C++ for Enterprise Applications

VINCENT LEXTRAIT

Cppcon
The C++ Conference

2022 | September 12th-16th

Agenda

- Enterprise Applications, a C++ paradox
- Why this paradox?
- C++ can do way better than existing frameworks
- Examples
  - Referential Integrity
  - Objects automatic destruction as a “free” byproduct
  - Replacing SQL with C++ functions

Enterprise Applications, a C++ paradox

- C++ has one of the **largest shares** of the **number** of **back-end transactions processed**
- **Large Enterprise Applications** are mostly C++-based
- Yet, C++' share in **number of Enterprise Applications** is very small*
- Why?
  - Our **C++ community toolkits** are **inadequate** or **absent**
  - Only **medium to large companies** can afford compensating and **building their own**
  - Leads to the general (mis)understanding that **C++** is not meant for **large-scale abstractions** and is only meant for **low-level/infrastructure software**
  - Yet, increasing trend (only in large companies) to **elevate C++ abstractions to specification**

*Source: SlashData, "State of the Developer Nation" 22nd Edition, Q1 2022

Enterprise Applications

- The need is to **"handle persistent data concurrently"** (and remotely for online)
  - Lots of data that do not fit in RAM – need to rely on databases and factor in their constraints
  - Need for **fast response time**, **scalability**, **availability**, **fault-tolerance***, **consistency**
  - **Enterprise Applications** are a perfect example of **constrained environment**
- Everything starts with **data types and relationships**
  - **Few algorithms** – mostly binary trees access
  - **Lots of types**, **lots of relationships**
  - Data **integrity** issues
- Enterprise Applications suffer from **low engineering productivity**, **little reuse**, **high testing costs** and with time, collapsing **agility** due to gradually increased **complexity**
  - All **tell-tale** signs of **insufficient abstraction. "Raise the level of abstraction!"**
  - **Generating code from specifications** always failed

*Read "partition tolerance".

Our C++ Tools are Inadequate

- `std::unique_ptr` and `std::shared_ptr` as is are insufficient to describe relationships
- **Boost** is very useful but still falls short
- **No Application Server** available
- **SQL** makes our C++ code **fragile**: it relies upon a **cross-cut of the type system**
- **Zero-cost abstractions** pave the way to **layered abstractions** (aka **Hyperautomation/Hyper-abstraction**), let's use them
- Let's **learn from the past** and go **back to basics**

*Source: SlashData, "State of the Developer Nation" 22nd Edition, Q1 2022

So many failed attempts at Diagram-Based Engineering

- 1980s had the **Flow Charts**
- +20: 2000s had **UML**
- +20: In 2020 ”**No Code**” (exclusively diagrams) regained interest
- Nobody will try as hard as Grady Booch who unsuccessfully tried diagram-based software engineering with UML
- We need a **different approach**: what if we always started building the bridge from the **wrong end**?

Specification or Code? Both!

- What is "**code**"? People usually mean (low-level) "**programs**" by that
- What is "**specification**"? Describing the "what", not the "how"
- **Spreadsheets**, which replaced programs using financial libraries showed that **formulating only the what is possible**
- **Separating** levels of "**what**" and "**how**" is precisely what programming is: **defining abstractions**
- By **layering abstractions**, at a level high enough, **specification and code become the same**
- Let's **start** with **data modeling**

"Data Model" is tired, "Ontology" is wired

Wikipedia for "**Ontology (information science)**":

In computer science and information science, an **ontology** encompasses a representation, formal naming, and definition of the **categories**, **properties**, and **relations** between the concepts, data, and entities that substantiate one, many, or all **domains of discourse.** More simply, an ontology is a way of showing the properties of a subject area and how they are related, by defining a set of concepts and categories that represent the subject.

Adopted first by **AI** (knowledge acquisition), gaining momentum in **data science**.

Let's aim at offering a generic **C++ toolkit** allowing to **describe ontologies easily**, **quickly**, **reusably**, and embed all the logic we need to produce **complete Enterprise Applications for any domain**.

Let's have the ambition to produce applications way **cheaper**, **more sophisticated** and **more efficient** than anything **written by hand**.

*Source: SlashData, "State of the Developer Nation" 22nd Edition, Q1 2022

METASPEX

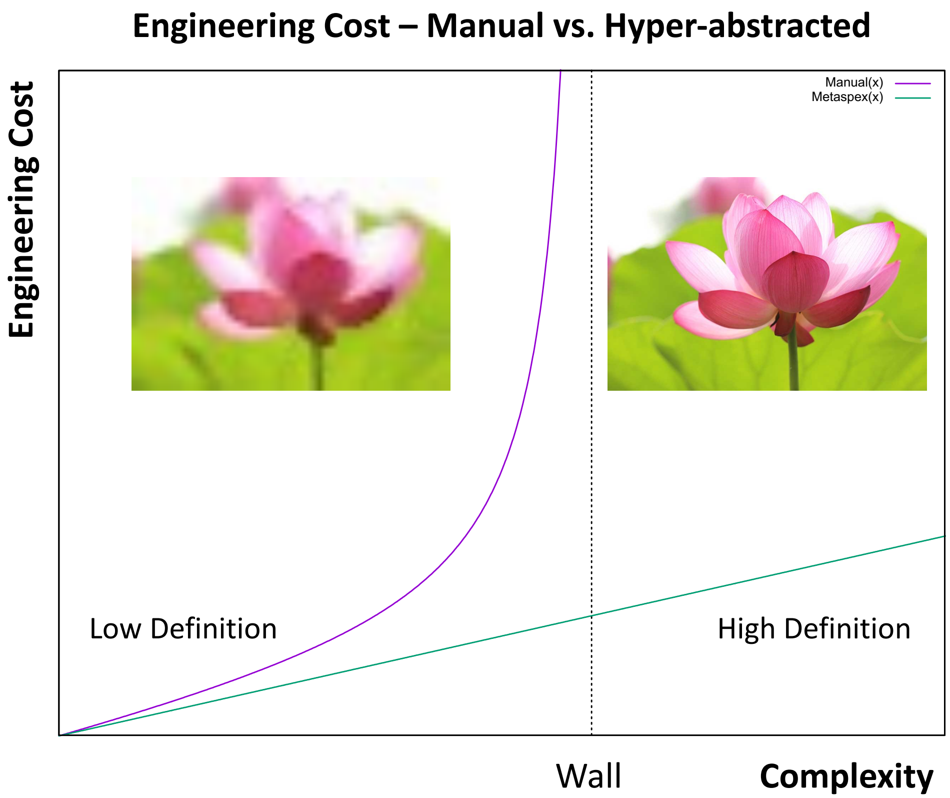
Hyper-abstracted C++: Starting from the "Other End"
Bridging abstraction arches from low-level code to specification

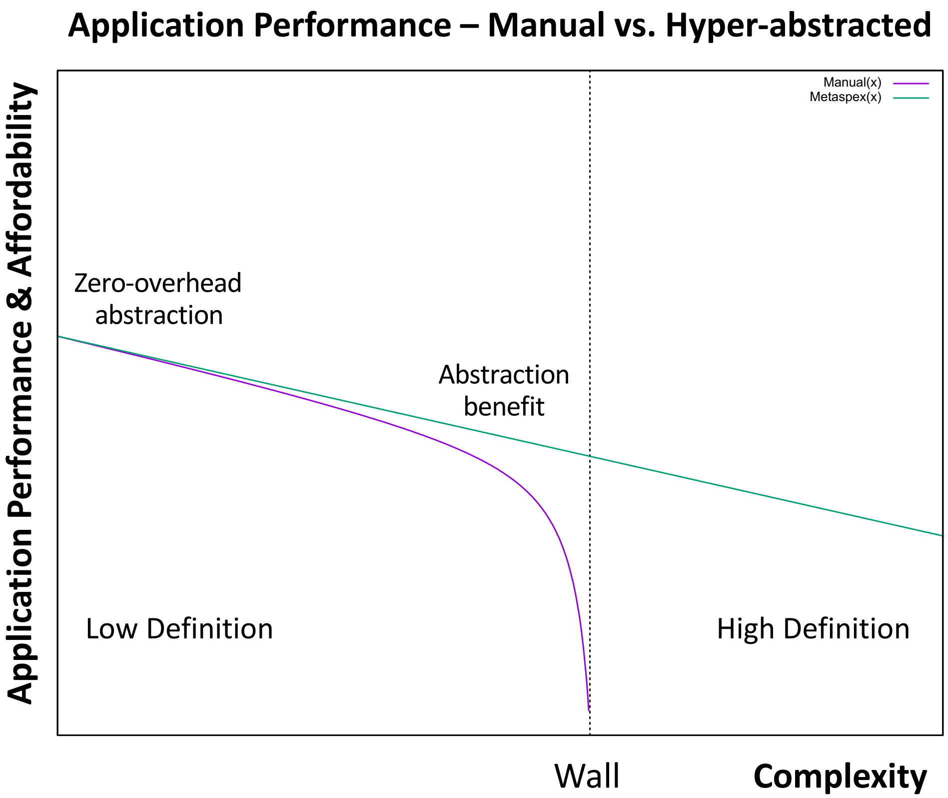
- By **specification** we mean **concise** and **precise** source that goes **directly** into a regular **C++ compiler**
- We started from the **standard library** and **Boost**
- It takes **5 to 8** additional carefully crafted abstraction/template "**arches**" to **bridge implementation to specification**
- Example of **abstraction layering**: raw pointer, smart pointer, relationship, ontology type (generic or not), ontology, service
- At every layer, **productivity increases** (compound effect), the **need for testing decreases**, and performance starts identical vs. manual programming, ending up much higher (***abstraction benefit***) – "Complexity wall"

Image courtesy of Pexels from Pixabay

METASPEX

Ontology Complexity "Wall"- Hyper-abstraction benefits



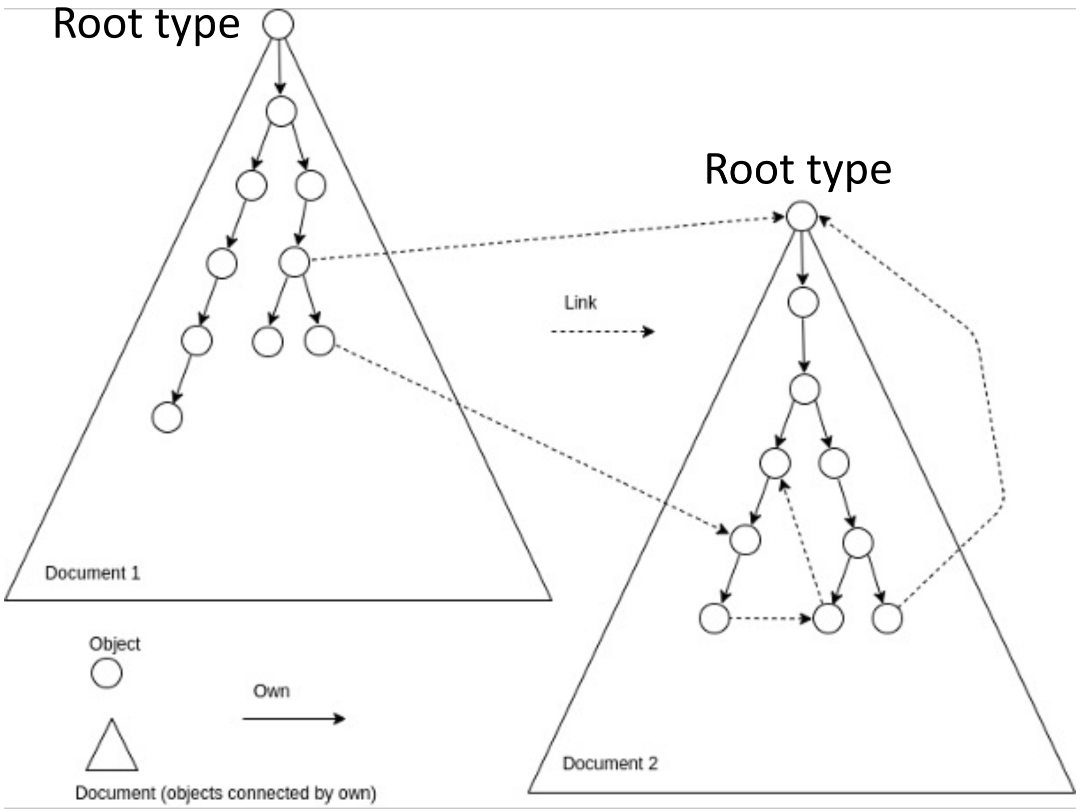


Our Specification-Level Abstractions in Numbers

METASPEX

- Implementation: ~**300 klocs** of layered template **abstractions**, offering:
  - A general-purpose **C++ application server** (HTTP REST JSON-based)
  - Tools to describe **high-definition ontologies** (types and their relationships), including **inter-document links**
  - **Automatic persistence** in document databases (MongoDB, Couchbase, CouchDB) described in **configuration files**
  - Etc.
- C++ specifications become **independent from Operating System**, **Web server** and **database**
- Lots of **reusable types** and ~**200 ready-made services** coming with the **Foundation Ontology**
- **Advanced security**
- **Concise**
  - **10 lines** to declare a **type that persists automatically**
  - **8 lines** to declare a **service creating a document** in a database
  - **1 line** to declare a **multi-dimensional index** (KD-Tree)
- **Efficient**: < **100 <u>microseconds</u>** of application tier total CPU time to **run a service** call **creating a document** in Couchbase (Ubuntu 20.04, conventional CPU)
- Let's see a few of our **high-level abstractions**

A simple document model: own<> and link<>



- Generic, **strongly typed**
- Similar to **HTML** with **URLs**
- **Links** possible **across databases**, database **products**, **datacenters**
- Integrates nicely with **document databases**
- **Simple**: complexity sealed inside
- More **finesse** in the ontology means **less complication later** on, **type as much as you can**
- All this comes with **integrity rules**

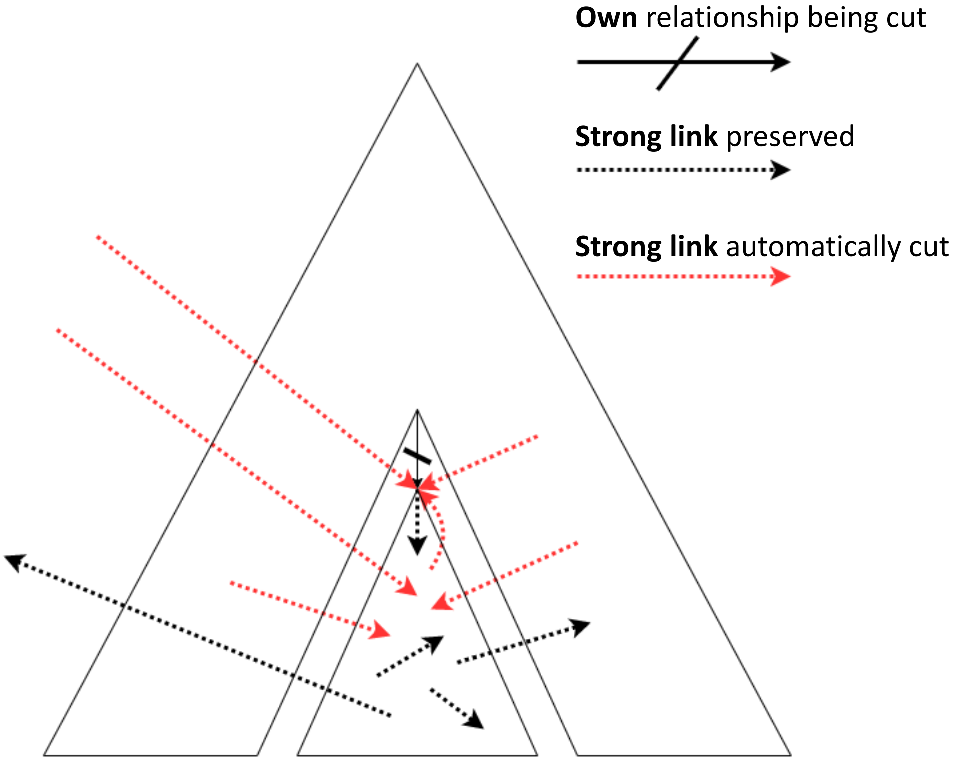
METASPEX

Embedded Referential Integrity Rules

- An object can **own** another one
- An object can be owned **at most by one** other object
- **Root** types cannot be owned (they have a UUID and are referenced in dictionaries)
- A **document** is the set of objects owned transitively by a root object
- An object can **own** another one only if the **first is in a document**
- An object can possess a **link** to another only if the **second is in a document**
- An object can be the target of **as many links as needed**
- **Links cycles are authorized**
- The system **automatically maintains these rules** when an ownership is broken

METASPEX

"Relaxed" Referential Integrity Automatic Maintenance



- Operates on **strong links**
- Weak links are merely broken
- Done in **cascade**
- **Logical operation**, no direct link with objects' lifespan
- **No SQL** running
- More sophisticated than RDBMS constraints

METASPEX

Example: an eCommerce Product

```
struct characteristics;
struct category;

struct product: public root<>
{
   HX2A_ROOT(product, "prod", 1, root);
   product(reserved_t): root(reserved),
      _chars(*this), _category(*this) {}

   own<characteristics, "ch"> _chars;
   strong_link<category, "ct"> _category;
};
```

Also: `weak_link`, `own_list`, `own_vector`, `link_list`, `link_vector` etc...

METASPEX

Zero-Cost Byproduct: Automatic Object Destruction

- All document roots are in **dictionaries**, held by **smart pointers** (reference counting)
- **Own** relationships rely on **smart pointers** (reference counting as well)
- **Links** contain **<u>only regular pointers</u>**
- Objects are **automatically** and **immediately destroyed**

Theorem: <u>Thanks to Referential Integrity, links never point at destroyed objects</u>

Better and… Free

Far better than **Garbage Collection**

- Traversals are guided, **limited** to specific documents/sub-documents
- Compatible with **real-time**, with **guarantees** (and **frugal**)

Better than **`std::shared_pointer`**, same guarantees

- More **frugal**, link loops allowed without deadlocks

Templates get compiled into **fast lightweight binaries**

See animation (Metaspex Channel)

https://www.youtube.com/watch?v=M-71t1Az-8c

Replacing SQL

We already have a language: **C++**

Object-oriented (can resolve SQL's cross cut maintenance headache issue)

Most powerful **metaprogramming model** in existence

Compiles into **lightweight** binary code

In the "domain of discourse", types and owns/links form the **syntax**, let's add **semantics**

METASPEX

Semantic Attributes Example

```
class contract: public root<>
{
   // …
   slot<time_t, "s"> _start;
   slot<uint32_t, "d"> _duration;

   time_t end(){
      return _start + _duration;
   }
   attribute<time_t, &contract::end, "e"> _end;
};
```

Bidirectional!

Their **values persist** and **indexes can be built** on them

METASPEX

Semantic Attributes

- Fairly **easy to read**, describe the *what* not the *how* (rules)
- Fast: run **compiled C++ functions** above layered abstractions, **including links**
- **Precalculated** and **lazily evaluated**
- **Object-oriented**, resilient to ontology updates (reusable)
- **Recalculated incrementally** and **minimally**
- Can calculate **documents over documents** (and etc.)
- **No SQL** running for **evaluation**
- Queries are **O(log(N))**, as they browse only an index

Service Creating a Persistent Document

```
class my_service: public basic_service<"my_service_name",
                                       name_payload>
{
  reply_p call(http_request&, const session_info*,
               const organization_p&, const user_p&,
               const rfr<name_payload>& p){
    db::connector c("hx2a");
    make_rfr<mytype>(p->name);
    return {};
  }
} _my_service;
```

One of the Foundation Ontology predefined payloads

Logical name mapped to a physical description in the configuration file

Compiles into a **Nginx** or **Apache** module, can interact with any supported database.

To call the service:

```
curl http://myhost/my_service_name –d '{"name": "mydocname"}'
```

METASPEX

Capitalizing on Ontologies & Abstractions

We can now capture complete statically-validated specification-level **domain ontologies**

They can be reused to produce quickly **entire** database and Web server-neutral **high-definition applications** using additional abstractions:

- General-purpose C++ Application Server (can be used with any database)
- Automatic Multitenancy
- Basic services
- Services with full credential checks
- Paginated services
- Offline/batch applications
- Persistent Queues
- Meta-Ontologies
- Multi-dimensional search (KD-trees)
- Strongly-typed database cursors
- Outgoing services (post, fanout…)
- Growing Foundation Ontology
- Full-fledge security
- Invertible relationships
- Etc.

METASPEX

High-Level Benefits

- **Time**
  - **Back-ends/offline applications in hours** instead of months/years
  - **Iterate** at a **rapid pace**
  - **Customize implementations** (including datamodel changes) quickly
- **Cost**
  - **Supercharge** engineering teams
  - Use **fewer hardware**
  - Embrace **gradually** and **effortlessly** the **most modern tech**
- **Value**
  - Develop **high-definition applications** too complex today
  - **Agility is high**, technology **never lags behind** business or industry mandates
  - Improved **security**
  - **Low latency**

Reusability Apache NGINX Application Server

Microservices

Ontologies

Cloud

Metaprogramming

Specification Level

SaaS

CouchDB

Object-Oriented DB

CouchBase™

Efficiency

CAP Theorem

METASPEX

Let the machine program

AES-256

PBKDF2

SHA-224

Unicode

SOA

Mersenne Twister

Multitenancy

Infinite Scalability

Resilience Quality Performance

JSON

Noumenon

Abstraction: the Essence of Programming
What it is, what it is not

- A **methodology** (automated or not) to build systems on top of a subsystem or several subsystems
  - E.g.: structured programming on top of assembly programming, assembly programming on top of processor microcode, algorithms on top of iterators on top of containers in the C++ STL, a Linux filesystem on top of a disk driver, a unicode iterator on top of raw strings, device drivers API making application code independent from various physical devices, etc.
- Abstracting is **not scripting** on top of the subsystems
  - It is not opening up a Turing-complete programming model on top of subsystems API
- Abstracting is **not automation** on top of subsystems
  - Abstractions do not generate the exact client code which would be written by hand if the abstraction did not exist. Abstractions are *opinionated*

Abstraction: the Essence of Programming
How to – Adopt a discipline to allow abstraction emergence

METASPEX

- Subsystems must expose **orthogonal APIs**
  - No function exposed in a subsystem API must be a combination of others
  - Otherwise it factors client code within its own layer, and prevents abstractions to emerge
  - C++ operators which are syntactic helpers are an exception
- Above subsystems: **factoring is not abstracting**
  - Factoring is mechanical, abstracting requires thinking and modeling
  - Factoring obscures the situation
  - Structured programming did not arise from factoring assembly code
  - Treat factoring with a lot of suspicion

Abstraction: the Essence of Programming

How to - Adopt a discipline to allow abstraction emergence

METASPEX

- **Do not fuse abstractions** together
  - Examples of such fusions are: integrity constraints on a relational database schema, which fuse together referential integrity and data lifespan, garbage collectors which fuse together expensive data traversals and objects lifespan, etc.
- Be clear on **subsystems control** and **document** how to use the subsystems API directly. These abilities can be complete **(transparent abstraction**), limited, or nonexistent (**opaque abstraction**)
  - If you remove a file in a Linux filesystem, you'll still see its contents if you read the disk byte per byte immediately after through the disk driver. Reading the disk byte per byte is not an operation allowed by the filesystem abstraction to access the content of files. It is allowed, though, to clone a disk
  - An abstraction must be carefully crafted so that a reasonably fine level of control can be obtained over the subsystems. The goal is not full control, but it is not to hamper control unreasonably either
- Abstractions must **not measurably affect performance**, offer a **higher productivity** and **quality** (reduce bugs)
  - Because of that, at scale they allow to develop systems impossible to develop without them (productivity boost pushing limits further, and offering more clarity)