How C++ eliminated Memory Issues

Ankur M. Satle

https://ankursatle.wordpress.com

ankursatle@gmail.com

 /ankursatle

 @ankursatle

Top 10 Most Common C++ Dev Mistakes (2015)

- Using “new” and ”delete” Pairs Incorrectly
- Forgotten Virtual Destructor
- Deleting an Array With “delete” or Using a Smart Pointer
- Returning a Local Object by Reference
- Using a Reference to a Deleted Resource
- Allowing Exceptions to Leave Destructors
- Using “auto_ptr” (Incorrectly)
- Using Invalidated Iterators and References
- Passing an Object by Value
- Using User Defined Conversions by Constructor and Conversion Operators

Source: https://www.toptal.com/c-plus-plus/top-10-common-c-plus-plus-developer-mistakes

The CWE Top 25

37889 CVE Records in 2022 in the Common Weakness Enumeration

Below is a list of the weaknesses in the 2022 CWE Top 25, including the overall score of each. The KEV ecords from the CISA KEV list that were mapped to the given weakness.

6/25 relate to memory issues

Rank	ID	Name	Score	KEV Count (CVEs)	Rank Change vs. 2021
1	CWE-787	Out-of-bounds Write	64.20	62	0
2	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	45.97	2	0
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	22.11	7	+3 ▲
4	CWE-20	Improper Input Validation	20.63	20	0
5	CWE-125	Out-of-bounds Read	17.67	1	-2 ▼
6	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	17.53	32	-1 ▼
7	CWE-416	Use After Free	15.50	28	0
8	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	14.08	19	0
9	CWE-352	Cross-Site Request Forgery (CSRF)	11.53	1	0
10	CWE-434	Unrestricted Upload of File with Dangerous Type	9.56	6	0
11	CWE-476	NULL Pointer Dereference	7.15	0	+4 ▲
12	CWE-502	Deserialization of Untrusted Data	6.68	7	+1 ▲
13	CWE-190	Integer Overflow or Wraparound	6.53	2	-1 ▼
14	CWE-287	Improper Authentication	6.35	4	0
15	CWE-798	Use of Hard-coded Credentials	5.66	0	+1 ▲
16	CWE-862	Missing Authorization	5.53	1	+2 ▲
17	CWE-77	Improper Neutralization of Special Elements used in a Command ('Command Injection')	5.42	5	+8 ▲
18	CWE-306	Missing Authentication for Critical Function	5.15	6	-7 ▼
19	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	4.85	6	-2 ▼
20	CWE-276	Incorrect Default Permissions	4.84	0	-1 ▼
21	CWE-918	Server-Side Request Forgery (SSRF)	4.27	8	+3 ▲
22	CWE-362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	3.57	6	+11 ▲
23	CWE-400	Uncontrolled Resource Consumption	3.56	2	+4 ▲
24	CWE-611	Improper Restriction of XML External Entity Reference	3.38	0	-1 ▼
25	CWE-94	Improper Control of Generation of Code ('Code Injection')	3.32	4	+3 ▲

In 2021, Memory issues accounted for 40% of the points scored for all categories the Most Dangerous Software Weaknesses listed

The picture is very gloomy

- Options to counter memory issues?
- Of course…
- Drum roll…

Sean Parent

Exceptions the Other Way Around

AssertMacros.h

```
/*
  File:       AssertMacros.h

  Contains:   This file defines structured error handling and assertion macros for
              programming in C. Originally used in QuickDraw GX and later enhanced.
              These macros are used throughout Apple's software.

              See "Living In an Exceptional World" by Sean Parent
              (develop, The Apple Technical Journal, Issue 11, August/September 1992)
              <http://developer.apple.com/dev/techsupport/develop/issue11toc.shtml> or
              <http://www.mactech.com/articles/develop/issue_11/Parent_final.html>
              for the methodology behind these error handling and assertion macros.

  *  Macro overview:
  *
  *      check(assertion)
  *          In production builds, pre-processed away
  *          In debug builds, if assertion evaluates to false, calls DEBUG_ASSERT_MESSAGE
  *
  *      require(assertion, exceptionLabel)
  *          In production builds, if the assertion expression evaluates to false, goto exceptionLabel
  *          In debug builds, if the assertion expression evaluates to false, calls DEBUG_ASSERT_MESSAGE
  *                           and jumps to exceptionLabel
  *
```

Meanwhile…

At isocpp.org

Asked

2022 Annual C++ Developer Survey "Lite"

Q6 Which of these do you find frustrating about C++ development?

Answered: 1,185 Skipped: 4

2022 Annual C++ Developer Survey "Lite"

	MAJOR PAIN POINT	MINOR PAIN POINT	NOT A SIGNIFICANT ISSUE FOR ME	TOTAL	WEIGHTED AVERAGE
Managing libraries my application depends on	47.63% 563	34.77% 411	17.60% 208	1,182	2.30
Build times	43.94% 515	38.65% 453	17.41% 204	1,172	2.27
Setting up a continuous integration pipeline from scratch (automated builds, tests, ...)	33.73% 394	40.75% 476	25.51% 298	1,168	2.08
Setting up a development environment from scratch (compiler, build system, IDE, ...)	27.83% 329	42.98% 508	29.19% 345	1,182	1.99
Concurrency safety: Races, deadlocks, performance bottlenecks	25.04% 293	46.67% 546	28.29% 331	1,170	1.97
Managing CMake projects	29.34% 343	38.15% 446	32.51% 380	1,169	1.97
Debugging issues in my code	17.85% 209	54.57% 639	27.58% 323	1,171	1.90
Parallelism support: Using more CPU/GPU/other cores to compute an answer faster	20.74% 242	37.79% 441	41.47% 484	1,167	1.79
Memory safety: Bounds safety issues (read/write beyond the bounds of an object or array)	14.81% 174	37.70% 443	47.49% 558	1,175	1.67
Memory safety: Use-after-delete/free (dangling pointers, iterators, spans, ...)	13.93% 163	37.01% 433	49.06% 574	1,170	1.65
Managing Makefiles	19.88% 226	21.37% 243	58.75% 668	1,137	1.61
Managing MSBuild projects	18.41% 209	19.30% 219	62.29% 707	1,135	1.56
Security issues: Overlaps with “safety” but includes other issues (secret disclosure, vulnerabilities, exploits, ...)	9.36% 109	35.40% 412	55.24% 643	1,164	1.54
Type safety: Using an object as the wrong type (unsafe downcasts, unsafe unions, ...)	8.77% 103	31.66% 372	59.57% 700	1,175	1.49
Memory safety: Forgot to delete/free (memory leaks)	8.68% 102	27.06% 318	64.26% 755	1,175	1.44
Moving existing code to the latest language standard	7.03% 83	27.54% 325	65.42% 772	1,180	1.42

2022 Annual C++ Developer Survey Results

- Memory Issues are a major pain point ONLY for
  - Bounds safety: 15%
  - Use after delete/free: 14%
  - Memory leaks: < 9%

- Why don't the C++ developers have memory issues?
- What changed?
- Let's find out

$whoami

- Architect at EXFO
- High-Performance Telecom Products
- Cloud-Native products
- Neophilia

- https://ankursatle.wordpress.com
- ankursatle@gmail.com
- https://www.linkedin.com/in/ankursatle/
- https://twitter.com/AnkurSatle

Memory Issues

Memory Issues

- Incorrect delete
  - `new[]`  `delete`
  - `new`  `delete[]`
- Memory Leak
  - `new`  ~~`delete`~~
- Double delete
  - `new`  `delete; delete`
- Use-after-free
  - `new`  `delete; use`
- Uninitialized use
  - ~~`new`~~  `use`
- Out of bounds use
  - `new[N]`  `use(N+1)`

Which of the statements are correct?

- Given:

```
int* f(...);
int* p = f(...);
```

```
*p;
p++;
p[5];
delete p;
delete[] p;
```

Pointer members

- Given:

```
class C1 {
   const char* a;
   X* b;
   P* c;
   const D* d;
};
```

- What should the destructor do for each member?

```
C1::~C1() {
   delete? a;
   delete? b;   //delete[]?
   delete? c;
   delete? d;
};
```

Pointer members - realistic

- Given:

```
class Order {
   const char* name;
   Item* items;
   Insurance_policy* policy;
   const Agent* agent;
};
```

- What should the destructor do for each member?

```
Order::~Order() {
   delete[] name;//const, many & owned
   delete[] items;//non-const,many, owned
   delete? policy;//non-const, single
   //shared ownership - with financial
   docket
   delete agent;//const, single, not
   owned
};
```

- All special member functions must handle the pointers correctly

C++ inherited pointers from C

- Enable reference semantics
- Passing pointers efficient for non-trivial types
- Problems
  - NULL or nullptr
  - Creation with a garbage value by default
  - Ownership
  - No of elements
  - Many pointers can point to the same address
- Our endeavor = ensure clarity & guarantees
  - Presence/absence of value
  - initialization
  - Manage ownership
  - Handle multiplicity
  - Shared instance

Why C++ exists!


Ólafur Waage
@olafurw
Replying to @pati_gallardo
Shamelessly relevant.
Translate Tweet
Ólafur Waage @olafurw · Apr 22, 2021
C
raw pointers
++
trying to prevent people from using raw pointers
ALT
12:24 AM · Aug 2, 2022 · TweetDeck

Special member functions – Rule of 3

Pointer members – handling in special functions

- Due to pointers:

```
class Order {
   const char* name;
   Item* items;
   Insurance_policy* policy;
   const Agent* agent;
```

- Due to pointers:

```
Order() : name(0), items(0),
policy(0), agent(0){}
Order& operator=(const Order&
other) {}
```

- Cannot let pointers to be copied:

```
Order(const Order& other) {
   delete[] name;
   name = strdup(other.name);
   …
};
~Order() {
    delete[] name;
    delete[] items;
    delete policy;
}
};
```

Special member functions – Rule of 5

Pointer members – handling in special functions

- C++11 got us move!

```
class Order {
    const char* name;
    Item* items;
    Insurance_policy* policy;
    const Agent* agent;
```

- Due to pointers:

```
Order() : name(0), items(0),
policy(0), agent(0){}
Order& operator=(const Order& other)
{}
Order(const Order& other) {}
~Order() {}
```

- We also need to take care of move operations

```
Order(Order&& other) {
    std::swap(name, other.name);
    delete[] other.name;
    other.name = nullptr;
    …
};
Order& operator=(Order&& other) {
    //Similar to above
    std::swap(name, other.name);
    delete[] other.name;
    other.name = nullptr;
}
};
```

Oh, so much to manage!

And a reason for errors to creep in!

Pointer members – we already have std::string

- Instead of:

```
class Order {
   const char* name;
   Item* items;
   Insurance_policy* policy;
   const Agent* agent;
};
```

- We can have:

```
class Order {
   std::string name;
   Item* items;
   Insurance_policy* policy;
   const Agent* agent;
};
```

Pointer members – std::string

- Using std::string

```
class Order {
   std::string name;
   Item* items;
   Insurance_policy* policy;
   const Agent* agent;
```

- Default constructs

```
Order() : name(0), items(0),
policy(0), agent(0){}
Order& operator=(const Order&
other) {}
```

- Smooth

```
Order(const Order& other) :
   name(other.name)
   { … };
~Order() {
    delete[] name;
    delete[] items;
    delete policy; }
};
```

- We do not have to do anything for std::string!

Pointer members – std::string

- C++11 got us move!

```
class Order {
   std::string name;
   Item* items;
   Insurance_policy* policy;
   const Agent* agent;
```

- So:

```
Order() : name(0), items(0),
policy(0), agent(0){}
Order& operator=(const Order&
other) {}
Order(const Order& other) {}
~Order() {}
```

- Move operations

```
Order(Order&& other) :
   name(other.name)
   { … };
Order& operator=(Order&& other) {
   std::swap(name, other.name);
   other.name = {};
   …
}
};
```

I want the same thing for my type!

No management is the best management!

Pun!? ☺

Pointer members – std::unique_ptr

- Due to pointers:

```
class Order {
    std::string name;
    Item* items;
    std::unique_ptr<Insurance_policy>
    policy;
    const Agent* agent;
```

- Default constructs

```
Order() : name(0), items(0),
policy(0), agent(0){}
Order& operator=(const Order& other)
{}
```

- Cannot let pointers to be copied:

```
Order(const Order& other) :
    name(other.name),
    policy(
    std::make_unique<Insurance_policy
    >(policy.get())) //deleted copy
    { … };
~Order() {
    delete[] name;
    delete[] items;
    delete policy; }
};
```

- Pretty much like the std::string!

std::unique_ptr does not allow for copy

A welcome guarantee for unique ownership

Pointer members – std::unique_ptr

- Move constructor & move assignment with unique_ptr

```
class Order {
    std::string name;
    Item* items;
    std::unique_ptr<Insurance_policy>
    policy;
    const Agent* agent;
};
```

- Free of management:

```
Order::Order() : items(0), policy(0),
agent(0){}
Order::operator=(const Order& other)
{}
```

- Cannot let pointers to be copied:

```
Order(Order&& other) :
    //handle for name
    policy(std::move(other.policy));
    …
};
Order::~Order() {
    delete[] items;
    delete policy;
}
```

Pointer members – manage many objects similarly

- For array of objects

```
class Order {
   std::string name;
   std::unique_ptr<Item[]> items;
   std::unique_ptr<Insurance_policy>
   policy;
   const Agent* agent;
};
```

- Free of management:

```
Order::Order() : items(0),
agent(0){}

Order::operator=(const Order&
other) {}
```

- Cannot let pointers to be copied:

```
Order(const Order& other) :
   items(
   std::make_unique<Item[]>(size))
   {…};
Order::~Order() {
    delete[] items;
}
```

Pointer members – manage many objects similarly

- Move constructor & move assignment with unique_ptr

```
class Order {
   std::string name;
   std::unique_ptr<Item[]> items;
   std::unique_ptr<Insurance_policy>
   policy;
   const Agent* agent;
};
```

- Free of management:

```
Order::Order() : items(0), agent(0){}
Order::operator=(const Order& other)
{}
```

- Cannot let pointers to be copied:

```
Order(Order&& other) {
   //handle for name
   std::swap(items,
   std::move(other.items));
   …
};
Order::~Order() {
    delete[] items;
}
```

Pointer members – manage single object similarly

- With all this:

```
class Order {
   std::string name;
   Item items;
   Insurance_policy* policy;
   const Agent* agent;
};
```

- The destructor need to do nothing for:

```
class Order {
   std::string name;
   std::unique_ptr<Item[]> items;
   std::unique_ptr<Insurance_policy>
   policy;
   const Agent* agent;
};
```

Shared ownership

Ladies first! But who goes last!? ☺

Pointer members – std::shared_ptr

- Move constructor & move assignment with unique_ptr

```
class Order {
   std::string name;
   std::unique_ptr<Item[]> items;
   std::shared_ptr<Insurance_policy>
   policy;
   const Agent* agent;
};
```

- Free of management:

```
Order::Order() : agent(0){}
Order::operator=(const Order& other)
{}
```

- Copy Constructor

```
Order(const Order& other) :
   //copy results in shared ownership
   policy(other.policy) {
   …
};
Order::~Order() {}
```

Pointer members – std::shared_ptr

- Move constructor & move assignment with unique_ptr

```
class Order {
   std::string name;
   std::unique_ptr<Item[]> items;
   std::shared_ptr<Insurance_policy>
   policy;
   const Agent* agent;
};
```

- Free of management:

```
Order::Order() : items(0), agent(0){}
Order::operator=(const Order& other)
{}
```

- Cannot let pointers to be copied:

```
Order(Order&& other) :
   //handle for name
   policy(std::move(other.policy)) {
   …
};
Order::~Order() {}
```

Pointer members – std::shared_ptr

- With all this:

```
class Order {
   std::string name;
   Item items;
   Insurance_policy* policy;
   const Agent* agent;
};
```

- The destructor need to do nothing for:

```
class Order {
   std::string name;
   std::unique_ptr<Item[]> items;
   std::shared_ptr<Insurance_policy>
   policy;
   const Agent* agent;
};
```

I tricked you previously

Let me correct it

Pointer members – many items

- Move constructor & move assignment with unique_ptr

```
class Order {
   std::string name;
   std::unique_ptr<Item[]> items;
   std::shared_ptr<Insurance_policy>
   policy;
   const Agent* agent;
};
```

- Free of management:

```
Order::Order() : agent(0){}
Order::operator=(const Order& other)
{}
```

- Copy Constructor

```
Order(const Order& other) :
   //copy results in shared ownership
   items(
   std::make_unique<Item[]>(size));
   …
};
Order::~Order() {}
```

- We do not have the size of the array
- unique_ptr does not track the size
- It manages ownership only

Pointer members – many items

- Move constructor & move assignment with unique_ptr

```
class Order {
   std::string name;
   std::vector<Item> items;
   std::shared_ptr<Insurance_policy>
   policy;
   const Agent* agent;
};
```

- Free of management:

```
Order::Order() : agent(0){}
Order::operator=(const Order& other)
{}
```

- Copy Constructor

```
Order(const Order& other) :
   //copy results in shared ownership
   items(other.items) {
   …
};
Order::~Order() {}
```

- std::vector is also a manages its own memory!
- All good now!

Pointer members – std::shared_ptr

- Move constructor & move assignment with unique_ptr

```
class Order {
   std::string name;
   std::unique_ptr<Item[]> items;
   std::shared_ptr<Insurance_policy>
   policy;
   const Agent* agent;
};
```

- Free of management:

```
Order::Order() : items(0), agent(0){}
Order::operator=(const Order& other)
{}
Order::~Order() {}
```

- Move

```
Order(Order&& other) :
   //vector defines move constructor
   policy(other.policy) {
   …
};
Order::operator=(Order&& other){
   policy = other.policy;
}
```

Nothing to do in special functions now!

No code is the best code!

```
class Order {
   std::string name;
   std::unique_ptr<Item[]> items;
   std::shared_ptr<Insurance_policy> policy;
   const Agent* agent;

   Order(const Order&) = default;
   Order(Order&&) = default;
   Order& operator=(const Order&) = default;
   Order& operator=(Order&&) = default;
   ~Order() = default;
};
```

- Focus on business logic, shall we?

Nullability of pointers

Do not pass null pointers

```
void Order::setAgent(Agent* a)
{
    if (a != nullptr) {
        agent = a;
    }
}
```

- Called function checks all the time
- Most probably, the function cannot do it's job without the object
- Optional parameters = Can improve design to remove optionality

Fool-proof management

Leak

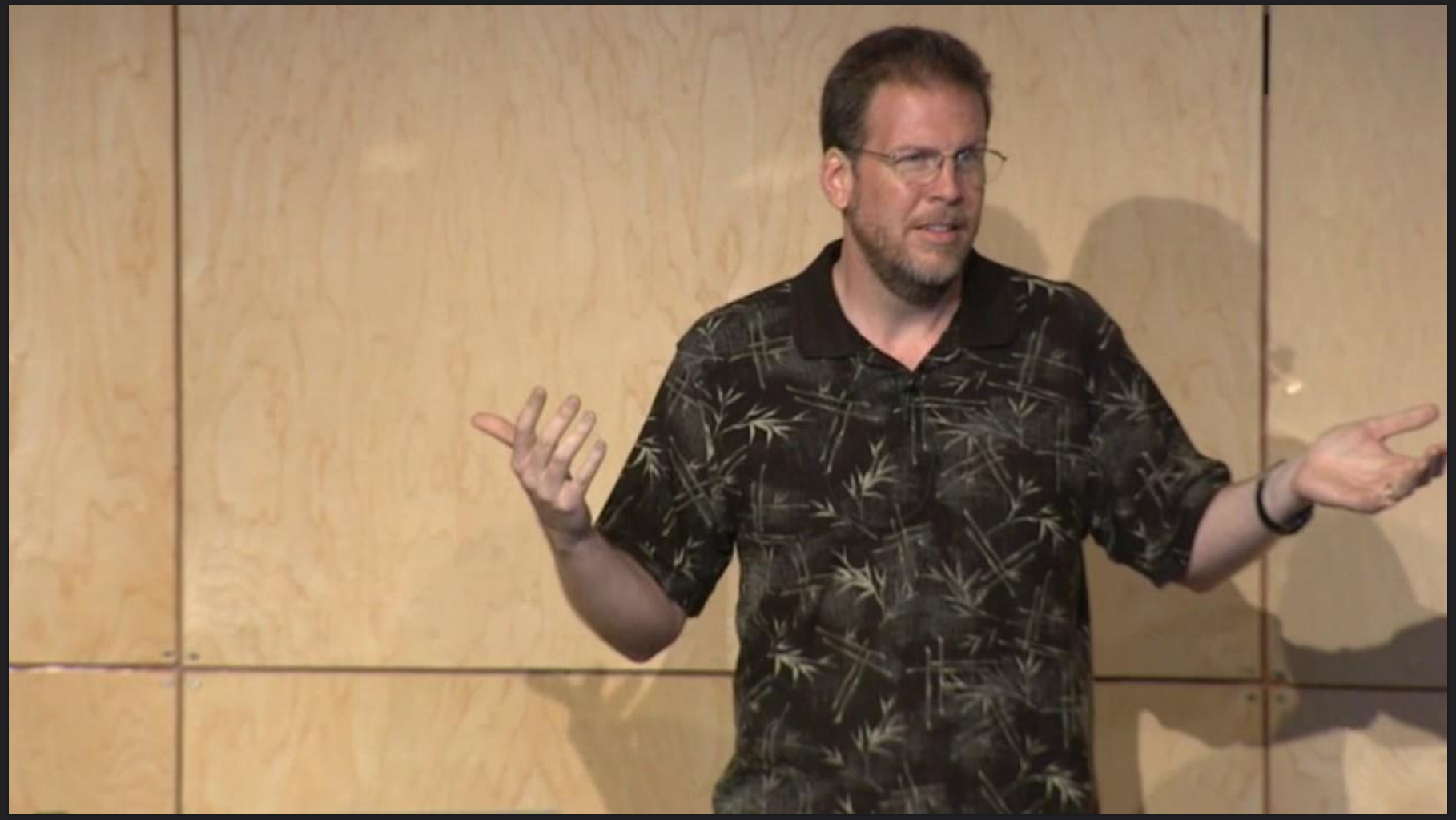
Leak

```
Server* create_server() {
   Server s = new Server;
   s->load_config("config.json");
   if (error) return nullptr;
   s->connect();
   if (error) throw connection_error;
   s->subscribe();
   if (error) return nullptr;
   s->keep_alive_timer(timer1);
   return s;
}
```

Get Smart!

```
std::unique_ptr<Server> create_server() {
    auto s = make_unique<Server>();
    s->load_config("config.json");
    if (error) return nullptr;
    s->connect();
    if (error) throw connection_error;
    s->subscribe();
    if (error) return nullptr;
    s->keep_alive_timer(timer1);
    return s;
}
```

No Raw Pointers (that own the memory)



Other managed types

std::array<T, N>

- Fixed size & managed

```
std::array conns = {Connection{ip_port1},
                    Connection{ip_port2},
                    Connection{ip_port3}};
```

- Can be instantiated and then initialised

```
std::array<Connection, 3> conns;
conns[0] = Connection{ip_port1};
conns[1] = Connection{ip_port1};
conns[2] = Connection{ip_port1};
```

std::variant<T, U, V>

- Type-safe **union**

```
struct connected { Timer refresh_timer; };
struct disconnected { time_point last_connected; };
std::variant<connected, disconnected> con_state = connected;
//Use refresh_timer of con_state
std::variant<connected, disconnected> con_state = disconnected;
//Use last_connected of con_state
```

- Need to use `std::visit` to process values

```
std::visit([](const auto& val) { std::cout << val; }, con_state);
```

Return Value Optimization

C++17

- Brought return value optimization
- Before

```
Connection con = create_con(ip_port1); //copy!!!???
Connection* pcon = create_con(ip_port1);  //what a relief!
```

- To circumvent this:

```
Connection con;
//someone uses con already!!! Oops!
create_con(ip_port1, &con);   //instantiation-init separation!
```

- But Return Value Optimization helps us write

```
Connection con = create_con(ip_port1); //no copy guaranteed
```

- We can stay in object land without any penalty! Zero Overhead!

std::move

- Even if you have to move, it's very inexpensive!

```
std::vector<User> users_cache;
use(users_cache);
//User changed connection config - time to reload
users_cache = reload_users(db_connection);   //Move!
```

- With a little cost of swapping some pointers and integers, we have a whole new list

RVO & std::move magical together

std::optional<T>

https://en.cppreference.com/w/cpp/utility/optional

Member types

Member type	Definition
value_type	T

Member functions

(constructor)	constructs the optional object (public member function)
(destructor)	destroys the contained value, if there is one (public member function)
operator=	assigns contents (public member function)

Observers

operator-> operator*	accesses the contained value (public member function)
operator bool has_value	checks whether the object contains a value (public member function)
value	returns the contained value (public member function)
value_or	returns the contained value if available, another value otherwise (public member function)

Monadic operations

and_then (C++23)	returns the result of the given function on the contained value if it exists, or an empty optional otherwise (public member function)
transform (C++23)	returns an optional containing the transformed contained value if it exists, or an empty optional otherwise (public member function)
or_else (C++23)	returns the optional itself if it contains a value, or the result of the given function otherwise (public member function)

Modifiers

swap	exchanges the contents (public member function)
reset	destroys any contained value (public member function)
emplace	constructs the contained value in-place (public member function)

Non-member functions

operator== (C++17)	
operator!= (C++17)	
operator< (C++17)	compares optional objects

std::optional

```
std::optional emp = find_employee(name);
if (emp.has_value()) {
   use(emp);
}

std::optional<Address> use(Employee& e) { … }
find_employee(name).and_then(use).and_then(print_address);

std::optional<Employee*> p = find_a_ptr(…);  //Works for pointers!
```

Impossible to use incorrectly with and_then continuation

Life’s cool with these value-oriented types

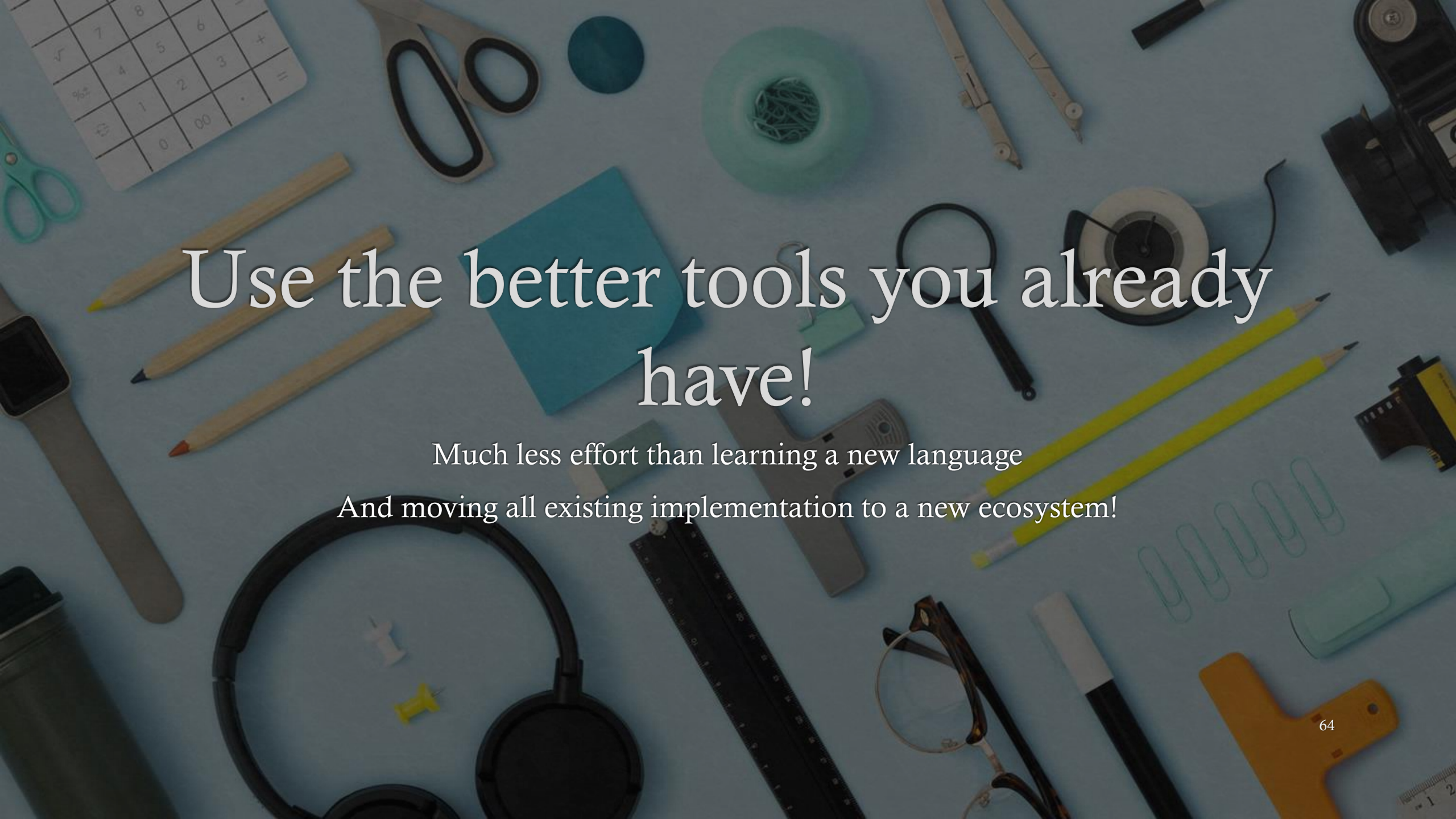
- Previously at CppIndia

```
return get_env("MQInitClientConfigFile")
        .and_then(get_file_contents)
        .and_then(make_client)
        .transform(apply_config)
        .transform(connect)
        .and_then(subscribe);
```

Conclusion

- Do not manually invoke raw memory management primitives
- Pointers do not own memory
- Use types that manage themselves!

- Uphold your values high! You are in C++!



Use the better tools you already have!

Much less effort than learning a new language

And moving all existing implementation to a new ecosystem!

Uphold your high values!

No room for pointers & manual memory management!

Thank you!

Ankur Satle

CppCon 2015: Herb Sutter "Writing Good C++14… By Default"