Plugging the leaks

SMART POINTERS

C++11, 14, 17

- Most of what we have taught you in this class are language features that were part of C++ since the C++98 standard
- New, helpful features have been added in C++11, 14, and now 17 standards
  - Beware: compilers are often a bit slow to implement the standards so check the documentation and compiler version
  - You often must turn on special compile flags to tell the compiler to look for C++11 features, etc.
    - For g++ you would need to add: **-std=c++11** or **-std=c++0x**
- Many of the features in the these revisions to C++ are originally part of 3rd party libraries such as the Boost library

Pointers or Objects? Both!

- In C++, the dereference operator (*) should appear before…
  - A pointer to an object
  - An actual object
- "Good" answer is
  - A Pointer to an object
- "Technically correct" answer…
  - EITHER!!!!
- Due to operator overloading we can make an object behave as a pointer
  - Overload operator *, &, ->, ++, etc.

```
class Thing
{

};

int main()
{
  Thing t1;
  Thing *ptr = &t1

  // Which is legal?
  *t1;
  *ptr;
}
```

A "Dumb" Pointer Class

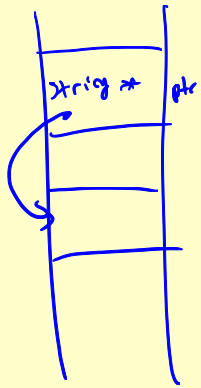
- We can make a class operate like a pointer
- Use template parameter as the type of data the pointer will point to
- Keep an actual pointer as private data
- Overload operators
- This particular class doesn't really do anything useful
  - It just does what a normal pointer would do

```
template <typename T>
class dumb_ptr
{ private:
   T* p_;
  public:
   dumb_ptr(T* p) : p_(p) { }
   T& operator*() { return *p_; }
   T* operator->() { return p_; }
   dumb_ptr& operator++()  // pre-inc
    { ++p_; return *this; }
};

int main()
{
  int data[10];
  dumb_ptr<int> ptr(data);

  for(int i=0; i < 10; i++){
    cout << *ptr;  ++ptr;
  }
}
```

A "Useful" Pointer Class

- I can add automatic memory deallocation so that when my local "unique_ptr" goes out of scope, it will automatically delete what it is pointing at

```
template <typename T>
class unique_ptr
{ private:
   T* p_;
  public:
   unique_ptr(T* p) : p_(p) { }
   ~unique_ptr() { delete p_; }
   T& operator*() { return *p_; }
   T* operator->() { return p_; }
   unique_ptr& operator++() // pre-inc
    { ++p_; return *this; }
};

int main()
{
  unique_ptr<Obj> ptr(new Obj);
  // ...
  ptr->all_words();
  // Do I need to delete Obj?
}
```



A "Useful" Pointer Class

- What happens when I make a copy?
- Can we make it impossible for anyone to make a copy of an object?
  - Remember C++ provides a default "shallow" copy constructor and assignment operator

```
template <typename T>
class unique_ptr
{ private:
   T* p_;
  public:
   unique_ptr(T* p) : p_(p) { }
   ~unique_ptr() { delete p_; }
   T& operator*() { return *p_; }
   T* operator->() { return p_; }
   unique_ptr& operator++() // pre-inc
    { ++p_; return *this; }
};

int main()
{
  unique_ptr<Obj> ptr(new Obj);
  unique_ptr<Obj> ptr2 = ptr;
  // ...
  ptr2->all_words();
  // Does anything bad happen here?
}
```

Hiding Functions

- Can we make it impossible for anyone to make a copy of an object?
  - Remember C++ provides a default "shallow" copy constructor and assignment operator
- Yes!!
  - Put the copy constructor and operator= declaration in the private section…now the implementations that the compiler provides will be private (not accessible)
- You can use this technique to hide "default constructors" or other functions

```
template <typename T>
class unique_ptr
{ private:
    T* p_;
  public:
    unique_ptr(T* p) : p_(p) { }
    ~unique_ptr() { delete p_; }
    T& operator*() { return *p_; }
    T* operator->() { return p_; }
    unique_ptr& operator++() // pre-inc
     { ++p_; return *this; }
  private:
    unique_ptr(const UsefultPtr& n);
    unique_ptr& operator=(const
                        UsefultPtr& n);
};

int main()
{
  unique_ptr<Obj> ptr(new Obj);
  unique_ptr<Obj> ptr2 = ptr;
  // Try to compile this?
}
```

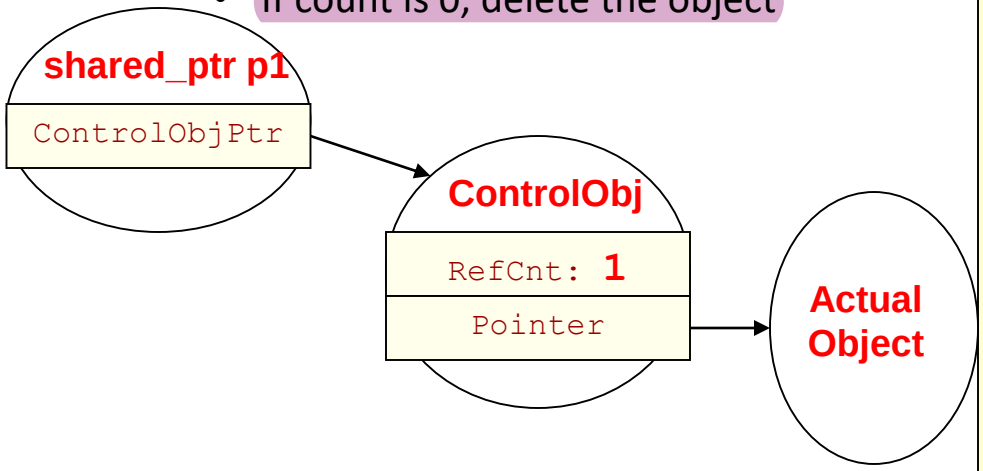
A "shared" Pointer Class

- Could we write a pointer class where we can make copies that somehow "know" to only delete the underlying object when the last copy of the smart pointer dies?
- Basic idea
  - shared_ptr class will keep a count of how many copies are alive
  - shared_ptr destructor simply decrements this count
    - If count is 0, delete the object

```
template <typename T>
class shared_ptr
{ public:
   shared_ptr(T* p);
   ~shared_ptr();
   T& operator*();
   shared_ptr& operator++();
}
shared_ptr<Obj> f1()
{
  shared_ptr<Obj> ptr(new Obj);
  cout << "In F1\n" << *ptr << endl;
  return ptr;
}

int main()
{
  shared_ptr<Obj> p2 = f1();
  cout << "Back in main\n" << *p2;
  cout << endl;
  return 0;
}
```

A "shared" Pointer Class

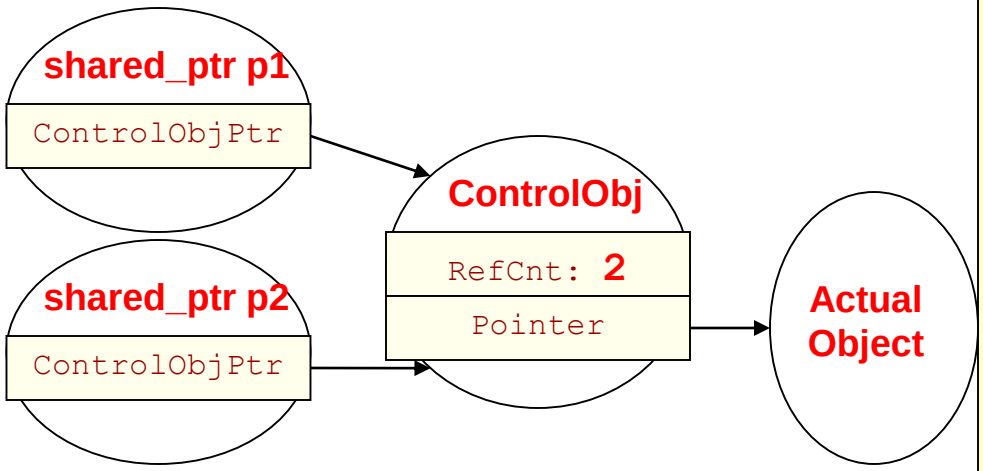
- Basic idea
  - shared_ptr class will keep a count of how many copies are alive
  - Constructors/copies increment this count
  - shared_ptr destructor simply decrements this count
    - If count is 0, delete the object




```
int main()
{
  shared_ptr<Obj> p1(new Obj);
  doit(p1);
  return 0;
}

void doit(shared_ptr<Obj> p2)
{
 if(...){
     shared_ptr<Obj> p3 = p2;

 }
}
```

A "shared" Pointer Class

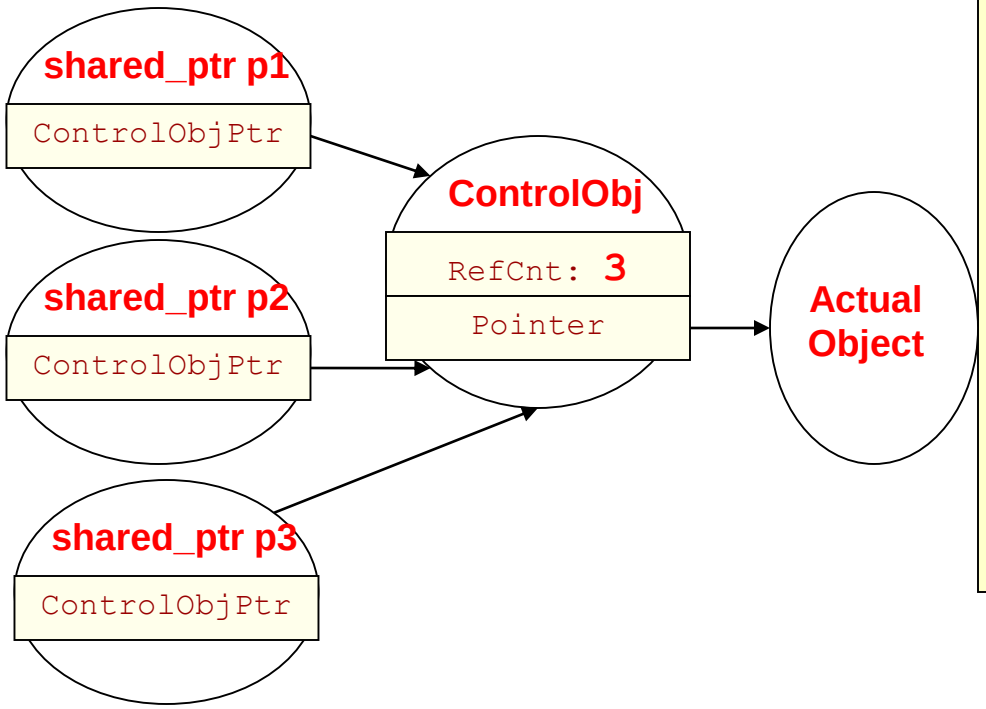
- Basic idea
  - shared_ptr class will keep a count of how many copies are alive
  - shared_ptr destructor simply decrements this count
    - If count is 0, delete the object




```
int main()
{
  shared_ptr<Obj> p1(new Obj);
  doit(p1);
  return 0;
}

void doit(shared_ptr<Obj> p2)
{
 if(...){
     shared_ptr<Obj> p3 = p2;

 }
}
```

A "shared" Pointer Class

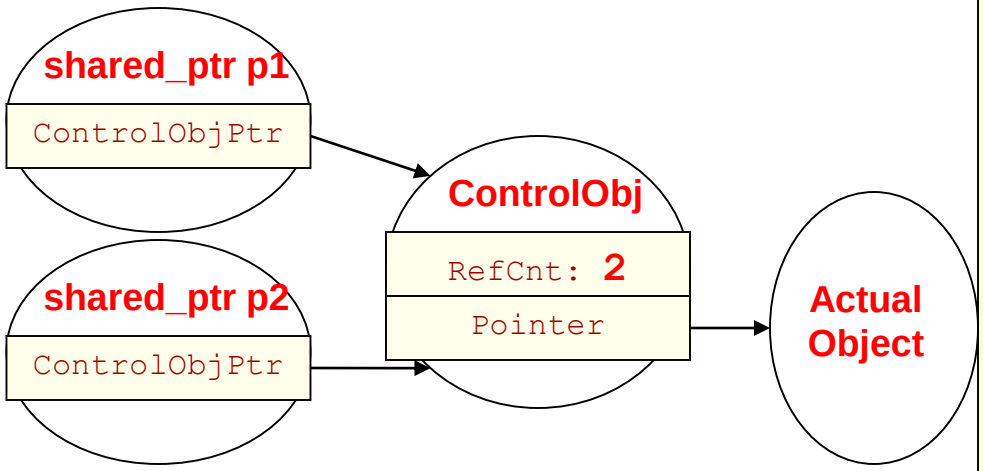
- Basic idea
  - shared_ptr class will keep a count of how many copies are alive
  - shared_ptr destructor simply decrements this count
    - If count is 0, delete the object




```
int main()
{
  shared_ptr<Obj> p1(new Obj);
  doit(p1);
  return 0;
}

void doit(shared_ptr<Obj> p2)
{
 if(...){
     shared_ptr<Obj> p3 = p2;

 }
}
```

A "shared" Pointer Class

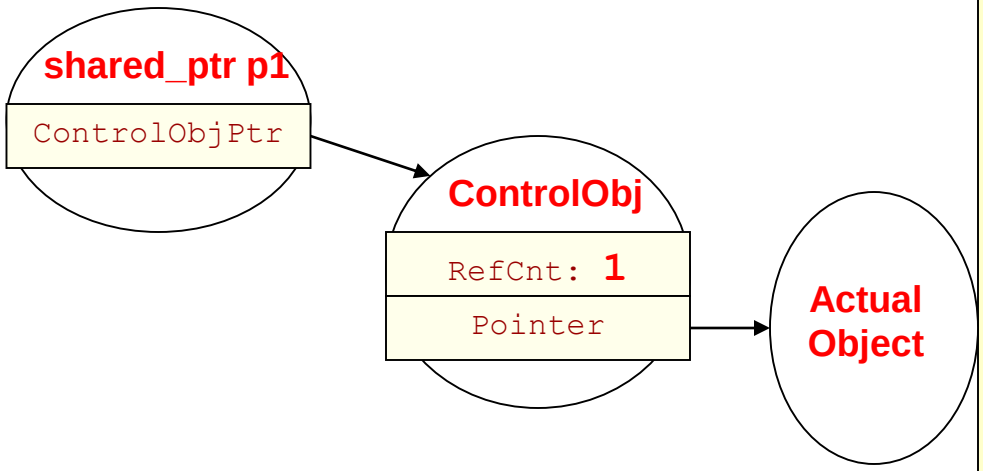
- Basic idea
  - shared_ptr class will keep a count of how many copies are alive
  - shared_ptr destructor simply decrements this count
    - If count is 0, delete the object




```
int main()
{
  shared_ptr<Obj> p1(new Obj);
  doit(p1);
  return 0;
}

void doit(shared_ptr<Obj> p2)
{
 if(...){
     shared_ptr<Obj> p3 = p2;

 } // p3 dies
}
```

A "shared" Pointer Class

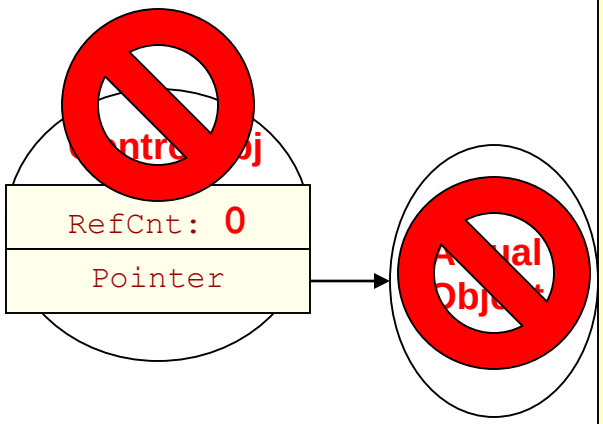
- Basic idea
  - shared_ptr class will keep a count of how many copies are alive
  - shared_ptr destructor simply decrements this count
    - If count is 0, delete the object




```
int main()
{
  shared_ptr<Obj> p1(new Obj);
  doit(p1);
  return 0;
}

void doit(shared_ptr<Obj> p2)
{
 if(...){
     shared_ptr<Obj> p3 = p2;

 } // p3 dies
} // p2 dies
```

A "shared" Pointer Class

- Basic idea
  - shared_ptr class will keep a count of how many copies are alive
  - shared_ptr destructor simply decrements this count
    - If count is 0, delete the object




```
int main()
{
  shared_ptr<Obj> p1(new Obj);
  doit(p1);
  return 0;
} // p1 dies

void doit(shared_ptr<Obj> p2)
{
 if(...){
     shared_ptr<Obj> p3 = p2;

 } // p3 dies
} // p2 dies
```

C++ shared_ptr

- C++ std::shared_ptr / boost::shared_ptr
  - Boost is a best-in-class C++ library of code you can download and use with all kinds of useful classes
- Can only be used to point at dynamically allocated data (since it is going to call delete on the pointer when the reference count reaches 0)
- Compile in g++ using '-std=c++11' since this class is part of the new standard library version

```
#include <memory>
#include "obj.h"
using namespace std;

shared_ptr<Obj> f1()
{
  shared_ptr<Obj> ptr(new Obj);
  // ...
  cout << "In F1\n" << *ptr << endl;
  return ptr;
}

int main()
{
  shared_ptr<Obj> p2 = f1();
  cout << "Back in main\n" << *p2;
  cout << endl;
  return 0;
}
```

$ g++ -std=c++11 shared_ptr1.cpp obj.cpp

C++ shared_ptr

- Using shared_ptr's you can put pointers into container objects (vectors, maps, etc) and not have to worry about iterating through and deleting them
- When myvec goes out of scope, it deallocates what it is storing (shared_ptr's), but that causes the shared_ptr destructor to automatically delete the Objs
- Think about your project homeworks…this might be (have been) nice

```
#include <memory>
#include <vector>
#include "obj.h"
using namespace std;

int main()
{
  vector<shared_ptr<Obj> > myvec;

  shared_ptr<Obj> p1(new Obj);
  myvec.push_back( p1 );

  shared_ptr<Obj> p2(new Obj);
  myvec.push_back( p2 );

  return 0;
  // myvec goes out of scope...
}
```

$ g++ -std=c++11 shared_ptr1.cpp obj.cpp

shared_ptr vs. unique_ptr

- Both will perform automatic deallocation
- Unique_ptr only allows one pointer to the object at a time
  - Copy constructor and assignment operator are hidden as private functions
  - Object is deleted when pointer goes out of scope
  - Does allow "move" operation
    - If interested read more about this on your own
    - C++11 defines "move" constructors (not just copy constructors) and "rvalue references" etc.
- Shared_ptr allow any number of copies of the pointer
  - Object is deleted when last pointer copy goes out of scope
- Note: Many languages like python, Java, C#, etc. all use this idea of reference counting and automatic deallocation (aka garbage collection) to remove the burden of memory management from the programmer