**SMART POINTERS**

(C++11, 14, 17)

**A “Dumb” Pointer Class**

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++() {++p\_; return \*this; } //pre-increment

};

int main(){

int data[10];

dumb\_ptr<int> ptr(data);

for (int i = 0; i < 10; i++){

cout << \*ptr;

++ptr;

}

}

Big three are not needed because we need to copy the pointer itself etc.

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.

**A “Useful” Pointer Class**

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++() {++p\_; return \*this; } //pre-incr

};

int main(){

unique\_ptr<string> ptr(new string);

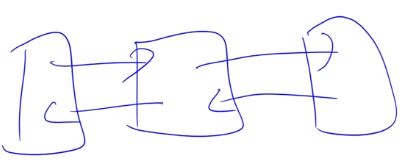
cout << \*ptr;

}

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.

You don’t have to delete your objects with this way.

But you cannot have 2 pointers that are pointing to same location (you can but that would be bad so you shouldn’t).

This is where you need 2 pointers located to same location.



in main:

unique\_ptr<Obj> ptr(new Obj);

unique\_ptr<Obj> ptr2 = ptr;

…

ptr2->all\_words();

When we go out of scope, end of main, ptr2 will delete new object. ptr will try to same thing again. That would be a crash.

**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 constructor” 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 unique\_ptr& n);

unique\_ptr& operator=(const unique\_ptr& n);

};

int main(){

unique\_ptr<Obj> ptr(new Obj);

unique\_ptr<Obj> ptr2 = ptr;

// Try to compile this?

// Copy constructor is private.

}

Now you cannot have 2 pointers pointing to the same location.

**A “Shared” Pointer Class**

Shared pointer will keep the count of pointers to the same object.

If I am pointing to the same object more than once, that object knows that i.e 5 pointers are pointing to me.

As soon as the number of counts of pointing objects drop to zero, that object says that “Okay it is time to die, I am going to delete myself.”.

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;

shared\_ptr<Obj> p1(new Obj);

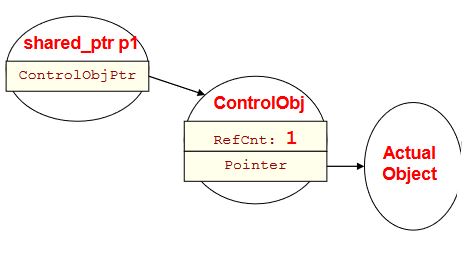
doit(p1);

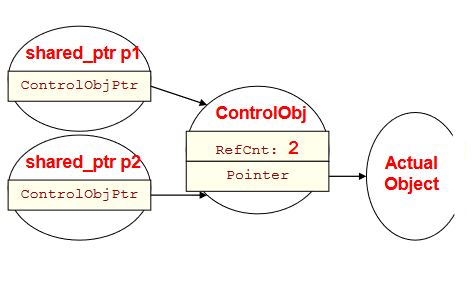
return 0;

}

void doit(shared\_ptr<Obj> p2)  
{  
 if(...){  
 shared\_ptr<Obj> p3 = p2;  
 }  
}

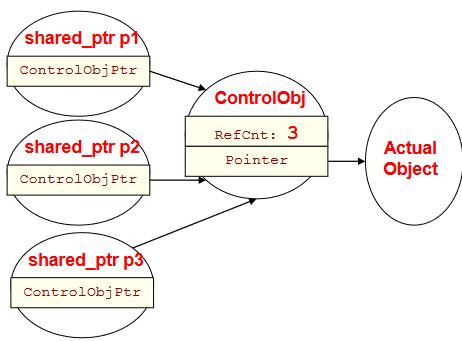
Create p1:



Making copy of shared\_ptr (call-by-value):

When we leave the doit function, p2 will die bc it is a parameter. When p2 die, destructor will decrement the count.

Creating p3:

After if block, p3 will die and count will be decremented to 2 by destructor.

After doit function block, p2 will die bc it is a parameter. When p2 die, destructor will decrement the count.

When we reach end of the main function, refcnt drops to 0. p1 and object will be deleted to.

**C++ shared\_ptr**

#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;

}

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)

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

#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...

}

use\_count function for shared\_ptr returns the number of shared\_ptr objects referring to the same managed object.

One of the shared\_ptr constructors takes a pointer. Other one uses the result of “make\_shared” global function. Both of them are used equally.

#include <iostream>

#include <memory>

#include <type\_traits>

struct C

{

C(int i) : i(i) {} //< constructor needed (until C++20)

int i;

};

int main()

{

// using `auto` for the type of `sp1`

auto sp1 = std::make\_shared<C>(12);

// Make a new C object in the heap, initialize it with 12 and take

//the pointer and keep it in shared\_ptr named sp1.

// Temporary shared pointer (std::make\_shared<C>(12);) will go

//away bc it is temporary.

static\_assert(std::is\_same\_v<decltype(sp1), std::shared\_ptr<C>>);

std::cout << sp1->i << '\n'; //will print 12

// being explicit with the type of `sp2`

std::shared\_ptr<C> sp2 = std::make\_shared<C>(13);

static\_assert(std::is\_same\_v<decltype(sp2), std::shared\_ptr<C>>);

static\_assert(std::is\_same\_v<decltype(sp1), decltype(sp2)>);

std::cout << sp2->i << '\n'; //will print 13

}

IN JAVA ONLY SMART POINTERS ARE AVAILABLE