[368] Smart Pointers

Tyler Caraza-Harter

Outline

Worksheet and TopHat

Resources

Unique Pointers

Demos

- Unique Pointers
- File I/O

Shared Pointers

Demos

What will you learn today?

Learning objectives

- manage resources using the RAII pattern
- write code that uses smart pointers (and avoids regular pointers)
- describe how shared_pointers using reference counting
- identify scenarios where share_pointers leak

Outline

Worksheet and TopHat

Resources

Unique Pointers

Demos

- Unique Pointers
- File I/O

Shared Pointers

Demos

Resources

Examples of resources

- stack memory
- heap memory
- file handles
- sockets
- web tokens
- threads
- processes
- locks
- ...

Goal: don't retain resources we're not using (e.g., don't leak memory), but don't release resources we're still using!

reminder: don't forget about exceptions!

Resources

Examples of resources

- stack memory ← release upon function return (Java, Python, C++)
- heap memory
- file handles
- sockets
- web tokens
- threads
- processes
- locks
- ...

exception safe

Goal: don't retain resources we're not using (e.g., don't leak memory), but don't release resources we're still using!

reminder: don't forget about exceptions!

Resources

Examples of resources

- stack memory
- heap memory ← **garbage collector** (Java, Python), or **it's your job** (C++)
- file handles
- sockets
- web tokens
- threads
- processes
- locks
- ...

easier, fewer bugs

less overhead
release sooner
more "online"

destructors are crucial!

Goal: don't retain resources we're not using (e.g., don't leak memory), but don't release resources we're still using!

reminder: don't forget about exceptions!

Resources

Examples of resources

- stack memory
- heap memory
- file handles
- sockets
- web tokens
- threads
- processes
- locks
- ...

← **it's your job** (Python, Java, and C++)

might still need to do reference counting in a language like Python!

can still leverage C++ destructors

Python+Java have worse primitives, like try/finally, with statement

Goal: don't retain resources we're not using (e.g., don't leak memory), but don't release resources we're still using!

reminder: don't forget about exceptions!

Heap vs. File

```
# Python
s = "hello " + name            garbage collected
f = open("file.txt", "w")      leaks!
f.write(s)

// Java
public static void main(String[] args) throws IOException {
  String s = new String("hello " + name);  garbage collected
  BufferedWriter f = new BufferedWriter(   leaks!
    new FileWriter("file.txt")
  );
  f.write(str);
}

// C++
int main() {
  string* s = new string("hello " + name);   leaks!
  ofstream* f = new ofstream("file.txt");    leaks!
  *f << *s;
}
```

Observation: other languages can leak too!

Heap vs. File

```
# Python
s = "hello " + name      garbage collected
f = open("file.txt", "w")
f.write(s)
f.close()   manual cleanup

// Java
public static void main(String[] args) throws IOException {
  String s = new String("hello " + name);  garbage collected
  BufferedWriter f = new BufferedWriter(
    new FileWriter("file.txt")
  );
  f.write(str);
  f.close();   manual cleanup
}

// C++
int main() {
  string* s = new string("hello " + name);
  ofstream* f = new ofstream("file.txt");
  *f << *s;
  delete s;   manual cleanup
  delete f;   manual cleanup
}
```

Observation: C++ handles different resource types more consistently

ofstream destructor calls close!

Heap vs. File

```
# Python
s = "hello " + name
f = open("file.txt", "w")
f.write(s)
f.close()
```

garbage collected *eventually*

manual cleanup

```
// Java
public static void main(String[] args) throws IOException {
  String s = new String("hello " + name);
  BufferedWriter f = new BufferedWriter(
    new FileWriter("file.txt")
  );
  f.write(str);
  f.close();
}
```

garbage collected *eventually*

manual cleanup

Observation: C++ releases memory back sooner

```
// C++
int main() {
  string* s = new string("hello " + name);
  ofstream* f = new ofstream("file.txt");
  *f << *s;
  delete s;
  delete f;
}
```

manual cleanup *right now*

manual cleanup

Heap vs. File

```
# Python
s = "hello " + name        garbage collected
f = open("file.txt", "w")
f.write(s) exception!
f.close()       leak!

// Java
public static void main(String[] args) throws IOException {
  String s = new String("hello " + name);  garbage collected
  BufferedWriter f = new BufferedWriter(
    new FileWriter("file.txt")
  );
  f.write(str); exception!
  f.close();      leak!
}

// C++
int main() {
  string* s = new string("hello " + name);
  ofstream* f = new ofstream("file.txt");
  *f << *s; exception!
  delete s;      leak!
  delete f;      leak!
}
```

Observation: exceptions make resource management trickier!

with, finally, destructor

```
# Python
s = "hello " + name
with open("file.txt", "w") as f:
  f.write(s)
```

"with" closes file for us

```
// Java
public static void main(String[] args) throws IOException {
  String s = new String("hello " + name);
  try (BufferedWriter f = new BufferedWriter(...)) {
    f.write(str);
  }
}
```

"try with resources" closes file for us ("finally" in older Java code)

```
// C++
int main() {
  auto s = string(hello " + name);
  auto f = ofstream("file.txt");
  f << s;
}
```

Observations:

- string and ofstream can be on stack
- string **destructor** calls delete on char array
- ofstream **destructor** calls close on file handle

Lifetime

```
// C++
class PrimeWriter {
  ofstream file{"primes.txt"};
  int prime{2};
public:
  void WriteNext() {
    file << prime << "\n";
    // TODO: find next prime...
  }
};

int main() {
  PrimeWriter pw;
  pw.WriteNext();
}
```

pw removed from stack, PrimeWrite destructor called and ofstream destuctor called (closing primes.txt)

Observation: destructor pattern is more general than a "with resources" pattern because resource lifetime doesn't always correspond to a block of code

RAII Resource Management

Resource Acquisition Is **Initialiation**

for example, opening a file

init => constructor
(open the file in the constructor)

Ideas

- every resource is owned by an object
- acquire resource: constructor
- release resource: destructor
- resource is held for duration of object's lifetime

RAII Resource Management

Resource Acquisition Is **Initialiation**

for example, opening a file — *init => constructor (open the file in the constructor)*

Ideas

- every resource is owned by an object
- acquire resource: constructor
- release resource: destructor
- resource is held for duration of object's **lifetime**

```
void f() {
    MyClass obj;
}
```

lifetime: until f returns

```
void f() {
    {
        MyClass obj;
        ...
    }
    ...
}
```

lifetime: this block of code

```
MyClass obj; // global
```

lifetime: until program exits

```
class OtherClass {
    MyClass obj;
}
```

lifetime: same as that of OtherClass

```
vector<MyClass> vec{MyClass(...), ...};
```

lifetime: until vector is released, cleared, resized, etc.

Outline

Worksheet and TopHat

Resources

Unique Pointers

Demos

- Unique Pointers
- File I/O

Shared Pointers

Demos

Unique Pointers

Idea

- assume we're the only pointer to an object
- then we can automatically delete it when done!
- prevents you from making common programming mistakes (double free, leak)

```
class unique_ptr {
    int* ptr;
    unique_ptr(int* ptr) : ptr(ptr) {}
    ~unique_ptr() {
        if (ptr)
            delete ptr;
    }
    // do NOT allow copying (whole point is to not have
    // two pointers to same object)
    // DO allow move: a new pointer can point to the
    // object if the old pointer is set to nullptr
}
```

only member is ptr. sizeof(unique_ptr) == sizeof(ptr)

simplified unique_ptr to an integer
(actual implementation is generic)

Access

```
auto coord1 = unique_ptr<Coord>(new Coord(3, 4));
auto coord2 = new Coord(5, 6));
```

accesing through a pointer would be annoying!

```
cout << coord1.ptr->x << "\n";
cout << coord2->x << "\n";
```

Access

```
auto coord1 = unique_ptr<Coord>(new Coord(3, 4));
auto coord2 = new Coord(5, 6));


cout << coord1->x << "\n";
cout << coord2->x << "\n";

// overloading -> and *
class unique_ptr {
    Coord* ptr;
    Coord* operator->() {
        return ptr;
    }
    Coord& operator*() {
        return *ptr;
    }
    ...
}
```

after operator->, perform another -> on the returned result

return reference so we can modify it

Creation

```
auto coord1 = unique_ptr<Coord>(new Coord(3, 4));
auto coord2 = make_unique<Coord>(3,4);
```

Advantages of make_unique

- only mention "Coord" once
- with smart pointers, we can nearly always avoid "new" -- avoiding it here lets us search to identify possible bugs
- exception safety

Exception Safety

```
f(unique_ptr<A>(new A), unique_ptr<B>(new B))
```

Possible order

- new A
- new B ← if we have an exception here, A leaks!
- unique_ptr<A> constructor
- unique_ptr<B> constructor

Outline

Worksheet and TopHat

Resources

Unique Pointers

Demos

- Unique Pointers
- File I/O

Shared Pointers

Demos

Outline

Worksheet and TopHat

Resources

Unique Pointers

Demos

- Unique Pointers
- File I/O

Shared Pointers

Demos

Shared Pointers

Unique Pointers

- wrap a raw pointer inside a unique_ptr
- when the unique_ptr goes out of scope (e.g., it was on the stack), automatically call delete on the raw pointer
- no leaks/double delete because we take care (e.g., deleting copy constructors) to prevent multiple pointers refer to the same address!

What if we want multiple pointers to the same address?

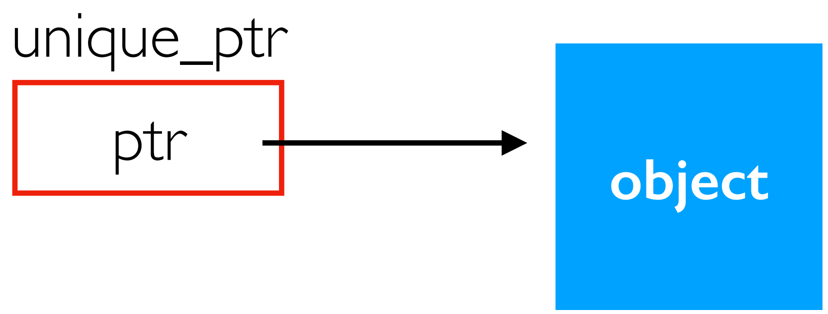
Observations:

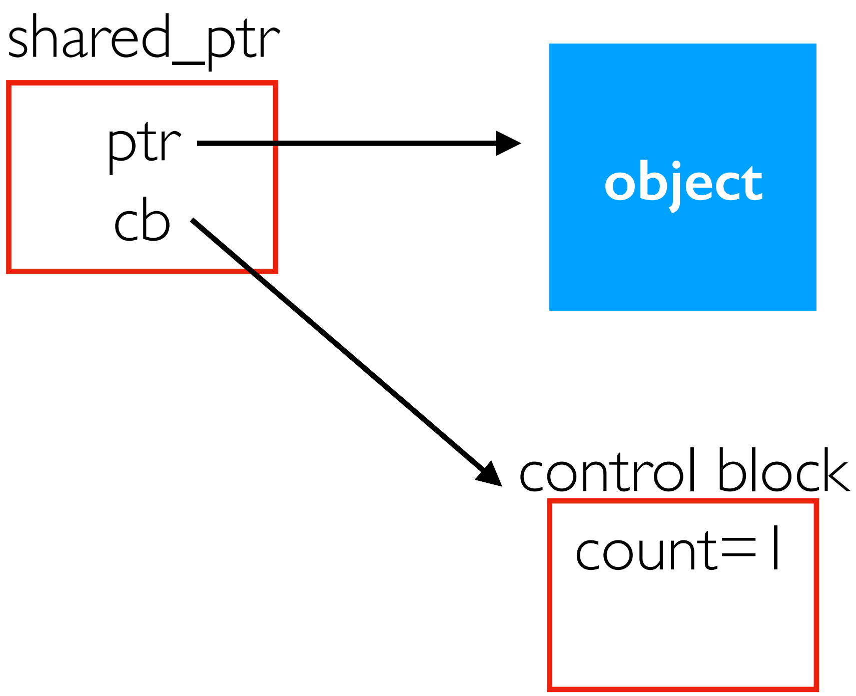
- cannot delete while there are still active pointers (corruption!)
- cannot delete later than that (leak!)
- cannot delete more than one (double free!)

Solution: maintain a reference count that indicates how many active pointers there are. When it goes to zero, free the object!

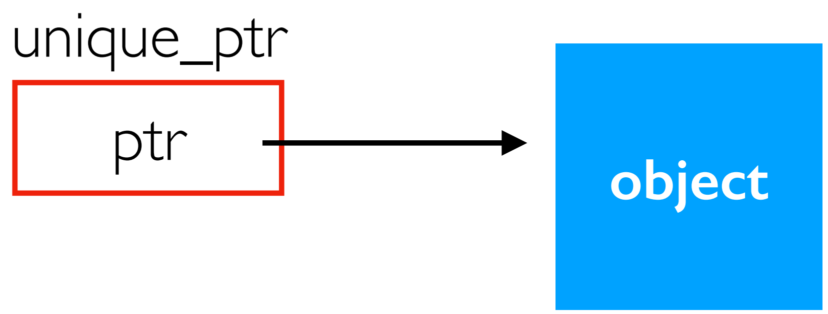
Unique Pointers

Shared Pointers



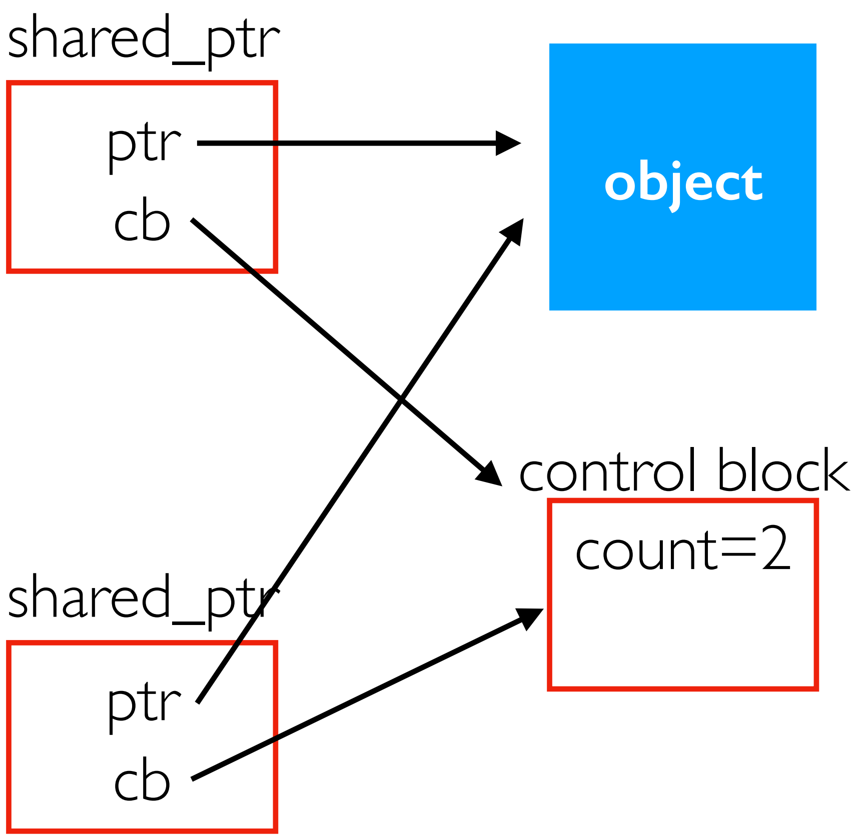


Unique Pointers



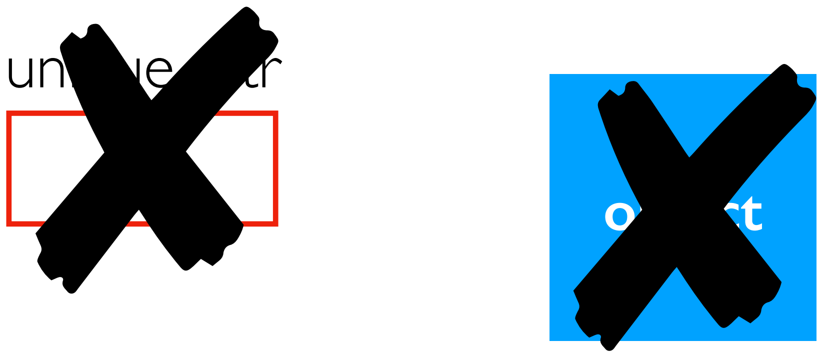
cannot copy unique_ptr!

Shared Pointers



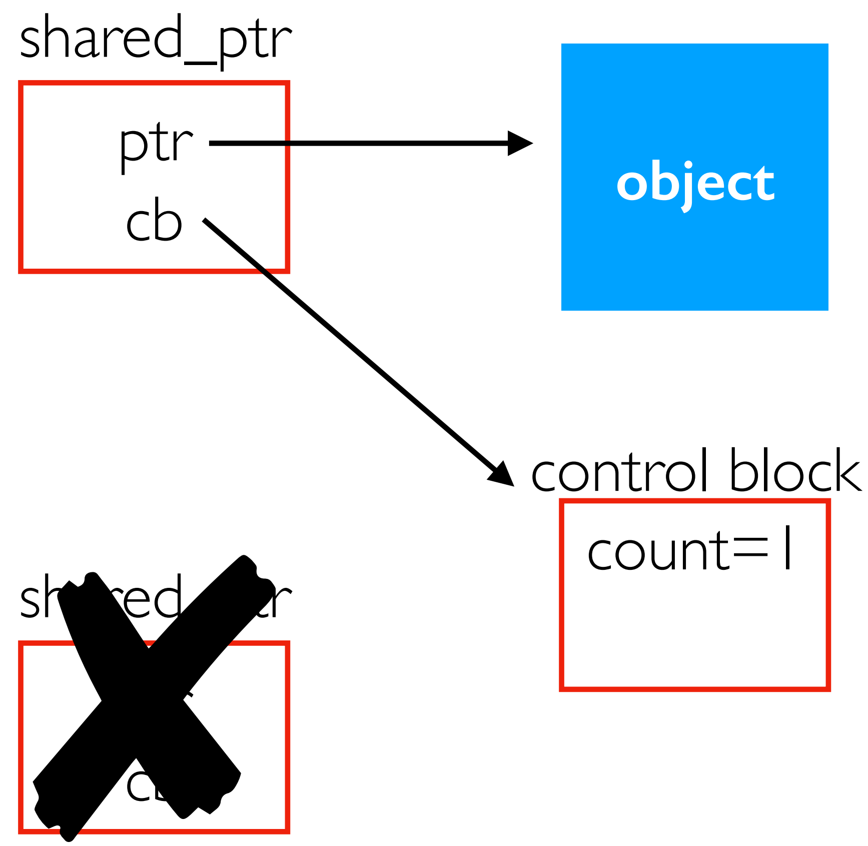
copying a shared_ptr increments the reference count in the control block

Unique Pointers



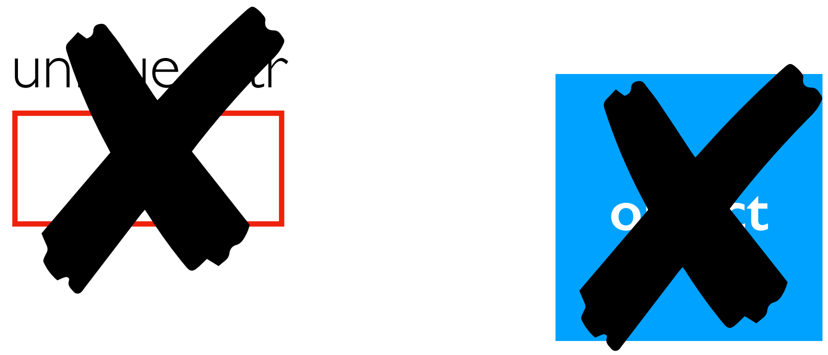
destroying the unique_ptr
deletes the object

Shared Pointers

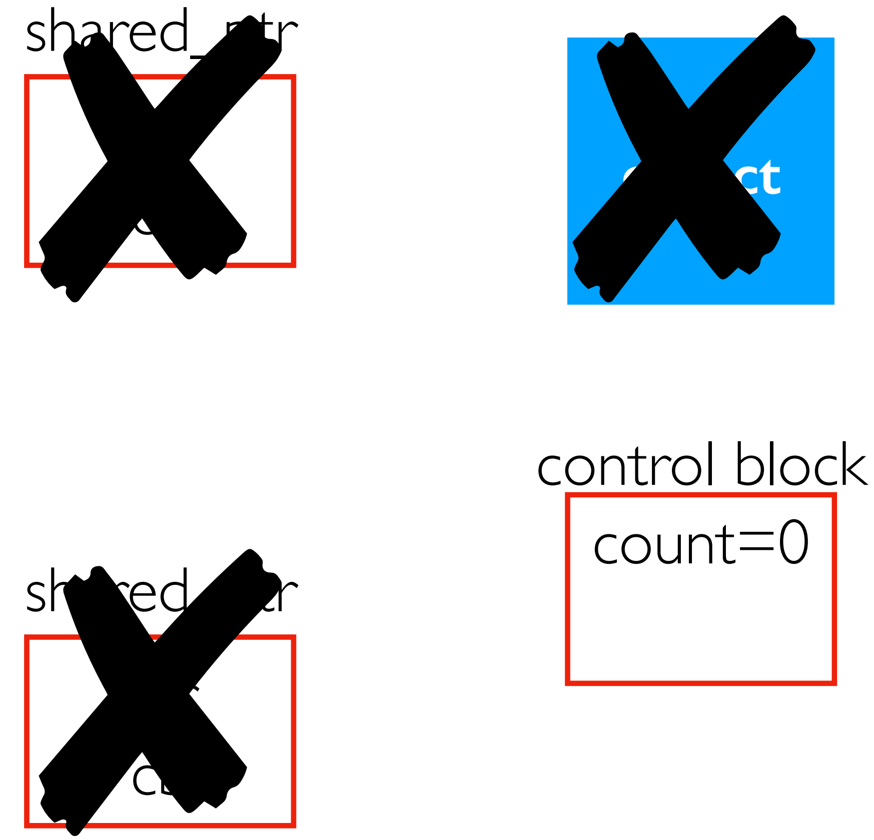


destroying a shared_ptr subtracts one
from the reference count

Unique Pointers

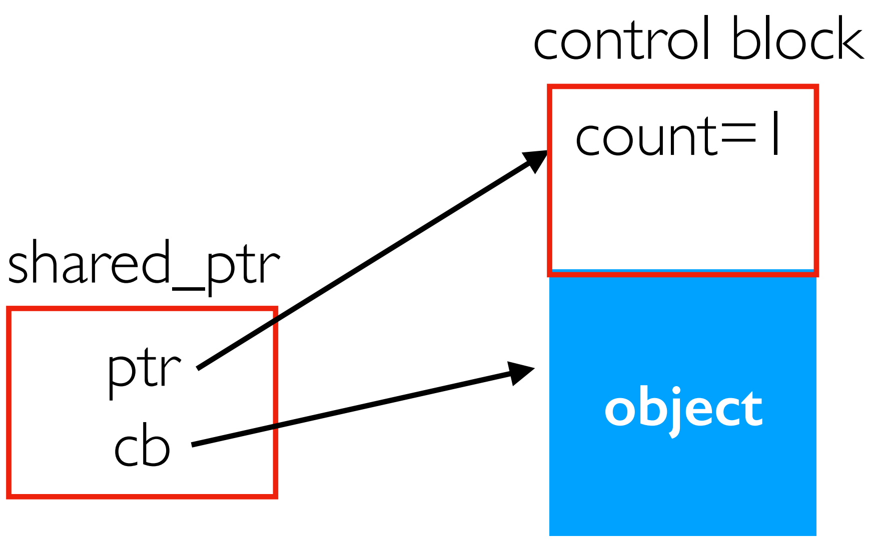


Shared Pointers



delete object when the reference count goes to zero

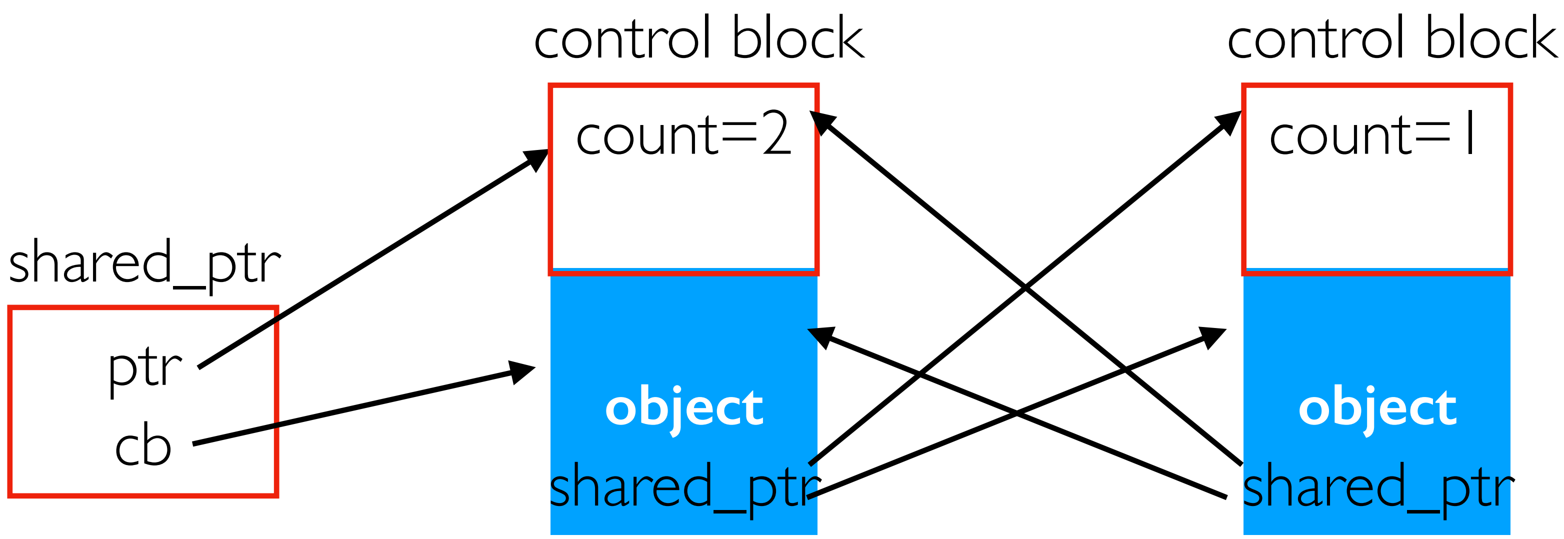
make_shared



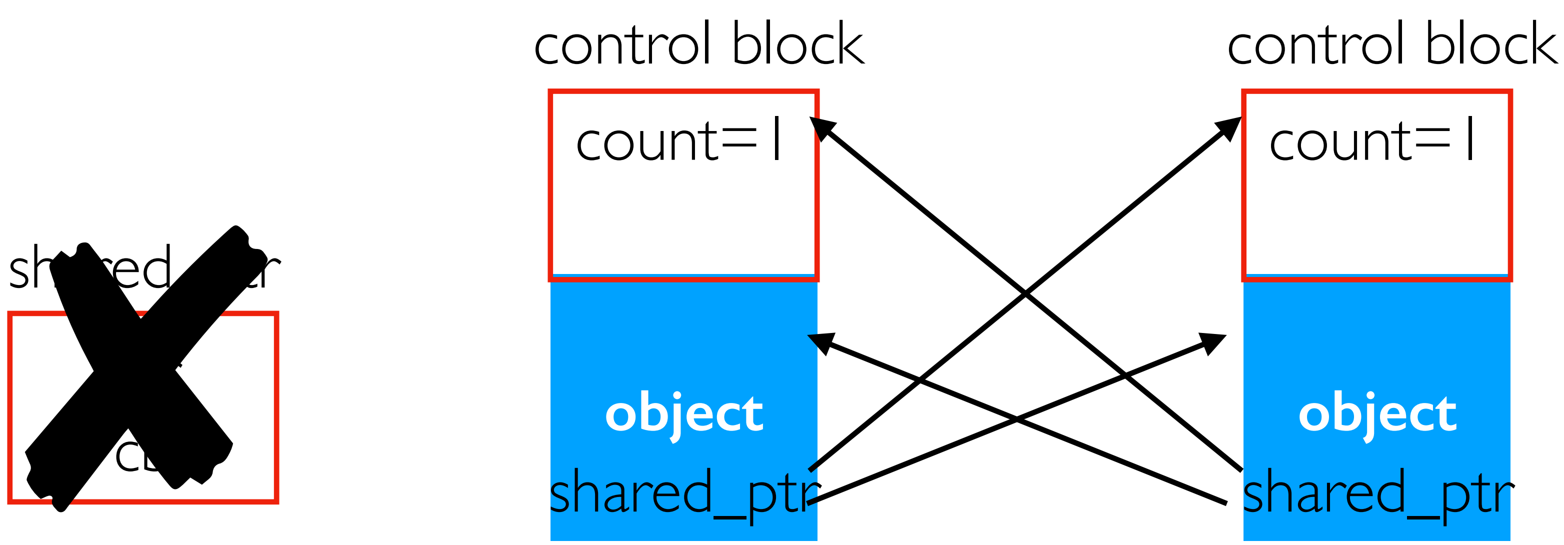
make_shared advantages

- concise syntax
- exception safe
- cache-friendly layout (control block and associated object adjacent)

Cycles


control block
count=2
control block
count=1
shared_ptr
ptr
cb
object
shared_ptr
object
shared_ptr

Cycles



shared_ptr's are not as advanced as an actual garbage collector!

- GC can detect "islands" of related objects, shared_ptrs cannot
- it's your job (e.g., by designing references to avoid loops, or writing extra cleanup code)

Outline

Worksheet and TopHat

Resources

Unique Pointers

Demos

- Unique Pointers
- File I/O

Shared Pointers

Demos