13.1答：如果一个构造函数的第一个参数是自身类类型的引用，且任何额外的参数都有默认值则此函数是拷贝构造函数，而且拷贝构造函数通常不应该是explicit的。

a．将一个对象作为实参传递给一个非引用类型的形参

b．从一个返回类型为非引用类型的函数返回一个对象

c．用花括号列表初始化一个数组中的元素或一个聚合类中的成员

13.2答：参数不是引用类型

13.3答：拷贝一个strblob对象会增加右侧对象的引用计数，起weakcount不会变，当拷贝一个伴随类指针不会增加其shared\_ptr，但是其weak\_ptr会增加。

#include "strBlob.h"

int main()

{

/\*\*

\* @brief ex13.3. Use debug step by step to understand the whole procedure

\*/

//! default constructor is called making the shared\_ptr within it point to ,say, @0x8803018.

//! usecount == 1

//! weakcount == 1

StrBlob sb;

//! copy constructor is called making the pointer point to the same address i.e.@0x8803018.

//! usecount == 2

//! weakcount == 1

StrBlob sb\_duplic = sb;

//! default constructor. the weak pointer inside points to the same address @0x8803018.

//! usecount == 2

//! weakcount == 2

StrBlobPtr sb\_p(sb);

//! copy constructor is called, the pointer points to the same address i.e.@0x8803018

//! usecount == 2

//! weakcount == 3

StrBlobPtr sb\_p\_duplic = sb\_p;

//! For the rationale that the weekcount start by 1 rather than 0 ,go to the post on SO:

//! http://stackoverflow.com/questions/5671241/how-does-weak-ptr-work

return 0;

}

13.4Point foo\_bar(Point arg) // 调用此函数时，将实参对象的副本传递给形参Point的对象 arg

{

Pint local = arg; // 调用复制构造函数，将局部

对象 local初始化为形参arg的副本。

Point \*heap = new Point(global); // 调用复制

构造函数用全局对象 global来初始化

// Point 对象

\*heap

\*heap = local;

Point pa[4] = { local, \*heap }; // 使用数组初始化列表来初始化数组的每个元素。

return \*heap; // 从函数返回 Point 对象\*heap

的副本

13.5答：#include <string>

#include <iostream>

class HasPtr

{

public:

HasPtr(const std::string &s = std::string()) :

ps(new std::string(s)), i(0) { }

HasPtr(const HasPtr& oring) : ps(new std::string(\*oring.ps)), i(oring.i) { }

private:

std::string \*ps;

int i;

};

int main()

{

return 0;

}

13.6答：包含指针成员的时候或者在对象赋值操作的时候需要完成一些特性操作

13.7答： #include <iostream>

#include <string>

class HasPtr

{

public:

HasPtr(const std::string &s = std::string()) :

ps(new std::string(s)), i(0)

{

std::cout << "use the default cpnstructor" << std::endl;

}

HasPtr(const HasPtr& oring) : ps(new std::string(\*oring.ps)), i(oring.i)

{ std::cout << "use the copy constructor" << std::endl; }

HasPtr &operator=(const HasPtr &rhs)

{

\*ps = \*rhs.ps;

i = rhs.i;

std::cout << "cout use the copy assignment - operator" << std::endl;

return \*this;

}

private:

std::string \*ps;

int i;

};

int main()

{

HasPtr my\_hasptr("hello");

HasPtr my\_hasptr1 = my\_hasptr;

return 0;

}

13.8答：如上题

13.9答：

// As with the copy constructor and the copy-assignment operator, for some classes,

// the synthesized destructor is defined to disallow objects of the type from being

// destroyed (§ 13.1.6, p. 508). Otherwise, the synthesized destructor has an empty

// function body.

//

// When no user-defined destructor, the compiler will define a synthesized version.

13.10答：

//! What happens when a StrBlob object is destroyed? What about a StrBlobPtr?

//! When a StrBlobPter object is destroyed the object dynamicaly allocated will not be

//! freed.

//! When a StrBlob object destroyed, the use count of the dynamic object will decrement.

//! It wiil be freed if no shared\_ptr to that dynamic object.

13.11答： ~HasPtr()

{

delete ps;

}

13.12答：

13.13答：#include <iostream>

#include <string>

struct X

{

X() { std::cout << "X()" << std::endl; }

X(const X&) { std::cout << "X(const X&)" << std::endl; }

X& operator = (const X& x)

{

std::cout << "operator = (const X&)"<<std::endl;

return \*this;

}

~X()

{

std::cout << "~X()\n";

}

};

int main()

{

X x, y;

x = y;

return 0;

}

13.14 15答：#include <iostream>

#include <string>

struct Numbered

{

//! for ex13.14

Numbered()

{

static unsigned i = 0;

++i;

mySn = i;

}

//! for ex13.15

Numbered(const Numbered& num)

{

static unsigned j = 99;

++j;

mySn = j;

}

unsigned mySn;

};

//void f (Numbered s)

void f(Numbered &s)

{

std::cout << s.mySn << std::endl;

}

int main()

{

Numbered a, b = a, c = b;

f(a); f(b); f(c);

return 0;

}

13.16答：如上题所示

13.17答：如上题所示

13.18答：#include <iostream>

#include <string>

class Employee

{

public:

Employee() :name(std::string()), id(get\_id()){ }

Employee(const std::string&s) :name(s), id(get\_id()){ }

Employee(const Employee&rhs) :name(rhs.name), id(rhs.id){ }

Employee &operator=(const Employee&rhs)

{

name = rhs.name;

id = rhs.id;

return \*this;

}

unsigned get\_id();

std::string name;

unsigned id;

};

inline unsigned Employee::get\_id()

{

static unsigned i = 0;

return ++i;

}

int main()

{

Employee e1;

Employee e2("sss");

e1 = e2;

std::cout << e1.id << " " << e2.id<<std::endl;

std::cout << e1.name << std::endl;

std::cout << e2.name << std::endl;

return 0;

}

13.19答：

//! Exercise 13.19:

//! Does your Employee class need to define its own versions of the copy-control members?

//! If so, why? If not, why not? Implement whatever copy-control members you think Employee

//! needs.

// Yes it need the user-defined copy constructor and copy-assignment operator, which makes

// sure that each object has unique id.The user-defined destructor is not required.

13.20答：

//! Exercise 13.20:

//! Explain what happens when we copy, assign, or destroy objects of our TextQuery

//! and QueryResult classes from § 12.3 (p. 484).

//

//!

//! Exercise 13.21:

//! Do you think the TextQuery and QueryResult classes need to define their own

//! versions of the copy-control members? If so, why? If not, why not? Implement

//! whichever copy-control operations you think these classes require.

//!

// No copy-control members needed.

// Because, all these classes are using smart poitners to manage dynamic memory which

// can be freed automatically by calling synthesized destructors. The objects of these

// classes should share the same dynamic memory.Hence no user-defined version needed

// as well.

//!

13.22答：

#include<iostream>

#include<string>

using namespace std;

class HasPtr

{

public:

//default constructor

HasPtr(const std::string &s = std::string())

:ps(new string(s)), i(0){cout << 1 << endl;}

//copy constructor

HasPtr(const HasPtr &rhs) :ps(new std::string(\*rhs.ps)), i(rhs.i){ cout << 2 << endl; }

//copy assignment-operator

HasPtr &opterator(const HasPtr&rhs)

{

delete ps;

ps = new std::string(\*rhs.ps);

i = rhs.i;

cout << 3 << endl;

return \*this;

}

private:

std::string \*ps;

int i;

};

int main()

{

HasPtr hp("chen"), hp2("xun");

hp = hp2;

return 0;

}

13.25 26答：

#include <iostream>

#include <vector>

#include <string>

#include <memory>

#include <string>

#include <fstream>

using namespace std;

class StrBlobPtr;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

class StrBlob{

friend class StrBlobPtr;

public:

typedef std::vector<std::string>::size\_type size\_type;

StrBlob() :data(make\_shared<vector<string>>()){}

StrBlob(initializer\_list<string> il) :data(make\_shared<vector<string>>(il)){}

//! copy constructor for ex13.26

StrBlob(const StrBlob& sb);

//! copy-assignment operator for ex13.26

StrBlob& operator = (const StrBlob &sb);

size\_type size() const { return data->size(); }

bool empty() const { return data->empty(); }

void push\_back(const std::string &t) { data->push\_back(t); }

void pop\_back();

std::string &front();

std::string &back();

const std::string& front() const;

const std::string& back() const;

StrBlobPtr begin();

StrBlobPtr end();

private:

std::shared\_ptr<std::vector<std::string>> data;

void check(size\_type i, const std::string &msg) const;

};

//! copy constructor for ex13.26

inline StrBlob::StrBlob(const StrBlob &sb) :

data(std::make\_shared<std::vector<std::string>>(\*sb.data)){ }

inline StrBlob&

StrBlob::operator =(const StrBlob &sb)

{

auto p = std::make\_shared<std::vector<std::string>>(\*sb.data);

std::swap(data, p);

return \*this;

}

void StrBlob::check(size\_t i, const string &msg) const

{

if (i >= data->size())

throw out\_of\_range(msg);

}

string& StrBlob::front()

{

check(0, "front on empty StrBlob");

return data->front();

}

string& StrBlob::back()

{

check(0, "back on empty StrBlob");

return data->back();

}

void StrBlob::pop\_back()

{

check(0, "pop\_back on empty StrBlob");

data->pop\_back();

}

const std::string& StrBlob::front() const

{

check(0, "front on empty StrBlob");

return data->front();

}

const std::string& StrBlob::back() const

{

check(0, "back on empty StrBlob");

return data->back();

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

class StrBlobPtr{

friend bool eq(const StrBlobPtr&, const StrBlobPtr&);

public:

StrBlobPtr() :curr(0){};

StrBlobPtr(StrBlob &a, rsize\_t sz = 0) :wptr(a.data), curr(sz){ };

StrBlobPtr(const StrBlob &a, const size\_t sz = 0) : wptr(a.data), curr(sz) { }

std::string &deref() const;

StrBlobPtr &incr();//前缀递增

private:

std::shared\_ptr < std::vector < std::string >>

check(std::size\_t, const std::string&) const;

std::weak\_ptr<std::vector<std::string>> wptr;

std::size\_t curr;

};

std::shared\_ptr < std::vector < std::string >> StrBlobPtr::check(std::size\_t i, const std::string& msg) const

{

auto ret = wptr.lock();//vector是否存在

if (!ret)

throw std::runtime\_error("unbound StrBlobPtr");

if (i >= ret->size())

throw std::out\_of\_range(msg);

return ret;

}

std::string& StrBlobPtr::deref() const

{

auto p = check(curr, "dereference past end");

return (\*p)[curr];

}

StrBlobPtr & StrBlobPtr::incr()

{

check(curr, "increment past end of StrBlobPtr");

++curr;

return \*this;

}

StrBlobPtr StrBlob::begin(){ return StrBlobPtr(\*this); }

StrBlobPtr StrBlob::end()

{

auto ret = StrBlobPtr(\*this, data->size());

return ret;

}

// named equality operators for StrBlobPtr

inline

bool eq(const StrBlobPtr &lhs, const StrBlobPtr &rhs)

{

auto l = lhs.wptr.lock(), r = rhs.wptr.lock();

// if the underlying vector is the same

if (l == r)

// then they're equal if they're both null or

// if they point to the same element

return (!r || lhs.curr == rhs.curr);

else

return false; // if they point to difference vectors, they're not equal

}

inline

bool neq(const StrBlobPtr &lhs, const StrBlobPtr &rhs)

{

return !eq(lhs, rhs);

}

13.27答：#include<iostream>

#include<string>

using namespace std;

class HasPtr

{

public:

//default constructor默认构造函数

HasPtr(const std::string &s = std::string())

:ps(new string(s)), i(0), use(new std::size\_t(1)) {cout << 1 << std::endl;}

//copy constructor

HasPtr(const HasPtr &rhs) :ps(rhs.ps), i(rhs.i),use(rhs.use)

{

++\*use;

cout << 2 << endl;

}

//copy assignment-operator

HasPtr& operator=(const HasPtr& rhs);

~HasPtr()

{

if (--\*use == 0)

{

delete ps;

delete use;

}

}

private:

std::string \*ps;

int i;

std::size\_t \*use;

};

inline HasPtr& HasPtr::operator = (const HasPtr&rhs)

{

++\*rhs.use;

if (--\*use == 0)

{

delete ps;

delete use;

}

ps = rhs.ps;

i = rhs.i;

use = rhs.use;

cout << 3 << std::endl;

return \*this;

}

//inline HasPtr& HasPtr::operator = (const HasPtr& rhs)

//{

// delete ps;

// ps = new std::string(\*rhs.ps);

// i = rhs.i;

// cout << 3 << endl;

// return \*this;

//}

int main()

{

HasPtr hp;

HasPtr hp1("chen"), hp2("xun");

hp = hp2;

return 0;

}

13.28答：

13.29答：

//!

//! Exercise 13.29:

//! Explain why the calls to swap inside swap(HasPtr&, HasPtr&) do not cause a recursion loop.

// Because the parameters of the two swap functions have different types.

//!

//! Exercise 13.30:

//! Write and test a swap function for your valuelike version of HasPtr.

//! Give your swap a print statement that notes when it is executed.

//!

//! Exercise 13.31:

//! Give your class a < operator and define a vector of HasPtrs. Give that

//! vector some elements and then sort the vector. Note when swap is called.

//!

//! Exercise 13.32:

//! Would the pointerlike version of HasPtr benefit from defining a swap function?

//! If so, what is the benefit? If not, why not?

// Essentially, the specific avoiding memory allocation is the reason why it improve

// the performance. As for the pointerlike version, no dynamic memory allocation anyway.

// Thus a specific version for it will not improve the performance.

//!

#include<iostream>

#include<string>

#include<vector>

#include<algorithm>

class HasPtr

{

friend void swap(HasPtr&, HasPtr&);

friend bool operator<(const HasPtr&lhs, const HasPtr&rhs);

public:

HasPtr(const std::string &s = std::string()) :ps(new std::string(s)), i(0){}

HasPtr(const HasPtr&rhs) :ps(new std::string(\*rhs.ps)), i(rhs.i){}

HasPtr& operator=(HasPtr rhs);

~HasPtr()

{

delete ps;

}

private:

std::string \*ps;

int i;

};

inline void swap(HasPtr& lhs, HasPtr&rhs)

{

using std::swap;

swap(lhs.ps, rhs.ps);

swap(lhs.i, rhs.i);

std::cout << "using operator==" << std::endl;

}

inline bool operator <(const HasPtr& lhs, const HasPtr& rhs)

{

std::cout << "operator <" << std::endl;

return \*lhs.ps < \*rhs.ps;

}

HasPtr& HasPtr::operator=(HasPtr rhs)

{

swap(\*this, rhs);

return \*this;

}

int main()

{

HasPtr h1("dd"), h2("bb"), h3("cc");

std::vector<HasPtr> v = { h1, h2, h3 };

std::sort(v.begin(),v.end());

return 0;

}

13.33答：

13.38答：复制和交换是一种优雅的方式工作的动态分配的内存的时候。消息类中，没有什么是动态分配。因此，使用这个习语是没有意义的，将使其更复杂的实现由于指针指向。

13.39答：