Smart Pointers

Smart Pointers

- Manual memory management is error-prone
  as we have seen with memory leaks in particular

- The C++ Standard library introduced new generic data types that are commonly referred to as smart pointers

- These data types, smart pointers, automatically manage the dynamic allocation and release of memory

Smart Pointers

- The smart pointer class has-a raw pointer member variable which “owns” the dynamically allocated memory
- When a smart pointer object is instantiated, the pointer member will also be created, pointing to allocated memory
- When the smart pointer goes out of scope, smart pointers automatically free the allocated memory

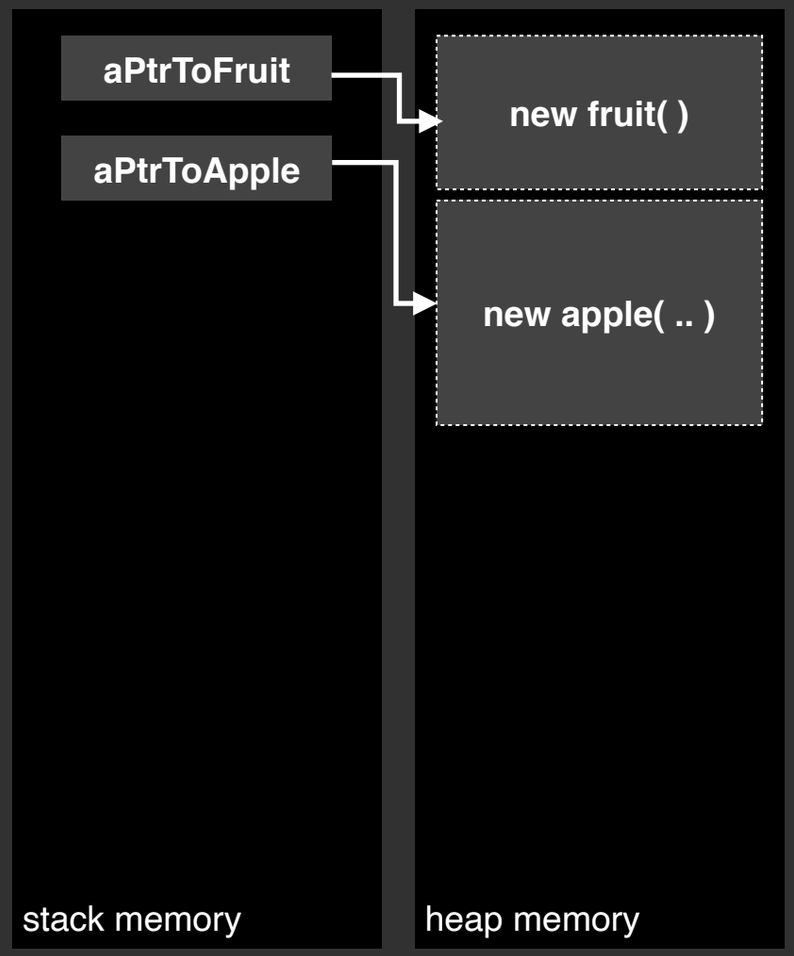
Smart Pointers

source code instructions

```
 9  #include "fruit.h"
10  #include "apple.h"
11  #include "banana.h"
12
13
14  int main()
15  {
16      fruit* aPtrToFruit{ new fruit() };
17      apple* aPtrToApple{ new apple(apple::appleType::BRAEBURN) };
18
19      aPtrToFruit->printName();
20      aPtrToApple->printName();
21
22      delete aPtrToFruit;
23      delete aPtrToApple;
24
25      return 0;
26  }
27
```

Traditional use of **raw pointers** for memory allocation & release

computer memory | lines 21



Smart Pointers

source code instructions

```
 9 #include "fruit.h"
10 #include "apple.h"
11 #include "banana.h"
12
13 #include <memory>   // smart pointer functionality
14
15
16 int main()
17 {
18     std::shared_ptr< fruit > sharedFruitPtr{ new fruit("hello") };
19     std::shared_ptr< apple > sharedApplePtr{ new apple(apple::appleType::BRAEBURN) };
20
21     (*sharedFruitPtr).printName();  // dereferening & calling the function via "."
22     sharedApplePtr->printName();    // directly calling the function via "->"
23
24     return 0;
25 }
26
```

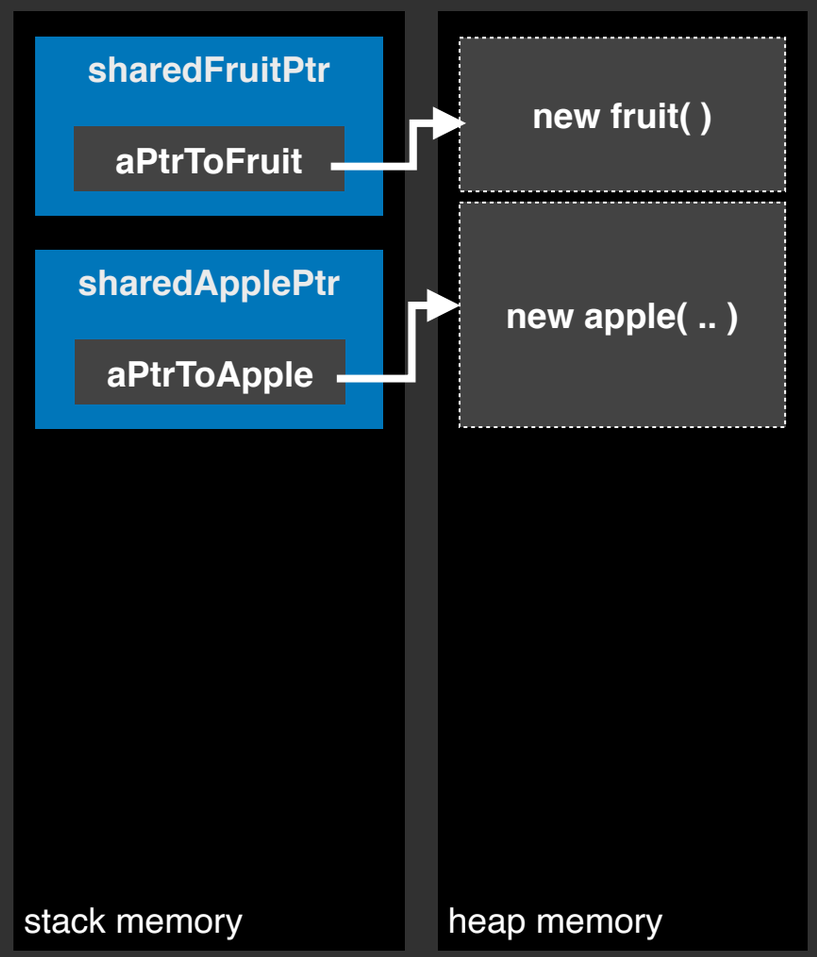
Smart Pointers

source code instructions

```
 9 #include "fruit.h"
10 #include "apple.h"
11 #include "banana.h"
12
13 #include <memory>   // smart pointer functionality
14
15
16 int main()
17 {
18     std::shared_ptr< fruit > sharedFruitPtr{ new fruit("hello") };
19     std::shared_ptr< apple > sharedApplePtr{ new apple(apple::appleType::BRAEBURN) };
20
21     (*sharedFruitPtr).printName();  // dereferening & calling the function via "."
22     sharedApplePtr->printName();    // directly calling the function via "->"
23
24     return 0;
25 }
26
```

When the smart pointer class “shared_ptr” goes out of scope at the end of the function, it releases the allocated memory by calling “delete” in its destructor automatically.

computer memory



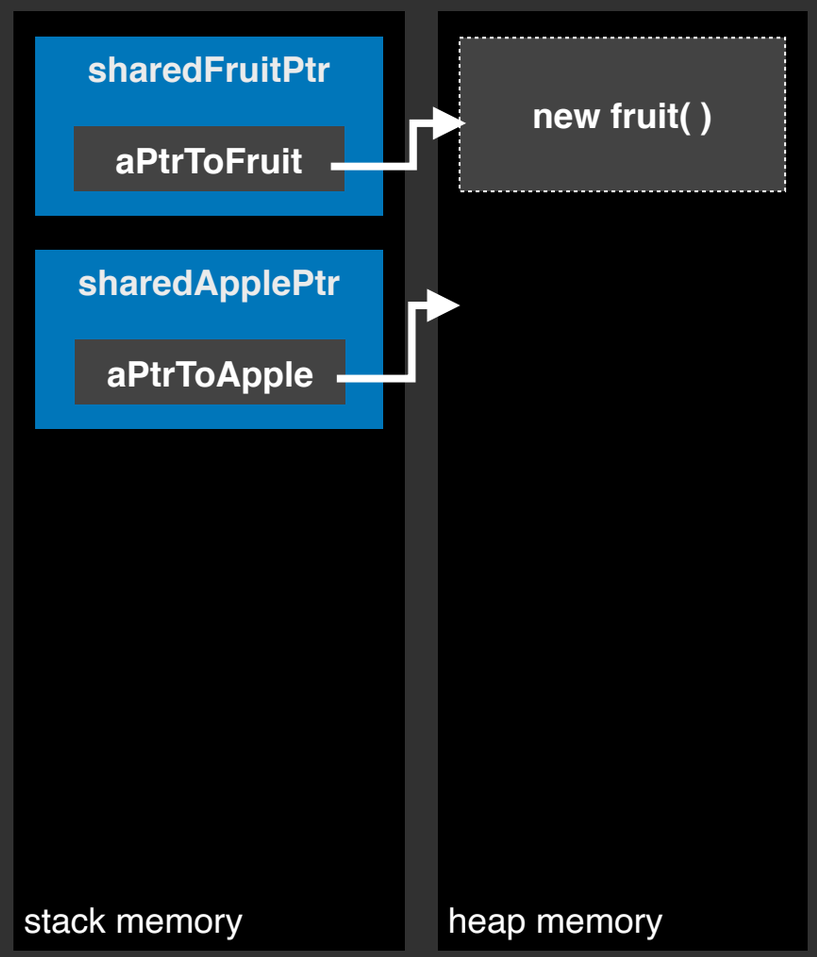
Smart Pointers

source code instructions

```
 9 #include "fruit.h"
10 #include "apple.h"
11 #include "banana.h"
12
13 #include <memory>   // smart pointer functionality
14
15
16 int main()
17 {
18     std::shared_ptr< fruit > sharedFruitPtr{ new fruit("hello") };
19     std::shared_ptr< apple > sharedApplePtr{ new apple(apple::appleType::BRAEBURN) };
20
21     (*sharedFruitPtr).printName();  // dereferening & calling the function via "."
22     sharedApplePtr->printName();    // directly calling the function via "->"
23
24     return 0;
25 }
26
```

When the smart pointer class “shared_ptr” goes out of scope at the end of the function, it releases the allocated memory by calling “delete” in its destructor automatically.

computer memory | line 25



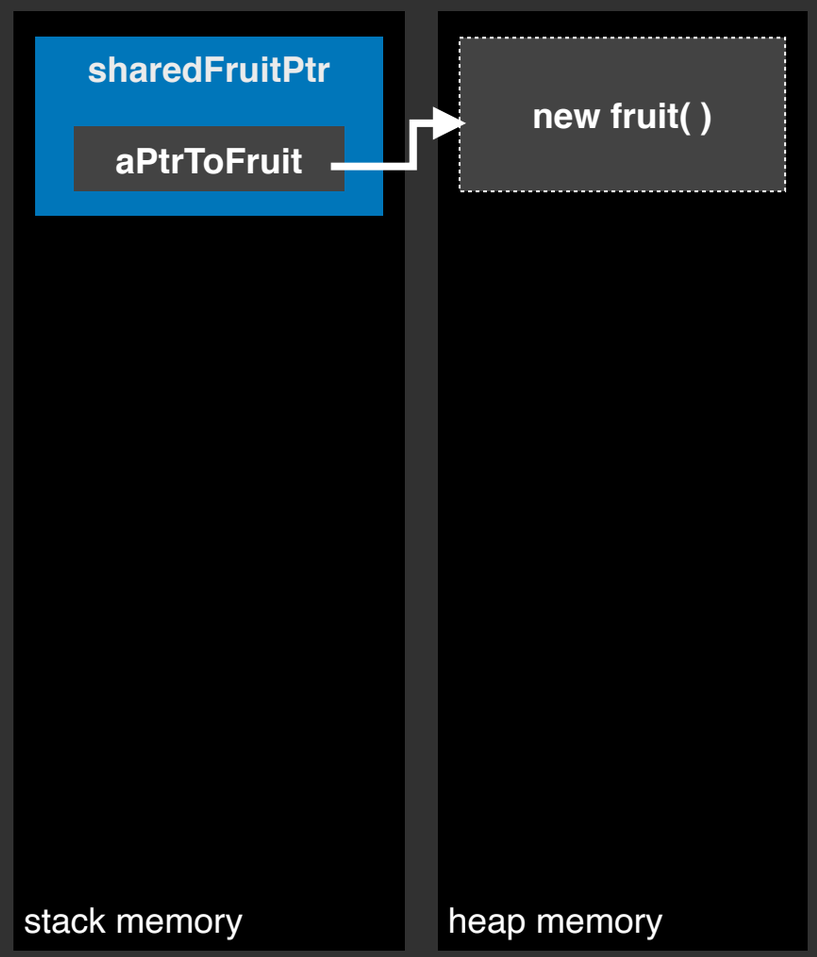
Smart Pointers

source code instructions

```
 9 #include "fruit.h"
10 #include "apple.h"
11 #include "banana.h"
12
13 #include <memory>   // smart pointer functionality
14
15
16 int main()
17 {
18     std::shared_ptr< fruit > sharedFruitPtr{ new fruit("hello") };
19     std::shared_ptr< apple > sharedApplePtr{ new apple(apple::appleType::BRAEBURN) };
20
21     (*sharedFruitPtr).printName();  // dereferening & calling the function via "."
22     sharedApplePtr->printName();    // directly calling the function via "->"
23
24     return 0;
25 }
26
```

When the smart pointer class “shared_ptr” goes out of scope at the end of the function, it releases the allocated memory by calling “delete” in its destructor automatically.

computer memory | line 25



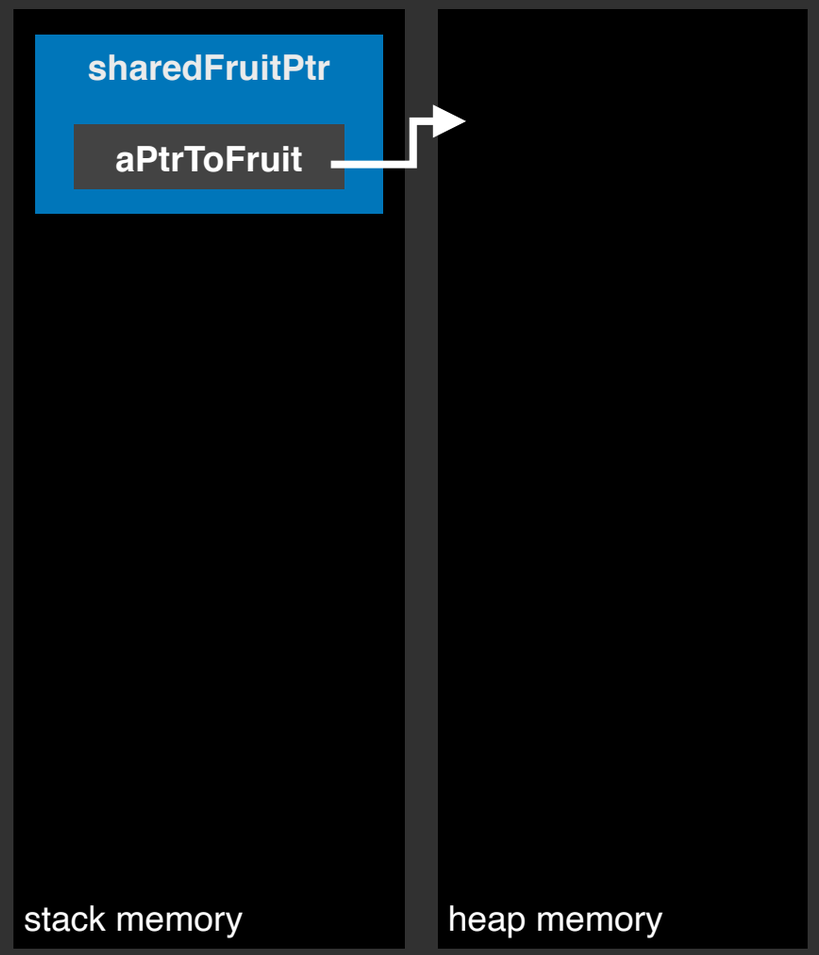
Smart Pointers

source code instructions

```
 9 #include "fruit.h"
10 #include "apple.h"
11 #include "banana.h"
12
13 #include <memory>   // smart pointer functionality
14
15
16 int main()
17 {
18     std::shared_ptr< fruit > sharedFruitPtr{ new fruit("hello") };
19     std::shared_ptr< apple > sharedApplePtr{ new apple(apple::appleType::BRAEBURN) };
20
21     (*sharedFruitPtr).printName();  // dereferening & calling the function via "."
22     sharedApplePtr->printName();    // directly calling the function via "->"
23
24     return 0;
25 }
26
```

When the smart pointer class “shared_ptr” goes out of scope at the end of the function, it releases the allocated memory by calling “delete” in its destructor automatically.

computer memory | line 25



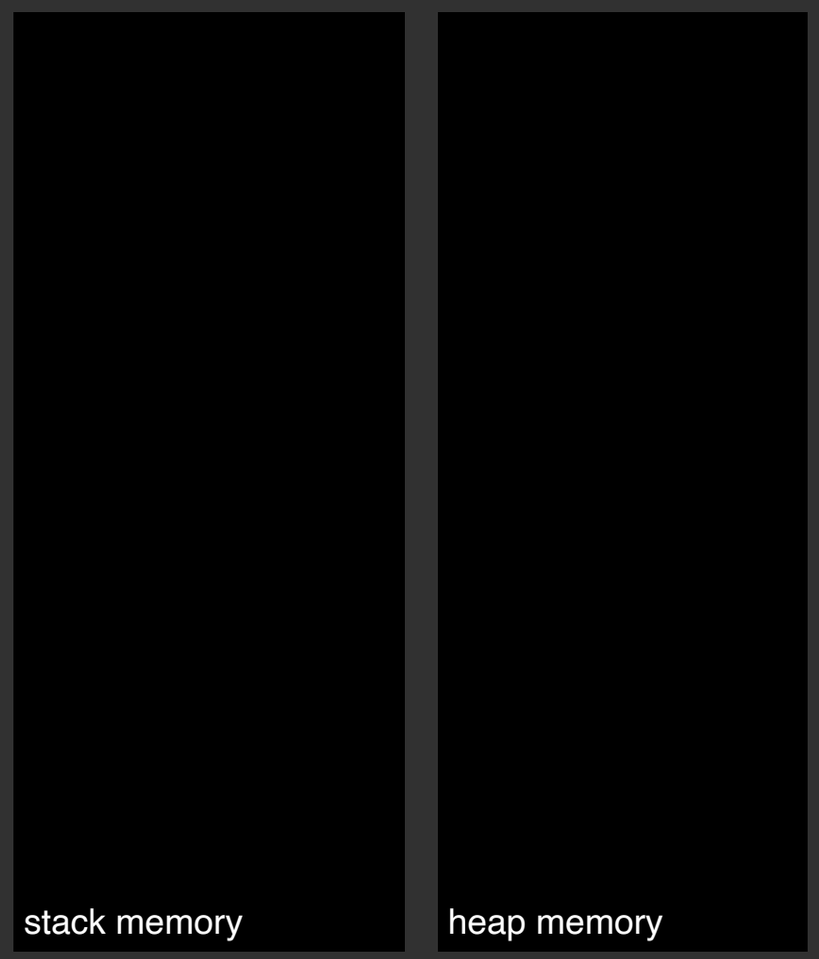
Smart Pointers

source code instructions

```
 9 #include "fruit.h"
10 #include "apple.h"
11 #include "banana.h"
12
13 #include <memory>   // smart pointer functionality
14
15
16 int main()
17 {
18     std::shared_ptr< fruit > sharedFruitPtr{ new fruit("hello") };
19     std::shared_ptr< apple > sharedApplePtr{ new apple(apple::appleType::BRAEBURN) };
20
21     (*sharedFruitPtr).printName();  // dereferening & calling the function via "."
22     sharedApplePtr->printName();    // directly calling the function via "->"
23
24     return 0;
25 }
26
```

When the smart pointer class “shared_ptr” goes out of scope at the end of the function, it releases the allocated memory by calling “delete” in its destructor automatically.

computer memory | line 25



Smart Pointers

- C++ standard library supports few different types of smart pointers that can be used by including the <memory> header

- Most prominent smart pointer classes are
  - std::unique_ptr< T > — featuring unique ownership
  - std::shared_ptr< T > — featuring shared ownership

Smart Pointers

- std::shared_ptr< T > is also used quite heavily in openFrameworks, replacing traditional raw pointers
  - See this tutorial on memory management
  - https://openframeworks.cc/ofBook/chapters/memory.html

- More information on smart pointers in general
  - Chapter 15 “Move semantics & smart pointers”
  - http://www.learncpp.com/cpp-tutorial/15-1-intro-to-smart-pointers-move-semantics/

Take Away

- Understanding basics of C++ memory allocation is crucial when having to handle dynamic data
- Smart pointers simplify and automate the use and management of dynamically allocated memory
- Stick with automatic memory allocation for fixed values and make use of smart pointers for any dynamic data