# SECTION 03
Smart pointers

# Problems with 'raw' pointers

```cpp
class X
{
};

void func(X& theX)
{
   theX = *(new X);
}


int main()
{
   X& x = *(new X);
   func(x);
}
```

Raw pointer types do not perform any management of their resource.  This
can lead to all sorts of nasty problems.

# Smart pointer types

std::shared_ptr

std::unique_ptr

std::weak_ptr

C++11 has three managed pointer types:

std::shared_ptr
A reference-counted pointer, introduced in C++98 TR1

std::unique_ptr
Single-owner managed pointer which replaces (the now deprecated)
auto_ptr

std::weak_ptr
Works with shared_ptr in situations where circular references could be a
problem

# Single ownership pointer

```
#include <memory>

class X
{
public:
  void op();
};

int main()
{
  std::unique_ptr<X> p1(new X);     // NB: You can't do:
                                    // std::unique_ptr<X> p1 = new X;
  p1->op();

  std::unique_ptr<X> p2(p1);           // ERROR - Copy construction
  std::unique_ptr<X> p3(std::move(p1)); // OK - move constructor called

}
```

unique_ptr allows single ownership of a resource.  That is, only one
unique_ptr can ever be pointing at the resource

unique_ptr supports move semantics but not copying; unlike auto_ptr.

# Custom delete functor

```cpp
class SpecialDeleter
{
public:
  void operator() (SpecialCase* pObj)
  {
    cout << "Custom deleter for SpecialCase objects" << endl;
    delete pObj;
  }
};


int main()
{
  std::unique_ptr<SpecialCase, SpecialDeleter> p(new SpecialCase);
}
```

unique_ptr allows a custom delete functor; for situations requiring special cleanup (for example, files)

The deletor functor operator()  must take a pointer to the type referenced by the unique_ptr

# Reference-counted pointer

```cpp
#include <memory>

class X
{
public:
  X(int val);
  void op();
};

void func(std::shared_ptr<X> p)       // Pass-by-value; copy made on call
                                      // ref count incremented
  p->op();
}                                     // temp object deleted;
                                      // ref count decremented


int main()
{
  auto ptr = std::make_shared<X>(100);  // Creates a new shared_ptr<X>.

  ptr -> op();
  func(ptr);
}                                     // Last shared_ptr goes out of
                                      // scope; delete memory.
```

std::shared_ptr is a reference-counted smart pointer.

Each time a shared pointer is copied the reference count is incremented.
Each time one of the pointers goes out of scope the reference count on the
resource is decremented.  When the reference count is zero (that is, the last
shared_ptr referencing the resource goes out of scope) the resource is
deleted.

The standard library also provides a helper function, std::make_shared,
that constructs a new resource (on the free store) and returns a
shared_ptr to it.  std::make_shared  is a tempate function that takes as
many parameters as required by the type being constructed

# Problems with circular references

```
class A;
class B;
void bind (std::shared_ptr<A>& a, std::shared_ptr<B>& b);

class A
{
public:
  void opA() { cout << "A::opA()" << endl; }

private:
  friend void bind (shared_ptr<A>& a, shared_ptr<B>& b);
  std::shared_ptr<B> pB;
};
```

```
class B
{
  public:
  void opB() { cout << "B::opB()" << endl; }
  void go();

private:
  friend void bind (shared_ptr<A>& a, shared_ptr<B>&
b);
  std::shared_ptr<A> pA;
};
```

Note the circular
reference

Here we have a common situation: Two classes with a bi-directional
association.  We have modelled the association this time using shared_ptrs
rather than raw pointers.

The bind function allows us to connect objects together at construction
time.  Note in this case the bind() function takes shared_ptrs as
parameters, rather than references to objects.

# Constructing the objects

```
void bind(std::shared_ptr<A>& a,
std::shared_ptr<B>& b)
{
  a->pB = b;
  b->pA = a;
}

int main()
{
  auto p1 = std::make_shared<A>();
  auto p2 = std::make_shared<B>();

  bind(p1, p2);

  p2->go();
}
```
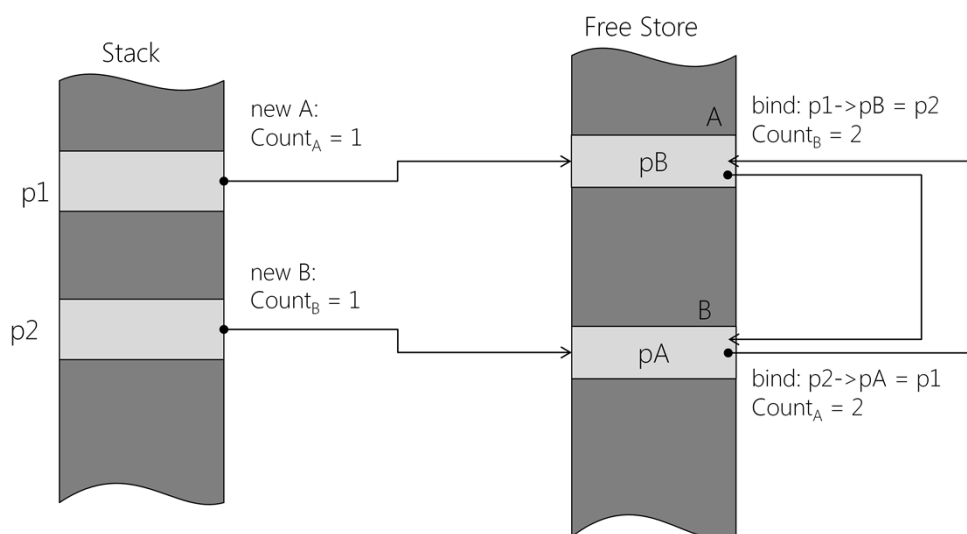
*What's the problem with this code?*

The bind function builds the association between the objects. The copy of the `shared_ptrs` inside the `bind()` is what causes the problem: this code will cause a memory leak!

But how?...

# Construction

Stack

Free Store

new A:
$Count_A = 1$

new B:
$Count_B = 1$

p1

p2

A

pB

B

pA

bind: p1->pB = p2
$Count_B = 2$
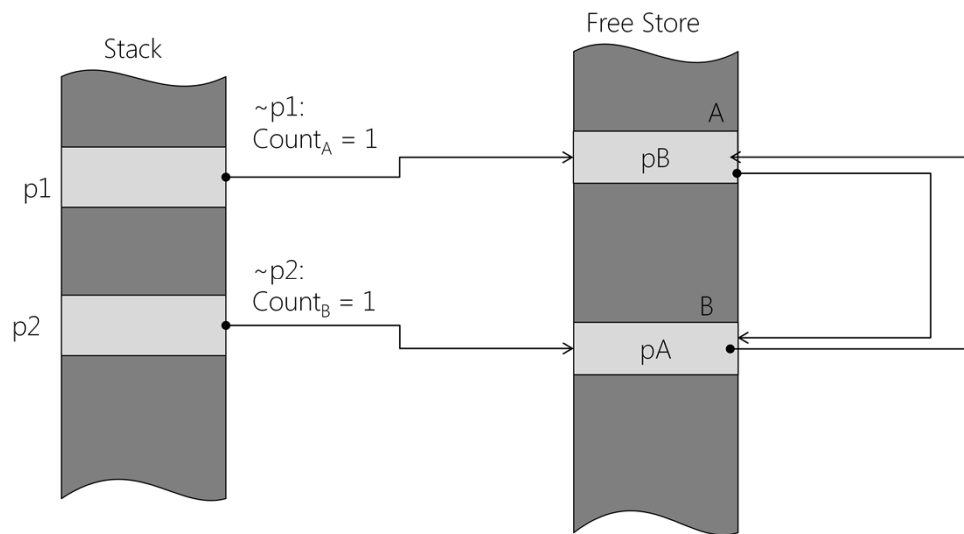
bind: p2->pA = p1
$Count_A = 2$

Let's examine the problem.

When we construct the A and B objects we get two `shared_ptrs`: p1 and p2. Their reference counts are set to 1.

The `bind` function makes copies of p1 and p2, in `p2->pA` and `p1->pB` respectively. This increments the reference counts on the `shared_ptrs`.

There are now two shared pointers referencing each A or B object.

So far, there is no problem.

# Destruction…

Stack | Free Store diagram

```
Stack                                    Free Store

          ~p1:                                    A
          Count_A = 1                      pB
p1                    ──────────→

          ~p2:                                    B
          Count_B = 1                      pA
p2                    ──────────→
```

When p1 goes out of scope its reference count is decremented.  However, since there is another shared_ptr referencing the resource (B::pA) the A object is not deleted.

Similarly, when p2 goes out of scope its reference count is decremented, but the B object cannot be deleted since A::pB still references it.

Now we have a problem:  memory for A and B cannot be deleted since pA and pB reference it; but they cannot be accessed any more (because p1 and p2 are no longer in scope!)

The solution is to use a std::weak_ptr

# std::weak_ptr

```
#include <memory>
void func(std::weak_ptr<int> p);

int main()
{
  auto shared = std::make_shared<int>(100);   // shared ctor: count = 1
  std::weak_ptr<int> weak(shared);            // weak ctor:   count = 1

  *weak = 200;                                // ERROR!

  func(weak);                                 // weak copy:   count = 1

}                                             // weak dtor:   count = 1
                                              // shared dtor: count = 0

void func(std::weak_ptr<int> p)               // p:   count = 1
{
  if(!p.expired())                            // Is p valid?
  {
    std::shared_ptr<int> temp(p);             // temp ctor:   count = 2
    *temp = 200;
  }                                           // temp dtor:   count = 1
}                                             // p dtor:      count = 1
```

A std::weak_ptr is related to a shared_ptr. A std::weak_ptr can be assigned to a std::shared_ptr; but doing so does not increment the reference count on the resource.

Since weak_ptrs can have a different lifetime to their associated shared_ptr there is a chance the shared_ptr could go out of scope (and delete its resource) before the weak_ptr is destroyed. A weak_ptr can therefore be *invalid* - that is, referencing a resource that is no longer viable. You should use the expired() method on the weak_ptr to see if it is still valid, before attempting to access it.

You cannot directly use a weak_ptr. You must convert it back to a shared_ptr first.

# Resolving circular dependencies

```cpp
class B
{
  public:
  void opB() { cout << "B::opB()" << endl; }
  void go();

private:
  friend void bind (shared_ptr<A>& a, shared_ptr<B>& b);
  std::weak_ptr<A> pA;
};
```
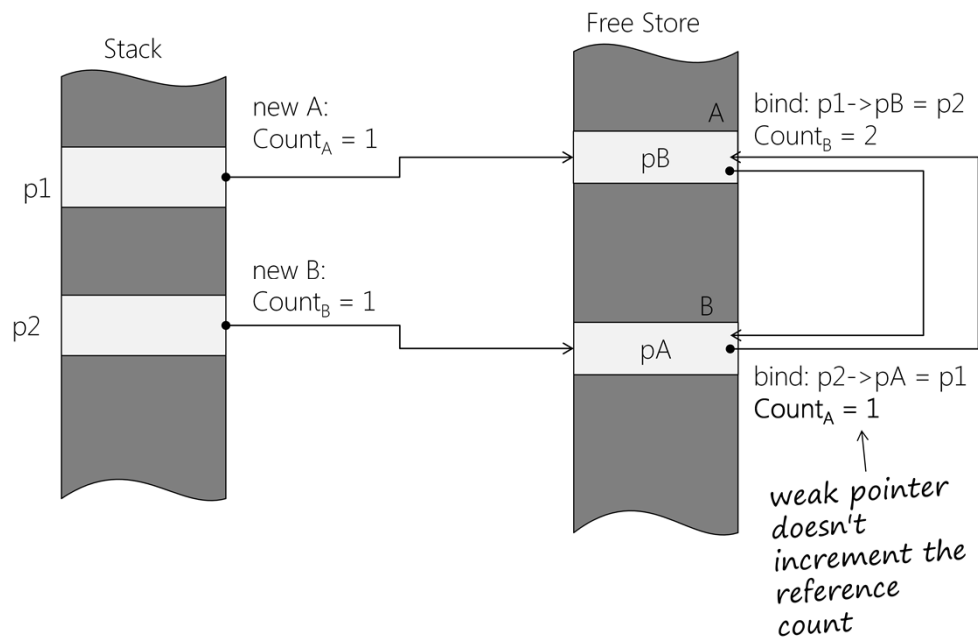
Either A or B (or both)
could use weak_ptrs

```cpp
void B::go()
  {
    if (!pA.expired())          // Check to see if pA
    {                           // is valid.
      std::shared_ptr<A> ptr(pA); // Construct a temporary
                                  // shared_ptr to access
      ptr->opA();
    }
  }
```

Using weak pointers solves our previous problem.

In this example you could make either, or both, pointers weak if you wish.

# Construction with weak_ptr

Stack

Free Store

new A:
Count$_A$ = 1

new B:
Count$_B$ = 1

p1

p2

A

pB

B

pA

bind: p1->pB = p2
Count$_B$ = 2

bind: p2->pA = p1
Count$_A$ = 1

*weak pointer doesn't increment the reference count*

Since B uses a `weak_ptr` to associate with A, when the objects are bound together the reference count for the A object remains a 1.

# Destruction with weak_ptr…

Stack

Free Store

~p1:
$Count_A = 0$

p1

~p2:
$Count_B = 0$

p2

A

pB

~A:
$Count_B = 1$

B

pA

~B:
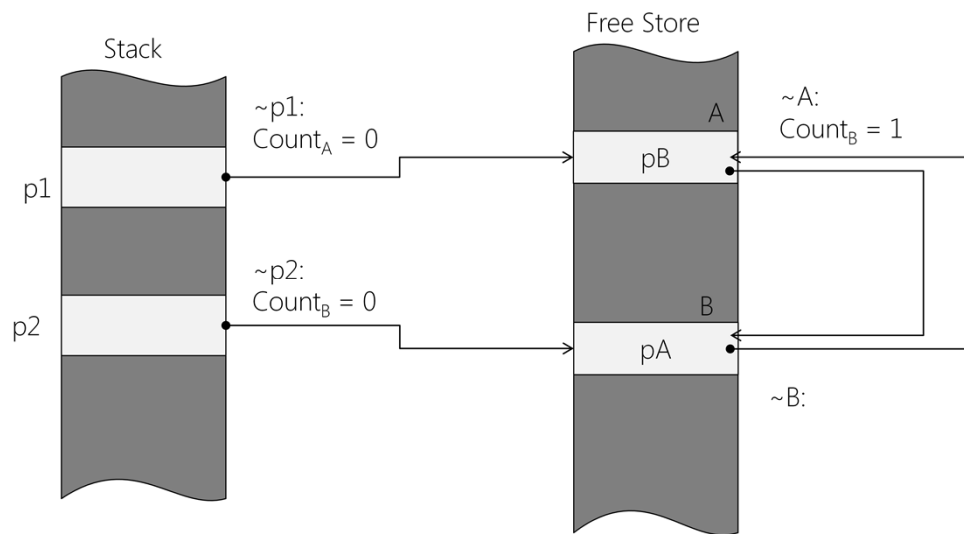
When **p1** is destroyed it decrements its reference count.  Since B uses a `weak_ptr` to refer to A, **p1**'s reference count is decremented to zero; and the A object is deleted.

As the A object is destroyed its `shared_ptr` (pB) goes out of scope, decrementing the reference count (down to 1)

Now, when **p2** is destroyed its reference count is decremented to zero; and its resource is deleted.

# Key points

Raw pointers in C++ require careful (manual) management from the programmer.

Using 'smart' pointers uses RAII / RDID to allow automatic management of resource handling.

std::unique_ptr should be used for management of a resource within a single scope (block).

std::shared_ptr should be used for managing resources across scopes

std::weak_ptr resolves problems with circular references on shared_ptrs