The LOVE-HATE Relationships in Modern C++

Ankur M. Satle

https://ankursatle.wordpress.com

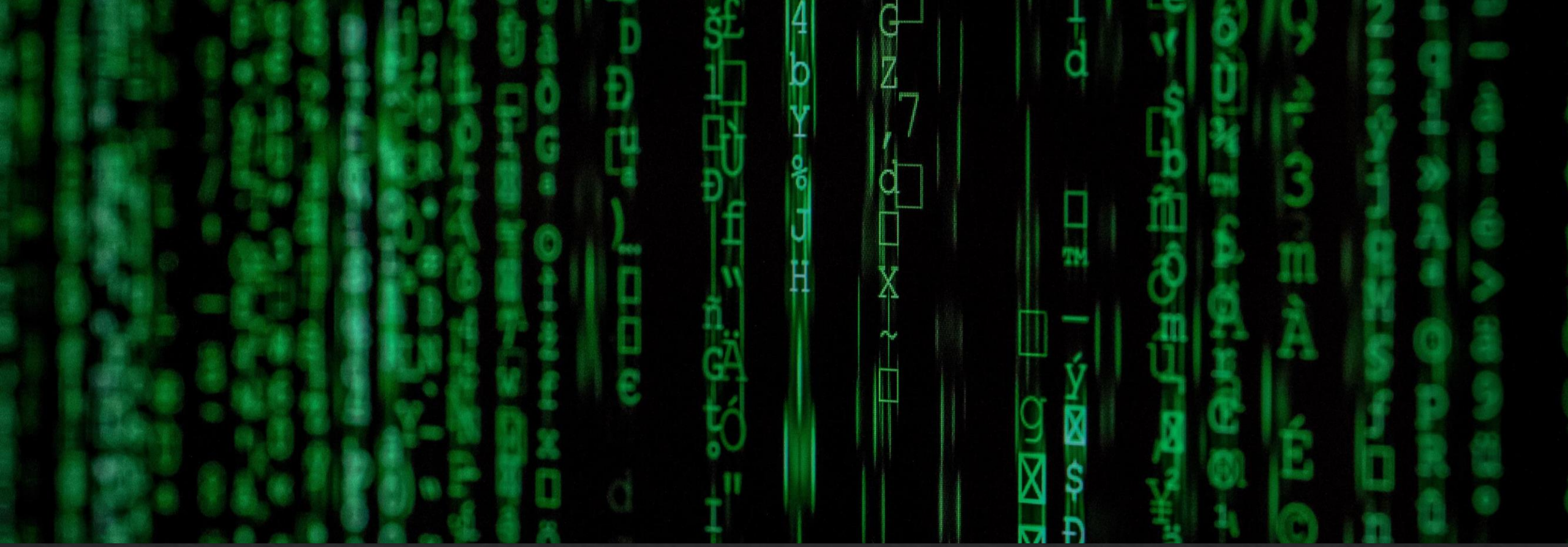
https://www.linkedin.com/in/ankursatle

It’s Valentine Week…

And love is in the air…

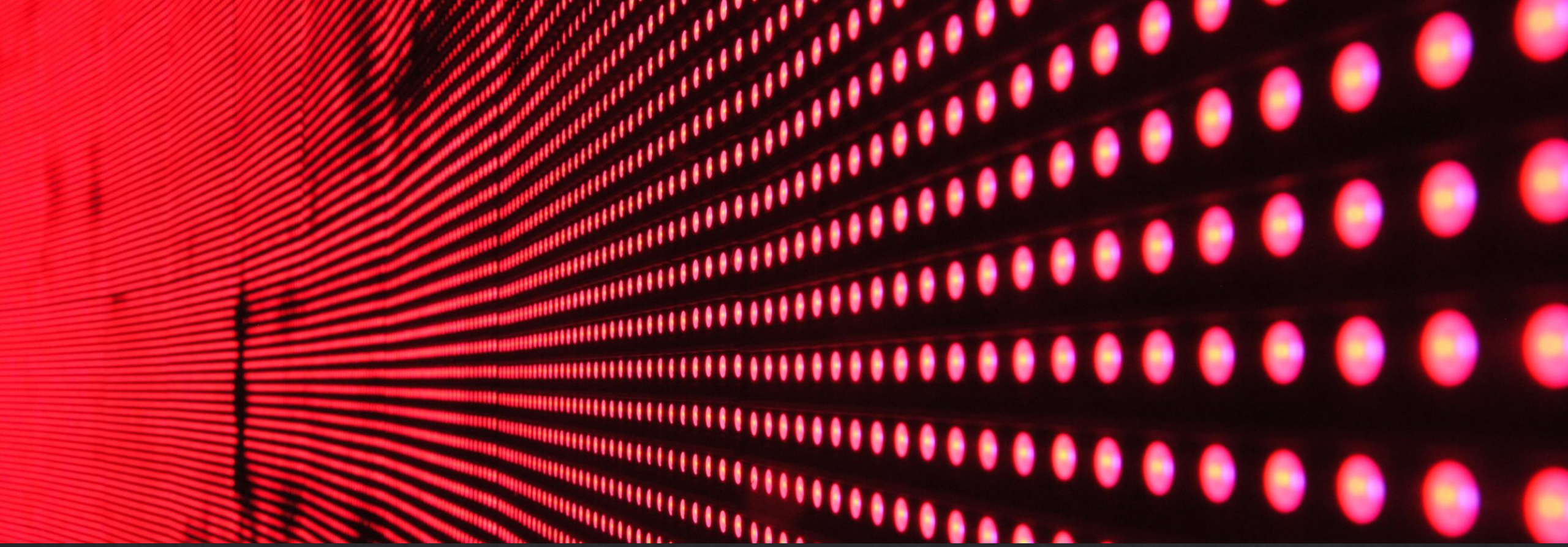
It is inevitable…

Inescapable!



For techies too… it is just the same!

The world which truly is like this

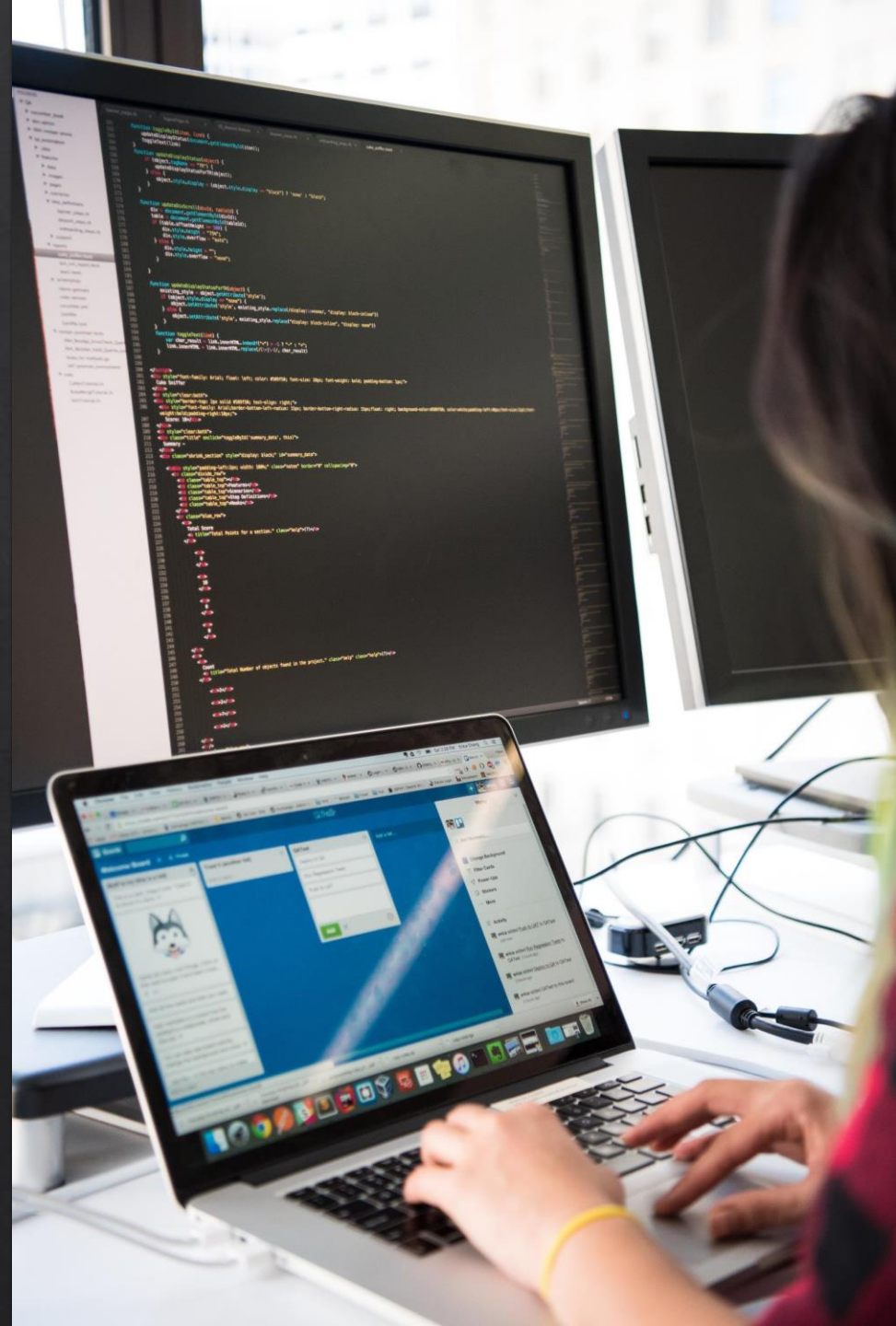


Starts looking like this…

♥ ♥ ♥

With *LOVE* on their minds!

- They are all pumped up…
- They are in deep love…
- And thinking about the inevitable…
- The love relationships…
- And the hate relationships…
- What goes well together ❤
- And what doesn't ✕
- In their own tech world! ☺



Today, in our own way, we will look at…

- The Romeos and Juliets
- The Jai & Veerus
- The Thakurs and Gabbars
- The Ram Prasad and Laxman Prasad
- The Mr. Indias and Mogambos of the world!

Love to be together = `std::pair<❤️,❤️>`

- std::pair is the easiest way to bundle two things together

```
std::pair<Employee, Department>
belongs_to(emp1, dept1);
```

- But please define a special struct even if it is to name the pair

```
struct ErrorDetail {
   error_code ec;
   std::string help_text;
}
```

- With C++17 structured bindings, you can easily dismantle both easily like this:

```
auto [emp, dept] = belongs_to;
auto [ec, txt] = returns_error();
if (ec)  print(txt);
```

decltype ❤ auto

- Compilers can deduce datatypes
- Both keywords request the compiler to do so
- Enable the implementor not to have to specify datatypes

```
auto i = 10;
auto ul = 10ul;
auto close = [](FILE* f) { return fclose(f); };
unique_ptr<FILE, ??> file_ptr(fopen("file.txt", "r"), close);
unique_ptr<FILE, decltype(close)> file_ptr(fopen("file.txt", "r"),
close);
```

decltype 💔 auto

```
template<typename Container>
auto sum(const Container& c) {
   typename c::value_type total = {};  //'c' does not name a type
   typename decltype(c)::value_type total = {}; //Aal izz well!
   for (const auto& x : c) { total += x; }
   return total;
}
vector<int> v1 = {1, 2, 3, 4};
std::cout << sum(v1);
```

default ❤ delete

- Ensure classes have their semantic meaning
- Get compiler to automate things!
- Can use with special member functions

```
class DbCache  //Not copyable, move only
{
    Connection conn;
    vector<Record> results;

    DbCache(const DbCache&) = delete;
    DbCache& operator=(const DbCache&) = delete;
    DbCache(DbCache&&) = default;
    DbCache& operator=(DbCache&&) = default;
};
```

virtual ❤ override

- They work together to make sure you do not go wrong – at compile time

```
class Timer
{
   virtual void start(Duration sec) { … };
   virtual void restart() { … };
};
class AdvancedTimer : public Timer
{
   void start(Duration sec);
   void restart(Duration sec);
   void restart(Duration sec) override;    //error: No such function
};
```

const char* X std::string_view

- Consider this code

```
const char* msg = "Cpp India!";

string to_string(const char* s);
//Every char has to be inspected
```

- How about this?

```
string to_string(string_view sv);
//sv knows the length
//copy can happen in bulk
```

- How is sv constructed though?

```
string_view sv{const char* data,
strlen(data)};
```

#define X constexpr

- Consider this code

```
const char* msg = "Cpp India!";
#define msglen 9
```

- How this improves:

```
string to_string(string_view sv);
//sv knows the length
//copy can happen in bulk
```

- Now, the call can be like this:

```
to_string( {msg, msglen} );
```

- You are fully in control!
- But manually ☹
- Did you spot the `msglen` was incorrect?

#define X constexpr

- Consider this code

```
constexpr int get_strlen(const
char* s) { ... }

const char* msg = "Cpp India!";
constexpr int msglen =
get_strlen(msg);
```

- How this improves:

```
string to_string(string_view sv);
//sv knows the length
//copy can happen in bulk
```

- Now, the call like this is better:

```
to_string( {msg, msglen} );
```

- Automate the donkey-work

String-like ❤ std::string_view

- Finally, the `to_string` helps generically handle many datatypes

- With definition as:

```
string to_string(string_view sv);
```

- Code works for any string-like type

```
const char* msg = "Cpp India!";
to_string(msg);
to_string({msg, 3});//null terminated?
string_view part(msg, 3);
to_string(part);
std::string string1("Long string");
to_string(string1);
to_string({string1, 4});//no substr!
```

#include X import

- Instead of textually including header files

```
#include <memory>
```

- It is better to use C++20 packages which are pre-compiled

```
import memory;
```

- Save lots on compilation
- No macro interference

```
#define private public
```

new ❤ delete

- They go together!
- Make sure to `delete` what you `new`

```
Interview* setup_iview(referral lead) {
    auto iview = new Interview();

    if (vacancies.empty()) {
        delete iview;
        return nullptr;
    }

    try {
        auto vacancy = vacancies.find(lead.skills);

        iview.schedule(vacancy.panelists.get_availability());
    } catch (...) {
        delete iview;
        throw;
    }
    return iview;
}
```

T* X std::unique_ptr

- As we saw on the previous slide, a T* does not automatically get deleted
- Does not convey ownership semantics
- Can cause leaks

```
{  //usage
   Interview* iv =
setup_iview(ref); //delete tp?
   return;
}
```

- Instead use this:

```
unique_ptr<Interview> setup_iview(ref){
    unique_ptr<Interview> iview(new
Interview());
   //Now, early returns ensure deletion
   //exceptions always clean-up
   //return ensures ownership transfer
   return iview;
}
```

std::unique_ptr ❤ Deleter

- std::unique_ptr has two template parameters

```
template<typename T, typename Deleter = std::default_delete<T>>
class unique_ptr;
```

- Deleter helps with customizing resource management

```
static pointer allocate(size_t n);
void deallocate(pointer p);
```

```
std::unique_ptr<Emp> emp(allocate(sizeof(Emp)), &deallocate);
```

new T X std::make_unique

- Consider this code:

```
struct Pair {
   A* a;
   B* b;
   Pair(A*, B*);
};
auto* p = Pair(new A(), new B());
```

- What if either constructor throws?
- Does this help?

```
struct Pair {
   unique_ptr<A> a;
   unique_ptr<B> b;
   Pair(A*, B*);
};
auto* p = Pair(new A(), new B());
```

new T X std::make_unique

- Better now?

```
struct Pair {
   unique_ptr<A> a;
   unique_ptr<B> b;
   Pair(unique_ptr<A>,
unique_ptr<B>);
};
auto* p = Pair(
   new A(), new B()
);
```

- Functions execute all the way. Why not make use of them?

```
struct Pair {
   unique_ptr<A> a;
   unique_ptr<B> b;
   Pair(unique_ptr<A>,
unique_ptr<B>);
};
auto* p = Pair(
   make_unique<A>(),
   make_unique<B>()
); //Pass values to make_unique
```

std::unique_ptr ❤ std::shared_ptr

- Both convey ownership
- For shared ownership use shared_ptrs
- The last shared_ptr going out of scope cleans up the object
- No naked **new** or **delete** in code!

- In action:

```
auto iview = make_unique<Interview>();
//unique_ptr<Interview>


auto meeting = make_shared<Meeting>();
//shared_ptr<Meeting>
```

Mutex::`lock()` ❤ Mutex::`unlock()`

- Always together!
- `unlock()` what you `lock()`

```
Interview* setup_iview(referral
lead) {
    vacancies_mutex.lock();

    if (vacancies.empty()) {
        vacancies_mutex.unlock();
        return nullptr;
    }
```

```
    try {
        auto vacancy =
vacancies.find(lead.skills);
    auto iview = schedule(vacancy
.panelists.get_availability());
    } catch (...) {
        vacancies_mutex.unlock();
        throw;
    }
    vacancies_mutex.unlock();
    return iview;
}
```

Mutex::lock() ✕ std::lock_guard

- We solved memory leak with unique_ptr
- Can we apply the same technique here??
- We fondly call it RAII ☺

```
Interview* setup_iview(referral lead) {
   vacancies_mutex.lock();

   if (vacancies.empty()) {
      vacancies_mutex.unlock();
      return nullptr;
   }
   try {
      auto vacancy = vacancies.find(lead.skills);
      auto iview = schedule(vacancy.panelists.get_availability());
   } catch (...) {
      vacancies_mutex.unlock();
      throw;
   }
   vacancies_mutex.unlock();
   return iview;
}
```

Mutex::lock() ✕ std::lock_guard

- We solved memory leak with unique_ptr
- Can we apply the same technique here??
- We fondly call it RAII ☺

```
Interview* setup_iview(referral lead) {
    std::lock_guard<std::mutex> guard(vacancies_mutex);

    if (vacancies.empty()) {
        vacancies_mutex.unlock();
        return nullptr;
    }

    try {
        auto vacancy = vacancies.find(lead.skills);
        auto iview = schedule(vacancy.panelists.get_availability());
    } catch (...) {
        vacancies_mutex.unlock();
        throw;
    }
    vacancies_mutex.unlock();
    return iview;
}
```

Mutex::`lock()` ✕ `std::lock_guard`

- We solved memory leak with unique_ptr
- Can we apply the same technique here??
- We fondly call it RAII ☺
- Life is simplified – the function becomes just this:
- Now, what if there is a surprise and the call to `vacancies.empty()` throws?
- The gains are more than what you aimed for! ☺

```
Interview* setup_iview(referral lead) {
   std::lock_guard<std::mutex> guard(vacancies_mutex);
   if (vacancies.empty()) {
      return nullptr;
   }
   auto vacancy = vacancies.find(lead.skills);
    return schedule(vacancy.panelists.get_availability());
}
```

Mutex ❤ lock_guard, lock, unique_lock, scoped_lock

- **lock_guard** = RAII mutual exclusion

```
void transfer(from, to) {
   //validate & condition input
   //critical section
   //condition output & return
}
```

- This becomes:

```
void transfer(from, to) {
   //validate & condition input
   {
      lock_guard<mutex> lg(trans_mut);
      //Do the work
   }
   //condition output & return
}
```

Mutex ❤ lock_guard, lock, unique_lock, scoped_lock

- **lock** = Deals with multiple mutexes
- **unique_lock** = Ownership & transfer

```
void transfer(from, to) {
    //validate & condition input
    //critical section
    //condition output & return
}
```

- This becomes:

```
void transfer(from, to) {
    //validate & condition input
    {
        unique_lock<mutex> lk1(from.mut, std::defer_lock);
        unique_lock<mutex> lk2(to.mut , std::defer_lock);
        std::lock(lk1, lk2);
        //Do the work
    }
    //condition output & return
}
```

Mutex ❤ lock_guard, lock, unique_lock, scoped_lock

- **scoped_lock** = Simplifies dealing with multiple mutexes

```
void transfer(from, to) {
   //validate & condition input
   //critical section
   //condition output & return
}
```

- Now becomes:

```
void transfer(from, to) {
   //validate & condition input
   {
      scoped_lock l(from.mut, to.mut);
      //Do the work
   }
   //condition output & return
}
```

- This is equivalent to the code with `unique_lock` & `lock`

Happy Valentine’s Day!

We all can do with more ❤️ and less ❌

References

- GotW #102: Exception-Safe Function Calls – Herb Sutter - https://herbsutter.com/gotw/_102
- https://en.cppreference.com/


Thank you!