*Ion Gaztañaga (igaztanaga at gmail dot com)*
*Alexander Zaitsev (zamazan4ik at tut dot by)*
*2017-07-21*

---

# Memory Mapped Files For C++

## Contents



## Introduction

Interprocess communication is an important missing part in the C++ standard library. This paper proposes some classes to fill two basic areas in interprocess communications: memory mapped files and shared memory.

Standard file mapping and shared memory utilities can offer a wide range of possibilities for many applications, to achieve data persistence, data cache and complex data serialization between processes.

## Memory Mapped Files

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, and the user can associate parts of the file with the address space of the process without mapping the entire file in the address space. 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++ memory utilities (STL containers, algorithms) in files.
- Shared memory between two or more applications.
- Allows efficient work with a large files, without loading 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. POSIX and Windows mapped file management is very similar, and highly portable.

# Shared Memory

POSIX defines a shared memory object as *"An object that represents memory that can be mapped concurrently into the address space of more than one process."*

Shared memory is similar to file mapping, and the user can map several regions of a shared memory object, just like with memory mapped files. In some operating systems, like Windows, shared memory is an special case of file mapping, where the file mapping object accesses memory backed by the system paging file. However, in Windows, the lifetime of this memory ends when the last process connected to the shared memory object closes connection or the application exits, so there is no data persistence. If an application creates shared memory, fills it with data and exits, the data is lost. This lifetime is known as **process** lifetime

In POSIX operating systems the shared memory lifetime is different since for semaphores, shared memory, and message queues it's mandatory that the object and its state (including data, if any) is preserved after the object is no longer referenced by any process. Persistence of an object does not imply that the state of the object is preserved after a system crash or reboot, but this can be achieved since shared memory objects can actually be implemented as mapped files of a permanent file system. The shared memory destruction happens with an explicit call to *unlink()*, which is similar to the file destruction mechanism. POSIX shared memory is required to have **kernel** lifetime (the object is explicitly destroyed or it's destroyed when the operating system reboots) or **filesystem** persistence (the shared memory object has the same lifetime as a file).

This lifetime difference is important to achieve portability. Many portable runtimes have tried to achieve perfect portability between Windows and POSIX shared memory but the author of this paper has not seen any satisfactory effort. Adding a reference count to POSIX shared memory is effective only as long as a process does not crash, something that it's very usual. Emulating POSIX behaviour in Windows using native shared memory is not possible since we could try to dump shared memory to a file to obtain persistence, but a process crash would avoid persistence. The only viable alternative is to use memory mapped files in Windows simulating shared memory, but avoiding file-memory synchronization as much as possible.

Many other named synchronization primitives (like named mutexes or semaphores) suffer the same lifetime portability problem. Automatic shared memory cleanup is useful in many contexts, like shared libraries or DLL-s communicating with other DLL-s or processes. Even when there is a crash, resources are automatically cleaned up by the operating systems. POSIX persistence is also useful when a launcher program can create and fill shared memory that another process can read or modify. Persistence also allows data recovery if a server process crashes. All the data is still in the shared memory, and the server can recover its state.

This paper proposes POSIX lifetime (kernel or filesystem lifetime) as a more portable solution, but has no strong opinion about this. The C++ committee should take into account the use cases of both approaches to decide which behaviour is better or if both options should be available, forcing the modification of both POSIX and Windows systems.

# Unifying Mapped Regions: `std::memory_mappable` and `std::mapped_region`

In both POSIX and Windows systems shared memory, memory mapping and other input-output device mapping mechanisms are similar. The memory mapping operation returns a handle that can be used to create a mapped region using *`mmap`* (POSIX) or *`MapViewOfFile`* (Windows) functions. This shows that a mapped region is independent from the memory mapping source, and this paper proposes a generic class representing a mapped region that can represent a shared memory, a mapped file or a mapped device region.

The `memory_mappable` class represents any resource that can be read and written using the address space of the process. It's an abstract class from which we can derive several mappable classes, like file mappings, shared memory objects, device mappings... Like the `iostream` framework, the derived classes must initialize `memory_mappable` in the constructor, calling `init` to store the mapping handle and the mapping mode. This information will be stored in the `memory_mappable` class and it will be used to created `mapped_region` objects:

## class memory_mappable sinopsys

```
namespace std {

class memory_mappable
{
   public:

   typedef /*implementation-defined*/ accessmode_t;
   typedef /*implementation-defined*/ mapping_handle_t;
   static const mapping_handle_t      invalid_handle;
   static const accessmode_t          read_only;
   static const accessmode_t          read_write;
   static const accessmode_t          invalid_mode;

   public:
   memory_mappable();

   protected:
   void init(mapping_handle_t handle, accessmode_t mode);

   public:
   memory_mappable(memory_mappable &&other);

   // Non-copyable
   memory_mappable(const memory_mappable &) = delete;

   // Non-assignable
   memory_mappable & operator=(const memory_mappable &) = delete;

   mapping_handle_t get_handle() const;

   accessmode_t get_mode() const;

   virtual ~memory_mappable() = 0;
};

}  //namespace std {
```

## class memory_mappable members

```
memory_mappable();
```

**Effects:** Default constructs the object.

**Postconditions:** this->get_handle() == *invalid_handle* & & this->get_mode() == *invalid_mode*.

**Throws:** An exception derived from `std::exception` on error.

```
void init(mapping_handle_t handle, accessmode_t mode);
```

**Effects:** Stores internally the mapping handle and the mode of the derived class. This function is intended to be called by the derived class once the derived class initialization has been completed.

**Posconditions:** this->get_handle() == *handle* && this->get_mode() == *mode*.

**Throws:** An exception derived from `std::exception` on error.

```
memory_mappable(memory_mappable &&other);
```

**Effects:** Moves resources from *other* to *this.

**Posconditions:** other.get_handle() == *invalid_handle* && other.get_mode() == *invalid_mode*.

**Throws:** Nothing.

```
mapping_handle_t get_handle() const;
```

**Effects:** Returns the handle registered by the init() function.

**Returns:** Returns the handle registered by the init() function. If init() has not been called, returns *invalid_handle*.

**Throws:** Nothing.

```
accessmode_t get_mode() const;
```

**Effects:** Returns the mode registered by the init() function.

**Returns:** Returns the handle registered by the init() function. If init() has not been called, returns *invalid_mode*.

**Throws:** Nothing.

```
virtual ~memory_mappable() = 0;
```

**Effects:** Destroys the object. The class does not free the handle registered in init(). This task should be performed by the derived class.

**Throws:** Nothing.

## class mapped_region synopsis

The `mapped_region` class represents a portion or region created from a `memory_mappable` object. Once a `mapped_region` object is constructed, a part of the file, shared memory or device is mapped in the address space of the calling process:

```
namespace std {

class mapped_region
{
```

```
   public:

   typedef /*implementation-defined*/ offset_t;
   typedef /*implementation-defined*/ accessmode_t;
   static const accessmode_t          invalid_mode;
   static const accessmode_t          read_only;
   static const accessmode_t          read_write;
   static const accessmode_t          copy_on_write;

   mapped_region();

   mapped_region( const memory_mappable & mappable
                , accessmode_t            mode
                , offset_t                mappable_offset
                , std::size_t             size = 0
                , const void *            address = 0);

   mapped_region(mapped_region &&other);

   // Non-copyable
   mapped_region(const mapped_region &) = delete;

   // Non-assignable
   mapped_region &operator=(const mapped_region &) = delete;

   std::size_t get_size() const;

   void*       get_address() const;

   offset_t    get_offset() const;

   accessmode_t get_mode() const;

   void flush(std::size_t region_offset = 0, std::size_t numbytes = 0);

   void swap(mapped_region &other);

   ~mapped_region();
};

}  //namespace std {
```

## class mapped_region members

```
mapped_region();
```

**Effects:** Default constructs a mapped region.

**Postconditions:** this->get_address() == 0 && this->get_size() == 0 && this->get_offset() == 0 && other.get_mode() == *invalid_mode*.

**Throws:** Nothing.

```
mapped_region(mapped_region &&other);
```

**Effects:** Constructs a mapped region taking ownership of the resources of another mapped_region.

**Postconditions:** The object is an default constructed state.

**Throws:** Nothing.

```
mapped_region( const memory_mappable & mappable
             , accessmode_t            mode
             , offset_t                mappable_offset = 0
```

```
   , std::size_t            size = 0
   , const void *           address = 0);
```

**Effects:** Creates a mapped region of the *mappable* memory mappable object. This *mappable* object can represent any mappable object that can be mapped using a generic mapping_handle_t (a file, a shared memory object or a input-output device). The *mappable* object can be inmediatelly destroyed after the constructor if no more mapped regions are needed from the *mappable* object.

A *mappable* object with *read_only* access should only be used to create *read_only* mapped_region objects. A *mappable* object with *read_write* access can create *read_only*, *read_write* or *copy_on_write* mapped_region objects.

When an object is created with *read_only* mode, writing to the memory range expressed by this object can terminate the process and never will change the *mappable* object.

When an object is created with *read_write* mode:

- If the user writes data to the memory range represented by this object, the modifications will be transferred to the *mappable resource* represented by the *mappable* object (a file, for example) when the implementation considers it adequate.
- If multiple processes map the same *mappable* resource, the modifications are propagated to the other processes, so the change will be visible in the memory regions referring to the same resource in other processes.

A *copy_on_write* mapped_region is an special *read_write* mapped_region:

- If the user writes data to the memory range represented by this object, the modifications will **not** be transferred to the *mappable* resource represented by the *mappable* object (a file, for example...).
- If multiple processes refer to the same underlying storage, the modifications are not propagated to the other processes.

The constructor will map a region of the *mappable* object starting in the offset *offset* of the *mappable* object (a file, or shared memory object), and the region's size will be *size*.

The user can request a mapping address in the process, using the *address* parameter. If *address* is 0, the implementation will choose the mapping address. If *size* is 0, the mapped_region's size will cover from *offset* until the end of the *mappable* object.

**Postconditions:** this->get_address() != 0 && this->get_size() != 0 && this->get_offset() == *offset*.

**Throws:** An exception derived from std::exception on any error, for example when the *mappable* object can't be mapped in the address passed in the constructor or when the mappable can't be mapped with the required access mode.

```
std::size_t get_size() const;
```

**Effects:** Returns the size of the mapped region. Never throws.

**Returns:** If default constructed or moved returns 0. Otherwise the size passed in the non-default constructor.

**Throws:** Nothing.

```
void *get_address() const;
```

**Effects:** Returns the address where the mappable has been mapped. If the non-default constructor's *address* argument was 0, this value will be chosen by the implementation.

**Returns:** If the object is default constructed or has been moved returns 0. Otherwise the address where the mappable object was mapped.

**Throws:** Nothing.

```
offset_t get_offset() const;
```

**Effects:** Returns the offset from the beginning of the mappable where the mapped region starts.

**Returns:** If default constructed or moved returns 0. Otherwise the offset passed in the non-default constructor.

**Throws:** Nothing.

```
accessmode_t get_mode() const;
```

**Effects:** Returns the mode of the mapped region.

**Returns:** If default constructed or moved returns invalid_mode. Otherwise the mode passed in the non-default constructor.

**Throws:** Nothing.

```
void flush(std::size_t region_offset = 0, std::size_t numbytes = 0);
```

**Effects:** If we call *mapping_address* to the result of the expression

```
static_cast<char*>(this->get_address())
```

the function flushes the range [*mapping_address* + region_offset, *mapping_address* + region_offset + numbytes) to the mappable object passed in the non-default constructor. If the object is default constructed, or moved, the function does nothing.

**Throws:** An error derived from `std::exception` on error.

```
void swap(mapped_region &other);
```

**Effects:** Swaps the resources contents of this object with the resources of *other*.

**Throws:** Nothing.

```
~mapped_region();
```

**Effects:** Destroys the object. Reading from and writing to the memory address range that the object was holding is now undefined and might crash the process.

**Throws:** Nothing.

The `mapped_region` class is constructed taking a reference to a `memory_mappable` class. This class contains the mapping handle needed to map a region of the mapping in the process address space. The `memory_mappable` object used in the constructor can be destroyed before destroying all the created `mapped_region` objects from it since both POSIX and Windows support this possibility. This simplifies mapping management, and there is no need to store the `memory_mappable` object after the

user has constructed all the needed `mapped_region` objects.

`mapped_region` is a single class representing a region of a file, shared memory or a device in the address space of the process, so the user can develop device independent code without the need of virtual functions.

`memory_mappable` and `mapped_region` are *non-copyable* and *non-assignable* but both are *movable*, so we can store them safely in STL containers, to create groups of mapped regions of files or shared memory objects.

# Mapping Files: `std::file_mapping`

This paper proposes a standard file mapping class derived from `memory_mappable`. The standard interface would allow just linking a file with the process, so that we can create several `mapped_region` objects from the file:

## class file_mapping synopsis

```
namespace std {

class file_mapping : public memory_mappable
{
   public:

   file_mapping(const filesystem::path& filename,
memory_mappable::accessmode_t mode);

   ~file_mapping();
};

}  //namespace std {
```

## class file_mapping members

```
file_mapping(const filesystem::path& filename,
memory_mappable::accessmode_t mode);
```

**Effects:** Allocates resources to so that the user can map parts of a file using mapped_regions. The constructor initializes the base `memory_mappable` class calling `memory_mappable::init().` with valid handle and mode values.

**Postcondition:** this->get_handle() != `memory_mappable::invalid_handle` && this->get_mode() != `memory_mappable::invalid_mode`.

**Throws:** Throws a exception derived from `std::exception` on error.

```
~file_mapping();
```

**Effects:** Closes the file mapping and frees resources. All mapped regions created from this object will be valid after the destruction.

**Throws:** Nothing.

## File Mapping Example

Using `file_mapping` and `mapped_region`, a file can be easily mapped in memory:

```
//Create a file_mapping object
std::file_mapping  mapping("/home/user/file",
std::memory_mappable::read_write);

//Create a mapped_region covering the whole file
std::mapped_region region (mapping, std::mapped_region::read_write);

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

//Set the whole file
std::memset(address, 0xFF, size);

//Make sure all the data is flushed to disk
region.flush();
```

# Shared Memory Mapping: `std::shared_memory_object`

C++ has file management functions, so that it can create, delete, read and write files. That's why the `file_mapping` class has no functions to create, open or modify files. However, there are no such functions for shared memory objects. This paper proposes two options:

- We can choose to implement new stream classes for shared memory (named, for example, `shmistream`, `shmostream` and `shmstream`)
- Add creation and connection functions to the shared memory object class.

This paper does not make any recommendation, but it's clear that adding new stream classes is more complicated than adding these functions to the mapping class. However, there is no doubt that a shared memory stream class can be very useful for several applications, like logging or tracing. This paper proposes an interface for a shared memory object with shared memory creation and connection interface that can be complementary to the stream classes:

## class shared_memory_object synopsis

```
namespace std {

namespace detail{

typedef /*implementation-defined*/ create_t;
typedef /*implementation-defined*/ open_t;
typedef /*implementation-defined*/ open_or_create_t;

}  //namespace detail{

extern std::detail::create_t create_only;
extern std::detail::create_t open_only;
extern std::detail::create_t open_or_create;

class shared_memory_object : public memory_mappable
{
   public:

   shared_memory_object(detail::create_t, const char *name,
memory_mappable::accessmode_t mode);

   shared_memory_object(detail::open_or_create_t, const char *name,
memory_mappable::accessmode_t mode);

   shared_memory_object(detail::open_t, const char *name,
memory_mappable::accessmode_t mode);

   void truncate(std::size_t new_size);
```

```
   static bool remove(const char *name);

   ~shared_memory_object();
};

}  //namespace std {
```

## class shared_memory_object members

```
shared_memory_object(detail::create_t, const char *name,
memory_mappable::accessmode_t mode);
```

**Effects:** Creates a shared memory object with name *name*, with the accessmode *mode*. If the shared memory object previously exists, the construction fails. If successful, the constructor initializes the base `memory_mappable` class calling `memory_mappable::init()` with valid handle and mode values.

**Postcondition:** this->get_handle() != `memory_mappable::invalid_handle` && this->get_mode() != `memory_mappable::invalid_mode`.

**Throws:** An exception derived from `std::exception` on error.

```
shared_memory_object(detail::open_or_create_t, const char *name,
memory_mappable::accessmode_t mode);
```

**Effects:** Tries to create a shared memory object with name *name*, with the accessmode *mode*. If it's previously created, tries to open the shared memory object. If successful, the constructor initializes the base `memory_mappable` class calling `memory_mappable::init()` with valid handle and mode values.

**Postcondition:** this->get_handle() != `memory_mappable::invalid_handle` && this->get_mode() != `memory_mappable::invalid_mode`.

**Throws:** An exception derived from `std::exception` on error.

```
shared_memory_object(detail::open_t, const char *name,
memory_mappable::accessmode_t mode);
```

**Effects:** Tries to open a shared memory object with name *name*, with the accessmode *mode*. If successful, the constructor initializes the base `memory_mappable` class calling `memory_mappable::init()` with valid handle and mode values.

**Postcondition:** this->get_handle() != `memory_mappable::invalid_handle` && this->get_mode() != `memory_mappable::invalid_mode`.

**Throws:** An exception derived from `std::exception` on error.

```
void truncate(std::size_t new_size);
```

**Effects:** Sets the new size of the shared memory. The object must have been opened in *read_write* mode.

**Throws:** An exception derived from `std::exception` on error.

```
static bool remove(const char *name);
```

**Effects:** Erases the shared memory object with name *name* from the system. This might fail if any process has a `mapped_region` object created from this object, or the shared memory object does not exist.

**Throws:** Nothing.

```
~shared_memory_object();
```

**Effects:** Destructor, closes the shared memory object. The shared memory object is not destroyed and can be again opened using another `shared_memory_object`. All mapped regions are still valid after destructor ends.

**Throws:** Nothing.

### Shared Memory Object Example

Once the shared memory object has been created, any process can map any part of the shared memory object in the process' address space, creating `mapped_region` objects from the shared memory object:

```
//Create a shared memory object
std::shared_memory_object shm(std::create_only, "name",
std::memory_mappable::read_write);

//Set the size of the shared memory
shm.truncate(1000);

//Create a mapped_region using the shared memory object
std::mapped_region region (shm, std::mapped_region::read_write);

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

//Write the whole memory
std::memset(address, 0xFF, size);
```

## Unifying With Iostreams

### Using basic_filebuf as file mapping source

The file mapping class can also offer file mapping possibilities using basic_fstream or basic_filebuf objects as constructor arguments. This way, libraries with access to a `basic_filebuf` object don't need to know the path of the file.

```
class file_mapping
{
   public:
   //...

   template
   file_mapping(const basic_filebuf<E, T> &file,
memory_mappable::accessmode_t mode);
};
```

In most systems, the implementation just needs to duplicate the operating system's file handle contained in the basic_filebuf class and also obtain access to the read-write capabilities of the file to see if the mapping capabilities requested by the user are allowed by `basic_filebuf`.

### Using `std::ios_base::openmode` flags

The library can also be simplified reusing `std::ios_base` bitmask values instead of creating new *`read_only`* and *`read_write`* bitmasks. However, `std::ios_base` is missing *`copy_on_write`* and there is no way to express *`create_only`*, *`open_or_create`* or *`open_only`* creation modes. Expanding `std::ios_base::openmode` to hold these new values is also a possibility.

### `basic_filebuf` is a `mappable` class

Another simplification would be to avoid the `file_mapping` class and make `basic_filebuf` a class derived from `memory_mappable`. This way a C++ file would be used to produce `mapped_region` objects:

```
//Open a file
std::fstream file("filename");

//Map the whole file
std::mapped_region region(*file.rdbuf(),
std::mapped_region::copy_on_write);
```

### Shared memory stream classes

If `basic_filebuf` is defined as a *mappable* object, we could create the shared memory equivalent of this class. This would unify file and shared memory mapping mechanism:

```
//Open shared memory object
std::shmstream shm("shmname");

//Map the whole shared memory object
std::mapped_region region(*shm.rdbuf(), std::mapped_region::copy_on_write);
```

As mentioned, we would still need additional `std::ios_base` flags to express creation possibilities like create only, open or create or open only, independently from the read-write mode.

## Portable Shared Memory Name

The portability of the shared memory object name is an old problem even in POSIX. In Posix, the name of the shared memory object conforms to the construction rules for a pathname. If the name begins with the slash, then processes opening a shared memory object with the same value of name refer to the same shared memory object. If the initial slash is not specified, or there are more slashes in the name, the effect is implementation defined. So the only portable shared memory has the following pattern: `/SharedMemoryName`

In Windows platforms the name can contain any character except the backlash character and the user must add some special prefix to share the object in the global or session namespace.

So the only portable name between both platforms is to use the `/SharedMemoryName` pattern, but this name conforms to a UNIX path rule for a file in the root directory. For an easier C++ usage, this paper proposes a portable name that conforms to a C++ variable name rule or a C++ reserved word:

- 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...*

POSIX implementation just needs to add an initial slash to the name. Windows implementation needs no change.

# Prior Art

The proposed shared memory and memory mapped files interface is based in the new interface proposed for the Boost.Interprocess library. This library is being used by several projects needing high performance interprocess communications.

# Conclusions

This paper is mainly presented to test if there is interest for standard shared memory and memory mapped classes in the C++ committee. The presented interface is a first approach to the solution and by no means a definitive proposal.

Shared memory and memory mapped files are basic building blocks for interprocess communication for advanced interprocess communications, like message queues or named fifos. This paper proposes the standardization of shared memory and memory mapped files present in nearly all UNIX and Windows operating systems presenting a common implementable subset of features.

Memory mapped files can be also useful for a single process environment in order to obtain data persistence. However, shared memory and memory mapped files for interprocess communications need process synchronization mechanisms. These should follow the same interface as their thread-synchronization counterparts, so several utilities like scoped locks could be reused with process-shared synchronization utilities.

# Acknowledgments

Thanks to all Boost mailing list members for their suggestions and the thorough Shmem library review. Special thanks to Andrew Bromage, who suggested separating file mapping and mapped region concepts.