

What is a smart pointer and when should I use one?

Asked 16 years, 3 months ago Modified 3 months ago

Viewed 752k times



What is a smart pointer and when should I use one?

2183

c++

pointers

c++11

smart-pointers

c++-faq



Share

Improve this question

Follow

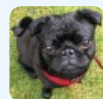
edited Apr 27, 2020 at 7:03



[user2357112](#)

278k ● 31 ● 472 ● 552

asked Sep 20, 2008 at 0:09



[Alex Reynolds](#)

96.8k ● 59 ● 250 ● 351

2 Note that the implementation of `std::auto_ptr` in Visual Studio 2005 is horribly broken.

<http://connect.microsoft.com/VisualStudio/feedback/ViewFeedback.aspx?FeedbackID=98871>


<http://connect.microsoft.com/VisualStudio/feedback/ViewFeedback.aspx?FeedbackID=101842> Use the boost ones

instead. – [Richard](#) Sep 21, 2008 at 10:14

34 Two excellent articles on the subject: - [Smart Pointers - What, Why, Which?](#) - [Guru of the Week #25](#) – [Lazer](#) Jun 25, 2010 at 19:15

- 1 Here's Alexandrescu's (free) chapter on the nitty gritty of creating smart pointers of different flavors: informit.com/articles/article.aspx?p=31529 In his implementation, he uses template arguments as "policies" to specify which attributes he wants (e.g., reference counting), whereas the standard library uses separate classes. Note that he was also writing before rvalue references were available to make something like `std::unique_ptr` possible.
– [metal](#) Mar 19, 2013 at 16:30

I would like to add one more point to the above question, smart pointer `std::shared_ptr` doesn't have subscript operator and doesn't support pointer arithmetic, we can use `get()` to obtain a built in pointer. – [suresh m](#) Feb 7, 2018 at 13:57

[It was added in C++17](#), but is not guaranteed to work for anything but plain arrays. – [HolyBlackCat](#) Feb 7, 2018 at 14:00 

14 Answers

Sorted by:

Highest score (default)



UPDATE

2057



This answer is rather old, and so describes what was 'good' at the time, which was smart pointers provided by the Boost library. Since C++11, the standard library has provided sufficient smart pointers types, and so you should favour the use of `std::unique_ptr`, `std::shared_ptr` and `std::weak_ptr`.



There was also `std::auto_ptr`. It was very much like a scoped pointer, except that it also had the "special" dangerous ability to be copied — which also

unexpectedly transfers ownership.

It was deprecated in C++11 and removed in C++17, so you shouldn't use it.

```
std::auto_ptr<MyObject> p1 (new MyObject());
std::auto_ptr<MyObject> p2 = p1; // Copy and transfer
                                // p1 gets set to emp
p2->DoSomething(); // Works.
p1->DoSomething(); // Oh oh. Hopefully raises some NUL
```

OLD ANSWER

A smart pointer is a class that wraps a 'raw' (or 'bare') C++ pointer, to manage the lifetime of the object being pointed to. There is no single smart pointer type, but all of them try to abstract a raw pointer in a practical way.

Smart pointers should be preferred over raw pointers. If you feel you need to use pointers (first consider if you *really* do), you would normally want to use a smart pointer as this can alleviate many of the problems with raw pointers, mainly forgetting to delete the object and leaking memory.

With raw pointers, the programmer has to explicitly destroy the object when it is no longer useful.

```
// Need to create the object to achieve some goal
MyObject* ptr = new MyObject();
ptr->DoSomething(); // Use the object in some way
delete ptr; // Destroy the object. Done with it.
// Wait, what if DoSomething() raises an exception...?
```

A smart pointer by comparison defines a policy as to when the object is destroyed. You still have to create the object, but you no longer have to worry about destroying it.

```
SomeSmartPtr<MyObject> ptr(new MyObject());  
ptr->DoSomething(); // Use the object in some way.  
  
// Destruction of the object happens, depending  
// on the policy the smart pointer class uses.  
  
// Destruction would happen even if DoSomething()  
// raises an exception
```

The simplest policy in use involves the scope of the smart pointer wrapper object, such as implemented by

`boost::scoped_ptr` or `std::unique_ptr`.

```
void f()  
{  
    {  
        std::unique_ptr<MyObject> ptr(new MyObject());  
        ptr->DoSomethingUseful();  
    } // ptr goes out of scope --  
        // the MyObject is automatically destroyed.  
  
    // ptr->Oops(); // Compile error: "ptr" not defined  
                    // since it is no longer in scope.  
}
```

Note that `std::unique_ptr` instances cannot be copied. This prevents the pointer from being deleted multiple times (incorrectly). You can, however, pass references to it around to other functions you call.

`std::unique_ptr`s are useful when you want to tie the lifetime of the object to a particular block of code, or if you embedded it as member data inside another object, the lifetime of that other object. The object exists until the containing block of code is exited, or until the containing object is itself destroyed.

A more complex smart pointer policy involves reference counting the pointer. This does allow the pointer to be copied. When the last "reference" to the object is destroyed, the object is deleted. This policy is implemented by `boost::shared_ptr` and `std::shared_ptr`.

```
void f()
{
    typedef std::shared_ptr<MyObject> MyObjectPtr; //
    MyObjectPtr p1; // Empty

    {
        MyObjectPtr p2(new MyObject());
        // There is now one "reference" to the created
        p1 = p2; // Copy the pointer.
        // There are now two references to the object.
    } // p2 is destroyed, leaving one reference to the
} // p1 is destroyed, leaving a reference count of zero
// The object is deleted.
```

Reference counted pointers are very useful when the lifetime of your object is much more complicated, and is not tied directly to a particular section of code or to another object.

There is one drawback to reference counted pointers — the possibility of creating a dangling reference:

```
// Create the smart pointer on the heap
MyObjectPtr* pp = new MyObjectPtr(new MyObject())
// Hmm, we forgot to destroy the smart pointer,
// because of that, the object is never destroyed!
```

Another possibility is creating circular references:

```
struct Owner {
    std::shared_ptr<Owner> other;
};

std::shared_ptr<Owner> p1 (new Owner());
std::shared_ptr<Owner> p2 (new Owner());
p1->other = p2; // p1 references p2
p2->other = p1; // p2 references p1

// Oops, the reference count of p1 and p2 never goes to 0
// The objects are never destroyed!
```

To work around this problem, both Boost and C++11 have defined a `weak_ptr` to define a weak (uncounted) reference to a `shared_ptr`.

Share Improve this answer

Follow

edited Aug 25, 2019 at 0:12



Marc.2377

8,634 ● 7 ● 54 ● 99

answered Sep 20, 2008 at 0:48



Lloyd

22.9k ● 2 ● 20 ● 7

7 Do you mean `std::auto_ptr<MyObject> p1 (new MyObject());` instead of `std::auto_ptr<MyObject> p1 (new Owner());` ? – [Mateen Ulhaq](#) Jul 16, 2011 at 23:06

42 Awesome answer. It would be nice if it were updated for c++11. I found this answer looking for info about the new 11 standard and it would be nice if future visitors could find the updated info. I know `auto_ptr` has been deprecated. I believe `shared_ptr` and `weak_ptr` exist as described, and I think the `scoped_ptr` is now `unique_ptr` in the standard. If this is true, can this answer be updated please? – [SaulBack](#) Sep 11, 2012 at 20:50

24 To say that the possibility of creating a dangling reference is a drawback to reference counted pointers is absolutely insane. Possible dangling references are a drawback of **any C++ pointer**. In fact, it is *exactly that drawback* which smart pointers are intended to *alleviate*. – [Michael Francis](#) Aug 13, 2014 at 21:59 ✎

27 If you declare a pointer to a smart pointer (as was done in the example) you knowingly give up all benefits of the smart pointer. This is not a drawback or a design flaw, it is the most idiotic usage imaginable. – [Michael Francis](#) Aug 13, 2014 at 22:02 ✎

3 A `const std::auto_ptr` is safe to use, if you're stuck with C++03. I used it for pimpl pattern quite a lot until I got access to C++11. – [Toby Speight](#) Sep 17, 2015 at 11:55



Here's a simple answer for these days of modern C++ (C++11 and later):

430

- "What is a smart pointer?"

It's a type whose values can be used like pointers,



but which provides the additional feature of automatic memory management: When a smart pointer is no longer in use, the memory it points to is deallocated (see also [the more detailed definition on Wikipedia](#)).

- **"When should I use one?"**

Well, you are often better off avoiding the user of pointers altogether, smart or otherwise. Having said that - smart pointers may be useful in code which involves tracking the ownership of a piece of memory, allocating or de-allocating; the smart pointer often saves you the need to do these things explicitly.

Note that smart pointers don't keep track of the *size* of the allocated area (even if the de-allocation can figure that out); if you need that size accessible, you may be better off with some [contiguous \(owning\) container](#) (from the standard library or otherwise).

- **"But which smart pointer should I use in which of those cases?"**

- Use `std::unique_ptr` when you want your object to live just as long as a single owning reference to it lives. For example, use it for a pointer to memory which gets allocated on entering some scope and de-allocated on exiting the scope.
- Use `std::shared_ptr` when you do want to refer to your object from multiple places - and do not want your object to be de-allocated until all these references are themselves gone.

- Use `std::weak_ptr` when you do want to refer to your object from multiple places - for those references for which it's ok to ignore and deallocate (so they'll just note the object is gone when you try to dereference).
- There is a [proposal](#) to add [hazard pointers](#) to C++26, but for now you don't have them.
- Don't use the `boost::` smart pointers or `std::auto_ptr` except in special cases which you can read up on if you must.
- **"Hey, I didn't ask which one to use!"**
Ah, but you really wanted to, admit it.
- **"So when should I use regular pointers then?"**
Mostly in code that is oblivious to memory ownership. This would typically be in functions which get a pointer from someplace else and do not allocate nor de-allocate, and do not store a copy of the pointer which outlasts their execution.

Share Improve this answer

edited Sep 12 at 14:31

Follow

answered May 9, 2015 at 19:06



[einpoklum](#)

131k ● 76 ● 408 ● 832

14 It is worth noting that while smart (owning) pointers help with proper memory management, raw (non-owning) pointers are still useful for other organizational purposes in data structures. Herb Sutter gave a great presentation upon this

matter at CppCon 2016, that you can see on YouTube: [Leak-Freedom in C++... By Default.](#) – [wiktor.wandachowicz](#) Nov 13, 2016 at 23:22

1 @wiktor.wandachowicz `T*` is to `std::unique_ptr<T>` what `std::weak_ptr<T>` is to `std::shared_ptr<T>` – [Caleth](#) Sep 17, 2019 at 12:50

5 @Caleth: No, I wouldn't say that. – [einpoklum](#) Sep 17, 2019 at 14:27

2 @Caleth That is not accurate. `T*` is to `std::unique_ptr<T>` what `T*` is to `std::shared_ptr<T>`. In both cases, if you want a non-owning pointer to the managed object you should use a raw pointer. `weak_ptr` is not well suited for that purpose. – [François Andrieux](#) Mar 22, 2022 at 13:58 ✎



UPDATE:

122

This answer is outdated concerning C++ types which were used in the past.



`std::auto_ptr` is deprecated and removed in new standards.



Instead of `boost::shared_ptr` the [std::shared_ptr](#) should be used which is part of the standard.



The links to the concepts behind the rationale of smart pointers still mostly relevant.

Modern C++ has the following smart pointer types and doesn't require [boost smart pointers](#):

- [std::shared_ptr](#)

- `std::weak_ptr`
- `std::unique_ptr`

There is also 2-nd edition of the book mentioned in the answer: [C++ Templates: The Complete Guide 2nd Edition by David Vandevoorde Nicolai, M. Josuttis, Douglas Gregor](#)

OLD ANSWER:

A [smart pointer](#) is a pointer-like type with some additional functionality, e.g. automatic memory deallocation, reference counting etc.

A small intro is available on the page [Smart Pointers - What, Why, Which?](#).

One of the simple smart-pointer types is `std::auto_ptr` (chapter 20.4.5 of C++ standard), which allows one to deallocate memory automatically when it out of scope and which is more robust than simple pointer usage when exceptions are thrown, although less flexible.

Another convenient type is `boost::shared_ptr` which implements reference counting and automatically deallocates memory when no references to the object remains. This helps avoiding memory leaks and is easy to use to implement [RAII](#).

The subject is covered in depth in book ["C++ Templates: The Complete Guide" by David Vandevoorde, Nicolai M.](#)

[Josuttis](#), chapter Chapter 20. Smart Pointers. Some topics covered:

- Protecting Against Exceptions
- Holders, (note, [std::auto_ptr](#) is implementation of such type of smart pointer)
- [Resource Acquisition Is Initialization](#) (This is frequently used for exception-safe resource management in C++)
- Holder Limitations
- [Reference Counting](#)
- Concurrent Counter Access
- Destruction and Deallocation

Share Improve this answer

edited Sep 8, 2022 at 6:42

Follow

answered Sep 20, 2008 at 0:32



[sergtk](#)

11k ● 15 ● 78 ● 133

-
- 6 Warning `std::auto_ptr` is deprecated and highly discourage as you can accidentally transfer ownership. -- C++11 removes the need of Boost, use: `std::unique_ptr`, `std::shared_ptr` and `std::weak_ptr` – [ninMonkey](#) Jun 12, 2019 at 13:38
-



47



Definitions provided by Chris, Sergdev and Llyod are correct. I prefer a simpler definition though, just to keep my life simple: A smart pointer is simply a class that overloads the `->` and `*` operators. Which means that your object semantically looks like a pointer but you can make it do way cooler things, including reference counting, automatic destruction etc. `shared_ptr` and `auto_ptr` are sufficient in most cases, but come along with their own set of small idiosyncrasies.

Share Improve this answer

Follow

edited Apr 7, 2017 at 14:45



NathanOliver

179k ● 29 ● 314 ● 427

answered Sep 20, 2008 at 1:53



Sridhar Iyer

2,830 ● 1 ● 22 ● 28



34



A smart pointer is like a regular (typed) pointer, like `"char*"`, except when the pointer itself goes out of scope then what it points to is deleted as well. You can use it like you would a regular pointer, by using `"->"`, but not if you need an actual pointer to the data. For that, you can use `"&*ptr"`.

It is useful for:

- Objects that must be allocated with `new`, but that you'd like to have the same lifetime as something on that stack. If the object is assigned to a smart pointer,

then they will be deleted when the program exits that function/block.

- Data members of classes, so that when the object is deleted all the owned data is deleted as well, without any special code in the destructor (you will need to be sure the destructor is virtual, which is almost always a good thing to do).

You may *not* want to use a smart pointer when:

- ... the pointer shouldn't actually own the data... i.e., when you are just using the data, but you want it to survive the function where you are referencing it.
- ... the smart pointer isn't itself going to be destroyed at some point. You don't want it to sit in memory that never gets destroyed (such as in an object that is dynamically allocated but won't be explicitly deleted).
- ... two smart pointers might point to the same data. (There are, however, even smarter pointers that will handle that... that is called [reference counting](#).)

See also:

- [garbage collection](#).
- [This stack overflow question](#) regarding data ownership

Share Improve this answer

edited May 23, 2017 at 12:02

Follow



Community Bot

1 • 1



21

A smart pointer is an object that acts like a pointer, but additionally provides control on construction, destruction, copying, moving and dereferencing.



One can implement one's own smart pointer, but many libraries also provide smart pointer implementations each with different advantages and drawbacks.



For example, [Boost](#) provides the following smart pointer implementations:

- `shared_ptr<T>` is a pointer to `T` using a reference count to determine when the object is no longer needed.
- `scoped_ptr<T>` is a pointer automatically deleted when it goes out of scope. No assignment is possible.
- `intrusive_ptr<T>` is another reference counting pointer. It provides better performance than `shared_ptr`, but requires the type `T` to provide its own reference counting mechanism.
- `weak_ptr<T>` is a weak pointer, working in conjunction with `shared_ptr` to avoid circular references.
- `shared_array<T>` is like `shared_ptr`, but for arrays of `T`.

- `scoped_array<T>` is like `scoped_ptr`, but for arrays of `T`.

These are just one linear descriptions of each and can be used as per need, for further detail and examples one can look at the documentation of Boost.

Additionally, the C++ standard library provides three smart pointers; `std::unique_ptr` for unique ownership, `std::shared_ptr` for shared ownership and `std::weak_ptr`. `std::auto_ptr` existed in C++03 but is now deprecated.

Share Improve this answer

edited Mar 12, 2013 at 10:03

Follow



user142019

answered Mar 12, 2013 at 9:51



Saqlain

17.9k ● 4 ● 30 ● 33

Please explain why `scoped_ptr` is not like a locally-declared `const unique_ptr` - which also gets deleted on exiting the scope. – [einpoklum](#) Dec 29, 2015 at 21:54



20



Most kinds of smart pointers handle disposing of the pointer-to object for you. It's very handy because you don't have to think about disposing of objects manually anymore.



The most commonly-used smart pointers are



`std::tr1::shared_ptr` (or `boost::shared_ptr`), and, less commonly, `std::auto_ptr`. I recommend regular use of `shared_ptr`.

`shared_ptr` is very versatile and deals with a large variety of disposal scenarios, including cases where objects need to be "passed across DLL boundaries" (the common nightmare case if different `libc`s are used between your code and the DLLs).

Share Improve this answer

answered Sep 20, 2008 at 0:14

Follow



C. K. Young

223k ● 47 ● 390 ● 443



Here is the Link for similar answers :

http://sickprogrammersarea.blogspot.in/2014/03/technical-interview-questions-on-c_6.html

13



A smart pointer is an object that acts, looks and feels like a normal pointer but offers more functionality. In C++, smart pointers are implemented as template classes that encapsulate a pointer and override standard pointer operators. They have a number of advantages over regular pointers. They are guaranteed to be initialized as either null pointers or pointers to a heap object.



Indirection through a null pointer is checked. No delete is ever necessary. Objects are automatically freed when the last pointer to them has gone away. One significant problem with these smart pointers is that unlike regular pointers, they don't respect inheritance. Smart pointers

are unattractive for polymorphic code. Given below is an example for the implementation of smart pointers.

Example:

```
template <class X>
class smart_pointer
{
    public:
        smart_pointer();

        smart_pointer(const X& x) //
        X& operator *( );
        const X& operator*( ) const;
        X* operator->( ) const;

        smart_pointer(const smart_pointer <X> &
        const smart_pointer <X> & operator =(co
        ~smart_pointer();
    private:
        //...
};
```

This class implement a smart pointer to an object of type X. The object itself is located on the heap. Here is how to use it:

```
smart_pointer <employee> p= employee("Harris",1333);
```

Like other overloaded operators, p will behave like a regular pointer,

```
cout<<*p;
p->raise_salary(0.5);
```

Share Improve this answer

answered Mar 7, 2014 at 9:03

Follow



Santosh

1,304 ● 2 ● 16 ● 31



Let T be a class in this tutorial Pointers in C++ can be divided into 3 types :

10



1) **Raw pointers** :

```
T a;  
T * _ptr = &a;
```



They hold a memory address to a location in memory. Use with caution , as programs become complex hard to keep track.

Pointers with const data or address { Read backwards }

```
T a ;  
const T * ptr1 = &a ;  
T const * ptr1 = &a ;
```

Pointer to a data type T which is a const. Meaning you cannot change the data type using the pointer. ie `*ptr1 = 19` ; will not work. But you can move the pointer. ie `ptr1++` , `ptr1--` ; etc will work. Read backwards : pointer to type T which is const

```
T * const ptr2 ;
```

A const pointer to a data type T . Meaning you cannot move the pointer but you can change the value pointed to by the pointer. ie `*ptr2 = 19` will work but `ptr2++ ;` `ptr2--` etc will not work. Read backwards : const pointer to a type T

```
const T * const ptr3 ;
```

A const pointer to a const data type T . Meaning you cannot either move the pointer nor can you change the data type pointer to be the pointer. ie . `ptr3-- ; ptr3++ ;` `*ptr3 = 19;` will not work

3) Smart Pointers : { `#include <memory>` }

Shared Pointer:

```
T a ;  
    //shared_ptr<T> shptr(new T) ; not recommended but  
    shared_ptr<T> shptr = make_shared<T>(); // faster  
  
    std::cout << shptr.use_count() ; // 1 // gives the  
things " pointing to it.  
    T * temp = shptr.get(); // gives a pointer to object  
  
    // shared_ptr used like a regular pointer to  
    shptr->memFn();  
    (*shptr).memFn();  
  
    //  
    shptr.reset() ; // frees the object pointed to by  
    shptr = nullptr ; // frees the object  
    shptr = make_shared<T>() ; // creates a new  
object
```

Implemented using reference counting to keep track of how many " things " point to the object pointed to by the pointer. When this count goes to 0 , the object is automatically deleted , ie objected is deleted when all the share_ptr pointing to the object goes out of scope. This gets rid of the headache of having to delete objects which you have allocated using new.

Weak Pointer : Helps deal with cyclic reference which arises when using Shared Pointer If you have two objects pointed to by two shared pointers and there is an internal shared pointer pointing to each others shared pointer then there will be a cyclic reference and the object will not be deleted when shared pointers go out of scope. To solve this , change the internal member from a shared_ptr to weak_ptr. Note : To access the element pointed to by a weak pointer use lock() , this returns a weak_ptr.

```
T a ;
shared_ptr<T> shr = make_shared<T>() ;
weak_ptr<T> wk = shr ; // initialize a weak_ptr from a
wk.lock()->memFn() ; // use lock to get a shared_ptr
//    ^^^ Can lead to exception if the shared ptr has g
if(!wk.expired()) wk.lock()->memFn() ;
// Check if shared ptr has gone out of scope before ac
```

See : [When is std::weak_ptr useful?](#)

Unique Pointer : Light weight smart pointer with exclusive ownership. Use when pointer points to unique objects without sharing the objects between the pointers.

```
unique_ptr<T> uptr(new T);
uptr->memFn();

//T * ptr = uptr.release(); // uptr becomes null and o
ptr
uptr.reset() ; // deletes the object pointed to by uptr
```

To change the object pointed to by the unique ptr , use move semantics

```
unique_ptr<T> uptr1(new T);
unique_ptr<T> uptr2(new T);
uptr2 = std::move(uptr1);
// object pointed by uptr2 is deleted and
// object pointed by uptr1 is pointed to by uptr2
// uptr1 becomes null
```

References : They can essentially be thought of as const pointers, ie a pointer which is const and cannot be moved with better syntax.

See : [What are the differences between a pointer variable and a reference variable in C++?](#)

```
r-value reference : reference to a temporary object
l-value reference : reference to an object whose address
const reference : reference to a data type which is const
modified
```

Reference :

https://www.youtube.com/channel/UCEOGtxYTB6vo6MQ-WQ9W_nQ. Thanks to Andre for pointing out this question.

Share Improve this answer

Follow

edited May 23, 2017 at 11:47



Community Bot

1 • 1

answered Mar 3, 2016 at 0:58



nnrales

1,519 • 2 • 19 • 28



8



http://en.wikipedia.org/wiki/Smart_pointer

In computer science, a smart pointer is an abstract data type that simulates a pointer while providing additional features, such as automatic garbage collection or bounds checking. These additional features are intended to reduce bugs caused by the misuse of pointers while retaining efficiency. Smart pointers typically keep track of the objects that point to them for the purpose of memory management. The misuse of pointers is a major source of bugs: the constant allocation, deallocation and referencing that must be performed by a program written using pointers makes it very likely that some memory leaks will occur. Smart pointers try to prevent memory leaks by making the resource deallocation automatic: when the pointer to an object (or the last in a series of pointers) is destroyed, for example because it goes out of scope, the pointed object is destroyed too.

Share Improve this answer

Follow

answered Sep 20, 2008 at 0:12



Jorge Ferreira

97.8k ● 25 ● 126 ● 134



5



A smart pointer is a class, a wrapper of a normal pointer. Unlike normal pointers, smart pointer's life cycle is based on a reference count (how many times the smart pointer object is assigned). So whenever a smart pointer is assigned to another one, the internal reference count plus plus. And whenever the object goes out of scope, the reference count minus minus.

Automatic pointer, though looks similar, is totally different from smart pointer. It is a convenient class that deallocates the resource whenever an automatic pointer object goes out of variable scope. To some extent, it makes a pointer (to dynamically allocated memory) work similar to a stack variable (statically allocated in compiling time).

Share Improve this answer

Follow

edited Aug 7, 2017 at 7:54



Bhargav Rao

52k ● 29 ● 126 ● 141

answered Jun 12, 2017 at 23:23



Trombe

197 ● 3 ● 8



What is a smart pointer.

Long version, In principle:



A modern C++ idiom:

RAII: Resource Acquisition Is Initialization.

- When you initialize an object, it should already have acquired any resources it **needs** (in the constructor)
- When an object goes out of scope, it should release resource it is **using** (**using** the destructor).

key point:

- There should never be a half-ready **or** half-dead object
- When an object is created, it should be in a ready state
- When an object goes out of scope, it should release resources
- The user shouldn't have to **do** anything more.

Raw Pointers violate RAI: It needs user to delete manually when the pointers go out of scope.

RAII solution is:

Have a smart pointer **class**:

- Allocates the memory when initialized
- Frees the memory when destructor is called
- Allows access to underlying pointer

For smart pointer need copy and share, use `shared_ptr`:

- use another memory to store Reference counting **and** share
- increment when copy, decrement when destructor.

- `delete` memory when Reference counting is `0`.
also `delete` memory that store Reference counting.

for smart pointer not own the raw pointer, use `weak_ptr`:

- `not` change Reference counting.

`shared_ptr` usage:

correct way:

```
std::shared_ptr<T> t1 = std::make_shared<T>(TArgs);  
std::shared_ptr<T> t2 = std::shared_ptr<T>(new T(TArgs))
```

wrong way:

```
T* pt = new T(TArgs); // never exposure the raw pointer  
shared_ptr<T> t1 = shared_ptr<T>(pt);  
shared_ptr<T> t2 = shared_ptr<T>(pt);
```

Always avoid using raw pointer.

For scenario that have to use raw pointer:

<https://stackoverflow.com/a/19432062/2482283>

For raw pointer that not `nullptr`, use reference instead.

```
not use T*  
use T&
```

For optional reference which maybe `nullptr`, use raw pointer, and which means:

```
T* pt; is optional reference and maybe nullptr.  
Not own the raw pointer,
```

Raw pointer is managed by some one [else](#).
I only know that the caller is sure it is [not](#) released

Share Improve this answer

[edited Aug 9, 2020 at 10:33](#)

Follow

answered Aug 9, 2020 at 10:04



[andrewchan2022](#)

5,270 ● 50 ● 50



2

Smart Pointers are those where you don't have to worry about Memory De-Allocation, Resource Sharing and Transfer.



You can very well use these pointer in the similar way as any allocation works in Java. In java Garbage Collector does the trick, while in Smart Pointers, the trick is done by Destructors.



Share Improve this answer

answered Nov 7, 2016 at 4:07

Follow



[Daksh Gupta](#)

7,804 ● 2 ● 28 ● 36



2

The existing answers are good but don't cover what to do when a smart pointer is not the (complete) answer to the problem you are trying to solve.



Among other things (explained well in other answers) using a smart pointer is a possible solution to [How do we](#)



[use a abstract class as a function return type?](#) which has been marked as a duplicate of this question. However, the first question to ask if tempted to specify an abstract (or in fact, any) base class as a return type in C++ is "what do you really mean?". There is a good discussion (with further references) of idiomatic object oriented programming in C++ (and how this is different to other languages) in the documentation of the [boost pointer container library](#). In summary, in C++ you have to think about ownership. Which smart pointers help you with, but are not the only solution, or always a complete solution (they don't give you polymorphic copy) and are not always a solution you want to expose in your interface (and a function return sounds an awful lot like an interface). It might be sufficient to return a reference, for example. But in all of these cases (smart pointer, pointer container or simply returning a reference) you have changed the return from a *value* to some form of *reference*. If you really needed copy you may need to add more boilerplate "idiom" or move beyond idiomatic (or otherwise) OOP in C++ to more generic polymorphism using libraries like [Adobe Poly](#) or [Boost.TypeErasure](#).

Share Improve this answer

answered Jan 26, 2018 at 3:10

Follow



da77a

73 ● 1 ● 4



Highly active question. Earn 10 reputation (not counting the [association bonus](#)) in order to answer this question. The reputation

requirement helps protect this question from spam and non-answer activity.