Smart Pointers in C++11

Karl Stratos

1 Background

1.1 Pointers

When we write `T x` to create an instance `x` of type `T`, we allocate the amount of memory required for `T` and names it as `x`. We can "reference" (i.e., take the address of) `x` by writing `&x`. The type of this address is denoted by `T *` and read as the **pointer** to an instance of type `T`. We can "dereference" the pointer by writing `*p`.

```
string x("foo")    // x is an instance of string with value "foo".
string *p = &x;    // p is a pointer to x.
cout << p << endl; // 0x7ffee0c6f980
cout << *p << endl; // foo
```

1.2 Persistent Memory Allocation

When we write `T x`, we create a temporary instance ("stack-based creation") and `x` dies outside of its local scope.

```
string *allocate_foo_naive() {
  string x("foo");
  return &x;
}
string *p = allocate_foo_naive(); // p is a dangling pointer.
cout << *p << endl; // 's??Qi???Pi???Pi???Pi???Pi???Pi???Pi???Pi????i???
```

In C++, we can make the lifetime of an instance persist beyond its local scope by using the `new` operator ("heap-based creation"). Writing `new T()` allocates a persistent memory block for type `T` and returns a pointer to this block.

```
string *allocate_foo() {
  string *p = new string("foo");
  return p;
}
string *p = allocate_foo();
cout << *p << endl; // foo

delete p; // Now p is a dangling pointer.
```

Without the last line, the code above would have a **memory leak**. This is the curse of heap-based instance creation: the memory block allocated by `new` persists so well that it is never freed until an explicit `delete` is performed on a pointer to the block. But even if we fastidiously delete all heap-based instances, there can be other memory-related problems including unintenteded creation of dangling pointers.

Below, we create two pointers `p`, `q` that point to the same memory block at address `0x7fb868c02740` (which contains an instance of string with value "foo"). Suppose we free the block by calling `delete p` at some point. Later, we forget that we have already freed the block and call `delete q`. The result is the infamous runtime error `pointer being freed was not allocated`.

```
string *p = allocate_foo();
string *q = p;
cout << p << endl; // 0x7fb868c02740
cout << q << endl; // 0x7fb868c02740
cout << *p << endl; // foo
cout << *q << endl; // foo

delete p; // Now both p and q are dangling pointers.
delete q; // error for object 0x7f921fc02740: pointer being freed was not
    allocated
```

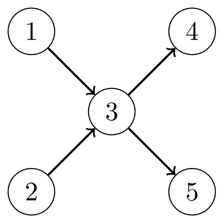
1.3 Illustrative Example

Consider an object type `Node` that represents a vertex in a directed acyclic graph (DAG).

```
struct Node {
  void AddParent(Node *parent) {
    parents.push_back(parent);
    parent->children.push_back(this);
  }
  void AddChild(Node *child) {
    children.push_back(child);
    child->parents.push_back(this);
  }
  vector<Node *> parents;
  vector<Node *> children;
};
```

It is clear that the lifetime of a single node must be persistent (otherwise we have to do all our work in a single local scope), as in the following construction.

```
void create_graph(vector<Node *> *nodes) {
  Node *X1 = new Node();
  Node *X2 = new Node();
  Node *X3 = new Node();
  Node *X4 = new Node();
  Node *X5 = new Node();
  X1->AddChild(X3);
  X2->AddChild(X3);
  X3->AddChild(X4);
  X3->AddChild(X5);
  *nodes = {X1, X2, X3, X4, X5};
}
```



To avoid memory leaks, we must delete every single node that we created using `new` before we exit the program. It is easy in this case because we explicitly maintain a list of all the nodes, so we can do

```
vector<Node *> nodes;
create_graph(&nodes);
// Do stuff with nodes.
for (auto node : nodes) { delete node; }
```

But maintaining such a list is tedious and error-prone. We may alternatively free memory by traversing nodes, but we have to be careful not to delete a dangling pointer. In general, there is always a danger of not freeing some memory blocks and causing memory leaks.

2 Smart Pointers

Smart pointers, introduced in C++11, obviate the need to manually free heap-based instances by using the concept of **ownership**. If an instance `x` is created through a smart pointer construction, then `x` is *owned* by the smart pointer (except weak pointer, see below). When all the owners of `x` expire, then `x` is automatically deallocated.

2.1 Shared Pointers

A **shared pointer** owns an instance possibly along with other shared pointers. We create a shared pointer `p` that points to an instance `x` of type `T` by writing `shared_ptr<T> p = make_shared<T>()`. This causes two allocations:

1. Memory is allocated for `x`, just as in using `new`.
2. Memory is allocated for a **control block** which maintains the number of shared pointers that own `x`.

Each time we create another shared pointer that points to `x`, the number of shared pointers goes up by one.

```
shared_ptr<string> p = make_shared<string>("foo");
cout << *p << endl;            // foo
cout << p.use_count() << endl; // 1

vector<shared_ptr<string>> v = {p, p, p};
cout << p.use_count() << endl; // 4

shared_ptr<string> q = v[0];
cout << p.use_count() << endl; // 5
cout << q.use_count() << endl; // 5

const shared_ptr<string> &r = v[1]; // Reference to avoid incrementing
cout << p.use_count() << endl;     // 5
cout << q.use_count() << endl;     // 5
cout << r.use_count() << endl;     // 5

for (shared_ptr<string> &z : v) { z.reset(); }
cout << p.use_count() << endl;     // 2
cout << q.use_count() << endl;     // 2
cout << r.use_count() << endl;     // 0
```

2.1.1 Passing around shared pointers

As function arguments. When a shared pointer `p` is passed to a function by value, then a *copy* is created and thus the count goes up by one. The copy is destroyed once we exit the function, upon which the count goes down by one.

```
void value(shared_ptr<string> q) { cout << q.use_count() << endl; }
void ref(shared_ptr<string> &q) { cout << q.use_count() << endl; }
void cref(const shared_ptr<string> &q) { cout << q.use_count() << endl; }
void reset(shared_ptr<string> &q) { q.reset(); }

shared_ptr<string> p = make_shared<string>("foo");
cout << p.use_count() << endl; // 1
value(p); // 2
ref(p); // 1
cref(p); // 1
reset(p);
cout << p.use_count() << endl; // 0
```

Thus we almost always pass a shared pointer by reference to avoid creating unnecessary copies and incrementing/decrementing the count. Moreover, unless we wish to edit the passed pointer itself (e.g., the `reset` function above), we pass it by constant reference. Passing a shared pointer by constant reference does *not* mean that the count will not change since we can always create a copy inside the function.

```
vector<shared_ptr<string>> v;
void push(const shared_ptr<string> &p) { v.push_back(p); }

shared_ptr<string> p = make_shared<string>("foo");
push(p);
cout << p.use_count() << endl; // 2
```

As return values. In contrast, we typically return a shared pointer by value. If we return it by reference, then we risk deleting the instance at a wrong time. A simple example is a function creating a shared pointer.

```
shared_ptr<string> &create_foo_wrong() {
  shared_ptr<string> p = make_shared<string>("foo");
  return p;
}
shared_ptr<string> p = create_foo_wrong();
cout << *p << endl; // error: pointer being freed was not allocated
```

2.2 Weak Pointers

A **weak pointer** is a non-owning reference to an instance owned by a shared pointer. Accordingly, it is always created from a shared pointer. It cannot be dereferenced without first creating a temporary shared pointer via `lock`. It can also check if the instance has expired.

```
shared_ptr<string> p = make_shared<string>("foo");
vector<weak_ptr<string>> v = {p, p, p};
cout << *v[2] << endl;          // error
cout << *v[2].lock() << endl;   // foo
cout << v[2].use_count() << endl; // 1
cout << v[2].expired() << endl; // false
p.reset();
cout << v[2].expired() << endl; // true
```

2.3 Previous Example Revisited

Using a combination of shared and weak pointers, we can implement the class `Node` before representing a vertex in a DAG in such a way that it does not require any manual memory deallocation.

```
struct Node : enable_shared_from_this<Node> {
  void AddParent(const shared_ptr<Node> &parent) {
    parents.push_back(parent);
    parent->children.push_back(shared_from_this());
  }
  void AddChild(const shared_ptr<Node> &child) {
    children.push_back(child);
    child->parents.push_back(shared_from_this());
  }
  vector<weak_ptr<Node>> parents; // Child doesn't own parents.
  vector<shared_ptr<Node>> children; // Parent owns children.
};
```

There are several parts that deserve noting:

- The public inheritance from `enable_shared_from_this<Node>` equips `Node` with a member function `shared_from_this`.
- If `x` is a particular instance of `Node` which is owned by a shared pointer `p`, then calling `shared_from_this()` from `x` creates a new shared pointer `q` that shares the ownership of `x` with `p`.

  ```
  shared_ptr<Node> p = make_shared<Node>();
  shared_ptr<Node> q = p->shared_from_this();
  cout << q.use_count() << endl; // 2
  ```

- Parents are weak pointers. If they were shared pointers, we would get memory leaks from a graph as simple as `p` $\to$ `q` because neither node expires: `p` waits until `q` relinquishes its ownership and vice versa.

Now we can create the same DAG example before without worrying about memory leaks.

```
shared_ptr<Node> X1 = make_shared<Node>();
shared_ptr<Node> X2 = make_shared<Node>();
{  // These local instances will persist because they are owned.
  shared_ptr<Node> X3 = make_shared<Node>();
  X1->AddChild(X3);
  X2->AddChild(X3);
  shared_ptr<Node> X4 = make_shared<Node>();
  X4->AddParent(X3);
  shared_ptr<Node> X5 = make_shared<Node>();
```

```
  X5->AddParent(X3);
}
const shared_ptr<Node> &X3 = X1->children[0];
const shared_ptr<Node> &X4 = X2->children[0]->children[0];
const shared_ptr<Node> &X5 = X2->children[0]->children[1];
cout << X3.use_count() << endl; // 2
cout << X4.use_count() << endl; // 1
cout << X5.use_count() << endl; // 1
```

2.4 Unique Pointers

A **unique pointer** is a stripped down version of a shared pointer in which the ownership is limited to one pointer at any given time.

```
unique_ptr<string> p(new string("foo"));
cout << *p << endl; // foo
unique_ptr<string> q = p; // error: can only have one owner
```

Although there can be no more than one owner, it can be moved:

```
unique_ptr<string> q = move(p); // okay
cout << *p << endl; // Segmentation fault: 11
cout << *q << endl; // foo
```

If we indeed have a situation satisfying the single ownership constraint (e.g., a binary tree in which each node has at most one owning parent), then a unique pointer is preferred over a shared pointer for efficiency reasons. Otherwise, we must use a shared pointer.