## 1.智能指针

在C11之后尽量不要再用new 和 delete，用智能指针； C11 之前可以用boost智能指针，大部分新标准都来源于boost；

智能指针分类：

1. std::shared_ptr (boost::shared_ptr) 这个用的最多，就是可以共享所有权，每增加一个所有这，智能指针计数加1，放弃所有权，计数减1；计数为0释放：

```
#include <iostream>
#include <memory>
#include <thread>
#include <chrono>
#include <mutex>

struct Base
{
    Base() { std::cout << "  Base::Base()\n"; }
    // 注意：此处非虚析构函数 OK
    ~Base() { std::cout << "  Base::~Base()\n"; }
};

struct Derived: public Base
{
    Derived() { std::cout << "  Derived::Derived()\n"; }
    ~Derived() { std::cout << "  Derived::~Derived()\n"; }
};

void thr(std::shared_ptr<Base> p)
{
    std::this_thread::sleep_for(std::chrono::seconds(1));
    std::shared_ptr<Base> lp = p; // 线程安全，虽然自增共享的 use_count
    {
        static std::mutex io_mutex;
        std::lock_guard<std::mutex> lk(io_mutex);
        std::cout << "local pointer in a thread:\n"
                  << "  lp.get() = " << lp.get()
                  << ", lp.use_count() = " << lp.use_count() << '\n';
    }
}

int main()
{
```

```
    std::shared_ptr<Base> p = std::make_shared<Derived>();

    std::cout << "Created a shared Derived (as a pointer to Base)\n"
              << "  p.get() = " << p.get()
              << ", p.use_count() = " << p.use_count() << '\n';
    std::thread t1(thr, p), t2(thr, p), t3(thr, p);
    p.reset(); // 从 main 释放所有权
    std::cout << "Shared ownership between 3 threads and released\n"
              << "ownership from main:\n"
              << "  p.get() = " << p.get()
              << ", p.use_count() = " << p.use_count() << '\n';
    t1.join(); t2.join(); t3.join();
    std::cout << "All threads completed, the last one deleted Derived\n";
}
```

2. std::unique_ptr (boost::unique_ptr)
   独享，可以传递所有权,A传递给B，A必须失去所有权std::move(p)；

```
// 消费 unique_ptr 的函数能以值或以右值引用接收它
std::unique_ptr<D> pass_through(std::unique_ptr<D> p)
{
    p->bar();
    return p;
}

void close_file(std::FILE* fp) { std::fclose(fp); }

int main()
{
  std::cout << "unique ownership semantics demo\n";
  {
      auto p = std::make_unique<D>(); // p 是占有 D 的 unique_ptr
      auto q = pass_through(std::move(p)); //交给其他人了
      assert(!p); // 现在 p 不占有任何内容并保有空指针
      q->bar();   // 而 q 占有 D 对象
  } // ~D 调用于此
}
```

3. std::weak_ptr (boost::weak_ptr) shared_ptr的伴生指针，不能独立存在；从shared_ptr中构造，但是不影响shared_ptr的引用计数；在需要使用的时候，需要升级为shared_ptr、避免被释放；

```cpp
#include <iostream>
#include <memory>

std::weak_ptr<int> gw;

void f()
{
    if (auto spt = gw.lock()) { // 使用之前必须复制到 shared_ptr
    std::cout << *spt << "\n";
    }
    else {
        std::cout << "gw is expired\n";
    }
}

int main()
{
    {
        auto sp = std::make_shared<int>(42);
    gw = sp;

    f();
    }

    f();
}
```

4. boost::scope_ptr 和std::unique_ptr类似，但是不可以转移所有权,必须持有到释放；

```cpp
#include <boost/scoped_ptr.hpp>
#include <iostream>
```

```
struct Shoe { ~Shoe() { std::cout << "Buckle my shoe\n"; } };

class MyClass {
    boost::scoped_ptr<int> ptr;
  public:
    MyClass() : ptr(new int) { *ptr = 0; }
    int add_one() { return ++*ptr; }
};

int main()
{
   {
    boost::scoped_ptr<Shoe> x(new Shoe);
    } // 此处会释放
    MyClass my_instance;
    std::cout << my_instance.add_one() << '\n';
    std::cout << my_instance.add_one() << '\n';
}
```

5. std::auto_ptr *这是个设计缺陷，任何时候都不要用。*