Based on:
https://en.cppreference.com/book/intro/smart_pointers and https://en.cppreference.com

Introductory Examples and Notes on C++ Smart Pointers

Smart pointers are used to make sure that an object is deleted if it is no longer used (referenced).

Consider the simple example

```
void my_func()
{
    int* valuePtr = new int(15);
    int x = 45;
    // ...
    if (x == 45)
        return;   // here we have a memory leak, valuePtr is not deleted
    // ...
    delete valuePtr;
}

int main()
{
}
```

The same example using the `unique_ptr<>` template takes the form:

```
#include <memory>

void my_func()
{
    std::unique_ptr<int> valuePtr(new int(15));
    int x = 45;
    // ...
    if (x == 45)
        return;   // no memory leak anymore!
    // ...
}

int main()
{
}
```

The `unique_ptr<>` template holds a pointer to an object and deletes this object when the `unique_ptr<>` object is deleted. This means that in the example above, it does not matter if the function scope is left through the return statement, at the end of the function or through an exception: The `unique_ptr<>` destructor is always called and therefore the object (`int` in this example) is always deleted.

The following three smart pointer templates are available

unique_ptr

```
template<

    class T,
    class Deleter = std::default_delete<T>
> class unique_ptr;

template <

    class T,
    class Deleter
> class unique_ptr<T[], Deleter>;
```

`std::unique_ptr` is a smart pointer that owns and manages another object through a pointer and disposes of that object when the `unique_ptr` goes out of scope.

The object is disposed of using the associated deleter when either of the following happens:

- the managing `unique_ptr` object is destroyed
- the managing `unique_ptr` object is assigned another pointer via operator= or `reset()`.

The object is disposed of using a potentially user-supplied deleter by calling `get_deleter()(ptr)`. The default deleter uses the delete operator, which destroys the object and deallocates the memory.

A `unique_ptr` may alternatively own no object, in which case it is called empty.

There are two versions of `std::unique_ptr`:

1. Manages a single object (e.g. allocated with `new`)
2. Manages a dynamically-allocated array of objects (e.g. allocated with `new[]`)

As the name implies, it makes sure that only exactly one copy of an object exists. Can be used as in the example above for handling dynamically allocated objects in a restricted scope.

A unique pointer can be initiated with a pointer upon creation

```
std::unique_ptr<int> valuePtr(new int(47));
```

or it can be created without a pointer and assigned one later

```
std::unique_ptr<int> valuePtr;
valuePtr.reset(new int(47));
```

Note: In this second case, if the `unique_ptr<>` already holds a pointer to an existing object, this object is deleted first and then the new pointer is stored.

Afterwards, an object managed by a `unique_ptr<>` can be accessed just like when you would use a raw pointer.

```
std::unique_ptr<std::string>  strPtr(new std::string);   strPtr->assign("Hello
world");
```

The `unique_ptr<>` does not support copying. If you try to copy a `unique_ptr<>`, you will get compiler errors. However, it supports move semantics, where the pointer is moved from one `unique_ptr<>` to another, which invalidates the first `unique_ptr<>`.

See the following example:

```
#include <iostream>
#include <memory>
#include <utility>

int main()
{
    std::unique_ptr<int> valuePtr(new int(15));
    std::unique_ptr<int> valuePtrNow(std::move(valuePtr));

    std::cout << "valuePtrNow = " << *valuePtrNow << '\n';
    std::cout << "Has valuePtr an associated object? "
              << std::boolalpha
              << static_cast<bool>(valuePtr) << '\n';
}
```

The output is:

```
valuePtrNow = 15
Has valuePtr an associated object? false
```

shared_ptr

```
template< class T > class shared_ptr;
```

`std::shared_ptr` is a smart pointer that retains shared ownership of an object through a pointer. Several `shared_ptr` objects may own the same object. The object is destroyed and its memory deallocated when either of the following happens:

- the last remaining `shared_ptr` owning the object is destroyed;
- the last remaining `shared_ptr` owning the object is assigned another pointer via operator= or `reset()`.

The object is destroyed using delete-expression or a custom deleter that is supplied to shared_ptr during construction.

A `shared_ptr` can share ownership of an object while storing a pointer to another object. This feature can be used to point to member objects while owning the object they belong to. The stored pointer is the one accessed by `get()`, the dereference and the comparison operators. The managed pointer is the one passed to the deleter when use count reaches zero.

A `shared_ptr` may also own no objects, in which case it is called empty (an empty `shared_ptr` may have a non-null stored pointer if the aliasing constructor was used to create it).

All member functions (including copy constructor and copy assignment) can be called by multiple threads on different instances of `shared_ptr` without additional synchronization even if these instances are copies and share ownership of the same object. If multiple threads of execution access the same `shared_ptr` without synchronization and any of those accesses uses a non-const member function of `shared_ptr` then a data race will occur; the `shared_ptr` overloads of atomic functions can be used to prevent the data race.

The shared_pointer is a reference counting smart pointer that can be used to store and pass a reference beyond the scope of a function. This is particularly useful in the context of Object Oriented Programming, to store a pointer as a member variable and return it to access the referenced value outside the scope of the class. Consider the following example:

```
#include <memory>

class Foo
{
        public void doSomething();
};

class Bar
{
```

```
private:
        std::shared_ptr<Foo> pFoo;
public:
        Bar()
        {
                pFoo = std::shared_ptr<Foo>(new Foo());
        }

        std::shared_ptr<Foo> getFoo()
        {
                return pFoo;
        }
};
```

When an object of the `Bar` class is created it creates a new object of the `Foo` class, which is stored in `pFoo`. To Access `pFoo` we can call `Bar::getFoo` which returns a `std::shared_ptr` to the `Foo` object created in the `Bar` constructor. Internally, a copy of the `std::shared_ptr` object is created and returned. The copy constructor of `std::shared_ptr` copies the internal pointer to the Foo object and increases the reference count. This would, for example, happen in the following example:

```
void SomeAction()
{
        Bar* pBar = new Bar(); //with the Bar object, a new Foo is created and
stored
        //reference counter = 1

        std::shared_ptr<Foo> pFoo = pBar->getFoo(); //a copy of the shared pointer
is created
        //reference counter = 2

        pFoo->doSomething();

        delete pBar; //with pBar the private pFoo is destroyed
        //reference counter = 1

        return; //with the return the local pFoo is destroyed automatically
        //reference counter = 0
        //internally the std::shared_ptr destroys the reference to the Foo object
}
```

So there's no need for `Bar` to care about deleting `pFoo`, which increases memory management severely.

```
void SomeOtherAction(std::shared_ptr<Bar> pBar)
{
        std::shared_ptr<Foo> pFoo = pBar->getFoo(); //a copy of the shared pointer
is created
        //reference counter = 2

        pFoo->doSomething();

        return; //local pFoo is destroyed
        //reference counter = 1
}
```

When the function returns, `pBar` is deleted, but there is still a copy of the `std::shared_ptr` outside the scope of the function, therefor the internal Bar object will not be destroyed, which sustains the reference to the Foo object, which is in turn not destroyed either. The usage of `smart_ptr` allows us to easily pass and return references to objects without running into memory leaks or invalid attempts to access deleted references. They are thus a cornerstone of modern memory management.

weak_ptr

```
template< class T > class weak_ptr;
```

`std::weak_ptr` is a smart pointer that holds a non-owning ("weak") reference to an object that is managed by `std::shared_ptr`. It must be converted to `std::shared_ptr` in order to access the referenced object.

`std::weak_ptr` models temporary ownership: when an object needs to be accessed only if it exists, and it may be deleted at any time by someone else, `std::weak_ptr` is used to track the object, and it is converted to `std::shared_ptr` to assume temporary ownership. If the original `std::shared_ptr` is destroyed at this time, the object's lifetime is extended until the temporary `std::shared_ptr` is destroyed as well.

In addition, `std::weak_ptr` is used to break circular references of std::shared_ptr.

Like `std::shared_ptr`, a typical implementation of `weak_ptr` stores two pointers:

- a pointer to the control block; and
- the stored pointer of the `shared_ptr` it was constructed from.

A separate stored pointer is necessary to ensure that converting a `shared_ptr` to `weak_ptr` and then back works correctly, even for aliased `shared_ptrs`. It is not possible to access the stored pointer in a `weak_ptr` without locking it into a `shared_ptr`.

Based on: https://en.cppreference.com/book/intro/smart_pointers and https://en.cppreference.com

An example that demonstrates how lock is used to ensure validity of the pointer is shown below:

```
#include <iostream>
#include <memory>

std::weak_ptr<int> gw;

void observe()
{
    std::cout << "use_count == " << gw.use_count() << ": ";
    if (auto spt = gw.lock()) { // Has to be copied into a shared_ptr before usage
        std::cout << *spt << "\n";
    }
    else {
        std::cout << "gw is expired\n";
    }
}

int main()
{
    {
        auto sp = std::make_shared<int>(42);
        gw = sp;

        observe();
    }

    observe();
}
```

The output is:

```
use_count == 1: 42
use_count == 0: gw is expired
```