

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

The screenshot shows a code editor interface with a dark theme. On the left, there's a vertical toolbar with various icons: a file icon, a search icon, a '10K+' icon, a play/pause icon, a copy/paste icon, a GitHub icon, and a refresh/circular arrow icon. The main area displays a file named 'main.cpp' with the following code:

```
// **** Memory leak ****
// ----- 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;
}
```

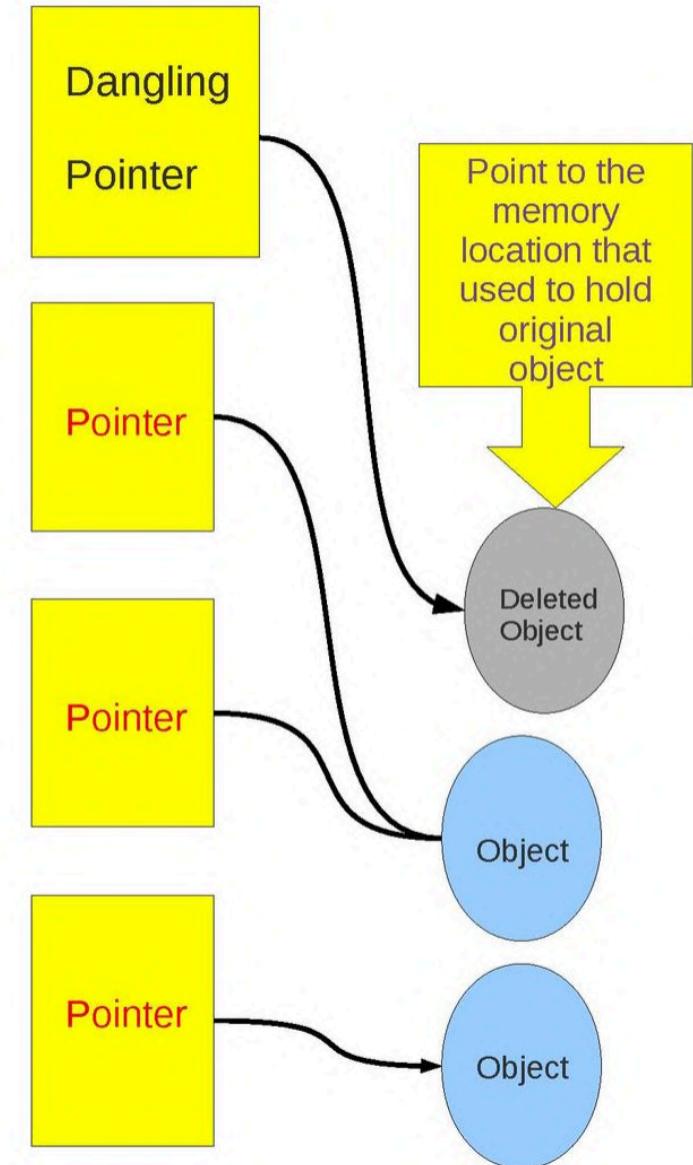
Below the code editor, there are tabs for 'PROBLEMS', 'OUTPUT', 'TERMINAL', and 'DEBUG CONSOLE'. The 'OUTPUT' tab is selected, showing the following terminal output:

```
----- 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
```

The status bar at the bottom indicates: 'Ln 50, Col 1 Spaces: 4 UTF-8 LF C++ Mac'. There are also icons for file operations like 'New', 'Open', 'Save', and 'Close'.

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**?



DANGLING POINTER

The screenshot shows a Microsoft Visual Studio Code (VS Code) interface. On the left is a dark sidebar with various icons for file operations, search, and project management. The main area displays a C++ file named `main.cpp`. The code illustrates a dangling pointer issue:

```
42     cout << p2->shapeName() << endl;
43     delete p1;
44 }
45
46 //***** Dangling pointer *****
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 //*****
```

The terminal below the editor shows the execution of the program:

```
----- 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 -----
```

The status bar at the bottom indicates the file is on master*, line 56, column 27, with 4 spaces, in UTF-8 encoding, and supports LF, C++, Mac, and Live Share.

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

```
template <class T>
class SmartPtr {

private:
    T* ptr;

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

    // Destructor
    ~SmartPtr();

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

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

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

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

```
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

The screenshot shows a Microsoft Visual Studio Code (VS Code) interface. On the left is a dark sidebar with various icons for file operations, search, and navigation. The main area displays a C++ file named `main.cpp`. The code demonstrates the use of smart pointers to handle dynamic memory. It includes comments explaining the creation of a dangling pointer and the use of a smart pointer to manage it safely.

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

Below the code editor is a navigation bar with tabs for PROBLEMS, OUTPUT, TERMINAL, and DEBUG CONSOLE. The TERMINAL tab is active, showing the output of the program's execution:

```
----- 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$
```

The status bar at the bottom indicates the current file is `main.cpp`, the line is 93, column is 5, and the encoding is UTF-8. It also shows options for LF, C++, Mac, and Live Share.

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_ptr`s 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_ptr`s 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

The screenshot shows a Visual Studio Code (VS Code) interface with the following details:

- File Explorer:** Shows files: main.cpp (1), Polygon.h, smartPtr.h.
- Sidebar:** Includes icons for search, file operations, and settings.
- Code Editor:** Displays the main.cpp file with code demonstrating std::unique_ptr. The code includes sections for std::shared_ptr and std::unique_ptr, with comments about incompatibility between them.
- Terminal:** Shows the output of the code execution:

```
----- std::unique_ptrs -----
0x7fc5be400640
23
0x7fc5be400640
23
(base) Ivaldo:week6 admin$
```
- Bottom Status Bar:** Shows file path (master*+), line count (0 △ 0), Live Share icon, and status bar with Ln 112, Col 5, Spaces: 4, UTF-8, LF, C++, Mac, and a few other icons.

std::move()

The screenshot shows the Microsoft Visual Studio Code interface. On the left is the sidebar with various icons. The main area has tabs for 'main.cpp' (active), 'Polygon.h', and 'smartPtr.h'. The code editor displays the following C++ code:

```
main.cpp:83: cout << p2->area() << endl;
main.cpp:84: }
main.cpp:85:
main.cpp:86: //***** std::unique_ptrs *****
main.cpp:87: // ----- std::unique_ptrs -----
main.cpp:88: //***** std::unique_ptrs *****
main.cpp:89: cout << "----- std::unique_ptrs -----" << endl;
main.cpp:90:
main.cpp:91: // not recommended unique_ptr<int> ptr1(new int(23));
main.cpp:92: unique_ptr<int> ptr1 = make_unique<int>(23);
main.cpp:93: cout << ptr1 << endl;
main.cpp:94: cout << *ptr1 << endl;
main.cpp:95:
main.cpp:96: int *ptr2;
main.cpp:97: // ptr2 = ptr1; incompatibility
main.cpp:98: ptr2 = ptr1.get(); // Returns the stored pointer
main.cpp:99: cout << ptr2 << endl;
main.cpp:100: cout << *ptr2 << endl;
main.cpp:101:
main.cpp:102: // std::unique_ptr does not have a copy constructor:
main.cpp:103: // unique_ptr<int> ptr3 = ptr1;
main.cpp:104: unique_ptr<int> ptr3 = move(ptr1); // memory resource is transferred to another unique_ptr
main.cpp:105: cout << ptr1 << endl;
main.cpp:106: cout << ptr2 << endl;
main.cpp:107: cout << ptr3 << endl;
main.cpp:108:
main.cpp:109:
main.cpp:110:
main.cpp:111:
main.cpp:112:
```

The terminal window at the bottom shows the output of the program:

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

A blue callout bubble points from the text 'memory resource is transferred to another unique_ptr' in the terminal output to the explanatory text below.

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.

 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 owned by the unique_ptr of another one.

Releases ownership of the object, returning its value as a *null pointer*.

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
```



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

EXPLORER ... Polygon.cpp main.cpp ourFunctions.h smartPtr.h Polygon.h

WEEK6
>.vscode
Assignment.txt
libraries
libraries.h
main
main.cpp
main.d
main.o
Makefile
ourFunctions.h
Polygon.cpp
Polygon.d
Polygon.h
Polygon.o
smartPtr.h
tempCodeRunnerFile

main.cpp > main()

```
78 cout << unPolyPtr->getNumberSides() << endl;
79
80 // Smart pointers will deallocate themselves if they leave the scope
81 {
82     unique_ptr<Polygon> ptr1 = make_unique<Polygon>(2,3);
83 }
84
85 }
86 //***** std::shared_ptrs *****
87 // ----- std::shared_ptrs -----
88 //*****
89 cout << "----- std::shared_ptrs -----" << endl;
90
91 {
92     shared_ptr<Polygon> shPtr1 = make_shared<Polygon>(2,5);
93     cout << shPtr1->shapeName() << endl;
94     cout << "use_count() = " << shPtr1.use_count() << endl;
95     {
96         shared_ptr<Polygon> shPtr2 = shPtr1;
97         cout << "use_count() = " << shPtr1.use_count() << endl;
98     }
99     cout << "use_count() = " << shPtr1.use_count() << endl;
100 }
101
102 // ----- Issues with arrays ----- //
103 cout << "----- Issues with arrays -----" << endl;
104
105 polyArray_on_heap();
106
107 {
108     // Error: pointer being freed was not allocated
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

----- std::shared_ptrs -----
Constructor Invoked
pentagon
use_count() = 1
use_count() = 2
use_count() = 1
Polygon destroyed
----- Issues with arrays -----

bash

> OUTLINE > TIMELINE

master*+ ⏪ ⏴ 0 ▲ 0 Ln 85, Col 6 Spaces: 4 UTF-8 LF C++ Mac 🔍 ⏺

ISSUES WITH ARRAYS

The screenshot shows a code editor interface with the following details:

- File Explorer:** Shows files: main.cpp, Polygon.h, smartPtr.h.
- Sidebar:** Includes icons for file operations, search, and a "10K+" badge.
- Code Editor:** Displays the main.cpp file with code related to shared pointers and arrays. The code includes shared pointer assignments, use count checks, and a section titled "Issues with arrays".
- Terminal:** Shows the output of the code execution, displaying the shape names (triangle, square, pentagon) and a message indicating the destruction of the polygons.
- Bottom Bar:** Includes tabs for PROBLEMS, OUTPUT, TERMINAL, DEBUG CONSOLE, and icons for bash, terminal, and file operations.
- Bottom Status Bar:** Shows the current file (master*+), line (Ln 175, Col 5), and encoding (UTF-8).

```
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 }
```

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

ISSUES WITH ARRAYS

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

```
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     // Solution:
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 }
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE



```
----- 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$
```

ISSUES WITH ARRAYS

The screenshot shows a Visual Studio Code (VS Code) interface with the following details:

- File Explorer (Left Sidebar):** Shows files: main.cpp, Polygon.h, smartPtr.h, and a folder named "10K+".
- Code Editor (Main Area):** Displays C++ code in `main.cpp`. The code demonstrates issues with arrays and shared pointers. It includes comments about pointer deallocation and shared_ptr usage.
- Terminal (Bottom):** Shows the output of the code execution. The output is:

```
----- Issues with arrays -----
Constructor Invoked
Constructor Invoked
Constructor Invoked
triangle
square
pentagon
Polygon destroyed
Polygon destroyed
Polygon destroyed
(base) Ivaldo:week6 admin$
```
- Bottom Status Bar:** Shows file path (master*+), line count (0 △ 0), and other status information.

ISSUES WITH ARRAYS

The screenshot shows a code editor interface with several tabs at the top: main.cpp (active), Polygon.h, smartPtr.h, and a few others. The main.cpp tab shows the following code:

```
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 //     // Solution:
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 }
```

The code uses both raw `new`/`delete[]` and `std::shared_ptr`/`std::default_delete` for managing the array of polygons. The terminal below shows the output of the program:

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

The status bar at the bottom indicates the file is master*, line 195, column 5, with 4 spaces, in UTF-8 encoding, and the keyboard layout is set to LF C++ Mac.



circular reference.

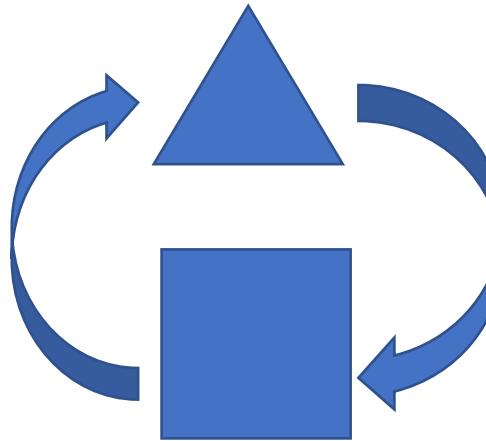
Circular dependencies are natural in many domain models where certain objects of the same domain depend on each other.^[1]

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 reuse 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 ● C libraries.h C smartPtr.h



```
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 + ^ X

----- Circular reference -----

9

1

(base) Ivaldo:week6 admin\$ █

WEAK POINTER

The screenshot shows a Visual Studio Code (VS Code) interface with the following details:

- File Explorer:** On the left sidebar, there are icons for files, search, file history, 10K+, and other development tools.
- Code Editor:** The main area displays code in `main.cpp`. A specific section of the code is highlighted with a blue border:

```
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     triangle->companion = square;
196     square->companion = triangle;
197     cout << square.owner_before(triangle) << endl;
198     cout << triangle.owner_before(square) << endl;
199 }
200
201
202
203
204
205
206 struct Pair
207 {
208     string name;
209     // shared_ptr<Pair> companion;
210     weak_ptr<Pair> companion;

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

212     ~Pair() {cout << name << " was deleted\n"; };
213 }
```
- Terminal:** At the bottom, the terminal window shows the output of the program:

```
----- Circular reference -----
0
1
square was deleted
triangle was deleted
(base) Ivaldo:week6 admin$
```
- Bottom Bar:** Includes tabs for PROBLEMS, OUTPUT, TERMINAL, and DEBUG CONSOLE, along with icons for bash, terminal, file, trash, and others.

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

The screenshot shows a Visual Studio Code (VS Code) interface with the following details:

- File Explorer:** On the left sidebar, there are icons for files, search, and other project-related functions.
- Code Editor:** The main area displays C++ code in `main.cpp`. The code demonstrates the use of `weak_ptr` and `shared_ptr` to manage a polygon object.
- Terminal:** At the bottom, the terminal window shows the execution of the program. The output is:

```
----- 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$
```
- Status Bar:** The bottom status bar shows the file path as `master*+`, line count as `0 △ 0`, and the status `Live Share`.
- Right Panel:** A large panel on the right side displays a detailed call stack or memory dump, likely from a debugger, showing numerous memory locations and their values.

Performance Comparison

The screenshot shows a dark-themed code editor interface with the following details:

- Top Bar:** Shows tabs for "main.cpp", "libraries.h", and "smartPtr.h".
- Sidebar:** On the left, there is a vertical toolbar with icons for file operations, search, and other development tools. A "10K+" badge is visible on one of the icons.
- Code Area:** The main content area displays C++ code for a performance comparison. The code includes comments for different memory management techniques: simple new/delete, std::unique_ptr, std::make_unique, std::shared_ptr, and std::make_shared. It measures the time taken for 10,000 iterations of each method.
- Terminal Output:** Below the code area, the "TERMINAL" tab is active, showing the command-line session where the code was run. The session includes compilation with g++, execution of the program, and three runs of the performance test. The results show times of approximately 1.47, 1.49, and 4.16 seconds respectively.
- Bottom Bar:** Shows navigation icons (back, forward, search) and status information: "master*+", "Live Share", "Ln 213, Col 1", "Spaces: 4", "UTF-8", "LF", "C++", "Mac", and a small icon.

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