WIKIPEDIA

Smart pointer

In computer science, a **smart pointer** is an abstract data type that simulates a pointer while providing added features, such as automatic memory management or bounds checking. Such features are intended to reduce bugs caused by the misuse of pointers, while retaining efficiency. Smart pointers typically keep track of the memory they point to, and may also be used to manage other resources, such as network connections and file handles. Smart pointers originated in the programming language C++.

Pointer misuse can be a major source of bugs. Smart pointers prevent most situations of memory leaks by making the memory deallocation automatic. More generally, they make object destruction automatic: an object controlled by a smart pointer is automatically destroyed (finalized and then deallocated) when the last (or only) owner of an object is destroyed, for example because the owner is a local variable, and execution leaves the variable's scope. Smart pointers also eliminate dangling pointers by postponing destruction until an object is no longer in use.

Several types of smart pointers exist. Some work with reference counting, others by assigning ownership of an object to one pointer. If a language supports automatic garbage collection (for example, Java or C#), then smart pointers are unneeded for the reclaiming and safety aspects of memory management, yet are useful for other purposes, such as cache data structure residence management and resource management of objects such as file handles or network sockets.

Contents

Features

In C++, a smart pointer is implemented as a template class that mimics, by means of operator overloading, the behaviors of a traditional (raw) pointer, (e.g. dereferencing, assignment) while providing additional memory management features.

Smart pointers can facilitate intentional programming by expressing, in the type, how the memory of the referent of the pointer will be managed. For example, if a C++ function returns a pointer, there is no way to know whether the caller should delete the memory of the referent when the caller is finished with the information.

```
some_type* ambiguous_function(); // What should be done with the result?
```

Traditionally, naming conventions have been used to resolve the ambiguity,[1] which is an error-prone, labor-intensive approach. C++11 introduced a way to ensure correct memory management in this case by declaring the function to return a `unique_ptr`,

```
unique_ptr<some_type> obvious_function1();
```

The declaration of the function return type as a unique_ptr makes explicit the fact that the caller takes ownership of the result, and the C++ runtime ensures that the memory for *some_type will be reclaimed automatically. Before C++11, unique_ptr can be replaced with auto_ptr.

Creating new objects

To ease the allocation of a

```
std::shared_ptr<some_type>
```

C++11 introduced:

```
 auto s = std::make_shared<some_type>(constructor, parameters, here);
```

and similarly

```
std::unique_ptr<some_type>
```

Since C++14 one can use:

```
 auto u = std::make_unique<some_type>(constructor, parameters, here);
```

It is preferred, in almost all circumstances, to use these facilities over the `new` keyword:[2]

unique_ptr

C++11 introduces `std::unique_ptr`, defined in the header `<memory>`.[3]

A `unique_ptr` is a container for a raw pointer, which the `unique_ptr` is said to own. A `unique_ptr` explicitly prevents copying of its contained pointer (as would happen with normal assignment), but the `std::move` function can be used to transfer ownership of the contained pointer to another `unique_ptr`. A `unique_ptr` cannot be copied because its copy constructor and assignment operators are explicitly deleted.

```
std::unique_ptr<int> p1(new int(5));
std::unique_ptr<int> p2 = p1; //Compile error.
std::unique_ptr<int> p3 = std::move(p1); //Transfers ownership. p3 now owns the memory and p1 is set to
nullptr.

p3.reset(); //Deletes the memory.
p1.reset(); //Does nothing.
```

`std::auto_ptr` is deprecated under C++11 and completely removed from C++17. The copy constructor and assignment operators of `auto_ptr` do not actually copy the stored pointer. Instead, they transfer it, leaving the prior `auto_ptr` object empty. This was one way to implement strict ownership, so that only one `auto_ptr` object can own the pointer at any given time. This means that `auto_ptr` should not be used where copy semantics are needed.[4] Since `auto_ptr` already existed with its copy semantics, it could not be upgraded to be a move-only pointer without breaking backward compatibility with existing code.

shared_ptr and weak_ptr

C++11 introduces `std::shared_ptr` and `std::weak_ptr`, defined in the header `<memory>`.[3]

A `shared_ptr` is a container for a raw pointer. It maintains reference counting ownership of its contained pointer in cooperation with all copies of the `shared_ptr`. An object referenced by the contained raw pointer will be destroyed when and only when all copies of the `shared_ptr` have been destroyed.

```
std::shared_ptr<int> p0(new int(5));        // valid, allocates 1 integer and initialize it with value 5
std::shared_ptr<int[]> p1(new int[5]);      // valid, allocates 5 integers
std::shared_ptr<int[]> p2 = p1; //Both now own the memory.

p1.reset(); //Memory still exists, due to p2.
p2.reset(); //Deletes the memory, since no one else owns the memory.
```

A `weak_ptr` is a container for a raw pointer. It is created as a copy of a `shared_ptr`. The existence or destruction of `weak_ptr` copies of a `shared_ptr` have no effect on the `shared_ptr` or its other copies. After all copies of a `shared_ptr` have been destroyed, all `weak_ptr` copies become empty.

```
std::shared_ptr<int> p1(new int(5));
std::weak_ptr<int> wp1 = p1; //p1 owns the memory.

{
  std::shared_ptr<int> p2 = wp1.lock(); //Now p1 and p2 own the memory.
  if(p2) // As p2 is initialized from a weak pointer, you have to check if the memory still exists!
  {
    //Do something with p2
  }
} //p2 is destroyed. Memory is owned by p1.

p1.reset(); //Memory is deleted.

std::shared_ptr<int> p3 = wp1.lock(); //Memory is gone, so we get an empty shared_ptr.
if(p3)
{
  //Will not execute this.
}
```

Because the implementation of `shared_ptr` uses reference counting, circular references are potentially a problem. A circular `shared_ptr` chain can be broken by changing the code so that one of the references is a `weak_ptr`.

Multiple threads can safely simultaneously access different `shared_ptr` and `weak_ptr` objects that point to the same object.[5]

The referenced object must be protected separately to ensure thread safety.

`shared_ptr` and `weak_ptr` are based on versions used by the Boost libraries. C++ Technical Report 1 (TR1) first introduced them to the standard, as general utilities, but C++11 adds more functions, in line with the Boost version.

See also

- Resource acquisition is initialization (RAII)
- auto_ptr
- Opaque pointer
- Reference (computer science)
- Boost (C++ libraries)

References

1. "Taligent's Guide to Designing Programs, section Use special names for copy, create, and adopt routines" (https://root.cern.ch/TaligentDocs/TaligentOnline/DocumentRoot/1.0/Docs/books/WM/WM_67.html#HEADING81).
2. Sutter, Herb (20 April 2013). "Trip Report: ISO C++ Spring 2013 Meeting" (http://isocpp.org/blog/2013/04/trip-report-iso-c-spring-2013-meeting). *isocpp.org*. Retrieved 14 June 2013.
3. ISO 14882:2011 20.7.1
4. CERT C++ Secure Coding Standard
5. boost::shared_ptr thread safety (http://www.boost.org/libs/smart_ptr/shared_ptr.htm#ThreadSafety) (does not formally cover std::shared_ptr, but is believed to have the same threading limitations)

External links

- Smart Pointers (http://www.informit.com/articles/article.aspx?p=25264). *Modern C++ Design: Generic Programming and Design Patterns Applied* by Andrei Alexandrescu, Addison-Wesley, 2001.
- countptr.hpp (http://www.josuttis.com/libbook/cont/countptr.hpp.html). *The C++ Standard Library - A Tutorial and Reference (http://www.josuttis.com/libbook/)* by Nicolai M. Josuttis
- Boost Smart Pointers (http://boost.org/libs/smart_ptr/smart_ptr.htm)
- The New C++: Smart(er) Pointers (http://www.drdobbs.com/184403837/). Herb Sutter August 1, 2002
- Smart Pointers - What, Why, Which? (http://ootips.org/yonat/4dev/smart-pointers.html). Yonat Sharon
- Smart Pointers Overview (http://dlugosz.com/Repertoire/refman/Classics/Smart%20Pointers%20Overview.html). John M. Dlugosz
- Smart Pointers in Delphi (http://barrkel.blogspot.com/2008/09/smart-pointers-in-delphi.html)
- Smart Pointers in Rust (http://static.rust-lang.org/doc/0.8/tutorial.html#boxes)
- Smart Pointers in Modern C++ (http://quant-coder.blogspot.com/2015/12/built-in-smart-pointers-in-modern-c.html)