Object Oriented Programming - Lecture 8

Diana Borza - diana.borza@ubbcluj.ro

April 9, 2024

Content

- Move semnatics
- RAII
- Smart pointers
- threads

L-values and R-values I

Each C++ expression (an operator with its operands, a literal, a variable name, etc.) is characterized by two independent properties: *a type* and *a value category*. In C++ there are several types of *value categories*:

1. l-values - so-called, historically, because lvalues could appear on the left-hand side of an assignment expression;
2. r-values - so-called, historically, because rvalues could appear on the right-hand side of an assignment expression;
3. gl-value - "generalized" lvalue;
4. pr-value - "pure" rvalue;
5. xvalue - an "eXpiring" value

In this course, we'll just cover l-values and r-values. For further information, you can check: https://en.cppreference.com/w/cpp/language/value_category

L-values and R-values II

- l-values
  - "values that are suitable to be on the left-hand side of an assignment expression";
  - all l-values have assigned memory addresses;
  - a function or an object (or an expression that evaluates to a function or object).
- r-values
  - "everything that is not an l-value";
  - e.g. literals (10, 'a' etc.), temporary values (x+3), anonymous objects (Student"John Smith")
  - r-values are typically evaluated for their values
  - r-values have expression scope (they die at the end of the expression they are in)
  - typically r-values cannot be assigned to.

Select the l-values and the r-values from the following code snippet:

```
int a = 2;
int b = 3;
int sum = a + b;
MyClass cl = MyClass(5, 7);
int d = myFunction();
```

```cpp
int a = 2; // a is an lvalue, 2 is an rvalue
int b = 3; // b is an lvalue, 3 is an rvalue
int sum = a + b; // sum is an l values, a+b is an rvalue
MyClass cl = MyClass(5, 7); // cl is an lvalue, MyClass(5, 7) is an rvalue (anonymous  object)
int d = myFunction(); // d is an lvalue
                   // the temporary value returned by myFunction()
                   // (a function which returns an int ) is an rvalue
```

R-value references

- Since C++11 it is possible to have a reference for an r-value.
- An r-value reference is a reference that is designed to be initialized with an r-value (**only**).
- While an l-value reference is created using a single ampersand, an r-value reference is created using a double ampersand:
  - int x{5}; int &xRef{x}; // xRef is a l-value reference
  - int &&rRef {5}; // rref is a r-value reference
- r-value references **extend the lifespan** of the object they are initialized with to the lifespan of the r-value reference;
- (non-const) r-value references allow you to modify the r-value.

L-values references and R-values references

- L-value references

L-value reference	Can be initialized with	Can modify
Modifiable l-values	Yes	Yes
Non-modifiable l-values	No	No
R-values	No	No

- L-value references to const

L-value reference to const	Can be initialized with	Can modify
Modifiable l-values	Yes	No
Non-modifiable l-values	Yes	No
R-values	Yes	No

L-values references and R-values references

- R-value references

R-value reference	Can be initialized with	Can modify
Modifiable l-values	No	No
Non-modifiable l-values	No	No
R-values	Yes	Yes

- R-value references to const

R-value reference to const	Can be initialized with	Can modify
Modifiable l-values	No	No
Non-modifiable l-values	No	No
R-values	Yes	No

Move semantics - idea

- If we construct an object or do an assignment where the argument is an **l-value**, the only thing we can reasonably do is **copy** the l-value.
  - We cannot alter the l-value as it may be used again in the program.
- If we construct an object or do an assignment where the argument is an **r-value**, then we know that r-value is just a temporary object of some kind.
  - Instead of copying it (which can be expensive), we can simply **transfer its resources** (which is cheap) to the object we're constructing or assigning.
  - the temporary will be destroyed at the end of the expression anyway, so we know it will never be used again.

Move constructor and move assignment

- The move constructor and move assignment are called when those functions have been defined, and **the argument for construction or assignment is an r-value** (literal or temporary value).
- A move constructor and move assignment operator will NOT be provided by default.

Move constructor

- Whereas the goal of the copy constructor and assignment op. is to make a copy of one object to another, the goal of the move constructor and assignment op. is to **move ownership** of the resources from one object to another (which is much less expensive than making a copy).
- syntax: class_name(class_name &&);
- Disabling copy constructor: use delete keyword:
    - If, instead of a function body, the special syntax =delete ; is used, the function is defined as *deleted*.
    - Any use of a deleted function is ill-formed (the program will not compile).

Move assignment

- Move assignment operators typically "**steal**" the resources held by the argument (e.g. pointers to dynamically-allocated objects, file descriptors, TCP sockets, I/O streams, running threads, etc.), rather than make copies of them;
- they leave the argument in some valid but otherwise indeterminate state.
- syntax:
- syntax: class_name & class_name :: operator=(class_name &&);

Demo

Move constructor and move assignment.

Rule of five I

- The rule of five is a modern expansion of the rule of three.
- Remember the rule of three?

Rule of five II

- The rule of three specifies that if a class implements any of the following functions, it should implement all of them:
  - copy constructor;
  - copy assignment operator;
  - destructor.
- **In addition**, the rule of five identifies that it is usually appropriate to also provide the following functions to allow for optimized copies from temporary objects:
  - move constructor;
  - move assignment operator.

std::move

- std::move is a library function to convert its argument into an r-value.
- We pass an l-value to std::move, and it will return an r-value reference.
- std::move is defined in the <utility> header.

RAII

- RAII - *Resource Acquisition Is Initialization* - is a programming technique in which a resource is tied to the lifetime of objects which acquires it.
- A resource can be memory location, a database connection, a file, a network socket etc.
- The resource is *acquired* (allocated) during object creation (specifically *initialization*), by the constructor, while resource *deallocation* (release) is done during object destruction (specifically *finalization*), by the destructor.
  - *Constructor Acquires, Destructor Releases (CADRe)*
- RAII is used to **avoid resource leaks** and to write **exception-safe code**.

RAII

- Advantages over garbage collection (from other programming languages):
  - RAII is a programming idiom which offers automatic management for different kinds of resources, not just memory.
  - The runtime environment is faster, as there is no separate mechanism involved (like the garbage collector).

RAII in STL

- You already used RAII in your programs:
  1. Files in C++:
     - When using an object of type ifstream or ofstream, the constructor will automatically open the file.
     - When the object gets destroyed, the destructor automatically closes the file.
  2. The STL containers manage memory using the RAII programming technique.

Implementing RAII - I

- The lifetime of the objects allocated on the stack is automatically managed by the compiler
  - The compiler automatically invokes constructors to initialize objects;
  - The compiler automatically calls the destructor when the object goes out of scope.

1. Create a wrapper for your resource, based on the following rule: allocate the resource in constructor, release it in the destructor.
2. Use the resource (directly) wherever you need it.
3. The resource will be automatically deallocated when the wrapper's scope is left.

Demo

RAII - smart pointer

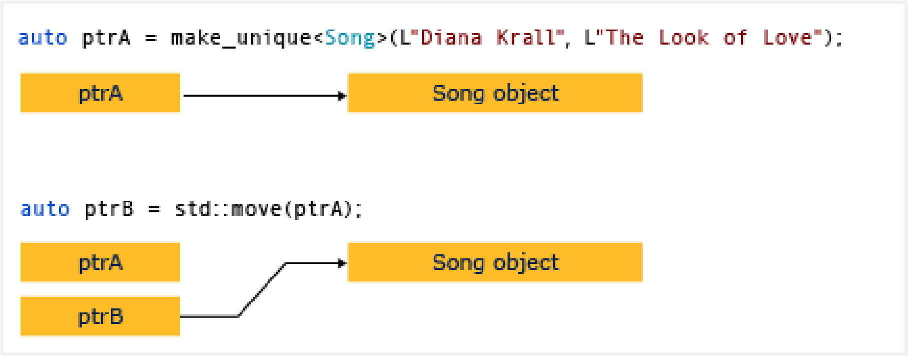
Smart pointers in STL

- A smart pointer is a composition class that is designed to manage dynamically allocated memory, and ensure that memory gets deleted when the smart pointer object goes out of scope.
- Smart pointers should never be dynamically allocated themselves. Why?
- Smart pointers are defined in the <memory> header.
- In STL there are 3 types of smart pointers:
  - std::unique_ptr;
  - std::shared_ptr;
  - std::weak_ptr.

std::unique_ptr I

- It is used to manage any dynamically allocated object that is not shared by multiple objects.
- It retains *exclusive ownership* of the object, it does not share the object.
- A std::unique_ptr **cannot be**:
  - copied to another unique_ptr;
  - passed by value to a function;
  - used in any STL algorithm that requires copies to be made.
- A std::unique_ptr **can only** be **moved**: std::move().

std::unique_ptr II



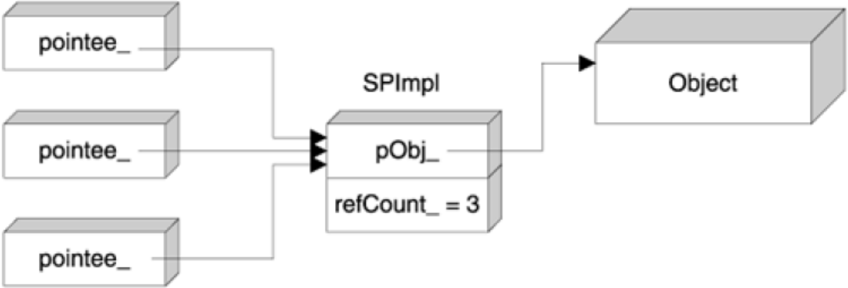
std::unique_ptr III

- std::unique_ptr provides two overloaded operators to access the raw pointer: operator* (reference to the resource) and operator$->$ (pointer to the reference).
- checking that std::unique_ptr actually holds a resource: std::unique_ptr has a cast to bool that returns true if the std::unique_ptr is managing a resource.
- It is recommended that std::unique_ptr are created via the std::make_unique() function.
- To pass std::unique_ptr to a function you should use std::move() $\rightarrow$ **!!!** the function will take ownership of the object. Alternative: pass the resource itself to the function.

std::shared_ptr

- std::shared_ptr was developed for the cases where you need multiple smart pointers *co-owning* a resource.
- Internally, std::shared_ptr uses **reference counting**. Reference counting keeps track of the **number of smart pointers** pointing to (or owning) the same object.
- If at least one std::shared_ptr is pointing to the resource (i.e. the reference number is greater than 0), the resource will not be deallocated.
- When the reference number becomes zero, the pointee object is deleted.

std::shared_ptr



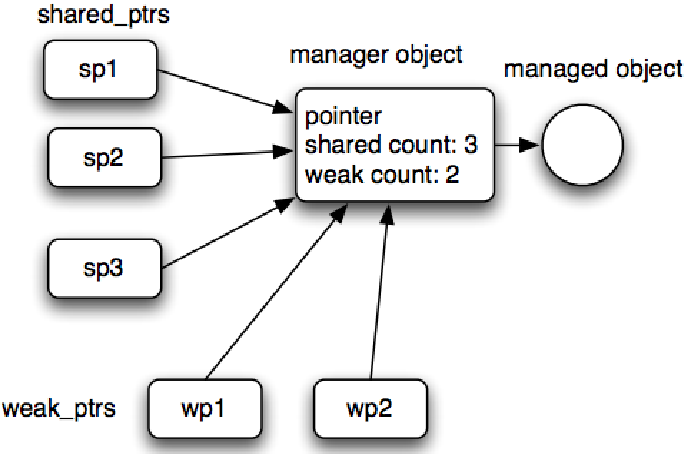
std::shared_ptr

- std::shared_ptr can be copied and moved (move transfers ownership).
- has more overhead than unique ptr (because of the internal reference counting), therefore, whenever possible, prefer unique_ptr.
- it should be constructed with the std::make_shared function.
- can be created from a std::unique_ptr, but not the otherway around.
- Problems with shared pointers: **cyclic dependencies**.
- A circular reference (cyclical reference, cyclical dependency) refers to a series of references where each object references the next, and the last object references back to the first, causing a *referential loop*.

std::weak_ptr

- std::weak_ptr was designed to solve the cyclical dependency problem which can occur when using shared pointer.
- Can be seen like an observer – it can observe and access the same object as a std::shared_ptr, but **it is not considered an owner**.
- Used to access the underlying object of a std::shared_ptr without causing the reference count to be incremented.

std::weak_ptr


shared_ptrs
sp1
sp2
sp3
manager object
pointer
shared count: 3
weak count: 2
managed object
weak_ptrs
wp1
wp2

std::weak_ptr

- You are using two classes to capture the information about a soccer team. You have 2 classes *Team* and *Member*.
  - A team has pointers to its members.
  - Each member can have a pointer to the team it belongs to.
  - If all pointers (to members and to team) are std::shared_ptr, what happens when the team goes out of scope? → memory leak.
  - Therefore, the members should have a weak pointer to their team.

Smart pointers - best practices

"Smart developers use smart pointers." - Kate Gregory

"In modern C++, raw pointers are only used in small code blocks of limited scope, loops, or helper functions where performance is critical and there is no chance of confusion about ownership." - https://docs.microsoft.com/en-us/cpp/cpp/smart-pointers-modern-cpp?redirectedfrom=MSDN&view=vs-2019

Smart pointers - advantages

- Smart pointers increase productivity and improve the robustness of the program.
- The programmer does not need to be concerned with memory management (provided the smart pointers are used correctly).
- Avoid memory leaks.
- Write exception safe code.

Threads

Processes

- **Program** is the code that is stored on your computer that is intended to fulfill a certain task; stored on disk or in non-volatile memory in a form that can be executed by your computer (machine language).
- A **process** is a program that has been loaded into memory along with all the resources it needs to operate.
- Each process has a separate memory address space (a process runs independently and is isolated from other processes).
- Switching from one process to another is relatively slow: saving and loading registers, memory maps, and other resources.

Resources of a process

- “registers” - data storage locations which are part of the computer processor (CPU).
- ”program counter”/“instruction pointer” - keeps track of the current location of the program sequence.
- stack? heap?

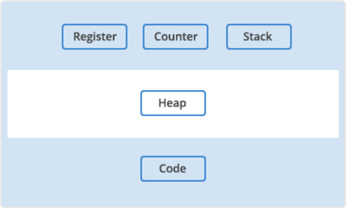


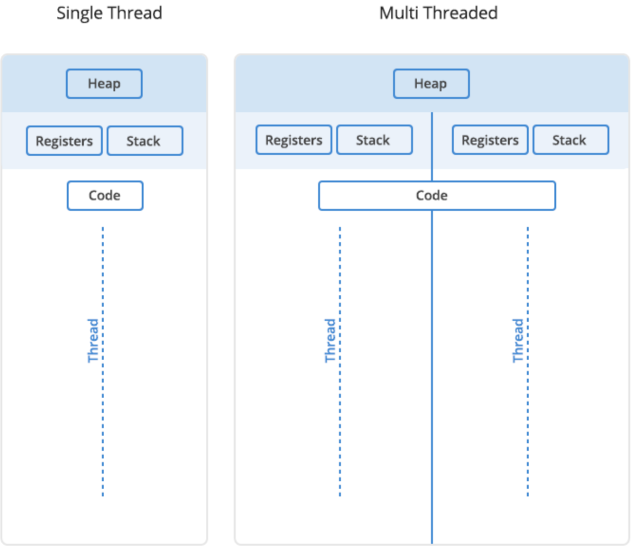
Figure source:

https://www.backblaze.com/blog/whats-the-diff-programs-processes-and-threads/

Thread

- A thread is the unit of execution within a process.
- A process can have anywhere from just one thread (single-threaded app.) to many threads (multi-threaded app.).
- When a process starts, it is assigned memory zone and resources.
- Each thread in the process shares that memory and resources.
- Each **thread will have its own stack**, but all the threads in a process will **share the heap**.
- The disadvantage is that a problem with one thread in a process will certainly affect other threads and the viability of the process itself.

Threads vs. Processes I



Threads vs. Processes II

Threads

- Threads use the memory of the process they belong to;
- Inter-thread communication can be faster than IPC (threads share memory with the process they belong to);
- Context switching between threads of the same process is less expensive;
- Threads are lighter weight operations.

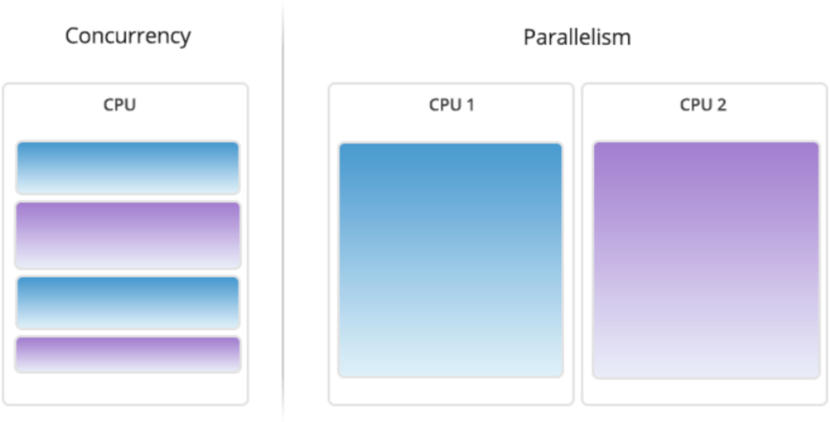
Processes

- Each process has its own memory space;
- Inter-process communication is slow as processes have different memory addresses;
- Context switching between processes is more expensive;
- Processes don't share memory with other processes;
- Processes are heavyweight operations.

Parallelism and concurrency I

- On a system with multiple processors or CPU cores, multiple processes or threads can be executed in parallel.
- On a single processor, though, it is not possible to have processes or threads truly executing at the same time.
  - the CPU is shared among running processes or threads using a process scheduling algorithm that divides the CPU's time and yields the **illusion** of parallel execution;
  - the time given to each task is called a *time slice*.
- **Parallelism** - genuine simultaneous execution.
- **Concurrency** - interleaving of processes in time to give the appearance of simultaneous execution.

Parallelism and concurrency II



Threads in C++

- std::thread - class to represent individual threads of execution.
- declared in the header <thread> .
- The constructor of std::thread can take as parameters:
  - a pointer to function that will be executed by the thread;
  - arguments passed to the call of this function (optional).
- The join() function from the std::thread class returns when the thread execution has completed.
  - This blocks the execution of **the thread that calls this function** until the function called on construction returns (if it hasn't yet).

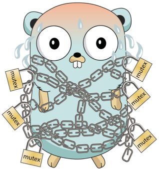
Thread synchronization I

- Synchronization - refers to one of the following concepts:
  - **Process synchronization** refers to the idea that multiple processes are to join up or handshake at a certain point, in order to reach an agreement or commit to a certain sequence of action.
  - **Data synchronization** refers to the idea of keeping multiple copies of a dataset in coherence with one another, or to maintain data integrity.

Thread synchronization II

Thread synchronization III

- **Mutual exclusion** algorithms prevent multiple threads from simultaneously accessing shared resources. This prevents data races and provides support for synchronization between threads.
- `https://en.cppreference.com/w/cpp/thread/mutex`
- `https://en.cppreference.com/w/cpp/thread/lock_guard` - a mutex wrapper that provides a convenient RAII-style mechanism for owning a mutex for the duration of a scoped block.



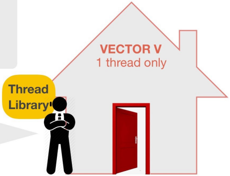
Mutex idea

`door.lock()`

THREAD
A

Good morning **Mr. ThreadLibrary**!
Can I visit Vector V? I need to give him something.

Yes, no one is in there. Remember to **unlock the door** and leave as soon as you can! Other people want to see him.



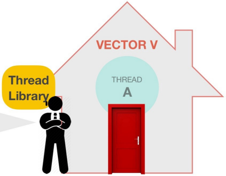
after some nanoseconds . . .

`door.lock()`

THREAD
B

Good morning **Mr. ThreadLibrary**!
Can I see Vector V?

Sorry, the **door is locked**. Wait here until the other guy leaves.



Other synchronization mechanisms

- Other synchronization mechanisms (not a subject of this lecture):
  - semaphores: `https://en.cppreference.com/w/cpp/thread/counting_semaphore`
  - condition variables: `https://en.cppreference.com/w/cpp/thread/condition_variable`.