Lecture 9

Smart Pointers

Review: Dynamic Memory

Instantiating an object with “new” uses the free store – **dynamic memory** – and returns a pointer to the object:

```
StubbornPouch<std::string>* myPouchPtr =
       new StubbornPouch<std::string>();
```

Now, we use the following syntax to call a method on the object:

```
myPouchPtr ->setItem(shockItem);
```

When using dynamic memory, **we must deallocate to avoid memory leaks:**

```
delete myPouchPtr;
```

`myPouchPtr = nullptr;` (only if the pointer is not within an object or function)

Review: Dynamic Memory

Dynamic memory management is complex

→ Example: Avoiding memory leaks or dangling pointers when deleting a Node in a linked list

Smart Pointers

Languages like Java and Python do not allow direct references (pointers) because of associated memory issues.

These languages use **reference counting** to track the number of references to an object, and objects with no remaining references are periodically deallocated automatically – this is **garbage collection.**

Smart Pointers

Languages like Java and Python do not allow direct references (pointers) because of associated memory issues.

These languages use **reference counting** to track the number of references to an object, and objects with no remaining references are periodically deallocated automatically – this is **garbage collection.**

→ Safer memory management, but can slow performance

Smart Pointers

In C++, **smart pointers** provide some automatic memory management features.

Smart Pointers

In C++, **smart pointers** provide some automatic memory management features.

→ Recall, automatic memory management handles deallocation based on **scope** (e.g., object data, function local variables)

Smart Pointers

In C++, **smart pointers** provide some automatic memory management features.

→ Recall, automatic memory management handles deallocation based on **scope** (e.g., object data, function local variables)

→ Also called **managed pointers**

Smart Pointers

In C++, **smart pointers** provide some automatic memory management features.

→ Recall, automatic memory management handles deallocation based on **scope** (e.g., object data, function local variables)

→ Also called **managed pointers**

Pointers that we have been using so far are sometimes called **raw pointers** to distinguish them from smart pointers.

Smart Pointers

When a smart pointer goes out of scope, its destructor **automatically invokes the destructor of the referenced object.**

Smart Pointers

When a smart pointer goes out of scope, its destructor **automatically invokes the destructor of the referenced object.**

→ You do not have to use “delete” when using smart pointers!

Smart Pointers

```
#include <memory>
```

```
shared_ptr
```

```
unique_ptr
```

```
weak_ptr
```

Smart Pointers

`#include <memory>`

`shared_ptr` : shared ownership of an object

→ Several instances can reference the same object

→ Each shared pointer increases an object's reference count

`unique_ptr`

`weak_ptr`

Smart Pointers

`#include <memory>`

`shared_ptr` : shared ownership of an object

→ Several instances can reference the same object

→ Each shared pointer increases an object’s reference count

`unique_ptr` : unique ownership of an object

→ No other pointers can reference the same object

`weak_ptr`

Smart Pointers

```
#include <memory>
```

`shared_ptr` : shared ownership of an object
→ Several instances can reference the same object
→ Each shared pointer increases an object's reference count

`unique_ptr` : unique ownership of an object
→ No other pointers can reference the same object

`weak_ptr` : reference to an object already managed by a shared pointer
→ Does not have ownership of the object, so cannot deallocate

Smart Pointers

```
std:shared_ptr< PlainPouch<std::string> >
   myPouchPtr( new PlainPouch<std::string>() );
```

Smart Pointers

```
std:shared_ptr< PlainPouch<std::string> >
  myPouchPtr( new PlainPouch<std::string>() );
```

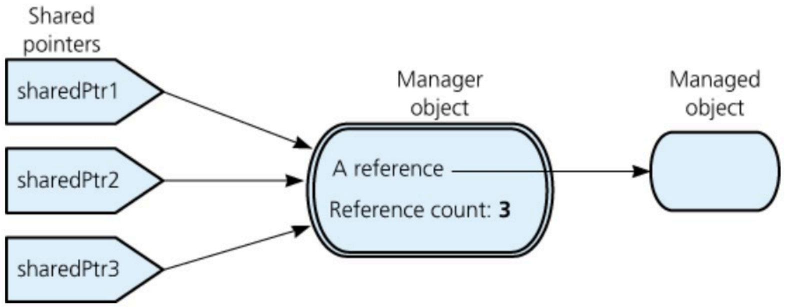
…

```
myPouchPtr->getItem()
```

Smart Pointers

```
std:shared_ptr< PlainPouch<std::string> >
   myPouchPtr( new PlainPouch<std::string>() );
```

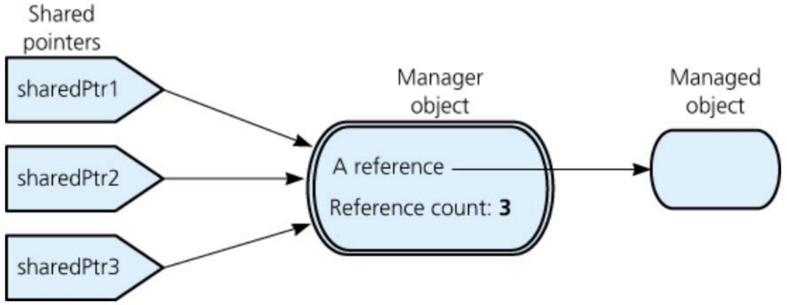
This actually dynamically creates PlainPouch and then the shared pointer – so the shared_ptr constructor allocates a **manager** object:



Smart Pointers

```
std:shared_ptr< PlainPouch<std::string> >
   myPouchPtr( new PlainPouch<std::string>() );
```

This actually dynamically creates PlainPouch and then the shared pointer – so the shared_ptr constructor allocates a **manager** object:



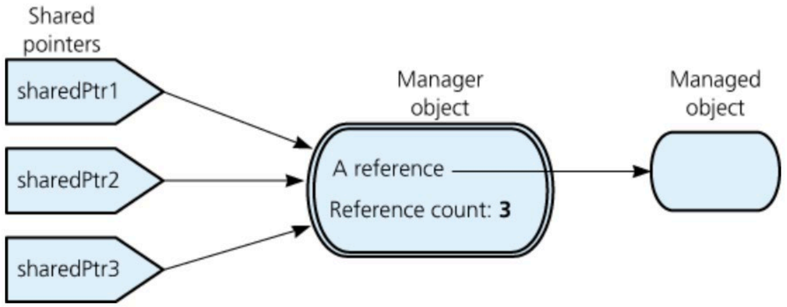
Combine into just one allocation for better performance:

```
std::shared_ptr< PlainPouch<std::string> >
   myPouchPtr = std::make_shared< PlainPouch<std::string> >();
```

Smart Pointers

```
std:shared_ptr< PlainPouch<std::string> >
   myPouchPtr( new PlainPouch<std::string>() );
```

This actually dynamically creates PlainPouch and then the shared pointer – so the shared_ptr constructor allocates a **manager** object:



Combine into just one allocation for better performance:

```
std::shared_ptr< PlainPouch<std::string> >
   myPouchPtr = std::make_shared< PlainPouch<std::string> >();
```

Or, with C++14 and later:

```
auto  myPouchPtr = std::make_shared< PlainPouch<std::string> >();
```

Smart Pointers: Summary

Smart pointers:

- Provide a safer mechanism for memory management
- Maintain a count of references to an object
  - Increase reference count with each shared pointer that references that object
  - Decrease reference count when shared pointers go out of scope (or are assigned nullptr)
- Call the destructor of the managed object when reference count reaches 0

Smart Pointers: Summary

Smart pointers:

- Provide a safer mechanism for memory management
- Maintain a count of references to an object
  - Increase reference count with each shared pointer that references that object
  - Decrease reference count when shared pointers go out of scope (or are assigned nullptr)
- Call the destructor of the managed object when reference count reaches 0

Don’t mix smart pointers and raw pointers!

Assignment/Homework

- P2 due Tomorrow
- ICE 5 due on Tuesday
- P3 released: XML Parser