<https://cppstyle.wordpress.com/c11-smart-pointers/>

**C++: Smart pointers, part 1**

[20111014](https://oopscenities.net/2011/10/14/c-smart-pointers/)[OOPSCENE](https://oopscenities.net/author/oopscene/)

*This is the first of several posts I wrote related to smart pointers:*

1. *Smart pointers*
2. [*unique\_ptr*](https://oopscenities.net/2013/04/09/smart-pointers-part-2-unique_ptr-2/)
3. [*More on unique\_ptr*](https://oopscenities.net/2013/08/06/c-smart-pointers-part-3-more-on-unique_ptr/)
4. [*shared\_ptr*](https://oopscenities.net/2013/10/06/smart-pointers-part-4-shared_ptr/)
5. [*weak\_ptr*](https://oopscenities.net/2014/08/03/c-smart-pointers-part-5-weak_ptr/)

Memory management in C is too error prone because keeping track of each bunch of bytes allocated and deallocated can be really confusing and stressing.

Although C++ has the same manual memory management than C, it provides us some additional features that let us to do this management easier:

* When an object is instantiated in the stack (e.g. Object o;); the C++ runtime ensure the destructor of such object is invoked when the object goes out of scope (the end of the enclosing block is reached, a premature ‘return’ is found or an exception is thrown); thus, releasing all memory and resources allocated for such object.
* (Ab)using the feature of operator overloading, we can create classes that simulate the behavior of the pointers. Such classes are called: **Smart pointers**.

So, smart pointers are classes (typically implemented as class templates, to make them highly reusable) that wrap a pointer and simulate its same behavior, but implement also some policies to release the objects “automatically”. The “magic” of smart pointers lies in the fact that they are always defined as stack variables, thus, the invocation of their destructor is guaranteed.

Consider this piece of code:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | #include <iostream>    using namespace std;    class A  {    private:      int val;    public:      A(int val) : val(val) { cout << "A ctor" << endl; }      ~A() { cout << "A dtor" << endl; }      int get\_value() const { return val; }  };    int main()  {    cout << "Enter a number: " << endl;    int n;    cin >> n;      A\* a = new A(n);    if (n == 2)      return 0;      cout << "Value entered: " << a->get\_value() << endl;    delete a;    return 0;  } |

The code shown above asks the user to enter a number, the user enters it and if the  
number is 2, the program finishes, otherwise, the program shows the number entered before exiting. If you look into the code you will realize that if the value entered equals 2, the program finishes WITHOUT calling the destructor of A. Though my example is quite trivial, forgetting to call the destructors when returning in several points in our code is one of the most common sources of memory leaks.

Smart pointers are good candidates in this point; we can write a smart pointer class template that releases the object when the smart pointer gets out of scope. To simulate the behavior of plain-old-pointers, the operator\* and operator-> must be overloaded: Look into this piece of code implementing my smart pointer:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | template <typename T>  class ptr  {    private:      T\* pointee;      public:      explicit ptr(T\* pointee) : pointee(pointee) { }      ~ptr() { delete pointee; }        T\* operator->() { return pointee; }      const T\* operator->() const { return pointee; }        T& operator\*() { return \*pointee; }      const T& operator\*() const { return \*pointee; }  }; |

And now look into a modified main() function that uses the smart pointer instead of the normal one:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | int main()  {    cout << "Enter a number: " << endl;    int n;    cin >> n;      ptr<A> a(new A(n));    if (n == 2)      return 0;      cout << "Value entered: " << a->get\_value() << endl;    return 0;  } |

My variable a is defined as a smart pointer and will hold the pointer to a new instance of A() allocated in the heap. As you can see, since the ptr<a> adeclaration is a stack declaration, the C++ runtime guarantees that when the main() method will be finished (through the premature return of through the final one), the a destructor will be invoked and it will call the destructor of the pointee object.

A very similar policy is also implemented in the deprecated std::auto\_ptr<T>and in the C++11 std::unique\_ptr<T> smart pointers.

Other approach of smart pointers is implementing a policy that will keep the track of how many smart pointers are pointing to the same object in the heap. Using reference counting, this kind of smart pointers call the destructor of the objects when the reference count has reached 0 (meaning that there are zero smart pointers pointing to such object in the heap).

A reference counter smart pointer can be implemented in a way similar to this one:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52 | struct refcntptrdata  {    T\* pointee;    int refcnt;  };    template <typename T>  class refcntptr  {    private:      refcntptrdata<T>\* data;        void release()      {        data->refcnt--;        if (data->refcnt == 0)        {          delete data->pointee;          delete data;        }      }      public:      explicit refcntptr(T\* pointee) : data(new refcntptrdata<T>())      {        data->pointee = pointee;        data->refcnt = 1;      }        refcntptr(const refcntptr<T>& source) : data(source.data)      {        data->refcnt++;      }        refcntptr<T>& operator=(const refcntptr<T>& source)      {        release();        data = source.data;        data->refcnt++;      }        ~refcntptr()      {        release();      }        T\* operator->() { return data->pointee; }      const T\* operator->() const { return data->pointee; }        T& operator\*() { return \*(data->pointee); }      const T& operator\*() const { return \*(data->pointee); }  }; |

As you can see in my implementation, the refcntptr<T> object just points to a refcntptrdata<T> object that is the one that keeps the pointer to the pointee and the reference counter.

The nice thing on this kind of pointers is that the copy constructor is very cheap because it just increments the reference counter.

C++11 has also a smart pointer implementing this policy: std::shared\_ptr<T>

See how this smart pointer can be used in the example below:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | int main()  {    refcntptr<A> a(new A(2));    refcntptr<A> b(new A(12));    refcntptr<A> c = a;    refcntptr<A> d = b;    refcntptr<A> e(new A(5));    e = b; //A=5 should be destroyed here because no one is using it anymore    b = a;      cout << "A: " << a->get\_value() << endl;    cout << "B: " << b->get\_value() << endl;    cout << "C: " << c->get\_value() << endl;    cout << "D: " << d->get\_value() << endl;    cout << "E: " << (\*e).get\_value() << endl;    } |

# unique\_ptr

C++11 ships with a set of out-of-the-box smart pointers that help us to manage the memory easily.

One of those smart pointers is the unique\_ptr.

Consider this piece of code:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | #include <iostream>  #include <ctime>    using namespace std;    class A  {    public:      A() { cout << "ctor invoked" << endl; }      virtual ~A() { cout << "dtor invoked" << endl; }      void sayHi() const { cout << "HI" << endl; }  };    class B : public A { };    void test()  {      if (clock() % 5 == 0)          throw std::exception();  }    int main()  {    A\* a = new A();    A\* b = new B();      clock\_t clk = clock();    if (clk % 2 == 0)      return -1;      test();      a->sayHi();      delete b;    delete a;  } |

If you execute this code several times, you will get three types of output:

This one:  
  
ctor invoked  
ctor invoked  
HI  
dtor invoked  
dtor invoked

Or this, that occurs when the number returned by clock() is even:  
  
ctor invoked  
ctor invoked

Or this, that occurs when the exception in test() is thrown:  
  
ctor invoked  
ctor invoked  
terminate called after throwing an instance of 'std::exception'  
what(): std::exception  
Abort trap: 6

As you see, the expected behavior just occurs in the first case, in the other ones (with a premature return or when an exception is thrown), we are not invoking delete, so we are leaving memory leaks.

What would be the perfect solution? Using stack variables, but that is not always possible: Maybe we invoke a method that returns a pointer to something, or returns a pointer of a base class to be used “polymorphically” or we need to rely on polymorphism (i.e. in virtual functions).

The ugly solution? Patching our code:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | int main()  {    A\* a = new A();    A\* b = new B();      clock\_t clk = clock();    if (clk % 2 == 0)    {      delete a;      delete b;      return -1;    }      try    {      test();    }    catch (const std::exception& ex)    {      cerr << "An expected error occurred" << endl;      delete a;      delete b;      return -2;    }      a->sayHi();      delete b;    delete a;  } |

Cons:

* Ugly
* A lot of error handling code
* Hard to maintain, a lot of duplicate code

The nice way? Using unique\_ptr.

unique\_ptr is a smart pointer that invokes automatically the destructor of the pointee object that is wrapping, when it reaches the end of its lifetime.

That mechanism is useful in these scenarios:

* You rely on objects allocated in the freestore, but you want to have them alive just in the method you are using them.
* You have variable members of your class declared as pointers to objects allocated in the freestore. Using unique\_ptr instead of raw pointers, avoid you to invoke explicitly their destructors in the destructor of your class AND ensures your objects will be properly released if some exception occurs while constructing the objects of your class.

So, the code I implemented above would look like this using unique\_ptr:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45 | #include <ctime>  #include <iostream>  #include <memory>    using namespace std;    class A  {    public:      A() { cout << "ctor invoked" << endl; }      virtual ~A() { cout << "dtor invoked" << endl; }      void sayHi() const { cout << "HI" << endl; }  };    class B : public A { };    void test()  {      if (clock() % 5 == 0)          throw std::exception();  }    int main()  {    unique\_ptr<A> a(new A { });    unique\_ptr<A> b(new B { });      clock\_t clk = clock();    if (clk % 2 == 0)    {      return -1;    }      try    {      test();    }    catch (const std::exception& ex)    {      cerr << "An expected error occurred" << endl;      return -2;    }      a->sayHi();  } |

To get the raw pointer of the pointee object inside this smart pointer, the unique\_ptr class template implements a method called .get(). This method should be only used in case you are using old code and you need to pass it raw pointers instead.

unique\_ptr deletes the copy constructor and the assignment operator but provides the move constructor and the move assignment operator to transfer the pointee to other unique\_ptr instance.

Look at this piece of code:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | #include <iostream>  #include <memory>    using namespace std;    class X  {      int x;    public:      X(int x) : x(x) { cout << "ctor invoked" << endl; }      ~X() { cout << "dtor invoked" << endl; }      void sayHi() const { cout << "HI " << x << endl; }  };    int main()  {    unique\_ptr<X> a(new X { 2 });    // unique\_ptr<X> c = a; //does not compile! no copy constructor    unique\_ptr<X> b = std::move(a); //valid: move constructor    cout << a.get() << endl;    b->sayHi();  } |

If you uncomment the second line of code in the main function, your program will not compile because the copy constructor has been explicitly removed. Instead of it, the move constructor is in place. This move constructor “steals” the pointer from a and sets it to b. As you see, when a.get() is executed, it returns a null pointer (because the pointer was moved to b).

Why is that behavior good? Because in that way the smart pointer ensure us that there is just one owner of the pointed object.

unique\_ptr is the C++11 replacement to the old auto\_ptr that has similar behavior, but that was deprecated because it does not support move semantics (and implements a “hack” in its copy constructor that moves the pointer to the target object (what is not legal for a copy constructor).

unique\_ptr default behavior consists on take ownership of a pointer created with new and that would normally be released with delete.

That is because the main unique\_ptr declaration is:

|  |  |
| --- | --- |
| 1  2 | template <typename T, typename Deleter = std::default\_delete<T>>  class unique\_ptr; |

The template class default\_delete implements a function object that performs a delete to the object pointed by the argument passed to the function object. Something like this:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | template <typename T>  class default\_delete  {  public:      void operator()(T\* obj) const      {          delete obj;      }  }; |

The idea of having the “deleter” as a template argument, lets us to use a unique\_ptr to hold a pointer to a data structure created with, say, malloc or to free the memory using some custom deallocator.

Let’s see both cases:

**1. Using with malloc**

Let’s create a struct that holds two integers:

|  |  |
| --- | --- |
| 1  2  3  4  5 | struct my\_point  {      int x;      int y;  }; |

I want to create one instance of such struct in the heap using malloc, to have something like this:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | void show\_point(const my\_point& o)  {    cout << "(" << o.x << "; " << o.y << endl;  }    void function\_that\_possibly\_throws\_exception()  {     .....     .....  }    int main()  {    my\_point\* p = static\_cast<my\_point\*>(malloc(sizeof(my\_point));    p->x = 5;    p->y = 8;    show\_point(\*p);    function\_that\_possibly\_throws\_exception();    free(p);  } |

If I want to use unique\_ptr to make sure that free is always invoked no matter if an exception occurs or not or if my function contains several return points; I need to implement a “custom deleter” to replace the std::default\_delete deleter; something like this:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | struct my\_point\_deleter  {    void operator()(my\_point\* p) const    {      free(p);    }  }; |

Take into account that my implementation is very similar to the default\_delete implementation but uses free instead of delete.

To use it into my code, I need to modify my main function to be similar to this one:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | int main()  {    unique\_ptr<my\_point, my\_point\_deleter> p(static\_cast<my\_point\*>(malloc(sizeof(my\_point)));    p->x = 5;    p->y = 8;    show\_point(\*p);    function\_that\_possibly\_throws\_exception();    //free(p); --> no need for free, the unique\_ptr will take care of this  } |

**2. Using with a custom deallocator**

For example, I want to use unique\_ptr to invoke fclose() everytime a FILE\* gets out of scope.

Look at my implementation, as you see below, you can pass the custom deleter as a function pointer in the constructor; so I can provide that to my unique\_ptralso as a lambda expression:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | int main()  {    unique\_ptr<FILE, void (\*)(FILE\*)> f(fopen("file.cpp", "r"),                                        [](FILE\* f)                                        {                                          fclose(f);                                        });      char aux[100];    while (!feof(f.get())) //f.get returns me the actual plain old pointer    {       fgets(aux, 100, f.get());       cout << aux;    }      //No need to invoke fclose() because the unique\_ptr will take care of it.  } |

C++14 will ship with a new variadic template function called make\_unique that will make easier to create objects in the heap and wrap them into unique\_ptrinstances.

So, consider this instantiation using new:

|  |  |
| --- | --- |
| 1 | unique\_ptr<my\_point> point(new my\_point { 6, 5 }); |

can be rewritten to this one, using make\_unique:

|  |  |
| --- | --- |
| 1 | auto point = make\_unique<my\_point>(6, 5); |

Pros? As Stephan T. Lavavej states, it saves to write “my\_point” twice, it is consistent with make\_shared (I will write about it in a later post), and it hides the usage of the operator new (that is good because we do not want to have our programs with new but without delete ;) ).

# shared\_ptr

As I mentioned in other posts, C++11 brings a new set of smart pointers into C++. The most useful smart pointer is shared\_ptr: Its memory management policy consists in counting the number of shared\_ptr instances that refer to the same object in the heap.

For example, if you have something like this:  
  
shared\_ptr<int> x(new int { 6 });  
shared\_ptr<int> y = x;

Your smart pointers x and y refer to the same integer created in the heap and both store the number of smart pointers that point to the same object (in our case, 2).

If we add something like:  
  
x = nullptr;

The number of smart pointers referring to the same object is decremented to 1 because only y is referring it. When y gets out of scope, the shared\_ptrdestructor is invoked automatically (because of RAII) and the reference counter is decremented again (this time to 0, because no shared\_ptr is pointing anymore to the object). Since the reference counter is 0, the object (in this case, our int { 6 }) destructor is invoked.

shared\_ptr solves a lot of memory management problems and render the naked pointers (e.g. DataType\*) unnecessary.

Consider this example in C++03:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | #include <cstdio>  #include <cstring>    class Integer  {      int n;      public:        Integer(int n) : n(n) { }        ~Integer() { printf("Deleting %d\n", n); }        int get() const { return n; }  };    int main()  {      Integer\* a = new Integer(10);      Integer\* b = new Integer(20);      Integer\* c = a;      Integer\* d = new Integer(30);      Integer\* e = b;      a = d;      b = new Integer(40);      Integer\* f = c;      b = f;        printf("%d\n", a->get());      printf("%d\n", b->get());      printf("%d\n", c->get());      printf("%d\n", d->get());      printf("%d\n", e->get());      printf("%d\n", f->get());        delete a;      delete b;      delete c;      delete e;      delete f;  } |

When it runs, it returns something like:

30  
10  
10  
30  
20  
10  
Deleting 30  
Deleting 10  
Deleting 10  
v(663) malloc: \*\*\* error for object 0x7fb4024000e0: pointer being freed was not allocated  
\*\*\* set a breakpoint in malloc\_error\_break to debug  
Abort trap: 6

The program contains two severe problems: Memory leaks and crashes. If you look at the result, the Integer(40) instance has disappeared and its destructor has never been invoked, turning it into a memory leak. Also, we created 4 instances of the Integer class and we are invoking delete with 6 variables; what occurs right there is that one object is being deleted twice, producing a crash the second time.

shared\_ptr comes to the rescue in C++11 and turns our code into something like this:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | #include <cstdio>  #include <cstring>    #include <memory>    using namespace std;    class Integer  {      int n;      public:        Integer(int n) : n(n) { }        ~Integer() { printf("Deleting %d\n", n); }        int get() const { return n; }  };    int main()  {      shared\_ptr<Integer> a(new Integer{ 10 });      shared\_ptr<Integer> b(new Integer{ 20 });      shared\_ptr<Integer> c = a;      shared\_ptr<Integer> d(new Integer{ 30 });      shared\_ptr<Integer> e = b;      a = d;      b = shared\_ptr<Integer>(new Integer(40));      shared\_ptr<Integer> f = c;      b = f;        printf("%d\n", a->get());      printf("%d\n", b->get());      printf("%d\n", c->get());      printf("%d\n", d->get());      printf("%d\n", e->get());      printf("%d\n", f->get());  } |

The output is similar to this one:

Deleting 40  
30  
10  
10  
30  
20  
10  
Deleting 20  
Deleting 10  
Deleting 30

Result: No crashes and all the objects are released when they are not being used anymore.

In order to hide the operator new and to provide an optimization while allocating the object to be shared, the variadic template function **make\_shared**was created. It is a template function that performs three tasks:

1. Allocates contiguous memory for the object and for the reference counter. This makes the creation and destruction of objects faster because only one allocation and deallocation will be needed when creating the object to be shared and its reference counter.
2. Invokes to the constructor of the class being instantiated forwarding the arguments used when this function was invoked.
3. Returns a shared\_ptr to the newly created object.

make\_shared<T> is a variadic template function that receives as arguments, the arguments that the constructor of class T needs.

So, our main function will look like this:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | int main()  {      auto a = make\_shared<Integer>(10);      auto b = make\_shared<Integer>(20);      auto c = a;      auto d = make\_shared<Integer>(30);      auto e = b;      a = d;      b = make\_shared<Integer>(40);      auto f = c;      b = f;        printf("%d\n", a->get());      printf("%d\n", b->get());      printf("%d\n", c->get());      printf("%d\n", d->get());      printf("%d\n", e->get());      printf("%d\n", f->get());  } |

What other uses can we think for shared\_ptr?

Look at this C++03 code (I am reusing the Integer class):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | #include <cstdio>  #include <cstring>    #include <vector>    using namespace std;    class Integer  {      int n;      public:        Integer(int n) : n(n) { }        ~Integer() { printf("Deleting %d\n", n); }        int get() const { return n; }  };    Integer\* get\_instance(int n)  {      return new Integer(n);  }    int main()  {      vector<Integer\*> vec;        for (int i = 0; i < 100; i++)          vec.push\_back(get\_instance(i));        int sum = 0;      for (vector<Integer\*>::const\_iterator it = vec.begin(); it != vec.end(); ++it)          sum += (\*it)->get();        //We do something with the elements of the vector      printf("Sum: %d\n", sum);        //We need to release them manually      for (vector<Integer\*>::iterator it = vec.begin(); it != vec.end(); ++it)          delete \*it;  } |

And compare it against this C++11 code:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | #include <cstdio>  #include <cstring>    #include <vector>  #include <memory>    using namespace std;    class Integer  {      int n;      public:        Integer(int n) : n(n) { }        ~Integer() { printf("Deleting %d\n", n); }        int get() const { return n; }  };    shared\_ptr<Integer> get\_instance(int n)  {      return make\_shared<Integer>(n);  }    int main()  {      vector<shared\_ptr<Integer>> vec;        for (int i = 0; i < 100; i++)          vec.push\_back(get\_instance(i));        //We do something with the elements of the vector      int sum = 0;      for (auto& i : vec)          sum += i->get();        printf("Sum: %d\n", sum);  } |

What do you see? These advantages of the C++11 version:

* You can return shared\_ptr instances from functions. This is far better than returning a naked pointer, because when returning a naked pointer, the programmer that invokes the function does not know a priori if he will need to use free, delete, a custom deallocator or nothing after using the object pointed by the returned pointer. When returning a shared\_ptr instance, the programmer knows that the object will be released automatically when it will not be referred anymore.
* You can store shared\_ptr instances inside a STL container. This makes the memory management for your application easier, because you do not need to deallocate the objects manually anymore.
* Because of the automatic memory deallocation, your program is exception safe. In the C++03 version, any problem occurring before the block in charge to release the pointers would leave a lot of objects leaking in the heap.
* Though the C++11 is still using pointers, because of make\_shared usage, the operator new is not used anymore.
* Not related to shared\_ptr; but [auto](https://oopscenities.net/2011/05/03/c0x-auto/) and the [range-based for loop](https://oopscenities.net/2011/06/09/c0x-range-based-for-loop/) make your program shorter and easier to write and understand :)

I based heavily my example in an example given by Herb Sutter in his talk: [“(Not your father’s) C++”](http://channel9.msdn.com/Events/Lang-NEXT/Lang-NEXT-2012/-Not-Your-Father-s-C-).

Shortcomings of shared\_ptr? It does not work for circular references. If you want to implement something where circular references exist, you need to use std::weak\_ptr. I will write about it in a new entry.

# weak\_ptr

In modern C++ applications (C++11 and later), you can replace almost all your naked pointers to shared\_ptr and unique\_ptr in order to have automatic resource administration in a deterministic way so you will not need (almost, again) to release the memory manually.

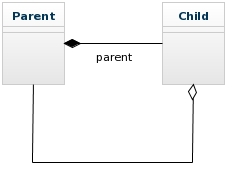
The “almost” means that there is one scenario where the smart pointers, specifically, the shared\_ptr instances, will not work: When you have circular references. In this scenario, since every shared\_ptr is pointing to the other one, the memory will never be released.

Let’s look this scenario in code:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | struct Child;    struct Parent  {      shared\_ptr<Child> child;        ~Parent() { cout << "Bye Parent" << endl; }        void hi() const { cout << "Hello" << endl; }  };    struct Child  {      shared\_ptr<Parent> parent;        ~Child() { cout << "Bye Child" << endl; }  };    int main()  {      auto parent = make\_shared<Parent>();      auto child = make\_shared<Child>();      parent->child = child;      child->parent = parent;      child->parent->hi();  } |

In this program, the Parent and the Child have pointers to their respective child and  
parent (creating a circular reference between both); when you execute the program, you will notice the destructors are never invoked because I used shared\_ptr instances.

If we model this problem, probably we can establish the Parent as “owner” of the Child object (it would mean that the child lifetime will be based on the parent lifetime) and the Child will just refer to its parent. If we diagram this using a class diagram, we can draw it like this:

[](https://oopscenities.files.wordpress.com/2014/08/weak_ptr.jpg)

If we use C++11 smart pointers to implement this kind of scenarios, we could use shared\_ptr instances to implement the composition relationship and weak\_ptr instances to implement the aggregation relationship:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | struct Child;    struct Parent  {      shared\_ptr<Child> child;        ~Parent() { cout << "Bye Parent" << endl; }        void hi() const { cout << "Hello" << endl; }  };    struct Child  {      weak\_ptr<Parent> parent;        ~Child() { cout << "Bye Child" << endl; }  };    int main()  {      auto parent = make\_shared<Parent>();      auto child = make\_shared<Child>();      parent->child = child;      child->parent = parent;      child->parent.lock()->hi();  } |

When you execute this program, you will notice the destructors are invoked correctly.

weak\_ptr is a wrapper for a shared\_ptr that does not represent ownership of the pointee object and, therefore, does not avoid the object to be released when the parent shared\_ptr reference counter goes to 0.

In my code I am using a weak\_ptr method called lock. weak\_ptr does not implement the operator->, so, if you want to access the methods of the pointee object, you need to invoke the lock(9 method. It creates a temporary shared\_ptr instance that, since it increases the pointee reference counter, actually locks the object to be released while it is executing the invoked method.

Other nice feature of weak\_ptr is that it can tell you if its pointee object has been already released. Look at this code:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | struct A  {      int x;      A(int x)  : x(x) { cout << "HI" << endl; }      ~A() { cout << "Bye" << endl; }  };    weak\_ptr<A> m()  {      auto a = make\_shared<A>(12);      cout << a->x << endl;        return a;  }    int main()  {      auto a = m();        cout << "After m()" << endl;        if (a.expired())          cout << "Expired" << endl;      else          cout << a.lock()->x << endl;  } |

The function main() receives an instance of a weak\_ptr. If you look into the code, the function m() has a shared\_ptr instance, but the object pointed by it lives only inside the function and it is released when returning the weak\_ptr. Back to the main() function, before accessing the value of x, I am invoking the method expired() that returns true if the weak\_ptr instance points to an already released method.

Thus, with unique\_ptr, shared\_ptr, and weak\_ptr, all the scenarios where you need to use dynamically allocated memory are covered and you will not need to release your objects manually anymore.