Module05-04
C++ Boost：智能指针

C++ Boost - 智能指针

- 容器相关
- 字符串和文字处理
- 正则表达式
- → 智能指针
- 函数对象相关
- 序列化
- 日期与时间
- 多线程
- 网络

智能指针 - 关于 boost.smart_ptr

- 关于 smart_ptr
  - smart pointers（智能指针）是存储“指向动态分配（在堆上）的对象的指针”的对象。他们的行为很像 C++ 的普通指针，只是它们可以在适当的时候自动删除它们所指向的对象。智能指针在面对异常时有非常显著的作用，它们可以确保动态分配对象的完全析构。它们还可以用于跟踪多个所有者共享的动态分配对象。
  - 在概念上，智能指针可以看作拥有它所指向的对象，并因此在对象不再需要时负责将它删除。

智能指针 - 关于 boost.smart_ptr

- boost.smart_ptr
  - boost.smart_ptr 提供了 6 个智能指针类模板：

scoped_ptr	<boost/scoped_ptr.hpp>	简单的单一对象的唯一所有权。不可拷贝。
scoped_array	<boost/scoped_array.hpp>	简单的数组的唯一所有权。不可拷贝。
shared_ptr	<boost/shared_ptr.hpp>	在多个指针间共享的对象所有权。
shared_array	<boost/shared_array.hpp>	在多个指针间共享的数组所有权。
weak_ptr	<boost/weak_ptr.hpp>	一个属于 shared_ptr 的对象的无所有权的观察者。
intrusive_ptr	<boost/intrusive_ptr.hpp>	带有一个侵入式引用计数的对象的共享所有权。

  - 多个 make_shared 函数

智能指针 - scoped_ptr

- 智能指针：
  - 关于 boost.smart_ptr
  - scoped_ptr, scoped_array
  - shared_ptr, shared_array (In TR1)
  - weak_ptr (In TR1)
  - intrusive_ptr
  - make_shared, allocate_shared
  - enable_shared_from_this
  - 利用 weak_ptr 打破 shared_ptr 的循环引用
  - 关于 pimpl(private implements)

智能指针 - scoped_ptr

- 关于 scoped_ptr
  - scoped_ptr class template 存储一个指向动态分配对象的指针（动态分配对象是用 C++ new 表达式分配的）。在 scoped_ptr 的析构过程中，或者显式的 reset，要保证它所指向的对象被删除
  - 实际上是对裸指针的包装，但是限制了指针的复制行为
  - 同 std::auto_ptr 类似，但没有所有权转移（复制）的行为
  - 由于不能复制， scoped_ptr 和 scoped_array 不能当作容器的元素
  - scoped_array 在性质上与 scoped_ptr 一致，只是用于管理动态分配的数组对象

智能指针 - scoped_ptr

- scoped_ptr 接口

```
template<class T> class scoped_ptr: noncopyable {
public:
    typedef T element_type;

    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);
};
```

智能指针 - scoped_ptr

- scoped_ptr 示例

```
struct A {
    A() { cout << "A::A()" << endl; }

    ~A() { cout << "A::~A()" << endl; }

    void f() { cout << "A::f()" << endl; }
};

void func() {
    boost::scoped_ptr<int> sp(new int(128));
    cout << ++*sp << endl; // 129
    boost::scoped_ptr<int> sp2(sp); // Error

    boost::scoped_ptr<A> ap(new A);
    ap->f();
}
```

智能指针 - shared_ptr

- 智能指针：
  - 关于 boost.smart_ptr
  - scoped_ptr, scoped_array
  - shared_ptr, shared_array (In TR1)
  - weak_ptr (In TR1)
  - intrusive_ptr
  - make_shared, allocate_shared
  - enable_shared_from_this
  - 利用 weak_ptr 打破 shared_ptr 的循环引用
  - 关于 pimpl(private implements)

智能指针 - shared_ptr

- 关于 shared_ptr
  - shared_ptr 类模板存储一个指向动态分配对象（一般是用 C++ new-expression 生成的）的指针。在最后一个 shared_ptr 所指向的对象被销毁或重置时，要保证它所指向的对象被删除。
  - 每一个 shared_ptr 都符合 C++ 标准库的 CopyConstructible 和 Assignable 的必要条件，并因此能够用于标准库容器。因为提供了比较操作，因此 shared_ptr 还可以和标准库中的关联式容器一起工作。
  - 只要 T* 能被隐式地转换到 U*，则 shared_ptr<T> 就能被隐式地转换到 shared_ptr<U>。特别是，shared_ptr<T> 隐式转换到 shared_ptr<T const>，当 U 是 T 的一个可访问基类的时候，还能转换到 shared_ptr<U>，以及转换到 shared_ptr<void>

智能指针 - shared_ptr

- shared_ptr 接口

```
template<class T> class shared_ptr {
public:
    shared_ptr();
    template<class Y> explicit shared_ptr(Y * p);
    template<class Y, class D> shared_ptr(Y * p, D d);
    template<class Y, class D, class A> shared_ptr(Y * p, D d, A
a);
    ~shared_ptr();

    shared_ptr(shared_ptr const& r);
    template<class Y> shared_ptr(shared_ptr<Y> const& r);
    template<class Y> explicit shared_ptr(weak_ptr<Y> const& r);
    template<class Y> explicit shared_ptr(std::auto_ptr<Y>& r);

    shared_ptr & operator=(shared_ptr const& r);
    template<class Y> shared_ptr& operator=(shared_ptr<Y> const
& r);
    template<class Y> shared_ptr& operator=(std::auto_ptr<Y>&
r);
```

智能指针 - shared_ptr

- shared_ptr 接口（续）

```
    void reset();
    template<class Y> void reset(Y * p);
    template<class Y, class D> void reset(Y * p, D d);
    template<class Y, class D, class A> void reset(Y * p, D d, A
a);
    template<class Y> void reset(shared_ptr<Y> const & r, T *
p);

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

    bool unique() const;
    long use_count() const;

    operator unspecified-bool-type() const;

    void swap(shared_ptr & b);
};
```

智能指针 - shared_ptr

- shared_ptr 相关操作

```
template<class T, class U>
bool operator==(shared_ptr<T> const& a, shared_ptr<U> const& b);

template<class T, class U>
bool operator!=(shared_ptr<T> const& a, shared_ptr<U> const& b);

template<class T, class U>
bool operator<(shared_ptr<T> const& a, shared_ptr<U> const& b);

template<class T> void swap(shared_ptr<T> & a, shared_ptr<T> &
b);

template<class T> T * get_pointer(shared_ptr<T> const& p);
```

智能指针 - shared_ptr

- shared_ptr 相关操作（续）

```
template<class T, class U>
shared_ptr<T> static_pointer_cast(shared_ptr<U> const& r);

template<class T, class U>
shared_ptr<T> const_pointer_cast(shared_ptr<U> const& r);

template<class T, class U>
shared_ptr<T> dynamic_pointer_cast(shared_ptr<U> const& r);

template<class E, class T, class Y>
std::basic_ostream<E, T> & operator<<(std::basic_ostream<E, T> &
os, shared_ptr<Y> const& p);
```

智能指针 - weak_ptr

- 智能指针：
  - 关于 boost.smart_ptr
  - scoped_ptr, scoped_array
  - shared_ptr, shared_array（In TR1）
  - weak_ptr （In TR1）
  - intrusive_ptr
  - make_shared, allocate_shared
  - enable_shared_from_this
  - 利用 weak_ptr 打破 shared_ptr 的循环引用
  - 关于 pimpl(private implements)

智能指针 - weak_ptr

- 关于 weak_ptr
  - weak_ptr 类模板存储一个引向已被 shared_ptr 管理的对象的 "weak reference"（弱引用）。为了访问这个对象，一个 weak_ptr 可以利用 shared_ptr 的构造函数或成员函数 lock 转换为 shared_ptr。当最后一个指向对象的 shared_ptr 消失，而对象也被删除后，从一个引向已被删除对象的 weak_ptr 实例获取 shared_ptr 的企图就会失败：构造函数会抛出一个 boost::bad_weak_ptr 类型的异常，而 weak_ptr::lock 会返回一个 empty shared_ptr。
  - 每一个 weak_ptr 都符合 C++ 标准库的 CopyConstructible 和 Assignable 的必要条件，并因此能够用于标准库容器。因为提供了比较操作，因此 weak_ptr 也可以和标准库中的关联式容器一起工作。

智能指针 - weak_ptr

- weak_ptr 接口

```
template<class T> class weak_ptr {
public:
    weak_ptr();
    template<class Y> weak_ptr(shared_ptr<Y> const & r);
    weak_ptr(weak_ptr const & r);
    template<class Y> weak_ptr(weak_ptr<Y> const & r);
    ~weak_ptr();

    weak_ptr & operator=(weak_ptr const & r);
    template<class Y> weak_ptr & operator=(weak_ptr<Y> const&
r);
    template<class Y> weak_ptr & operator=(shared_ptr<Y> const &
r);
    long use_count() const;
    bool expired() const;
    shared_ptr<T> lock() const;
    void reset();
    void swap(weak_ptr<T> & b);
};
```

智能指针 - shared_ptr

- weak_ptr 相关操作

```
template<class T, class U>
bool operator<(weak_ptr<T> const & a, weak_ptr<U> const & b);

template<class T>
void swap(weak_ptr<T> & a, weak_ptr<T> & b);
```

智能指针 - weak_ptr

- 智能指针：
  - 关于 boost.smart_ptr
  - scoped_ptr, scoped_array
  - shared_ptr, shared_array (In TR1)
  - weak_ptr (In TR1)
  - intrusive_ptr
  - make_shared, allocate_shared
  - enable_shared_from_this
  - 利用 weak_ptr 打破 shared_ptr 的循环引用
  - 关于 pimpl(private implements)

智能指针 - intrusive_ptr

- 关于 intrusive_ptr
  - intrusive_ptr 类模板存储一个指向带有侵入式引用计数的对象的指针。每一个新的 intrusive_ptr 实例都通过对函数 intrusive_ptr_add_ref 的无条件调用（将指针作为参数）增加引用计数。同样，当一个 intrusive_ptr 被销毁，它会调用 intrusive_ptr_release，这个函数负责当引用计数降为 0 时销毁这个对象。这两个函数的适当定义由用户提供。在支持 argument-dependent lookup （参数依赖查找）的编译器上，intrusive_ptr_add_ref 和 intrusive_ptr_release 应该和它们的参数定义在同一个名字空间中，否则，就定义名字空间 boost 中。
  - 这个类模板以 T 为参数，T 是被指向的对象的类型。只要 T* 能被隐式地转换到 U*，则 intrusive_ptr<T> 就能被隐式地转换到 intrusive_ptr<U>。

智能指针 - intrusive_ptr

- 为什么会用到 intrusive_ptr
  - 一些已有的 frameworks 和操作系统提供带有侵入式引用计数的对象；
  - intrusive_ptr 的内存占用量和相应的裸指针一样。
  - intrusive_ptr<T> 能够从任意一个类型为 T * 的裸指针构造出来。

智能指针 - intrusive_ptr

- intrusive_ptr 接口

```
template<class T> class intrusive_ptr {
public:
    intrusive_ptr();
    intrusive_ptr(T* p, bool add_ref = true);
    intrusive_ptr(intrusive_ptr const& r);
    template<class Y> intrusive_ptr(intrusive_ptr<Y> const& r);
    ~intrusive_ptr();
    intrusive_ptr& operator=(intrusive_ptr const& r);
    template<class Y>
    intrusive_ptr& operator=(intrusive_ptr<Y> const& r);
    template<class Y> intrusive_ptr& operator=(T* r);
    void reset(T* r);
    T& operator*() const;
    T* operator->() const;
    T* get() const;

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

智能指针 - intrusive_ptr

- intrusive_ptr 相关操作

```
template<class T, class U>
bool operator==(intrusive_ptr<T> const& a, intrusive_ptr<U>
const& b);
template<class T, class U>
bool operator!=(intrusive_ptr<T> const& a, intrusive_ptr<U>
const& b);
template<class T>
bool operator==(intrusive_ptr<T> const& a, T* b);
template<class T>
bool operator!=(intrusive_ptr<T> const& a, T* b);
template<class T>
bool operator==(T* a, intrusive_ptr<T> const& b);
template<class T>
bool operator!=(T* a, intrusive_ptr<T> const& b);
template<class T, class U>
bool operator<(intrusive_ptr<T> const& a, intrusive_ptr<U>
const& b);
```

智能指针 - intrusive_ptr

- intrusive_ptr 相关操作（续）

```
template<class T> void swap(intrusive_ptr<T>& a,
intrusive_ptr<T>& b);

template<class T> T* get_pointer(intrusive_ptr<T> const& p);

template<class T, class U>
intrusive_ptr<T> static_pointer_cast(intrusive_ptr<U> const& r);

template<class T, class U>
intrusive_ptr<T> const_pointer_cast(intrusive_ptr<U> const& r);

template<class T, class U>
intrusive_ptr<T> dynamic_pointer_cast(intrusive_ptr<U> const&
r);

template<class E, class T, class Y>
std::basic_ostream<E, T>& operator<<(std::basic_ostream<E, T>&
os, intrusive_ptr<Y> const& p);
```

智能指针 - intrusive_ptr

- 使用 intrusive_ptr
  - 由于 intrusive_ptr 是侵入式智能指针，需要其管理的对象提供合适的引用计数方法
  - 除此之外，还需保证 intrusive_ptr_add_ref() 和 intrusive_ptr_release() 两个函数正确定义

智能指针 - intrusive_ptr

- 使用 intrusive_ptr（示例）

```
class ReferenceCounter {
    int refCount;
public:
    ReferenceCounter() :
        refCount(0) {
    }

    virtual ~ReferenceCounter() {
    }

    friend void intrusive_ptr_add_ref(ReferenceCounter* p) {
        ++p->refCount;
    }

    friend void intrusive_ptr_release(ReferenceCounter* p) {
        if (--p->refCount == 0)
            delete p;
    }
```

智能指针 - intrusive_ptr

- 使用 intrusive_ptr（示例）（续 1 ）

```
protected:
    ReferenceCounter& operator=(const ReferenceCounter&) {
        // 无操作
        return *this;
    }
private:
    // 禁止复制构造函数
    ReferenceCounter(const ReferenceCounter&);
};
```

```
class X: public ReferenceCounter {
public:
    X() {
        cout << "X::X()" << endl;
    }
    ~X() {
        cout << "X::~X()" << endl;
    }
};
```

智能指针 - intrusive_ptr

- 使用 intrusive_ptr（示例）（续 2 ）

```
// 测试代码
void f() {
    cout << "Before start of scope\n";
    {
        boost::intrusive_ptr<X> p1(new X());
        boost::intrusive_ptr<X> p2(p1);
    }
    cout << "After end of scope \n";
}
```

智能指针 - make_shared

- 智能指针：
  - 关于 boost.smart_ptr
  - scoped_ptr, scoped_array
  - shared_ptr, shared_array (In TR1)
  - weak_ptr (In TR1)
  - intrusive_ptr
  - make_shared, allocate_shared
  - enable_shared_from_this
  - 利用 weak_ptr 打破 shared_ptr 的循环引用
  - 关于 pimpl(private implements)

智能指针 - make_shared

- 关于 make_shared 和 allocate_shared
  - 使用 shared_ptr 可以消除对显式 delete 的使用，但是它没有提供避免显式 new 的支持。有用户反复地要求提供一个工厂函数，用于创建给定类型的对象并返回一个指向它的 shared_ptr. 除了方便使用和保持风格以外，这样的函数还具有异常安全性且明显更快，因为它可以对对象和相应的控制块两者同时使用单次的内存分配，以消除 shared_ptr 的构造过程中最大的一部分开销。这消除了对于 shared_ptr 的一个主要的抱怨。
  - 头文件 <boost/make_shared.hpp> 提供了一组重载的函数模板， make_shared 和 allocate_shared，它们解决了这些需要。 make_shared 使用全局的 operator new 来分配内存，而 allocate_shared 则使用用户所提供的分配器，可以更好地进行控制。

智能指针 - make_shared

- make_shared 和 allocate_shared 接口

```
template<typename T>
shared_ptr<T> make_shared();

template<typename T, typename A> // A是Allocator
shared_ptr<T> allocate_shared(A const &);

template<class T, class A1>
boost::shared_ptr<T> make_shared(A1 const & a1);

// A是Allocator, A1 a1是用于T类型的T(A1)形式的(有一个参数的)构造函数
template<class T, class A, class A1>
boost::shared_ptr<T> allocate_shared(A const & a, A1 const &
a1);
```

智能指针 - make_shared

- make_shared 和 allocate_shared 接口（续）

```
template<class T, class A1, class A2, ..., class AN>
boost::shared_ptr<T> make_shared(A1 const & a1,
    A2 const & a2, ..., AN const & an);

template<class T, class A, class A1, class A2, ..., class AN>
boost::shared_ptr<T> allocate_shared(A const & a, A1 const & a1,
A2 const & a2, ..., AN const & an);
```

智能指针 - make_shared

- make_shared 示例

```
struct D {
    D() {}
    D(int k, double d) {}
};

struct E {
    E(string s) {}
};

void f() {
    // 调用D的默认构造函数: D()
    boost::shared_ptr<D> dp1 = boost::make_shared<D>();

    // 调用D的非默认构造函数: D(int, double)
    boost::shared_ptr<D> dp2 =
        boost::make_shared<D>(12, .618);

    // 调用E的构造函数: E(string)
    boost::shared_ptr<E> ep = boost::make_shared<E>("test");
}
```

智能指针 - enable_shared_from_this

- 智能指针：
  - 关于 boost.smart_ptr
  - scoped_ptr, scoped_array
  - shared_ptr, shared_array (In TR1)
  - weak_ptr (In TR1)
  - intrusive_ptr
  - make_shared, allocate_shared
  - enable_shared_from_this
  - 利用 weak_ptr 打破 shared_ptr 的循环引用
  - 关于 pimpl(private implements)

智能指针 - enable_shared_from_this

- 从 this 指针创建 shared_ptr
  - 某些时候需要从 this 获得 shared_ptr ，即是说，你希望你的类被 shared_ptr 所管理，你需要把 " 自身 " 转换为 shared_ptr 的方法
  - 通过存储一个指向 this 的 weak_ptr 作为类的成员，就可以在需要的时候获得一个指向 this 的 shared_ptr 。为了你可以不必编写代码来保存一个指向 this 的 weak_ptr ，接着又从 weak_ptr 获 shared_ptr 得， Boost.Smart_ptr 为这个任务提供了一个助手类，称为 enable_shared_from_this. 只要简单地让你的类公有地派生自 enable_shared_from_this ，然后在需要访问管理 this 的 shared_ptr 时，使用函数 shared_from_this 就行了

智能指针 - enable_shared_from_this

- enable_shared_from_this 接口

```
template<class T> class enable_shared_from_this {
protected:
    enable_shared_from_this();
    enable_shared_from_this(enable_shared_from_this const &);
    enable_shared_from_this& operator=(enable_shared_from_this
const &);
    ~enable_shared_from_this();
public:
    shared_ptr<T> shared_from_this();
    shared_ptr<T const> shared_from_this() const;
private:
    mutable weak_ptr<T> weak_this_;
};
```

智能指针 - enable_shared_from_this

- 使用 enable_shared_from_this

```
class A;
void func(boost::shared_ptr<A> p) {
    //...
}

class A: public boost::enable_shared_from_this<A> {
public:
    void f() {
        func(shared_from_this());
    }
};
```

智能指针 - 打破 shared_ptr 的循环引用

- 智能指针：
  - 关于 boost.smart_ptr
  - scoped_ptr, scoped_array
  - shared_ptr, shared_array (In TR1)
  - weak_ptr (In TR1)
  - intrusive_ptr
  - make_shared, allocate_shared
  - enable_shared_from_this
  - 利用 weak_ptr 打破 shared_ptr 的循环引用
  - 关于 pimpl(private implements)

智能指针 - 打破 shared_ptr 的循环引用

- 使用 weak_ptr 打破 shared_ptr 循环引用

```
struct B;

struct A {
    ~A() { cout << "~A()" << endl; }
    boost::shared_ptr<B> b;
};
struct B {
    ~B() { cout << "~B()" << endl; }
    boost::shared_ptr<A> a; // 循环引用
    // boost::weak_ptr<A> a; // 可以打破循环引用
};

int main() {
    boost::shared_ptr<A> ap(new A);
    boost::shared_ptr<B> bp(new B);
    ap->b = bp;
    bp->a = ap;
    cout << "----------------------\n";
}
```

智能指针 - 关于 pimpl

- 智能指针：
  - 关于 boost.smart_ptr
  - scoped_ptr, scoped_array
  - shared_ptr, shared_array (In TR1)
  - weak_ptr (In TR1)
  - intrusive_ptr
  - make_shared, allocate_shared
  - enable_shared_from_this
  - 利用 weak_ptr 打破 shared_ptr 的循环引用
  - 关于 pimpl(private implements)

智能指针 - 关于 pimpl

- 关于 pimpl 技术
  - pimpl: private implements，是隐藏类定义细节的一种手法
  - 示例：

```
// 头文件 example.h
class example {
public:
    example();
    void do_something();
private:
    class implementation;

    // hide implementation details
    boost::shared_ptr<implementation> _imp;
};
```

智能指针 - pimpl

- 关于 pimpl 技术
  - 示例（续 1 ）：

```
// 实现文件: example.cpp
#include "example.h"

class example::implementation {
public:
    ~implementation() {
        std::cout << "destroying implementation\n";
    }
};

example::example() :
    _imp(new implementation) {
}
void example::do_something() {
    std::cout << "use_count() is "
              << _imp.use_count() << "\n";
}
```

智能指针 - pimpl

- 关于 pimpl 技术
  - 示例（续 2 ）：

```
// 测试代码

#include "example.h"

int main() {
    example a;
    a.do_something();
    example b(a);
    b.do_something();
    example c;
    c = a;
    c.do_something();
    return 0;
}
```

智能指针 - More Stuffs...

- boost.smart_ptr 提供一族易用且高效的智能指针，在多数情况下，我们应该趋向使用这些智能指针，而不是使用裸指针 (Row Pointers)
- 除 boost.smart_ptr 的智能指针实现外， C++ 社区中还有其它的实现，如 loki 库中的智能指针，请参考：
  - Alexandrescu Andrei: C++ Modern Design (Book)