Ch 4: The Object Life Cycle

CSCI 330

Overview

- Lifetime of objects: automatic, dynamic, static, and thread storage duration
- Construction and destruction order
- Copy/move constructors and assignment operators
- Resource Acquisition Is Initialization) RAII
- Object slicing (and how to avoid it)
- std::unique_ptr and std::shared_ptr

Objects

- An object = region of storage with a type and a value
- A variable is a named object
- Each object has a lifetime and storage duration

Object Lifecycle Stages

- Storage allocated
- Constructor called → Lifetime begins
- Object is used
- Destructor called → Lifetime ends
- Storage deallocated

Storage Duration Categories

- Automatic (local variables): function scope
- Static (global/static): program lifetime
- Dynamic (via new/delete): manual control
- Thread-local: lasts for the lifetime of the thread

Example: Automatic Duration

Scope of local variables:

- Recreated every function call.
- Cannot be accessed outside of function

Example:

- Function:
  - power_up_rat_thing (line 5)
- Local variables:
  - nuclear_isotopes (line 5)
    waste_heat (line 7)

```
1  #include <cstdio>
2
3  static int rat_things_power = 200;
4
5  void power_up_rat_thing(int nuclear_isotopes){
6      rat_things_power = rat_things_power + nuclear_isotopes;
7      const auto waste_heat = rat_things_power * 20;
8      if(waste_heat > 1000){
9          printf("Warning! Hot doggie!\n");
10     }
11 }
12
13 int main(){
14     printf("Rat-thing power: %d\n", rat_things_power);
15     power_up_rat_thing(100);
16     printf("Rat-thing power: %d\n", rat_things_power);
17     power_up_rat_thing(500);
18     printf("Rat-thing power: %d\n", rat_things_power);
19
20 }
```

Example: Static Duration

Initialed once w/ keyword

- static: w/in file
- extern between files

Global Scope: Same object across function calls, initialized outside function.

Example

- rat_thing_power (line 3)

```
1  #include <cstdio>
2
3  static int rat_things_power = 200;
4
5  void power_up_rat_thing(int nuclear_isotopes){
6      rat_things_power = rat_things_power + nuclear_isotopes;
7      const auto waste_heat = rat_things_power * 20;
8      if(waste_heat > 1000){
9          printf("Warning! Hot doggie!\n");
10     }
11 }
12
13 int main(){
14     printf("Rat-thing power: %d\n", rat_things_power);
15     power_up_rat_thing(100);
16     printf("Rat-thing power: %d\n", rat_things_power);
17     power_up_rat_thing(500);
18     printf("Rat-thing power: %d\n", rat_things_power);
19
20 }
```

Example: Dynamic Duration

Allocated Object are manually allocated and deallocated on request (manual control)

- Must call delete (avoid memory leaks
- Forgetting delete causes memory leaks

Global Scope: Same object across function calls, initialized outside function.

Example

- allocation (line 9)
- deallocation (line 24)

```cpp
1  #include <iostream>
2
3  int main(){
4      int size;
5      std::cout<<"enter a number of elements: ";
6      std::cin >> size;
7
8      //Dynamic allocation of array of integers
9      int* numbers = new int[size];
10
11     //Populate the array
12     for (int i = 0; i < size; ++i) {
13         numbers[i] = i * 2; //generic data
14     }
15
16     //print array
17     std::cout <<"Array contents:\n";
18     for (int i = 0; i < size; ++i){
19         std::cout << numbers[i] << "  ";
20     }
21     std::cout << "\n"; //Extra line
22
23     //Deallocate the member
24     delete[] numbers;
25     return 0;
26 }
```

OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTGRESQL

```
davideve@MacBook-Pro-2024 ~/DevEnv/cpp/ch4
% ./dynamicAllocationExample
enter a number of elements: 11
Array contents:
0  2  4  6  8  10  12  14  16  18  20
davideve@MacBook-Pro-2024 ~/DevEnv/cpp/ch4
%
```

What’s a Memory Leak?

Memory leaks occur when:

- You allocate memory dynamically (new, new[])
- but you don’t release it with delete or delete[]
- The program loses the ability to access the memory because it’s still reserved in the heap, but it can’t be reclaimed or reused.

Consequences:

- Increased memory usage (leaked memory accumulates, increasing the app’s footprint.
- System slows down as memory fills up
- Resource exhaustion – running out of resources

Best Practice:

- Always pair a new with delete, new[] with delete[], or
- Use smart pointers: std::vector (back to this later)

Memory Management in C++

- C++: does not use automatic garbage collection
- Programmer is responsible for:
  - Allocating memory (e.g., new, new[])
  - Deallocating memory (e.g., delete, delete[])
- Errors like double delete, dangling pointers, and memory leaks are possible
- Prefer smart pointers and containers to reduce risk

Smart Pointers

Why Smart Pointers?

- Manual memory management (new/delete) is error prone
- Smart pointers help manage dynamic memory safely and automatically
- Integrate with RAII to prevent leaks and dangling pointers

Types of smart pointer:

- std::unique_ptr<T> : sole ownership
- std::shared_ptr<T>: shared ownership, reference counted
- Automatically deleted when out of scope

Resource Acquisition is Initialization (RAII)

- C++ idiom: manage resources via object lifetime
- Constructor acquires resource
- Destructor releases
- Example:
    - std::fstream
    - std::mutex
    - std::lock_guard

Object Slicing

- Happens when passing derived object by value to base class
- Avoid slicing:
  - pass by reference, or
  - use smart pointers

Constructors and Destructors

- Constructor: initializes object
- Destructor: cleans up before object is destroyed
- Special functions:
  - Copy constructor
  - Move constructor
  - Copy assigned operator
  - Move assigned operator

Exception safety and Object Lifetimes

- C++ does not use garbage collection—resources must still be released even when exceptions are thrown
- RAII ensures destructors are called automatically when exceptions unwind the stack
- Smart pointers and RAII-wrapped resources prevent leaks during exceptions

Try/Catch Blocks

- try defines the protected block
- catch handles specific exceptions
- Always catch by reference to avoid slicing

std::exception

All standard exceptions derive from std::exception

Common exception subclasses:

- std::logic_error: violations of logical preconditions (detected before runtime)
- std::runtime_error: Errors detected during program execution

Logic_error Derivatives

- std::invalid_argument : bad input argument
- std::domain_error : Input outside function’s domain
- std::length_error : Exceeding container maximum size
- std::out_of_range : Accessing element outside bounds

runtime_error Derivatives

- std::overflow_error : Arithmetic overflow
- std::underflow : Arithmetic underflow
- std::range_error : Numerical result out of range
- std::ios_base::failure : I/O operation failure

Call Stacks and Exceptions

Call Stack:

- A data structure that tracks active function calls in a program
- Each function call adds a stack frame
- Stack frame contains:
  - Return address
  - Function parameters
  - Local variables

How exceptions affect the call stack:

- When exception is thrown:
  - Stack unwnding begins
  - Destructors for all in=scope local objects are called
  - Control transfers to the nearest matching catch block
  - If no catch is found, program calls std::terminate()

Best practices

- Ensure all heap-allocated resources are managed via RAII
- Never leak resources during stack unwinding
- Avoid raw (read: uncontrolled) new in exception-prone code path

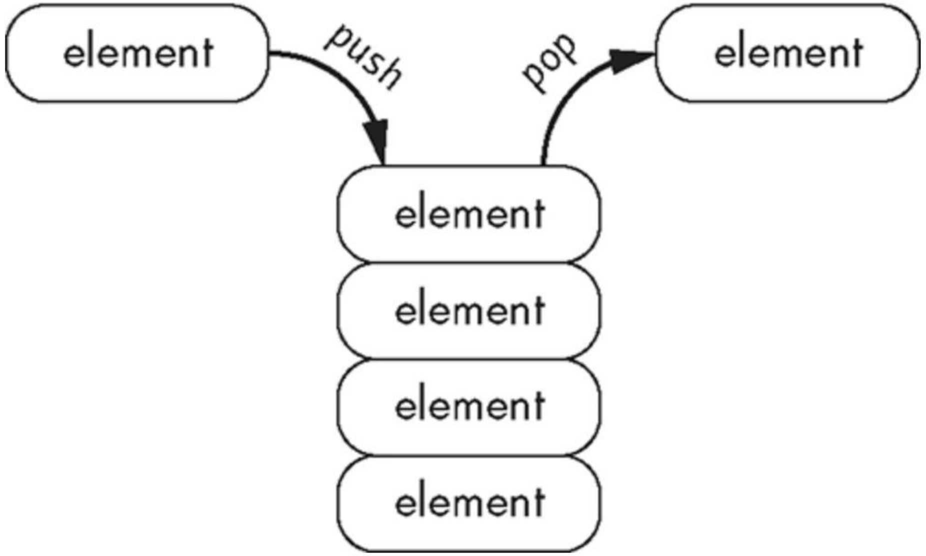

Figure 4-2: Elements being pushed onto and popped off of a stack

Exception Best Practices

- Throw exceptions by **value**, catch by **const reference**
- Never throw from destructors
- Ensure dynamically allocated memory is managed by RAII
- Avoid resource leaks by using smart pointers
- define noexcept for functions that never throw