[Boost C++ Libraries](http://www.boost.org/)

...one of the most highly regarded and expertly designed C++ library projects in the world.— [Herb Sutter](http://www.gotw.ca/) and [Andrei Alexandrescu](http://en.wikipedia.org/wiki/Andrei_Alexandrescu), [C++ Coding Standards](http://safari.awprofessional.com/?XmlId=0321113586)

[This is the documentation for an old version of Boost. Click here to view this page for the latest version.](http://www.boost.org/doc/libs/release/doc/html/interprocess/quick_guide.html)

[PrevUp](http://www.boost.org/doc/libs/1_41_0/doc/html/interprocess.html)[Home](http://www.boost.org/doc/libs/1_41_0/doc/html/index.html)[Next](http://www.boost.org/doc/libs/1_41_0/doc/html/interprocess/some_basic_explanations.html)

**[Quick Guide for the Impatient](http://www.boost.org/doc/libs/1_41_0/doc/html/interprocess/quick_guide.html" \o "Quick Guide for the Impatient)**

[Using shared memory as a pool of unnamed memory blocks](http://www.boost.org/doc/libs/1_41_0/doc/html/interprocess/quick_guide.html#interprocess.quick_guide.qg_memory_pool)

[Creating named shared memory objects](http://www.boost.org/doc/libs/1_41_0/doc/html/interprocess/quick_guide.html#interprocess.quick_guide.qg_named_interprocess)

[Using an offset smart pointer for shared memory](http://www.boost.org/doc/libs/1_41_0/doc/html/interprocess/quick_guide.html#interprocess.quick_guide.qg_offset_ptr)

[Creating vectors in shared memory](http://www.boost.org/doc/libs/1_41_0/doc/html/interprocess/quick_guide.html#interprocess.quick_guide.qg_interprocess_container)

[Creating maps in shared memory](http://www.boost.org/doc/libs/1_41_0/doc/html/interprocess/quick_guide.html#interprocess.quick_guide.qg_interprocess_map)

**[Using shared memory as a pool of unnamed memory blocks](http://www.boost.org/doc/libs/1_41_0/doc/html/interprocess/quick_guide.html" \l "interprocess.quick_guide.qg_memory_pool" \o "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:

#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](http://www.boost.org/doc/libs/1_41_0/doc/html/interprocess/quick_guide.html" \l "interprocess.quick_guide.qg_named_interprocess" \o "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:

#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;

//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](http://www.boost.org/doc/libs/1_41_0/doc/html/interprocess/quick_guide.html" \l "interprocess.quick_guide.qg_offset_ptr" \o "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:

#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**](http://www.boost.org/doc/libs/1_41_0/doc/html/interprocess/quick_guide.html#interprocess.quick_guide.qg_interprocess_container)

**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](http://www.boost.org/doc/libs/1_41_0/doc/html/boost_interprocess_reference.html#boost.interprocess.managed_shared_memory) and it's easy to use. Just execute this example without arguments:

#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](http://www.boost.org/doc/libs/1_41_0/doc/html/interprocess/quick_guide.html" \l "interprocess.quick_guide.qg_interprocess_map" \o "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:

#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](http://www.boost.org/doc/libs/1_41_0/doc/html/interprocess/allocators_containers.html#interprocess.allocators_containers.containers_explained.containers_of_containers).

|  |  |
| --- | --- |
|  | Copyright © 2005 - 2008 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>) |

[PrevUp](http://www.boost.org/doc/libs/1_41_0/doc/html/interprocess.html)[Home](http://www.boost.org/doc/libs/1_41_0/doc/html/index.html)[Next](http://www.boost.org/doc/libs/1_41_0/doc/html/interprocess/some_basic_explanations.html)