Code Reading -- boost

2017011307 张晨

1 Introduction to Boost

1.1 What is Boost

Boost is a set of libraries for the C++ programming language that provide support for tasks and structures such as linear algebra, pseudorandom number generation, multithreading, image processing, regular expressions and unit testing.

These libraries are aimed at a wide range of C++ users and application domains. In order to ensure efficiency and flexibility, Boost makes extensive use of templates. Boost has been a source of extensive work and research into generic programming and metaprogramming in C++. The detail will be discussed in the follow sections.

Boost libraries uses a lot of inline functions and templates. Most Boost libraries are header based, stored in header files with .hpp suffix (.h+.cpp). Therefore, they do not need to be built in advance of their use.

1.2 An Overview of Libraries in Boost

As Boost is huge, I will only introduce some popular libraries in boost.

- **GIL (Generic Image Library)**

  GIL abstracts image representations from algorithms and allows writing code that can work on a variety of images with performance similar to hand-writing for a specific image type.

- **Graph**

  Graph provides two data structures that can store graphs($adjancency\_list$ and $adjacency\_matrix$) and the implementation of some classic graph algorithm based on the data structures.

- **MPL**

  The Boost.MPL library is a general-purpose, high-level C++ template metaprogramming framework of compile-time algorithms, sequences and metafunctions. It provides a conceptual foundation and an extensive set of powerful and coherent tools that make doing explict metaprogramming in C++ as easy and enjoyable as possible within the current language

- **pool**

  Pool allocation is a memory allocation scheme that is very fast, but limited in its usage. It is suggested to be used when more control over memory is needed and when there is a lot of allocation and deallocation of small objects.

- **python**

  The Boost Python Library is a framework for interfacing Python and C++. It allows you to quickly and seamlessly expose C++ classes functions and objects to Python, and vice-versa, using no special tools-- just your C++ compiler.

- **regex**

  Regular expressions(Re) are a form of pattern-matching that are often used in text processing. $boost :: regex$ provides support for Re in C++, which includes match, search and replace operations. It has been a member of standard C++0x libraries.

- **smart_ptr**

  Smart pointers are pointers that behave much like built-in C++ pointers except that they can automatically delete the object they point to at the appropriate time, which can avoid memory leak. The Smart Pointer Library implements six kinds of smart pointers-- $shared\_ptr$, $weak\_ptr$, $scoped\_ptr$, $scoped\_array$, $shared\_array$, and $intrusive\_ptr$.

  $shared\_ptr$ and $weaked\_ptr$ are part of the C++11 standard.

- **timer**

  The Boost Timer Library can answer the question "How long does my C++ code take to run" and does so portably with as little as one #include and one additional line of code.

- **thread**

  The Boost Thread Library enables the use of multiple threads of execution with shared data in portable C++ code. It provides classes and functions for managing the threads themselves, along with others for synchronizing data between the threads or providing separate copies of data specific to individual threads.

- **uBLAS (Boost Linear Algebra Library)**

  uBLAS provides matrix and vector classes as well as basic linear algebra routines. Several dense, packed and sparse storage schemes are supported.

1.3 The Installation of Boost

The procedure of installing Boost on Windows will be discussed in this section.

1.3.1 Get Boost

Download the zip file of latest version of boost via the link https://www.boost.org/users/download/. Then unpack it. The zip file with .7z suffix is recommended as it is smaller.

1.3.2 Build Boost

The good news is that, there's nothing to be built as most Boost Libraries are header-only.

The only Boost Libraries that MUST be built seperately are:

$Boost.Chrono$, $Boost.Context$, $Boost.Filesystem$, $Boost.GraphParallel$, $Boost.IOStreams$, $Boost.Locale$, $Boost.Log$, $Boost.MPI$, $Boost.ProgramOptions$, $Boost.Python$, $Boost.Regex$, $Boost.Serialization$, $Boost.Signals$, $Boost.System$, $Boost.Thread$, $Boost.Timer$, $Boost.Wave$

1.3.3 A Simple Program Using Boost

```
#include <boost/lambda/lambda.hpp>
#include <iostream>
#include <iterator>
#include <algorithm>

int main()
{
    using namespace boost::lambda;
    typedef std::istream_iterator<int> in;

    std::for_each(
        in(std::cin), in(), std::cout << (_1 * 3) << " " );
}
```

This program reads an arbitrary number of integers from standard input, uses $boost :: lambda$ to multiply each number by 3 times and print it to the standard output.

Assume that these codes are stored in a file called $example.cpp$. Both IDE and command line can be used to build this code.

- **Build from the codeblocks IDE**
  - create an empty project and add $example.cpp$ into the project.
  - click `build->build options->Search directories` and add the address of Boost library(which include a subfolder called *boost*)
  - type F9 to compile and run the project
- **Build via the command line**
  - Assume that the address of Boost library is *D:\boost_1_67_0*
  - type the following command to compile and run the source code

```
g++ example.cpp -o example.exe -std=c++0x -I D:\boost_1_67_0 &&
example.exe
```

1.4 The Size of Boost

There are 13364 source files in boost.

2 Design Patterns

Boost uses almost all the classic design patterns. It even encapsulates some design patterns in order to make the use of them easier.

2.1 Singleton

$boost.serialization$ encapsulates a class that can be used to generate singletons - $singleton$. It is defined in namespace $boost::serialization$ and if you want to use it, you need to include the header file <boost/serialization/singleton.hpp> .

```
#include <boost/serialization/singleton.hpp>
using boost::serialization::singleton;
```

$singleton$ is a template class and there are two ways to use it.

```
struct Point
{
    ...
    void print();
}

int main()
{
    typedef singleton<Point> s_Point;
    s_Point::get_const_instance().print(); // only use when print is marked
as const
    s_Point::get_mutable_instance().print();
    return 0;
}
```

It is easy to create a singleton of most class in this way and the singleton has no effect on the origin class (The creation of non-singleton objects of the origin class is still permitted.)

Another way is to use public inheritance.

```
struct s_Point : public singleton<s_Point>
{
    ...
    void print();
}

int main()
{
    s_Point::get_const_instance().print(); // only use when print is marked
as const
    s_Point::get_mutable_instance().print();
    return 0;
}
```

In this way, the class $point$ becomes a complete singleton and the creation of other $point$ is forbidden.

Singleton is implemented as follows.

```
template <class T>
class singleton : public singleton_module
{
private:
    static T & m_instance;
    static void use(T const *) {}
    static T & get_instance() {
        class singleton_wrapper : public T {};
        static singleton_wrapper *t = 0;
        BOOST_ASSERT(! is_destroyed());
        use(& m_instance);
        if (!t)
            t = new singleton_wrapper;
        return static_cast<T &>(*t);
    }
    static bool & get_is_destroyed(){
        static bool is_destroyed;
        return is_destroyed;
    }

public:

    BOOST_DLLEXPORT static T & get_mutable_instance(){
```

```
        BOOST_ASSERT(! is_locked());
        return get_instance();
    }
    BOOST_DLLEXPORT static const T & get_const_instance(){
        return get_instance();
    }
    BOOST_DLLEXPORT static bool is_destroyed(){
        return get_is_destroyed();
    }
    BOOST_DLLEXPORT singleton(){
        get_is_destroyed() = false;
    }
    BOOST_DLLEXPORT ~singleton() {
        if (!get_is_destroyed()) {
            delete &(get_instance());
        }
        get_is_destroyed() = true;
    }
};

template<class T>
T & singleton< T >::m_instance = singleton< T >::get_instance();
```

The constructor of *singleton* is thrown in the above codes and the copy constructor and the operator = is thrown in the base class of its base class -- *noncopyable*.

singleton_modual is the base class of *singleton*, it provides a mechanism to detect when a non-const function is called, which can make debugging easier.

```
class BOOST_SYMBOL_VISIBLE singleton_module :
    public boost::noncopyable
{
private:
    BOOST_DLLEXPORT static bool & get_lock() BOOST_USED {
        static bool lock = false;
        return lock;
    }

public:

    BOOST_DLLEXPORT static void lock(){
```

```
        get_lock() = true;
    }
    BOOST_DLLEXPORT static void unlock(){
        get_lock() = false;
    }
    BOOST_DLLEXPORT static bool is_locked(){
        return get_lock();
    }
};
```

$boost :: noncopyable$ is the base class of $singleton\_model$ and the function of it is to prevent copy of the derived classes- by throwing copy constructor and operator =.

```
class noncopyable
{
protected:
  noncopyable() {}
  ~noncopyable() {}
private:
  noncopyable(const noncopyable&);
  const noncopyable& operator=(const noncopyable&);
};
```

2.2 Proxy

$boost.smart\_ptr$ is a classic application of proxy. Smart pointers, such as $scoped\_ptr$, $shared\_ptr$ wrap the original pointers and add other actions of the call of pointers. Reference counting can be implement by the proxy of pointers. The detail of reference counting will be shown in section 3.

$scoped\_ptr$ is a simple smart pointer which is similar to $auto\_ptr$. It stores a pointer to a dynamically allocated object(created by the expression new). The object pointed to is guaranteed to be deleted, either on destruction of the $scoped\_ptr$, or via an explicit $reset$.

scoped_ptr is a simple solution for simple needs. It supplies a basic "resource acquisition is initialization" facility, without shared-ownership or transfer-of-ownership semantics. Both its name and enforcement of semantics (by being noncopyable) signal its intent to retain ownership solely within the current scope. Because it is noncopyable, it is safer than *shared_ptr* for pointers which should not be copied.

Because *scoped_ptr* is simple, in its usual implementation every operation is as fast as for a built-in pointer and it has no more space overhead that a built-in pointer.

```
template <class T>
class scoped_ptr {
    private:
        T *px;
        scoped_ptr(scoped_ptr const &);
        scoped_ptr & operator = (scoped_ptr const &);
    public:
        explicit scoped_ptr(T *p=0);
        ~scoped_ptr();

        void reset(T *p=0);

        T& operator *() const;
        T* operator ->() const;
        T* get() const;

        operator unspecified-bool-type() const;
        void swap(scoped_ptr &b);
}
```

scoped_ptr overloads the operator * and -> to simulate the build-in pointer and the expression `BOOST_ASSERT( px != 0 );` makes the use of * and -> safer. This is a proxy.

```
T & operator*() const BOOST_SP_NOEXCEPT_WITH_ASSERT
{
    BOOST_ASSERT( px != 0 );
    return *px;
}

T * operator->() const BOOST_SP_NOEXCEPT_WITH_ASSERT
{
     BOOST_ASSERT( px != 0 );
     return px;
}
```

2.3 Factory Method

$boost.xpressive$ is a powerful library that provides support for the matching of regular expressions. $basic\_regex$ is one of the core classes in $xpressive$ and it uses factory method.

The abstract of the class is as follows.

```
template<typename BidiIter>
struct basic_regex
{
    basic_regex();
    basic_regex(basic_regex<BidiIter> const &);
    template<typename Expr> basic_regex(Expr const &);

    std::size_t mark_count() const;
    regex_id_type regex_id() const;
    void swap(basic_regex<BidiIter> &);

    template<typename InputRange>
    static basic_regex<BidiIter> compile(InputRange const &pat, flag_type
flags = regex_constants::ECMAScript);
};
```

$basic\_regex$ is the base class of regular expressions and there are two frequently-used $typedef$ -- $sregex$ and $cregex$, which are defined as follows

```
typedef basic_regex<std::string::const_iterator> sregex;
typedef basic_regex<char const *> cregex;
```

$sregex$ is used to manipulate strings of type $std :: string$, and $cregex$ is used to manipulate $char$ arrays.

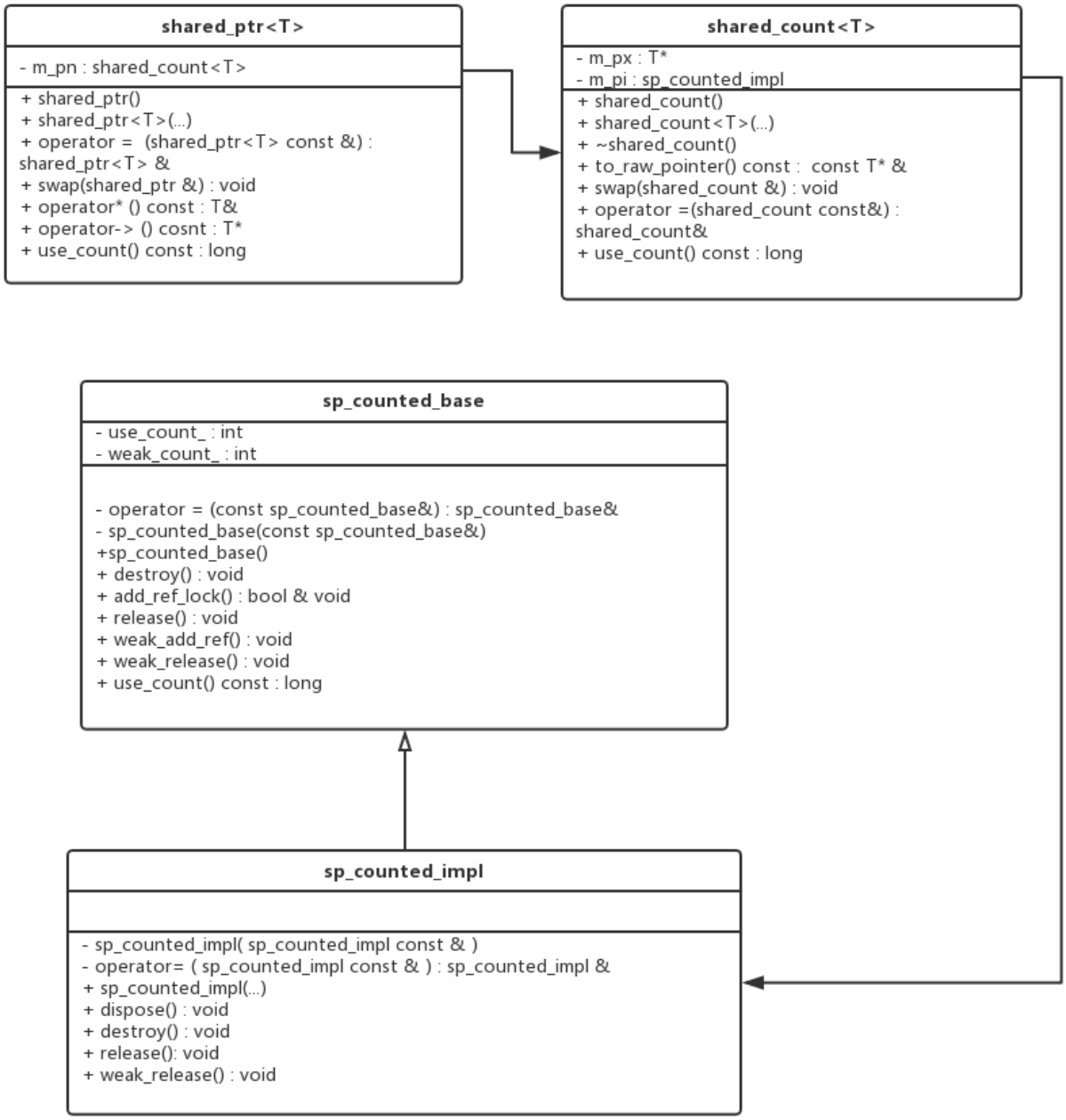
The member function `compile` can generate an object of type `basic_regex<BitiIter>`. The actual type of the object is determined by the template parameter `BitiIter`.

Though it uses template to implement compile-time polymorphism,its key principle is similar to classic factory method, which is based on virtual functions.

3 Reference Counting

Reference counting is a good way to determine when the object should be deleted -- when there is no pointer points to it and there is no reference. $shared\_ptr$, defined in $smart\_ptr$ library, is one of the best implementations of smart pointer based on reference counting.

$shared\_ptr$ 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.



There are several constructors in $shared\_ptr$

- `shared_ptr()` creates a `shared_ptr` with a null pointer `m_pn`
- `shared_ptr(shared_ptr<T> &r)` is a copy constructor. It gets the address from another `shared_ptr` and share the object it pointed to with r. This constructor will increase `m_pn.m_pi` (the counter) by 1.
- `shared_ptr(shared_ptr<T> &&r)` is a move constructor. It gets the address from another `shared_ptr` without increasing `m_pn.m_pi` and r becomes a null pointer.
- `shared_ptr(std::auto_ptr<Y> & r)` is a constructor from `std::auto_ptr`. It gets the address from the `auto_ptr` and set `m_pn.m_pi` to 1. After that r becomes a null pointer.

  In addition, the operator = is overloaded to make sure that the copy assignment is correct.

```
shared_ptr & operator=(shared_ptr && r) noexcept;
template<class Y> shared_ptr & operator=(shared_ptr<Y> && r) noexcept;
template<class Y> shared_ptr & operator=(std::auto_ptr<Y> && r);
template<class Y, class D> shared_ptr & operator=(std::unique_ptr<Y, D> &&
r);
```

WARNING

Because the implementation uses reference counting, cycles of shared_ptr instances will not be reclaimed. Here is one example.

```
class parent;
class child;

struct parent {
	shared_ptr<child> children;
};

struct child {
	shared_ptr<parent> parent;
};

int main()

{
```

```
    shared_ptr<parent> father(new parent());
    shared_ptr<child> son(new child);
    father->children = son;
    son->parent = father;
    return 0;
}
```

Before the end of the program, the reference counting of father and son will always be 2. That is to say, the destructor of *father and *son will not be called and memory leak appears. One way of solving the problem is to use $weak\_ptr$ to "break circles". It is defined in $smart\_ptr$ library too.

4 Introduction to boost.pool

4.1 What is boost.pool

Pool allocation is a memory allocation scheme that is very fast. Pools gives the users more control over how memory is used in their program. For example, you could have a situation where you want to allocate a bunch of small objects at one point, and then reach a point in your program where none of them are needed any more. Using pool interfaces, you can choose to run their destructors or just drop them off into oblivion; the pool interface will guarantee that there are no system memory leaks.

Pools are generally used when there is a lot of allocation and deallocation of small objects. Another common usage is the situation above, where many objects may be dropped out of memory. In general, use Pools when a more efficient way to do unusual memory control is needed.

There are three different pools in boost.pool - ls - `pool`, `Object_pool`, `Singleton_pool`. They will be introduced seperately in section 4.1.1-4.1.3.

4.1.1 pool

The `pool` interface is a simple Object Usage interface with Null Return. It is a fast memory allocator, and guarantees proper alignment of all allocated chunks. `pool.hpp` provides two `UserAllocator` classes and a template class `pool`, which extend and generalize the framework provided by the simple segregated storage solution.

Synposis

```
struct default_user_allocator_new_delete
{
  typedef std::size_t size_type;
  typedef std::ptrdiff_t difference_type;

  static char * malloc(const size_type bytes)
  { return new (std::nothrow) char[bytes]; }

  static void free(char * const block)
```

```
  { delete [] block; }
};

struct default_user_allocator_malloc_free
{
  typedef std::size_t size_type;
  typedef std::ptrdiff_t difference_type;

  static char * malloc(const size_type bytes)
  { return reinterpret_cast<char *>(std::malloc(bytes)); }
  static void free(char * const block)
  { std::free(block); }
};

template <typename UserAllocator = default_user_allocator_new_delete>
class pool
{
  private:
    pool(const pool &);
    void operator=(const pool &);

  public:
    typedef UserAllocator user_allocator;
    typedef typename UserAllocator::size_type size_type;
    typedef typename UserAllocator::difference_type difference_type;

    explicit pool(size_type requested_size);
    ~pool();

    bool release_memory();
    bool purge_memory();

    bool is_from(void * chunk) const;
    size_type get_requested_size() const;

    void * malloc();
    void * ordered_malloc();
    void * ordered_malloc(size_type n);

    void free(void * chunk);
    void ordered_free(void * chunk);
    void free(void * chunks, size_type n);
```

```
    void ordered_free(void * chunks, size_type n);
};
```

One example of `pool` is as follows.

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

4.1.2 Objected_pool

The template class `objected_pool` interface is an Object Usage interface with Null Return, too. However, it is aware of the type of the object for which it is allocation chunks. On destruction, any chunks that have been allocated from that `object_pool` will have their destructors called.

`objected_pool` is defined in `object_pool.hpp`.

Synopsis

```
template <typename ElementType, typename UserAllocator =
default_user_allocator_new_delete>
class object_pool
{
  private:
    object_pool(const object_pool &);
    void operator=(const object_pool &);

  public:
    typedef ElementType element_type;
    typedef UserAllocator user_allocator;

    typedef typename pool<UserAllocator>::size_type size_type;
```

```
    typedef typename pool<UserAllocator>::difference_type difference_type;

    object_pool();
    ~object_pool();

    element_type * malloc();
    void free(element_type * p);
    bool is_from(element_type * p) const;

    element_type * construct();
    // other construct() functions
    void destroy(element_type * p);
};
```

There are two template parameters in this template class.

ElementType is the type of object to allocate/deallocate. It must have a non-throwing destructor.

UserAllocator defines the method that the underlying Pool will use to allocate memory from the system. Default is `default_user_allocator_new_delete`.

Example

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

4.1.3 Singleton_pool

The `singleton_pool` interface at `singleton_pool.hpp` is a Singleton Usage interface with Null Return. It is just the same as the pool interface but with Singleton Usage instead.

Synopsis

```
template <typename Tag, unsigned RequestedSize,
    typename UserAllocator = default_user_allocator_new_delete>
struct singleton_pool
{
  public:
    typedef Tag tag;
    typedef UserAllocator user_allocator;
    typedef typename pool<UserAllocator>::size_type size_type;
    typedef typename pool<UserAllocator>::difference_type difference_type;

    static const unsigned requested_size = RequestedSize;

  private:
    static pool<size_type> p; // exposition only!

    singleton_pool();

  public:
    static bool is_from(void * ptr);

    static void * malloc();
    static void * ordered_malloc();
    static void * ordered_malloc(size_type n);

    static void free(void * ptr);
    static void ordered_free(void * ptr);
    static void free(void * ptr, std::size_t n);
    static void ordered_free(void * ptr, size_type n);

    static bool release_memory();
    static bool purge_memory();
};
```

There is three template parameters.

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

$RequestedSize$ is the size of memory chunks to allocate. This is passed as a constructor parameter to the underlying pool. Must be greater than 0.

$UserAllocator$ defines the method that the underlying pool will use to allocate memory from the system.

Example

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

4.2 Principle of pool

4.2.1 PODptr

`PODptr` holds the location and size of a memory block allocated from the system. Each memory block is split logically into three sections:

- **Chunk area**. This section may be different sizes. `PODptr` does not care what the size of the chunks is, but it does care (and keep track of) the total size of the chunk area.
- **Next pointer**. This section is always the same size for a given `SizeType`. It holds a pointer to the location of the next memory block in the memory block list, or 0 if there is no such block.

  > However, as the memory of a pointer is related to the operating system, the size of the pointer is in fact defined as `lcm(sizeof(SizeType), sizeof(void *))` .

- **Next size**. This section is always the same size for a given `SizeType`. It holds the size of the next memory block in the memory block list.

The complete definition `PODptr` can be found in `boost/pool/pool.hpp`. Here is a simplified definition which erases all member functions and holds all member objects.

```
class PODptr
{
    char * ptr;
    SizeType sz;
}
```

Member functions are provided to get the address of the chunk area and the address of the next pointer area.

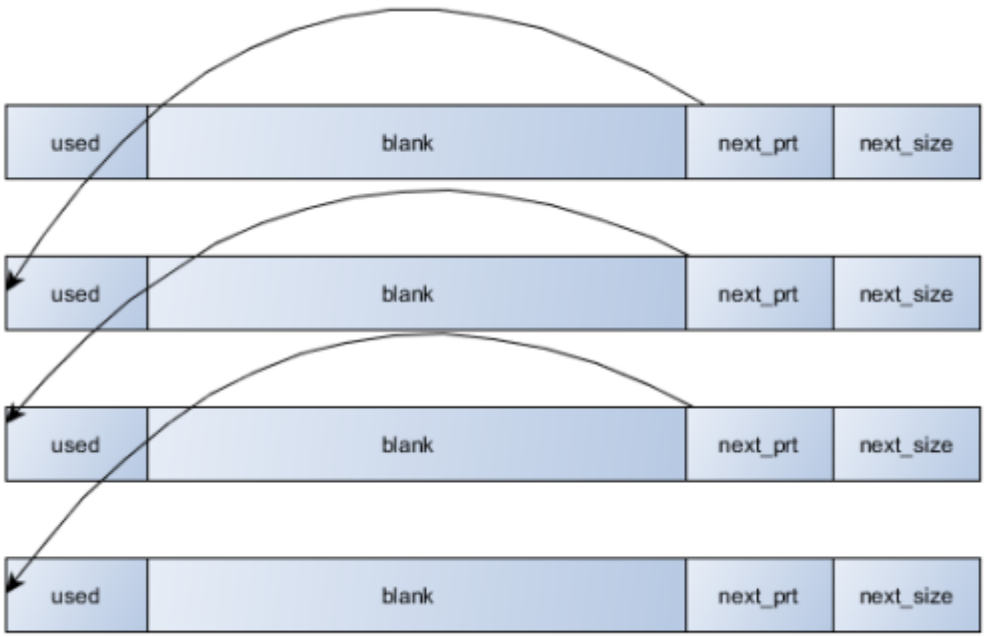
```
class PODptr
{
private:
    char * ptr_next_size() const
    {
        return (ptr + sz - sizeof(size_type));
    }
    char * ptr_next_ptr() const
    {
        return (ptr_next_size() -
              integer::static_lcm<sizeof(size_type), sizeof(void *)>::value);
    }
public:
    char * begin() const
    {
        return ptr;
    }
    char * end() const //a 'past the end' value
    {
        return ptr_next_ptr();
    }

    char * & next_ptr() const
```

```
    {
      return *(static_cast<char **>(static_cast<void*>(ptr_next_ptr())));
    }
};
```

4.2.2 pool

`pool` is a fast memory allocator that guarantees proper alignment of all allocated chunks. Section 4.1.1 showed the interface of `pool`.

`pool` has several blocks, which are of type `PODptr`. When the blocks are used up, it will apply for new blocks. The blocks are stored in a `std::list` and the `next_ptr` of each block points to the next element in the list. The blocks are divided into several fixed-size chunks and in each block application, the number of chunks to be applied for is doubled. The chunks are managed by the base class of pool -- `simple_segregated_storage` .



`pool` is a template class, with one template parameter `UserAllocator`, which defines the strategy to allocate the memory. Whenever an object of type pool needs memory from system, it will request it from its `UserAllocator`.

The declaration of pool's constructor is:

```
explicit pool(const size_type nrequested_size,
        const size_type nnext_size = 32,
        const size_type nmax_size = 0)
```

- `nrequested_size` requested fixed chunk size.
- `nnext_size` parameter is of type size_type. It is the number of chunks to request from the system the first time that object needs to allocate system memory. The default is 32 and this parameter may not be 0.
- `nmax_size` is the maximum number of chunks to allocate in one block. When this parameter takes the default value of 0, then there is no upper limit on chunk size.

Member function `is_from` can test a chunk to determine if it belongs to a block at `i` of size `sizeof_i`. Note that `std::less_equal` and `std::less` are used to test chunk against the array bounds because standard operators < <= > >= are only defined for pointers that are in the same array or subobjects of the same object while `std::less_equal` and `std::less` use function objects to guarantee a total order for any pointer.

```
static bool is_from(void * const chunk, char * const i,
    const size_type sizeof_i)
{
    std::less_equal<void *> lt_eq;
    std::less<void *> lt;
    return (lt_eq(i, chunk) && lt(chunk, i + sizeof_i));
}
```

`malloc` can allocate a chunk of memory. It searches in the list of memory blocks for a block that has a free chunk, and returns that free chunk if found. Otherwise, it creates a new memory block, adds its free list to pool's free list, returns a free chunk from that block.

```
void * malloc BOOST_PREVENT_MACRO_SUBSTITUTION()
{
    if (!store().empty())
        return (store().malloc)(); // defined in base class
    return malloc_need_resize();
}
```

`store()` is a protected member function that returns a reference upcasted to its base class. There are two versions - const and non-const.

```
simple_segregated_storage<size_type> & store()
{
    return *this;
}
const simple_segregated_storage<size_type> & store() const
{
    return *this;
}
```

`malloc_need_resize` make a new block, allocates chunk from the new block and returns pointer to the chunk. The number of chunks in the new block doubles the last block.

`ordered_malloc` is the same as `malloc` except that it merges the free lists.

```
void * ordered_malloc()
{
    if (!store().empty())
    return (store().malloc)();
    return ordered_malloc_need_resize();
}
```

The group of functions corresponding to `malloc()` and `ordered_malloc` is `free()` and `malloc_free()`. They call `free()` and `ordered_free` (both are defined in the base class) to deallocate a chunk of memory.

```
void free BOOST_PREVENT_MACRO_SUBSTITUTION(void * const chunk)
{
    (store().free)(chunk);
}

void ordered_free(void * const chunk)
{
    store().ordered_free(chunk);
}
```

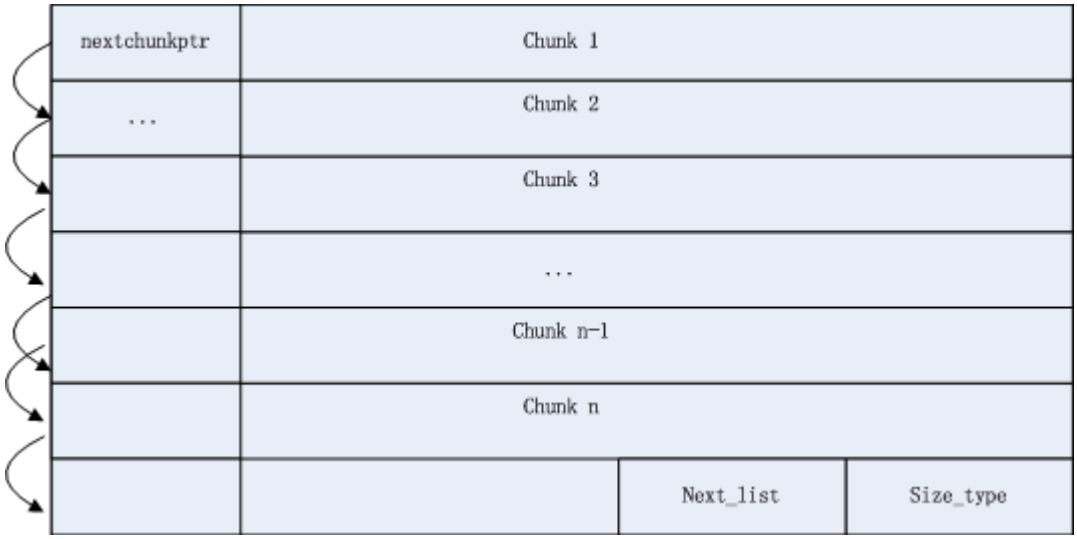
The last two public member functions . `release_memory()` releases memory blocks that don't have chunks allocated and `purge_memory()` releases *all* memory blocks, even if chunks are still allocated. And actually, `purge_memory` is the function that the destructor calls. Note that though `free()` , `ordered_free()`, `release_memory()` , `purge_memory()` can be called by the users, they are not suggested to be called by the users.

4.2.3 simple_segregated_storage

Simple Segregated Storage is the simplest, and probably the fastest memory allocation/deallocation algorithm. It is responsible for partitioning a memory block into fixed-size chunks : where the block comes from is determined by the client of the class.

Template class `simple_segregated_storage<SizeType>` controls access to a free list of memory chunks. An object of it is empty if its free list is empty. If it is not empty, then it is ordered if its free list is ordered. (Therefore, some member functions has two versions, such as `add_block()` / `add_ordered_block` and `free()` / `ordered_free`)

A chunk is divided into two parts. The first part is a pointer that points to the next chunk and the second part is the real chunk. The relationship between chunk and block is shown in the figure below.



sysnopsis

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

  public:
    typedef SizeType size_type;

    simple_segregated_storage();
    ~simple_segregated_storage();

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

    bool empty() const;

    void * malloc();
    void free(void * chunk);
    void ordered_free(void * chunk);

    void * malloc_n(size_type n, size_type partition_sz);
```

```
    void free_n(void * chunks, size_type n,
        size_type partition_sz);
    void ordered_free_n(void * chunks, size_type n,
        size_type partition_sz);
};
```

5 Other Programming Skills

In this section, I will show some small tricks I have learned from code reading.

5.1 BOOST_STATIC_CONSTANT

Some compilers don't allow in-class initialization of static constant members, so we must use `enum` or out-of-class definition if we want the constants to be available at compile-time. This macro gives us a convenient way to declare such constants.

```
#  ifdef BOOST_NO_INCLASS_MEMBER_INITIALIZATION
#       define BOOST_STATIC_CONSTANT(type, assignment) enum { assignment }
#  else
#     define BOOST_STATIC_CONSTANT(type, assignment) static const type
assignment
#  endif
```

`BOOST_NO_INCLASS_MEMBER_INITIALIZATION` can be understood in terms of literal meaning.

If you want to use

```
struct foo{
   static const int value = 2;
};
```

just type

```
struct foo{
   BOOST_STATIC_CONSTANT(int, value = 2);
};
```

5.2 Pass Special Status to Functions

When no constructors are defined, the compiler will synthetize a default one. However, in some cases, the default constructor may be deleted. Boost::python::class_ uses a clever way to distinguish them.

```
enum no_init_t { no_init };
class class_
{
public:
    class_(char const* name);
    class_(char const* name, no_init_t);
}
```

Then, we can use `class_("foo1")` to create a class with a default constructor and use `class("foo2",no_init)` to create a class without a default constructor. Compared with the constructor `class_(char const* name, bool init_deleted=0)` with function calls `class_("foo1")` and `class("foo2,0")`, they are easier to be understood.

5.3 Same Type

Class templates can be used to check whether type T1 and T2 are the same type.

```
template< class T1, class T2 > struct is_same
{
    BOOST_STATIC_CONSTANT( bool, value = false );
};

template< class T > struct is_same< T, T >
{
    BOOST_STATIC_CONSTANT( bool, value = true );
};
```

The n, `is_same<T1,T2>::value` is $true$ when $T1 == T2$ and `is_same<T1,T2>::value` is $false$ when $T1 != T2$. The source code can be found in `boost\core\is_same.hpp` and is_same can be used by including this header file and using the namespace `boost::core`

5.4 GCD by template metaprogramming

Boost provides a compile-time greatest common divisor evaluator-`static_gcd<V1,V2>`. This evaluator is defined in header file `boost\integer\common_factor_ct.hpp` and in namespace `boost::integer`. Here is an example of calculating $gcd(9,6)$:

```
int a=static_gcd<9,6>::value;
```

The definition `static_gcd<V1,V2>` is as follows.

```
template < static_gcd_type Value1, static_gcd_type Value2 > struct static_gcd
{
    BOOST_STATIC_CONSTANT( static_gcd_type, value =
(detail::static_gcd_helper_t<Value1, Value2>::value) );
};  // boost::integer::static_gcd
```

`static_gcd_type` can be regarded as the largest integer type that the compiler supports and the definition of `static_gcd_helper_t` can be found in the same file. It calculates $gcd(v1,v2)$ with Euclid's recursive algorithm.

```
template < static_gcd_type Value1, static_gcd_type Value2 >
    struct static_gcd_helper_t
    {
    private:
        BOOST_STATIC_CONSTANT( static_gcd_type, new_value1 = Value2 );
        BOOST_STATIC_CONSTANT( static_gcd_type, new_value2 = Value1 % Value2
);

        #define BOOST_DETAIL_GCD_HELPER_VAL(Value)
static_cast<static_gcd_type>(Value)
        typedef static_gcd_helper_t< BOOST_DETAIL_GCD_HELPER_VAL(new_value1),
         BOOST_DETAIL_GCD_HELPER_VAL(new_value2) >  next_step_type;
        #undef BOOST_DETAIL_GCD_HELPER_VAL

    public:
        BOOST_STATIC_CONSTANT( static_gcd_type, value = next_step_type::value
);
    };
```

And the termination of the recursion is:

```
template < static_gcd_type Value1 >
    struct static_gcd_helper_t< Value1, 0UL >
    {
        BOOST_STATIC_CONSTANT( static_gcd_type, value = Value1 );
    };
```

Other algorithms based on recursion can be implemented by imitating the implementation of GCD.

5.5 Base Class for Template Class

As templates will sacrifice efficiency during compiling(It won't reduce operating efficiency), the template classes should be simplified wherever it can. One requirement is that, codes that are not dependent on template parameters should be put in a base class. For example, in `boost/python/object/class.hpp`, a base class `class_base` is created for the independent codes of `class_`.

```
struct BOOST_PYTHON_DECL class_base : python::api::object
{
    class_base();
    void enable_pickling_(bool getstate_manages_dict);

 protected:
    void add_property(char const* name, object const& fget, char const*
docstr);
    void add_property(char const* name, object const& fget, object const&
fset, char const* docstr);

    void add_static_property(char const* name, object const& fget);
    void add_static_property(char const* name, object const& fget, object
const& fset);

    void setattr(char const* name, object const&);
    void set_instance_size(std::size_t bytes);
    void def_no_init();

    void make_method_static(const char *method_name);
```

```
};
```

The template class `class_` is a derived class of `class_base`. The codes that depends on the template parameters are put in this class.

```
template <class W, class X1, class X2, class X3>
class class_ : public objects::class_base
{
...
}
```