SMART POINTERS

FUNDAMENTALS ON COMPUTING FOR ROBOTICS, GRAPHICS AND COMPUTER VISION

Darío Suárez - Adolfo Muñoz

DYNAMIC MEMORY ISSUES

- Each `new` always requires a matching `delete`
- Possible Issues:

1. Access an unitilized pointer
2. **Memory leaks**: unreachable objects in the heap
3. **Dangling pointers**: pointers to already released objects
4. **Multiple deletion**: calling delete several times for the same object

Universidad Zaragoza

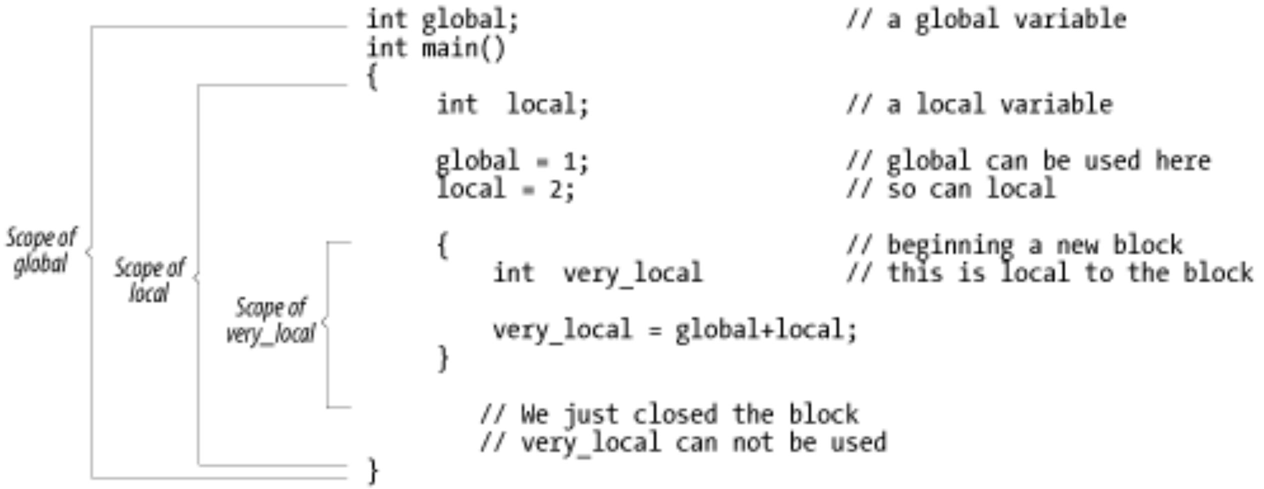
1542

POSSIBLE SOLUTION

- How we can automatically guarantee the release of the object?
- Think in objects in the stack and RAII (resource adquisition is initialization)

DETOUR: C++ SCOPES

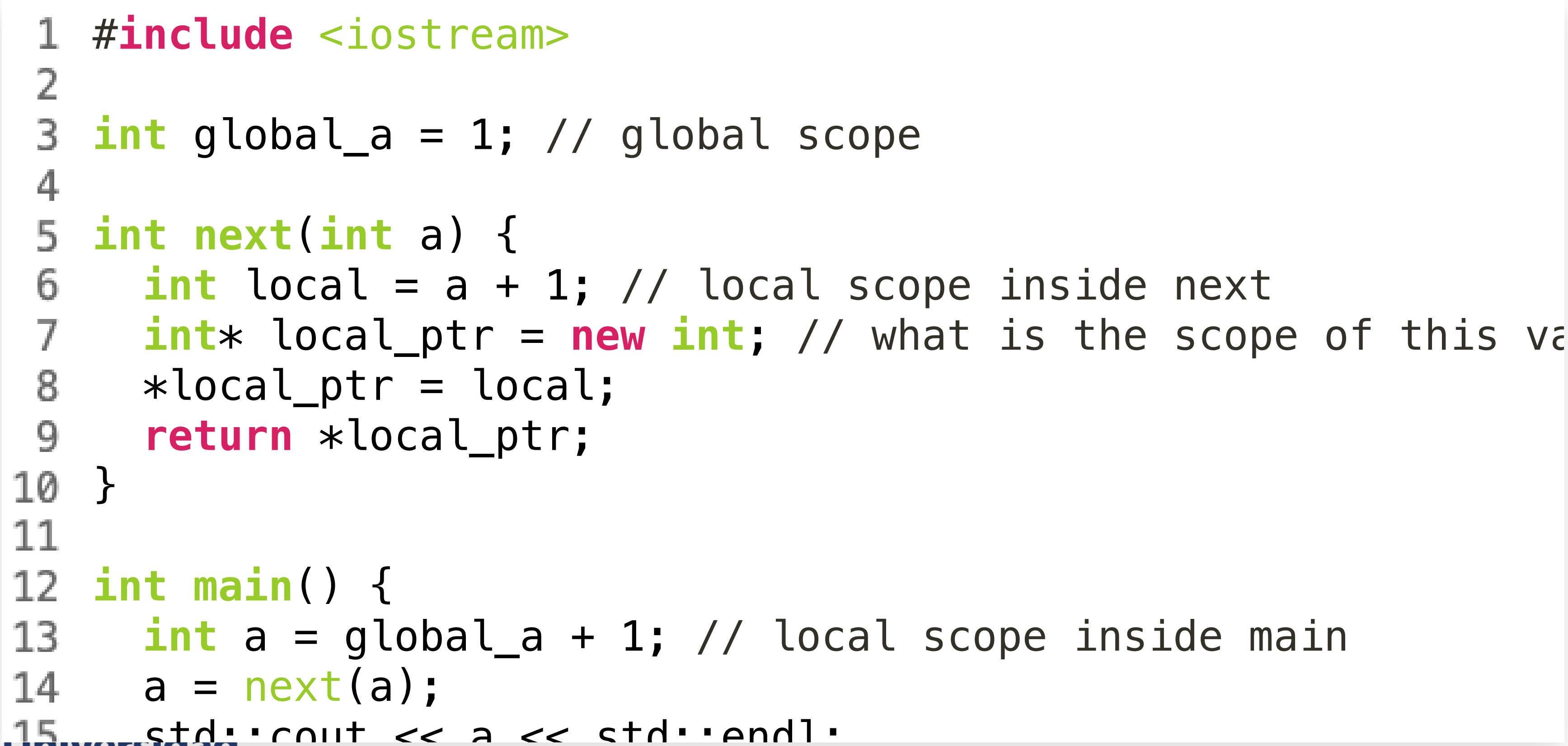
- **Scope**: context where an element (variable, object, class, function, ...) is visible



```
int global;                       // a global variable
int main()
{
     int  local;                  // a local variable

     global = 1;                  // global can be used here
     local = 2;                   // so can local

     {                            // beginning a new block
         int  very_local          // this is local to the block

         very_local = global+local;
     }

        // We just closed the block
        // very_local can not be used
}
```

source: https://www.oreilly.com/library/view/practical-c-programming/0596004192/ch09.html

QUIZ: WHAT ARE THE SCOPES WITHIN THIS PROGRAM

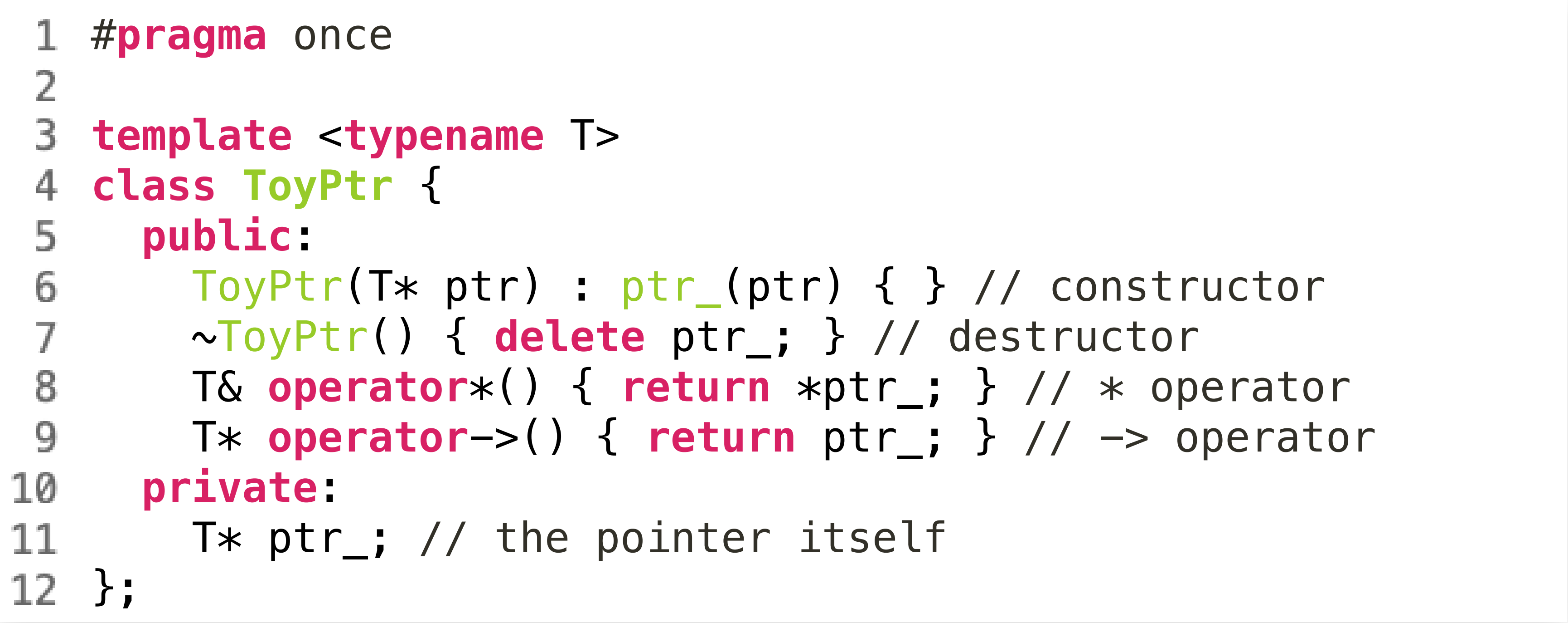


```cpp
#include <iostream>

int global_a = 1; // global scope

int next(int a) {
  int local = a + 1; // local scope inside next
  int* local_ptr = new int; // what is the scope of this va
  *local_ptr = local;
  return *local_ptr;
}

int main() {
  int a = global_a + 1; // local scope inside main
  a = next(a);
  std::cout << a << std::endl;
```

SMART POINTERS

- Ensure that objects are released after the last use; e.g., going out of scope
- Automatically call `delete`
- `std::unique_ptr` and `std::shared_ptr` templated classes release the object in their destructors
- Enable same access as regular pointers: `*` and `->`

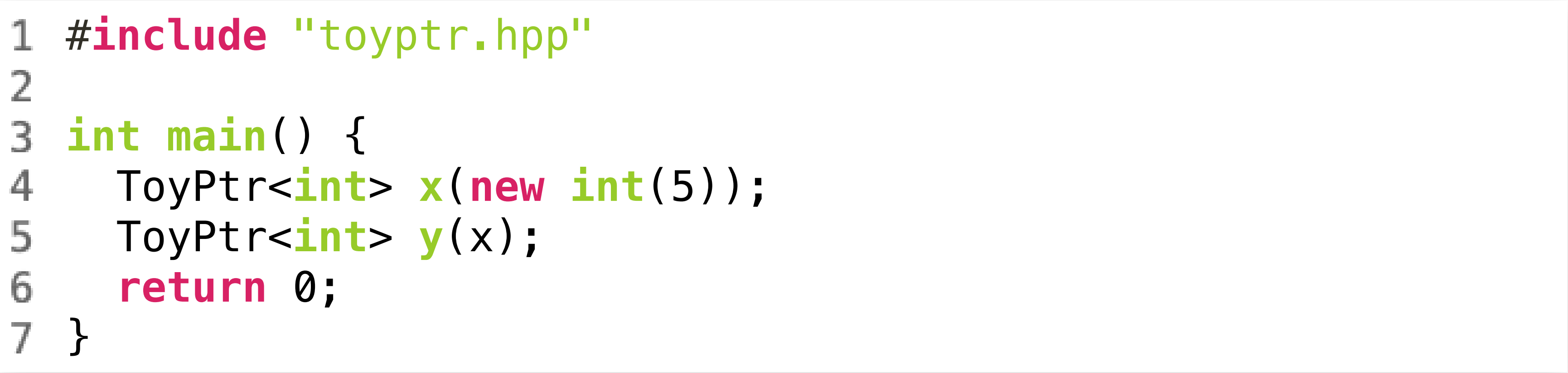
RATIONALE EXAMPLE



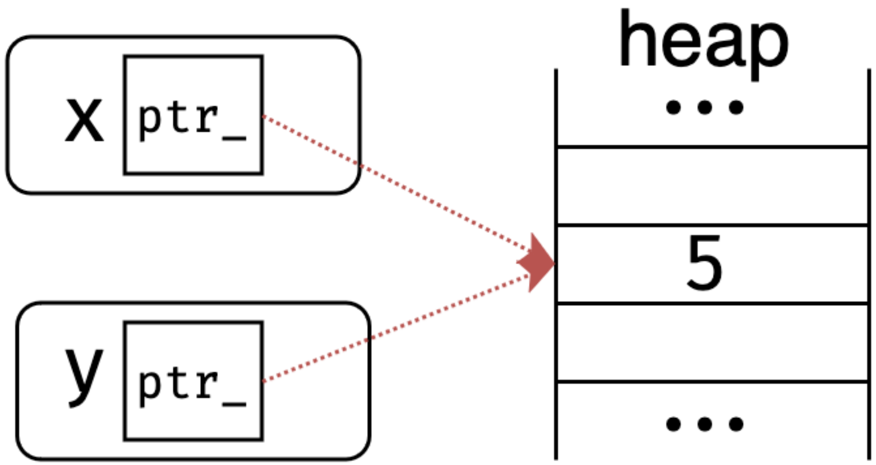
```cpp
#pragma once

template <typename T>
class ToyPtr {
  public:
    ToyPtr(T* ptr) : ptr_(ptr) { } // constructor
    ~ToyPtr() { delete ptr_; } // destructor
    T& operator*() { return *ptr_; } // * operator
    T* operator->() { return ptr_; } // -> operator
  private:
    T* ptr_; // the pointer itself
};
```

source: https://courses.cs.washington.edu/courses/cse333/22wi/lectures/16/16-smartptrs_22wi_ink.pdf

USE OF ToyPtr



```cpp
#include "toyptr.hpp"

int main() {
  ToyPtr<int> x(new int(5));
  ToyPtr<int> y(x);
  return 0;
}
```



How many times is called `delete`?

`std::unique_ptr<T>`

- Solution for single ownership of an object through a pointer
- Dealocates when going out of scope
- Cannot be copied and can be moved
- Use std::make_unique to create them

std::shared_ptr<T>

- Solution for multiple pointers to the same object
- Internally counts the number of refereces to the object
- Allocation increases the counter, deallocation reduces the counter. When reaching 0, the object is deallocated
- Use std::make_shared to create them

TO LEARN MORE

- R.20 to R.33 from Cpp Core Guidelines
- What are Smart Pointers