Smart Pointers

VGP 131 – OOP II

Instructor: Ivaldo Tributino

Memory management

Pointers in C and C++ languages are extremely powerful, yet so dangerous:

Since their management is completely left up to you, every dynamically allocated object (new T) must be followed by manual deallocation (delete T). Forget to do that and you will end up with a **memory leak**.

Moreover, dynamically allocated arrays (new T[N]) require to be deleted with a different operator (delete[]). This forces you to mentally keep track of what you have allocated and call the right operator accordingly. Using the wrong form results in undefined behavior[8], something you really want to avoid at all costs when working in C++.[4]

Memory leaks

The memory consumed by **p2** will not be deallocated because we forgot to use **delete p2**; at the end of the function.

```
cout << "- Polygon on heap" << endl;
{
    Polygon* p1 = new Polygon();
    Polygon* p2 = new Polygon(4,4); // memory leak
    cout << p1->shapeName() << endl;
    cout << p2->shapeName() << endl;
    delete p1;
}
```

That means the memory will not be free to be used by other resources. However, we do not need the variable anymore, but we need the memory.

MEMORY LEAK

main.cpp | Polygon.h | smartPtr.h

main.cpp > main()

```cpp
//***********************************************
//        -------- Memory leak --------
//***********************************************
cout << "-------- Memory leak --------" << endl;

/* Note: If your object is in the stack memory, your destructor is going to be called as
soon as the function returns. If it's on the heap, the destructor is only called when
that delete keyword is used.*/

cout << "- Polygon on stack" << endl;
{
    Polygon poly;
    cout << poly.shapeName() << endl;
}
cout << "- Polygon on heap" << endl;
{
    Polygon* p1 = new Polygon();
    Polygon* p2 = new Polygon(4,4); // memory leak
    cout << p1->shapeName() << endl;
    cout << p2->shapeName() << endl;
    delete p1;
}
{
    Polygon* p = new Polygon;  // memory leak
    Polygon square(2,4);
    p = &square;
}
```

PROBLEMS | OUTPUT | TERMINAL | DEBUG CONSOLE

bash

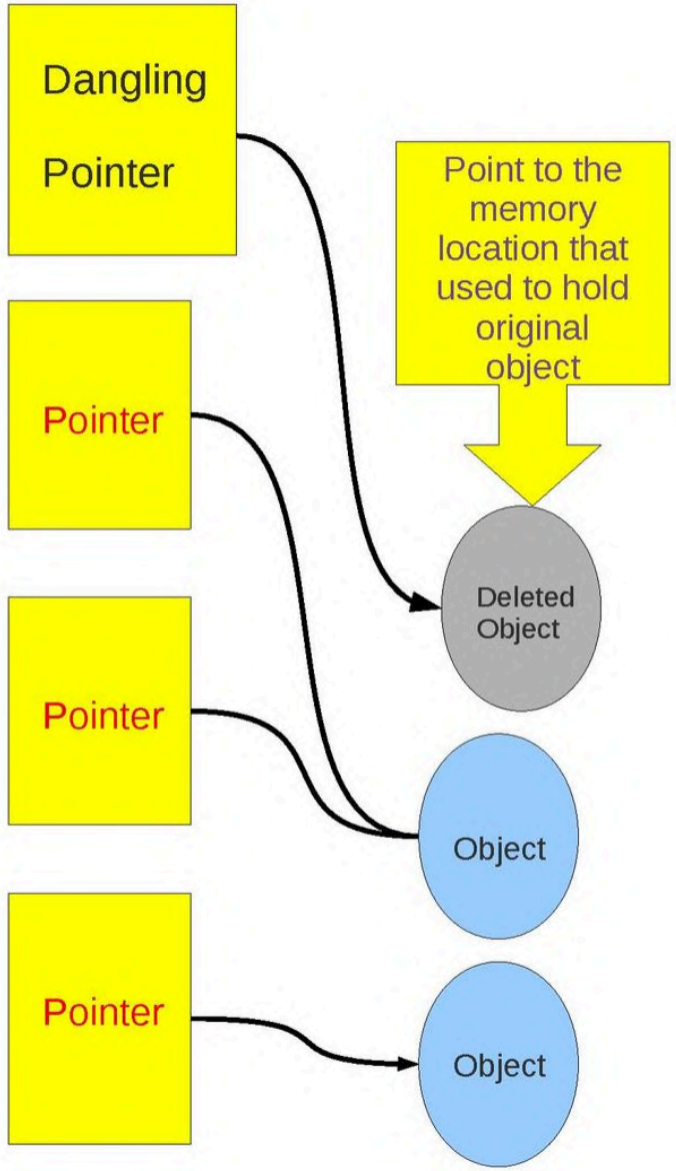
```
-------- Memory leak --------
- Polygon on stack
Default Constructor Invoked
triangle
Polygon destroyed
- Polygon on heap
Default Constructor Invoked
Constructor Invoked
triangle
square
Polygon destroyed
Default Constructor Invoked
Constructor Invoked
Polygon destroyed
```

master*+ | 0 0 | Live Share | Ln 50, Col 1 | Spaces: 4 | UTF-8 | LF | C++ | Mac

Dangling pointer

- A pointer pointing to a memory location that has been deleted/de-allocated or falls out of scope is called **Dangling Pointer**.

- How can we prevent the pointer from being incorrectly **deleted multiple times**?



https://en.wikipedia.org/wiki/Dangling_pointer

DANGLING POINTER

main.cpp

main.cpp > main()

```
42          cout << p2->shapeName() << endl;
43          delete p1;
44      }
45
46      //************************************************
47      //       -------- Dangling pointer --------
48      //************************************************
49      cout << "-------- Dangling pointer --------" << endl;
50
51      //Declaring two pointer variables to Polygon
52      Polygon * p1;
53      Polygon * p2;
54
55      // Allocating dynamic memory in the heap
56      p1 = new Polygon(3,5);
57      p2 = p1;    // Having both pointers to point same dynamic memory location
58
59      //deleting the dynamic memory location
60      delete p1; // release the memory
61      p1 = nullptr;
62
63      /*
64      p2 is still pointing the already deleted memory location, We call p2 dangling pointer
65
66      If we try to release that location once more like delete p2, it will crash after
67      throwing an exception.
68      */
69
70      //************************************************
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

bash

```
-------- Memory leak --------
- Polygon on stack
Default Constructor Invoked
triangle
Polygon destroyed
- Polygon on heap
Default Constructor Invoked
Constructor Invoked
triangle
square
Polygon destroyed
-------- Dangling pointer --------
```

master*+ 0 0 Live Share Ln 56, Col 27 Spaces: 4 UTF-8 LF C++ Mac

Smart pointers

- Smart pointers were first popularized in the programming language C++ during the first half of the 1990s as rebuttal to criticisms of C++'s lack of automatic garbage collection.[2]

- Smart pointers prevent most situations of memory leaks by making the memory deallocation automatic. An object controlled by a smart pointer is automatically destroyed (finalized and then deallocated). Smart pointers also eliminate dangling pointers by postponing destruction until an object is no longer in use.

- 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.

Our Smart pointer

- A Smart Pointer is a wrapper class over a pointer with an operator like * and -> overloaded. Objects of the smart pointer class look like normal pointers. But unlike normal pointers, it can deallocate and free the destroyed object's memory. [3]

- So before we use the libraries that give us smart pointers, let's create our own to get a better idea of how they work.

- The idea is to get a class with a pointer, destructor, and overloaded operators like * and ->. Let's consider the following simple SmartPtr class.

OUR SMART POINTER

```cpp
template <class T>
class SmartPtr {

private:

    T* ptr;

public:

    // Constructor
    explicit SmartPtr(T* p = nullptr); // controls unwanted implicit type conversions.

    // Destructor
    ~SmartPtr();

    // Overloading dereferncing operator
    T& operator*();

    // Overloading arrow operator
    T* operator->();

    // Preventing the compiler from generating copy
    SmartPtr(const SmartPtr&) = delete;

    SmartPtr& operator=(const SmartPtr&) = delete;

};
```

```cpp
template <class T>
SmartPtr<T> :: SmartPtr(T* p) : ptr(p)
{
    cout << "Pointer Constructor Invoked"  << endl;
}

template <class T>
SmartPtr<T> :: ~SmartPtr()
{
    delete (ptr);
    cout << "Pointer deleted" << endl;
}

template <class T>
T& SmartPtr<T> :: operator*()
{
    return *ptr;
}

template <class T>
T* SmartPtr<T> :: operator->()
{
    return ptr;
}
```

OUR SMART POINTER

main.cpp | Polygon.h | smartPtr.h

main.cpp > main()

```
63
64      //deleting the dynamic memory location
65      delete p1; // release the memory
66      p1 = nullptr;
67
68      /*
69      p2 is still pointing the already deleted memory location, We call p2 dangling pointer
70
71      If we try to release that location once more like delete p2, it will crash after
72      throwing an exception.
73      */
74
75      //************************************************
76      //   -------- Our Smart pointer in action --------
77      //************************************************
78      cout << "-------- Our Smart pointer in action --------" << endl;
79      {
80          SmartPtr<Polygon> p1(new Polygon());
81          SmartPtr<Polygon> p2(new Polygon(4,3));
82          cout << p1->area() << endl;
83          cout << p2->area() << endl;
84      }
85
86
87
88
89
90
```

PROBLEMS | OUTPUT | TERMINAL | DEBUG CONSOLE

bash

```
-------- Our Smart pointer in action --------
Default Constructor Invoked
Pointer Constructor Invoked
Constructor Invoked
Pointer Constructor Invoked
0.433013
6.9282
Polygon destroyed
Pointer deleted
Polygon destroyed
Pointer deleted
(base) Ivaldo:week6 admin$
```

master*+ | 0 ⚠ 0 | Live Share | Ln 93, Col 5 | Spaces: 4 | UTF-8 | LF | C++ | Mac

Smart pointers in modern C++

- C++11 has introduced three types of smart pointers, all of them defined in the <memory> header from the Standard Library:

- **std::unique_ptr** — a smart pointer that owns a dynamically allocated resource (exclusive ownership);

- **std::shared_ptr** — a smart pointer that owns a shared dynamically allocated resource. Several std::shared_ptrs may own the same resource and an internal count keeps track of them;

- **std::weak_ptr** — like a std::shared_ptr, but it doesn't increment the count.

unique_ptr

- As the name implies, make sure that only exactly one copy of an object exists. So std::unique_ptr doesn't have a copy constructor:

- The unique_ptr<> template holds a pointer to an object and deletes this object when the unique_ptr<> object is deleted.

- A unique pointer can be initialized with a pointer upon creation:

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

- It is also possible to construct std::unique_ptrs with the help of the special function std::make_unique, like this:

```
std::unique_ptr<int> ptr = std::make_unique<int>(47);
```

- Afterward, an object managed by a unique_ptr<> can be accessed just like when you would use a raw pointer:

std::make_unique/ unique_ptr::get

main.cpp ● | Polygon.h | smartPtr.h

main.cpp > main()

```
78  cout << "-------- Our Smart pointer in action --------" << endl;
79  {
80      SmartPtr<Polygon> p1(new Polygon());
81      SmartPtr<Polygon> p2(new Polygon(4,3));
82      cout << p1->area() << endl;
83      cout << p2->area() << endl;
84  }
85
86  //***********************************************
87  //    -------- std::unique_ptrs --------
88  //***********************************************
89  cout << "-------- std::unique_ptrs --------" << endl;
90  {
91  // not recommended  unique_ptr<int> ptr1(new int(23));
92  unique_ptr<int> ptr1 = make_unique<int>(23);
93  cout << ptr1 << endl;
94  cout << *ptr1 << endl;
95
96  int *ptr2;
97  // ptr2 = ptr1; incompatibility
98  ptr2 = ptr1.get(); // Returns the stored pointer
99  cout << ptr2 << endl;
100 cout << *ptr2 << endl;
101
102
103
104
105
106
107
108
109
```

PROBLEMS | OUTPUT | TERMINAL | DEBUG CONSOLE

bash

```
-------- std::unique_ptrs --------
0x7fc5be400640
23
0x7fc5be400640
23
(base) Ivaldo:week6 admin$
```

master*+ | 0 ⚠ 0 | Live Share | Ln 112, Col 5 | Spaces: 4 | UTF-8 | LF | C++ | Mac

std::move()

main.cpp | Polygon.h | smartPtr.h

main.cpp > main()

```
    cout << p2->area() << endl;
}

//************************************************
//   -------- std::unique_ptrs --------
//************************************************
cout << "-------- std::unique_ptrs --------" << endl;

// not recommended  unique_ptr<int> ptr1(new int(23));
unique_ptr<int> ptr1 = make_unique<int>(23);
cout << ptr1 << endl;
cout << *ptr1 << endl;

int *ptr2;
// ptr2 = ptr1; incompatibility
ptr2 = ptr1.get(); // Returns the stored pointer
cout << ptr2 << endl;
cout << *ptr2 << endl;

// std::unique_ptr does not have a copy constructor:
// unique_ptr<int> ptr3 = ptr1;
unique_ptr<int> ptr3 = move(ptr1); // memory resource is transferred to another unique_ptr
cout << ptr1 << endl;
cout << ptr2 << endl;
cout << ptr3 << endl;
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
Pointer deleted
-------- std::unique_ptrs --------
0x7fb808400640
23
0x7fb808400640
23
0x0
0x7fb808400640
0x7fb808400640
(base) Ivaldo:week6 admin$
```

A unique_ptr can only be moved. This means that the ownership of the memory resource is transferred to another unique_ptr and the original unique_ptr no longer owns it.

master*+ ⊗ 0 ⚠ 0 Live Share Ln 118, Col 1 Spaces: 4 UTF-8 LF C++ Mac

Reset & Release

main.cpp | Polygon.h | smartPtr.h

main.cpp > main()

```
100     cout << *ptr2 << endl;
101
102     // std::unique_ptr does not have a copy constructor:
103     // unique_ptr<int> ptr3 = ptr1;
104     unique_ptr<int> ptr3 = move(ptr1); // memory resource is transferred to another unique_ptr
105     cout << ptr1 << endl;
106     cout << ptr2 << endl;
107     cout << ptr3 << endl;
108
109     {
110     unique_ptr<Polygon> unPolyPtr = make_unique<Polygon>(2,10);
111     cout << unPolyPtr->getNumberSides() << endl;
112
113     unPolyPtr.reset(new Polygon(2,4));
114     cout << "Using reset" << endl;
115
116     Polygon *polyPtr;
117     polyPtr = unPolyPtr.release();
118     cout << "Using release" << endl;
119     cout << polyPtr->shapeName() << endl;
120     if (unPolyPtr == nullptr){
121         cout << "null pointer" << endl;
122     };
123
124     delete polyPtr;
125     }
126
127
128
129
```

Destroys the object currently managed by the unique_ptr (if any) and takes ownership of *another one*.

Releases ownership of its stored pointer, by returning its value and replacing it with a *null pointer*.

This call does not destroy the managed object, but the unique_ptr object is released from the responsibility of deleting the object.

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
Constructor Invoked
10
Constructor Invoked
Polygon destroyed
Using reset
Using release
square
null pointer
Polygon destroyed
(base) Ivaldo:week6 admin$
```

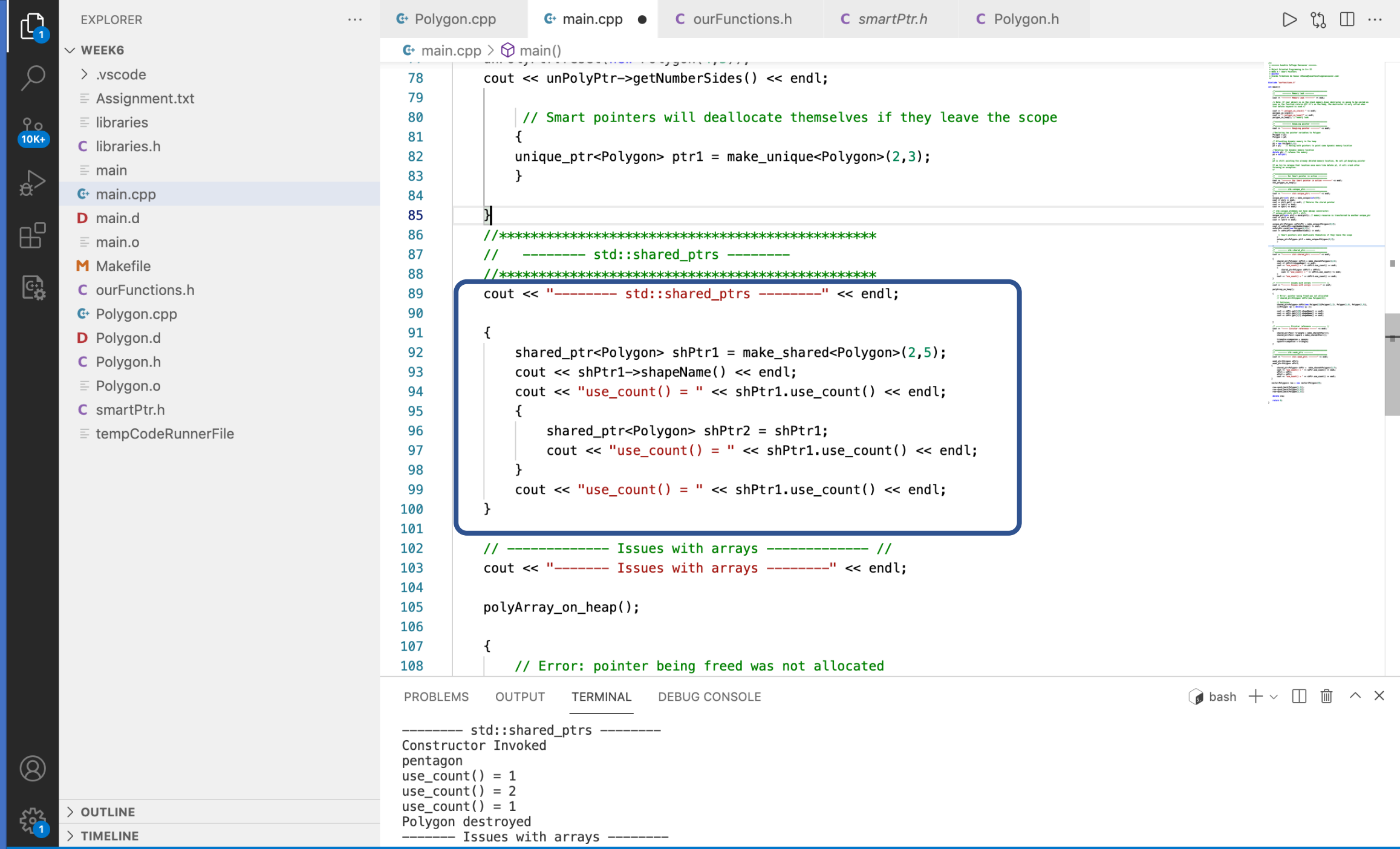
master*+ ⊗ 0 ⚠ 0 Live Share Ln 121, Col 30 Spaces: 4 UTF-8 LF C++ Mac

shared_ptr

- 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.

- After you initialize a shared_ptr you can copy it, pass it by value in function arguments, and assign it to other shared_ptr instances. All the instances point to the same object, and share access to one "control block" that increments and decrements the reference count whenever a new shared_ptr is added, goes out of scope, or is reset. When the reference count reaches zero, the control block deletes the memory resource and itself.[7]

SHARED POINTER



ISSUES WITH ARRAYS

main.cpp | Polygon.h | smartPtr.h

main.cpp > main()

```
144     {
145         shared_ptr<Polygon> shPtr1 = make_shared<Polygon>(2,5);
146         cout << shPtr1->shapeName() << endl;
147         cout << "use_count() = " << shPtr1.use_count() << endl;
148         {
149             shared_ptr<Polygon> shPtr2 = shPtr1;
150             cout << "use_count() = " << shPtr1.use_count() << endl;
151         }
152         cout << "use_count() = " << shPtr1.use_count() << endl;
153     }
154
155     // -------------- Issues with arrays -------------- //
156     cout << "------- Issues with arrays --------" << endl;
157
158     {
159         Polygon* polys = new Polygon[3]{Polygon(1,3), Polygon(1,4), Polygon(1,5)};
160
161         cout << polys[0].shapeName() << endl;
162         cout << polys[1].shapeName() << endl;
163         cout << polys[2].shapeName() << endl;
164
165         delete[] polys;
166     }
167
168
169
170
171
```

PROBLEMS | OUTPUT | TERMINAL | DEBUG CONSOLE

bash

```
------- Issues with arrays --------
Constructor Invoked
Constructor Invoked
Constructor Invoked
triangle
square
pentagon
Polygon destroyed
Polygon destroyed
Polygon destroyed
(base) Ivaldo:week6 admin$
```

master*+ | 0 0 | Live Share | Ln 175, Col 5 | Spaces: 4 | UTF-8 | LF | C++ | Mac

smart pointer always calls delete by default (and not delete[])

ISSUES WITH ARRAYS

```
156     cout << "------- Issues with arrays --------" << endl;
157
158     // {
159     //     Polygon* polys = new Polygon[3]{Polygon(1,3), Polygon(1,4), Polygon(1,5)};
160
161     //     cout << polys[0].shapeName() << endl;
162     //     cout << polys[1].shapeName() << endl;
163     //     cout << polys[2].shapeName() << endl;
164
165     //     delete[] polys;
166     // }
167
168     {
169         // Error: pointer being freed was not allocated
170         shared_ptr<Polygon> shPtr(new Polygon[3]{Polygon(1,3), Polygon(1,4), Polygon(1,5)});
171         // Soltuion:
172         // shared_ptr<Polygon> shPtr(new Polygon[3]{Polygon(1,3), Polygon(1,4), Polygon(1,5)},
173         // [](Polygon *p) { delete[] p; });
174
175         // cout << shPtr.get()[0].shapeName() << endl;
176         // cout << shPtr.get()[1].shapeName() << endl;
177         // cout << shPtr.get()[2].shapeName() << endl;
178     }
179
180
181
182
183
184
185
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

bash

```
------- Issues with arrays --------
Constructor Invoked
Constructor Invoked
Constructor Invoked
Polygon destroyed
main(70434,0x1176c85c0) malloc: *** error for object 0x7fdf03502c18: pointer being freed was not allocated
main(70434,0x1176c85c0) malloc: *** set a breakpoint in malloc_error_break to debug
Abort trap: 6
(base) Ivaldo:week6 admin$
```

master*+ 0 0 Live Share Ln 183, Col 1 Spaces: 4 UTF-8 LF C++ Mac

ISSUES WITH ARRAYS

main.cpp | Polygon.h | smartPtr.h

main.cpp > main()

```
156     cout << "------- Issues with arrays --------" << endl;
157
158     // {
159     //     Polygon* polys = new Polygon[3]{Polygon(1,3), Polygon(1,4), Polygon(1,5)};
160
161     //     cout << polys[0].shapeName() << endl;
162     //     cout << polys[1].shapeName() << endl;
163     //     cout << polys[2].shapeName() << endl;
164
165     //     delete[] polys;
166     // }
167
168     {
169         // Error: pointer being freed was not allocated
170         // shared_ptr<Polygon> shPtr(new Polygon[3]{Polygon(1,3), Polygon(1,4), Polygon(1,5)});
171         // Soltuion:
172         shared_ptr<Polygon> shPtr(new Polygon[3]{Polygon(1,3), Polygon(1,4), Polygon(1,5)},
173         [](Polygon *p) { delete[] p; });
174
175         cout << shPtr.get()[0].shapeName() << endl;
176         cout << shPtr.get()[1].shapeName() << endl;
177         cout << shPtr.get()[2].shapeName() << endl;
178     }
179
180
181
182
183
184
```

PROBLEMS | OUTPUT | TERMINAL | DEBUG CONSOLE

bash

```
------- Issues with arrays --------
Constructor Invoked
Constructor Invoked
Constructor Invoked
triangle
square
pentagon
Polygon destroyed
Polygon destroyed
Polygon destroyed
(base) Ivaldo:week6 admin$
```

master*+ | 0 0 | Live Share | Ln 184, Col 1 | Spaces: 4 | UTF-8 | LF | C++ | Mac

ISSUES WITH ARRAYS

main.cpp ● | Polygon.h | smartPtr.h

main.cpp > main()

```
168     // {
169     //     // Error: pointer being freed was not allocated
170     //     // shared_ptr<Polygon> shPtr(new Polygon[3]{Polygon(1,3), Polygon(1,4), Polygon(1,5)});
171     //     // Soltuion:
172     //     shared_ptr<Polygon> shPtr(new Polygon[3]{Polygon(1,3), Polygon(1,4), Polygon(1,5)},
173     //     [](Polygon *p) { delete[] p; });
174
175     //     cout << shPtr.get()[0].shapeName() << endl;
176     //     cout << shPtr.get()[1].shapeName() << endl;
177     //     cout << shPtr.get()[2].shapeName() << endl;
178     // }
179     {
180         std::shared_ptr<Polygon> shPtr(new Polygon[3]{Polygon(1,3), Polygon(1,4), Polygon(1,5)},
181         std::default_delete<Polygon[]>());
182         cout << shPtr.get()[0].shapeName() << endl;
183         cout << shPtr.get()[1].shapeName() << endl;
184         cout << shPtr.get()[2].shapeName() << endl;
185     }
186
187
188
189
190
191
192
193
194
195
196
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

bash

```
-------- Issues with arrays ---------
Constructor Invoked
Constructor Invoked
Constructor Invoked
triangle
square
pentagon
Polygon destroyed
Polygon destroyed
Polygon destroyed
(base) Ivaldo:week6 admin$
```

master*+ ⊗ 0 ⚠ 0 Live Share Ln 195, Col 5 Spaces: 4 UTF-8 LF C++ Mac

circular reference.

Circular dependencies are natural in many domain models where certain objects of the same domain depend on each other.[1]

Problems:

- Circular dependencies can cause many unwanted effects in software programs. Most problematic from a software design point of view is the tight coupling of the mutually dependent modules which reduces or makes impossible the separate re-use of a single module.

- Circular dependencies can cause a domino effect when a small local change in one module spreads into other modules and has unwanted global effects (program errors, compile errors). Circular dependencies can also result in infinite recursions or other unexpected failures.

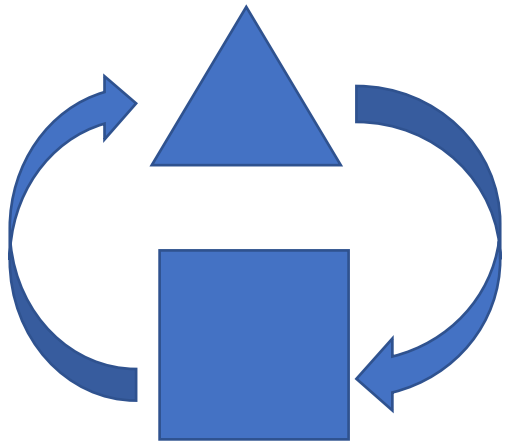
- Circular dependencies may also cause memory leaks by preventing certain very primitive automatic garbage collectors (those that use reference counting) from deallocating unused objects. [6]

SHARED POINTER

main.cpp ● | libraries.h | smartPtr.h

main.cpp > main()

```
186  //          cout << shPtr.get()[1].shapeName() << endl;
187  //          cout << shPtr.get()[2].shapeName() << endl;
188  //      }
189
190      // -------------- Circular reference -------------- //
191      cout << "----- Circular reference ------" << endl;
192      {
193          shared_ptr<Pair> triangle = make_shared<Pair>("triangle");
194          shared_ptr<Pair> square = make_shared<Pair>("square");
195
196          triangle->companion = square;
197          square->companion = triangle;
198          cout << square.owner_before(triangle) << endl;
199          cout << triangle.owner_before(square) << endl;
200      }
201
202
203
204
205
206
207  struct Pair
208  {
209      string name;
210      shared_ptr<Pair> companion;
211      // weak_ptr<Pair> companion;
212
213
214      Pair(string na) : name(na){};
215
216
217      ~Pair() {cout << name << " was deleted\n"; };
218  };
219
```



PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

bash

```
----- Circular reference ------
0
1
(base) Ivaldo:week6 admin$
```

master*+ ⊗ 0 ⚠ 0 Live Share Ln 211, Col 1 (2 selected) Spaces: 4 UTF-8 LF C++ Mac

WEAK POINTER

main.cpp | libraries.h | smartPtr.h

main.cpp > deleter > count_

```cpp
    // ------------- Circular reference ------------- //
    cout << "----- Circular reference ------" << endl;
    {
        shared_ptr<Pair> triangle = make_shared<Pair>("triangle");
        shared_ptr<Pair> square = make_shared<Pair>("square");

        triangle->companion = square;
        square->companion = triangle;
        cout << square.owner_before(triangle) << endl;
        cout << triangle.owner_before(square) << endl;
    }
```

```cpp
struct Pair
{
    string name;
    // shared_ptr<Pair> companion;
    weak_ptr<Pair> companion;

    Pair(string na) : name(na){};

    ~Pair() {cout << name << " was deleted\n"; };
};
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

bash

```
----- Circular reference ------
0
1
square was deleted
triangle was deleted
(base) Ivaldo:week6 admin$
```

master*+ 0 0 Live Share Ln 26, Col 16 Spaces: 4 UTF-8 LF C++ Mac

weak_ptr

- 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.

- A weak pointers can't access an object directly, but they can tell whether the object still exists or if it has expired. A weak pointer can be temporarily converted to a shared pointer to access the pointed-to object.

- A weak pointer lets you locate an object if it's still alive but doesn't keep it alive if nothing else needs it (doesn't extend an object's lifetime).

WEAK POINTER

main.cpp | libraries.h | smartPtr.h

main.cpp > main()

```
206
207     weak_ptr<Polygon> wPtr1;
208     weak_ptr<Polygon> wPtr2;
209     {
210         shared_ptr<Polygon> shPtr =  make_shared<Polygon>(3,7);
211         cout << "use_count() = " << shPtr.use_count() << endl;
212         wPtr1 = shPtr;
213         if(auto temp = wPtr1.lock()){
214             cout << (*temp).shapeName() << endl;
215         }
216         else
217             std::cout << "weak_ptr is expired\n";
218         wPtr2 = wPtr1;
219         if(auto temp = wPtr2.lock()){
220             cout << (*temp).shapeName() << endl;
221         }
222         else
223             std::cout << "weak_ptr is expired\n";
224
225         cout << "use_count() = " << shPtr.use_count() << endl;
226     }
227     if(auto temp = wPtr1.lock()){
228         cout << (*temp).shapeName() << endl;
229     }
230     else
231         std::cout << "weak_ptr is expired\n";
232
233     cout << "The polygon has already been destroyed."<< endl;
234
235
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

bash

```
-------- std::weak_ptrs --------
Constructor Invoked
use_count() = 1
heptagon
heptagon
use_count() = 1
Polygon destroyed
weak_ptr is expired
The polygon has already been destroyed.
(base) Ivaldo:week6 admin$
```

master*+ ⊗ 0 ⚠ 0 Live Share Ln 233, Col 5 Spaces: 4 UTF-8 LF C++ Mac

Performance Comparison

main.cpp | libraries.h | smartPtr.h

main.cpp > main()

```
215    //   -------- Performance Comparison --------
216    // *************************************************
217    cout << "------ Performance Comparison ------" << endl;
218
219    auto st = std::chrono::system_clock::now();
220
221    for(long long i = 0; i < pow(10,7) ; ++i ){
222        // int *test(new int(i));
223        // delete test;
224        // std::unique_ptr<int> tmp(new int(i));
225        // std::unique_ptr<int> tmp(std::make_unique<int>(i));
226        // std::shared_ptr<int> tmp(new int(i));
227        // std::shared_ptr<int> tmp(std::make_shared<int>(i));
228
229    }
230    std::chrono::duration<double> dur = std::chrono::system_clock::now() - st;
231    std::cout << "time: " << dur.count() << " seconds" << std::endl;
232
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

bash

```
------ Performance Comparison ------
time: 1.47332 seconds
(base) Ivaldo:week6 admin$ make
g++ -std=c++17 -Wall -o main.o -c main.cpp
g++ -std=c++17 -Wall -o main Polygon.o main.o
(base) Ivaldo:week6 admin$ ./main
------ Performance Comparison ------
time: 1.49897 seconds
(base) Ivaldo:week6 admin$ mke
bash: mke: command not found
(base) Ivaldo:week6 admin$ make
g++ -std=c++17 -Wall -o main.o -c main.cpp
g++ -std=c++17 -Wall -o main Polygon.o main.o
(base) Ivaldo:week6 admin$ ./main
------ Performance Comparison ------
time: 4.16201 seconds
(base) Ivaldo:week6 admin$ make
g++ -std=c++17 -Wall -o main.o -c main.cpp
g++ -std=c++17 -Wall -o main Polygon.o main.o
(base) Ivaldo:week6 admin$ .main
bash: .main: command not found
(base) Ivaldo:week6 admin$ ./main
------ Performance Comparison ------
time: 3.21839 seconds
(base) Ivaldo:week6 admin$
```

master*+ 0 0 Live Share Ln 213, Col 1 Spaces: 4 UTF-8 LF C++ Mac

Bibliography

1. Dzone (2022/02/02) Dangling Pointer in C++, https://dzone.com/articles/dangling-pointer-in-c
2. Wikipedia (2022/02/02) Smart pointer, https://en.wikipedia.org/wiki/Smart_pointer
3. Geeksforgeeks (2022/02/02) Smart Pointers in C++ and How to Use Them, https://www.geeksforgeeks.org/smart-pointers-cpp/
4. Twitter (2022/02/02) A beginner's look at smart pointers in modern C++, https://www.internalpointers.com/post/beginner-s-look-smart-pointers-modern-c
5. Stackoverflow (2022/02/02) Advantages of using std::make_unique, https://stackoverflow.com/questions/37514509/advantages-of-using-stdmake-unique-over-new-operator
6. Wikipedia (2022/02/02) Circular dependency, https://en.wikipedia.org/wiki/Circular_dependency
7. Microsoft (2022/02/02) How to: Create and Use shared_ptr instances, https://docs.microsoft.com/en-us/cpp/cpp/how-to-create-and-use-shared-ptr-instances?view=msvc-160
8. C++reference (2022/02/02) Undefined behavior, https://en.cppreference.com/w/cpp/language/ub

Appendix

- Libraries.h
- smartPtr.h
- Polygon.h
- Polygon.cpp
- main.cpp