The chunks contains pointers to form a doubly linked list of chunks and an additional pointer to create a singly linked list of free nodes placed on that chunk. So unlike the segregated storage algorithm, the free list of nodes is implemented **per chunk**.

- The pool maintains the chunks in increasing order of free nodes. This improves locality and minimizes the dispersion of node allocations across the chunks facilitating the creation of totally free chunks.

- The pool has a pointer to the chunk with the minimum (but not zero) free nodes. This chunk is called the "active" chunk.

- Allocating a node is just returning the first free node of the "active" chunk. The list of chunks is reordered according to the free nodes count. The pointer to the "active" pool is updated if necessary.

- If the pool runs out of nodes, a new chunk is allocated, and pushed back in the list of chunks. The pointer to the "active" pool is updated if necessary.

- Deallocation returns the node to the free node list of its chunk and updates the "active" pool accordingly.

- If the number of totally free chunks exceeds the limit, chunks are returned to the segment manager.

- When the pool is destroyed, the list of chunks is traversed and memory is returned to the segment manager.

The adaptive pool is implemented by the private_adaptive_node_pool and adaptive_node_pool classes.

Boost.Interprocess containers

Boost.Interprocess containers are standard conforming counterparts of STL containers in `boost::interprocess` namespace, but with these little details:

- **Boost.Interprocess** STL containers don't assume that memory allocated with an allocator can be deallocated with other allocator of the same type. They always compare allocators with `operator==()` to know if this is possible.

- The pointers of the internal structures of the **Boost.Interprocess** containers are of the same type the `pointer` type defined by the allocator of the container. This allows placing containers in managed memory segments mapped in different base addresses.

Performance of Boost.Interprocess

This section tries to explain the performance characteristics of **Boost.Interprocess**, so that you can optimize **Boost.Interprocess** usage if you need more performance.

Performance of raw memory allocations

You can have two types of raw memory allocations with **Boost.Interprocess** classes:

- **Explicit**: The user calls `allocate()` and `deallocate()` functions of managed_shared_memory/managed_mapped_file... managed memory segments. This call is translated to a `MemoryAlgorithm::allocate()` function, which means that you will need just the time that the memory algorithm associated with the managed memory segment needs to allocate data.

- **Implicit**: For example, you are using `boost::interprocess::allocator<...>` with **Boost.Interprocess** containers. This allocator calls the same `MemoryAlgorithm::allocate()` function than the explicit method, **every** time a vector/string has to reallocate its buffer or **every** time you insert an object in a node container.

If you see that memory allocation is a bottleneck in your application, you have these alternatives:

- If you use map/set associative containers, try using `flat_map` family instead of the map family if you mainly do searches and the insertion/removal is mainly done in an initialization phase. The overhead is now when the ordered vector has to reallocate its storage and move data. You can also call the `reserve()` method of these containers when you know beforehand how much data you will insert. However in these containers iterators are invalidated in insertions so this substitution is only effective in some applications.

- Use a **Boost.Interprocess** pooled allocator for node containers, because pooled allocators call `allocate()` only when the pool runs out of nodes. This is pretty efficient (much more than the current default general-purpose algorithm) and this can save a lot of memory. See Segregated storage node allocators and Adaptive node allocators for more information.

- Write your own memory algorithm. If you have experience with memory allocation algorithms and you think another algorithm is better suited than the default one for your application, you can specify it in all **Boost.Interprocess** managed memory segments. See the section Writing a new shared memory allocation algorithm to know how to do this. If you think its better than the default one for general-purpose applications, be polite and donate it to **Boost.Interprocess** to make it default!

Performance of named allocations

Boost.Interprocess allows the same parallelism as two threads writing to a common structure, except when the user creates/searches named/unique objects. The steps when creating a named object are these:

- Lock a recursive mutex (so that you can make named allocations inside the constructor of the object to be created).

- Try to insert the [name pointer, object information] in the name/object index. This lookup has to assure that the name has not been used before. This is achieved calling `insert()` function in the index. So the time this requires is dependent on the index type (ordered vector, tree, hash...). This can require a call to the memory algorithm allocation function if the index has to be reallocated, it's a node allocator, uses pooled allocations...

- Allocate a single buffer to hold the name of the object, the object itself, and meta-data for destruction (number of objects, etc...).

- Call the constructors of the object being created. If it's an array, one construtor per array element.

- Unlock the recursive mutex.

The steps when destroying a named object using the name of the object (`destroy<T>(name)`) are these:

- Lock a recursive mutex .

- Search in the index the entry associated to that name. Copy that information and erase the index entry. This is done using `find(const key_type &)` and `erase(iterator)` members of the index. This can require element reordering if the index is a balanced tree, an ordered vector...

- Call the destructor of the object (many if it's an array).

- Deallocate the memory buffer containing the name, metadata and the object itself using the allocation algorithm.

- Unlock the recursive mutex.

The steps when destroying a named object using the pointer of the object (`destroy_ptr(T *ptr)`) are these:

- Lock a recursive mutex .

- Depending on the index type, this can be different:

- If the index is a node index, (marked with `boost::interprocess::is_node_index` specialization): Take the iterator stored near the object and call `erase(iterator)`. This can require element reordering if the index is a balanced tree, an ordered vector...

- If it's not an node index: Take the name stored near the object and erase the index entry calling `erase(const key &)`. This can require element reordering if the index is a balanced tree, an ordered vector...

- Call the destructor of the object (many if it's an array).

- Deallocate the memory buffer containing the name, metadata and the object itself using the allocation algorithm.

- Unlock the recursive mutex.

If you see that the performance is not good enough you have these alternatives:

- Maybe the problem is that the lock time is too big and it hurts parallelism. Try to reduce the number of named objects in the global index and if your application serves several clients try to build a new managed memory segment for each one instead of using a common one.

- Use another **Boost.Interprocess** index type if you feel the default one is not fast enough. If you are not still satisfied, write your own index type. See Building custom indexes for this.

- Destruction via pointer is at least as fast as using the name of the object and can be faster (in node containers, for example). So if your problem is that you make at lot of named destructions, try to use the pointer. If the index is a node index you can save some time.

Customizing Boost.Interprocess

Writing a new shared memory allocation algorithm

If the default algorithm does not satisfy user requirements, it's easy to provide different algorithms like bitmapping or more advanced segregated lists to meet requirements. The class implementing the algorithm must be compatible with shared memory, so it shouldn't have any virtual function or virtual inheritance or any indirect base class with virtual function or inheritance.

This is the interface to be implemented:

```
class my_algorithm
{
   public:

   //!The mutex type to be used by the rest of Interprocess framework
   typedef implementation_defined   mutex_family;

   //!The pointer type to be used by the rest of Interprocess framework
   typedef implementation_defined   void_pointer;

   //!Constructor. "size" is the total size of the managed memory segment,
   //!"extra_hdr_bytes" indicates the extra bytes after the sizeof(my_algorithm)
   //!that the allocator should not use at all.
   my_algorithm (std::size_t size, std::size_t extra_hdr_bytes);

   //!Obtains the minimum size needed by the algorithm
   static std::size_t get_min_size (std::size_t extra_hdr_bytes);

   //!Allocates bytes, returns 0 if there is not more memory
   void* allocate (std::size_t nbytes);

   //!Deallocates previously allocated bytes
   void  deallocate (void *adr);

   //!Returns the size of the memory segment
   std::size_t get_size()  const;

   //!Increases managed memory in extra_size bytes more
   void grow(std::size_t extra_size);
   /*...*/
};
```

Let's see the public typedefs to define:

```
typedef /* . . . */ void_pointer;
typedef /* . . . */ mutex_family;
```

The `void_pointer` typedef specifies the pointer type to be used in the **Boost.Interprocess** framework that uses the algorithm. For example, if we define

```
typedef void * void_pointer;
```

all **Boost.Interprocess** framework using this algorithm will use raw pointers as members. But if we define:

```
typedef offset_ptr<void> void_pointer;
```

then all **Boost.Interprocess** framework will use relative pointers.

The `mutex_family` is a structure containing typedefs for different interprocess_mutex types to be used in the **Boost.Interprocess** framework. For example the defined

```
struct mutex_family
{
   typedef boost::interprocess::interprocess_mutex             mutex_type;
   typedef boost::interprocess::interprocess_recursive_mutex   recursive_mutex_type;
};
```

defines all interprocess_mutex types using boost::interprocess interprocess_mutex types. The user can specify the desired mutex family.

```
typedef mutex_family mutex_family;
```

The new algorithm (let's call it **my_algorithm**) must implement all the functions that boost::interprocess::rbtree_best_fit class offers:

- **my_algorithm**'s constructor must take 2 arguments:

  - **size** indicates the total size of the managed memory segment, and **my_algorithm** object will be always constructed a at offset 0 of the memory segment.

  - The **extra_hdr_bytes** parameter indicates the number of bytes after the offset sizeof(my_algorithm) that **my_algorithm** can't use at all. This extra bytes will be used to store additional data that should not be overwritten. So, **my_algorithm** will be placed at address XXX of the memory segment, and will manage the **[XXX + sizeof(my_algorithm) + extra_hdr_bytes, XXX + size)** range of the segment.

- The **get_min_size()** function should return the minimum space the algorithm needs to be valid with the passed **extra_hdr_bytes** parameter. This function will be used to check if the memory segment is big enough to place the algorithm there.

- The **allocate()** function must return 0 if there is no more available memory. The memory returned by **my_algorithm** must be aligned to the most restrictive memory alignment of the system, for example, to the value returned by **detail::alignment_of<boost::detail::max_align>::value**. This function should be executed with the synchronization capabilities offered by typename mutex_family::mutex_type interprocess_mutex. That means, that if we define typedef mutex_family mutex_family; then this function should offer the same synchronization as if it was surrounded by an interprocess_mutex lock/unlock. Normally, this is implemented using a member of type mutex_family::mutex_type, but it could be done using atomic instructions or lock free algorithms.

- The **deallocate()** function must make the returned buffer available for new allocations. This function should offer the same synchronization as allocate().

- The **size()** function will return the passed **size** parameter in the constructor. So, **my_algorithm** should store the size internally.

- The **grow()** function will expand the managed memory by **my_algorithm** in **extra_size** bytes. So **size()** function should return the updated size, and the new managed memory range will be (if the address where the algorithm is constructed is XXX): **[XXX + sizeof(my_algorithm) + extra_hdr_bytes, XXX + old_size + extra_size)**. This function should offer the same synchronization as allocate().

That's it. Now we can create new managed shared memory that uses our new algorithm:

```
//Managed memory segment to allocate named (c-string) objects
//using a user-defined memory allocation algorithm
basic_managed_shared_memory<char,
                          ,my_algorithm
                          ,flat_map_index>
   my_managed_shared_memory;
```

Building custom STL compatible allocators for Boost.Interprocess

If provided STL-like allocators don't satisfy user needs, the user can implement another STL compatible allocator using raw memory allocation and named object construction functions. The user can this way implement more suitable allocation schemes on top of basic shared memory allocation schemes, just like more complex allocators are built on top of new/delete functions.

When using a managed memory segment, **get_segment_manager()** function returns a pointer to the segment manager. With this pointer, the raw memory allocation and named object construction functions can be called directly:

```
//Create the managed shared memory and initialize resources
managed_shared_memory segment
   (create_only
   ,"/MySharedMemory"   //segment name
   ,65536);             //segment size in bytes

//Obtain the segment manager
managed_shared_memory::segment_manager *segment_mngr
   = segment.get_segment_manager();

//With the segment manager, now we have access to all allocation functions
segment_mngr->deallocate(segment_mngr->allocate(32));
segment_mngr->construct<int>("My_Int")[32](0);
segment_mngr->destroy<int>("My_Int");

//Initialize the custom, managed memory segment compatible
//allocator with the segment manager.
//
//MySTLAllocator uses segment_mngr->xxx functions to
//implement its allocation scheme
MySTLAllocator<int> stl_alloc(segment_mngr);

//Alias a new vector type that uses the custom STL compatible allocator
typedef std::vector<int, MySTLAllocator<int> > MyVect;

//Construct the vector in shared memory with the allocator as constructor parameter
segment.construct<MyVect>("MyVect_instance")(stl_alloc);
```

The user can create new STL compatible allocators that use the segment manager to access to all memory management/object construction functions. All **Boost.Interprocess**' STL compatible allocators are based on this approach. **Remember** that to be compatible with managed memory segments, allocators should define their **pointer** typedef as the same pointer family as `segment_manager::void_pointer` typedef. This means that if `segment_manager::void_pointer` is `offset_ptr<void>`, `MySTLAllocator<int>` should define `pointer` as `offset_ptr<int>`. The reason for this is that allocators are members of containers, and if we want to put the container in a managed memory segment, the allocator should be ready for that.

Building custom indexes

The managed memory segment uses a name/object index to speed up object searching and creation. Default specializations of managed memory segments (`managed_shared_memory` for example), use `boost::interprocess::flat_map` as index.

However, the index type can be chosen via template parameter, so that the user can define its own index type if he needs that. To construct a new index type, the user must create a class with the following guidelines:

• The interface of the index must follow the common public interface of std::map and std::tr1::unordered_map including public typedefs. The `value_type` typedef can be of type:

```
std::pair<key_type, mapped_type>
```

or

```
std::pair<const key_type, mapped_type>
```

so that ordered arrays or deques can be used as index types. Some known classes following this basic interface are `boost::unordered_map`, `boost::interprocess::flat_map` and `boost::interprocess::map`.

• The class must be a class template taking only a traits struct of this type:

```
struct index_traits
{
   typedef /*...*/   key_type;
   typedef /*...*/   mapped_type;
   typedef /*...*/   segment_manager;
};
```

```
template <class IndexTraits>
class my_index_type;
```

The `key_type` typedef of the passed `index_traits` will be a specialization of the following class:

```
//!The key of the named allocation information index. Stores a to
//!a null string and the length of the string to speed up sorting
template<...>
struct index_key
{
   typedef /*...*/                          char_type;
   typedef /*...*/                          const_char_ptr_t;

   //Pointer to the object's name (null terminated)
   const_char_ptr_t                         mp_str;

   //Length of the name buffer (null NOT included)
   std::size_t                              m_len;

   //!Constructor of the key
   index_key (const CharT *name, std::size_t length);

   //!Less than function for index ordering
   bool operator < (const index_key & right) const;

   //!Equal to function for index ordering
   bool operator == (const index_key & right) const;
};
```

The `mapped_type` is not directly modified by the customized index but it is needed to define the index type. The **segment_manager** will be the type of the segment manager that will manage the index. `segment_manager` will define interesting internal types like `void_pointer` or `mutex_family`.

• The constructor of the customized index type must take a pointer to segment_manager as constructor argument:

```
constructor(segment_manager *segment_mngr);
```

• The index must provide a memory reservation function, that optimizes the index if the user knows the number of elements to be inserted in the index:

```
void reserve(std::size_t n);
```

For example, the index type `flat_map_index` based in `boost::interprocess::flat_map` is just defined as:

```cpp
namespace boost { namespace interprocess {

//!Helper class to define typedefs from IndexTraits
template <class MapConfig>
struct flat_map_index_aux
{
   typedef typename MapConfig::key_type              key_type;
   typedef typename MapConfig::mapped_type           mapped_type;
   typedef typename MapConfig::
      segment_manager_base                           segment_manager_base;
   typedef std::less<key_type>                       key_less;
   typedef std::pair<key_type, mapped_type>          value_type;
   typedef allocator<value_type
                     ,segment_manager_base>   allocator_type;
   typedef flat_map<key_type,   mapped_type,
                     key_less, allocator_type>        index_t;
};

//!Index type based in flat_map. Just derives from flat_map and
//!defines the interface needed by managed memory segments.
template <class MapConfig>
class flat_map_index
   //Derive class from flat_map specialization
   : public flat_map_index_aux<MapConfig>::index_t
{
   /// @cond
   typedef flat_map_index_aux<MapConfig>  index_aux;
   typedef typename index_aux::index_t    base_type;
   typedef typename index_aux::
      segment_manager_base            segment_manager_base;
   /// @endcond

   public:
   //!Constructor. Takes a pointer to the segment manager. Can throw
   flat_map_index(segment_manager_base *segment_mngr)
      : base_type(typename index_aux::key_less(),
                  typename index_aux::allocator_type(segment_mngr))
   {}

   //!This reserves memory to optimize the insertion of n elements in the index
   void reserve(std::size_t n)
   { base_type::reserve(n);   }

   //!This frees all unnecessary memory
   void shrink_to_fit()
   { base_type::shrink_to_fit();   }
};

}}   //namespace boost { namespace interprocess
```

If the user is defining a node container based index (a container whose iterators are not invalidated when inserting or erasing other elements), **Boost.Interprocess** can optimize named object destruction when destructing via pointer. **Boost.Interprocess** can store an iterator next to the object and instead of using the name of the object to erase the index entry, it uses the iterator, which is a faster operation. So if you are creating a new node container based index (for example, a tree), you should define an specialization of `boost::interprocess::is_node_index<...>` defined in `<boost/interprocess/detail/utilities.hpp>`:

```
//!Trait classes to detect if an index is a node
//!index. This allows more efficient operations
//!when deallocating named objects.
template<class MapConfig>
struct is_node_index
    <my_index<MapConfig> >
{
    enum {   value = true };
};
```

Interprocess also defines other index types:

- **boost::map_index** uses **boost::interprocess::map** as index type.

- **boost::null_index** that uses an dummy index type if the user just needs anonymous allocations and wants to save some space and class instantations.

Defining a new managed memory segment that uses the new index is easy. For example, a new managed shared memory that uses the new index:

```
//!Defines a managed shared memory with a c-strings as
//!a keys, the red-black tree best fit algorithm (with process-shared mutexes
//!and offset_ptr pointers) as raw shared memory management algorithm
//!and a custom index
typedef
    basic_managed_shared_memory <
                                char,
                                rbtree_best_fit<mutex_family>,
                                my_index_type
                         >
    my_managed_shared_memory;
```

Acknowledgements, notes and links

Thanks to...

People

Many people have contributed with ideas and revisions, so this is the place to thank them:

- Thanks to all people who have shown interest in the library and have downloaded and tested the snapshots.

- Thanks to **Francis Andre** and **Anders Hybertz** for their ideas and suggestions. Many of them are not implemented yet but I hope to include them when library gets some stability.

- Thanks to **Matt Doyle**, **Steve LoBasso**, **Glenn Schrader**, **Hiang Swee Chiang**, **Phil Endecott**, **Rene Rivera**, **Harold Pirtle**, **Paul Ryan**, **Shumin Wu**, **Michal Wozniak**, **Peter Johnson**, **Alex Ott**, **Shane Guillory**, **Steven Wooding** and **Kim Barrett** for their bug fixes and library testing.

- Thanks to **Martin Adrian** who suggested the use of Interprocess framework for user defined buffers.

- Thanks to **Synge Todo** for his boostbook-doxygen patch to improve Interprocess documentation.

- Thanks to **Olaf Krzikalla** for his Intrusive library. I have taken some ideas to improve red black tree implementation from his library.

- Thanks to **Daniel James** for his unordered_map/set family and his help with allocators. His great unordered implementation has been a reference to design exception safe containers.

- Thanks to **Howard Hinnant** for his amazing help, specially explaining allocator swapping, move semantics and for developing upgradable mutex and lock transfer features.

- Thanks to **Pavel Vozenilek** for his continuous review process, suggestions, code and help. He is the major supporter of Interprocess library. The library has grown with his many and great advices.

- And finally, thank you to all Boosters. **Long live to C++!**

Release Notes

Boost 1.45 Release

- Fixed bugs #1080, #3284, #3439, #3448, #3582, #3682, #3829, #3846, #3914, #3947, #3950, #3951, #3985, #4010, #4417, #4019, #4039, #4218, #4230, #4250, #4297, #4350, #4352, #4426, #4516, #4524, #4557, #4606, #4685, #4694.

- Added support for standard rvalue reference move semantics (tested on GCC 4.5 and VC10).

- Permissions can be detailed for interprocess named resources.

- `mapped_region::flush` initiates disk flushing but does not guarantee it's completed when returns, since it is not portable.

- FreeBSD and MacOS now use posix semaphores to implement named semaphores and mutex.

Boost 1.41 Release

- Support for POSIX shared memory in Mac OS.

- **ABI breaking**: Generic `semaphore` and `named_semaphore` now implemented more efficiently with atomic operations.

- More robust file opening in Windows platforms with active Anti-virus software.

Boost 1.40 Release

- Windows shared memory is created in Shared Documents folder so that it can be shared between services and processes

- Fixed bugs #2967, #2973, #2992, #3138, #3166, #3205.

Boost 1.39 Release

- Added experimental `stable_vector` container.

- `shared_memory_object::remove` has now POSIX `unlink` semantics and `file_mapping::remove` was added to obtain POSIX `unlink` semantics with mapped files.

- Shared memory in windows has now kernel lifetime instead of filesystem lifetime: shared memory will disappear when the system reboots.

- Updated move semantics.

- Fixed bugs #2722, #2729, #2766, #1390, #2589,

Boost 1.38 Release

- Updated documentation to show rvalue-references funcions instead of emulation functions.

- More non-copyable classes are now movable.

- Move-constructor and assignments now leave moved object in default-constructed state instead of just swapping contents.

- Several bugfixes (#2391, #2431, #1390, #2570, #2528.

Boost 1.37 Release

- Containers can be used now in recursive types.

- Added `BOOST_INTERPROCESS_FORCE_GENERIC_EMULATION` macro option to force the use of generic emulation code for process-shared synchronization primitives instead of native POSIX functions.

- Added placement insertion members to containers

- `boost::posix_time::pos_inf` value is now handled portably for timed functions.

- Update some function parameters from `iterator` to `const_iterator` in containers to keep up with the draft of the next standard.

- Documentation fixes.

Boost 1.36 Release

- Added anonymous shared memory for UNIX systems.

- Fixed erroneous `void` return types from `flat_map::erase()` functions.

- Fixed missing move semantics on managed memory classes.

- Added copy_on_write and open_read_only options for shared memory and mapped file managed classes.

- **ABI breaking**: Added to `mapped_region` the mode used to create it.

- Corrected instantiation errors in void allocators.

- `shared_ptr` is movable and supports aliasing.

Boost 1.35 Release

- Added auxiliary utilities to ease the definition and construction of `shared_ptr`, `weak_ptr` and `unique_ptr`. Added explanations and examples of these smart pointers in the documentation.

- Optimized vector:

  - 1) Now works with raw pointers as much as possible when using allocators defining `pointer` as an smart pointer. This increases performance and improves compilation times.

  - 2) A bit of metaprogramming to avoid using move_iterator when the type has trivial copy constructor or assignment and improve performance.

  - 3) Changed custom algorithms with standard ones to take advantage of optimized standard algorithms.

  - 4) Removed unused code.

- **ABI breaking**: Containers don't derive from allocators, to avoid problems with allocators that might define virtual functions with the same names as container member functions. That would convert container functions in virtual functions and might disallow some of them if the returned type does not lead to a covariant return. Allocators are now stored as base classes of internal structs.

- Implemented `named_mutex` and `named_semaphore` with POSIX named semaphores in systems supporting that option. `named_condition` has been accordingly changed to support interoperability with `named_mutex`.

- Reduced template bloat for node and adaptive allocators extracting node implementation to a class that only depends on the memory algorithm, instead of the segment manager + node size + node number...

- Fixed bug in `mapped_region` in UNIX when mapping address was provided but the region was mapped in another address.

- Added `aligned_allocate` and `allocate_many` functions to managed memory segments.

- Improved documentation about managed memory segments.

- **Boost.Interprocess** containers are now documented in the Reference section.

- Correction of typos and documentation errors.

- Added `get_instance_name`, `get_instance_length` and `get_instance_type` functions to managed memory segments.

- Corrected suboptimal buffer expansion bug in `rbtree_best_fit`.

- Added iteration of named and unique objects in a segment manager.

- Fixed leak in `vector`.

- Added support for Solaris.

- Optimized `segment_manager` to avoid code bloat associated with templated instantiations.

- Fixed bug for UNIX: No slash ('/') was being added as the first character for shared memory names, leading to errors in some UNIX systems.

- Fixed bug in VC-8.0: Broken function inlining in core offset_ptr functions.

- Code examples changed to use new BoostBook code import features.

- Added aligned memory allocation function to memory algorithms.

- Fixed bug in `deque::clear()` and `deque::erase()`, they were declared private.

- Fixed bug in `deque::erase()`. Thanks to Steve LoBasso.

- Fixed bug in `atomic_dec32()`. Thanks to Glenn Schrader.

- Improved (multi)map/(multi)set constructors taking iterators. Now those have linear time if the iterator range is already sorted.

- **ABI breaking**: (multi)map/(multi)set now reduce their node size. The color bit is embedded in the parent pointer. Now, the size of a node is the size of 3 pointers in most systems. This optimization is activated for raw and `offset_ptr` pointers.

- (multi)map/(multi)set now reuse memory from old nodes in the assignment operator.

- **ABI breaking**: Implemented node-containers based on intrusive containers. This saves code size, since many instantiations share the same algorithms.

- Corrected code to be compilable with Visual C++ 8.0.

- Added function to zero free memory in memory algorithms and the segment manager. This function is useful for security reasons and to improve compression ratios for files created with `managed_mapped_file`.

- Added support for intrusive index types in managed memory segments. Intrusive indexes save extra memory allocations to allocate the index since with just one allocation, we allocate room for the value, the name and the hook to insert the object in the index.

- Created new index type: **iset_index**. It's an index based on an intrusive set (rb-tree).

- Created new index type: **iunordered_set_index**. It's an index based on a pseudo-intrusive unordered set (hash table).

- **ABI breaking**: The intrusive index **iset_index** is now the default index type.

- Optimized vector to take advantage of `boost::has_trivial_destructor`. This optimization avoids calling destructors of elements that have a trivial destructor.

- Optimized vector to take advantage of `has_trivial_destructor_after_move` trait. This optimization avoids calling destructors of elements that have a trivial destructor if the element has been moved (which is the case of many movable types). This trick was provided by Howard Hinnant.

- Added security check to avoid integer overflow bug in allocators and named construction functions.

- Added alignment checks to forward and backwards expansion functions.

- Fixed bug in atomic functions for PPC.

- Fixed race-condition error when creating and opening a managed segment.

- Added adaptive pools.

- **Source breaking**: Changed node allocators' template parameter order to make them easier to use.

- Added support for native windows shared memory.

- Added more tests.

- Corrected the presence of private functions in the reference section.

- Added function (`deallocate_free_chunks()`) to manually deallocate completely free chunks from node allocators.

- Implemented N1780 proposal to LWG issue 233: *Insertion hints in associative containers* in interprocess `multiset` and `multimap` classes.

- **Source breaking**: A shared memory object is now used including `shared_memory_object.hpp` header instead of `shared memory.hpp`.

- **ABI breaking**: Changed global mutex when initializing managed shared memory and memory mapped files. This change tries to minimize deadlocks.

- **Source breaking**: Changed shared memory, memory mapped files and mapped region's open mode to a single `mode_t` type.

- Added extra WIN32_LEAN_AND_MEAN before including DateTime headers to avoid socket redefinition errors when using Interprocess and Asio in windows.

- **ABI breaking**: `mapped_region` constructor no longer requires classes derived from memory_mappable, but classes must fulfill the MemoryMappable concept.

- Added in-place reallocation capabilities to basic_string.

- **ABI breaking**: Reimplemented and optimized small string optimization. The narrow string class has zero byte overhead with an internal 11 byte buffer in 32 systems!

- Added move semantics to containers. Improves performance when using containers of containers.

- **ABI breaking**: End nodes of node containers (list, slist, map/set) are now embedded in the containers instead of allocated using the allocator. This allows no-throw move-constructors and improves performance.

- **ABI breaking**: **slist** and **list** containers now have constant-time *size()* function. The size of the container is added as a member.

Books and interesting links

Some useful references about the C++ programming language, C++ internals, shared memory, allocators and containers used to design **Boost.Interprocess**.

Books

- Great book about multithreading, and POSIX: **"Programming with Posix Threads"**, **David R. Butenhof**

- The UNIX inter-process bible: **"UNIX Network Programming, Volume 2: Interprocess Communications"**, **W. Richard Stevens**

- Current STL allocator issues: **"Effective STL"**, **Scott Meyers**

- My C++ bible: **"Thinking in C++, Volume 1 & 2"**, **Bruce Eckel and Chuck Allison**

- The book every C++ programmer should read: **"Inside the C++ Object Model"**, **Stanley B. Lippman**

- A must-read: **"ISO/IEC TR 18015: Technical Report on C++ Performance"**, **ISO WG21-SC22 members.**

Links

- A framework to put the STL in shared memory: *"A C++ Standard Allocator for the Standard Template Library"* .

- Instantiating C++ objects in shared memory: *"Using objects in shared memory for C++ application"* .

- A shared memory allocator and relative pointer: *"Taming Shared Memory"* .

Future improvements...

There are some Interprocess features that I would like to implement and some **Boost.Interprocess** code that can be much better. Let's see some ideas:

Win32 synchronization is too basic

Win32 version of shared mutexes and shared conditions are based on "spin and wait" atomic instructions. This leads to poor performance and does not manage any issues like priority inversions. We would need very serious help from threading experts on this. And I'm not sure that this can be achieved in user-level software. Posix based implementations use PTHREAD_PROCESS_SHARED attribute to place mutexes in shared memory, so there are no such problems. I'm not aware of any implementation that simulates PTHREAD_PROCESS_SHARED attribute for Win32. We should be able to construct these primitives in memory mapped files, so that we can get filesystem persistence just like with POSIX primitives.

Use of wide character names on Boost.Interprocess basic resources

Currently Interprocess only allows **char** based names for basic named objects. However, several operating systems use **wchar_t** names for resources (mapped files, for example). In the future Interprocess should try to present a portable narrow/wide char interface. To do this, it would be useful to have a boost wstring <-> string conversion utilities to translate resource names (escaping needed characters that can conflict with OS names) in a portable way. It would be interesting also the use of **boost::filesystem** paths to avoid operating system specific issues.

Security attributes

Boost.Interprocess does not define security attributes for shared memory and synchronization objects. Standard C++ also ignores security attributes with files so adding security attributes would require some serious work.

Future inter-process communications

Boost.Interprocess offers a process-shared message queue based on **Boost.Interprocess** primitives like mutexes and conditions. I would want to develop more mechanisms, like stream-oriented named fifo so that we can use it with a iostream-interface wrapper (we can imitate Unix pipes).

C++ needs more complex mechanisms and it would be nice to have a stream and datagram oriented PF_UNIX-like mechanism in C++. And for very fast inter-process remote calls Solaris doors is an interesting alternative to implement for C++. But the work to

implement PF_UNIX-like sockets and doors would be huge (and it might be difficult in a user-level library). Any network expert volunteer?

Boost.Interprocess Reference

Header <boost/interprocess/allocators/adaptive_pool.hpp>

Describes adaptive_pool pooled shared memory STL compatible allocator

```
namespace boost {
  namespace interprocess {
    template<typename T, typename SegmentManager, std::size_t NodesPerBlock,
             std::size_t MaxFreeBlocks, unsigned char OverheadPercent>
      class adaptive_pool;
    template<typename T, typename S, std::size_t NodesPerBlock, std::size_t F,
             unsigned char OP>
      bool operator==(const adaptive_pool< T, S, NodesPerBlock, F, OP > &,
                      const adaptive_pool< T, S, NodesPerBlock, F, OP > &);
    template<typename T, typename S, std::size_t NodesPerBlock, std::size_t F,
             unsigned char OP>
      bool operator!=(const adaptive_pool< T, S, NodesPerBlock, F, OP > &,
                      const adaptive_pool< T, S, NodesPerBlock, F, OP > &);
  }
}
```

Class template adaptive_pool

boost::interprocess::adaptive_pool

Synopsis

```cpp
// In header: <boost/interprocess/allocators/adaptive_pool.hpp>

template<typename T, typename SegmentManager, std::size_t NodesPerBlock,
         std::size_t MaxFreeBlocks, unsigned char OverheadPercent>
class adaptive_pool {
public:
  // types
  typedef implementation_defined::segment_manager segment_manager;
  typedef segment_manager::void_pointer          void_pointer;
  typedef implementation_defined::pointer         pointer;
  typedef implementation_defined::const_pointer   const_pointer;
  typedef T                                       value_type;
  typedef unspecified                             reference;
  typedef unspecified                             const_reference;
  typedef std::size_t                             size_type;
  typedef std::ptrdiff_t                          difference_type;

  // member classes/structs/unions
  template<typename T2>
  struct rebind {
    // types
    typedef adaptive_pool< T2, SegmentManager, NodesPerBlock, MaxFreeBlocks, OverheadPercent > oth↵
er;
  };

  // construct/copy/destruct
  adaptive_pool(segment_manager *);
  adaptive_pool(const adaptive_pool &);
  template<typename T2>
    adaptive_pool(const adaptive_pool< T2, SegmentManager, NodesPerBlock, MaxFreeBlocks, Over↵
headPercent > &);
  template<typename T, typename SegmentManager2, std::size_t N2,
           std::size_t F2, unsigned char OP2>
    adaptive_pool&
    operator=(const adaptive_pool< T2, SegmentManager2, N2, F2, OP2 > &);
  ~adaptive_pool();

  // public member functions
  void * get_node_pool() const;
  segment_manager * get_segment_manager() const;
  size_type max_size() const;
  pointer allocate(size_type, cvoid_pointer = 0);
  void deallocate(const pointer &, size_type);
  void deallocate_free_blocks();
  pointer address(reference) const;
  const_pointer address(const_reference) const;
  void construct(const pointer &, const_reference);
  void destroy(const pointer &);
  size_type size(const pointer &) const;
  std::pair< pointer, bool >
  allocation_command(boost::interprocess::allocation_type, size_type,
                     size_type, size_type &, const pointer & = 0);
  multiallocation_chain allocate_many(size_type, std::size_t);
  multiallocation_chain allocate_many(const size_type *, size_type);
  void deallocate_many(multiallocation_chain);
  pointer allocate_one();
```

```
   multiallocation_chain allocate_individual(std::size_t);
   void deallocate_one(const pointer &);
   void deallocate_individual(multiallocation_chain);

   // friend functions
   friend void swap(self_t &, self_t &);
};
```

Description

An STL node allocator that uses a segment manager as memory source. The internal pointer type will of the same type (raw, smart) as "typename SegmentManager::void_pointer" type. This allows placing the allocator in shared memory, memory mapped-files, etc...

This node allocator shares a segregated storage between all instances of adaptive_pool with equal sizeof(T) placed in the same segment group. NodesPerBlock is the number of nodes allocated at once when the allocator needs runs out of nodes. MaxFreeBlocks is the maximum number of totally free blocks that the adaptive node pool will hold. The rest of the totally free blocks will be deallocated with the segment manager.

OverheadPercent is the (approximated) maximum size overhead (1-20%) of the allocator: (memory usable for nodes / total memory allocated from the segment manager)

`adaptive_pool` public construct/copy/destruct

1.
```
adaptive_pool(segment_manager * segment_mngr);
```

Not assignable from other adaptive_pool Constructor from a segment manager. If not present, constructs a node pool. Increments the reference count of the associated node pool. Can throw boost::interprocess::bad_alloc

2.
```
adaptive_pool(const adaptive_pool & other);
```

Copy constructor from other adaptive_pool. Increments the reference count of the associated node pool. Never throws

3.
```
template<typename T2>
   adaptive_pool(const adaptive_pool< T2, SegmentManager, NodesPerBlock, MaxFreeBlocks, Over↵
headPercent > & other);
```

Copy constructor from related adaptive_pool. If not present, constructs a node pool. Increments the reference count of the associated node pool. Can throw boost::interprocess::bad_alloc

4.
```
template<typename T2, typename SegmentManager2, std::size_t N2,
         std::size_t F2, unsigned char OP2>
   adaptive_pool&
   operator=(const adaptive_pool< T2, SegmentManager2, N2, F2, OP2 > &);
```

Not assignable from related adaptive_pool

5.
```
~adaptive_pool();
```

Destructor, removes node_pool_t from memory if its reference count reaches to zero. Never throws

`adaptive_pool` public member functions

1.
```
void * get_node_pool() const;
```

Returns a pointer to the node pool. Never throws

2.
```
segment_manager * get_segment_manager() const;
```

Returns the segment manager. Never throws

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

Returns the number of elements that could be allocated. Never throws

4.
```
pointer allocate(size_type count, cvoid_pointer hint = 0);
```

Allocate memory for an array of count elements. Throws boost::interprocess::bad_alloc if there is no enough memory

5.
```
void deallocate(const pointer & ptr, size_type count);
```

Deallocate allocated memory. Never throws

6.
```
void deallocate_free_blocks();
```

Deallocates all free blocks of the pool

7.
```
pointer address(reference value) const;
```

Returns address of mutable object. Never throws

8.
```
const_pointer address(const_reference value) const;
```

Returns address of non mutable object. Never throws

9.
```
void construct(const pointer & ptr, const_reference v);
```

Copy construct an object. Throws if T's copy constructor throws

10.
```
void destroy(const pointer & ptr);
```

Destroys object. Throws if object's destructor throws

11.
```
size_type size(const pointer & p) const;
```

Returns maximum the number of objects the previously allocated memory pointed by p can hold. This size only works for memory allocated with allocate, allocation_command and allocate_many.

12.
```
std::pair< pointer, bool >
allocation_command(boost::interprocess::allocation_type command,
                   size_type limit_size, size_type preferred_size,
                   size_type & received_size, const pointer & reuse = 0);
```

13.
```
multiallocation_chain
allocate_many(size_type elem_size, std::size_t num_elements);
```

Allocates many elements of size elem_size in a contiguous block of memory. The minimum number to be allocated is min_elements, the preferred and maximum number is preferred_elements. The number of actually allocated elements is will be assigned to received_size. The elements must be deallocated with deallocate(...)

14.
```
multiallocation_chain
allocate_many(const size_type * elem_sizes, size_type n_elements);
```

Allocates n_elements elements, each one of size elem_sizes[i]in a contiguous block of memory. The elements must be deallocated

15.
```
void deallocate_many(multiallocation_chain chain);
```

Allocates many elements of size elem_size in a contiguous block of memory. The minimum number to be allocated is min_elements, the preferred and maximum number is preferred_elements. The number of actually allocated elements is will be assigned to received_size. The elements must be deallocated with deallocate(...)

16.
```
pointer allocate_one();
```

Allocates just one object. Memory allocated with this function must be deallocated only with deallocate_one(). Throws boost::interprocess::bad_alloc if there is no enough memory

17.
```
multiallocation_chain allocate_individual(std::size_t num_elements);
```

Allocates many elements of size == 1 in a contiguous block of memory. The minimum number to be allocated is min_elements, the preferred and maximum number is preferred_elements. The number of actually allocated elements is will be assigned to received_size. Memory allocated with this function must be deallocated only with deallocate_one().

18.
```
void deallocate_one(const pointer & p);
```

Deallocates memory previously allocated with allocate_one(). You should never use deallocate_one to deallocate memory allocated with other functions different from allocate_one(). Never throws

19.
```
void deallocate_individual(multiallocation_chain it);
```

Allocates many elements of size == 1 in a contiguous block of memory. The minimum number to be allocated is min_elements, the preferred and maximum number is preferred_elements. The number of actually allocated elements is will be assigned to received_size. Memory allocated with this function must be deallocated only with deallocate_one().

`adaptive_pool` **friend functions**

1.
```
friend void swap(self_t & alloc1, self_t & alloc2);
```

Swaps allocators. Does not throw. If each allocator is placed in a different memory segment, the result is undefined.

Struct template rebind

boost::interprocess::adaptive_pool::rebind

Synopsis

```
// In header: <boost/interprocess/allocators/adaptive_pool.hpp>


template<typename T2>
struct rebind {
  // types
  typedef adaptive_pool< T2, SegmentManager, NodesPerBlock, MaxFreeBlocks, OverheadPercent > other;
};
```

Description

Obtains adaptive_pool from adaptive_pool

Function template operator==

boost::interprocess::operator==

Synopsis

```
// In header: <boost/interprocess/allocators/adaptive_pool.hpp>


template<typename T, typename S, std::size_t NodesPerBlock, std::size_t F,
         unsigned char OP>
  bool operator==(const adaptive_pool< T, S, NodesPerBlock, F, OP > & alloc1,
                  const adaptive_pool< T, S, NodesPerBlock, F, OP > & alloc2);
```

Description

Equality test for same type of adaptive_pool

Function template operator!=

boost::interprocess::operator!=

Synopsis

```
// In header: <boost/interprocess/allocators/adaptive_pool.hpp>


template<typename T, typename S, std::size_t NodesPerBlock, std::size_t F,
         unsigned char OP>
  bool operator!=(const adaptive_pool< T, S, NodesPerBlock, F, OP > & alloc1,
                  const adaptive_pool< T, S, NodesPerBlock, F, OP > & alloc2);
```

Description

Inequality test for same type of adaptive_pool

Header <boost/interprocess/allocators/allocator.hpp>

Describes an allocator that allocates portions of fixed size memory buffer (shared memory, mapped file...)

```
namespace boost {
  namespace interprocess {
    template<typename T, typename SegmentManager> class allocator;
    template<typename T, typename SegmentManager>
      bool operator==(const allocator< T, SegmentManager > &,
                      const allocator< T, SegmentManager > &);
    template<typename T, typename SegmentManager>
      bool operator!=(const allocator< T, SegmentManager > &,
                      const allocator< T, SegmentManager > &);
  }
}
```

Class template allocator

boost::interprocess::allocator

Synopsis

```cpp
// In header: <boost/interprocess/allocators/allocator.hpp>

template<typename T, typename SegmentManager>
class allocator {
public:
  // types
  typedef SegmentManager                                  segment_manager;
  typedef SegmentManager::void_pointer                    void_pointer;
  typedef T                                               value_type;
  typedef boost::pointer_to_other< cvoid_ptr, T >::type   pointer;
  typedef boost::pointer_to_other< pointer, const T >::type const_pointer;
  typedef unspecified                                     reference;
  typedef unspecified                                     const_reference;
  typedef std::size_t                                     size_type;
  typedef std::ptrdiff_t                                  difference_type;
  typedef boost::interprocess::version_type< allocator, 2 > version;

  // member classes/structs/unions
  template<typename T2>
  struct rebind {
    // types
    typedef allocator< T2, SegmentManager > other;
  };

  // construct/copy/destruct
  allocator(segment_manager *);
  allocator(const allocator &);
  template<typename T2> allocator(const allocator< T2, SegmentManager > &);

  // public member functions
  segment_manager * get_segment_manager() const;
  pointer allocate(size_type, cvoid_ptr = 0);
  void deallocate(const pointer &, size_type);
  size_type max_size() const;
  size_type size(const pointer &) const;
  std::pair< pointer, bool >
  allocation_command(boost::interprocess::allocation_type, size_type,
                     size_type, size_type &, const pointer & = 0);
  multiallocation_chain allocate_many(size_type, std::size_t);
  multiallocation_chain allocate_many(const size_type *, size_type);
  void deallocate_many(multiallocation_chain);
  pointer allocate_one();
  multiallocation_chain allocate_individual(std::size_t);
  void deallocate_one(const pointer &);
  void deallocate_individual(multiallocation_chain);
  pointer address(reference) const;
  const_pointer address(const_reference) const;
  void construct(const pointer &, const_reference);
  void construct(const pointer &);
  void destroy(const pointer &);

  // friend functions
  friend void swap(self_t &, self_t &);
};
```

Description

An STL compatible allocator that uses a segment manager as memory source. The internal pointer type will of the same type (raw, smart) as "typename SegmentManager::void_pointer" type. This allows placing the allocator in shared memory, memory mapped-files, etc...

`allocator` public construct/copy/destruct

1.
```
allocator(segment_manager * segment_mngr);
```

Constructor from the segment manager. Never throws

2.
```
allocator(const allocator & other);
```

Constructor from other allocator. Never throws

3.
```
template<typename T2> allocator(const allocator< T2, SegmentManager > & other);
```

Constructor from related allocator. Never throws

`allocator` public member functions

1.
```
segment_manager * get_segment_manager() const;
```

Returns the segment manager. Never throws

2.
```
pointer allocate(size_type count, cvoid_ptr hint = 0);
```

Allocates memory for an array of count elements. Throws boost::interprocess::bad_alloc if there is no enough memory

3.
```
void deallocate(const pointer & ptr, size_type);
```

Deallocates memory previously allocated. Never throws

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

Returns the number of elements that could be allocated. Never throws

5.
```
size_type size(const pointer & p) const;
```

Returns maximum the number of objects the previously allocated memory pointed by p can hold. This size only works for memory allocated with allocate, allocation_command and allocate_many.

6.
```
std::pair< pointer, bool >
allocation_command(boost::interprocess::allocation_type command,
                   size_type limit_size, size_type preferred_size,
                   size_type & received_size, const pointer & reuse = 0);
```

7.
```
multiallocation_chain
allocate_many(size_type elem_size, std::size_t num_elements);
```

165

Allocates many elements of size elem_size in a contiguous block of memory. The minimum number to be allocated is min_elements, the preferred and maximum number is preferred_elements. The number of actually allocated elements is will be assigned to received_size. The elements must be deallocated with deallocate(...)

8.
```
multiallocation_chain
allocate_many(const size_type * elem_sizes, size_type n_elements);
```

Allocates n_elements elements, each one of size elem_sizes[i]in a contiguous block of memory. The elements must be deallocated

9.
```
void deallocate_many(multiallocation_chain chain);
```

Allocates many elements of size elem_size in a contiguous block of memory. The minimum number to be allocated is min_elements, the preferred and maximum number is preferred_elements. The number of actually allocated elements is will be assigned to received_size. The elements must be deallocated with deallocate(...)

10.
```
pointer allocate_one();
```

Allocates just one object. Memory allocated with this function must be deallocated only with deallocate_one(). Throws boost::interprocess::bad_alloc if there is no enough memory

11.
```
multiallocation_chain allocate_individual(std::size_t num_elements);
```

Allocates many elements of size == 1 in a contiguous block of memory. The minimum number to be allocated is min_elements, the preferred and maximum number is preferred_elements. The number of actually allocated elements is will be assigned to received_size. Memory allocated with this function must be deallocated only with deallocate_one().

12.
```
void deallocate_one(const pointer & p);
```

Deallocates memory previously allocated with allocate_one(). You should never use deallocate_one to deallocate memory allocated with other functions different from allocate_one(). Never throws

13.
```
void deallocate_individual(multiallocation_chain chain);
```

Allocates many elements of size == 1 in a contiguous block of memory. The minimum number to be allocated is min_elements, the preferred and maximum number is preferred_elements. The number of actually allocated elements is will be assigned to received_size. Memory allocated with this function must be deallocated only with deallocate_one().

14.
```
pointer address(reference value) const;
```

Returns address of mutable object. Never throws

15.
```
const_pointer address(const_reference value) const;
```

Returns address of non mutable object. Never throws

16.
```
void construct(const pointer & ptr, const_reference v);
```

Copy construct an object Throws if T's copy constructor throws

17.
```
void construct(const pointer & ptr);
```

Default construct an object. Throws if T's default constructor throws

18.
```
void destroy(const pointer & ptr);
```

Destroys object. Throws if object's destructor throws

allocator **friend functions**

1.
```
friend void swap(self_t & alloc1, self_t & alloc2);
```

Swap segment manager. Does not throw. If each allocator is placed in different memory segments, the result is undefined.

Struct template rebind

boost::interprocess::allocator::rebind

Synopsis

```
// In header: <boost/interprocess/allocators/allocator.hpp>


template<typename T2>
struct rebind {
  // types
  typedef allocator< T2, SegmentManager > other;
};
```

Description

Obtains an allocator that allocates objects of type T2

Function template operator==

boost::interprocess::operator==

Synopsis

```
// In header: <boost/interprocess/allocators/allocator.hpp>


template<typename T, typename SegmentManager>
  bool operator==(const allocator< T, SegmentManager > & alloc1,
                  const allocator< T, SegmentManager > & alloc2);
```

Description

Equality test for same type of allocator

Function template operator!=

boost::interprocess::operator!=

Synopsis

```
// In header: <boost/interprocess/allocators/allocator.hpp>


template<typename T, typename SegmentManager>
  bool operator!=(const allocator< T, SegmentManager > & alloc1,
                  const allocator< T, SegmentManager > & alloc2);
```

Description

Inequality test for same type of allocator

Header <boost/interprocess/allocators/cached_adaptive_pool.hpp>

Describes cached_adaptive_pool pooled shared memory STL compatible allocator

```
namespace boost {
  namespace interprocess {
    template<typename T, typename SegmentManager, std::size_t NodesPerBlock,
             std::size_t MaxFreeBlocks, unsigned char OverheadPercent>
      class cached_adaptive_pool;
    template<typename T, typename S, std::size_t NodesPerBlock, std::size_t F,
             std::size_t OP>
      bool operator==(const cached_adaptive_pool< T, S, NodesPerBlock, F, OP > &,
                      const cached_adaptive_pool< T, S, NodesPerBlock, F, OP > &);
    template<typename T, typename S, std::size_t NodesPerBlock, std::size_t F,
             std::size_t OP>
      bool operator!=(const cached_adaptive_pool< T, S, NodesPerBlock, F, OP > &,
                      const cached_adaptive_pool< T, S, NodesPerBlock, F, OP > &);
  }
}
```

Class template cached_adaptive_pool

boost::interprocess::cached_adaptive_pool

Synopsis

```
// In header: <boost/interprocess/allocators/cached_adaptive_pool.hpp>

template<typename T, typename SegmentManager, std::size_t NodesPerBlock,
         std::size_t MaxFreeBlocks, unsigned char OverheadPercent>
class cached_adaptive_pool {
public:
  // types
  typedef implementation_defined::segment_manager segment_manager;
  typedef segment_manager::void_pointer          void_pointer;
  typedef implementation_defined::pointer         pointer;
  typedef implementation_defined::const_pointer   const_pointer;
  typedef T                                       value_type;
  typedef unspecified                             reference;
  typedef unspecified                             const_reference;
  typedef std::size_t                             size_type;
  typedef std::ptrdiff_t                          difference_type;

  // member classes/structs/unions
  template<typename T2>
  struct rebind {
    // types
    typedef cached_adaptive_pool< T2, SegmentManager, NodesPerBlock, MaxFreeBlocks, OverheadPer↵
cent > other;
  };

  // construct/copy/destruct
  cached_adaptive_pool(segment_manager *);
  cached_adaptive_pool(const cached_adaptive_pool &);
  template<typename T2>
    cached_adaptive_pool(const cached_adaptive_pool< T2, SegmentManager, NodesPerBlock, MaxFreeB↵
locks, OverheadPercent > &);
  template<typename T2, typename SegmentManager2, std::size_t N2,
           std::size_t F2, unsigned char OP2>
    cached_adaptive_pool&
    operator=(const cached_adaptive_pool< T2, SegmentManager2, N2, F2, OP2 > &);
  cached_adaptive_pool& operator=(const cached_adaptive_pool &);
  ~cached_adaptive_pool();

  // public member functions
  node_pool_t * get_node_pool() const;
  segment_manager * get_segment_manager() const;
  size_type max_size() const;
  pointer allocate(size_type, cvoid_pointer = 0);
  void deallocate(const pointer &, size_type);
  void deallocate_free_blocks();
  pointer address(reference) const;
  const_pointer address(const_reference) const;
  void construct(const pointer &, const_reference);
  void destroy(const pointer &);
  size_type size(const pointer &) const;
  std::pair< pointer, bool >
  allocation_command(boost::interprocess::allocation_type, size_type,
                     size_type, size_type &, const pointer & = 0);
  multiallocation_chain allocate_many(size_type, std::size_t);
  multiallocation_chain allocate_many(const size_type *, size_type);
  void deallocate_many(multiallocation_chain);
```

```
  pointer allocate_one();
  multiallocation_chain allocate_individual(std::size_t);
  void deallocate_one(const pointer &);
  void deallocate_individual(multiallocation_chain);
  void set_max_cached_nodes(std::size_t);
  std::size_t get_max_cached_nodes() const;

  // friend functions
  friend void swap(self_t &, self_t &);
};
```

Description

An STL node allocator that uses a segment manager as memory source. The internal pointer type will of the same type (raw, smart) as "typename SegmentManager::void_pointer" type. This allows placing the allocator in shared memory, memory mapped-files, etc...

This node allocator shares a segregated storage between all instances of cached_adaptive_pool with equal sizeof(T) placed in the same memory segment. But also caches some nodes privately to avoid some synchronization overhead.

NodesPerBlock is the minimum number of nodes of nodes allocated at once when the allocator needs runs out of nodes. MaxFreeBlocks is the maximum number of totally free blocks that the adaptive node pool will hold. The rest of the totally free blocks will be deallocated with the segment manager.

OverheadPercent is the (approximated) maximum size overhead (1-20%) of the allocator: (memory usable for nodes / total memory allocated from the segment manager)

`cached_adaptive_pool` public construct/copy/destruct

1.
```
cached_adaptive_pool(segment_manager * segment_mngr);
```

Constructor from a segment manager. If not present, constructs a node pool. Increments the reference count of the associated node pool. Can throw boost::interprocess::bad_alloc

2.
```
cached_adaptive_pool(const cached_adaptive_pool & other);
```

Copy constructor from other cached_adaptive_pool. Increments the reference count of the associated node pool. Never throws

3.
```
template<typename T2>
  cached_adaptive_pool(const cached_adaptive_pool< T2, SegmentManager, NodesPerBlock, MaxFreeB↵
locks, OverheadPercent > & other);
```

Copy constructor from related cached_adaptive_pool. If not present, constructs a node pool. Increments the reference count of the associated node pool. Can throw boost::interprocess::bad_alloc

4.
```
template<typename T2, typename SegmentManager2, std::size_t N2,
         std::size_t F2, unsigned char OP2>
  cached_adaptive_pool&
  operator=(const cached_adaptive_pool< T2, SegmentManager2, N2, F2, OP2 > &);
```

Not assignable from related cached_adaptive_pool

5.
```
cached_adaptive_pool& operator=(const cached_adaptive_pool &);
```

Not assignable from other cached_adaptive_pool

6.
```
~cached_adaptive_pool();
```

Destructor, removes node_pool_t from memory if its reference count reaches to zero. Never throws

cached_adaptive_pool **public member functions**

1.
```
node_pool_t * get_node_pool() const;
```

Returns a pointer to the node pool. Never throws

2.
```
segment_manager * get_segment_manager() const;
```

Returns the segment manager. Never throws

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

Returns the number of elements that could be allocated. Never throws

4.
```
pointer allocate(size_type count, cvoid_pointer hint = 0);
```

Allocate memory for an array of count elements. Throws boost::interprocess::bad_alloc if there is no enough memory

5.
```
void deallocate(const pointer & ptr, size_type count);
```

Deallocate allocated memory. Never throws

6.
```
void deallocate_free_blocks();
```

Deallocates all free blocks of the pool

7.
```
pointer address(reference value) const;
```

Returns address of mutable object. Never throws

8.
```
const_pointer address(const_reference value) const;
```

Returns address of non mutable object. Never throws

9.
```
void construct(const pointer & ptr, const_reference v);
```

Copy construct an object. Throws if T's copy constructor throws

10.
```
void destroy(const pointer & ptr);
```

Destroys object. Throws if object's destructor throws

11.
```
size_type size(const pointer & p) const;
```

Returns maximum the number of objects the previously allocated memory pointed by p can hold. This size only works for memory allocated with allocate, allocation_command and allocate_many.

12.
```
std::pair< pointer, bool >
allocation_command(boost::interprocess::allocation_type command,
                   size_type limit_size, size_type preferred_size,
                   size_type & received_size, const pointer & reuse = 0);
```

13.
```
multiallocation_chain
allocate_many(size_type elem_size, std::size_t num_elements);
```

Allocates many elements of size elem_size in a contiguous block of memory. The minimum number to be allocated is min_elements, the preferred and maximum number is preferred_elements. The number of actually allocated elements is will be assigned to received_size. The elements must be deallocated with deallocate(...)

14.
```
multiallocation_chain
allocate_many(const size_type * elem_sizes, size_type n_elements);
```

Allocates n_elements elements, each one of size elem_sizes[i]in a contiguous block of memory. The elements must be deallocated

15.
```
void deallocate_many(multiallocation_chain chain);
```

Allocates many elements of size elem_size in a contiguous block of memory. The minimum number to be allocated is min_elements, the preferred and maximum number is preferred_elements. The number of actually allocated elements is will be assigned to received_size. The elements must be deallocated with deallocate(...)

16.
```
pointer allocate_one();
```

Allocates just one object. Memory allocated with this function must be deallocated only with deallocate_one(). Throws boost::interprocess::bad_alloc if there is no enough memory

17.
```
multiallocation_chain allocate_individual(std::size_t num_elements);
```

Allocates many elements of size == 1 in a contiguous block of memory. The minimum number to be allocated is min_elements, the preferred and maximum number is preferred_elements. The number of actually allocated elements is will be assigned to received_size. Memory allocated with this function must be deallocated only with deallocate_one().

18.
```
void deallocate_one(const pointer & p);
```

Deallocates memory previously allocated with allocate_one(). You should never use deallocate_one to deallocate memory allocated with other functions different from allocate_one(). Never throws

19.
```
void deallocate_individual(multiallocation_chain chain);
```

Allocates many elements of size == 1 in a contiguous block of memory. The minimum number to be allocated is min_elements, the preferred and maximum number is preferred_elements. The number of actually allocated elements is will be assigned to received_size. Memory allocated with this function must be deallocated only with deallocate_one().

20.
```
void set_max_cached_nodes(std::size_t newmax);
```

Sets the new max cached nodes value. This can provoke deallocations if "newmax" is less than current cached nodes. Never throws

21.
```
std::size_t get_max_cached_nodes() const;
```

Returns the max cached nodes parameter. Never throws

`cached_adaptive_pool` friend functions

1.
```
friend void swap(self_t & alloc1, self_t & alloc2);
```

Swaps allocators. Does not throw. If each allocator is placed in a different memory segment, the result is undefined.

Struct template rebind

boost::interprocess::cached_adaptive_pool::rebind

Synopsis

```
// In header: <boost/interprocess/allocators/cached_adaptive_pool.hpp>


template<typename T2>
struct rebind {
  // types
  typedef cached_adaptive_pool< T2, SegmentManager, NodesPerBlock, MaxFreeBlocks, OverheadPer↵
cent > other;
};
```

Description

Obtains cached_adaptive_pool from cached_adaptive_pool

Function template operator==

boost::interprocess::operator==

Synopsis

```
// In header: <boost/interprocess/allocators/cached_adaptive_pool.hpp>


template<typename T, typename S, std::size_t NodesPerBlock, std::size_t F,
         std::size_t OP>
  bool operator==(const cached_adaptive_pool< T, S, NodesPerBlock, F, OP > & alloc1,
                  const cached_adaptive_pool< T, S, NodesPerBlock, F, OP > & alloc2);
```

Description

Equality test for same type of cached_adaptive_pool

Function template operator!=

boost::interprocess::operator!=

Synopsis

```
// In header: <boost/interprocess/allocators/cached_adaptive_pool.hpp>


template<typename T, typename S, std::size_t NodesPerBlock, std::size_t F,
         std::size_t OP>
  bool operator!=(const cached_adaptive_pool< T, S, NodesPerBlock, F, OP > & alloc1,
                  const cached_adaptive_pool< T, S, NodesPerBlock, F, OP > & alloc2);
```

Description

Inequality test for same type of cached_adaptive_pool

Header <boost/interprocess/allocators/cached_node_allocator.hpp>

Describes cached_cached_node_allocator pooled shared memory STL compatible allocator

```
namespace boost {
  namespace interprocess {
    template<typename T, typename SegmentManager, std::size_t NodesPerBlock>
      class cached_node_allocator;
    template<typename T, typename S, std::size_t NPC>
      bool operator==(const cached_node_allocator< T, S, NPC > &,
                      const cached_node_allocator< T, S, NPC > &);
    template<typename T, typename S, std::size_t NPC>
      bool operator!=(const cached_node_allocator< T, S, NPC > &,
                      const cached_node_allocator< T, S, NPC > &);
  }
}
```

Class template cached_node_allocator

boost::interprocess::cached_node_allocator

Synopsis

```
// In header: <boost/interprocess/allocators/cached_node_allocator.hpp>

template<typename T, typename SegmentManager, std::size_t NodesPerBlock>
class cached_node_allocator {
public:
  // types
  typedef implementation_defined::segment_manager segment_manager;
  typedef segment_manager::void_pointer           void_pointer;
  typedef implementation_defined::pointer         pointer;
  typedef implementation_defined::const_pointer   const_pointer;
  typedef T                                       value_type;
  typedef unspecified                             reference;
  typedef unspecified                             const_reference;
  typedef std::size_t                             size_type;
  typedef std::ptrdiff_t                          difference_type;

  // member classes/structs/unions
  template<typename T2>
  struct rebind {
    // types
    typedef cached_node_allocator< T2, SegmentManager > other;
  };

  // construct/copy/destruct
  cached_node_allocator(segment_manager *);
  cached_node_allocator(const cached_node_allocator &);
  template<typename T2>
    cached_node_allocator(const cached_node_allocator< T2, SegmentManager, NodesPerBlock > &);
  template<typename T2, typename SegmentManager2, std::size_t N2>
    cached_node_allocator&
    operator=(const cached_node_allocator< T2, SegmentManager2, N2 > &);
  cached_node_allocator& operator=(const cached_node_allocator &);
  ~cached_node_allocator();

  // public member functions
  node_pool_t * get_node_pool() const;
  segment_manager * get_segment_manager() const;
  size_type max_size() const;
  pointer allocate(size_type, cvoid_pointer = 0);
  void deallocate(const pointer &, size_type);
  void deallocate_free_blocks();
  pointer address(reference) const;
  const_pointer address(const_reference) const;
  void construct(const pointer &, const_reference);
  void destroy(const pointer &);
  size_type size(const pointer &) const;
  std::pair< pointer, bool >
  allocation_command(boost::interprocess::allocation_type, size_type,
                     size_type, size_type &, const pointer & = 0);
  multiallocation_chain allocate_many(size_type, std::size_t);
  multiallocation_chain allocate_many(const size_type *, size_type);
  void deallocate_many(multiallocation_chain);
  pointer allocate_one();
  multiallocation_chain allocate_individual(std::size_t);
  void deallocate_one(const pointer &);
  void deallocate_individual(multiallocation_chain);
```

```
  void set_max_cached_nodes(std::size_t);
  std::size_t get_max_cached_nodes() const;

  // friend functions
  friend void swap(self_t &, self_t &);
};
```

Description

`cached_node_allocator` public construct/copy/destruct

1.
```
cached_node_allocator(segment_manager * segment_mngr);
```

Constructor from a segment manager. If not present, constructs a node pool. Increments the reference count of the associated node pool. Can throw boost::interprocess::bad_alloc

2.
```
cached_node_allocator(const cached_node_allocator & other);
```

Copy constructor from other cached_node_allocator. Increments the reference count of the associated node pool. Never throws

3.
```
template<typename T2>
  cached_node_allocator(const cached_node_allocator< T2, SegmentManager, NodesPerBlock > &
other);
```

Copy constructor from related cached_node_allocator. If not present, constructs a node pool. Increments the reference count of the associated node pool. Can throw boost::interprocess::bad_alloc

4.
```
template<typename T2, typename SegmentManager2, std::size_t N2>
  cached_node_allocator&
  operator=(const cached_node_allocator< T2, SegmentManager2, N2 > &);
```

Not assignable from related cached_node_allocator

5.
```
cached_node_allocator& operator=(const cached_node_allocator &);
```

Not assignable from other cached_node_allocator

6.
```
~cached_node_allocator();
```

Destructor, removes node_pool_t from memory if its reference count reaches to zero. Never throws

`cached_node_allocator` public member functions

1.
```
node_pool_t * get_node_pool() const;
```

Returns a pointer to the node pool. Never throws

2.
```
segment_manager * get_segment_manager() const;
```

Returns the segment manager. Never throws

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

Returns the number of elements that could be allocated. Never throws

4.
```
pointer allocate(size_type count, cvoid_pointer hint = 0);
```

Allocate memory for an array of count elements. Throws boost::interprocess::bad_alloc if there is no enough memory

5.
```
void deallocate(const pointer & ptr, size_type count);
```

Deallocate allocated memory. Never throws

6.
```
void deallocate_free_blocks();
```

Deallocates all free blocks of the pool

7.
```
pointer address(reference value) const;
```

Returns address of mutable object. Never throws

8.
```
const_pointer address(const_reference value) const;
```

Returns address of non mutable object. Never throws

9.
```
void construct(const pointer & ptr, const_reference v);
```

Default construct an object. Throws if T's default constructor throws

10.
```
void destroy(const pointer & ptr);
```

Destroys object. Throws if object's destructor throws

11.
```
size_type size(const pointer & p) const;
```

Returns maximum the number of objects the previously allocated memory pointed by p can hold. This size only works for memory allocated with allocate, allocation_command and allocate_many.

12.
```
std::pair< pointer, bool >
allocation_command(boost::interprocess::allocation_type command,
                   size_type limit_size, size_type preferred_size,
                   size_type & received_size, const pointer & reuse = 0);
```

13.
```
multiallocation_chain
allocate_many(size_type elem_size, std::size_t num_elements);
```

Allocates many elements of size elem_size in a contiguous block of memory. The minimum number to be allocated is min_elements, the preferred and maximum number is preferred_elements. The number of actually allocated elements is will be assigned to received_size. The elements must be deallocated with deallocate(...)

14.
```
multiallocation_chain
allocate_many(const size_type * elem_sizes, size_type n_elements);
```

Allocates n_elements elements, each one of size elem_sizes[i]in a contiguous block of memory. The elements must be deallocated

---

15.
```
void deallocate_many(multiallocation_chain chain);
```

Allocates many elements of size elem_size in a contiguous block of memory. The minimum number to be allocated is min_elements, the preferred and maximum number is preferred_elements. The number of actually allocated elements is will be assigned to received_size. The elements must be deallocated with deallocate(...)

16.
```
pointer allocate_one();
```

Allocates just one object. Memory allocated with this function must be deallocated only with deallocate_one(). Throws boost::interprocess::bad_alloc if there is no enough memory

17.
```
multiallocation_chain allocate_individual(std::size_t num_elements);
```

Allocates many elements of size == 1 in a contiguous block of memory. The minimum number to be allocated is min_elements, the preferred and maximum number is preferred_elements. The number of actually allocated elements is will be assigned to received_size. Memory allocated with this function must be deallocated only with deallocate_one().

18.
```
void deallocate_one(const pointer & p);
```

Deallocates memory previously allocated with allocate_one(). You should never use deallocate_one to deallocate memory allocated with other functions different from allocate_one(). Never throws

19.
```
void deallocate_individual(multiallocation_chain it);
```

Allocates many elements of size == 1 in a contiguous block of memory. The minimum number to be allocated is min_elements, the preferred and maximum number is preferred_elements. The number of actually allocated elements is will be assigned to received_size. Memory allocated with this function must be deallocated only with deallocate_one().

20.
```
void set_max_cached_nodes(std::size_t newmax);
```

Sets the new max cached nodes value. This can provoke deallocations if "newmax" is less than current cached nodes. Never throws

21.
```
std::size_t get_max_cached_nodes() const;
```

Returns the max cached nodes parameter. Never throws

cached_node_allocator friend functions

1.
```
friend void swap(self_t & alloc1, self_t & alloc2);
```

Swaps allocators. Does not throw. If each allocator is placed in a different memory segment, the result is undefined.

Struct template rebind

boost::interprocess::cached_node_allocator::rebind

Synopsis

```
// In header: <boost/interprocess/allocators/cached_node_allocator.hpp>


template<typename T2>
struct rebind {
  // types
  typedef cached_node_allocator< T2, SegmentManager > other;
};
```

Description

Obtains cached_node_allocator from cached_node_allocator

Function template operator==

boost::interprocess::operator==

Synopsis

```
// In header: <boost/interprocess/allocators/cached_node_allocator.hpp>


template<typename T, typename S, std::size_t NPC>
  bool operator==(const cached_node_allocator< T, S, NPC > & alloc1,
                  const cached_node_allocator< T, S, NPC > & alloc2);
```

Description

Equality test for same type of cached_node_allocator

Function template operator!=

boost::interprocess::operator!=

Synopsis

```
// In header: <boost/interprocess/allocators/cached_node_allocator.hpp>


template<typename T, typename S, std::size_t NPC>
  bool operator!=(const cached_node_allocator< T, S, NPC > & alloc1,
                  const cached_node_allocator< T, S, NPC > & alloc2);
```

Description

Inequality test for same type of cached_node_allocator

Header <boost/interprocess/allocators/node_allocator.hpp>

Describes node_allocator pooled shared memory STL compatible allocator

```
namespace boost {
  namespace interprocess {
    template<typename T, typename SegmentManager, std::size_t NodesPerBlock>
      class node_allocator;
    template<typename T, typename S, std::size_t NPC>
      bool operator==(const node_allocator< T, S, NPC > &,
                      const node_allocator< T, S, NPC > &);
    template<typename T, typename S, std::size_t NPC>
      bool operator!=(const node_allocator< T, S, NPC > &,
                      const node_allocator< T, S, NPC > &);
  }
}
```

Class template node_allocator

boost::interprocess::node_allocator

Synopsis

```cpp
// In header: <boost/interprocess/allocators/node_allocator.hpp>


template<typename T, typename SegmentManager, std::size_t NodesPerBlock>
class node_allocator {
public:
  // types
  typedef implementation_defined::segment_manager  segment_manager;
  typedef segment_manager::void_pointer            void_pointer;
  typedef implementation_defined::pointer          pointer;
  typedef implementation_defined::const_pointer    const_pointer;
  typedef T                                        value_type;
  typedef unspecified                              reference;
  typedef unspecified                              const_reference;
  typedef std::size_t                              size_type;
  typedef std::ptrdiff_t                           difference_type;

  // member classes/structs/unions
  template<typename T2>
  struct rebind {
    // types
    typedef node_allocator< T2, SegmentManager, NodesPerBlock > other;
  };

  // construct/copy/destruct
  node_allocator(segment_manager *);
  node_allocator(const node_allocator &);
  template<typename T2>
    node_allocator(const node_allocator< T2, SegmentManager, NodesPerBlock > &);
  template<typename T2, typename SegmentManager2, std::size_t N2>
    node_allocator&
    operator=(const node_allocator< T2, SegmentManager2, N2 > &);
  ~node_allocator();

  // public member functions
  void * get_node_pool() const;
  segment_manager * get_segment_manager() const;
  size_type max_size() const;
  pointer allocate(size_type, cvoid_pointer = 0);
  void deallocate(const pointer &, size_type);
  void deallocate_free_blocks();
  pointer address(reference) const;
  const_pointer address(const_reference) const;
  void construct(const pointer &, const_reference);
  void destroy(const pointer &);
  size_type size(const pointer &) const;
  std::pair< pointer, bool >
  allocation_command(boost::interprocess::allocation_type, size_type,
                     size_type, size_type &, const pointer & = 0);
  multiallocation_chain allocate_many(size_type, std::size_t);
  multiallocation_chain allocate_many(const size_type *, size_type);
  void deallocate_many(multiallocation_chain);
  pointer allocate_one();
  multiallocation_chain allocate_individual(std::size_t);
```

```
  void deallocate_one(const pointer &);
  void deallocate_individual(multiallocation_chain);

  // friend functions
  friend void swap(self_t &, self_t &);
};
```

Description

An STL node allocator that uses a segment manager as memory source. The internal pointer type will of the same type (raw, smart) as "typename SegmentManager::void_pointer" type. This allows placing the allocator in shared memory, memory mapped-files, etc... This node allocator shares a segregated storage between all instances of node_allocator with equal sizeof(T) placed in the same segment group. NodesPerBlock is the number of nodes allocated at once when the allocator needs runs out of nodes

`node_allocator` public construct/copy/destruct

1.
```
node_allocator(segment_manager * segment_mngr);
```

Not assignable from other node_allocator Constructor from a segment manager. If not present, constructs a node pool. Increments the reference count of the associated node pool. Can throw boost::interprocess::bad_alloc

2.
```
node_allocator(const node_allocator & other);
```

Copy constructor from other node_allocator. Increments the reference count of the associated node pool. Never throws

3.
```
template<typename T2>
   node_allocator(const node_allocator< T2, SegmentManager, NodesPerBlock > & other);
```

Copy constructor from related node_allocator. If not present, constructs a node pool. Increments the reference count of the associated node pool. Can throw boost::interprocess::bad_alloc

4.
```
template<typename T2, typename SegmentManager2, std::size_t N2>
   node_allocator& operator=(const node_allocator< T2, SegmentManager2, N2 > &);
```

Not assignable from related node_allocator

5.
```
~node_allocator();
```

Destructor, removes node_pool_t from memory if its reference count reaches to zero. Never throws

`node_allocator` public member functions

1.
```
void * get_node_pool() const;
```

Returns a pointer to the node pool. Never throws

2.
```
segment_manager * get_segment_manager() const;
```

Returns the segment manager. Never throws

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

Returns the number of elements that could be allocated. Never throws

4.
```
pointer allocate(size_type count, cvoid_pointer hint = 0);
```

Allocate memory for an array of count elements. Throws boost::interprocess::bad_alloc if there is no enough memory

5.
```
void deallocate(const pointer & ptr, size_type count);
```

Deallocate allocated memory. Never throws

6.
```
void deallocate_free_blocks();
```

Deallocates all free blocks of the pool

7.
```
pointer address(reference value) const;
```

Returns address of mutable object. Never throws

8.
```
const_pointer address(const_reference value) const;
```

Returns address of non mutable object. Never throws

9.
```
void construct(const pointer & ptr, const_reference v);
```

Copy construct an object. Throws if T's copy constructor throws

10.
```
void destroy(const pointer & ptr);
```

Destroys object. Throws if object's destructor throws

11.
```
size_type size(const pointer & p) const;
```

Returns maximum the number of objects the previously allocated memory pointed by p can hold. This size only works for memory allocated with allocate, allocation_command and allocate_many.

12.
```
std::pair< pointer, bool >
   allocation_command(boost::interprocess::allocation_type command,
                      size_type limit_size, size_type preferred_size,
                      size_type & received_size, const pointer & reuse = 0);
```

13.
```
multiallocation_chain
   allocate_many(size_type elem_size, std::size_t num_elements);
```

Allocates many elements of size elem_size in a contiguous block of memory. The minimum number to be allocated is min_elements, the preferred and maximum number is preferred_elements. The number of actually allocated elements is will be assigned to received_size. The elements must be deallocated with deallocate(...)

14.
```
multiallocation_chain
   allocate_many(const size_type * elem_sizes, size_type n_elements);
```

Allocates n_elements elements, each one of size elem_sizes[i]in a contiguous block of memory. The elements must be deallocated

15.
```
void deallocate_many(multiallocation_chain chain);
```

Allocates many elements of size elem_size in a contiguous block of memory. The minimum number to be allocated is min_elements, the preferred and maximum number is preferred_elements. The number of actually allocated elements is will be assigned to received_size. The elements must be deallocated with deallocate(...)

16.
```
pointer allocate_one();
```

Allocates just one object. Memory allocated with this function must be deallocated only with deallocate_one(). Throws boost::interprocess::bad_alloc if there is no enough memory

17.
```
multiallocation_chain allocate_individual(std::size_t num_elements);
```

Allocates many elements of size == 1 in a contiguous block of memory. The minimum number to be allocated is min_elements, the preferred and maximum number is preferred_elements. The number of actually allocated elements is will be assigned to received_size. Memory allocated with this function must be deallocated only with deallocate_one().

18.
```
void deallocate_one(const pointer & p);
```

Deallocates memory previously allocated with allocate_one(). You should never use deallocate_one to deallocate memory allocated with other functions different from allocate_one(). Never throws

19.
```
void deallocate_individual(multiallocation_chain chain);
```

Allocates many elements of size == 1 in a contiguous block of memory. The minimum number to be allocated is min_elements, the preferred and maximum number is preferred_elements. The number of actually allocated elements is will be assigned to received_size. Memory allocated with this function must be deallocated only with deallocate_one().

`node_allocator` friend functions

1.
```
friend void swap(self_t & alloc1, self_t & alloc2);
```

Swaps allocators. Does not throw. If each allocator is placed in a different memory segment, the result is undefined.

Struct template rebind

boost::interprocess::node_allocator::rebind

Synopsis

```
// In header: <boost/interprocess/allocators/node_allocator.hpp>


template<typename T2>
struct rebind {
  // types
  typedef node_allocator< T2, SegmentManager, NodesPerBlock > other;
};
```

Description

Obtains node_allocator from node_allocator

Function template operator==

boost::interprocess::operator==

Synopsis

```
// In header: <boost/interprocess/allocators/node_allocator.hpp>


template<typename T, typename S, std::size_t NPC>
  bool operator==(const node_allocator< T, S, NPC > & alloc1,
                  const node_allocator< T, S, NPC > & alloc2);
```

Description

Equality test for same type of node_allocator

Function template operator!=

boost::interprocess::operator!=

Synopsis

```
// In header: <boost/interprocess/allocators/node_allocator.hpp>


template<typename T, typename S, std::size_t NPC>
  bool operator!=(const node_allocator< T, S, NPC > & alloc1,
                  const node_allocator< T, S, NPC > & alloc2);
```

Description

Inequality test for same type of node_allocator

Header <boost/interprocess/allocators/private_adaptive_pool.hpp>

Describes private_adaptive_pool_base pooled shared memory STL compatible allocator

```
namespace boost {
  namespace interprocess {
    template<typename T, typename SegmentManager, std::size_t NodesPerBlock,
             std::size_t MaxFreeBlocks, unsigned char OverheadPercent>
      class private_adaptive_pool;
    template<typename T, typename S, std::size_t NodesPerBlock, std::size_t F,
             unsigned char OP>
      bool operator==(const private_adaptive_pool< T, S, NodesPerBlock, F, OP > &,
                      const private_adaptive_pool< T, S, NodesPerBlock, F, OP > &);
    template<typename T, typename S, std::size_t NodesPerBlock, std::size_t F,
             unsigned char OP>
      bool operator!=(const private_adaptive_pool< T, S, NodesPerBlock, F, OP > &,
                      const private_adaptive_pool< T, S, NodesPerBlock, F, OP > &);
  }
}
```

Class template private_adaptive_pool

boost::interprocess::private_adaptive_pool

Synopsis

```cpp
// In header: <boost/interprocess/allocators/private_adaptive_pool.hpp>

template<typename T, typename SegmentManager, std::size_t NodesPerBlock,
         std::size_t MaxFreeBlocks, unsigned char OverheadPercent>
class private_adaptive_pool {
public:
  // types
  typedef implementation_defined::segment_manager segment_manager;
  typedef segment_manager::void_pointer          void_pointer;
  typedef implementation_defined::pointer         pointer;
  typedef implementation_defined::const_pointer   const_pointer;
  typedef T                                       value_type;
  typedef unspecified                             reference;
  typedef unspecified                             const_reference;
  typedef std::size_t                             size_type;
  typedef std::ptrdiff_t                          difference_type;

  // member classes/structs/unions
  template<typename T2>
  struct rebind {
    // types
    typedef private_adaptive_pool< T2, SegmentManager, NodesPerBlock, MaxFreeBlocks, OverheadPer↵
cent > other;
  };

  // construct/copy/destruct
  private_adaptive_pool(segment_manager *);
  private_adaptive_pool(const private_adaptive_pool &);
  template<typename T2>
    private_adaptive_pool(const private_adaptive_pool< T2, SegmentManager, NodesPerBlock, Max↵
FreeBlocks, OverheadPercent > &);
  template<typename T2, typename SegmentManager2, std::size_t N2,
           std::size_t F2, unsigned char OP2>
    private_adaptive_pool&
    operator=(const private_adaptive_pool< T2, SegmentManager2, N2, F2 > &);
  private_adaptive_pool& operator=(const private_adaptive_pool &);
  ~private_adaptive_pool();

  // public member functions
  node_pool_t * get_node_pool() const;
  segment_manager * get_segment_manager() const;
  size_type max_size() const;
  pointer allocate(size_type, cvoid_pointer = 0);
  void deallocate(const pointer &, size_type);
  void deallocate_free_blocks();
  pointer address(reference) const;
  const_pointer address(const_reference) const;
  void construct(const pointer &, const_reference);
  void destroy(const pointer &);
  size_type size(const pointer &) const;
  std::pair< pointer, bool >
  allocation_command(boost::interprocess::allocation_type, size_type,
                     size_type, size_type &, const pointer & = 0);
  multiallocation_chain allocate_many(size_type, std::size_t);
  multiallocation_chain allocate_many(const size_type *, size_type);
  void deallocate_many(multiallocation_chain);
```

```
  pointer allocate_one();
  multiallocation_chain allocate_individual(std::size_t);
  void deallocate_one(const pointer &);
  void deallocate_individual(multiallocation_chain);

  // friend functions
  friend void swap(self_t &, self_t &);
};
```

Description

An STL node allocator that uses a segment manager as memory source. The internal pointer type will of the same type (raw, smart) as "typename SegmentManager::void_pointer" type. This allows placing the allocator in shared memory, memory mapped-files, etc... This allocator has its own node pool.

NodesPerBlock is the minimum number of nodes of nodes allocated at once when the allocator needs runs out of nodes. MaxFreeBlocks is the maximum number of totally free blocks that the adaptive node pool will hold. The rest of the totally free blocks will be deallocated with the segment manager.

OverheadPercent is the (approximated) maximum size overhead (1-20%) of the allocator: (memory usable for nodes / total memory allocated from the segment manager)

`private_adaptive_pool` public construct/copy/destruct

1.
```
private_adaptive_pool(segment_manager * segment_mngr);
```

Constructor from a segment manager. If not present, constructs a node pool. Increments the reference count of the associated node pool. Can throw boost::interprocess::bad_alloc

2.
```
private_adaptive_pool(const private_adaptive_pool & other);
```

Copy constructor from other private_adaptive_pool. Increments the reference count of the associated node pool. Never throws

3.
```
template<typename T2>
  private_adaptive_pool(const private_adaptive_pool< T2, SegmentManager, NodesPerBlock, Max↵
FreeBlocks, OverheadPercent > & other);
```

Copy constructor from related private_adaptive_pool. If not present, constructs a node pool. Increments the reference count of the associated node pool. Can throw boost::interprocess::bad_alloc

4.
```
template<typename T2, typename SegmentManager2, std::size_t N2,
         std::size_t F2, unsigned char OP2>
  private_adaptive_pool&
  operator=(const private_adaptive_pool< T2, SegmentManager2, N2, F2 > &);
```

Not assignable from related private_adaptive_pool

5.
```
private_adaptive_pool& operator=(const private_adaptive_pool &);
```

Not assignable from other private_adaptive_pool

6.
```
~private_adaptive_pool();
```

Destructor, removes node_pool_t from memory if its reference count reaches to zero. Never throws

`private_adaptive_pool` public member functions

1.
```
node_pool_t * get_node_pool() const;
```

Returns a pointer to the node pool. Never throws

2.
```
segment_manager * get_segment_manager() const;
```

Returns the segment manager. Never throws

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

Returns the number of elements that could be allocated. Never throws

4.
```
pointer allocate(size_type count, cvoid_pointer hint = 0);
```

Allocate memory for an array of count elements. Throws boost::interprocess::bad_alloc if there is no enough memory

5.
```
void deallocate(const pointer & ptr, size_type count);
```

Deallocate allocated memory. Never throws

6.
```
void deallocate_free_blocks();
```

Deallocates all free blocks of the pool

7.
```
pointer address(reference value) const;
```

Returns address of mutable object. Never throws

8.
```
const_pointer address(const_reference value) const;
```

Returns address of non mutable object. Never throws

9.
```
void construct(const pointer & ptr, const_reference v);
```

Copy construct an object. Throws if T's copy constructor throws

10.
```
void destroy(const pointer & ptr);
```

Destroys object. Throws if object's destructor throws

11.
```
size_type size(const pointer & p) const;
```

Returns maximum the number of objects the previously allocated memory pointed by p can hold. This size only works for memory allocated with allocate, allocation_command and allocate_many.

12.
```
std::pair< pointer, bool >
allocation_command(boost::interprocess::allocation_type command,
                   size_type limit_size, size_type preferred_size,
                   size_type & received_size, const pointer & reuse = 0);
```

13.
```
multiallocation_chain
allocate_many(size_type elem_size, std::size_t num_elements);
```

Allocates many elements of size elem_size in a contiguous block of memory. The minimum number to be allocated is min_elements, the preferred and maximum number is preferred_elements. The number of actually allocated elements is will be assigned to received_size. The elements must be deallocated with deallocate(...)

14.
```
multiallocation_chain
allocate_many(const size_type * elem_sizes, size_type n_elements);
```

Allocates n_elements elements, each one of size elem_sizes[i]in a contiguous block of memory. The elements must be deallocated

15.
```
void deallocate_many(multiallocation_chain chain);
```

Allocates many elements of size elem_size in a contiguous block of memory. The minimum number to be allocated is min_elements, the preferred and maximum number is preferred_elements. The number of actually allocated elements is will be assigned to received_size. The elements must be deallocated with deallocate(...)

16.
```
pointer allocate_one();
```

Allocates just one object. Memory allocated with this function must be deallocated only with deallocate_one(). Throws boost::interprocess::bad_alloc if there is no enough memory

17.
```
multiallocation_chain allocate_individual(std::size_t num_elements);
```

Allocates many elements of size == 1 in a contiguous block of memory. The minimum number to be allocated is min_elements, the preferred and maximum number is preferred_elements. The number of actually allocated elements is will be assigned to received_size. Memory allocated with this function must be deallocated only with deallocate_one().

18.
```
void deallocate_one(const pointer & p);
```

Deallocates memory previously allocated with allocate_one(). You should never use deallocate_one to deallocate memory allocated with other functions different from allocate_one(). Never throws

19.
```
void deallocate_individual(multiallocation_chain chain);
```

Allocates many elements of size == 1 in a contiguous block of memory. The minimum number to be allocated is min_elements, the preferred and maximum number is preferred_elements. The number of actually allocated elements is will be assigned to received_size. Memory allocated with this function must be deallocated only with deallocate_one().

`private_adaptive_pool` friend functions

1.
```
friend void swap(self_t & alloc1, self_t & alloc2);
```

Swaps allocators. Does not throw. If each allocator is placed in a different memory segment, the result is undefined.

Struct template rebind

boost::interprocess::private_adaptive_pool::rebind

Synopsis

```
// In header: <boost/interprocess/allocators/private_adaptive_pool.hpp>


template<typename T2>
struct rebind {
  // types
  typedef private_adaptive_pool< T2, SegmentManager, NodesPerBlock, MaxFreeBlocks, OverheadPer↵
cent > other;
};
```

Description

Obtains private_adaptive_pool from private_adaptive_pool

Function template operator==

boost::interprocess::operator==

Synopsis

```
// In header: <boost/interprocess/allocators/private_adaptive_pool.hpp>


template<typename T, typename S, std::size_t NodesPerBlock, std::size_t F,
         unsigned char OP>
  bool operator==(const private_adaptive_pool< T, S, NodesPerBlock, F, OP > & alloc1,
                  const private_adaptive_pool< T, S, NodesPerBlock, F, OP > & alloc2);
```

Description

Equality test for same type of private_adaptive_pool

Function template operator!=

boost::interprocess::operator!=

Synopsis

```
// In header: <boost/interprocess/allocators/private_adaptive_pool.hpp>


template<typename T, typename S, std::size_t NodesPerBlock, std::size_t F,
         unsigned char OP>
  bool operator!=(const private_adaptive_pool< T, S, NodesPerBlock, F, OP > & alloc1,
                  const private_adaptive_pool< T, S, NodesPerBlock, F, OP > & alloc2);
```

Description

Inequality test for same type of private_adaptive_pool

Header <boost/interprocess/allocators/private_node_allocator.hpp>

Describes private_node_allocator_base pooled shared memory STL compatible allocator

```
namespace boost {
  namespace interprocess {
    template<typename T, typename SegmentManager, std::size_t NodesPerBlock>
      class private_node_allocator;
    template<typename T, typename S, std::size_t NodesPerBlock, std::size_t F,
             unsigned char OP>
      bool operator==(const private_node_allocator< T, S, NodesPerBlock, F, OP > &,
                      const private_node_allocator< T, S, NodesPerBlock, F, OP > &);
    template<typename T, typename S, std::size_t NodesPerBlock, std::size_t F,
             unsigned char OP>
      bool operator!=(const private_node_allocator< T, S, NodesPerBlock, F, OP > &,
                      const private_node_allocator< T, S, NodesPerBlock, F, OP > &);
  }
}
```

Class template private_node_allocator

boost::interprocess::private_node_allocator

Synopsis

```cpp
// In header: <boost/interprocess/allocators/private_node_allocator.hpp>

template<typename T, typename SegmentManager, std::size_t NodesPerBlock>
class private_node_allocator {
public:
  // types
  typedef implementation_defined::segment_manager  segment_manager;
  typedef segment_manager::void_pointer            void_pointer;
  typedef implementation_defined::pointer          pointer;
  typedef implementation_defined::const_pointer    const_pointer;
  typedef T                                        value_type;
  typedef unspecified                              reference;
  typedef unspecified                              const_reference;
  typedef std::size_t                              size_type;
  typedef std::ptrdiff_t                           difference_type;

  // member classes/structs/unions
  template<typename T2>
  struct rebind {
    // types
    typedef private_node_allocator< T2, SegmentManager, NodesPerBlock > other;
  };

  // construct/copy/destruct
  private_node_allocator(segment_manager *);
  private_node_allocator(const private_node_allocator &);
  template<typename T2>
    private_node_allocator(const private_node_allocator< T2, SegmentManager, NodesPerBlock > &);
  template<typename T2, typename SegmentManager2, std::size_t N2>
    private_node_allocator&
    operator=(const private_node_allocator< T2, SegmentManager2, N2 > &);
  private_node_allocator& operator=(const private_node_allocator &);
  ~private_node_allocator();

  // public member functions
  node_pool_t * get_node_pool() const;
  segment_manager * get_segment_manager() const;
  size_type max_size() const;
  pointer allocate(size_type, cvoid_pointer = 0);
  void deallocate(const pointer &, size_type);
  void deallocate_free_blocks();
  pointer address(reference) const;
  const_pointer address(const_reference) const;
  void construct(const pointer &, const_reference);
  void destroy(const pointer &);
  size_type size(const pointer &) const;
  std::pair< pointer, bool >
  allocation_command(boost::interprocess::allocation_type, size_type,
                     size_type, size_type &, const pointer & = 0);
  multiallocation_chain allocate_many(size_type, std::size_t);
  multiallocation_chain allocate_many(const size_type *, size_type);
  void deallocate_many(multiallocation_chain);
  pointer allocate_one();
  multiallocation_chain allocate_individual(std::size_t);
```

```
   void deallocate_one(const pointer &);
   void deallocate_individual(multiallocation_chain);

   // friend functions
   friend void swap(self_t &, self_t &);
};
```

Description

An STL node allocator that uses a segment manager as memory source. The internal pointer type will of the same type (raw, smart) as "typename SegmentManager::void_pointer" type. This allows placing the allocator in shared memory, memory mapped-files, etc... This allocator has its own node pool. NodesPerBlock is the number of nodes allocated at once when the allocator needs runs out of nodes

`private_node_allocator` public construct/copy/destruct

1.
```
private_node_allocator(segment_manager * segment_mngr);
```

Constructor from a segment manager. If not present, constructs a node pool. Increments the reference count of the associated node pool. Can throw boost::interprocess::bad_alloc

2.
```
private_node_allocator(const private_node_allocator & other);
```

Copy constructor from other private_node_allocator. Increments the reference count of the associated node pool. Never throws

3.
```
template<typename T2>
  private_node_allocator(const private_node_allocator< T2, SegmentManager, NodesPerBlock > &
 other);
```

Copy constructor from related private_node_allocator. If not present, constructs a node pool. Increments the reference count of the associated node pool. Can throw boost::interprocess::bad_alloc

4.
```
template<typename T2, typename SegmentManager2, std::size_t N2>
  private_node_allocator&
  operator=(const private_node_allocator< T2, SegmentManager2, N2 > &);
```

Not assignable from related private_node_allocator

5.
```
private_node_allocator& operator=(const private_node_allocator &);
```

Not assignable from other private_node_allocator

6.
```
~private_node_allocator();
```

Destructor, removes node_pool_t from memory if its reference count reaches to zero. Never throws

`private_node_allocator` public member functions

1.
```
node_pool_t * get_node_pool() const;
```

Returns a pointer to the node pool. Never throws

2.
```
segment_manager * get_segment_manager() const;
```

Returns the segment manager. Never throws

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

Returns the number of elements that could be allocated. Never throws

4.
```
pointer allocate(size_type count, cvoid_pointer hint = 0);
```

Allocate memory for an array of count elements. Throws boost::interprocess::bad_alloc if there is no enough memory

5.
```
void deallocate(const pointer & ptr, size_type count);
```

Deallocate allocated memory. Never throws

6.
```
void deallocate_free_blocks();
```

Deallocates all free blocks of the pool

7.
```
pointer address(reference value) const;
```

Returns address of mutable object. Never throws

8.
```
const_pointer address(const_reference value) const;
```

Returns address of non mutable object. Never throws

9.
```
void construct(const pointer & ptr, const_reference v);
```

Copy construct an object. Throws if T's copy constructor throws

10.
```
void destroy(const pointer & ptr);
```

Destroys object. Throws if object's destructor throws

11.
```
size_type size(const pointer & p) const;
```

Returns maximum the number of objects the previously allocated memory pointed by p can hold. This size only works for memory allocated with allocate, allocation_command and allocate_many.

12.
```
std::pair< pointer, bool >
allocation_command(boost::interprocess::allocation_type command,
                    size_type limit_size, size_type preferred_size,
                    size_type & received_size, const pointer & reuse = 0);
```

13.
```
multiallocation_chain
allocate_many(size_type elem_size, std::size_t num_elements);
```

Allocates many elements of size elem_size in a contiguous block of memory. The minimum number to be allocated is min_elements, the preferred and maximum number is preferred_elements. The number of actually allocated elements is will be assigned to received_size. The elements must be deallocated with deallocate(...)

14.
```
multiallocation_chain
allocate_many(const size_type * elem_sizes, size_type n_elements);
```

Allocates n_elements elements, each one of size elem_sizes[i]in a contiguous block of memory. The elements must be deallocated

15.
```
void deallocate_many(multiallocation_chain chain);
```

Allocates many elements of size elem_size in a contiguous block of memory. The minimum number to be allocated is min_elements, the preferred and maximum number is preferred_elements. The number of actually allocated elements is will be assigned to received_size. The elements must be deallocated with deallocate(...)

16.
```
pointer allocate_one();
```

Allocates just one object. Memory allocated with this function must be deallocated only with deallocate_one(). Throws boost::interprocess::bad_alloc if there is no enough memory

17.
```
multiallocation_chain allocate_individual(std::size_t num_elements);
```

Allocates many elements of size == 1 in a contiguous block of memory. The minimum number to be allocated is min_elements, the preferred and maximum number is preferred_elements. The number of actually allocated elements is will be assigned to received_size. Memory allocated with this function must be deallocated only with deallocate_one().

18.
```
void deallocate_one(const pointer & p);
```

Deallocates memory previously allocated with allocate_one(). You should never use deallocate_one to deallocate memory allocated with other functions different from allocate_one(). Never throws

19.
```
void deallocate_individual(multiallocation_chain chain);
```

Allocates many elements of size == 1 in a contiguous block of memory. The minimum number to be allocated is min_elements, the preferred and maximum number is preferred_elements. The number of actually allocated elements is will be assigned to received_size. Memory allocated with this function must be deallocated only with deallocate_one().

`private_node_allocator` **friend functions**

1.
```
friend void swap(self_t & alloc1, self_t & alloc2);
```

Swaps allocators. Does not throw. If each allocator is placed in a different memory segment, the result is undefined.

Struct template rebind

boost::interprocess::private_node_allocator::rebind

Synopsis

```
// In header: <boost/interprocess/allocators/private_node_allocator.hpp>


template<typename T2>
struct rebind {
  // types
  typedef private_node_allocator< T2, SegmentManager, NodesPerBlock > other;
};
```

Description

Obtains private_node_allocator from private_node_allocator

Function template operator==

boost::interprocess::operator==

Synopsis

```
// In header: <boost/interprocess/allocators/private_node_allocator.hpp>


template<typename T, typename S, std::size_t NodesPerBlock, std::size_t F,
         unsigned char OP>
  bool operator==(const private_node_allocator< T, S, NodesPerBlock, F, OP > & alloc1,
                  const private_node_allocator< T, S, NodesPerBlock, F, OP > & alloc2);
```

Description

Equality test for same type of private_node_allocator

Function template operator!=

boost::interprocess::operator!=

Synopsis

```
// In header: <boost/interprocess/allocators/private_node_allocator.hpp>


template<typename T, typename S, std::size_t NodesPerBlock, std::size_t F,
         unsigned char OP>
  bool operator!=(const private_node_allocator< T, S, NodesPerBlock, F, OP > & alloc1,
                  const private_node_allocator< T, S, NodesPerBlock, F, OP > & alloc2);
```

Description

Inequality test for same type of private_node_allocator

Header <boost/interprocess/anonymous_shared_memory.hpp>

Describes a function that creates anonymous shared memory that can be shared between forked processes

```
namespace boost {
  namespace interprocess {
    mapped_region anonymous_shared_memory(std::size_t, void * = 0);
  }
}
```

Function anonymous_shared_memory

boost::interprocess::anonymous_shared_memory

Synopsis

```
// In header: <boost/interprocess/anonymous_shared_memory.hpp>


mapped_region anonymous_shared_memory(std::size_t size, void * address = 0);
```

Description

A function that creates an anonymous shared memory segment of size "size". If "address" is passed the function will try to map the segment in that address. Otherwise the operating system will choose the mapping address. The function returns a mapped_region holding that segment or throws interprocess_exception if the function fails.

Header <boost/interprocess/containers/allocation_type.hpp>

Global allocate_new

boost::interprocess::allocate_new

Synopsis

```
// In header: <boost/interprocess/containers/allocation_type.hpp>

static const allocation_type allocate_new;
```

Global expand_fwd

boost::interprocess::expand_fwd

Synopsis

```
// In header: <boost/interprocess/containers/allocation_type.hpp>

static const allocation_type expand_fwd;
```

Global expand_bwd

boost::interprocess::expand_bwd

Synopsis

```
// In header: <boost/interprocess/containers/allocation_type.hpp>

static const allocation_type expand_bwd;
```

Global shrink_in_place

boost::interprocess::shrink_in_place

Synopsis

```
// In header: <boost/interprocess/containers/allocation_type.hpp>

static const allocation_type shrink_in_place;
```

Global try_shrink_in_place

boost::interprocess::try_shrink_in_place

Synopsis

```
// In header: <boost/interprocess/containers/allocation_type.hpp>

static const allocation_type try_shrink_in_place;
```

Global nothrow_allocation

boost::interprocess::nothrow_allocation

Synopsis

```
// In header: <boost/interprocess/containers/allocation_type.hpp>

static const allocation_type nothrow_allocation;
```

Global zero_memory

boost::interprocess::zero_memory

Synopsis

```
// In header: <boost/interprocess/containers/allocation_type.hpp>

static const allocation_type zero_memory;
```

Header <boost/interprocess/containers/deque.hpp>

Header <boost/interprocess/containers/flat_map.hpp>

Header <boost/interprocess/containers/flat_set.hpp>

Header <boost/interprocess/containers/list.hpp>

Header <boost/interprocess/containers/map.hpp>

Header <boost/interprocess/containers/pair.hpp>

Header <boost/interprocess/containers/set.hpp>

Header <boost/interprocess/containers/slist.hpp>

Header <boost/interprocess/containers/stable_vector.hpp>

Header <boost/interprocess/containers/string.hpp>

Header <boost/interprocess/containers/vector.hpp>

Header <boost/interprocess/containers/version_type.hpp>

Header <boost/interprocess/creation_tags.hpp>

```cpp
namespace boost {
  namespace interprocess {
    struct create_only_t;
    struct open_only_t;
    struct open_read_only_t;
    struct open_read_private_t;
    struct open_copy_on_write_t;
    struct open_or_create_t;

    static const create_only_t create_only;
    static const open_only_t open_only;
    static const open_read_only_t open_read_only;
    static const open_or_create_t open_or_create;
    static const open_copy_on_write_t open_copy_on_write;
  }
}
```

Struct create_only_t

boost::interprocess::create_only_t

Synopsis

```
// In header: <boost/interprocess/creation_tags.hpp>


struct create_only_t {
};
```

Description

Tag to indicate that the resource must be only created

Struct open_only_t

boost::interprocess::open_only_t

Synopsis

```
// In header: <boost/interprocess/creation_tags.hpp>


struct open_only_t {
};
```

Description

Tag to indicate that the resource must be only opened

Struct open_read_only_t

boost::interprocess::open_read_only_t

Synopsis

```
// In header: <boost/interprocess/creation_tags.hpp>


struct open_read_only_t {
};
```

Description

Tag to indicate that the resource must be only opened for reading

Struct open_read_private_t

boost::interprocess::open_read_private_t

Synopsis

```
// In header: <boost/interprocess/creation_tags.hpp>


struct open_read_private_t {
};
```

Description

Tag to indicate that the resource must be only opened privately for reading

Struct open_copy_on_write_t

boost::interprocess::open_copy_on_write_t

Synopsis

```
// In header: <boost/interprocess/creation_tags.hpp>


struct open_copy_on_write_t {
};
```

Description

Tag to indicate that the resource must be only opened for reading

Struct open_or_create_t

boost::interprocess::open_or_create_t

Synopsis

```
// In header: <boost/interprocess/creation_tags.hpp>


struct open_or_create_t {
};
```

Description

Tag to indicate that the resource must be created. If already created, it must be opened.

Global create_only

boost::interprocess::create_only

Synopsis

```
// In header: <boost/interprocess/creation_tags.hpp>

static const create_only_t create_only;
```

Description

Value to indicate that the resource must be only created

Global open_only

boost::interprocess::open_only

Synopsis

```
// In header: <boost/interprocess/creation_tags.hpp>

static const open_only_t open_only;
```

Description

Value to indicate that the resource must be only opened

Global open_read_only

boost::interprocess::open_read_only

Synopsis

```
// In header: <boost/interprocess/creation_tags.hpp>

static const open_read_only_t open_read_only;
```

Description

Value to indicate that the resource must be only opened for reading

Global open_or_create

boost::interprocess::open_or_create

Synopsis

```
// In header: <boost/interprocess/creation_tags.hpp>

static const open_or_create_t open_or_create;
```

Description

Value to indicate that the resource must be created. If already created, it must be opened.

Global open_copy_on_write

boost::interprocess::open_copy_on_write

Synopsis

```
// In header: <boost/interprocess/creation_tags.hpp>

static const open_copy_on_write_t open_copy_on_write;
```

Description

Value to indicate that the resource must be only opened for reading

Header <boost/interprocess/errors.hpp>

Describes the error numbering of interprocess classes

```
namespace boost {
  namespace interprocess {

    enum error_code_t { no_error =  0, system_error, other_error,
                        security_error, read_only_error, io_error, path_error,
                        not_found_error, busy_error, already_exists_error,
                        not_empty_error, is_directory_error,
                        out_of_space_error, out_of_memory_error,
                        out_of_resource_error, lock_error, sem_error,
                        mode_error, size_error, corrupted_error,
                        not_such_file_or_directory, invalid_argument };

    typedef int native_error_t;
  }
}
```

Header <boost/interprocess/exceptions.hpp>

Describes exceptions thrown by interprocess classes

```
namespace boost {
  namespace interprocess {
    class interprocess_exception;
    class lock_exception;
    class bad_alloc;
  }
}
```

Class interprocess_exception

boost::interprocess::interprocess_exception

Synopsis

```
// In header: <boost/interprocess/exceptions.hpp>


class interprocess_exception {
public:
  // construct/copy/destruct
  interprocess_exception(const char *);
  interprocess_exception(const error_info &, const char * = 0);
  ~interprocess_exception();

  // public member functions
  const char * what() const;
  native_error_t get_native_error() const;
  error_code_t get_error_code() const;
};
```

Description

This class is the base class of all exceptions thrown by boost::interprocess

`interprocess_exception` public construct/copy/destruct

1.
   ```
   interprocess_exception(const char * err);
   ```

2.
   ```
   interprocess_exception(const error_info & err_info, const char * str = 0);
   ```

3.
   ```
   ~interprocess_exception();
   ```

`interprocess_exception` public member functions

1.
   ```
   const char * what() const;
   ```

2.
   ```
   native_error_t get_native_error() const;
   ```

3.
   ```
   error_code_t get_error_code() const;
   ```

Class lock_exception

boost::interprocess::lock_exception

Synopsis

```
// In header: <boost/interprocess/exceptions.hpp>


class lock_exception : public boost::interprocess::interprocess_exception {
public:
  // construct/copy/destruct
  lock_exception();

  // public member functions
  const char * what() const;
};
```

Description

This is the exception thrown by shared interprocess_mutex family when a deadlock situation is detected or when using a interprocess_condition the interprocess_mutex is not locked

lock_exception public construct/copy/destruct

1.
```
lock_exception();
```

lock_exception public member functions

1.
```
const char * what() const;
```

Class bad_alloc

boost::interprocess::bad_alloc

Synopsis

```
// In header: <boost/interprocess/exceptions.hpp>


class bad_alloc : public boost::interprocess::interprocess_exception {
public:
  // construct/copy/destruct
  bad_alloc();

  // public member functions
  const char * what() const;
};
```

Description

This is the exception thrown by named interprocess_semaphore when a deadlock situation is detected or when an error is detected in the post/wait operation This is the exception thrown by synchronization objects when there is an error in a wait() function This exception is thrown when a named object is created in "open_only" mode and the resource was not already created This exception is thrown when a memory request can't be fulfilled.

`bad_alloc` public construct/copy/destruct

1.
```
bad_alloc();
```

`bad_alloc` public member functions

1.
```
const char * what() const;
```

Header <boost/interprocess/file_mapping.hpp>

Describes file_mapping and mapped region classes

```
namespace boost {
  namespace interprocess {
    class file_mapping;
    class remove_file_on_destroy;
  }
}
```

Class file_mapping

boost::interprocess::file_mapping

Synopsis

```
// In header: <boost/interprocess/file_mapping.hpp>


class file_mapping {
public:
  // construct/copy/destruct
  file_mapping();
  file_mapping(const char *, mode_t);
  file_mapping(file_mapping &&);
  file_mapping& operator=(file_mapping &&);
  ~file_mapping();

  // public member functions
  void swap(file_mapping &);
  mode_t get_mode() const;
  mapping_handle_t get_mapping_handle() const;
  const char * get_name() const;

  // public static functions
  static bool remove(const char *);
};
```

Description

A class that wraps a file-mapping that can be used to create mapped regions from the mapped files

`file_mapping` public construct/copy/destruct

1.
```
file_mapping();
```

Constructs an empty file mapping. Does not throw

2.
```
file_mapping(const char * filename, mode_t mode);
```

Opens a file mapping of file "filename", starting in offset "file_offset", and the mapping's size will be "size". The mapping can be opened for read-only "read_only" or read-write "read_write" modes. Throws interprocess_exception on error.

3.
```
file_mapping(file_mapping && moved);
```

Moves the ownership of "moved"'s file mapping object to *this. After the call, "moved" does not represent any file mapping object. Does not throw

4.
```
file_mapping& operator=(file_mapping && moved);
```

Moves the ownership of "moved"'s file mapping to *this. After the call, "moved" does not represent any file mapping. Does not throw

5.
```
~file_mapping();
```

Destroys the file mapping. All mapped regions created from this are still valid. Does not throw

`file_mapping` public member functions

1.
```
void swap(file_mapping & other);
```

Swaps to file_mappings. Does not throw.

2.
```
mode_t get_mode() const;
```

Returns access mode used in the constructor

3.
```
mapping_handle_t get_mapping_handle() const;
```

Obtains the mapping handle to be used with mapped_region

4.
```
const char * get_name() const;
```

Returns the name of the file used in the constructor.

`file_mapping` public static functions

1.
```
static bool remove(const char * filename);
```

Removes the file named "filename" even if it's been memory mapped. Returns true on success. The function might fail in some operating systems if the file is being used other processes and no deletion permission was shared.

Class remove_file_on_destroy

boost::interprocess::remove_file_on_destroy

Synopsis

```
// In header: <boost/interprocess/file_mapping.hpp>


class remove_file_on_destroy {
public:
  // construct/copy/destruct
  remove_file_on_destroy(const char *);
  ~remove_file_on_destroy();
};
```

Description

A class that stores the name of a file and tries to remove it in its destructor Useful to remove temporary files in the presence of exceptions

remove_file_on_destroy public construct/copy/destruct

1.
```
remove_file_on_destroy(const char * name);
```

2.
```
~remove_file_on_destroy();
```

Header <boost/interprocess/indexes/flat_map_index.hpp>

Describes index adaptor of boost::map container, to use it as name/shared memory index

```
namespace boost {
  namespace interprocess {
    template<typename MapConfig> struct flat_map_index_aux;

    template<typename MapConfig> class flat_map_index;
  }
}
```

Struct template flat_map_index_aux

boost::interprocess::flat_map_index_aux — Helper class to define typedefs from IndexTraits.

Synopsis

```
// In header: <boost/interprocess/indexes/flat_map_index.hpp>

template<typename MapConfig>
struct flat_map_index_aux {
  // types
  typedef MapConfig::key_type                              key_type;
  typedef MapConfig::mapped_type                           mapped_type;
  typedef MapConfig::segment_manager_base                  segment_manager_base;
  typedef std::less< key_type >                            key_less;
  typedef std::pair< key_type, mapped_type >              value_type;
  typedef allocator< value_type,segment_manager_base >    allocator_type;
  typedef flat_map< key_type, mapped_type, key_less, allocator_type > index_t;
};
```

Class template flat_map_index

boost::interprocess::flat_map_index

Synopsis

```
// In header: <boost/interprocess/indexes/flat_map_index.hpp>

template<typename MapConfig>
class flat_map_index : public boost::container::flat_map<MapConfig> {
public:
  // construct/copy/destruct
  flat_map_index(segment_manager_base *);

  // public member functions
  void reserve(std::size_t);
  void shrink_to_fit();
};
```

Description

Index type based in flat_map. Just derives from flat_map and defines the interface needed by managed memory segments.

`flat_map_index` public construct/copy/destruct

1.
```
flat_map_index(segment_manager_base * segment_mngr);
```

   Constructor. Takes a pointer to the segment manager. Can throw

`flat_map_index` public member functions

1.
```
void reserve(std::size_t n);
```

   This reserves memory to optimize the insertion of n elements in the index.

2.
```
void shrink_to_fit();
```

   This frees all unnecessary memory.

Header <boost/interprocess/indexes/iset_index.hpp>

Describes index adaptor of boost::intrusive::set container, to use it as name/shared memory index

```
namespace boost {
  namespace interprocess {
    template<typename MapConfig> class iset_index;
  }
}
```

Class template iset_index

boost::interprocess::iset_index

Synopsis

```cpp
// In header: <boost/interprocess/indexes/iset_index.hpp>

template<typename MapConfig>
class iset_index {
public:
  // types
  typedef index_type::iterator           iterator;
  typedef index_type::const_iterator     const_iterator;
  typedef index_type::insert_commit_data insert_commit_data;
  typedef index_type::value_type         value_type;

  // construct/copy/destruct
  iset_index(typename MapConfig::segment_manager_base *);

  // public member functions
  void reserve(std::size_t);
  void shrink_to_fit();
  iterator find(const intrusive_compare_key_type &);
  const_iterator find(const intrusive_compare_key_type &) const;
  std::pair< iterator, bool >
  insert_check(const intrusive_compare_key_type &, insert_commit_data &);
};
```

Description

Index type based in boost::intrusive::set. Just derives from boost::intrusive::set and defines the interface needed by managed memory segments

`iset_index` public construct/copy/destruct

1.
```cpp
iset_index(typename MapConfig::segment_manager_base *);
```

Constructor. Takes a pointer to the segment manager. Can throw

`iset_index` public member functions

1.
```cpp
void reserve(std::size_t);
```

This reserves memory to optimize the insertion of n elements in the index

2.
```cpp
void shrink_to_fit();
```

This frees all unnecessary memory.

3.
```cpp
iterator find(const intrusive_compare_key_type & key);
```

4.
```cpp
const_iterator find(const intrusive_compare_key_type & key) const;
```

5.
```
std::pair< iterator, bool >
insert_check(const intrusive_compare_key_type & key,
             insert_commit_data & commit_data);
```

Header <boost/interprocess/indexes/iunordered_set_index.hpp>

Describes index adaptor of boost::intrusive::unordered_set container, to use it as name/shared memory index

```
namespace boost {
  namespace interprocess {
    template<typename MapConfig> class iunordered_set_index;
  }
}
```

Class template iunordered_set_index

boost::interprocess::iunordered_set_index

Synopsis

```
// In header: <boost/interprocess/indexes/iunordered_set_index.hpp>

template<typename MapConfig>
class iunordered_set_index {
public:
  // types
  typedef index_type::iterator          iterator;
  typedef index_type::const_iterator    const_iterator;
  typedef index_type::insert_commit_data insert_commit_data;
  typedef index_type::value_type        value_type;
  typedef index_type::bucket_ptr        bucket_ptr;
  typedef index_type::bucket_type       bucket_type;
  typedef index_type::bucket_traits     bucket_traits;
  typedef index_type::size_type         size_type;

  // construct/copy/destruct
  iunordered_set_index(segment_manager_base *);
  ~iunordered_set_index();

  // public member functions
  void reserve(std::size_t);
  void shrink_to_fit();
  iterator find(const intrusive_compare_key_type &);
  const_iterator find(const intrusive_compare_key_type &) const;
  std::pair< iterator, bool >
  insert_check(const intrusive_compare_key_type &, insert_commit_data &);
  iterator insert_commit(value_type &, insert_commit_data &);
};
```

Description

Index type based in boost::intrusive::set. Just derives from boost::intrusive::set and defines the interface needed by managed memory segments

iunordered_set_index public construct/copy/destruct

1.
```
iunordered_set_index(segment_manager_base * mngr);
```

   Constructor. Takes a pointer to the segment manager. Can throw

2.
```
~iunordered_set_index();
```

iunordered_set_index public member functions

1.
```
void reserve(std::size_t new_n);
```

   This reserves memory to optimize the insertion of n elements in the index

2.
```
void shrink_to_fit();
```

This tries to free unused memory previously allocated.

3.
```
iterator find(const intrusive_compare_key_type & key);
```

4.
```
const_iterator find(const intrusive_compare_key_type & key) const;
```

5.
```
std::pair< iterator, bool >
insert_check(const intrusive_compare_key_type & key,
             insert_commit_data & commit_data);
```

6.
```
iterator insert_commit(value_type & val, insert_commit_data & commit_data);
```

Header <boost/interprocess/indexes/map_index.hpp>

Describes index adaptor of boost::map container, to use it as name/shared memory index

```
namespace boost {
  namespace interprocess {
    template<typename MapConfig> class map_index;
  }
}
```

Class template map_index

boost::interprocess::map_index

Synopsis

```
// In header: <boost/interprocess/indexes/map_index.hpp>

template<typename MapConfig>
class map_index : public boost::container::map<MapConfig> {
public:
  // construct/copy/destruct
  map_index(segment_manager_base *);

  // public member functions
  void reserve(std::size_t);
  void shrink_to_fit();
};
```

Description

Index type based in boost::interprocess::map. Just derives from boost::interprocess::map and defines the interface needed by managed memory segments

map_index public construct/copy/destruct

1.
```
map_index(segment_manager_base * segment_mngr);
```

Constructor. Takes a pointer to the segment manager. Can throw

map_index public member functions

1.
```
void reserve(std::size_t);
```

This reserves memory to optimize the insertion of n elements in the index

2.
```
void shrink_to_fit();
```

This tries to free previously allocate unused memory.

Header <boost/interprocess/indexes/null_index.hpp>

Describes a null index adaptor, so that if we don't want to construct named objects, we can use this null index type to save resources.

```
namespace boost {
  namespace interprocess {
    template<typename MapConfig> class null_index;
  }
}
```

Class template null_index

boost::interprocess::null_index

Synopsis

```
// In header: <boost/interprocess/indexes/null_index.hpp>

template<typename MapConfig>
class null_index {
public:
  // types
  typedef void *        iterator;
  typedef const void * const_iterator;

  // construct/copy/destruct
  null_index(segment_manager_base *);

  // public member functions
  const_iterator begin() const;
  iterator begin();
  const_iterator end() const;
  iterator end();
};
```

Description

Null index type used to save compilation time when named indexes are not needed.

`null_index` public construct/copy/destruct

1.
```
null_index(segment_manager_base *);
```

Empty constructor.

`null_index` public member functions

1.
```
const_iterator begin() const;
```

begin() is equal to end()

2.
```
iterator begin();
```

begin() is equal to end()

3.
```
const_iterator end() const;
```

begin() is equal to end()

4.
```
iterator end();
```

begin() is equal to end()

Header <boost/interprocess/indexes/unordered_map_index.hpp>

Describes index adaptor of boost::unordered_map container, to use it as name/shared memory index

```
namespace boost {
  namespace interprocess {
    template<typename MapConfig> class unordered_map_index;
  }
}
```

Class template unordered_map_index

boost::interprocess::unordered_map_index

Synopsis

```
// In header: <boost/interprocess/indexes/unordered_map_index.hpp>

template<typename MapConfig>
class unordered_map_index {
public:
  // construct/copy/destruct
  unordered_map_index(segment_manager_base *);

  // public member functions
  void reserve(std::size_t);
  void shrink_to_fit();
};
```

Description

Index type based in unordered_map. Just derives from unordered_map and defines the interface needed by managed memory segments

unordered_map_index public construct/copy/destruct

1.
```
unordered_map_index(segment_manager_base * segment_mngr);
```

Constructor. Takes a pointer to the segment manager. Can throw

unordered_map_index public member functions

1.
```
void reserve(std::size_t n);
```

This reserves memory to optimize the insertion of n elements in the index

2.
```
void shrink_to_fit();
```

This tries to free previously allocate unused memory.

Header <boost/interprocess/interprocess_fwd.hpp>

```
namespace boost {
  namespace interprocess {
    typedef basic_managed_external_buffer< char,rbtree_best_fit< null_mutex_family >,iset_in↵
dex > managed_external_buffer;
    typedef basic_managed_external_buffer< wchar_t,rbtree_best_fit< null_mutex_family >,iset_in↵
dex > wmanaged_external_buffer;
    typedef basic_managed_shared_memory< char,rbtree_best_fit< mutex_family >,iset_index > man↵
aged_shared_memory;
    typedef basic_managed_shared_memory< wchar_t,rbtree_best_fit< mutex_family >,iset_index > wman↵
aged_shared_memory;
    typedef basic_managed_windows_shared_memory< char,rbtree_best_fit< mutex_family >,iset_in↵
dex > managed_windows_shared_memory;
    typedef basic_managed_windows_shared_memory< wchar_t,rbtree_best_fit< mutex_family >,iset_in↵
dex > wmanaged_windows_shared_memory;
    typedef basic_managed_xsi_shared_memory< char,rbtree_best_fit< mutex_family >,iset_index > man↵
aged_xsi_shared_memory;
    typedef basic_managed_xsi_shared_memory< wchar_t,rbtree_best_fit< mutex_family >,iset_in↵
dex > wmanaged_xsi_shared_memory;
    typedef basic_managed_shared_memory< char,rbtree_best_fit< mutex_family, void * >,iset_in↵
dex > fixed_managed_shared_memory;
    typedef basic_managed_shared_memory< wchar_t,rbtree_best_fit< mutex_family, void * >,iset_in↵
dex > wfixed_managed_shared_memory;
    typedef basic_managed_heap_memory< char,rbtree_best_fit< null_mutex_family >,iset_index > man↵
aged_heap_memory;
    typedef basic_managed_heap_memory< wchar_t,rbtree_best_fit< null_mutex_family >,iset_in↵
dex > wmanaged_heap_memory;
    typedef basic_managed_mapped_file< char,rbtree_best_fit< mutex_family >,iset_index > man↵
aged_mapped_file;
    typedef basic_managed_mapped_file< wchar_t,rbtree_best_fit< mutex_family >,iset_index > wman↵
aged_mapped_file;
  }
}
```

Header <boost/interprocess/ipc/message_queue.hpp>

Describes an inter-process message queue. This class allows sending messages between processes and allows blocking, non-blocking and timed sending and receiving.

```
namespace boost {
  namespace interprocess {
    class message_queue;
  }
}
```

Class message_queue

boost::interprocess::message_queue

Synopsis

```cpp
// In header: <boost/interprocess/ipc/message_queue.hpp>


class message_queue {
public:
  // construct/copy/destruct
  message_queue(create_only_t, const char *, std::size_t, std::size_t,
                const permissions & = permissions());
  message_queue(open_or_create_t, const char *, std::size_t, std::size_t,
                const permissions & = permissions());
  message_queue(open_only_t, const char *);
  ~message_queue();

  // public member functions
  *void send(const void *, std::size_t, unsigned int);
  bool try_send(const void *, std::size_t, unsigned int);
  bool timed_send(const void *, std::size_t, unsigned int,
                  const boost::posix_time::ptime &);
  void receive(void *, std::size_t, std::size_t &, unsigned int &);
  bool try_receive(void *, std::size_t, std::size_t &, unsigned int &);
  bool timed_receive(void *, std::size_t, std::size_t &, unsigned int &,
                     const boost::posix_time::ptime &);
  std::size_t get_max_msg() const;
  std::size_t get_max_msg_size() const;
  std::size_t get_num_msg();

  // public static functions
  static bool remove(const char *);
};
```

Description

A class that allows sending messages between processes.

message_queue public construct/copy/destruct

1.
```cpp
message_queue(create_only_t create_only, const char * name,
              std::size_t max_num_msg, std::size_t max_msg_size,
              const permissions & perm = permissions());
```

Creates a process shared message queue with name "name". For this message queue, the maximum number of messages will be "max_num_msg" and the maximum message size will be "max_msg_size". Throws on error and if the queue was previously created.

2.
```cpp
message_queue(open_or_create_t open_or_create, const char * name,
              std::size_t max_num_msg, std::size_t max_msg_size,
              const permissions & perm = permissions());
```

Opens or creates a process shared message queue with name "name". If the queue is created, the maximum number of messages will be "max_num_msg" and the maximum message size will be "max_msg_size". If queue was previously created the queue will be opened and "max_num_msg" and "max_msg_size" parameters are ignored. Throws on error.

3.
```
message_queue(open_only_t open_only, const char * name);
```

Opens a previously created process shared message queue with name "name". If the was not previously created or there are no free resources, throws an error.

4.
```
~message_queue();
```

Destroys *this and indicates that the calling process is finished using the resource. All opened message queues are still valid after destruction. The destructor function will deallocate any system resources allocated by the system for use by this process for this resource. The resource can still be opened again calling the open constructor overload. To erase the message queue from the system use remove().

`message_queue` public member functions

1.
```
*void send(const void * buffer, std::size_t buffer_size,
          unsigned int priority);
```

Sends a message stored in buffer "buffer" with size "buffer_size" in the message queue with priority "priority". If the message queue is full the sender is blocked. Throws interprocess_error on error.

2.
```
bool try_send(const void * buffer, std::size_t buffer_size,
              unsigned int priority);
```

Sends a message stored in buffer "buffer" with size "buffer_size" through the message queue with priority "priority". If the message queue is full the sender is not blocked and returns false, otherwise returns true. Throws interprocess_error on error.

3.
```
bool timed_send(const void * buffer, std::size_t buffer_size,
                unsigned int priority,
                const boost::posix_time::ptime & abs_time);
```

Sends a message stored in buffer "buffer" with size "buffer_size" in the message queue with priority "priority". If the message queue is full the sender retries until time "abs_time" is reached. Returns true if the message has been successfully sent. Returns false if timeout is reached. Throws interprocess_error on error.

4.
```
void receive(void * buffer, std::size_t buffer_size, std::size_t & recvd_size,
             unsigned int & priority);
```

Receives a message from the message queue. The message is stored in buffer "buffer", which has size "buffer_size". The received message has size "recvd_size" and priority "priority". If the message queue is empty the receiver is blocked. Throws interprocess_error on error.

5.
```
bool try_receive(void * buffer, std::size_t buffer_size,
                 std::size_t & recvd_size, unsigned int & priority);
```

Receives a message from the message queue. The message is stored in buffer "buffer", which has size "buffer_size". The received message has size "recvd_size" and priority "priority". If the message queue is empty the receiver is not blocked and returns false, otherwise returns true. Throws interprocess_error on error.

6.
```
bool timed_receive(void * buffer, std::size_t buffer_size,
                   std::size_t & recvd_size, unsigned int & priority,
                   const boost::posix_time::ptime & abs_time);
```

Receives a message from the message queue. The message is stored in buffer "buffer", which has size "buffer_size". The received message has size "recvd_size" and priority "priority". If the message queue is empty the receiver retries until time "abs_time" is

reached. Returns true if the message has been successfully sent. Returns false if timeout is reached. Throws interprocess_error on error.

7.
```
std::size_t get_max_msg() const;
```

Returns the maximum number of messages allowed by the queue. The message queue must be opened or created previously. Otherwise, returns 0. Never throws

8.
```
std::size_t get_max_msg_size() const;
```

Returns the maximum size of message allowed by the queue. The message queue must be opened or created previously. Otherwise, returns 0. Never throws

9.
```
std::size_t get_num_msg();
```

Returns the number of messages currently stored. Never throws

`message_queue` public static functions

1.
```
static bool remove(const char * name);
```

Removes the message queue from the system. Returns false on error. Never throws

Header <boost/interprocess/managed_external_buffer.hpp>

Describes a named user memory allocation user class.

```
namespace boost {
  namespace interprocess {
    template<typename CharType, typename AllocationAlgorithm,
            template< class IndexConfig > class IndexType>
      class basic_managed_external_buffer;
  }
}
```

Class template basic_managed_external_buffer

boost::interprocess::basic_managed_external_buffer

Synopsis

```cpp
// In header: <boost/interprocess/managed_external_buffer.hpp>

template<typename CharType, typename AllocationAlgorithm,
         template< class IndexConfig > class IndexType>
class basic_managed_external_buffer {
public:
  // construct/copy/destruct
  basic_managed_external_buffer();
  basic_managed_external_buffer(create_only_t, void *, std::size_t);
  basic_managed_external_buffer(open_only_t, void *, std::size_t);
  basic_managed_external_buffer(basic_managed_external_buffer &&);
  basic_managed_external_buffer& operator=(basic_managed_external_buffer &&);

  // public member functions
  void grow(std::size_t);
  void swap(basic_managed_external_buffer &);
};
```

Description

A basic user memory named object creation class. Inherits all basic functionality from basic_managed_memory_impl<CharType, AllocationAlgorithm, IndexType>

`basic_managed_external_buffer` public construct/copy/destruct

1.
```cpp
basic_managed_external_buffer();
```

Default constructor. Does nothing. Useful in combination with move semantics

2.
```cpp
basic_managed_external_buffer(create_only_t, void * addr, std::size_t size);
```

Creates and places the segment manager. This can throw.

3.
```cpp
basic_managed_external_buffer(open_only_t, void * addr, std::size_t size);
```

Creates and places the segment manager. This can throw.

4.
```cpp
basic_managed_external_buffer(basic_managed_external_buffer && moved);
```

Moves the ownership of "moved"'s managed memory to *this. Does not throw.

5.
```cpp
basic_managed_external_buffer&
operator=(basic_managed_external_buffer && moved);
```

Moves the ownership of "moved"'s managed memory to *this. Does not throw.

`basic_managed_external_buffer` public member functions

1.
```cpp
void grow(std::size_t extra_bytes);
```

247

2.
```
void swap(basic_managed_external_buffer & other);
```

Swaps the ownership of the managed heap memories managed by *this and other. Never throws.

Header <boost/interprocess/managed_heap_memory.hpp>

Describes a named heap memory allocation user class.

```
namespace boost {
  namespace interprocess {
    template<typename CharType, typename AllocationAlgorithm,
             template< class IndexConfig > class IndexType>
      class basic_managed_heap_memory;
  }
}
```

Class template basic_managed_heap_memory

boost::interprocess::basic_managed_heap_memory

Synopsis

```
// In header: <boost/interprocess/managed_heap_memory.hpp>

template<typename CharType, typename AllocationAlgorithm,
         template< class IndexConfig > class IndexType>
class basic_managed_heap_memory {
public:
  // construct/copy/destruct
  basic_managed_heap_memory();
  basic_managed_heap_memory(std::size_t);
  basic_managed_heap_memory(basic_managed_heap_memory &&);
  basic_managed_heap_memory& operator=(basic_managed_heap_memory &&);
  ~basic_managed_heap_memory();

  // public member functions
  bool grow(std::size_t);
  void swap(basic_managed_heap_memory &);
};
```

Description

A basic heap memory named object creation class. Initializes the heap memory segment. Inherits all basic functionality from basic_managed_memory_impl<CharType, AllocationAlgorithm, IndexType>

`basic_managed_heap_memory` public construct/copy/destruct

1.
```
basic_managed_heap_memory();
```

Default constructor. Does nothing. Useful in combination with move semantics

2.
```
basic_managed_heap_memory(std::size_t size);
```

Creates heap memory and initializes the segment manager. This can throw.

3.
```
basic_managed_heap_memory(basic_managed_heap_memory && moved);
```

Moves the ownership of "moved"'s managed memory to *this. Does not throw.

4.
```
basic_managed_heap_memory& operator=(basic_managed_heap_memory && moved);
```

Moves the ownership of "moved"'s managed memory to *this. Does not throw.

5.
```
~basic_managed_heap_memory();
```

Destructor. Liberates the heap memory holding the managed data. Never throws.

`basic_managed_heap_memory` public member functions

1.
```
bool grow(std::size_t extra_bytes);
```

Tries to resize internal heap memory so that we have room for more objects. WARNING: If memory is reallocated, all the objects will be binary-copied to the new buffer. To be able to use this function, all pointers constructed in this buffer must be offset pointers. Otherwise, the result is undefined. Returns true if the growth has been successful, so you will have some extra bytes to allocate new objects. If returns false, the heap allocation has failed.

2.
```
void swap(basic_managed_heap_memory & other);
```

Swaps the ownership of the managed heap memories managed by *this and other. Never throws.

Header <boost/interprocess/managed_mapped_file.hpp>

```
namespace boost {
  namespace interprocess {
    template<typename CharType, typename AllocationAlgorithm,
             template< class IndexConfig > class IndexType>
      class basic_managed_mapped_file;
  }
}
```

Class template basic_managed_mapped_file

boost::interprocess::basic_managed_mapped_file

Synopsis

```
// In header: <boost/interprocess/managed_mapped_file.hpp>

template<typename CharType, typename AllocationAlgorithm,
         template< class IndexConfig > class IndexType>
class basic_managed_mapped_file {
public:
  // construct/copy/destruct
  basic_managed_mapped_file();
  basic_managed_mapped_file(create_only_t, const char *, std::size_t,
                            const void * = 0,
                            const permissions & = permissions());
  basic_managed_mapped_file(open_or_create_t, const char *, std::size_t,
                            const void * = 0,
                            const permissions & = permissions());
  basic_managed_mapped_file(open_only_t, const char *, const void * = 0);
  basic_managed_mapped_file(open_copy_on_write_t, const char *,
                            const void * = 0);
  basic_managed_mapped_file(open_read_only_t, const char *, const void * = 0);
  basic_managed_mapped_file(basic_managed_mapped_file &&);
  basic_managed_mapped_file& operator=(basic_managed_mapped_file &&);
  ~basic_managed_mapped_file();

  // public member functions
  void swap(basic_managed_mapped_file &);
  bool flush();

  // public static functions
  static bool grow(const char *, std::size_t);
  static bool shrink_to_fit(const char *);
};
```

Description

A basic mapped file named object creation class. Initializes the mapped file. Inherits all basic functionality from basic_managed_memory_impl<CharType, AllocationAlgorithm, IndexType>

`basic_managed_mapped_file` public construct/copy/destruct

1.
```
basic_managed_mapped_file();
```

Creates mapped file and creates and places the segment manager. This can throw.

2.
```
basic_managed_mapped_file(create_only_t create_only, const char * name,
                          std::size_t size, const void * addr = 0,
                          const permissions & perm = permissions());
```

Creates mapped file and creates and places the segment manager. This can throw.

3.
```
basic_managed_mapped_file(open_or_create_t open_or_create, const char * name,
                          std::size_t size, const void * addr = 0,
                          const permissions & perm = permissions());
```

Creates mapped file and creates and places the segment manager if segment was not created. If segment was created it connects to the segment. This can throw.

4.
```
basic_managed_mapped_file(open_only_t open_only, const char * name,
                          const void * addr = 0);
```

Connects to a created mapped file and its segment manager. This can throw.

5.
```
basic_managed_mapped_file(open_copy_on_write_t, const char * name,
                          const void * addr = 0);
```

Connects to a created mapped file and its segment manager in copy_on_write mode. This can throw.

6.
```
basic_managed_mapped_file(open_read_only_t, const char * name,
                          const void * addr = 0);
```

Connects to a created mapped file and its segment manager in read-only mode. This can throw.

7.
```
basic_managed_mapped_file(basic_managed_mapped_file && moved);
```

Moves the ownership of "moved"'s managed memory to *this. Does not throw

8.
```
basic_managed_mapped_file& operator=(basic_managed_mapped_file && moved);
```

Moves the ownership of "moved"'s managed memory to *this. Does not throw

9.
```
~basic_managed_mapped_file();
```

Destroys *this and indicates that the calling process is finished using the resource. The destructor function will deallocate any system resources allocated by the system for use by this process for this resource. The resource can still be opened again calling the open constructor overload. To erase the resource from the system use remove().

`basic_managed_mapped_file` public member functions

1.
```
void swap(basic_managed_mapped_file & other);
```

Swaps the ownership of the managed mapped memories managed by *this and other. Never throws.

2.
```
bool flush();
```

Flushes cached data to file. Never throws

`basic_managed_mapped_file` public static functions

1.
```
static bool grow(const char * filename, std::size_t extra_bytes);
```

Tries to resize mapped file so that we have room for more objects.

This function is not synchronized so no other thread or process should be reading or writing the file

2.
```
static bool shrink_to_fit(const char * filename);
```

Tries to resize mapped file to minimized the size of the file.

This function is not synchronized so no other thread or process should be reading or writing the file

Header <boost/interprocess/managed_shared_memory.hpp>

```
namespace boost {
  namespace interprocess {
    template<typename CharType, typename AllocationAlgorithm,
            template< class IndexConfig > class IndexType>
      class basic_managed_shared_memory;
  }
}
```

Class template basic_managed_shared_memory

boost::interprocess::basic_managed_shared_memory

Synopsis

```
// In header: <boost/interprocess/managed_shared_memory.hpp>

template<typename CharType, typename AllocationAlgorithm,
         template< class IndexConfig > class IndexType>
class basic_managed_shared_memory {
public:
  // construct/copy/destruct
  basic_managed_shared_memory();
  basic_managed_shared_memory(create_only_t, const char *, std::size_t,
                              const void * = 0,
                              const permissions & = permissions());
  basic_managed_shared_memory(open_or_create_t, const char *, std::size_t,
                              const void * = 0,
                              const permissions & = permissions());
  basic_managed_shared_memory(open_copy_on_write_t, const char *,
                              const void * = 0);
  basic_managed_shared_memory(open_read_only_t, const char *,
                              const void * = 0);
  basic_managed_shared_memory(open_only_t, const char *, const void * = 0);
  basic_managed_shared_memory(basic_managed_shared_memory &&);
  basic_managed_shared_memory& operator=(basic_managed_shared_memory &&);
  ~basic_managed_shared_memory();

  // public member functions
  void swap(basic_managed_shared_memory &);
  bool flush();

  // public static functions
  static bool grow(const char *, std::size_t);
  static bool shrink_to_fit(const char *);
};
```

Description

A basic shared memory named object creation class. Initializes the shared memory segment. Inherits all basic functionality from basic_managed_memory_impl<CharType, AllocationAlgorithm, IndexType>

`basic_managed_shared_memory` public construct/copy/destruct

1.
```
basic_managed_shared_memory();
```

Default constructor. Does nothing. Useful in combination with move semantics

2.
```
basic_managed_shared_memory(create_only_t create_only, const char * name,
                            std::size_t size, const void * addr = 0,
                            const permissions & perm = permissions());
```

Creates shared memory and creates and places the segment manager. This can throw.

3.
```cpp
basic_managed_shared_memory(open_or_create_t open_or_create,
                           const char * name, std::size_t size,
                           const void * addr = 0,
                           const permissions & perm = permissions());
```

Creates shared memory and creates and places the segment manager if segment was not created. If segment was created it connects to the segment. This can throw.

4.
```cpp
basic_managed_shared_memory(open_copy_on_write_t, const char * name,
                           const void * addr = 0);
```

Connects to a created shared memory and its segment manager. in copy_on_write mode. This can throw.

5.
```cpp
basic_managed_shared_memory(open_read_only_t, const char * name,
                           const void * addr = 0);
```

Connects to a created shared memory and its segment manager. in read-only mode. This can throw.

6.
```cpp
basic_managed_shared_memory(open_only_t open_only, const char * name,
                           const void * addr = 0);
```

Connects to a created shared memory and its segment manager. This can throw.

7.
```cpp
basic_managed_shared_memory(basic_managed_shared_memory && moved);
```

Moves the ownership of "moved"'s managed memory to *this. Does not throw

8.
```cpp
basic_managed_shared_memory& operator=(basic_managed_shared_memory && moved);
```

Moves the ownership of "moved"'s managed memory to *this. Does not throw

9.
```cpp
~basic_managed_shared_memory();
```

Destroys *this and indicates that the calling process is finished using the resource. The destructor function will deallocate any system resources allocated by the system for use by this process for this resource. The resource can still be opened again calling the open constructor overload. To erase the resource from the system use remove().

`basic_managed_shared_memory` public member functions

1.
```cpp
void swap(basic_managed_shared_memory & other);
```

Swaps the ownership of the managed shared memories managed by *this and other. Never throws.

2.
```cpp
bool flush();
```

`basic_managed_shared_memory` public static functions

1.
```cpp
static bool grow(const char * shmname, std::size_t extra_bytes);
```

Tries to resize the managed shared memory object so that we have room for more objects.

This function is not synchronized so no other thread or process should be reading or writing the file

2.
```
static bool shrink_to_fit(const char * shmname);
```

Tries to resize the managed shared memory to minimized the size of the file.

This function is not synchronized so no other thread or process should be reading or writing the file

Header <boost/interprocess/managed_windows_shared_memory.hpp>

```
namespace boost {
  namespace interprocess {
    template<typename CharType, typename AllocationAlgorithm,
             template< class IndexConfig > class IndexType>
      class basic_managed_windows_shared_memory;
  }
}
```

Class template basic_managed_windows_shared_memory

boost::interprocess::basic_managed_windows_shared_memory

Synopsis

```
// In header: <boost/interprocess/managed_windows_shared_memory.hpp>

template<typename CharType, typename AllocationAlgorithm,
         template< class IndexConfig > class IndexType>
class basic_managed_windows_shared_memory {
public:
  // construct/copy/destruct
  basic_managed_windows_shared_memory();
  basic_managed_windows_shared_memory(create_only_t, const char *,
                                      std::size_t, const void * = 0,
                                      const permissions & = permissions());
  basic_managed_windows_shared_memory(open_or_create_t, const char *,
                                      std::size_t, const void * = 0,
                                      const permissions & = permissions());
  basic_managed_windows_shared_memory(open_only_t, const char *,
                                      const void * = 0);
  basic_managed_windows_shared_memory(open_copy_on_write_t, const char *,
                                      const void * = 0);
  basic_managed_windows_shared_memory(open_read_only_t, const char *,
                                      const void * = 0);
  basic_managed_windows_shared_memory(basic_managed_windows_shared_memory &&);
  basic_managed_windows_shared_memory&
  operator=(basic_managed_windows_shared_memory &&);
  ~basic_managed_windows_shared_memory();

  // public member functions
  void swap(basic_managed_windows_shared_memory &);
};
```

Description

A basic managed windows shared memory creation class. Initializes the shared memory segment. Inherits all basic functionality from basic_managed_memory_impl<CharType, AllocationAlgorithm, IndexType> Unlike basic_managed_shared_memory, it has no kernel persistence and the shared memory is destroyed when all processes destroy all their windows_shared_memory objects and mapped regions for the same shared memory or the processes end/crash.

Warning: basic_managed_windows_shared_memory and basic_managed_shared_memory can't communicate between them.

`basic_managed_windows_shared_memory` public construct/copy/destruct

1.
```
basic_managed_windows_shared_memory();
```

   Default constructor. Does nothing. Useful in combination with move semantics

2.
```
basic_managed_windows_shared_memory(create_only_t create_only,
                                    const char * name, std::size_t size,
                                    const void * addr = 0,
                                    const permissions & perm = permissions());
```

   Creates shared memory and creates and places the segment manager. This can throw.

3.
```cpp
basic_managed_windows_shared_memory(open_or_create_t open_or_create,
                                    const char * name, std::size_t size,
                                    const void * addr = 0,
                                    const permissions & perm = permissions());
```

Creates shared memory and creates and places the segment manager if segment was not created. If segment was created it connects to the segment. This can throw.

4.
```cpp
basic_managed_windows_shared_memory(open_only_t open_only, const char * name,
                                    const void * addr = 0);
```

Connects to a created shared memory and its segment manager. This can throw.

5.
```cpp
basic_managed_windows_shared_memory(open_copy_on_write_t, const char * name,
                                    const void * addr = 0);
```

Connects to a created shared memory and its segment manager in copy_on_write mode. This can throw.

6.
```cpp
basic_managed_windows_shared_memory(open_read_only_t, const char * name,
                                    const void * addr = 0);
```

Connects to a created shared memory and its segment manager in read-only mode. This can throw.

7.
```cpp
basic_managed_windows_shared_memory(basic_managed_windows_shared_memory && moved);
```

Moves the ownership of "moved"'s managed memory to *this. Does not throw

8.
```cpp
basic_managed_windows_shared_memory&
operator=(basic_managed_windows_shared_memory && moved);
```

Moves the ownership of "moved"'s managed memory to *this. Does not throw

9.
```cpp
~basic_managed_windows_shared_memory();
```

Destroys *this and indicates that the calling process is finished using the resource. All mapped regions are still valid after destruction. When all mapped regions and basic_managed_windows_shared_memory objects referring the shared memory are destroyed, the operating system will destroy the shared memory.

`basic_managed_windows_shared_memory` public member functions

1.
```cpp
void swap(basic_managed_windows_shared_memory & other);
```

Swaps the ownership of the managed mapped memories managed by *this and other. Never throws.

Header <boost/interprocess/managed_xsi_shared_memory.hpp>

```cpp
namespace boost {
  namespace interprocess {
    template<typename CharType, typename AllocationAlgorithm,
             template< class IndexConfig > class IndexType>
      class basic_managed_xsi_shared_memory;
  }
}
```

Class template basic_managed_xsi_shared_memory

boost::interprocess::basic_managed_xsi_shared_memory

Synopsis

```
// In header: <boost/interprocess/managed_xsi_shared_memory.hpp>

template<typename CharType, typename AllocationAlgorithm,
         template< class IndexConfig > class IndexType>
class basic_managed_xsi_shared_memory {
public:
  // construct/copy/destruct
  basic_managed_xsi_shared_memory();
  basic_managed_xsi_shared_memory(create_only_t, const xsi_key &, std::size_t,
                                  const void * = 0,
                                  const permissions & = permissions());
  basic_managed_xsi_shared_memory(open_or_create_t, const xsi_key &,
                                  std::size_t, const void * = 0,
                                  const permissions & = permissions());
  basic_managed_xsi_shared_memory(open_read_only_t, const xsi_key &,
                                  const void * = 0);
  basic_managed_xsi_shared_memory(open_only_t, const xsi_key &,
                                  const void * = 0);
  basic_managed_xsi_shared_memory(basic_managed_xsi_shared_memory &&);
  basic_managed_xsi_shared_memory&
  operator=(basic_managed_xsi_shared_memory &&);
  ~basic_managed_xsi_shared_memory();

  // public member functions
  void swap(basic_managed_xsi_shared_memory &);
  int get_shmid() const;

  // public static functions
  static bool remove(int);
};
```

Description

A basic X/Open System Interface (XSI) shared memory named object creation class. Initializes the shared memory segment. Inherits all basic functionality from basic_managed_memory_impl<CharType, AllocationAlgorithm, IndexType>

`basic_managed_xsi_shared_memory` public construct/copy/destruct

1.
```
basic_managed_xsi_shared_memory();
```

Default constructor. Does nothing. Useful in combination with move semantics

2.
```
basic_managed_xsi_shared_memory(create_only_t create_only,
                                const xsi_key & key, std::size_t size,
                                const void * addr = 0,
                                const permissions & perm = permissions());
```

Creates shared memory and creates and places the segment manager. This can throw.

3.
```
basic_managed_xsi_shared_memory(open_or_create_t open_or_create,
                                const xsi_key & key, std::size_t size,
                                const void * addr = 0,
                                const permissions & perm = permissions());
```

Creates shared memory and creates and places the segment manager if segment was not created. If segment was created it connects to the segment. This can throw.

4.
```
basic_managed_xsi_shared_memory(open_read_only_t, const xsi_key & key,
                                const void * addr = 0);
```

Connects to a created shared memory and its segment manager. in read-only mode. This can throw.

5.
```
basic_managed_xsi_shared_memory(open_only_t open_only, const xsi_key & key,
                                const void * addr = 0);
```

Connects to a created shared memory and its segment manager. This can throw.

6.
```
basic_managed_xsi_shared_memory(basic_managed_xsi_shared_memory && moved);
```

Moves the ownership of "moved"'s managed memory to *this. Does not throw

7.
```
basic_managed_xsi_shared_memory&
operator=(basic_managed_xsi_shared_memory && moved);
```

Moves the ownership of "moved"'s managed memory to *this. Does not throw

8.
```
~basic_managed_xsi_shared_memory();
```

Destroys *this and indicates that the calling process is finished using the resource. The destructor function will deallocate any system resources allocated by the system for use by this process for this resource. The resource can still be opened again calling the open constructor overload. To erase the resource from the system use remove().

`basic_managed_xsi_shared_memory` public member functions

1.
```
void swap(basic_managed_xsi_shared_memory & other);
```

Swaps the ownership of the managed shared memories managed by *this and other. Never throws.

2.
```
int get_shmid() const;
```

`basic_managed_xsi_shared_memory` public static functions

1.
```
static bool remove(int shmid);
```

Erases a XSI shared memory object identified by shmid from the system. Returns false on error. Never throws

Header <boost/interprocess/mapped_region.hpp>

Describes mapped region class

```
namespace boost {
  namespace interprocess {
    class mapped_region;
  }
}
```

Class mapped_region

boost::interprocess::mapped_region

Synopsis

```
// In header: <boost/interprocess/mapped_region.hpp>


class mapped_region {
public:
  // construct/copy/destruct
  template<typename MemoryMappable>
    mapped_region(const MemoryMappable &, mode_t, offset_t = 0,
                  std::size_t = 0, const void * = 0);
  mapped_region();
  mapped_region(mapped_region &&);
  mapped_region& operator=(mapped_region &&);
  ~mapped_region();

  // public member functions
  std::size_t get_size() const;
  void * get_address() const;
  offset_t get_offset() const;
  mode_t get_mode() const;
  bool flush(std::size_t = 0, std::size_t = 0);
  void swap(mapped_region &);

  // public static functions
  static std::size_t get_page_size();
};
```

Description

The mapped_region class represents a portion or region created from a memory_mappable object.

mapped_region public construct/copy/destruct

1.
```
template<typename MemoryMappable>
  mapped_region(const MemoryMappable & mapping, mode_t mode,
                offset_t offset = 0, std::size_t size = 0,
                const void * address = 0);
```

Creates a mapping region of the mapped memory "mapping", starting in offset "offset", and the mapping's size will be "size". The mapping can be opened for read-only "read_only" or read-write "read_write.

2.
```
mapped_region();
```

Default constructor. Address and size and offset will be 0. Does not throw

3.
```
mapped_region(mapped_region && other);
```

Move constructor. *this will be constructed taking ownership of "other"'s region and "other" will be left in default constructor state.

4.
```
mapped_region& operator=(mapped_region && other);
```

Move assignment. If *this owns a memory mapped region, it will be destroyed and it will take ownership of "other"'s memory mapped region.

5.
```
~mapped_region();
```

Destroys the mapped region. Does not throw

`mapped_region` public member functions

1.
```
std::size_t get_size() const;
```

Returns the size of the mapping. Note for windows users: If windows_shared_memory is mapped using 0 as the size, it returns 0 because the size is unknown. Never throws.

2.
```
void * get_address() const;
```

Returns the base address of the mapping. Never throws.

3.
```
offset_t get_offset() const;
```

Returns the offset of the mapping from the beginning of the mapped memory. Never throws.

4.
```
mode_t get_mode() const;
```

Returns the mode of the mapping used to construct the mapped file. Never throws.

5.
```
bool flush(std::size_t mapping_offset = 0, std::size_t numbytes = 0);
```

Flushes to the disk a byte range within the mapped memory. Never throws

6.
```
void swap(mapped_region & other);
```

Swaps the mapped_region with another mapped region

`mapped_region` public static functions

1.
```
static std::size_t get_page_size();
```

Returns the size of the page. This size is the minimum memory that will be used by the system when mapping a memory mappable source.

Header <boost/interprocess/mem_algo/rbtree_best_fit.hpp>

Describes a best-fit algorithm based in an intrusive red-black tree used to allocate objects in shared memory. This class is intended as a base class for single segment and multi-segment implementations.

```
namespace boost {
  namespace interprocess {
    template<typename MutexFamily, typename VoidPointer,
            std::size_t MemAlignment>
      class rbtree_best_fit;
  }
}
```

Class template rbtree_best_fit

boost::interprocess::rbtree_best_fit

Synopsis

```
// In header: <boost/interprocess/mem_algo/rbtree_best_fit.hpp>

template<typename MutexFamily, typename VoidPointer, std::size_t MemAlignment>
class rbtree_best_fit {
public:
  // types
  typedef MutexFamily mutex_family;
  typedef VoidPointer void_pointer;            // Pointer type to be used with the rest of the ↵
Interprocess framework.
  typedef unspecified multiallocation_chain;

  // construct/copy/destruct
  rbtree_best_fit(std::size_t, std::size_t);
  ~rbtree_best_fit();

  // public member functions
  void * allocate(std::size_t);
  void deallocate(void *);
  std::size_t get_size() const;
  std::size_t get_free_memory() const;
  void zero_free_memory();
  void grow(std::size_t);
  void shrink_to_fit();
  bool all_memory_deallocated();
  bool check_sanity();
  template<typename T>
    std::pair< T *, bool >
    allocation_command(boost::interprocess::allocation_type, std::size_t,
                       std::size_t, std::size_t &, T * = 0);
  std::pair< void *, bool >
  raw_allocation_command(boost::interprocess::allocation_type, std::size_t,
                         std::size_t, std::size_t &, void * = 0,
                         std::size_t = 1);
  std::size_t size(const void *) const;
  void * allocate_aligned(std::size_t, std::size_t);

  // public static functions
  static std::size_t get_min_size(std::size_t);
  static const std::size_t PayloadPerAllocation;
};
```

Description

This class implements an algorithm that stores the free nodes in a red-black tree to have logarithmic search/insert times.

rbtree_best_fit public types

1. typedef MutexFamily mutex_family;

   Shared interprocess_mutex family used for the rest of the Interprocess framework

rbtree_best_fit public construct/copy/destruct

1.
```
rbtree_best_fit(std::size_t size, std::size_t extra_hdr_bytes);
```

Constructor. "size" is the total size of the managed memory segment, "extra_hdr_bytes" indicates the extra bytes beginning in the sizeof(rbtree_best_fit) offset that the allocator should not use at all.

2.
```
~rbtree_best_fit();
```

Destructor.

rbtree_best_fit public member functions

1.
```
void * allocate(std::size_t nbytes);
```

Allocates bytes, returns 0 if there is not more memory.

2.
```
void deallocate(void * addr);
```

Deallocates previously allocated bytes

3.
```
std::size_t get_size() const;
```

Returns the size of the memory segment.

4.
```
std::size_t get_free_memory() const;
```

Returns the number of free bytes of the segment.

5.
```
void zero_free_memory();
```

Initializes to zero all the memory that's not in use. This function is normally used for security reasons.

6.
```
void grow(std::size_t extra_size);
```

Increases managed memory in extra_size bytes more

7.
```
void shrink_to_fit();
```

Decreases managed memory as much as possible.

8.
```
bool all_memory_deallocated();
```

Returns true if all allocated memory has been deallocated.

9.
```
bool check_sanity();
```

Makes an internal sanity check and returns true if success

10.
```
template<typename T>
   std::pair< T *, bool >
   allocation_command(boost::interprocess::allocation_type command,
                      std::size_t limit_size, std::size_t preferred_size,
                      std::size_t & received_size, T * reuse_ptr = 0);
```

11.
```
std::pair< void *, bool >
raw_allocation_command(boost::interprocess::allocation_type command,
                       std::size_t limit_object,
                       std::size_t preferred_object,
                       std::size_t & received_object, void * reuse_ptr = 0,
                       std::size_t sizeof_object = 1);
```

12.
```
std::size_t size(const void * ptr) const;
```

Returns the size of the buffer previously allocated pointed by ptr.

13.
```
void * allocate_aligned(std::size_t nbytes, std::size_t alignment);
```

Allocates aligned bytes, returns 0 if there is not more memory. Alignment must be power of 2

`rbtree_best_fit` **public static functions**

1.
```
static std::size_t get_min_size(std::size_t extra_hdr_bytes);
```

Obtains the minimum size needed by the algorithm.

Header <boost/interprocess/mem_algo/simple_seq_fit.hpp>

Describes sequential fit algorithm used to allocate objects in shared memory.

```
namespace boost {
  namespace interprocess {
    template<typename MutexFamily, typename VoidPointer> class simple_seq_fit;
  }
}
```

Class template simple_seq_fit

boost::interprocess::simple_seq_fit

Synopsis

```
// In header: <boost/interprocess/mem_algo/simple_seq_fit.hpp>

template<typename MutexFamily, typename VoidPointer>
class simple_seq_fit {
public:
  // construct/copy/destruct
  simple_seq_fit(std::size_t, std::size_t);
};
```

Description

This class implements the simple sequential fit algorithm with a simply linked list of free buffers.

`simple_seq_fit` public construct/copy/destruct

1.
```
simple_seq_fit(std::size_t size, std::size_t extra_hdr_bytes);
```

Constructor. "size" is the total size of the managed memory segment, "extra_hdr_bytes" indicates the extra bytes beginning in the sizeof(simple_seq_fit) offset that the allocator should not use at all.

Header <boost/interprocess/offset_ptr.hpp>

Describes a smart pointer that stores the offset between this pointer and target pointee, called offset_ptr.

```
namespace boost {
  namespace interprocess {
    template<typename PointedType> class offset_ptr;
    template<typename T1, typename T2>
      bool operator==(const offset_ptr< T1 > &, const offset_ptr< T2 > &);
    template<typename T1, typename T2>
      bool operator!=(const offset_ptr< T1 > &, const offset_ptr< T2 > &);
    template<typename T1, typename T2>
      bool operator<(const offset_ptr< T1 > &, const offset_ptr< T2 > &);
    template<typename T1, typename T2>
      bool operator<=(const offset_ptr< T1 > &, const offset_ptr< T2 > &);
    template<typename T1, typename T2>
      bool operator>(const offset_ptr< T1 > &, const offset_ptr< T2 > &);
    template<typename T1, typename T2>
      bool operator>=(const offset_ptr< T1 > &, const offset_ptr< T2 > &);
    template<typename E, typename T, typename Y>
      std::basic_ostream< E, T > &
      operator<<(std::basic_ostream< E, T > &, offset_ptr< Y > const &);
    template<typename E, typename T, typename Y>
      std::basic_istream< E, T > &
      operator>>(std::basic_istream< E, T > &, offset_ptr< Y > &);
    template<typename T>
      offset_ptr< T > operator+(std::ptrdiff_t, const offset_ptr< T > &);
    template<typename T, typename T2>
      std::ptrdiff_t
      operator-(const offset_ptr< T > &, const offset_ptr< T2 > &);
    template<typename T>
      void swap(boost::interprocess::offset_ptr< T > &,
                boost::interprocess::offset_ptr< T > &);

    // Simulation of static_cast between pointers. Never throws.
    template<typename T, typename U>
      boost::interprocess::offset_ptr< T >
      static_pointer_cast(const boost::interprocess::offset_ptr< U > & r);

    // Simulation of const_cast between pointers. Never throws.
    template<typename T, typename U>
      boost::interprocess::offset_ptr< T >
      const_pointer_cast(const boost::interprocess::offset_ptr< U > & r);

    // Simulation of dynamic_cast between pointers. Never throws.
    template<typename T, typename U>
      boost::interprocess::offset_ptr< T >
      dynamic_pointer_cast(const boost::interprocess::offset_ptr< U > & r);

    // Simulation of reinterpret_cast between pointers. Never throws.
    template<typename T, typename U>
      boost::interprocess::offset_ptr< T >
      reinterpret_pointer_cast(const boost::interprocess::offset_ptr< U > & r);
  }
}
```

Class template offset_ptr

boost::interprocess::offset_ptr

Synopsis

```
// In header: <boost/interprocess/offset_ptr.hpp>

template<typename PointedType>
class offset_ptr {
public:
  // types
  typedef PointedType *                   pointer;
  typedef unspecified                     reference;
  typedef PointedType                     value_type;
  typedef std::ptrdiff_t                  difference_type;
  typedef std::random_access_iterator_tag iterator_category;

  // construct/copy/destruct
  offset_ptr(pointer = 0);
  template<typename T> offset_ptr(T *);
  offset_ptr(const offset_ptr &);
  template<typename T2> offset_ptr(const offset_ptr< T2 > &);
  template<typename Y> offset_ptr(const offset_ptr< Y > &, unspecified);
  template<typename Y> offset_ptr(const offset_ptr< Y > &, unspecified);
  template<typename Y> offset_ptr(const offset_ptr< Y > &, unspecified);
  template<typename Y> offset_ptr(const offset_ptr< Y > &, unspecified);
  offset_ptr& operator=(pointer);
  offset_ptr& operator=(const offset_ptr &);
  template<typename T2> offset_ptr& operator=(const offset_ptr< T2 > &);

  // public member functions
  pointer get() const;
  std::ptrdiff_t get_offset() const;
  pointer operator->() const;
  reference operator*() const;
  reference operator[](std::ptrdiff_t) const;
  offset_ptr operator+(std::ptrdiff_t) const;
  offset_ptr operator-(std::ptrdiff_t) const;
  offset_ptr & operator+=(std::ptrdiff_t);
  offset_ptr & operator-=(std::ptrdiff_t);
  offset_ptr & operator++(void);
  offset_ptr operator++(int);
  offset_ptr & operator--(void);
  offset_ptr operator--(int);
  operator unspecified_bool_type() const;
  bool operator!() const;
};
```

Description

A smart pointer that stores the offset between between the pointer and the the object it points. This allows offset allows special properties, since the pointer is independent from the address address of the pointee, if the pointer and the pointee are still separated by the same offset. This feature converts offset_ptr in a smart pointer that can be placed in shared memory and memory mapped files mapped in different addresses in every process.

`offset_ptr` public construct/copy/destruct

1.
```
offset_ptr(pointer ptr = 0);
```

Constructor from raw pointer (allows "0" pointer conversion). Never throws.

2.
```
template<typename T> offset_ptr(T * ptr);
```

Constructor from other pointer. Never throws.

3.
```
offset_ptr(const offset_ptr & ptr);
```

Constructor from other offset_ptr Never throws.

4.
```
template<typename T2> offset_ptr(const offset_ptr< T2 > & ptr);
```

Constructor from other offset_ptr. If pointers of pointee types are convertible, offset_ptrs will be convertibles. Never throws.

5.
```
template<typename Y> offset_ptr(const offset_ptr< Y > & r, unspecified);
```

Emulates static_cast operator. Never throws.

6.
```
template<typename Y> offset_ptr(const offset_ptr< Y > & r, unspecified);
```

Emulates const_cast operator. Never throws.

7.
```
template<typename Y> offset_ptr(const offset_ptr< Y > & r, unspecified);
```

Emulates dynamic_cast operator. Never throws.

8.
```
template<typename Y> offset_ptr(const offset_ptr< Y > & r, unspecified);
```

Emulates reinterpret_cast operator. Never throws.

9.
```
offset_ptr& operator=(pointer from);
```

Assignment from pointer (saves extra conversion). Never throws.

10.
```
offset_ptr& operator=(const offset_ptr & pt);
```

Assignment from other offset_ptr. Never throws.

11.
```
template<typename T2> offset_ptr& operator=(const offset_ptr< T2 > & pt);
```

Assignment from related offset_ptr. If pointers of pointee types are assignable, offset_ptrs will be assignable. Never throws.

offset_ptr public member functions

1.
```
pointer get() const;
```

Obtains raw pointer from offset. Never throws.

2.
```
std::ptrdiff_t get_offset() const;
```

3.
```
pointer operator->() const;
```

Pointer-like -> operator. It can return 0 pointer. Never throws.

4.
```
reference operator*() const;
```

Dereferencing operator, if it is a null offset_ptr behavior is undefined. Never throws.

5.
```
reference operator[](std::ptrdiff_t idx) const;
```

Indexing operator. Never throws.

6.
```
offset_ptr operator+(std::ptrdiff_t offset) const;
```

offset_ptr + std::ptrdiff_t. Never throws.

7.
```
offset_ptr operator-(std::ptrdiff_t offset) const;
```

offset_ptr - std::ptrdiff_t. Never throws.

8.
```
offset_ptr & operator+=(std::ptrdiff_t offset);
```

offset_ptr += std::ptrdiff_t. Never throws.

9.
```
offset_ptr & operator-=(std::ptrdiff_t offset);
```

offset_ptr -= std::ptrdiff_t. Never throws.

10.
```
offset_ptr & operator++(void);
```

++offset_ptr. Never throws.

11.
```
offset_ptr operator++(int);
```

offset_ptr++. Never throws.

12.
```
offset_ptr & operator--(void);
```

--offset_ptr. Never throws.

13.
```
offset_ptr operator--(int);
```

offset_ptr--. Never throws.

14.
```
operator unspecified_bool_type() const;
```

safe bool conversion operator. Never throws.

15.
```
bool operator!() const;
```

Not operator. Not needed in theory, but improves portability. Never throws

Function template operator==

boost::interprocess::operator==

Synopsis

```
// In header: <boost/interprocess/offset_ptr.hpp>


template<typename T1, typename T2>
  bool operator==(const offset_ptr< T1 > & pt1, const offset_ptr< T2 > & pt2);
```

Description

offset_ptr<T1> == offset_ptr<T2>. Never throws.

Function template operator!=

boost::interprocess::operator!=

Synopsis

```
// In header: <boost/interprocess/offset_ptr.hpp>


template<typename T1, typename T2>
  bool operator!=(const offset_ptr< T1 > & pt1, const offset_ptr< T2 > & pt2);
```

Description

offset_ptr<T1> != offset_ptr<T2>. Never throws.

Function template operator<

boost::interprocess::operator<

Synopsis

```
// In header: <boost/interprocess/offset_ptr.hpp>


template<typename T1, typename T2>
  bool operator<(const offset_ptr< T1 > & pt1, const offset_ptr< T2 > & pt2);
```

Description

offset_ptr<T1> < offset_ptr<T2>. Never throws.

Function template operator<=

boost::interprocess::operator<=

Synopsis

```
// In header: <boost/interprocess/offset_ptr.hpp>


template<typename T1, typename T2>
  bool operator<=(const offset_ptr< T1 > & pt1, const offset_ptr< T2 > & pt2);
```

Description

offset_ptr<T1> <= offset_ptr<T2>. Never throws.

Function template operator>

boost::interprocess::operator>

Synopsis

```
// In header: <boost/interprocess/offset_ptr.hpp>


template<typename T1, typename T2>
  bool operator>(const offset_ptr< T1 > & pt1, const offset_ptr< T2 > & pt2);
```

Description

offset_ptr<T1> > offset_ptr<T2>. Never throws.

Function template operator>=

boost::interprocess::operator>=

Synopsis

```
// In header: <boost/interprocess/offset_ptr.hpp>


template<typename T1, typename T2>
  bool operator>=(const offset_ptr< T1 > & pt1, const offset_ptr< T2 > & pt2);
```

Description

offset_ptr<T1> >= offset_ptr<T2>. Never throws.

Function template operator<<

boost::interprocess::operator<<

Synopsis

```
// In header: <boost/interprocess/offset_ptr.hpp>


template<typename E, typename T, typename Y>
  std::basic_ostream< E, T > &
  operator<<(std::basic_ostream< E, T > & os, offset_ptr< Y > const & p);
```

Description

operator<< for offset ptr

Function template operator>>

boost::interprocess::operator>>

Synopsis

```
// In header: <boost/interprocess/offset_ptr.hpp>


template<typename E, typename T, typename Y>
  std::basic_istream< E, T > &
  operator>>(std::basic_istream< E, T > & is, offset_ptr< Y > & p);
```

Description

operator>> for offset ptr

Function template operator+

boost::interprocess::operator+

Synopsis

```
// In header: <boost/interprocess/offset_ptr.hpp>


template<typename T>
  offset_ptr< T >
  operator+(std::ptrdiff_t diff, const offset_ptr< T > & right);
```

Description

std::ptrdiff_t + offset_ptr operation

Function template operator-

boost::interprocess::operator-

Synopsis

```
// In header: <boost/interprocess/offset_ptr.hpp>


template<typename T, typename T2>
  std::ptrdiff_t
  operator-(const offset_ptr< T > & pt, const offset_ptr< T2 > & pt2);
```

Description

offset_ptr - offset_ptr operation

Function template swap

boost::interprocess::swap

Synopsis

```
// In header: <boost/interprocess/offset_ptr.hpp>


template<typename T>
  void swap(boost::interprocess::offset_ptr< T > & pt,
            boost::interprocess::offset_ptr< T > & pt2);
```

Description

swap specialization for offset_ptr

Header <boost/interprocess/permissions.hpp>

Describes permissions class

```
namespace boost {
  namespace interprocess {
    class permissions;
  }
}
```

Class permissions

boost::interprocess::permissions

Synopsis

```
// In header: <boost/interprocess/permissions.hpp>


class permissions {
public:
  // construct/copy/destruct
  permissions(os_permissions_type);
  permissions();

  // public member functions
  void set_default();
  void set_unrestricted();
  void set_permissions(os_permissions_type);
  os_permissions_type get_permissions() const;
};
```

Description

The permissions class represents permissions to be set to shared memory or files, that can be constructed form usual permission representations: a SECURITY_ATTRIBUTES pointer in windows or ORed rwx chmod integer in UNIX.

`permissions` public construct/copy/destruct

1.
   ```
   permissions(os_permissions_type type);
   ```

   Constructs a permissions object from a user provided os-dependent permissions.

2.
   ```
   permissions();
   ```

   Constructs a default permissions object: A null security attributes pointer for windows or 0644 for UNIX.

`permissions` public member functions

1.
   ```
   void set_default();
   ```

   Sets permissions to default values: A null security attributes pointer for windows or 0644 for UNIX.

2.
   ```
   void set_unrestricted();
   ```

   Sets permissions to unrestricted access: A null DACL for windows or 0666 for UNIX.

3.
   ```
   void set_permissions(os_permissions_type perm);
   ```

   Sets permissions from a user provided os-dependent permissions.

4.
   ```
   os_permissions_type get_permissions() const;
   ```

   Returns stored os-dependent permissions

Header <boost/interprocess/segment_manager.hpp>

Describes the object placed in a memory segment that provides named object allocation capabilities for single-segment and multi-segment allocations.

```
namespace boost {
  namespace interprocess {
    template<typename MemoryAlgorithm> class segment_manager_base;
    template<typename CharType, typename MemoryAlgorithm,
             template< class IndexConfig > class IndexType>
      class segment_manager;
  }
}
```

Class template segment_manager_base

boost::interprocess::segment_manager_base

Synopsis

```cpp
// In header: <boost/interprocess/segment_manager.hpp>


template<typename MemoryAlgorithm>
class segment_manager_base : private MemoryAlgorithm {
public:
  // types
  typedef segment_manager_base< MemoryAlgorithm > segment_manager_base_type;
  typedef MemoryAlgorithm::void_pointer           void_pointer;
  typedef MemoryAlgorithm::mutex_family           mutex_family;
  typedef MemoryAlgorithm                         memory_algorithm;

  // construct/copy/destruct
  segment_manager_base(std::size_t, std::size_t);

  // public member functions
  std::size_t get_size() const;
  std::size_t get_free_memory() const;
  void * allocate(std::size_t, std::nothrow_t);
  void * allocate(std::size_t);
  void * allocate_aligned(std::size_t, std::size_t, std::nothrow_t);
  void * allocate_aligned(std::size_t, std::size_t);
  template<typename T>
    std::pair< T *, bool >
    allocation_command(boost::interprocess::allocation_type, std::size_t,
                       std::size_t, std::size_t &, T * = 0);
  std::pair< void *, bool >
  raw_allocation_command(boost::interprocess::allocation_type, std::size_t,
                         std::size_t, std::size_t &, void * = 0,
                         std::size_t = 1);
  void deallocate(void *);
  void grow(std::size_t);
  void shrink_to_fit();
  bool all_memory_deallocated();
  bool check_sanity();
  void zero_free_memory();
  std::size_t size(const void *) const;

  // public static functions
  static std::size_t get_min_size(std::size_t);
  static const std::size_t PayloadPerAllocation;
};
```

Description

This object is the public base class of segment manager. This class only depends on the memory allocation algorithm and implements all the allocation features not related to named or unique objects.

Storing a reference to segment_manager forces the holder class to be dependent on index types and character types. When such dependence is not desirable and only anonymous and raw allocations are needed, segment_manager_base is the correct answer.

segment_manager_base public construct/copy/destruct

1.
```cpp
segment_manager_base(std::size_t size, std::size_t reserved_bytes);
```

Constructor of the segment_manager_base

"size" is the size of the memory segment where the basic segment manager is being constructed.

"reserved_bytes" is the number of bytes after the end of the memory algorithm object itself that the memory algorithm will exclude from dynamic allocation

Can throw

`segment_manager_base` public member functions

1.
```
std::size_t get_size() const;
```

Returns the size of the memory segment

2.
```
std::size_t get_free_memory() const;
```

Returns the number of free bytes of the memory segment

3.
```
void * allocate(std::size_t nbytes, std::nothrow_t);
```

Allocates nbytes bytes. This function is only used in single-segment management. Never throws

4.
```
void * allocate(std::size_t nbytes);
```

Allocates nbytes bytes. Throws boost::interprocess::bad_alloc on failure

5.
```
void * allocate_aligned(std::size_t nbytes, std::size_t alignment,
                        std::nothrow_t);
```

Allocates nbytes bytes. This function is only used in single-segment management. Never throws

6.
```
void * allocate_aligned(std::size_t nbytes, std::size_t alignment);
```

Allocates nbytes bytes. This function is only used in single-segment management. Throws bad_alloc when fails

7.
```
template<typename T>
   std::pair< T *, bool >
   allocation_command(boost::interprocess::allocation_type command,
                      std::size_t limit_size, std::size_t preferred_size,
                      std::size_t & received_size, T * reuse_ptr = 0);
```

8.
```
std::pair< void *, bool >
raw_allocation_command(boost::interprocess::allocation_type command,
                       std::size_t limit_objects,
                       std::size_t preferred_objects,
                       std::size_t & received_objects, void * reuse_ptr = 0,
                       std::size_t sizeof_object = 1);
```

9.
```
void deallocate(void * addr);
```

Deallocates the bytes allocated with allocate/allocate_many() pointed by addr

10.
```
void grow(std::size_t extra_size);
```

Increases managed memory in extra_size bytes more. This only works with single-segment management.

11.
```
void shrink_to_fit();
```

Decreases managed memory to the minimum. This only works with single-segment management.

12.
```
bool all_memory_deallocated();
```

Returns the result of "all_memory_deallocated()" function of the used memory algorithm

13.
```
bool check_sanity();
```

Returns the result of "check_sanity()" function of the used memory algorithm

14.
```
void zero_free_memory();
```

Writes to zero free memory (memory not yet allocated) of the memory algorithm

15.
```
std::size_t size(const void * ptr) const;
```

Returns the size of the buffer previously allocated pointed by ptr.

segment_manager_base public static functions

1.
```
static std::size_t get_min_size(std::size_t size);
```

Obtains the minimum size needed by the segment manager

Class template segment_manager

boost::interprocess::segment_manager

Synopsis

```
// In header: <boost/interprocess/segment_manager.hpp>

template<typename CharType, typename MemoryAlgorithm,
         template< class IndexConfig > class IndexType>
class segment_manager :
  public boost::interprocess::segment_manager_base< MemoryAlgorithm >
{
public:
  // types
  typedef MemoryAlgorithm                                           memory_al↵
gorithm;
  typedef Base::void_pointer                                        ↵
 void_pointer;
  typedef CharType                                                  ↵
 char_type;
  typedef segment_manager_base< MemoryAlgorithm >                   seg↵
ment_manager_base_type;
  typedef Base::mutex_family                                        mu↵
tex_family;
  typedef transform_iterator< typename named_index_t::const_iterator, named_transform >  ↵
 const_named_iterator;
  typedef transform_iterator< typename unique_index_t::const_iterator, unique_trans↵
form > const_unique_iterator;

  // member classes/structs/unions
  template<typename T>
  struct allocator {
    // types
    typedef boost::interprocess::allocator< T, segment_manager > type;
  };
  template<typename T>
  struct deleter {
    // types
    typedef boost::interprocess::deleter< T, segment_manager > type;
  };

  // construct/copy/destruct
  segment_manager(std::size_t);

  // public member functions
  template<typename T> std::pair< T *, std::size_t > find(const CharType *);
  template<typename T> std::pair< T *, std::size_t > find(unspecified);
  template<typename T>
    std::pair< T *, std::size_t > find_no_lock(const CharType *);
  template<typename T> std::pair< T *, std::size_t > find_no_lock(unspecified);
  template<typename T> construct_proxy< T >::type construct(char_ptr_holder_t);
  template<typename T>
    construct_proxy< T >::type find_or_construct(char_ptr_holder_t);
  template<typename T>
    construct_proxy< T >::type construct(char_ptr_holder_t, std::nothrow_t);
  template<typename T>
    construct_proxy< T >::type
    find_or_construct(char_ptr_holder_t, std::nothrow_t);
  template<typename T>
    construct_iter_proxy< T >::type construct_it(char_ptr_holder_t);
  template<typename T>
```

```
      construct_iter_proxy< T >::type find_or_construct_it(char_ptr_holder_t);
  template<typename T>
      construct_iter_proxy< T >::type
      construct_it(char_ptr_holder_t, std::nothrow_t);
  template<typename T>
      construct_iter_proxy< T >::type
      find_or_construct_it(char_ptr_holder_t, std::nothrow_t);
  template<typename Func> *void atomic_func(Func &);
  template<typename Func> bool try_atomic_func(Func &);
  template<typename T> bool destroy(unspecified);
  template<typename T> bool destroy(const CharType *);
  template<typename T> void destroy_ptr(const T *);
  void reserve_named_objects(std::size_t);
  void reserve_unique_objects(std::size_t);
  void shrink_to_fit_indexes();
  std::size_t get_num_named_objects();
  std::size_t get_num_unique_objects();
  const_named_iterator named_begin() const;
  const_named_iterator named_end() const;
  const_unique_iterator unique_begin() const;
  const_unique_iterator unique_end() const;
  template<typename T> allocator< T >::type get_allocator();
  template<typename T> deleter< T >::type get_deleter();

  // public static functions
  template<typename T> static const CharType * get_instance_name(const T *);
  template<typename T> static std::size_t get_instance_length(const T *);
  template<typename T> static instance_type get_instance_type(const T *);
  static std::size_t get_min_size();
  static const std::size_t PayloadPerAllocation;
};
```

Description

This object is placed in the beginning of memory segment and implements the allocation (named or anonymous) of portions of the segment. This object contains two indexes that maintain an association between a name and a portion of the segment.

The first index contains the mappings for normal named objects using the char type specified in the template parameter.

The second index contains the association for unique instances. The key will be the const char * returned from type_info.name() function for the unique type to be constructed.

segment_manager<CharType, MemoryAlgorithm, IndexType> inherits publicly from segment_manager_base<MemoryAlgorithm> and inherits from it many public functions related to anonymous object and raw memory allocation. See segment_manager_base reference to know about those functions.

`segment_manager` public construct/copy/destruct

1.
```
segment_manager(std::size_t size);
```

Constructor of the segment manager "size" is the size of the memory segment where the segment manager is being constructed. Can throw

`segment_manager` public member functions

1.
```
template<typename T> std::pair< T *, std::size_t > find(const CharType * name);
```

Tries to find a previous named allocation. Returns the address and the object count. On failure the first member of the returned pair is 0.

2.
```
template<typename T> std::pair< T *, std::size_t > find(unspecified name);
```

Tries to find a previous unique allocation. Returns the address and the object count. On failure the first member of the returned pair is 0.

3.
```
template<typename T>
    std::pair< T *, std::size_t > find_no_lock(const CharType * name);
```

Tries to find a previous named allocation. Returns the address and the object count. On failure the first member of the returned pair is 0. This search is not mutex-protected!

4.
```
template<typename T>
    std::pair< T *, std::size_t > find_no_lock(unspecified name);
```

Tries to find a previous unique allocation. Returns the address and the object count. On failure the first member of the returned pair is 0. This search is not mutex-protected!

5.
```
template<typename T>
    construct_proxy< T >::type construct(char_ptr_holder_t name);
```

Returns throwing "construct" proxy object

6.
```
template<typename T>
    construct_proxy< T >::type find_or_construct(char_ptr_holder_t name);
```

Returns throwing "search or construct" proxy object

7.
```
template<typename T>
    construct_proxy< T >::type construct(char_ptr_holder_t name, std::nothrow_t);
```

Returns no throwing "construct" proxy object

8.
```
template<typename T>
    construct_proxy< T >::type
    find_or_construct(char_ptr_holder_t name, std::nothrow_t);
```

Returns no throwing "search or construct" proxy object

9.
```
template<typename T>
    construct_iter_proxy< T >::type construct_it(char_ptr_holder_t name);
```

Returns throwing "construct from iterators" proxy object.

10.
```
template<typename T>
    construct_iter_proxy< T >::type find_or_construct_it(char_ptr_holder_t name);
```

Returns throwing "search or construct from iterators" proxy object

11.
```
template<typename T>
    construct_iter_proxy< T >::type
    construct_it(char_ptr_holder_t name, std::nothrow_t);
```

Returns no throwing "construct from iterators" proxy object

12.

```
template<typename T>
   construct_iter_proxy< T >::type
   find_or_construct_it(char_ptr_holder_t name, std::nothrow_t);
```

Returns no throwing "search or construct from iterators" proxy object

13.

```
template<typename Func> *void atomic_func(Func & f);
```

Calls object function blocking recursive interprocess_mutex and guarantees that no new named_alloc or destroy will be executed by any process while executing the object function call

14.

```
template<typename Func> bool try_atomic_func(Func & f);
```

Tries to calls a functor guaranteeing that no new construction, search or destruction will be executed by any process while executing the object function call. If the atomic function can't be immediately executed because the internal mutex is already locked, returns false. If the functor throws, this function throws.

15.

```
template<typename T> bool destroy(unspecified);
```

Destroys a previously created unique instance. Returns false if the object was not present.

16.

```
template<typename T> bool destroy(const CharType * name);
```

Destroys the named object with the given name. Returns false if that object can't be found.

17.

```
template<typename T> void destroy_ptr(const T * p);
```

Destroys an anonymous, unique or named object using it's address

18.

```
void reserve_named_objects(std::size_t num);
```

Preallocates needed index resources to optimize the creation of "num" named objects in the managed memory segment. Can throw boost::interprocess::bad_alloc if there is no enough memory.

19.

```
void reserve_unique_objects(std::size_t num);
```

Preallocates needed index resources to optimize the creation of "num" unique objects in the managed memory segment. Can throw boost::interprocess::bad_alloc if there is no enough memory.

20.

```
void shrink_to_fit_indexes();
```

Calls shrink_to_fit in both named and unique object indexes to try to free unused memory from those indexes.

21.

```
std::size_t get_num_named_objects();
```

Returns the number of named objects stored in the segment.

22.

```
std::size_t get_num_unique_objects();
```

Returns the number of unique objects stored in the segment.

23.
```
const_named_iterator named_begin() const;
```

Returns a constant iterator to the beginning of the information about the named allocations performed in this segment manager

24.
```
const_named_iterator named_end() const;
```

Returns a constant iterator to the end of the information about the named allocations performed in this segment manager

25.
```
const_unique_iterator unique_begin() const;
```

Returns a constant iterator to the beginning of the information about the unique allocations performed in this segment manager

26.
```
const_unique_iterator unique_end() const;
```

Returns a constant iterator to the end of the information about the unique allocations performed in this segment manager

27.
```
template<typename T> allocator< T >::type get_allocator();
```

Returns an instance of the default allocator for type T initialized that allocates memory from this segment manager.

28.
```
template<typename T> deleter< T >::type get_deleter();
```

Returns an instance of the default allocator for type T initialized that allocates memory from this segment manager.

segment_manager public static functions

1.
```
template<typename T> static const CharType * get_instance_name(const T * ptr);
```

Returns the name of an object created with construct/find_or_construct functions. Does not throw

2.
```
template<typename T> static std::size_t get_instance_length(const T * ptr);
```

Returns the length of an object created with construct/find_or_construct functions. Does not throw.

3.
```
template<typename T> static instance_type get_instance_type(const T * ptr);
```

Returns is the the name of an object created with construct/find_or_construct functions. Does not throw

4.
```
static std::size_t get_min_size();
```

Obtains the minimum size needed by the segment manager

Struct template allocator

boost::interprocess::segment_manager::allocator

Synopsis

```
// In header: <boost/interprocess/segment_manager.hpp>


template<typename T>
struct allocator {
  // types
  typedef boost::interprocess::allocator< T, segment_manager > type;
};
```

Description

This is the default allocator to allocate types T from this managed segment

Struct template deleter

boost::interprocess::segment_manager::deleter

Synopsis

```
// In header: <boost/interprocess/segment_manager.hpp>


template<typename T>
struct deleter {
  // types
  typedef boost::interprocess::deleter< T, segment_manager > type;
};
```

Description

This is the default deleter to delete types T from this managed segment.

Header <boost/interprocess/shared_memory_object.hpp>

Describes a shared memory object management class.

```
namespace boost {
  namespace interprocess {
    class shared_memory_object;
    class remove_shared_memory_on_destroy;
  }
}
```

Class shared_memory_object

boost::interprocess::shared_memory_object

Synopsis

```
// In header: <boost/interprocess/shared_memory_object.hpp>


class shared_memory_object {
public:
  // construct/copy/destruct
  shared_memory_object();
  shared_memory_object(create_only_t, const char *, mode_t,
                       const permissions & = permissions());
  shared_memory_object(open_or_create_t, const char *, mode_t,
                       const permissions & = permissions());
  shared_memory_object(open_only_t, const char *, mode_t);
  shared_memory_object(shared_memory_object &&);
  shared_memory_object& operator=(shared_memory_object &&);
  ~shared_memory_object();

  // public member functions
  void swap(shared_memory_object &);
  void truncate(offset_t);
  const char * get_name() const;
  bool get_size(offset_t &) const;
  mode_t get_mode() const;
  mapping_handle_t get_mapping_handle() const;

  // public static functions
  static bool remove(const char *);
};
```

Description

A class that wraps a shared memory mapping that can be used to create mapped regions from the mapped files

`shared_memory_object` public construct/copy/destruct

1.
```
shared_memory_object();
```

Default constructor. Represents an empty shared_memory_object.

2.
```
shared_memory_object(create_only_t, const char * name, mode_t mode,
                     const permissions & perm = permissions());
```

Creates a shared memory object with name "name" and mode "mode", with the access mode "mode" If the file previously exists, throws an error.

3.
```
shared_memory_object(open_or_create_t, const char * name, mode_t mode,
                     const permissions & perm = permissions());
```

Tries to create a shared memory object with name "name" and mode "mode", with the access mode "mode". If the file previously exists, it tries to open it with mode "mode". Otherwise throws an error.

4.
```
shared_memory_object(open_only_t, const char * name, mode_t mode);
```

Tries to open a shared memory object with name "name", with the access mode "mode". If the file does not previously exist, it throws an error.

5.
```
shared_memory_object(shared_memory_object && moved);
```

Moves the ownership of "moved"'s shared memory object to *this. After the call, "moved" does not represent any shared memory object. Does not throw

6.
```
shared_memory_object& operator=(shared_memory_object && moved);
```

Moves the ownership of "moved"'s shared memory to *this. After the call, "moved" does not represent any shared memory. Does not throw

7.
```
~shared_memory_object();
```

Destroys *this and indicates that the calling process is finished using the resource. All mapped regions are still valid after destruction. The destructor function will deallocate any system resources allocated by the system for use by this process for this resource. The resource can still be opened again calling the open constructor overload. To erase the resource from the system use remove().

`shared_memory_object` public member functions

1.
```
void swap(shared_memory_object & moved);
```

Swaps the shared_memory_objects. Does not throw.

2.
```
void truncate(offset_t length);
```

Sets the size of the shared memory mapping.

3.
```
const char * get_name() const;
```

Returns the name of the shared memory object.

4.
```
bool get_size(offset_t & size) const;
```

Returns true if the size of the shared memory object can be obtained and writes the size in the passed reference

5.
```
mode_t get_mode() const;
```

Returns access mode.

6.
```
mapping_handle_t get_mapping_handle() const;
```

Returns mapping handle. Never throws.

`shared_memory_object` public static functions

1.
```
static bool remove(const char * name);
```

Erases a shared memory object from the system. Returns false on error. Never throws

Class remove_shared_memory_on_destroy

boost::interprocess::remove_shared_memory_on_destroy

Synopsis

```
// In header: <boost/interprocess/shared_memory_object.hpp>


class remove_shared_memory_on_destroy {
public:
  // construct/copy/destruct
  remove_shared_memory_on_destroy(const char *);
  ~remove_shared_memory_on_destroy();
};
```

Description

A class that stores the name of a shared memory and calls shared_memory_object::remove(name) in its destructor Useful to remove temporary shared memory objects in the presence of exceptions

remove_shared_memory_on_destroy public construct/copy/destruct

1.
```
remove_shared_memory_on_destroy(const char * name);
```

2.
```
~remove_shared_memory_on_destroy();
```

Header <boost/interprocess/smart_ptr/deleter.hpp>

Describes the functor to delete objects from the segment.

```
namespace boost {
  namespace interprocess {
    template<typename T, typename SegmentManager> class deleter;
  }
}
```

Class template deleter

boost::interprocess::deleter

Synopsis

```
// In header: <boost/interprocess/smart_ptr/deleter.hpp>

template<typename T, typename SegmentManager>
class deleter {
public:
  // types
  typedef boost::pointer_to_other< typename SegmentManager::void_pointer, T >::type pointer;

  // construct/copy/destruct
  deleter(segment_manager_pointer);

  // public member functions
  void operator()(const pointer &);
};
```

Description

A deleter that uses the segment manager's destroy_ptr function to destroy the passed pointer resource.

This deleter is used

`deleter` public construct/copy/destruct

1.
```
deleter(segment_manager_pointer pmngr);
```

`deleter` public member functions

1.
```
void operator()(const pointer & p);
```

Header <boost/interprocess/smart_ptr/en­able_shared_from_this.hpp>

Describes an utility to form a shared pointer from this

```
namespace boost {
  namespace interprocess {
    template<typename T, typename A, typename D> class enable_shared_from_this;
  }
}
```

Class template enable_shared_from_this

boost::interprocess::enable_shared_from_this

Synopsis

```
// In header: <boost/interprocess/smart_ptr/enable_shared_from_this.hpp>

template<typename T, typename A, typename D>
class enable_shared_from_this {
public:

  // public member functions
  shared_ptr< T, A, D > shared_from_this();
  shared_ptr< T const, A, D > shared_from_this() const;
};
```

Description

This class is used as a base class that allows a shared_ptr to the current object to be obtained from within a member function. enable_shared_from_this defines two member functions called shared_from_this that return a shared_ptr<T> and shared_ptr<T const>, depending on constness, to this.

enable_shared_from_this public member functions

1.
   ```
   shared_ptr< T, A, D > shared_from_this();
   ```

2.
   ```
   shared_ptr< T const, A, D > shared_from_this() const;
   ```

Header <boost/interprocess/smart_ptr/intrusive_ptr.hpp>

Describes an intrusive ownership pointer.

```
namespace boost {
  namespace interprocess {
    template<typename T, typename VoidPointer> class intrusive_ptr;
    template<typename T, typename U, typename VP>
      bool operator==(intrusive_ptr< T, VP > const &,
                      intrusive_ptr< U, VP > const &);
    template<typename T, typename U, typename VP>
      bool operator!=(intrusive_ptr< T, VP > const &,
                      intrusive_ptr< U, VP > const &);
    template<typename T, typename VP>
      bool operator==(intrusive_ptr< T, VP > const &,
                      const typename intrusive_ptr< T, VP >::pointer &);
    template<typename T, typename VP>
      bool operator!=(intrusive_ptr< T, VP > const &,
                      const typename intrusive_ptr< T, VP >::pointer &);
    template<typename T, typename VP>
      bool operator==(const typename intrusive_ptr< T, VP >::pointer &,
                      intrusive_ptr< T, VP > const &);
    template<typename T, typename VP>
      bool operator!=(const typename intrusive_ptr< T, VP >::pointer &,
                      intrusive_ptr< T, VP > const &);
    template<typename T, typename VP>
      bool operator<(intrusive_ptr< T, VP > const &,
                     intrusive_ptr< T, VP > const &);
    template<typename T, typename VP>
      void swap(intrusive_ptr< T, VP > &, intrusive_ptr< T, VP > &);
    template<typename E, typename T, typename Y, typename VP>
      std::basic_ostream< E, T > &
      operator<<(std::basic_ostream< E, T > & os,
                 intrusive_ptr< Y, VP > const & p);
    template<typename T, typename VP>
      boost::interprocess::intrusive_ptr< T, VP >::pointer
      get_pointer(intrusive_ptr< T, VP >);
  }
}
```

Class template intrusive_ptr

boost::interprocess::intrusive_ptr

Synopsis

```
// In header: <boost/interprocess/smart_ptr/intrusive_ptr.hpp>

template<typename T, typename VoidPointer>
class intrusive_ptr {
public:
  // types
  typedef boost::pointer_to_other< VoidPointer, T >::type pointer;        // Provides the type ↵
of the internal stored pointer.
  typedef T                                            element_type;  // Provides the type ↵
of the stored pointer.

  // construct/copy/destruct
  intrusive_ptr();
  intrusive_ptr(const pointer &, bool = true);
  intrusive_ptr(intrusive_ptr const &);
  template<typename U> intrusive_ptr(intrusive_ptr< U, VP > const &);
  intrusive_ptr& operator=(intrusive_ptr const &);
  template<typename U>
    intrusive_ptr& operator=(intrusive_ptr< U, VP > const &);
  intrusive_ptr& operator=(pointer);
  ~intrusive_ptr();

  // public member functions
  pointer & get();
  const pointer & get() const;
  T & operator*() const;
  const pointer & operator->() const;
  pointer & operator->();
  operator unspecified_bool_type() const;
  bool operator!() const;
  void swap(intrusive_ptr &);
};
```

Description

The intrusive_ptr class template stores a pointer to an object with an embedded reference count. intrusive_ptr is parameterized on T (the type of the object pointed to) and VoidPointer(a void pointer type that defines the type of pointer that intrusive_ptr will store). intrusive_ptr<T, void *> defines a class with a T* member whereas intrusive_ptr<T, offset_ptr<void> > defines a class with a offset_ptr<T> member. Relies on unqualified calls to:

void intrusive_ptr_add_ref(T * p); void intrusive_ptr_release(T * p);

with (p != 0)

The object is responsible for destroying itself.

`intrusive_ptr` public construct/copy/destruct

1.
```
intrusive_ptr();
```

Constructor. Initializes internal pointer to 0. Does not throw

2.
```
intrusive_ptr(const pointer & p, bool add_ref = true);
```

Constructor. Copies pointer and if "p" is not zero and "add_ref" is true calls intrusive_ptr_add_ref(get_pointer(p)). Does not throw

3.
```
intrusive_ptr(intrusive_ptr const & rhs);
```

Copy constructor. Copies the internal pointer and if "p" is not zero calls intrusive_ptr_add_ref(get_pointer(p)). Does not throw

4.
```
template<typename U> intrusive_ptr(intrusive_ptr< U, VP > const & rhs);
```

Constructor from related. Copies the internal pointer and if "p" is not zero calls intrusive_ptr_add_ref(get_pointer(p)). Does not throw

5.
```
intrusive_ptr& operator=(intrusive_ptr const & rhs);
```

Assignment operator. Equivalent to intrusive_ptr(r).swap(*this). Does not throw

6.
```
template<typename U>
   intrusive_ptr& operator=(intrusive_ptr< U, VP > const & rhs);
```

Assignment from related. Equivalent to intrusive_ptr(r).swap(*this). Does not throw

7.
```
intrusive_ptr& operator=(pointer rhs);
```

Assignment from pointer. Equivalent to intrusive_ptr(r).swap(*this). Does not throw

8.
```
~intrusive_ptr();
```

Destructor. If internal pointer is not 0, calls intrusive_ptr_release(get_pointer(m_ptr)). Does not throw

`intrusive_ptr` public member functions

1.
```
pointer & get();
```

Returns a reference to the internal pointer. Does not throw

2.
```
const pointer & get() const;
```

Returns a reference to the internal pointer. Does not throw

3.
```
T & operator*() const;
```

Returns *get(). Does not throw

4.
```
const pointer & operator->() const;
```

Returns *get(). Does not throw

5.
```
pointer & operator->();
```

Returns get(). Does not throw

6.
```
operator unspecified_bool_type() const;
```

Conversion to boolean. Does not throw

7.
```
bool operator!() const;
```

Not operator. Does not throw

8.
```
void swap(intrusive_ptr & rhs);
```

Exchanges the contents of the two smart pointers. Does not throw

```
operator unspecified_bool_type() const;
```

Function template operator==

boost::interprocess::operator==

Synopsis

```
// In header: <boost/interprocess/smart_ptr/intrusive_ptr.hpp>


template<typename T, typename U, typename VP>
  bool operator==(intrusive_ptr< T, VP > const & a,
                  intrusive_ptr< U, VP > const & b);
```

Description

Returns a.get() == b.get(). Does not throw

Function template operator!=

boost::interprocess::operator!=

Synopsis

```
// In header: <boost/interprocess/smart_ptr/intrusive_ptr.hpp>


template<typename T, typename U, typename VP>
  bool operator!=(intrusive_ptr< T, VP > const & a,
                  intrusive_ptr< U, VP > const & b);
```

Description

Returns a.get() != b.get(). Does not throw

Function template operator==

boost::interprocess::operator==

Synopsis

```
// In header: <boost/interprocess/smart_ptr/intrusive_ptr.hpp>


template<typename T, typename VP>
  bool operator==(intrusive_ptr< T, VP > const & a,
                  const typename intrusive_ptr< T, VP >::pointer & b);
```

Description

Returns a.get() == b. Does not throw

Function template operator!=

boost::interprocess::operator!=

Synopsis

```
// In header: <boost/interprocess/smart_ptr/intrusive_ptr.hpp>


template<typename T, typename VP>
  bool operator!=(intrusive_ptr< T, VP > const & a,
                  const typename intrusive_ptr< T, VP >::pointer & b);
```

Description

Returns a.get() != b. Does not throw

Function template operator==

boost::interprocess::operator==

Synopsis

```
// In header: <boost/interprocess/smart_ptr/intrusive_ptr.hpp>


template<typename T, typename VP>
  bool operator==(const typename intrusive_ptr< T, VP >::pointer & a,
                  intrusive_ptr< T, VP > const & b);
```

Description

Returns a == b.get(). Does not throw

Function template operator!=

boost::interprocess::operator!=

Synopsis

```
// In header: <boost/interprocess/smart_ptr/intrusive_ptr.hpp>


template<typename T, typename VP>
  bool operator!=(const typename intrusive_ptr< T, VP >::pointer & a,
                  intrusive_ptr< T, VP > const & b);
```

Description

Returns a != b.get(). Does not throw

Function template operator<

boost::interprocess::operator<

Synopsis

```
// In header: <boost/interprocess/smart_ptr/intrusive_ptr.hpp>


template<typename T, typename VP>
  bool operator<(intrusive_ptr< T, VP > const & a,
                 intrusive_ptr< T, VP > const & b);
```

Description

Returns a.get() < b.get(). Does not throw

Function template swap

boost::interprocess::swap

Synopsis

```
// In header: <boost/interprocess/smart_ptr/intrusive_ptr.hpp>


template<typename T, typename VP>
  void swap(intrusive_ptr< T, VP > & lhs, intrusive_ptr< T, VP > & rhs);
```

Description

Exchanges the contents of the two intrusive_ptrs. Does not throw

Function template get_pointer

boost::interprocess::get_pointer

Synopsis

```
// In header: <boost/interprocess/smart_ptr/intrusive_ptr.hpp>


template<typename T, typename VP>
  boost::interprocess::intrusive_ptr< T, VP >::pointer
  get_pointer(intrusive_ptr< T, VP > p);
```

Description

Returns p.get(). Does not throw

Header <boost/interprocess/smart_ptr/scoped_ptr.hpp>

Describes the smart pointer scoped_ptr

```
namespace boost {
  namespace interprocess {
    template<typename T, typename Deleter> class scoped_ptr;
    template<typename T, typename D>
      void swap(scoped_ptr< T, D > &, scoped_ptr< T, D > &);
    template<typename T, typename D>
      scoped_ptr< T, D >::pointer get_pointer(scoped_ptr< T, D > const &);
  }
}
```

Class template scoped_ptr

boost::interprocess::scoped_ptr

Synopsis

```
// In header: <boost/interprocess/smart_ptr/scoped_ptr.hpp>

template<typename T, typename Deleter>
class scoped_ptr {
public:
  // types
  typedef T                   element_type;
  typedef Deleter             deleter_type;
  typedef unspecified         pointer;
  typedef pointer this_type::* unspecified_bool_type;

  // construct/copy/destruct
  scoped_ptr(const pointer & = 0, const Deleter & = Deleter());
  ~scoped_ptr();

  // public member functions
  void reset(const pointer & = 0);
  void reset(const pointer &, const Deleter &);
  pointer release();
  reference operator*() const;
  pointer & operator->();
  const pointer & operator->() const;
  pointer & get();
  const pointer & get() const;
  operator unspecified_bool_type() const;
  bool operator!() const;
  void swap(scoped_ptr &);
};
```

Description

scoped_ptr stores a pointer to a dynamically allocated object. The object pointed to is guaranteed to be deleted, either on destruction of the scoped_ptr, or via an explicit reset. The user can avoid this deletion using release(). scoped_ptr is parameterized on T (the type of the object pointed to) and Deleter (the functor to be executed to delete the internal pointer). The internal pointer will be of the same pointer type as typename Deleter::pointer type (that is, if typename Deleter::pointer is offset_ptr<void>, the internal pointer will be offset_ptr<T>).

`scoped_ptr` public construct/copy/destruct

1.
```
scoped_ptr(const pointer & p = 0, const Deleter & d = Deleter());
```

Provides the type of the internal stored pointer.

Constructs a scoped_ptr, storing a copy of p(which can be 0) and d. Does not throw.

2.
```
~scoped_ptr();
```

If the stored pointer is not 0, destroys the object pointed to by the stored pointer. calling the operator() of the stored deleter. Never throws

`scoped_ptr` public member functions

1.
```
void reset(const pointer & p = 0);
```

Deletes the object pointed to by the stored pointer and then stores a copy of p. Never throws

2.
```
void reset(const pointer & p, const Deleter & d);
```

Deletes the object pointed to by the stored pointer and then stores a copy of p and a copy of d.

3.
```
pointer release();
```

Assigns internal pointer as 0 and returns previous pointer. This will avoid deletion on destructor

4.
```
reference operator*() const;
```

Returns a reference to the object pointed to by the stored pointer. Never throws.

5.
```
pointer & operator->();
```

Returns the internal stored pointer. Never throws.

6.
```
const pointer & operator->() const;
```

Returns the internal stored pointer. Never throws.

7.
```
pointer & get();
```

Returns the stored pointer. Never throws.

8.
```
const pointer & get() const;
```

Returns the stored pointer. Never throws.

9.
```
operator unspecified_bool_type() const;
```

Conversion to bool Never throws

10.
```
bool operator!() const;
```

Returns true if the stored pointer is 0. Never throws.

11.
```
void swap(scoped_ptr & b);
```

Exchanges the internal pointer and deleter with other scoped_ptr Never throws.

Function template swap

boost::interprocess::swap

Synopsis

```
// In header: <boost/interprocess/smart_ptr/scoped_ptr.hpp>


template<typename T, typename D>
  void swap(scoped_ptr< T, D > & a, scoped_ptr< T, D > & b);
```

Description

Exchanges the internal pointer and deleter with other scoped_ptr Never throws.

Function template get_pointer

boost::interprocess::get_pointer

Synopsis

```
// In header: <boost/interprocess/smart_ptr/scoped_ptr.hpp>


template<typename T, typename D>
  scoped_ptr< T, D >::pointer get_pointer(scoped_ptr< T, D > const & p);
```

Description

Returns a copy of the stored pointer Never throws

Header <boost/interprocess/smart_ptr/shared_ptr.hpp>

Describes the smart pointer shared_ptr

```
namespace boost {
  namespace interprocess {
    template<typename T, typename VoidAllocator, typename Deleter>
      class shared_ptr;

    template<typename T, typename ManagedMemory> struct managed_shared_ptr;
    template<typename T, typename VoidAllocator, typename Deleter, typename U,
             typename VoidAllocator2, typename Deleter2>
      bool operator==(shared_ptr< T, VoidAllocator, Deleter > const & a,
                      shared_ptr< U, VoidAllocator2, Deleter2 > const & b);
    template<typename T, typename VoidAllocator, typename Deleter, typename U,
             typename VoidAllocator2, typename Deleter2>
      bool operator!=(shared_ptr< T, VoidAllocator, Deleter > const & a,
                      shared_ptr< U, VoidAllocator2, Deleter2 > const & b);
    template<typename T, typename VoidAllocator, typename Deleter, typename U,
             typename VoidAllocator2, typename Deleter2>
      bool operator<(shared_ptr< T, VoidAllocator, Deleter > const & a,
                     shared_ptr< U, VoidAllocator2, Deleter2 > const & b);
    template<typename T, typename VoidAllocator, typename Deleter>
      void swap(shared_ptr< T, VoidAllocator, Deleter > & a,
                shared_ptr< T, VoidAllocator, Deleter > & b);
    template<typename T, typename VoidAllocator, typename Deleter, typename U>
      shared_ptr< T, VoidAllocator, Deleter >
      static_pointer_cast(shared_ptr< U, VoidAllocator, Deleter > const & r);
    template<typename T, typename VoidAllocator, typename Deleter, typename U>
      shared_ptr< T, VoidAllocator, Deleter >
      const_pointer_cast(shared_ptr< U, VoidAllocator, Deleter > const & r);
    template<typename T, typename VoidAllocator, typename Deleter, typename U>
      shared_ptr< T, VoidAllocator, Deleter >
      dynamic_pointer_cast(shared_ptr< U, VoidAllocator, Deleter > const & r);
    template<typename T, typename VoidAllocator, typename Deleter>
      T * get_pointer(shared_ptr< T, VoidAllocator, Deleter > const & p);
    template<typename E, typename T, typename Y, typename VoidAllocator,
             typename Deleter>
      std::basic_ostream< E, T > &
      operator<<(std::basic_ostream< E, T > & os,
                 shared_ptr< Y, VoidAllocator, Deleter > const & p);
    template<typename T, typename ManagedMemory>
      managed_shared_ptr< T, ManagedMemory >::type
      make_managed_shared_ptr(T *, ManagedMemory &);
  }
}
```

Class template shared_ptr

boost::interprocess::shared_ptr

Synopsis

```cpp
// In header: <boost/interprocess/smart_ptr/shared_ptr.hpp>

template<typename T, typename VoidAllocator, typename Deleter>
class shared_ptr {
public:
  // types
  typedef T                                                              ↵
 element_type;
  typedef T                                                              ↵
 value_type;
  typedef boost::pointer_to_other< typename VoidAllocator::pointer, T >::type   ↵
 pointer;
  typedef unspecified                                                    ↵
 reference;
  typedef unspecified                                                    ↵
 const_reference;
  typedef boost::pointer_to_other< typename VoidAllocator::pointer, const Deleter >::type   ↵
 const_deleter_pointer;
  typedef boost::pointer_to_other< typename VoidAllocator::pointer, const VoidAllocat↵
or >::type const_allocator_pointer;

  // construct/copy/destruct
  shared_ptr();
  shared_ptr(const pointer &, const VoidAllocator & = VoidAllocator(),
             const Deleter & = Deleter());
  shared_ptr(const shared_ptr &, const pointer &);
  template<typename Y>
    shared_ptr(shared_ptr< Y, VoidAllocator, Deleter > const &);
  template<typename Y>
    shared_ptr(weak_ptr< Y, VoidAllocator, Deleter > const &);
  shared_ptr(shared_ptr &&);
  template<typename Y>
    shared_ptr& operator=(shared_ptr< Y, VoidAllocator, Deleter > const &);
  shared_ptr& operator=(BOOST_INTERPROCESS_COPY_ASSIGN_REF(shared_ptr));
  shared_ptr& operator=(shared_ptr &&);

  // public member functions
  void reset();
  template<typename Pointer>
    void reset(const Pointer &, const VoidAllocator & = VoidAllocator(),
               const Deleter & = Deleter());
  template<typename Y>
    void reset(shared_ptr< Y, VoidAllocator, Deleter > const &,
               const pointer &);
  reference operator*() const;
  pointer operator->() const;
  pointer get() const;
  bool operator!() const;
  bool unique() const;
  long use_count() const;
  void swap(shared_ptr< T, VoidAllocator, Deleter > &);
};
```

Description

shared_ptr stores a pointer to a dynamically allocated object. The object pointed to is guaranteed to be deleted when the last shared_ptr pointing to it is destroyed or reset.

shared_ptr is parameterized on T (the type of the object pointed to), VoidAllocator (the void allocator to be used to allocate the auxiliary data) and Deleter (the deleter whose operator() will be used to delete the object).

The internal pointer will be of the same pointer type as typename VoidAllocator::pointer type (that is, if typename VoidAllocator::pointer is offset_ptr<void>, the internal pointer will be offset_ptr<T>).

Because the implementation uses reference counting, cycles of shared_ptr instances will not be reclaimed. For example, if main() holds a shared_ptr to A, which directly or indirectly holds a shared_ptr back to A, A's use count will be 2. Destruction of the original shared_ptr will leave A dangling with a use count of 1. Use weak_ptr to "break cycles."

`shared_ptr` public construct/copy/destruct

1.
```
shared_ptr();
```

Constructs an empty shared_ptr. Use_count() == 0 && get()== 0.

2.
```
shared_ptr(const pointer & p, const VoidAllocator & a = VoidAllocator(),
           const Deleter & d = Deleter());
```

Constructs a shared_ptr that owns the pointer p. Auxiliary data will be allocated with a copy of a and the object will be deleted with a copy of d. Requirements: Deleter and A's copy constructor must not throw.

3.
```
shared_ptr(const shared_ptr & other, const pointer & p);
```

Constructs a shared_ptr that shares ownership with r and stores p. Postconditions: get() == p && use_count() == r.use_count(). Throws: nothing.

4.
```
template<typename Y>
  shared_ptr(shared_ptr< Y, VoidAllocator, Deleter > const & r);
```

If r is empty, constructs an empty shared_ptr. Otherwise, constructs a shared_ptr that shares ownership with r. Never throws.

5.
```
template<typename Y>
  shared_ptr(weak_ptr< Y, VoidAllocator, Deleter > const & r);
```

Constructs a shared_ptr that shares ownership with r and stores a copy of the pointer stored in r.

6.
```
shared_ptr(shared_ptr && other);
```

Move-Constructs a shared_ptr that takes ownership of other resource and other is put in default-constructed state. Throws: nothing.

7.
```
template<typename Y>
  shared_ptr& operator=(shared_ptr< Y, VoidAllocator, Deleter > const & r);
```

Equivalent to shared_ptr(r).swap(*this). Never throws

8.
```
shared_ptr& operator=(BOOST_INTERPROCESS_COPY_ASSIGN_REF(shared_ptr) r);
```

Equivalent to shared_ptr(r).swap(*this). Never throws

9.
```
shared_ptr& operator=(shared_ptr && other);
```

Move-assignment. Equivalent to shared_ptr(other).swap(*this). Never throws

`shared_ptr` public member functions

1.
```
void reset();
```

This is equivalent to: this_type().swap(*this);

2.
```
template<typename Pointer>
  void reset(const Pointer & p, const VoidAllocator & a = VoidAllocator(),
             const Deleter & d = Deleter());
```

This is equivalent to: this_type(p, a, d).swap(*this);

3.
```
template<typename Y>
  void reset(shared_ptr< Y, VoidAllocator, Deleter > const & r,
             const pointer & p);
```

4.
```
reference operator*() const;
```

Returns a reference to the pointed type

5.
```
pointer operator->() const;
```

Returns the pointer pointing to the owned object

6.
```
pointer get() const;
```

Returns the pointer pointing to the owned object

7.
```
bool operator!() const;
```

Not operator. Returns true if this->get() != 0, false otherwise

8.
```
bool unique() const;
```

Returns use_count() == 1. unique() might be faster than use_count()

9.
```
long use_count() const;
```

Returns the number of shared_ptr objects, *this included, that share ownership with *this, or an unspecified nonnegative value when *this is empty. use_count() is not necessarily efficient. Use only for debugging and testing purposes, not for production code.

10.
```
void swap(shared_ptr< T, VoidAllocator, Deleter > & other);
```

Exchanges the contents of the two smart pointers.

Struct template managed_shared_ptr

boost::interprocess::managed_shared_ptr

Synopsis

```
// In header: <boost/interprocess/smart_ptr/shared_ptr.hpp>

template<typename T, typename ManagedMemory>
struct managed_shared_ptr {
  // types
  typedef ManagedMemory::template allocator< void >::type void_allocator;
  typedef ManagedMemory::template deleter< T >::type       deleter;
  typedef shared_ptr< T, void_allocator, deleter >         type;
};
```

Description

Returns the type of a shared pointer of type T with the allocator boost::interprocess::allocator allocator and boost::interprocess::deleter deleter that can be constructed in the given managed segment type.

Function template make_managed_shared_ptr

boost::interprocess::make_managed_shared_ptr

Synopsis

```
// In header: <boost/interprocess/smart_ptr/shared_ptr.hpp>


template<typename T, typename ManagedMemory>
  managed_shared_ptr< T, ManagedMemory >::type
  make_managed_shared_ptr(T * constructed_object,
                          ManagedMemory & managed_memory);
```

Description

Returns an instance of a shared pointer constructed with the default allocator and deleter from a pointer of type T that has been allocated in the passed managed segment

Header <boost/interprocess/smart_ptr/unique_ptr.hpp>

Describes the smart pointer unique_ptr

```
namespace boost {
  namespace interprocess {
    template<typename T, typename D> class unique_ptr;

    template<typename T, typename ManagedMemory> struct managed_unique_ptr;
    template<typename T, typename D>
      void swap(unique_ptr< T, D > & x, unique_ptr< T, D > & y);
    template<typename T1, typename D1, typename T2, typename D2>
      bool operator==(const unique_ptr< T1, D1 > & x,
                      const unique_ptr< T2, D2 > & y);
    template<typename T1, typename D1, typename T2, typename D2>
      bool operator!=(const unique_ptr< T1, D1 > & x,
                      const unique_ptr< T2, D2 > & y);
    template<typename T1, typename D1, typename T2, typename D2>
      bool operator<(const unique_ptr< T1, D1 > & x,
                     const unique_ptr< T2, D2 > & y);
    template<typename T1, typename D1, typename T2, typename D2>
      bool operator<=(const unique_ptr< T1, D1 > & x,
                      const unique_ptr< T2, D2 > & y);
    template<typename T1, typename D1, typename T2, typename D2>
      bool operator>(const unique_ptr< T1, D1 > & x,
                     const unique_ptr< T2, D2 > & y);
    template<typename T1, typename D1, typename T2, typename D2>
      bool operator>=(const unique_ptr< T1, D1 > & x,
                      const unique_ptr< T2, D2 > & y);
    template<typename T, typename ManagedMemory>
      managed_unique_ptr< T, ManagedMemory >::type
      make_managed_unique_ptr(T *, ManagedMemory &);
  }
}
```

Class template unique_ptr

boost::interprocess::unique_ptr

Synopsis

```
// In header: <boost/interprocess/smart_ptr/unique_ptr.hpp>

template<typename T, typename D>
class unique_ptr {
public:
  // types
  typedef T           element_type;
  typedef D           deleter_type;
  typedef unspecified pointer;

  // construct/copy/destruct
  unique_ptr();
  unique_ptr(pointer);
  unique_ptr(pointer, unspecified);
  unique_ptr(unique_ptr &&);
  template<typename U, typename E>
    unique_ptr(unique_ptr &&, unspecified = nat());
  unique_ptr& operator=(unique_ptr &&);
  template<typename U, typename E> unique_ptr& operator=(unique_ptr &&);
  unique_ptr& operator=(int nat::*);
  ~unique_ptr();

  // public member functions
  unspecified operator*() const;
  pointer operator->() const;
  pointer get() const;
  deleter_reference get_deleter();
  deleter_const_reference get_deleter() const;
  operator int nat::*() const;
  pointer release();
  void reset(pointer = 0);
  void swap(unique_ptr &);
};
```

Description

Template unique_ptr stores a pointer to an object and deletes that object using the associated deleter when it is itself destroyed (such as when leaving block scope.

The unique_ptr provides a semantics of strict ownership. A unique_ptr owns the object it holds a pointer to.

A unique_ptr is not CopyConstructible, nor CopyAssignable, however it is MoveConstructible and Move-Assignable.

The uses of unique_ptr include providing exception safety for dynamically allocated memory, passing ownership of dynamically allocated memory to a function, and returning dynamically allocated memory from a function

A client-supplied template argument D must be a function pointer or functor for which, given a value d of type D and a pointer ptr to a type T*, the expression d(ptr) is valid and has the effect of deallocating the pointer as appropriate for that deleter. D may also be an lvalue-reference to a deleter.

If the deleter D maintains state, it is intended that this state stay with the associated pointer as ownership is transferred from unique_ptr to unique_ptr. The deleter state need never be copied, only moved or swapped as pointer ownership is moved around. That is, the deleter need only be MoveConstructible, MoveAssignable, and Swappable, and need not be CopyConstructible (unless copied into the unique_ptr) nor CopyAssignable.

`unique_ptr` public construct/copy/destruct

1.
```
unique_ptr();
```

Requires: D must be default constructible, and that construction must not throw an exception. D must not be a reference type.

Effects: Constructs a unique_ptr which owns nothing.

Postconditions: get() == 0. get_deleter() returns a reference to a default constructed deleter D.

Throws: nothing.

2.
```
unique_ptr(pointer p);
```

Requires: The expression D()(p) must be well formed. The default constructor of D must not throw an exception.

D must not be a reference type.

Effects: Constructs a unique_ptr which owns p.

Postconditions: get() == p. get_deleter() returns a reference to a default constructed deleter D.

Throws: nothing.

3.
```
unique_ptr(pointer p, unspecified d);
```

Requires: The expression d(p) must be well formed.

Postconditions: get() == p. get_deleter() returns a reference to the internally stored deleter. If D is a reference type then get_deleter() returns a reference to the lvalue d.

Throws: nothing.

4.
```
unique_ptr(unique_ptr && u);
```

Requires: If the deleter is not a reference type, construction of the deleter D from an lvalue D must not throw an exception.

Effects: Constructs a unique_ptr which owns the pointer which u owns (if any). If the deleter is not a reference type, it is move constructed from u's deleter, otherwise the reference is copy constructed from u's deleter.

After the construction, u no longer owns a pointer. [Note: The deleter constructor can be implemented with boost::interprocess::forward<D>. -end note]

Postconditions: get() == value u.get() had before the construction. get_deleter() returns a reference to the internally stored deleter which was constructed from u.get_deleter(). If D is a reference type then get_- deleter() and u.get_deleter() both reference the same lvalue deleter.

Throws: nothing.

5.
```
template<typename U, typename E>
   unique_ptr(unique_ptr && u, unspecified = nat());
```

Requires: If D is not a reference type, construction of the deleter D from an rvalue of type E must be well formed and not throw an exception. If D is a reference type, then E must be the same type as D (diagnostic required). unique_ptr<U, E>::pointer must be implicitly convertible to pointer.

Effects: Constructs a unique_ptr which owns the pointer which u owns (if any). If the deleter is not a reference type, it is move constructed from u's deleter, otherwise the reference is copy constructed from u's deleter.

After the construction, u no longer owns a pointer.

postconditions get() == value u.get() had before the construction, modulo any required offset adjustments resulting from the cast from U* to T*. get_deleter() returns a reference to the internally stored deleter which was constructed from u.get_deleter().

Throws: nothing.

6.
```
unique_ptr& operator=(unique_ptr && u);
```

Requires: Assignment of the deleter D from an rvalue D must not throw an exception.

Effects: reset(u.release()) followed by a move assignment from u's deleter to this deleter.

Postconditions: This unique_ptr now owns the pointer which u owned, and u no longer owns it.

Returns: *this.

Throws: nothing.

7.
```
template<typename U, typename E> unique_ptr& operator=(unique_ptr && u);
```

Requires: Assignment of the deleter D from an rvalue D must not throw an exception. U* must be implicitly convertible to T*.

Effects: reset(u.release()) followed by a move assignment from u's deleter to this deleter. If either D or E is a reference type, then the referenced lvalue deleter participates in the move assignment.

Postconditions: This unique_ptr now owns the pointer which u owned, and u no longer owns it.

Returns: *this.

Throws: nothing.

8.
```
unique_ptr& operator=(int nat::*);
```

Assigns from the literal 0 or NULL.

Effects: reset().

Postcondition: get() == 0

Returns: *this.

Throws: nothing.

9.
```
~unique_ptr();
```

Effects: If get() == 0 there are no effects. Otherwise get_deleter()(get()).

Throws: nothing.

`unique_ptr` public member functions

1.
```
unspecified operator*() const;
```

Requires: get() != 0. Returns: *get(). Throws: nothing.

2.
```
pointer operator->() const;
```

Requires: get() != 0. Returns: get(). Throws: nothing.

3.
```
pointer get() const;
```

Returns: The stored pointer. Throws: nothing.

4.
```
deleter_reference get_deleter();
```

Returns: A reference to the stored deleter.

Throws: nothing.

5.
```
deleter_const_reference get_deleter() const;
```

Returns: A const reference to the stored deleter.

Throws: nothing.

6.
```
operator int nat::*() const;
```

Returns: An unspecified value that, when used in boolean contexts, is equivalent to get() != 0.

Throws: nothing.

7.
```
pointer release();
```

Postcondition: get() == 0.

Returns: The value get() had at the start of the call to release.

Throws: nothing.

8.
```
void reset(pointer p = 0);
```

Effects: If p == get() there are no effects. Otherwise get_deleter()(get()).

Postconditions: get() == p.

Throws: nothing.

9.
```
void swap(unique_ptr & u);
```

Requires: The deleter D is Swappable and will not throw an exception under swap.

Effects: The stored pointers of this and u are exchanged. The stored deleters are swapped (unqualified). Throws: nothing.

Struct template managed_unique_ptr

boost::interprocess::managed_unique_ptr

Synopsis

```
// In header: <boost/interprocess/smart_ptr/unique_ptr.hpp>

template<typename T, typename ManagedMemory>
struct managed_unique_ptr {
  // types
  typedef unique_ptr< T, typename ManagedMemory::template deleter< T >::type > type;
};
```

Description

Returns the type of a unique pointer of type T with boost::interprocess::deleter deleter that can be constructed in the given managed segment type.

Function template make_managed_unique_ptr

boost::interprocess::make_managed_unique_ptr

Synopsis

```
// In header: <boost/interprocess/smart_ptr/unique_ptr.hpp>


template<typename T, typename ManagedMemory>
  managed_unique_ptr< T, ManagedMemory >::type
  make_managed_unique_ptr(T * constructed_object,
                          ManagedMemory & managed_memory);
```

Description

Returns an instance of a unique pointer constructed with boost::interproces::deleter from a pointer of type T that has been allocated in the passed managed segment

Header <boost/interprocess/smart_ptr/weak_ptr.hpp>

Describes the smart pointer weak_ptr.

```
namespace boost {
  namespace interprocess {
    template<typename T, typename A, typename D> class weak_ptr;

    template<typename T, typename ManagedMemory> struct managed_weak_ptr;
    template<typename T, typename A, typename D, typename U, typename A2,
             typename D2>
      bool operator<(weak_ptr< T, A, D > const & a,
                     weak_ptr< U, A2, D2 > const & b);
    template<typename T, typename A, typename D>
      void swap(weak_ptr< T, A, D > & a, weak_ptr< T, A, D > & b);
    template<typename T, typename ManagedMemory>
      managed_weak_ptr< T, ManagedMemory >::type
      make_managed_weak_ptr(T *, ManagedMemory &);
  }
}
```

Class template weak_ptr

boost::interprocess::weak_ptr

Synopsis

```cpp
// In header: <boost/interprocess/smart_ptr/weak_ptr.hpp>

template<typename T, typename A, typename D>
class weak_ptr {
public:
  // types
  typedef T element_type;
  typedef T value_type;

  // construct/copy/destruct
  weak_ptr();
  template<typename Y> weak_ptr(weak_ptr< Y, A, D > const &);
  template<typename Y> weak_ptr(shared_ptr< Y, A, D > const &);
  template<typename Y> weak_ptr& operator=(weak_ptr< Y, A, D > const &);
  template<typename Y> weak_ptr& operator=(shared_ptr< Y, A, D > const &);

  // public member functions
  shared_ptr< T, A, D > lock() const;
  long use_count() const;
  bool expired() const;
  void reset();
  void swap(this_type &);
};
```

Description

The weak_ptr class template stores a "weak reference" to an object that's already managed by a shared_ptr. To access the object, a weak_ptr can be converted to a shared_ptr using the shared_ptr constructor or the member function lock. When the last shared_ptr to the object goes away and the object is deleted, the attempt to obtain a shared_ptr from the weak_ptr instances that refer to the deleted object will fail: the constructor will throw an exception of type bad_weak_ptr, and weak_ptr::lock will return an empty shared_ptr.

Every weak_ptr meets the CopyConstructible and Assignable requirements of the C++ Standard Library, and so can be used in standard library containers. Comparison operators are supplied so that weak_ptr works with the standard library's associative containers.

weak_ptr operations never throw exceptions.

The class template is parameterized on T, the type of the object pointed to.

weak_ptr public construct/copy/destruct

1.
```cpp
weak_ptr();
```

Effects: Constructs an empty weak_ptr. Postconditions: use_count() == 0.

2.
```cpp
template<typename Y> weak_ptr(weak_ptr< Y, A, D > const & r);
```

Effects: If r is empty, constructs an empty weak_ptr; otherwise, constructs a weak_ptr that shares ownership with r as if by storing a copy of the pointer stored in r.

Postconditions: use_count() == r.use_count().

Throws: nothing.

3.
```
template<typename Y> weak_ptr(shared_ptr< Y, A, D > const & r);
```

Effects: If r is empty, constructs an empty weak_ptr; otherwise, constructs a weak_ptr that shares ownership with r as if by storing a copy of the pointer stored in r.

Postconditions: use_count() == r.use_count().

Throws: nothing.

4.
```
template<typename Y> weak_ptr& operator=(weak_ptr< Y, A, D > const & r);
```

Effects: Equivalent to weak_ptr(r).swap(*this).

Throws: nothing.

Notes: The implementation is free to meet the effects (and the implied guarantees) via different means, without creating a temporary.

5.
```
template<typename Y> weak_ptr& operator=(shared_ptr< Y, A, D > const & r);
```

Effects: Equivalent to weak_ptr(r).swap(*this).

Throws: nothing.

Notes: The implementation is free to meet the effects (and the implied guarantees) via different means, without creating a temporary.

weak_ptr public member functions

1.
```
shared_ptr< T, A, D > lock() const;
```

Returns: expired()? shared_ptr<T>(): shared_ptr<T>(*this).

Throws: nothing.

2.
```
long use_count() const;
```

Returns: 0 if *this is empty; otherwise, the number of shared_ptr objects that share ownership with *this.

Throws: nothing.

Notes: use_count() is not necessarily efficient. Use only for debugging and testing purposes, not for production code.

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

Returns: Returns: use_count() == 0.

Throws: nothing.

Notes: expired() may be faster than use_count().

4.
```
void reset();
```

Effects: Equivalent to: weak_ptr().swap(*this).

5.
```cpp
void swap(this_type & other);
```

Effects: Exchanges the contents of the two smart pointers.

Throws: nothing.

5.
```cpp
void swap(this_type & other);
```

Struct template managed_weak_ptr

boost::interprocess::managed_weak_ptr

Synopsis

```
// In header: <boost/interprocess/smart_ptr/weak_ptr.hpp>

template<typename T, typename ManagedMemory>
struct managed_weak_ptr {
  // types
  typedef weak_ptr< T, typename ManagedMemory::template allocator< void >::type, typename Man↵
agedMemory::template deleter< T >::type > type;
};
```

Description

Returns the type of a weak pointer of type T with the allocator boost::interprocess::allocator allocator and boost::interprocess::deleter deleter that can be constructed in the given managed segment type.

Function template make_managed_weak_ptr

boost::interprocess::make_managed_weak_ptr

Synopsis

```
// In header: <boost/interprocess/smart_ptr/weak_ptr.hpp>


template<typename T, typename ManagedMemory>
  managed_weak_ptr< T, ManagedMemory >::type
  make_managed_weak_ptr(T * constructed_object,
                        ManagedMemory & managed_memory);
```

Description

Returns an instance of a weak pointer constructed with the default allocator and deleter from a pointer of type T that has been allocated in the passed managed segment

Header <boost/interprocess/streams/bufferstream.hpp>

This file defines basic_bufferbuf, basic_ibufferstream, basic_obufferstream, and basic_bufferstream classes. These classes represent streamsbufs and streams whose sources or destinations are fixed size character buffers.

```
namespace boost {
  namespace interprocess {
    template<typename CharT, typename CharTraits> class basic_bufferbuf;
    template<typename CharT, typename CharTraits> class basic_ibufferstream;
    template<typename CharT, typename CharTraits> class basic_obufferstream;
    template<typename CharT, typename CharTraits> class basic_bufferstream;

    typedef basic_bufferbuf< char > bufferbuf;
    typedef basic_bufferstream< char > bufferstream;
    typedef basic_ibufferstream< char > ibufferstream;
    typedef basic_obufferstream< char > obufferstream;
    typedef basic_bufferbuf< wchar_t > wbufferbuf;
    typedef basic_bufferstream< wchar_t > wbufferstream;
    typedef basic_ibufferstream< wchar_t > wibufferstream;
    typedef basic_obufferstream< wchar_t > wobufferstream;
  }
}
```

Class template basic_bufferbuf

boost::interprocess::basic_bufferbuf

Synopsis

```
// In header: <boost/interprocess/streams/bufferstream.hpp>

template<typename CharT, typename CharTraits>
class basic_bufferbuf {
public:
  // types
  typedef CharT                                       char_type;
  typedef CharTraits::int_type                        int_type;
  typedef CharTraits::pos_type                        pos_type;
  typedef CharTraits::off_type                        off_type;
  typedef CharTraits                                  traits_type;
  typedef std::basic_streambuf< char_type, traits_type > base_t;

  // construct/copy/destruct
  basic_bufferbuf(std::ios_base::openmode = std::ios_base::in|std::ios_base::out);
  basic_bufferbuf(CharT *, std::size_t,
                  std::ios_base::openmode = std::ios_base::in|std::ios_base::out);
  ~basic_bufferbuf();

  // public member functions
  std::pair< CharT *, std::size_t > buffer() const;
  void buffer(CharT *, std::size_t);
};
```

Description

A streambuf class that controls the transmission of elements to and from a basic_xbufferstream. The elements are transmitted from a to a fixed size buffer

`basic_bufferbuf` public construct/copy/destruct

1.
```
basic_bufferbuf(std::ios_base::openmode mode = std::ios_base::in|std::ios_base::out);
```

Constructor. Does not throw.

2.
```
basic_bufferbuf(CharT * buffer, std::size_t length,
                std::ios_base::openmode mode = std::ios_base::in|std::ios_base::out);
```

Constructor. Assigns formatting buffer. Does not throw.

3.
```
~basic_bufferbuf();
```

`basic_bufferbuf` public member functions

1.
```
std::pair< CharT *, std::size_t > buffer() const;
```

Returns the pointer and size of the internal buffer. Does not throw.

2.
```
void buffer(CharT * buffer, std::size_t length);
```

Sets the underlying buffer to a new value Does not throw.

Class template basic_ibufferstream

boost::interprocess::basic_ibufferstream

Synopsis

```cpp
// In header: <boost/interprocess/streams/bufferstream.hpp>

template<typename CharT, typename CharTraits>
class basic_ibufferstream {
public:
  // types
  typedef std::basic_ios< CharT, CharTraits >::char_type        char_type;
  typedef std::basic_ios< char_type, CharTraits >::int_type     int_type;
  typedef std::basic_ios< char_type, CharTraits >::pos_type     pos_type;
  typedef std::basic_ios< char_type, CharTraits >::off_type     off_type;
  typedef std::basic_ios< char_type, CharTraits >::traits_type traits_type;

  // construct/copy/destruct
  basic_ibufferstream(std::ios_base::openmode = std::ios_base::in);
  basic_ibufferstream(const CharT *, std::size_t,
                      std::ios_base::openmode = std::ios_base::in);
  ~basic_ibufferstream();

  // public member functions
  basic_bufferbuf< CharT, CharTraits > * rdbuf() const;
  std::pair< const CharT *, std::size_t > buffer() const;
  void buffer(const CharT *, std::size_t);
};
```

Description

A basic_istream class that uses a fixed size character buffer as its formatting buffer.

basic_ibufferstream public construct/copy/destruct

1.
```cpp
basic_ibufferstream(std::ios_base::openmode mode = std::ios_base::in);
```

Constructor. Does not throw.

2.
```cpp
basic_ibufferstream(const CharT * buffer, std::size_t length,
                    std::ios_base::openmode mode = std::ios_base::in);
```

Constructor. Assigns formatting buffer. Does not throw.

3.
```cpp
~basic_ibufferstream();
```

basic_ibufferstream public member functions

1.
```cpp
basic_bufferbuf< CharT, CharTraits > * rdbuf() const;
```

Returns the address of the stored stream buffer.

2.
```cpp
std::pair< const CharT *, std::size_t > buffer() const;
```

Returns the pointer and size of the internal buffer. Does not throw.

3.
```
void buffer(const CharT * buffer, std::size_t length);
```

Sets the underlying buffer to a new value. Resets stream position. Does not throw.

Class template basic_obufferstream

boost::interprocess::basic_obufferstream

Synopsis

```
// In header: <boost/interprocess/streams/bufferstream.hpp>

template<typename CharT, typename CharTraits>
class basic_obufferstream {
public:
  // types
  typedef std::basic_ios< CharT, CharTraits >::char_type        char_type;
  typedef std::basic_ios< char_type, CharTraits >::int_type     int_type;
  typedef std::basic_ios< char_type, CharTraits >::pos_type     pos_type;
  typedef std::basic_ios< char_type, CharTraits >::off_type     off_type;
  typedef std::basic_ios< char_type, CharTraits >::traits_type traits_type;

  // construct/copy/destruct
  basic_obufferstream(std::ios_base::openmode = std::ios_base::out);
  basic_obufferstream(CharT *, std::size_t,
                      std::ios_base::openmode = std::ios_base::out);
  ~basic_obufferstream();

  // public member functions
  basic_bufferbuf< CharT, CharTraits > * rdbuf() const;
  std::pair< CharT *, std::size_t > buffer() const;
  void buffer(CharT *, std::size_t);
};
```

Description

A basic_ostream class that uses a fixed size character buffer as its formatting buffer.

`basic_obufferstream` public construct/copy/destruct

1.
```
basic_obufferstream(std::ios_base::openmode mode = std::ios_base::out);
```

   Constructor. Does not throw.

2.
```
basic_obufferstream(CharT * buffer, std::size_t length,
                    std::ios_base::openmode mode = std::ios_base::out);
```

   Constructor. Assigns formatting buffer. Does not throw.

3.
```
~basic_obufferstream();
```

`basic_obufferstream` public member functions

1.
```
basic_bufferbuf< CharT, CharTraits > * rdbuf() const;
```

   Returns the address of the stored stream buffer.

2.
```
std::pair< CharT *, std::size_t > buffer() const;
```

Returns the pointer and size of the internal buffer. Does not throw.

3.
```
void buffer(CharT * buffer, std::size_t length);
```

Sets the underlying buffer to a new value. Resets stream position. Does not throw.

Class template basic_bufferstream

boost::interprocess::basic_bufferstream

Synopsis

```
// In header: <boost/interprocess/streams/bufferstream.hpp>

template<typename CharT, typename CharTraits>
class basic_bufferstream {
public:
  // types
  typedef std::basic_ios< CharT, CharTraits >::char_type       char_type;
  typedef std::basic_ios< char_type, CharTraits >::int_type    int_type;
  typedef std::basic_ios< char_type, CharTraits >::pos_type    pos_type;
  typedef std::basic_ios< char_type, CharTraits >::off_type    off_type;
  typedef std::basic_ios< char_type, CharTraits >::traits_type traits_type;

  // construct/copy/destruct
  basic_bufferstream(std::ios_base::openmode = std::ios_base::in|std::ios_base::out);
  basic_bufferstream(CharT *, std::size_t,
                     std::ios_base::openmode = std::ios_base::in|std::ios_base::out);
  ~basic_bufferstream();

  // public member functions
  basic_bufferbuf< CharT, CharTraits > * rdbuf() const;
  std::pair< CharT *, std::size_t > buffer() const;
  void buffer(CharT *, std::size_t);
};
```

Description

A basic_iostream class that uses a fixed size character buffer as its formatting buffer.

`basic_bufferstream` public construct/copy/destruct

1.
```
basic_bufferstream(std::ios_base::openmode mode = std::ios_base::in|std::ios_base::out);
```

   Constructor. Does not throw.

2.
```
basic_bufferstream(CharT * buffer, std::size_t length,
                   std::ios_base::openmode mode = std::ios_base::in|std::ios_base::out);
```

   Constructor. Assigns formatting buffer. Does not throw.

3.
```
~basic_bufferstream();
```

`basic_bufferstream` public member functions

1.
```
basic_bufferbuf< CharT, CharTraits > * rdbuf() const;
```

   Returns the address of the stored stream buffer.

2.
```
std::pair< CharT *, std::size_t > buffer() const;
```

Returns the pointer and size of the internal buffer. Does not throw.

3.
```
void buffer(CharT * buffer, std::size_t length);
```

Sets the underlying buffer to a new value. Resets stream position. Does not throw.

Header <boost/interprocess/streams/vectorstream.hpp>

This file defines basic_vectorbuf, basic_ivectorstream, basic_ovectorstream, and basic_vectorstreamclasses. These classes represent streamsbufs and streams whose sources or destinations are STL-like vectors that can be swapped with external vectors to avoid unnecessary allocations/copies.

```
namespace boost {
  namespace interprocess {
    template<typename CharVector, typename CharTraits> class basic_vectorbuf;
    template<typename CharVector, typename CharTraits> class std;
    template<typename CharVector, typename CharTraits> class basic_vectorstream;
  }
}
```

Class template basic_vectorbuf

boost::interprocess::basic_vectorbuf

Synopsis

```
// In header: <boost/interprocess/streams/vectorstream.hpp>

template<typename CharVector, typename CharTraits>
class basic_vectorbuf {
public:
  // types
  typedef CharVector            vector_type;
  typedef CharVector::value_type char_type;
  typedef CharTraits::int_type  int_type;
  typedef CharTraits::pos_type  pos_type;
  typedef CharTraits::off_type  off_type;
  typedef CharTraits            traits_type;

  // construct/copy/destruct
  basic_vectorbuf(std::ios_base::openmode = std::ios_base::in|std::ios_base::out);
  template<typename VectorParameter>
    basic_vectorbuf(const VectorParameter &,
                    std::ios_base::openmode = std::ios_base::in|std::ios_base::out);
  ~basic_vectorbuf();

  // public member functions
  void swap_vector(vector_type &);
  const vector_type & vector() const;
  void reserve(typename vector_type::size_type);
  void clear();
};
```

Description

A streambuf class that controls the transmission of elements to and from a basic_ivectorstream, basic_ovectorstream or basic_vectorstream. It holds a character vector specified by CharVector template parameter as its formatting buffer. The vector must have contiguous storage, like std::vector, boost::interprocess::vector or boost::interprocess::basic_string

`basic_vectorbuf` public construct/copy/destruct

1.
```
basic_vectorbuf(std::ios_base::openmode mode = std::ios_base::in|std::ios_base::out);
```

Constructor. Throws if vector_type default constructor throws.

2.
```
template<typename VectorParameter>
  basic_vectorbuf(const VectorParameter & param,
                  std::ios_base::openmode mode = std::ios_base::in|std::ios_base::out);
```

Constructor. Throws if vector_type(const VectorParameter &param) throws.

3.
```
~basic_vectorbuf();
```

`basic_vectorbuf` public member functions

1.
```
void swap_vector(vector_type & vect);
```

Swaps the underlying vector with the passed vector. This function resets the read/write position in the stream. Does not throw.

2.
```
const vector_type & vector() const;
```

Returns a const reference to the internal vector. Does not throw.

3.
```
void reserve(typename vector_type::size_type size);
```

Preallocates memory from the internal vector. Resets the stream to the first position. Throws if the internals vector's memory allocation throws.

4.
```
void clear();
```

Calls clear() method of the internal vector. Resets the stream to the first position.

Class template std

boost::interprocess::std

Synopsis

```cpp
// In header: <boost/interprocess/streams/vectorstream.hpp>

template<typename CharVector, typename CharTraits>
class std {
public:
  // types
  typedef CharVector                                                     vector_type;
  typedef std::basic_ios< typename CharVector::value_type, CharTraits >::char_type char_type;
  typedef std::basic_ios< char_type, CharTraits >::int_type              int_type;
  typedef std::basic_ios< char_type, CharTraits >::pos_type              pos_type;
  typedef std::basic_ios< char_type, CharTraits >::off_type              off_type;
  typedef std::basic_ios< char_type, CharTraits >::traits_type           traits_type;
  typedef CharVector                                                     vector_type;
  typedef std::basic_ios< typename CharVector::value_type, CharTraits >::char_type char_type;
  typedef std::basic_ios< char_type, CharTraits >::int_type              int_type;
  typedef std::basic_ios< char_type, CharTraits >::pos_type              pos_type;
  typedef std::basic_ios< char_type, CharTraits >::off_type              off_type;
  typedef std::basic_ios< char_type, CharTraits >::traits_type           traits_type;

  // public member functions
   basic_ivectorstream(std::ios_base::openmode = std::ios_base::in);
  template<typename VectorParameter>
     basic_ivectorstream(const VectorParameter &,
                         std::ios_base::openmode = std::ios_base::in);
   ~basic_ivectorstream();
  basic_vectorbuf< CharVector, CharTraits > * rdbuf() const;
  void swap_vector(vector_type &);
  const vector_type & vector() const;
  void reserve(typename vector_type::size_type);
  void clear();
   basic_ovectorstream(std::ios_base::openmode = std::ios_base::out);
  template<typename VectorParameter>
     basic_ovectorstream(const VectorParameter &,
                         std::ios_base::openmode = std::ios_base::out);
   ~basic_ovectorstream();
  basic_vectorbuf< CharVector, CharTraits > * rdbuf() const;
  void swap_vector(vector_type &);
  const vector_type & vector() const;
  void reserve(typename vector_type::size_type);
};
```

Description

A basic_istream class that holds a character vector specified by CharVector template parameter as its formatting buffer. The vector must have contiguous storage, like std::vector, boost::interprocess::vector or boost::interprocess::basic_string

A basic_ostream class that holds a character vector specified by CharVector template parameter as its formatting buffer. The vector must have contiguous storage, like std::vector, boost::interprocess::vector or boost::interprocess::basic_string

`std` public member functions

1.
```cpp
basic_ivectorstream(std::ios_base::openmode mode = std::ios_base::in);
```

Constructor. Throws if vector_type default constructor throws.

2.
```
template<typename VectorParameter>
    basic_ivectorstream(const VectorParameter & param,
                        std::ios_base::openmode mode = std::ios_base::in);
```

Constructor. Throws if vector_type(const VectorParameter &param) throws.

3.
```
~basic_ivectorstream();
```

4.
```
basic_vectorbuf< CharVector, CharTraits > * rdbuf() const;
```

Returns the address of the stored stream buffer.

5.
```
void swap_vector(vector_type & vect);
```

Swaps the underlying vector with the passed vector. This function resets the read position in the stream. Does not throw.

6.
```
const vector_type & vector() const;
```

Returns a const reference to the internal vector. Does not throw.

7.
```
void reserve(typename vector_type::size_type size);
```

Calls reserve() method of the internal vector. Resets the stream to the first position. Throws if the internals vector's reserve throws.

8.
```
void clear();
```

Calls clear() method of the internal vector. Resets the stream to the first position.

9.
```
basic_ovectorstream(std::ios_base::openmode mode = std::ios_base::out);
```

Constructor. Throws if vector_type default constructor throws.

10.
```
template<typename VectorParameter>
    basic_ovectorstream(const VectorParameter & param,
                        std::ios_base::openmode mode = std::ios_base::out);
```

Constructor. Throws if vector_type(const VectorParameter &param) throws.

11.
```
~basic_ovectorstream();
```

12.
```
basic_vectorbuf< CharVector, CharTraits > * rdbuf() const;
```

Returns the address of the stored stream buffer.

13.
```
void swap_vector(vector_type & vect);
```

Swaps the underlying vector with the passed vector. This function resets the write position in the stream. Does not throw.

14.

```
const vector_type & vector() const;
```

Returns a const reference to the internal vector. Does not throw.

15.

```
void reserve(typename vector_type::size_type size);
```

Calls reserve() method of the internal vector. Resets the stream to the first position. Throws if the internals vector's reserve throws.

```
const vector_type & vector() const;
```

Class template basic_vectorstream

boost::interprocess::basic_vectorstream

Synopsis

```
// In header: <boost/interprocess/streams/vectorstream.hpp>

template<typename CharVector, typename CharTraits>
class basic_vectorstream {
public:
  // types
  typedef CharVector                                             vector_type;
  typedef std::basic_ios< typename CharVector::value_type, CharTraits >::char_type char_type;
  typedef std::basic_ios< char_type, CharTraits >::int_type      int_type;
  typedef std::basic_ios< char_type, CharTraits >::pos_type      pos_type;
  typedef std::basic_ios< char_type, CharTraits >::off_type      off_type;
  typedef std::basic_ios< char_type, CharTraits >::traits_type   traits_type;

  // construct/copy/destruct
  basic_vectorstream(std::ios_base::openmode = std::ios_base::in|std::ios_base::out);
  template<typename VectorParameter>
    basic_vectorstream(const VectorParameter &,
                       std::ios_base::openmode = std::ios_base::in|std::ios_base::out);
  ~basic_vectorstream();

  // public member functions
  basic_vectorbuf< CharVector, CharTraits > * rdbuf() const;
  void swap_vector(vector_type &);
  const vector_type & vector() const;
  void reserve(typename vector_type::size_type);
  void clear();
};
```

Description

A basic_iostream class that holds a character vector specified by CharVector template parameter as its formatting buffer. The vector must have contiguous storage, like std::vector, boost::interprocess::vector or boost::interprocess::basic_string

`basic_vectorstream` public construct/copy/destruct

1.
```
basic_vectorstream(std::ios_base::openmode mode = std::ios_base::in|std::ios_base::out);
```

   Constructor. Throws if vector_type default constructor throws.

2.
```
template<typename VectorParameter>
  basic_vectorstream(const VectorParameter & param,
                     std::ios_base::openmode mode = std::ios_base::in|std::ios_base::out);
```

   Constructor. Throws if vector_type(const VectorParameter &param) throws.

3.
```
~basic_vectorstream();
```

`basic_vectorstream` public member functions

1.
```
basic_vectorbuf< CharVector, CharTraits > * rdbuf() const;
```

2.
```
void swap_vector(vector_type & vect);
```

Swaps the underlying vector with the passed vector. This function resets the read/write position in the stream. Does not throw.

3.
```
const vector_type & vector() const;
```

Returns a const reference to the internal vector. Does not throw.

4.
```
void reserve(typename vector_type::size_type size);
```

Calls reserve() method of the internal vector. Resets the stream to the first position. Throws if the internals vector's reserve throws.

5.
```
void clear();
```

Calls clear() method of the internal vector. Resets the stream to the first position.

Header <boost/interprocess/sync/file_lock.hpp>

Describes a class that wraps file locking capabilities.

```
namespace boost {
  namespace interprocess {
    class file_lock;
  }
}
```

Class file_lock

boost::interprocess::file_lock

Synopsis

```
// In header: <boost/interprocess/sync/file_lock.hpp>


class file_lock {
public:
  // construct/copy/destruct
  file_lock();
  file_lock(const char *);
  file_lock(file_lock &&);
  file_lock& operator=(file_lock &&);
  ~file_lock();

  // public member functions
  void swap(file_lock &);
  void lock();
  bool try_lock();
  bool timed_lock(const boost::posix_time::ptime &);
  void unlock();
  void lock_sharable();
  bool try_lock_sharable();
  bool timed_lock_sharable(const boost::posix_time::ptime &);
  void unlock_sharable();
};
```

Description

A file lock, is a mutual exclusion utility similar to a mutex using a file. A file lock has sharable and exclusive locking capabilities and can be used with scoped_lock and sharable_lock classes. A file lock can't guarantee synchronization between threads of the same process so just use file locks to synchronize threads from different processes.

`file_lock` public construct/copy/destruct

1.
   ```
   file_lock();
   ```

   Constructs an empty file mapping. Does not throw

2.
   ```
   file_lock(const char * name);
   ```

   Opens a file lock. Throws interprocess_exception if the file does not exist or there are no operating system resources.

3.
   ```
   file_lock(file_lock && moved);
   ```

   Moves the ownership of "moved"'s file mapping object to *this. After the call, "moved" does not represent any file mapping object. Does not throw

4.
   ```
   file_lock& operator=(file_lock && moved);
   ```

   Moves the ownership of "moved"'s file mapping to *this. After the call, "moved" does not represent any file mapping. Does not throw

---

5.
```
~file_lock();
```

Closes a file lock. Does not throw.

`file_lock` public member functions

1.
```
void swap(file_lock & other);
```

Swaps two file_locks. Does not throw.

2.
```
void lock();
```

Effects: The calling thread tries to obtain exclusive ownership of the mutex, and if another thread has exclusive, or sharable ownership of the mutex, it waits until it can obtain the ownership. Throws: interprocess_exception on error.

3.
```
bool try_lock();
```

Effects: The calling thread tries to acquire exclusive ownership of the mutex without waiting. If no other thread has exclusive, or sharable ownership of the mutex this succeeds. Returns: If it can acquire exclusive ownership immediately returns true. If it has to wait, returns false. Throws: interprocess_exception on error.

4.
```
bool timed_lock(const boost::posix_time::ptime & abs_time);
```

Effects: The calling thread tries to acquire exclusive ownership of the mutex waiting if necessary until no other thread has has exclusive, or sharable ownership of the mutex or abs_time is reached. Returns: If acquires exclusive ownership, returns true. Otherwise returns false. Throws: interprocess_exception on error.

5.
```
void unlock();
```

Precondition: The thread must have exclusive ownership of the mutex. Effects: The calling thread releases the exclusive ownership of the mutex. Throws: An exception derived from interprocess_exception on error.

6.
```
void lock_sharable();
```

Effects: The calling thread tries to obtain sharable ownership of the mutex, and if another thread has exclusive ownership of the mutex, waits until it can obtain the ownership. Throws: interprocess_exception on error.

7.
```
bool try_lock_sharable();
```

Effects: The calling thread tries to acquire sharable ownership of the mutex without waiting. If no other thread has has exclusive ownership of the mutex this succeeds. Returns: If it can acquire sharable ownership immediately returns true. If it has to wait, returns false. Throws: interprocess_exception on error.

8.
```
bool timed_lock_sharable(const boost::posix_time::ptime & abs_time);
```

Effects: The calling thread tries to acquire sharable ownership of the mutex waiting if necessary until no other thread has has exclusive ownership of the mutex or abs_time is reached. Returns: If acquires sharable ownership, returns true. Otherwise returns false. Throws: interprocess_exception on error.

9.
```
void unlock_sharable();
```

Precondition: The thread must have sharable ownership of the mutex. Effects: The calling thread releases the sharable ownership of the mutex. Throws: An exception derived from interprocess_exception on error.

Header <boost/interprocess/sync/interprocess_barrier.hpp>

```
namespace boost {
  namespace interprocess {
    class barrier;
  }
}
```

Class barrier

boost::interprocess::barrier

Synopsis

```
// In header: <boost/interprocess/sync/interprocess_barrier.hpp>


class barrier {
public:
  // construct/copy/destruct
  barrier(unsigned int);
  ~barrier();

  // public member functions
  bool wait();
};
```

Description

An object of class barrier is a synchronization primitive that can be placed in shared memory used to cause a set of threads from different processes to wait until they each perform a certain function or each reach a particular point in their execution.

`barrier` public construct/copy/destruct

1.
```
barrier(unsigned int count);
```

Constructs a barrier object that will cause count threads to block on a call to wait().

2.
```
~barrier();
```

Destroys *this. If threads are still executing their wait() operations, the behavior for these threads is undefined.

`barrier` public member functions

1.
```
bool wait();
```

Effects: Wait until N threads call wait(), where N equals the count provided to the constructor for the barrier object. Note that if the barrier is destroyed before wait() can return, the behavior is undefined. Returns: Exactly one of the N threads will receive a return value of true, the others will receive a value of false. Precisely which thread receives the return value of true will be implementation-defined. Applications can use this value to designate one thread as a leader that will take a certain action, and the other threads emerging from the barrier can wait for that action to take place.

Header <boost/interprocess/sync/interprocess_condition.hpp>

Describes process-shared variables interprocess_condition class

```
namespace boost {
  namespace interprocess {
    class interprocess_condition;
  }
  namespace posix_time {
  }
}
```

Class interprocess_condition

boost::interprocess::interprocess_condition

Synopsis

```
// In header: <boost/interprocess/sync/interprocess_condition.hpp>


class interprocess_condition {
public:
  // construct/copy/destruct
  interprocess_condition();
  ~interprocess_condition();

  // public member functions
  void notify_one();
  void notify_all();
  template<typename L> void wait(L &);
  template<typename L, typename Pr> void wait(L &, Pr);
  template<typename L> bool timed_wait(L &, const boost::posix_time::ptime &);
  template<typename L, typename Pr>
    bool timed_wait(L &, const boost::posix_time::ptime &, Pr);
};
```

Description

This class is a condition variable that can be placed in shared memory or memory mapped files.

`interprocess_condition` **public construct/copy/destruct**

1.
   ```
   interprocess_condition();
   ```

   Constructs a interprocess_condition. On error throws interprocess_exception.

2.
   ```
   ~interprocess_condition();
   ```

   Destroys *this liberating system resources.

`interprocess_condition` **public member functions**

1.
   ```
   void notify_one();
   ```

   If there is a thread waiting on *this, change that thread's state to ready. Otherwise there is no effect.

2.
   ```
   void notify_all();
   ```

   Change the state of all threads waiting on *this to ready. If there are no waiting threads, notify_all() has no effect.

3.
   ```
   template<typename L> void wait(L & lock);
   ```

   Releases the lock on the interprocess_mutex object associated with lock, blocks the current thread of execution until readied by a call to this->notify_one() or this->notify_all(), and then reacquires the lock.

4.
   ```
   template<typename L, typename Pr> void wait(L & lock, Pr pred);
   ```

The same as: while (!pred()) wait(lock)

5.
```cpp
template<typename L>
   bool timed_wait(L & lock, const boost::posix_time::ptime & abs_time);
```

Releases the lock on the interprocess_mutex object associated with lock, blocks the current thread of execution until readied by a call to this->notify_one() or this->notify_all(), or until time abs_time is reached, and then reacquires the lock. Returns: false if time abs_time is reached, otherwise true.

6.
```cpp
template<typename L, typename Pr>
   bool timed_wait(L & lock, const boost::posix_time::ptime & abs_time,
                   Pr pred);
```

The same as: while (!pred()) { if (!timed_wait(lock, abs_time)) return pred(); } return true;

Header <boost/interprocess/sync/interprocess_mutex.hpp>

Describes a mutex class that can be placed in memory shared by several processes.

```cpp
namespace boost {
  namespace interprocess {
    class interprocess_mutex;
  }
}
```

Class interprocess_mutex

boost::interprocess::interprocess_mutex

Synopsis

```
// In header: <boost/interprocess/sync/interprocess_mutex.hpp>


class interprocess_mutex {
public:
  // construct/copy/destruct
  interprocess_mutex();
  ~interprocess_mutex();

  // public member functions
  void lock();
  bool try_lock();
  bool timed_lock(const boost::posix_time::ptime &);
  void unlock();
};
```

Description

Wraps a interprocess_mutex that can be placed in shared memory and can be shared between processes. Allows timed lock tries

`interprocess_mutex` public construct/copy/destruct

1.
```
interprocess_mutex();
```

Constructor. Throws interprocess_exception on error.

2.
```
~interprocess_mutex();
```

Destructor. If any process uses the mutex after the destructor is called the result is undefined. Does not throw.

`interprocess_mutex` public member functions

1.
```
void lock();
```

Effects: The calling thread tries to obtain ownership of the mutex, and if another thread has ownership of the mutex, it waits until it can obtain the ownership. If a thread takes ownership of the mutex the mutex must be unlocked by the same mutex. Throws: interprocess_exception on error.

2.
```
bool try_lock();
```

Effects: The calling thread tries to obtain ownership of the mutex, and if another thread has ownership of the mutex returns immediately. Returns: If the thread acquires ownership of the mutex, returns true, if the another thread has ownership of the mutex, returns false. Throws: interprocess_exception on error.

3.
```
bool timed_lock(const boost::posix_time::ptime & abs_time);
```

Effects: The calling thread will try to obtain exclusive ownership of the mutex if it can do so in until the specified time is reached. If the mutex supports recursive locking, the mutex must be unlocked the same number of times it is locked. Returns: If the thread acquires ownership of the mutex, returns true, if the timeout expires returns false. Throws: interprocess_exception on error.

4.
```
void unlock();
```

Effects: The calling thread releases the exclusive ownership of the mutex. Throws: interprocess_exception on error.

Header <boost/interprocess/sync/interprocess_recursive_mutex.hpp>

Describes interprocess_recursive_mutex and shared_recursive_try_mutex classes

```
namespace boost {
  namespace interprocess {
    class interprocess_recursive_mutex;
  }
}
```

Class interprocess_recursive_mutex

boost::interprocess::interprocess_recursive_mutex

Synopsis

```
// In header: <boost/interprocess/sync/interprocess_recursive_mutex.hpp>


class interprocess_recursive_mutex {
public:
  // construct/copy/destruct
  interprocess_recursive_mutex();
  ~interprocess_recursive_mutex();

  // public member functions
  void lock();
  bool try_lock();
  bool timed_lock(const boost::posix_time::ptime &);
  void unlock();
};
```

Description

Wraps a interprocess_mutex that can be placed in shared memory and can be shared between processes. Allows several locking calls by the same process. Allows timed lock tries

`interprocess_recursive_mutex` public construct/copy/destruct

1.
```
interprocess_recursive_mutex();
```

   Constructor. Throws interprocess_exception on error.

2.
```
~interprocess_recursive_mutex();
```

   Destructor. If any process uses the mutex after the destructor is called the result is undefined. Does not throw.

`interprocess_recursive_mutex` public member functions

1.
```
void lock();
```

   Effects: The calling thread tries to obtain ownership of the mutex, and if another thread has ownership of the mutex, it waits until it can obtain the ownership. If a thread takes ownership of the mutex the mutex must be unlocked by the same mutex. The mutex must be unlocked the same number of times it is locked. Throws: interprocess_exception on error.

2.
```
bool try_lock();
```

   Tries to lock the interprocess_mutex, returns false when interprocess_mutex is already locked, returns true when success. The mutex must be unlocked the same number of times it is locked. Throws: interprocess_exception if a severe error is found

3.
```
bool timed_lock(const boost::posix_time::ptime & abs_time);
```

   Tries to lock the interprocess_mutex, if interprocess_mutex can't be locked before abs_time time, returns false. The mutex must be unlocked the same number of times it is locked. Throws: interprocess_exception if a severe error is found

---

4.
```
void unlock();
```

Effects: The calling thread releases the exclusive ownership of the mutex. If the mutex supports recursive locking, the mutex must be unlocked the same number of times it is locked. Throws: interprocess_exception on error.

Header <boost/interprocess/sync/interprocess_semaphore.hpp>

Describes a interprocess_semaphore class for inter-process synchronization

```
namespace boost {
  namespace interprocess {
    class interprocess_semaphore;
  }
}
```

Class interprocess_semaphore

boost::interprocess::interprocess_semaphore

Synopsis

```
// In header: <boost/interprocess/sync/interprocess_semaphore.hpp>


class interprocess_semaphore {
public:
  // construct/copy/destruct
  interprocess_semaphore(unsigned int);
  ~interprocess_semaphore();

  // public member functions
  void post();
  void wait();
  bool try_wait();
  bool timed_wait(const boost::posix_time::ptime &);
};
```

Description

Wraps a interprocess_semaphore that can be placed in shared memory and can be shared between processes. Allows timed lock tries

interprocess_semaphore public construct/copy/destruct

1.
```
interprocess_semaphore(unsigned int initialCount);
```

Creates a interprocess_semaphore with the given initial count. interprocess_exception if there is an error.

2.
```
~interprocess_semaphore();
```

Destroys the interprocess_semaphore. Does not throw

interprocess_semaphore public member functions

1.
```
void post();
```

Increments the interprocess_semaphore count. If there are processes/threads blocked waiting for the interprocess_semaphore, then one of these processes will return successfully from its wait function. If there is an error an interprocess_exception exception is thrown.

2.
```
void wait();
```

Decrements the interprocess_semaphore. If the interprocess_semaphore value is not greater than zero, then the calling process/thread blocks until it can decrement the counter. If there is an error an interprocess_exception exception is thrown.

3.
```
bool try_wait();
```

Decrements the interprocess_semaphore if the interprocess_semaphore's value is greater than zero and returns true. If the value is not greater than zero returns false. If there is an error an Interprocess_exception exception is thrown.

4.
```
bool timed_wait(const boost::posix_time::ptime & abs_time);
```

Decrements the interprocess_semaphore if the interprocess_semaphore's value is greater than zero and returns true. Otherwise, waits for the interprocess_semaphore to the posted or the timeout expires. If the timeout expires, the function returns false. If the interprocess_semaphore is posted the function returns true. If there is an error throws sem_exception

Header <boost/interprocess/sync/interprocess_upgradable_mutex.hpp>

Describes interprocess_upgradable_mutex class

```
namespace boost {
  namespace interprocess {
    class interprocess_upgradable_mutex;
  }
}
```

Class interprocess_upgradable_mutex

boost::interprocess::interprocess_upgradable_mutex

Synopsis

```
// In header: <boost/interprocess/sync/interprocess_upgradable_mutex.hpp>


class interprocess_upgradable_mutex {
public:
  // construct/copy/destruct
  interprocess_upgradable_mutex(const interprocess_upgradable_mutex &);
  interprocess_upgradable_mutex();
  interprocess_upgradable_mutex&
  operator=(const interprocess_upgradable_mutex &);
  ~interprocess_upgradable_mutex();

  // public member functions
  void lock();
  bool try_lock();
  bool timed_lock(const boost::posix_time::ptime &);
  void unlock();
  void lock_sharable();
  bool try_lock_sharable();
  bool timed_lock_sharable(const boost::posix_time::ptime &);
  void unlock_sharable();
  void lock_upgradable();
  bool try_lock_upgradable();
  bool timed_lock_upgradable(const boost::posix_time::ptime &);
  void unlock_upgradable();
  void unlock_and_lock_upgradable();
  void unlock_and_lock_sharable();
  void unlock_upgradable_and_lock_sharable();
  void unlock_upgradable_and_lock();
  bool try_unlock_upgradable_and_lock();
  *bool timed_unlock_upgradable_and_lock(const boost::posix_time::ptime &);
  bool try_unlock_sharable_and_lock();
  bool try_unlock_sharable_and_lock_upgradable();
};
```

Description

Wraps a interprocess_upgradable_mutex that can be placed in shared memory and can be shared between processes. Allows timed lock tries

interprocess_upgradable_mutex public construct/copy/destruct

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

2.
```
interprocess_upgradable_mutex();
```

Constructs the upgradable lock. Throws interprocess_exception on error.

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

4.
```
~interprocess_upgradable_mutex();
```

Destroys the upgradable lock. Does not throw.

`interprocess_upgradable_mutex` public member functions

1.
```
void lock();
```

Effects: The calling thread tries to obtain exclusive ownership of the mutex, and if another thread has exclusive, sharable or upgradable ownership of the mutex, it waits until it can obtain the ownership. Throws: interprocess_exception on error.

2.
```
bool try_lock();
```

Effects: The calling thread tries to acquire exclusive ownership of the mutex without waiting. If no other thread has exclusive, sharable or upgradable ownership of the mutex this succeeds. Returns: If it can acquire exclusive ownership immediately returns true. If it has to wait, returns false. Throws: interprocess_exception on error.

3.
```
bool timed_lock(const boost::posix_time::ptime & abs_time);
```

Effects: The calling thread tries to acquire exclusive ownership of the mutex waiting if necessary until no other thread has has exclusive, sharable or upgradable ownership of the mutex or abs_time is reached. Returns: If acquires exclusive ownership, returns true. Otherwise returns false. Throws: interprocess_exception on error.

4.
```
void unlock();
```

Precondition: The thread must have exclusive ownership of the mutex. Effects: The calling thread releases the exclusive ownership of the mutex. Throws: An exception derived from interprocess_exception on error.

5.
```
void lock_sharable();
```

Effects: The calling thread tries to obtain sharable ownership of the mutex, and if another thread has exclusive or upgradable ownership of the mutex, waits until it can obtain the ownership. Throws: interprocess_exception on error.

6.
```
bool try_lock_sharable();
```

Effects: The calling thread tries to acquire sharable ownership of the mutex without waiting. If no other thread has has exclusive or upgradable ownership of the mutex this succeeds. Returns: If it can acquire sharable ownership immediately returns true. If it has to wait, returns false. Throws: interprocess_exception on error.

7.
```
bool timed_lock_sharable(const boost::posix_time::ptime & abs_time);
```

Effects: The calling thread tries to acquire sharable ownership of the mutex waiting if necessary until no other thread has has exclusive or upgradable ownership of the mutex or abs_time is reached. Returns: If acquires sharable ownership, returns true. Otherwise returns false. Throws: interprocess_exception on error.

8.
```
void unlock_sharable();
```

Precondition: The thread must have sharable ownership of the mutex. Effects: The calling thread releases the sharable ownership of the mutex. Throws: An exception derived from interprocess_exception on error.

9.
```
void lock_upgradable();
```

Effects: The calling thread tries to obtain upgradable ownership of the mutex, and if another thread has exclusive or upgradable ownership of the mutex, waits until it can obtain the ownership. Throws: interprocess_exception on error.

10.
```
bool try_lock_upgradable();
```

Effects: The calling thread tries to acquire upgradable ownership of the mutex without waiting. If no other thread has has exclusive or upgradable ownership of the mutex this succeeds. Returns: If it can acquire upgradable ownership immediately returns true. If it has to wait, returns false. Throws: interprocess_exception on error.

11.
```
bool timed_lock_upgradable(const boost::posix_time::ptime & abs_time);
```

Effects: The calling thread tries to acquire upgradable ownership of the mutex waiting if necessary until no other thread has has exclusive or upgradable ownership of the mutex or abs_time is reached. Returns: If acquires upgradable ownership, returns true. Otherwise returns false. Throws: interprocess_exception on error.

12.
```
void unlock_upgradable();
```

Precondition: The thread must have upgradable ownership of the mutex. Effects: The calling thread releases the upgradable ownership of the mutex. Throws: An exception derived from interprocess_exception on error.

13.
```
void unlock_and_lock_upgradable();
```

Precondition: The thread must have exclusive ownership of the mutex. Effects: The thread atomically releases exclusive ownership and acquires upgradable ownership. This operation is non-blocking. Throws: An exception derived from interprocess_exception on error.

14.
```
void unlock_and_lock_sharable();
```

Precondition: The thread must have exclusive ownership of the mutex. Effects: The thread atomically releases exclusive ownership and acquires sharable ownership. This operation is non-blocking. Throws: An exception derived from interprocess_exception on error.

15.
```
void unlock_upgradable_and_lock_sharable();
```

Precondition: The thread must have upgradable ownership of the mutex. Effects: The thread atomically releases upgradable ownership and acquires sharable ownership. This operation is non-blocking. Throws: An exception derived from interprocess_exception on error.

16.
```
void unlock_upgradable_and_lock();
```

Precondition: The thread must have upgradable ownership of the mutex. Effects: The thread atomically releases upgradable ownership and acquires exclusive ownership. This operation will block until all threads with sharable ownership release their sharable lock. Throws: An exception derived from interprocess_exception on error.

17.
```
bool try_unlock_upgradable_and_lock();
```

Precondition: The thread must have upgradable ownership of the mutex. Effects: The thread atomically releases upgradable ownership and tries to acquire exclusive ownership. This operation will fail if there are threads with sharable ownership, but it will maintain upgradable ownership. Returns: If acquires exclusive ownership, returns true. Otherwise returns false. Throws: An exception derived from interprocess_exception on error.

18.
```
*bool timed_unlock_upgradable_and_lock(const boost::posix_time::ptime & abs_time);
```

Precondition: The thread must have upgradable ownership of the mutex. Effects: The thread atomically releases upgradable ownership and tries to acquire exclusive ownership, waiting if necessary until abs_time. This operation will fail if there are threads with sharable ownership or timeout reaches, but it will maintain upgradable ownership. Returns: If acquires exclusive ownership, returns true. Otherwise returns false. Throws: An exception derived from interprocess_exception on error.

19.
```
bool try_unlock_sharable_and_lock();
```

Precondition: The thread must have sharable ownership of the mutex. Effects: The thread atomically releases sharable ownership and tries to acquire exclusive ownership. This operation will fail if there are threads with sharable or upgradable ownership, but it will maintain sharable ownership. Returns: If acquires exclusive ownership, returns true. Otherwise returns false. Throws: An exception derived from interprocess_exception on error.

20.
```
bool try_unlock_sharable_and_lock_upgradable();
```

Precondition: The thread must have sharable ownership of the mutex. Effects: The thread atomically releases sharable ownership and tries to acquire upgradable ownership. This operation will fail if there are threads with sharable or upgradable ownership, but it will maintain sharable ownership. Returns: If acquires upgradable ownership, returns true. Otherwise returns false. Throws: An exception derived from interprocess_exception on error.

Header <boost/interprocess/sync/lock_options.hpp>

Describes the lock options with associated with interprocess_mutex lock constructors.

```
namespace boost {
  namespace interprocess {
    struct defer_lock_type;
    struct try_to_lock_type;
    struct accept_ownership_type;

    static const defer_lock_type defer_lock;
    static const try_to_lock_type try_to_lock;
    static const accept_ownership_type accept_ownership;
  }
  namespace posix_time {
  }
}
```

Struct defer_lock_type

boost::interprocess::defer_lock_type — Type to indicate to a mutex lock constructor that must not lock the mutex.

Synopsis

```
// In header: <boost/interprocess/sync/lock_options.hpp>


struct defer_lock_type {
};
```

Struct try_to_lock_type

boost::interprocess::try_to_lock_type — Type to indicate to a mutex lock constructor that must try to lock the mutex.

Synopsis

```
// In header: <boost/interprocess/sync/lock_options.hpp>


struct try_to_lock_type {
};
```

Struct accept_ownership_type

boost::interprocess::accept_ownership_type — Type to indicate to a mutex lock constructor that the mutex is already locked.

Synopsis

```
// In header: <boost/interprocess/sync/lock_options.hpp>


struct accept_ownership_type {
};
```

Global defer_lock

boost::interprocess::defer_lock

Synopsis

```
// In header: <boost/interprocess/sync/lock_options.hpp>

static const defer_lock_type defer_lock;
```

Description

An object indicating that the locking must be deferred.

Global try_to_lock

boost::interprocess::try_to_lock

Synopsis

```
// In header: <boost/interprocess/sync/lock_options.hpp>

static const try_to_lock_type try_to_lock;
```

Description

An object indicating that a try_lock() operation must be executed.

Global accept_ownership

boost::interprocess::accept_ownership

Synopsis

```
// In header: <boost/interprocess/sync/lock_options.hpp>

static const accept_ownership_type accept_ownership;
```

Description

An object indicating that the ownership of lockable object must be accepted by the new owner.

Header <boost/interprocess/sync/mutex_family.hpp>

Describes a shared interprocess_mutex family fit algorithm used to allocate objects in shared memory.

```
namespace boost {
  namespace interprocess {
    struct mutex_family;
    struct null_mutex_family;
  }
}
```

Struct mutex_family

boost::interprocess::mutex_family

Synopsis

```
// In header: <boost/interprocess/sync/mutex_family.hpp>


struct mutex_family {
  // types
  typedef boost::interprocess::interprocess_mutex           mutex_type;
  typedef boost::interprocess::interprocess_recursive_mutex recursive_mutex_type;
};
```

Description

Describes interprocess_mutex family to use with Interprocess framework based on boost::interprocess synchronization objects.

Struct null_mutex_family

boost::interprocess::null_mutex_family

Synopsis

```
// In header: <boost/interprocess/sync/mutex_family.hpp>


struct null_mutex_family {
  // types
  typedef boost::interprocess::null_mutex mutex_type;
  typedef boost::interprocess::null_mutex recursive_mutex_type;
};
```

Description

Describes interprocess_mutex family to use with Interprocess frameworks based on null operation synchronization objects.

Header <boost/interprocess/sync/named_condition.hpp>

Describes process-shared variables interprocess_condition class

```
namespace boost {
  namespace interprocess {
    class named_condition;
  }
}
```

Class named_condition

boost::interprocess::named_condition

Synopsis

```
// In header: <boost/interprocess/sync/named_condition.hpp>


class named_condition {
public:
  // construct/copy/destruct
  named_condition(create_only_t, const char *,
                  const permissions & = permissions());
  named_condition(open_or_create_t, const char *,
                  const permissions & = permissions());
  named_condition(open_only_t, const char *);
  ~named_condition();

  // public member functions
  *void notify_one();
  void notify_all();
  template<typename L> void wait(L &);
  template<typename L, typename Pr> void wait(L &, Pr);
  template<typename L> bool timed_wait(L &, const boost::posix_time::ptime &);
  template<typename L, typename Pr>
    bool timed_wait(L &, const boost::posix_time::ptime &, Pr);

  // public static functions
  static bool remove(const char *);
};
```

Description

A global condition variable that can be created by name. This condition variable is designed to work with named_mutex and can't be placed in shared memory or memory mapped files.

named_condition public construct/copy/destruct

1.
```
named_condition(create_only_t create_only, const char * name,
                const permissions & perm = permissions());
```

Creates a global condition with a name. If the condition can't be created throws interprocess_exception

2.
```
named_condition(open_or_create_t open_or_create, const char * name,
                const permissions & perm = permissions());
```

Opens or creates a global condition with a name. If the condition is created, this call is equivalent to named_condition(create_only_t, ...) If the condition is already created, this call is equivalent named_condition(open_only_t, ...) Does not throw

3.
```
named_condition(open_only_t open_only, const char * name);
```

Opens a global condition with a name if that condition is previously created. If it is not previously created this function throws interprocess_exception.

4.
```
~named_condition();
```

Destroys *this and indicates that the calling process is finished using the resource. The destructor function will deallocate any system resources allocated by the system for use by this process for this resource. The resource can still be opened again calling the open constructor overload. To erase the resource from the system use remove().

`named_condition` public member functions

1.
```
*void notify_one();
```

If there is a thread waiting on *this, change that thread's state to ready. Otherwise there is no effect.

2.
```
void notify_all();
```

Change the state of all threads waiting on *this to ready. If there are no waiting threads, notify_all() has no effect.

3.
```
template<typename L> void wait(L & lock);
```

Releases the lock on the named_mutex object associated with lock, blocks the current thread of execution until readied by a call to this->notify_one() or this->notify_all(), and then reacquires the lock.

4.
```
template<typename L, typename Pr> void wait(L & lock, Pr pred);
```

The same as: while (!pred()) wait(lock)

5.
```
template<typename L>
   bool timed_wait(L & lock, const boost::posix_time::ptime & abs_time);
```

Releases the lock on the named_mutex object associated with lock, blocks the current thread of execution until readied by a call to this->notify_one() or this->notify_all(), or until time abs_time is reached, and then reacquires the lock. Returns: false if time abs_time is reached, otherwise true.

6.
```
template<typename L, typename Pr>
   bool timed_wait(L & lock, const boost::posix_time::ptime & abs_time,
                   Pr pred);
```

The same as: while (!pred()) { if (!timed_wait(lock, abs_time)) return pred(); } return true;

`named_condition` public static functions

1.
```
static bool remove(const char * name);
```

Erases a named condition from the system. Returns false on error. Never throws.

Header <boost/interprocess/sync/named_mutex.hpp>

Describes a named mutex class for inter-process synchronization

```
namespace boost {
  namespace interprocess {
    class named_mutex;
  }
}
```

Class named_mutex

boost::interprocess::named_mutex

Synopsis

```cpp
// In header: <boost/interprocess/sync/named_mutex.hpp>


class named_mutex {
public:
  // construct/copy/destruct
  named_mutex(create_only_t, const char *,
              const permissions & = permissions());
  named_mutex(open_or_create_t, const char *,
              const permissions & = permissions());
  named_mutex(open_only_t, const char *);
  ~named_mutex();

  // public member functions
  void unlock();
  void lock();
  bool try_lock();
  bool timed_lock(const boost::posix_time::ptime &);

  // public static functions
  static bool remove(const char *);
};
```

Description

A mutex with a global name, so it can be found from different processes. This mutex can't be placed in shared memory, and each process should have it's own named_mutex.

named_mutex public construct/copy/destruct

1.
```cpp
named_mutex(create_only_t create_only, const char * name,
            const permissions & perm = permissions());
```

Creates a global interprocess_mutex with a name. Throws interprocess_exception on error.

2.
```cpp
named_mutex(open_or_create_t open_or_create, const char * name,
            const permissions & perm = permissions());
```

Opens or creates a global mutex with a name. If the mutex is created, this call is equivalent to named_mutex(create_only_t, ...) If the mutex is already created, this call is equivalent named_mutex(open_only_t, ...) Does not throw

3.
```cpp
named_mutex(open_only_t open_only, const char * name);
```

Opens a global mutex with a name if that mutex is previously created. If it is not previously created this function throws interprocess_exception.

4.
```cpp
~named_mutex();
```

Destroys *this and indicates that the calling process is finished using the resource. The destructor function will deallocate any system resources allocated by the system for use by this process for this resource. The resource can still be opened again calling the open constructor overload. To erase the resource from the system use remove().

`named_mutex` public member functions

1.
```cpp
void unlock();
```

Unlocks a previously locked interprocess_mutex.

2.
```cpp
void lock();
```

Locks interprocess_mutex, sleeps when interprocess_mutex is already locked. Throws interprocess_exception if a severe error is found

3.
```cpp
bool try_lock();
```

Tries to lock the interprocess_mutex, returns false when interprocess_mutex is already locked, returns true when success. Throws interprocess_exception if a severe error is found

4.
```cpp
bool timed_lock(const boost::posix_time::ptime & abs_time);
```

Tries to lock the interprocess_mutex until time abs_time, Returns false when timeout expires, returns true when locks. Throws interprocess_exception if a severe error is found

`named_mutex` public static functions

1.
```cpp
static bool remove(const char * name);
```

Erases a named mutex from the system. Returns false on error. Never throws.

Header <boost/interprocess/sync/named_recursive_mutex.hpp>

Describes a named named_recursive_mutex class for inter-process synchronization

```cpp
namespace boost {
  namespace interprocess {
    class named_recursive_mutex;
  }
}
```

Class named_recursive_mutex

boost::interprocess::named_recursive_mutex

Synopsis

```
// In header: <boost/interprocess/sync/named_recursive_mutex.hpp>


class named_recursive_mutex {
public:
  // construct/copy/destruct
  named_recursive_mutex(create_only_t, const char *,
                        const permissions & = permissions());
  named_recursive_mutex(open_or_create_t, const char *,
                        const permissions & = permissions());
  named_recursive_mutex(open_only_t, const char *);
  ~named_recursive_mutex();

  // public member functions
  void unlock();
  void lock();
  bool try_lock();
  bool timed_lock(const boost::posix_time::ptime &);

  // public static functions
  static bool remove(const char *);
};
```

Description

A recursive mutex with a global name, so it can be found from different processes. This mutex can't be placed in shared memory, and each process should have it's own named_recursive_mutex.

`named_recursive_mutex` public construct/copy/destruct

1.
```
named_recursive_mutex(create_only_t create_only, const char * name,
                      const permissions & perm = permissions());
```

Creates a global recursive_mutex with a name. If the recursive_mutex can't be created throws interprocess_exception

2.
```
named_recursive_mutex(open_or_create_t open_or_create, const char * name,
                      const permissions & perm = permissions());
```

Opens or creates a global recursive_mutex with a name. If the recursive_mutex is created, this call is equivalent to named_recursive_mutex(create_only_t, ...) If the recursive_mutex is already created, this call is equivalent named_recursive_mutex(open_only_t, ...) Does not throw

3.
```
named_recursive_mutex(open_only_t open_only, const char * name);
```

Opens a global recursive_mutex with a name if that recursive_mutex is previously created. If it is not previously created this function throws interprocess_exception.

4.
```
~named_recursive_mutex();
```

Destroys *this and indicates that the calling process is finished using the resource. The destructor function will deallocate any system resources allocated by the system for use by this process for this resource. The resource can still be opened again calling the open constructor overload. To erase the resource from the system use remove().

`named_recursive_mutex` public member functions

1.
```
void unlock();
```

Unlocks a previously locked named_recursive_mutex.

2.
```
void lock();
```

Locks named_recursive_mutex, sleeps when named_recursive_mutex is already locked. Throws interprocess_exception if a severe error is found.

3.
```
bool try_lock();
```

Tries to lock the named_recursive_mutex, returns false when named_recursive_mutex is already locked, returns true when success. Throws interprocess_exception if a severe error is found.

4.
```
bool timed_lock(const boost::posix_time::ptime & abs_time);
```

Tries to lock the named_recursive_mutex until time abs_time, Returns false when timeout expires, returns true when locks. Throws interprocess_exception if a severe error is found

`named_recursive_mutex` public static functions

1.
```
static bool remove(const char * name);
```

Erases a named recursive mutex from the system

Header <boost/interprocess/sync/named_semaphore.hpp>

Describes a named semaphore class for inter-process synchronization

```
namespace boost {
  namespace interprocess {
    class named_semaphore;
  }
}
```

Class named_semaphore

boost::interprocess::named_semaphore

Synopsis

```cpp
// In header: <boost/interprocess/sync/named_semaphore.hpp>


class named_semaphore {
public:
  // construct/copy/destruct
  named_semaphore(create_only_t, const char *, unsigned int,
                  const permissions & = permissions());
  named_semaphore(open_or_create_t, const char *, unsigned int,
                  const permissions & = permissions());
  named_semaphore(open_only_t, const char *);
  ~named_semaphore();

  // public member functions
  void post();
  void wait();
  bool try_wait();
  bool timed_wait(const boost::posix_time::ptime &);

  // public static functions
  static bool remove(const char *);
};
```

Description

A semaphore with a global name, so it can be found from different processes. Allows several resource sharing patterns and efficient acknowledgment mechanisms.

`named_semaphore` public construct/copy/destruct

1.
```cpp
named_semaphore(create_only_t, const char * name, unsigned int initialCount,
                const permissions & perm = permissions());
```

Creates a global semaphore with a name, and an initial count. If the semaphore can't be created throws interprocess_exception

2.
```cpp
named_semaphore(open_or_create_t, const char * name,
                unsigned int initialCount,
                const permissions & perm = permissions());
```

Opens or creates a global semaphore with a name, and an initial count. If the semaphore is created, this call is equivalent to named_semaphore(create_only_t, ...) If the semaphore is already created, this call is equivalent to named_semaphore(open_only_t, ...) and initialCount is ignored.

3.
```cpp
named_semaphore(open_only_t, const char * name);
```

Opens a global semaphore with a name if that semaphore is previously. created. If it is not previously created this function throws interprocess_exception.

4.
```cpp
~named_semaphore();
```

Destroys *this and indicates that the calling process is finished using the resource. The destructor function will deallocate any system resources allocated by the system for use by this process for this resource. The resource can still be opened again calling the open constructor overload. To erase the resource from the system use remove().

`named_semaphore` public member functions

1.
```
void post();
```

Increments the semaphore count. If there are processes/threads blocked waiting for the semaphore, then one of these processes will return successfully from its wait function. If there is an error an interprocess_exception exception is thrown.

2.
```
void wait();
```

Decrements the semaphore. If the semaphore value is not greater than zero, then the calling process/thread blocks until it can decrement the counter. If there is an error an interprocess_exception exception is thrown.

3.
```
bool try_wait();
```

Decrements the semaphore if the semaphore's value is greater than zero and returns true. If the value is not greater than zero returns false. If there is an error an interprocess_exception exception is thrown.

4.
```
bool timed_wait(const boost::posix_time::ptime & abs_time);
```

Decrements the semaphore if the semaphore's value is greater than zero and returns true. Otherwise, waits for the semaphore to the posted or the timeout expires. If the timeout expires, the function returns false. If the semaphore is posted the function returns true. If there is an error throws sem_exception

`named_semaphore` public static functions

1.
```
static bool remove(const char * name);
```

Erases a named semaphore from the system. Returns false on error. Never throws.

Header <boost/interprocess/sync/named_upgradable_mutex.hpp>

Describes a named upgradable mutex class for inter-process synchronization

```
namespace boost {
  namespace interprocess {
    class named_upgradable_mutex;
  }
}
```

Class named_upgradable_mutex

boost::interprocess::named_upgradable_mutex

Synopsis

```
// In header: <boost/interprocess/sync/named_upgradable_mutex.hpp>


class named_upgradable_mutex {
public:
  // construct/copy/destruct
  named_upgradable_mutex(create_only_t, const char *,
                         const permissions & = permissions());
  named_upgradable_mutex(open_or_create_t, const char *,
                         const permissions & = permissions());
  named_upgradable_mutex(open_only_t, const char *);
  ~named_upgradable_mutex();

  // public member functions
  void lock();
  bool try_lock();
  bool timed_lock(const boost::posix_time::ptime &);
  void unlock();
  void lock_sharable();
  bool try_lock_sharable();
  bool timed_lock_sharable(const boost::posix_time::ptime &);
  void unlock_sharable();
  void lock_upgradable();
  bool try_lock_upgradable();
  bool timed_lock_upgradable(const boost::posix_time::ptime &);
  void unlock_upgradable();
  void unlock_and_lock_upgradable();
  void unlock_and_lock_sharable();
  void unlock_upgradable_and_lock_sharable();
  void unlock_upgradable_and_lock();
  bool try_unlock_upgradable_and_lock();
  bool timed_unlock_upgradable_and_lock(const boost::posix_time::ptime &);
  bool try_unlock_sharable_and_lock();
  bool try_unlock_sharable_and_lock_upgradable();

  // public static functions
  static bool remove(const char *);
};
```

Description

A upgradable mutex with a global name, so it can be found from different processes. This mutex can't be placed in shared memory, and each process should have it's own named upgradable mutex.

named_upgradable_mutex public construct/copy/destruct

1.
```
named_upgradable_mutex(create_only_t create_only, const char * name,
                       const permissions & perm = permissions());
```

Creates a global upgradable mutex with a name. If the upgradable mutex can't be created throws interprocess_exception

2.
```
named_upgradable_mutex(open_or_create_t open_or_create, const char * name,
                       const permissions & perm = permissions());
```

Opens or creates a global upgradable mutex with a name, and an initial count. If the upgradable mutex is created, this call is equivalent to named_upgradable_mutex(create_only_t, ...) If the upgradable mutex is already created, this call is equivalent to named_upgradable_mutex(open_only_t, ...).

3.
```
named_upgradable_mutex(open_only_t open_only, const char * name);
```

Opens a global upgradable mutex with a name if that upgradable mutex is previously. created. If it is not previously created this function throws interprocess_exception.

4.
```
~named_upgradable_mutex();
```

Destroys *this and indicates that the calling process is finished using the resource. The destructor function will deallocate any system resources allocated by the system for use by this process for this resource. The resource can still be opened again calling the open constructor overload. To erase the resource from the system use remove().

`named_upgradable_mutex` public member functions

1.
```
void lock();
```

Effects: The calling thread tries to obtain exclusive ownership of the mutex, and if another thread has exclusive, sharable or upgradable ownership of the mutex, it waits until it can obtain the ownership. Throws: interprocess_exception on error.

2.
```
bool try_lock();
```

Effects: The calling thread tries to acquire exclusive ownership of the mutex without waiting. If no other thread has exclusive, sharable or upgradable ownership of the mutex this succeeds. Returns: If it can acquire exclusive ownership immediately returns true. If it has to wait, returns false. Throws: interprocess_exception on error.

3.
```
bool timed_lock(const boost::posix_time::ptime & abs_time);
```

Effects: The calling thread tries to acquire exclusive ownership of the mutex waiting if necessary until no other thread has has exclusive, sharable or upgradable ownership of the mutex or abs_time is reached. Returns: If acquires exclusive ownership, returns true. Otherwise returns false. Throws: interprocess_exception on error.

4.
```
void unlock();
```

Precondition: The thread must have exclusive ownership of the mutex. Effects: The calling thread releases the exclusive ownership of the mutex. Throws: An exception derived from interprocess_exception on error.

5.
```
void lock_sharable();
```

Effects: The calling thread tries to obtain sharable ownership of the mutex, and if another thread has exclusive or upgradable ownership of the mutex, waits until it can obtain the ownership. Throws: interprocess_exception on error.

6.
```
bool try_lock_sharable();
```

Effects: The calling thread tries to acquire sharable ownership of the mutex without waiting. If no other thread has has exclusive or upgradable ownership of the mutex this succeeds. Returns: If it can acquire sharable ownership immediately returns true. If it has to wait, returns false. Throws: interprocess_exception on error.

7.
```
bool timed_lock_sharable(const boost::posix_time::ptime & abs_time);
```

Effects: The calling thread tries to acquire sharable ownership of the mutex waiting if necessary until no other thread has has exclusive or upgradable ownership of the mutex or abs_time is reached. Returns: If acquires sharable ownership, returns true. Otherwise returns false. Throws: interprocess_exception on error.

8.
```
void unlock_sharable();
```

Precondition: The thread must have sharable ownership of the mutex. Effects: The calling thread releases the sharable ownership of the mutex. Throws: An exception derived from interprocess_exception on error.

9.
```
void lock_upgradable();
```

Effects: The calling thread tries to obtain upgradable ownership of the mutex, and if another thread has exclusive or upgradable ownership of the mutex, waits until it can obtain the ownership. Throws: interprocess_exception on error.

10.
```
bool try_lock_upgradable();
```

Effects: The calling thread tries to acquire upgradable ownership of the mutex without waiting. If no other thread has has exclusive or upgradable ownership of the mutex this succeeds. Returns: If it can acquire upgradable ownership immediately returns true. If it has to wait, returns false. Throws: interprocess_exception on error.

11.
```
bool timed_lock_upgradable(const boost::posix_time::ptime & abs_time);
```

Effects: The calling thread tries to acquire upgradable ownership of the mutex waiting if necessary until no other thread has has exclusive or upgradable ownership of the mutex or abs_time is reached. Returns: If acquires upgradable ownership, returns true. Otherwise returns false. Throws: interprocess_exception on error.

12.
```
void unlock_upgradable();
```

Precondition: The thread must have upgradable ownership of the mutex. Effects: The calling thread releases the upgradable ownership of the mutex. Throws: An exception derived from interprocess_exception on error.

13.
```
void unlock_and_lock_upgradable();
```

Precondition: The thread must have exclusive ownership of the mutex. Effects: The thread atomically releases exclusive ownership and acquires upgradable ownership. This operation is non-blocking. Throws: An exception derived from interprocess_exception on error.

14.
```
void unlock_and_lock_sharable();
```

Precondition: The thread must have exclusive ownership of the mutex. Effects: The thread atomically releases exclusive ownership and acquires sharable ownership. This operation is non-blocking. Throws: An exception derived from interprocess_exception on error.

15.
```
void unlock_upgradable_and_lock_sharable();
```

Precondition: The thread must have upgradable ownership of the mutex. Effects: The thread atomically releases upgradable ownership and acquires sharable ownership. This operation is non-blocking. Throws: An exception derived from interprocess_exception on error.

16.
```
void unlock_upgradable_and_lock();
```

Precondition: The thread must have upgradable ownership of the mutex. Effects: The thread atomically releases upgradable ownership and acquires exclusive ownership. This operation will block until all threads with sharable ownership release it. Throws: An exception derived from interprocess_exception on error.

17.
```
bool try_unlock_upgradable_and_lock();
```

Precondition: The thread must have upgradable ownership of the mutex. Effects: The thread atomically releases upgradable ownership and tries to acquire exclusive ownership. This operation will fail if there are threads with sharable ownership, but it will maintain upgradable ownership. Returns: If acquires exclusive ownership, returns true. Otherwise returns false. Throws: An exception derived from interprocess_exception on error.

18.
```
bool timed_unlock_upgradable_and_lock(const boost::posix_time::ptime & abs_time);
```

Precondition: The thread must have upgradable ownership of the mutex. Effects: The thread atomically releases upgradable ownership and tries to acquire exclusive ownership, waiting if necessary until abs_time. This operation will fail if there are threads with sharable ownership or timeout reaches, but it will maintain upgradable ownership. Returns: If acquires exclusive ownership, returns true. Otherwise returns false. Throws: An exception derived from interprocess_exception on error.

19.
```
bool try_unlock_sharable_and_lock();
```

Precondition: The thread must have sharable ownership of the mutex. Effects: The thread atomically releases sharable ownership and tries to acquire exclusive ownership. This operation will fail if there are threads with sharable or upgradable ownership, but it will maintain sharable ownership. Returns: If acquires exclusive ownership, returns true. Otherwise returns false. Throws: An exception derived from interprocess_exception on error.

20.
```
bool try_unlock_sharable_and_lock_upgradable();
```

`named_upgradable_mutex` public static functions

1.
```
static bool remove(const char * name);
```

Erases a named upgradable mutex from the system. Returns false on error. Never throws.

Header <boost/interprocess/sync/null_mutex.hpp>

Describes null_mutex classes

```
namespace boost {
  namespace interprocess {
    class null_mutex;
  }
  namespace posix_time {
  }
}
```

Class null_mutex

boost::interprocess::null_mutex

Synopsis

```cpp
// In header: <boost/interprocess/sync/null_mutex.hpp>


class null_mutex {
public:
  // construct/copy/destruct
  null_mutex();
  ~null_mutex();

  // public member functions
  void lock();
  bool try_lock();
  bool timed_lock(const boost::posix_time::ptime &);
  void unlock();
  void lock_sharable();
  bool try_lock_sharable();
  bool timed_lock_sharable(const boost::posix_time::ptime &);
  void unlock_sharable();
  void lock_upgradable();
  bool try_lock_upgradable();
  bool timed_lock_upgradable(const boost::posix_time::ptime &);
  void unlock_upgradable();
  void unlock_and_lock_upgradable();
  void unlock_and_lock_sharable();
  void unlock_upgradable_and_lock_sharable();
  void unlock_upgradable_and_lock();
  bool try_unlock_upgradable_and_lock();
  bool timed_unlock_upgradable_and_lock(const boost::posix_time::ptime &);
  bool try_unlock_sharable_and_lock();
  bool try_unlock_sharable_and_lock_upgradable();
};
```

Description

Implements a mutex that simulates a mutex without doing any operation and simulates a successful operation.

null_mutex public construct/copy/destruct

1.
   ```cpp
   null_mutex();
   ```

   Constructor. Empty.

2.
   ```cpp
   ~null_mutex();
   ```

   Destructor. Empty.

null_mutex public member functions

1.
   ```cpp
   void lock();
   ```

   Simulates a mutex lock() operation. Empty function.

2.

```cpp
bool try_lock();
```

Simulates a mutex try_lock() operation. Equivalent to "return true;"

3.

```cpp
bool timed_lock(const boost::posix_time::ptime &);
```

Simulates a mutex timed_lock() operation. Equivalent to "return true;"

4.

```cpp
void unlock();
```

Simulates a mutex unlock() operation. Empty function.

5.

```cpp
void lock_sharable();
```

Simulates a mutex lock_sharable() operation. Empty function.

6.

```cpp
bool try_lock_sharable();
```

Simulates a mutex try_lock_sharable() operation. Equivalent to "return true;"

7.

```cpp
bool timed_lock_sharable(const boost::posix_time::ptime &);
```

Simulates a mutex timed_lock_sharable() operation. Equivalent to "return true;"

8.

```cpp
void unlock_sharable();
```

Simulates a mutex unlock_sharable() operation. Empty function.

9.

```cpp
void lock_upgradable();
```

Simulates a mutex lock_upgradable() operation. Empty function.

10.

```cpp
bool try_lock_upgradable();
```

Simulates a mutex try_lock_upgradable() operation. Equivalent to "return true;"

11.

```cpp
bool timed_lock_upgradable(const boost::posix_time::ptime &);
```

Simulates a mutex timed_lock_upgradable() operation. Equivalent to "return true;"

12.

```cpp
void unlock_upgradable();
```

Simulates a mutex unlock_upgradable() operation. Empty function.

13.

```cpp
void unlock_and_lock_upgradable();
```

Simulates unlock_and_lock_upgradable(). Empty function.

14.

```cpp
void unlock_and_lock_sharable();
```

Simulates unlock_and_lock_sharable(). Empty function.

15.
```
void unlock_upgradable_and_lock_sharable();
```

Simulates unlock_upgradable_and_lock_sharable(). Empty function.

16.
```
void unlock_upgradable_and_lock();
```

Simulates unlock_upgradable_and_lock(). Empty function.

17.
```
bool try_unlock_upgradable_and_lock();
```

Simulates try_unlock_upgradable_and_lock(). Equivalent to "return true;"

18.
```
bool timed_unlock_upgradable_and_lock(const boost::posix_time::ptime &);
```

Simulates timed_unlock_upgradable_and_lock(). Equivalent to "return true;"

19.
```
bool try_unlock_sharable_and_lock();
```

Simulates try_unlock_sharable_and_lock(). Equivalent to "return true;"

20.
```
bool try_unlock_sharable_and_lock_upgradable();
```

Simulates try_unlock_sharable_and_lock_upgradable(). Equivalent to "return true;"

Header <boost/interprocess/sync/scoped_lock.hpp>

Describes the scoped_lock class.

```
namespace boost {
  namespace interprocess {
    template<typename Mutex> class scoped_lock;
  }
}
```

Class template scoped_lock

boost::interprocess::scoped_lock

Synopsis

```cpp
// In header: <boost/interprocess/sync/scoped_lock.hpp>

template<typename Mutex>
class scoped_lock {
public:
  // types
  typedef Mutex mutex_type;

  // construct/copy/destruct
  scoped_lock();
  scoped_lock(mutex_type &);
  scoped_lock(mutex_type &, defer_lock_type);
  scoped_lock(mutex_type &, accept_ownership_type);
  scoped_lock(mutex_type &, try_to_lock_type);
  scoped_lock(mutex_type &, const boost::posix_time::ptime &);
  scoped_lock(scoped_lock &&);
  template<typename T> scoped_lock(upgradable_lock< T > &&, unspecified = 0);
  template<typename T>
    scoped_lock(upgradable_lock< T > &&, try_to_lock_type, unspecified = 0);
  template<typename T>
    scoped_lock(upgradable_lock< T > &&, boost::posix_time::ptime &,
                unspecified = 0);
  template<typename T>
    scoped_lock(sharable_lock< T > &&, try_to_lock_type, unspecified = 0);
  scoped_lock& operator=(scoped_lock &&);
  ~scoped_lock();

  // public member functions
  void lock();
  *bool try_lock();
  *bool timed_lock(const boost::posix_time::ptime &);
  *void unlock();
  bool owns() const;
  operator unspecified_bool_type() const;
  mutex_type * mutex() const;
  mutex_type * release();
  void swap(scoped_lock< mutex_type > &);
};
```

Description

scoped_lock is meant to carry out the tasks for locking, unlocking, try-locking and timed-locking (recursive or not) for the Mutex. The Mutex need not supply all of this functionality. If the client of scoped_lock<Mutex> does not use functionality which the Mutex does not supply, no harm is done. Mutex ownership transfer is supported through the syntax of move semantics. Ownership transfer is allowed both by construction and assignment. The scoped_lock does not support copy semantics. A compile time error results if copy construction or copy assignment is attempted. Mutex ownership can also be moved from an upgradable_lock and sharable_lock via constructor. In this role, scoped_lock shares the same functionality as a write_lock.

`scoped_lock` public construct/copy/destruct

1.
```cpp
scoped_lock();
```

Effects: Default constructs a scoped_lock. Postconditions: owns() == false and mutex() == 0.

2.
```
scoped_lock(mutex_type & m);
```

Effects: m.lock(). Postconditions: owns() == true and mutex() == &m. Notes: The constructor will take ownership of the mutex. If another thread already owns the mutex, this thread will block until the mutex is released. Whether or not this constructor handles recursive locking depends upon the mutex.

3.
```
scoped_lock(mutex_type & m, defer_lock_type);
```

Postconditions: owns() == false, and mutex() == &m. Notes: The constructor will not take ownership of the mutex. There is no effect required on the referenced mutex.

4.
```
scoped_lock(mutex_type & m, accept_ownership_type);
```

Postconditions: owns() == true, and mutex() == &m. Notes: The constructor will suppose that the mutex is already locked. There is no effect required on the referenced mutex.

5.
```
scoped_lock(mutex_type & m, try_to_lock_type);
```

Effects: m.try_lock(). Postconditions: mutex() == &m. owns() == the return value of the m.try_lock() executed within the constructor. Notes: The constructor will take ownership of the mutex if it can do so without waiting. Whether or not this constructor handles recursive locking depends upon the mutex. If the mutex_type does not support try_lock, this constructor will fail at compile time if instantiated, but otherwise have no effect.

6.
```
scoped_lock(mutex_type & m, const boost::posix_time::ptime & abs_time);
```

Effects: m.timed_lock(abs_time). Postconditions: mutex() == &m. owns() == the return value of the m.timed_lock(abs_time) executed within the constructor. Notes: The constructor will take ownership of the mutex if it can do it until abs_time is reached. Whether or not this constructor handles recursive locking depends upon the mutex. If the mutex_type does not support try_lock, this constructor will fail at compile time if instantiated, but otherwise have no effect.

7.
```
scoped_lock(scoped_lock && scop);
```

Postconditions: mutex() == the value scop.mutex() had before the constructor executes. s1.mutex() == 0. owns() == the value of scop.owns() before the constructor executes. scop.owns(). Notes: If the scop scoped_lock owns the mutex, ownership is moved to thisscoped_lock with no blocking. If the scop scoped_lock does not own the mutex, then neither will this scoped_lock. Only a moved scoped_lock's will match this signature. An non-moved scoped_lock can be moved with the expression: "boost::interprocess::move(lock);". This constructor does not alter the state of the mutex, only potentially who owns it.

8.
```
template<typename T>
  scoped_lock(upgradable_lock< T > && upgr, unspecified = 0);
```

Effects: If upgr.owns() then calls unlock_upgradable_and_lock() on the referenced mutex. upgr.release() is called. Postconditions: mutex() == the value upgr.mutex() had before the construction. upgr.mutex() == 0. owns() == upgr.owns() before the construction. upgr.owns() == false after the construction. Notes: If upgr is locked, this constructor will lock this scoped_lock while unlocking upgr. If upgr is unlocked, then this scoped_lock will be unlocked as well. Only a moved upgradable_lock's will match this signature. An non-moved upgradable_lock can be moved with the expression: "boost::interprocess::move(lock);" This constructor may block if other threads hold a sharable_lock on this mutex (sharable_lock's can share ownership with an upgradable_lock).

9.
```
template<typename T>
  scoped_lock(upgradable_lock< T > && upgr, try_to_lock_type, unspecified = 0);
```

Effects: If upgr.owns() then calls try_unlock_upgradable_and_lock() on the referenced mutex: a)if try_unlock_upgradable_and_lock() returns true then mutex() obtains the value from upgr.release() and owns() is set to true. b)if try_unlock_upgrad-

able_and_lock() returns false then upgr is unaffected and this scoped_lock construction as the same effects as a default construction. c)Else upgr.owns() is false. mutex() obtains the value from upgr.release() and owns() is set to false Notes: This construction will not block. It will try to obtain mutex ownership from upgr immediately, while changing the lock type from a "read lock" to a "write lock". If the "read lock" isn't held in the first place, the mutex merely changes type to an unlocked "write lock". If the "read lock" is held, then mutex transfer occurs only if it can do so in a non-blocking manner.

10.
```
template<typename T>
   scoped_lock(upgradable_lock< T > && upgr,
               boost::posix_time::ptime & abs_time, unspecified = 0);
```

Effects: If upgr.owns() then calls timed_unlock_upgradable_and_lock(abs_time) on the referenced mutex: a)if timed_unlock_upgradable_and_lock(abs_time) returns true then mutex() obtains the value from upgr.release() and owns() is set to true. b)if timed_unlock_upgradable_and_lock(abs_time) returns false then upgr is unaffected and this scoped_lock construction as the same effects as a default construction. c)Else upgr.owns() is false. mutex() obtains the value from upgr.release() and owns() is set to false Notes: This construction will not block. It will try to obtain mutex ownership from upgr immediately, while changing the lock type from a "read lock" to a "write lock". If the "read lock" isn't held in the first place, the mutex merely changes type to an unlocked "write lock". If the "read lock" is held, then mutex transfer occurs only if it can do so in a non-blocking manner.

11.
```
template<typename T>
   scoped_lock(sharable_lock< T > && shar, try_to_lock_type, unspecified = 0);
```

Effects: If shar.owns() then calls try_unlock_sharable_and_lock() on the referenced mutex. a)if try_unlock_sharable_and_lock() returns true then mutex() obtains the value from shar.release() and owns() is set to true. b)if try_unlock_sharable_and_lock() returns false then shar is unaffected and this scoped_lock construction has the same effects as a default construction. c)Else shar.owns() is false. mutex() obtains the value from shar.release() and owns() is set to false Notes: This construction will not block. It will try to obtain mutex ownership from shar immediately, while changing the lock type from a "read lock" to a "write lock". If the "read lock" isn't held in the first place, the mutex merely changes type to an unlocked "write lock". If the "read lock" is held, then mutex transfer occurs only if it can do so in a non-blocking manner.

12.
```
scoped_lock& operator=(scoped_lock && scop);
```

Effects: If owns() before the call, then unlock() is called on mutex(). this gets the state of scop and scop gets set to a default constructed state. Notes: With a recursive mutex it is possible that both this and scop own the same mutex before the assignment. In this case, this will own the mutex after the assignment (and scop will not), but the mutex's lock count will be decremented by one.

13.
```
~scoped_lock();
```

Effects: if (owns()) mp_mutex->unlock(). Notes: The destructor behavior ensures that the mutex lock is not leaked.

`scoped_lock` public member functions

1.
```
void lock();
```

Effects: If mutex() == 0 or if already locked, throws a lock_exception() exception. Calls lock() on the referenced mutex. Postconditions: owns() == true. Notes: The scoped_lock changes from a state of not owning the mutex, to owning the mutex, blocking if necessary.

2.
```
*bool try_lock();
```

Effects: If mutex() == 0 or if already locked, throws a lock_exception() exception. Calls try_lock() on the referenced mutex. Postconditions: owns() == the value returned from mutex()->try_lock(). Notes: The scoped_lock changes from a state of not owning the mutex, to owning the mutex, but only if blocking was not required. If the mutex_type does not support try_lock(), this function will fail at compile time if instantiated, but otherwise have no effect.

3.
```
*bool timed_lock(const boost::posix_time::ptime & abs_time);
```

Effects: If mutex() == 0 or if already locked, throws a lock_exception() exception. Calls timed_lock(abs_time) on the referenced mutex. Postconditions: owns() == the value returned from mutex()-> timed_lock(abs_time). Notes: The scoped_lock changes from a state of not owning the mutex, to owning the mutex, but only if it can obtain ownership by the specified time. If the mutex_type does not support timed_lock (), this function will fail at compile time if instantiated, but otherwise have no effect.

4.
```
*void unlock();
```

Effects: If mutex() == 0 or if not locked, throws a lock_exception() exception. Calls unlock() on the referenced mutex. Postconditions: owns() == false. Notes: The scoped_lock changes from a state of owning the mutex, to not owning the mutex.

5.
```
bool owns() const;
```

Effects: Returns true if this scoped_lock has acquired the referenced mutex.

6.
```
operator unspecified_bool_type() const;
```

Conversion to bool. Returns owns().

7.
```
mutex_type * mutex() const;
```

Effects: Returns a pointer to the referenced mutex, or 0 if there is no mutex to reference.

8.
```
mutex_type * release();
```

Effects: Returns a pointer to the referenced mutex, or 0 if there is no mutex to reference. Postconditions: mutex() == 0 and owns() == false.

9.
```
void swap(scoped_lock< mutex_type > & other);
```

Effects: Swaps state with moved lock. Throws: Nothing.

Header <boost/interprocess/sync/sharable_lock.hpp>

Describes the upgradable_lock class that serves to acquire the upgradable lock of a mutex.

```
namespace boost {
  namespace interprocess {
    template<typename SharableMutex> class sharable_lock;
  }
}
```

Class template sharable_lock

boost::interprocess::sharable_lock

Synopsis

```
// In header: <boost/interprocess/sync/sharable_lock.hpp>

template<typename SharableMutex>
class sharable_lock {
public:
  // types
  typedef SharableMutex mutex_type;

  // construct/copy/destruct
  sharable_lock();
  sharable_lock(mutex_type &);
  sharable_lock(mutex_type &, defer_lock_type);
  sharable_lock(mutex_type &, accept_ownership_type);
  sharable_lock(mutex_type &, try_to_lock_type);
  sharable_lock(mutex_type &, const boost::posix_time::ptime &);
  sharable_lock(sharable_lock< mutex_type > &&);
  template<typename T> sharable_lock(upgradable_lock< T > &&, unspecified = 0);
  template<typename T> sharable_lock(scoped_lock< T > &&, unspecified = 0);
  sharable_lock& operator=(sharable_lock< mutex_type > &&);
  ~sharable_lock();

  // public member functions
  void lock();
  bool try_lock();
  bool timed_lock(const boost::posix_time::ptime &);
  void unlock();
  bool owns() const;
  operator unspecified_bool_type() const;
  mutex_type * mutex() const;
  mutex_type * release();
  void swap(sharable_lock< mutex_type > &);
};
```

Description

sharable_lock is meant to carry out the tasks for sharable-locking (such as read-locking), unlocking, try-sharable-locking and timed-sharable-locking (recursive or not) for the Mutex. The Mutex need not supply all of this functionality. If the client of sharable_lock<Mutex> does not use functionality which the Mutex does not supply, no harm is done. Mutex ownership can be shared among sharable_locks, and a single upgradable_lock. sharable_lock does not support copy semantics. But sharable_lock supports ownership transfer from an sharable_lock, upgradable_lock and scoped_lock via transfer_lock syntax.

`sharable_lock` public construct/copy/destruct

1.
```
sharable_lock();
```

Effects: Default constructs a sharable_lock. Postconditions: owns() == false and mutex() == 0.

2.
```
sharable_lock(mutex_type & m);
```

Effects: m.lock_sharable(). Postconditions: owns() == true and mutex() == &m. Notes: The constructor will take sharable-ownership of the mutex. If another thread already owns the mutex with exclusive ownership (scoped_lock), this thread will block

until the mutex is released. If another thread owns the mutex with sharable or upgradable ownership, then no blocking will occur. Whether or not this constructor handles recursive locking depends upon the mutex.

3.
```
sharable_lock(mutex_type & m, defer_lock_type);
```

Postconditions: owns() == false, and mutex() == &m. Notes: The constructor will not take ownership of the mutex. There is no effect required on the referenced mutex.

4.
```
sharable_lock(mutex_type & m, accept_ownership_type);
```

Postconditions: owns() == true, and mutex() == &m. Notes: The constructor will suppose that the mutex is already sharable locked. There is no effect required on the referenced mutex.

5.
```
sharable_lock(mutex_type & m, try_to_lock_type);
```

Effects: m.try_lock_sharable() Postconditions: mutex() == &m. owns() == the return value of the m.try_lock_sharable() executed within the constructor. Notes: The constructor will take sharable-ownership of the mutex if it can do so without waiting. Whether or not this constructor handles recursive locking depends upon the mutex. If the mutex_type does not support try_lock_sharable, this constructor will fail at compile time if instantiated, but otherwise have no effect.

6.
```
sharable_lock(mutex_type & m, const boost::posix_time::ptime & abs_time);
```

Effects: m.timed_lock_sharable(abs_time) Postconditions: mutex() == &m. owns() == the return value of the m.timed_lock_sharable() executed within the constructor. Notes: The constructor will take sharable-ownership of the mutex if it can do so within the time specified. Whether or not this constructor handles recursive locking depends upon the mutex. If the mutex_type does not support timed_lock_sharable, this constructor will fail at compile time if instantiated, but otherwise have no effect.

7.
```
sharable_lock(sharable_lock< mutex_type > && upgr);
```

Postconditions: mutex() == upgr.mutex(). owns() == the value of upgr.owns() before the construction. upgr.owns() == false after the construction. Notes: If the upgr sharable_lock owns the mutex, ownership is moved to this sharable_lock with no blocking. If the upgr sharable_lock does not own the mutex, then neither will this sharable_lock. Only a moved sharable_lock's will match this signature. An non-moved sharable_lock can be moved with the expression: "boost::interprocess::move(lock);". This constructor does not alter the state of the mutex, only potentially who owns it.

8.
```
template<typename T>
   sharable_lock(upgradable_lock< T > && upgr, unspecified = 0);
```

Effects: If upgr.owns() then calls unlock_upgradable_and_lock_sharable() on the referenced mutex. Postconditions: mutex() == the value upgr.mutex() had before the construction. upgr.mutex() == 0 owns() == the value of upgr.owns() before construction. upgr.owns() == false after the construction. Notes: If upgr is locked, this constructor will lock this sharable_lock while unlocking upgr. Only a moved sharable_lock's will match this signature. An non-moved upgradable_lock can be moved with the expression: "boost::interprocess::move(lock);".

9.
```
template<typename T> sharable_lock(scoped_lock< T > && scop, unspecified = 0);
```

Effects: If scop.owns() then calls unlock_and_lock_sharable() on the referenced mutex. Postconditions: mutex() == the value scop.mutex() had before the construction. scop.mutex() == 0 owns() == scop.owns() before the constructor. After the construction, scop.owns() == false. Notes: If scop is locked, this constructor will transfer the exclusive ownership to a sharable-ownership of this sharable_lock. Only a moved scoped_lock's will match this signature. An non-moved scoped_lock can be moved with the expression: "boost::interprocess::move(lock);".

10.
```
sharable_lock& operator=(sharable_lock< mutex_type > && upgr);
```

Effects: If owns() before the call, then unlock_sharable() is called on mutex(). this gets the state of upgr and upgr gets set to a default constructed state. Notes: With a recursive mutex it is possible that both this and upgr own the mutex before the assignment. In this case, this will own the mutex after the assignment (and upgr will not), but the mutex's lock count will be decremented by one.

11.
```
~sharable_lock();
```

Effects: if (owns()) mp_mutex->unlock_sharable(). Notes: The destructor behavior ensures that the mutex lock is not leaked.

`sharable_lock` public member functions

1.
```
void lock();
```

Effects: If mutex() == 0 or already locked, throws a lock_exception() exception. Calls lock_sharable() on the referenced mutex. Postconditions: owns() == true. Notes: The sharable_lock changes from a state of not owning the mutex, to owning the mutex, blocking if necessary.

2.
```
bool try_lock();
```

Effects: If mutex() == 0 or already locked, throws a lock_exception() exception. Calls try_lock_sharable() on the referenced mutex. Postconditions: owns() == the value returned from mutex()->try_lock_sharable(). Notes: The sharable_lock changes from a state of not owning the mutex, to owning the mutex, but only if blocking was not required. If the mutex_type does not support try_lock_sharable(), this function will fail at compile time if instantiated, but otherwise have no effect.

3.
```
bool timed_lock(const boost::posix_time::ptime & abs_time);
```

Effects: If mutex() == 0 or already locked, throws a lock_exception() exception. Calls timed_lock_sharable(abs_time) on the referenced mutex. Postconditions: owns() == the value returned from mutex()->timed_lock_sharable(elps_time). Notes: The sharable_lock changes from a state of not owning the mutex, to owning the mutex, but only if it can obtain ownership within the specified time interval. If the mutex_type does not support timed_lock_sharable(), this function will fail at compile time if instantiated, but otherwise have no effect.

4.
```
void unlock();
```

Effects: If mutex() == 0 or not locked, throws a lock_exception() exception. Calls unlock_sharable() on the referenced mutex. Postconditions: owns() == false. Notes: The sharable_lock changes from a state of owning the mutex, to not owning the mutex.

5.
```
bool owns() const;
```

Effects: Returns true if this scoped_lock has acquired the referenced mutex.

6.
```
operator unspecified_bool_type() const;
```

Conversion to bool. Returns owns().

7.
```
mutex_type * mutex() const;
```

Effects: Returns a pointer to the referenced mutex, or 0 if there is no mutex to reference.

8.
```
mutex_type * release();
```

Effects: Returns a pointer to the referenced mutex, or 0 if there is no mutex to reference. Postconditions: mutex() == 0 and owns() == false.

9.
```
void swap(sharable_lock< mutex_type > & other);
```

Effects: Swaps state with moved lock. Throws: Nothing.

Header <boost/interprocess/sync/upgradable_lock.hpp>

Describes the upgradable_lock class that serves to acquire the upgradable lock of a mutex.

```
namespace boost {
  namespace interprocess {
    template<typename UpgradableMutex> class upgradable_lock;
  }
}
```

Class template upgradable_lock

boost::interprocess::upgradable_lock

Synopsis

```
// In header: <boost/interprocess/sync/upgradable_lock.hpp>

template<typename UpgradableMutex>
class upgradable_lock {
public:
  // types
  typedef UpgradableMutex mutex_type;

  // construct/copy/destruct
  upgradable_lock();
  upgradable_lock(mutex_type &);
  upgradable_lock(mutex_type &, defer_lock_type);
  upgradable_lock(mutex_type &, accept_ownership_type);
  upgradable_lock(mutex_type &, try_to_lock_type);
  upgradable_lock(mutex_type &, const boost::posix_time::ptime &);
  upgradable_lock(upgradable_lock< mutex_type > &&);
  template<typename T> upgradable_lock(scoped_lock< T > &&, unspecified = 0);
  template<typename T>
    upgradable_lock(sharable_lock< T > &&, try_to_lock_type, unspecified = 0);
  upgradable_lock& operator=(upgradable_lock &&);
  ~upgradable_lock();

  // public member functions
  void lock();
  bool try_lock();
  bool timed_lock(const boost::posix_time::ptime &);
  void unlock();
  bool owns() const;
  operator unspecified_bool_type() const;
  mutex_type * mutex() const;
  mutex_type * release();
  void swap(upgradable_lock< mutex_type > &);
};
```

Description

upgradable_lock is meant to carry out the tasks for read-locking, unlocking, try-read-locking and timed-read-locking (recursive or not) for the Mutex. Additionally the upgradable_lock can transfer ownership to a scoped_lock using transfer_lock syntax. The Mutex need not supply all of the functionality. If the client of upgradable_lock<Mutex> does not use functionality which the Mutex does not supply, no harm is done. Mutex ownership can be shared among read_locks, and a single upgradable_lock. upgradable_lock does not support copy semantics. However upgradable_lock supports ownership transfer from a upgradable_locks or scoped_locks via transfer_lock syntax.

`upgradable_lock` public construct/copy/destruct

1.
```
upgradable_lock();
```

Effects: Default constructs a upgradable_lock. Postconditions: owns() == false and mutex() == 0.

2.
```
upgradable_lock(mutex_type & m);
```

3.

```
upgradable_lock(mutex_type & m, defer_lock_type);
```

Postconditions: owns() == false, and mutex() == &m. Notes: The constructor will not take ownership of the mutex. There is no effect required on the referenced mutex.

4.

```
upgradable_lock(mutex_type & m, accept_ownership_type);
```

Postconditions: owns() == true, and mutex() == &m. Notes: The constructor will suppose that the mutex is already upgradable locked. There is no effect required on the referenced mutex.

5.

```
upgradable_lock(mutex_type & m, try_to_lock_type);
```

Effects: m.try_lock_upgradable(). Postconditions: mutex() == &m. owns() == the return value of the m.try_lock_upgradable() executed within the constructor. Notes: The constructor will take upgradable-ownership of the mutex if it can do so without waiting. Whether or not this constructor handles recursive locking depends upon the mutex. If the mutex_type does not support try_lock_upgradable, this constructor will fail at compile time if instantiated, but otherwise have no effect.

6.

```
upgradable_lock(mutex_type & m, const boost::posix_time::ptime & abs_time);
```

Effects: m.timed_lock_upgradable(abs_time) Postconditions: mutex() == &m. owns() == the return value of the m.timed_lock_upgradable() executed within the constructor. Notes: The constructor will take upgradable-ownership of the mutex if it can do so within the time specified. Whether or not this constructor handles recursive locking depends upon the mutex. If the mutex_type does not support timed_lock_upgradable, this constructor will fail at compile time if instantiated, but otherwise have no effect.

7.

```
upgradable_lock(upgradable_lock< mutex_type > && upgr);
```

Effects: No effects on the underlying mutex. Postconditions: mutex() == the value upgr.mutex() had before the construction. upgr.mutex() == 0. owns() == upgr.owns() before the construction. upgr.owns() == false. Notes: If upgr is locked, this constructor will lock this upgradable_lock while unlocking upgr. If upgr is unlocked, then this upgradable_lock will be unlocked as well. Only a moved upgradable_lock's will match this signature. An non-moved upgradable_lock can be moved with the expression: "boost::interprocess::move(lock);". This constructor does not alter the state of the mutex, only potentially who owns it.

8.

```
template<typename T>
  upgradable_lock(scoped_lock< T > && scop, unspecified = 0);
```

Effects: If scop.owns(), m_.unlock_and_lock_upgradable(). Postconditions: mutex() == the value scop.mutex() had before the construction. scop.mutex() == 0. owns() == scop.owns() before the constructor. After the construction, scop.owns() == false. Notes: If scop is locked, this constructor will transfer the exclusive-ownership to an upgradable-ownership of this upgradable_lock. Only a moved sharable_lock's will match this signature. An non-moved sharable_lock can be moved with the expression: "boost::interprocess::move(lock);".

9.

```
template<typename T>
  upgradable_lock(sharable_lock< T > && shar, try_to_lock_type,
                  unspecified = 0);
```

Effects: If shar.owns() then calls try_unlock_sharable_and_lock_upgradable() on the referenced mutex. a)if try_unlock_sharable_and_lock_upgradable() returns true then mutex() obtains the value from shar.release() and owns() is set to true. b)if try_unlock_sharable_and_lock_upgradable() returns false then shar is unaffected and this upgradable_lock construction has the same effects as a default construction. c)Else shar.owns() is false. mutex() obtains the value from shar.release() and owns() is set to false. Notes: This construction will not block. It will try to obtain mutex ownership from shar immediately, while changing the lock type from a "read lock" to an "upgradable lock". If the "read lock" isn't held in the first place, the mutex merely changes type to an unlocked "upgradable lock". If the "read lock" is held, then mutex transfer occurs only if it can do so in a non-blocking manner.

10.
```
upgradable_lock& operator=(upgradable_lock && upgr);
```

Effects: If owns(), then unlock_upgradable() is called on mutex(). this gets the state of upgr and upgr gets set to a default constructed state. Notes: With a recursive mutex it is possible that both this and upgr own the mutex before the assignment. In this case, this will own the mutex after the assignment (and upgr will not), but the mutex's upgradable lock count will be decremented by one.

11.
```
~upgradable_lock();
```

Effects: if (owns()) m_->unlock_upgradable(). Notes: The destructor behavior ensures that the mutex lock is not leaked.

`upgradable_lock` public member functions

1.
```
void lock();
```

Effects: If mutex() == 0 or if already locked, throws a lock_exception() exception. Calls lock_upgradable() on the referenced mutex. Postconditions: owns() == true. Notes: The sharable_lock changes from a state of not owning the mutex, to owning the mutex, blocking if necessary.

2.
```
bool try_lock();
```

Effects: If mutex() == 0 or if already locked, throws a lock_exception() exception. Calls try_lock_upgradable() on the referenced mutex. Postconditions: owns() == the value returned from mutex()->try_lock_upgradable(). Notes: The upgradable_lock changes from a state of not owning the mutex, to owning the mutex, but only if blocking was not required. If the mutex_type does not support try_lock_upgradable(), this function will fail at compile time if instantiated, but otherwise have no effect.

3.
```
bool timed_lock(const boost::posix_time::ptime & abs_time);
```

Effects: If mutex() == 0 or if already locked, throws a lock_exception() exception. Calls timed_lock_upgradable(abs_time) on the referenced mutex. Postconditions: owns() == the value returned from mutex()->timed_lock_upgradable(abs_time). Notes: The upgradable_lock changes from a state of not owning the mutex, to owning the mutex, but only if it can obtain ownership within the specified time. If the mutex_type does not support timed_lock_upgradable(abs_time), this function will fail at compile time if instantiated, but otherwise have no effect.

4.
```
void unlock();
```

Effects: If mutex() == 0 or if not locked, throws a lock_exception() exception. Calls unlock_upgradable() on the referenced mutex. Postconditions: owns() == false. Notes: The upgradable_lock changes from a state of owning the mutex, to not owning the mutex.

5.
```
bool owns() const;
```

Effects: Returns true if this scoped_lock has acquired the referenced mutex.

6.
```
operator unspecified_bool_type() const;
```

Conversion to bool. Returns owns().

7.
```
mutex_type * mutex() const;
```

Effects: Returns a pointer to the referenced mutex, or 0 if there is no mutex to reference.

8.
```
mutex_type * release();
```

Effects: Returns a pointer to the referenced mutex, or 0 if there is no mutex to reference. Postconditions: mutex() == 0 and owns() == false.

9.
```
void swap(upgradable_lock< mutex_type > & other);
```

Effects: Swaps state with moved lock. Throws: Nothing.

Header <boost/interprocess/windows_shared_memory.hpp>

Describes a class representing a native windows shared memory.

```
namespace boost {
  namespace interprocess {
    class windows_shared_memory;
  }
}
```

Class windows_shared_memory

boost::interprocess::windows_shared_memory

Synopsis

```cpp
// In header: <boost/interprocess/windows_shared_memory.hpp>


class windows_shared_memory {
public:
  // construct/copy/destruct
  windows_shared_memory();
  windows_shared_memory(create_only_t, const char *, mode_t, std::size_t,
                        const permissions & = permissions());
  windows_shared_memory(open_or_create_t, const char *, mode_t, std::size_t,
                        const permissions & = permissions());
  windows_shared_memory(open_only_t, const char *, mode_t);
  windows_shared_memory(windows_shared_memory &&);
  windows_shared_memory& operator=(windows_shared_memory &&);
  ~windows_shared_memory();

  // public member functions
  void swap(windows_shared_memory &);
  const char * get_name() const;
  mode_t get_mode() const;
  mapping_handle_t get_mapping_handle() const;
};
```

Description

A class that wraps the native Windows shared memory that is implemented as a file mapping of the paging file. Unlike shared_memory_object, windows_shared_memory has no kernel persistence and the shared memory is destroyed when all processes destroy all their windows_shared_memory objects and mapped regions for the same shared memory or the processes end/crash.

Warning: Windows native shared memory and interprocess portable shared memory (boost::interprocess::shared_memory_object) can't communicate between them.

`windows_shared_memory` public construct/copy/destruct

1.
```cpp
windows_shared_memory();
```

   Default constructor. Represents an empty windows_shared_memory.

2.
```cpp
windows_shared_memory(create_only_t, const char * name, mode_t mode,
                      std::size_t size,
                      const permissions & perm = permissions());
```

   Creates a new native shared memory with name "name" and mode "mode", with the access mode "mode". If the file previously exists, throws an error.

3.
```cpp
windows_shared_memory(open_or_create_t, const char * name, mode_t mode,
                      std::size_t size,
                      const permissions & perm = permissions());
```

   Tries to create a shared memory object with name "name" and mode "mode", with the access mode "mode". If the file previously exists, it tries to open it with mode "mode". Otherwise throws an error.

4.
```
windows_shared_memory(open_only_t, const char * name, mode_t mode);
```

Tries to open a shared memory object with name "name", with the access mode "mode". If the file does not previously exist, it throws an error.

5.
```
windows_shared_memory(windows_shared_memory && moved);
```

Moves the ownership of "moved"'s shared memory object to *this. After the call, "moved" does not represent any shared memory object. Does not throw

6.
```
windows_shared_memory& operator=(windows_shared_memory && moved);
```

Moves the ownership of "moved"'s shared memory to *this. After the call, "moved" does not represent any shared memory. Does not throw

7.
```
~windows_shared_memory();
```

Destroys *this. All mapped regions are still valid after destruction. When all mapped regions and windows_shared_memory objects referring the shared memory are destroyed, the operating system will destroy the shared memory.

`windows_shared_memory` public member functions

1.
```
void swap(windows_shared_memory & other);
```

Swaps to shared_memory_objects. Does not throw.

2.
```
const char * get_name() const;
```

Returns the name of the shared memory.

3.
```
mode_t get_mode() const;
```

Returns access mode.

4.
```
mapping_handle_t get_mapping_handle() const;
```

Returns the mapping handle. Never throws.

Header <boost/interprocess/xsi_key.hpp>

Describes a class representing a xsi key type.

```
namespace boost {
  namespace interprocess {
    class xsi_key;
  }
}
```

Class xsi_key

boost::interprocess::xsi_key

Synopsis

```
// In header: <boost/interprocess/xsi_key.hpp>


class xsi_key {
public:
  // construct/copy/destruct
  xsi_key();
  xsi_key(const char *, boost::uint8_t);

  // public member functions
  key_t get_key() const;
};
```

Description

A class that wraps XSI (System V) key_t type. This type calculates key_t from path and id using ftok or sets key to IPC_PRIVATE using the default constructor.

xsi_key public construct/copy/destruct

1.
```
xsi_key();
```

Default constructor. Represents a private xsi_key.

2.
```
xsi_key(const char * path, boost::uint8_t id);
```

Creates a new XSI shared memory with a key obtained from a call to ftok (with path "path" and id "id"), of size "size" and permissions "perm". If the shared memory previously exists, throws an error.

xsi_key public member functions

1.
```
key_t get_key() const;
```

Returns the internal key_t value.

Header <boost/interprocess/xsi_shared_memory.hpp>

Describes a class representing a native xsi shared memory.

```
namespace boost {
  namespace interprocess {
    class xsi_shared_memory;
  }
}
```

Class xsi_shared_memory

boost::interprocess::xsi_shared_memory

Synopsis

```
// In header: <boost/interprocess/xsi_shared_memory.hpp>


class xsi_shared_memory {
public:
  // construct/copy/destruct
  xsi_shared_memory();
  xsi_shared_memory(open_only_t, int);
  xsi_shared_memory(create_only_t, const xsi_key &, std::size_t,
                    const permissions & = permissions());
  xsi_shared_memory(open_or_create_t, const xsi_key &, std::size_t,
                    const permissions & = permissions());
  xsi_shared_memory(open_only_t, const xsi_key &);
  xsi_shared_memory(xsi_shared_memory &&);
  xsi_shared_memory& operator=(xsi_shared_memory &&);
  ~xsi_shared_memory();

  // public member functions
  void swap(xsi_shared_memory &);
  int get_shmid() const;
  mapping_handle_t get_mapping_handle() const;

  // public static functions
  static bool remove(int);
};
```

Description

A class that wraps XSI (System V) shared memory. Unlike shared_memory_object, xsi_shared_memory needs a valid xsi_key to identify a shared memory object.

Warning: XSI shared memory and interprocess portable shared memory (boost::interprocess::shared_memory_object) can't communicate between them.

`xsi_shared_memory` public construct/copy/destruct

1.
```
xsi_shared_memory();
```

Default constructor. Represents an empty xsi_shared_memory.

2.
```
xsi_shared_memory(open_only_t, int shmid);
```

Initializes *this with a shmid previously obtained (possibly from another process) This lower-level initializer allows shared memory mapping without having a key.

3.
```
xsi_shared_memory(create_only_t, const xsi_key & key, std::size_t size,
                  const permissions & perm = permissions());
```

Creates a new XSI shared memory from 'key', with size "size" and permissions "perm". If the shared memory previously exists, throws an error.

4.

```
xsi_shared_memory(open_or_create_t, const xsi_key & key, std::size_t size,
                  const permissions & perm = permissions());
```

Opens an existing shared memory with identifier 'key' or creates a new XSI shared memory from identifier 'key', with size "size" and permissions "perm".

5.

```
xsi_shared_memory(open_only_t, const xsi_key & key);
```

Tries to open a XSI shared memory with identifier 'key' If the shared memory does not previously exist, it throws an error.

6.

```
xsi_shared_memory(xsi_shared_memory && moved);
```

Moves the ownership of "moved"'s shared memory object to *this. After the call, "moved" does not represent any shared memory object. Does not throw

7.

```
xsi_shared_memory& operator=(xsi_shared_memory && moved);
```

Moves the ownership of "moved"'s shared memory to *this. After the call, "moved" does not represent any shared memory. Does not throw

8.

```
~xsi_shared_memory();
```

Destroys *this. The shared memory won't be destroyed, just this connection to it. Use remove() to destroy the shared memory.

`xsi_shared_memory` public member functions

1.

```
void swap(xsi_shared_memory & other);
```

Swaps two xsi_shared_memorys. Does not throw.

2.

```
int get_shmid() const;
```

Returns the shared memory ID that identifies the shared memory

3.

```
mapping_handle_t get_mapping_handle() const;
```

Returns the mapping handle. Never throws

`xsi_shared_memory` public static functions

1.

```
static bool remove(int shmid);
```

Erases the XSI shared memory object identified by shmid from the system. Returns false on error. Never throws