In-memory and Persistent Representations of C++

Gabriel Dos Reis
Microsoft

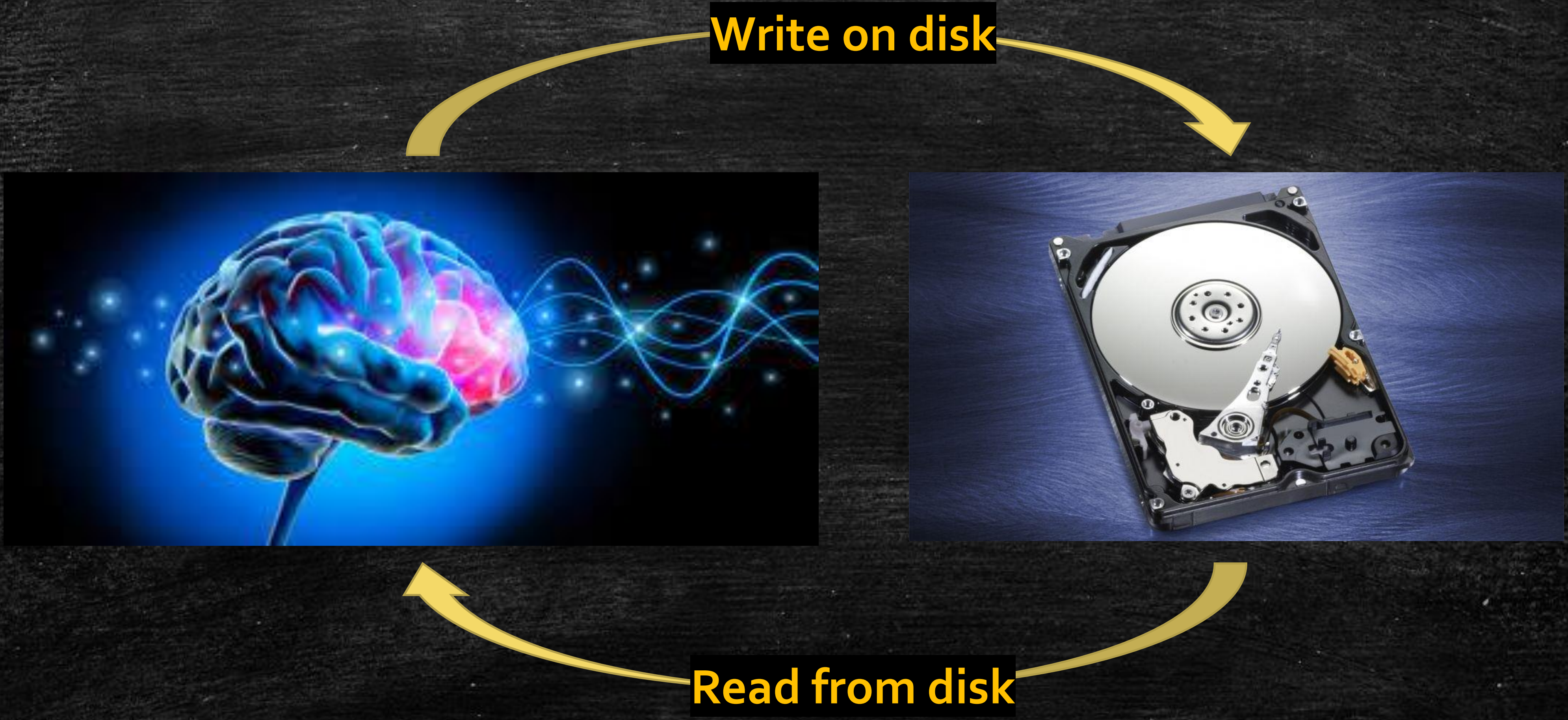
Overview

- What?
  - Democratize access to in-memory and on-disk representations of C++ programs
  - Facilities developed by the C++ community
- Why?
  - Readily produced by C++20 compilers (because C++ Modules), but thrown away
  - Need to access to higher level abstractions for effective use of C++
- How?
  - Aim for a general, regular, simple semantics model capable of expressing C++
  - Implementations of said semantics model by and for the C++ community

Problems

- High level abstractions often exposed through brittle, error-prone, low-level interfaces (e.g. `extern “C”`) to other languages because of lack of pervasive rich semantic representation and tools
- Lack of common representation tends to push to vendor lock-in solutions for semantics-based analysis and transformation tools
- High risk of fragmentation: N compilers with incompatible representations lead to N incompatible views over the same program
- Source file as sequence of characters not always the best representation of C++ programs
- …

Proposal: Complete and efficient representations of C++

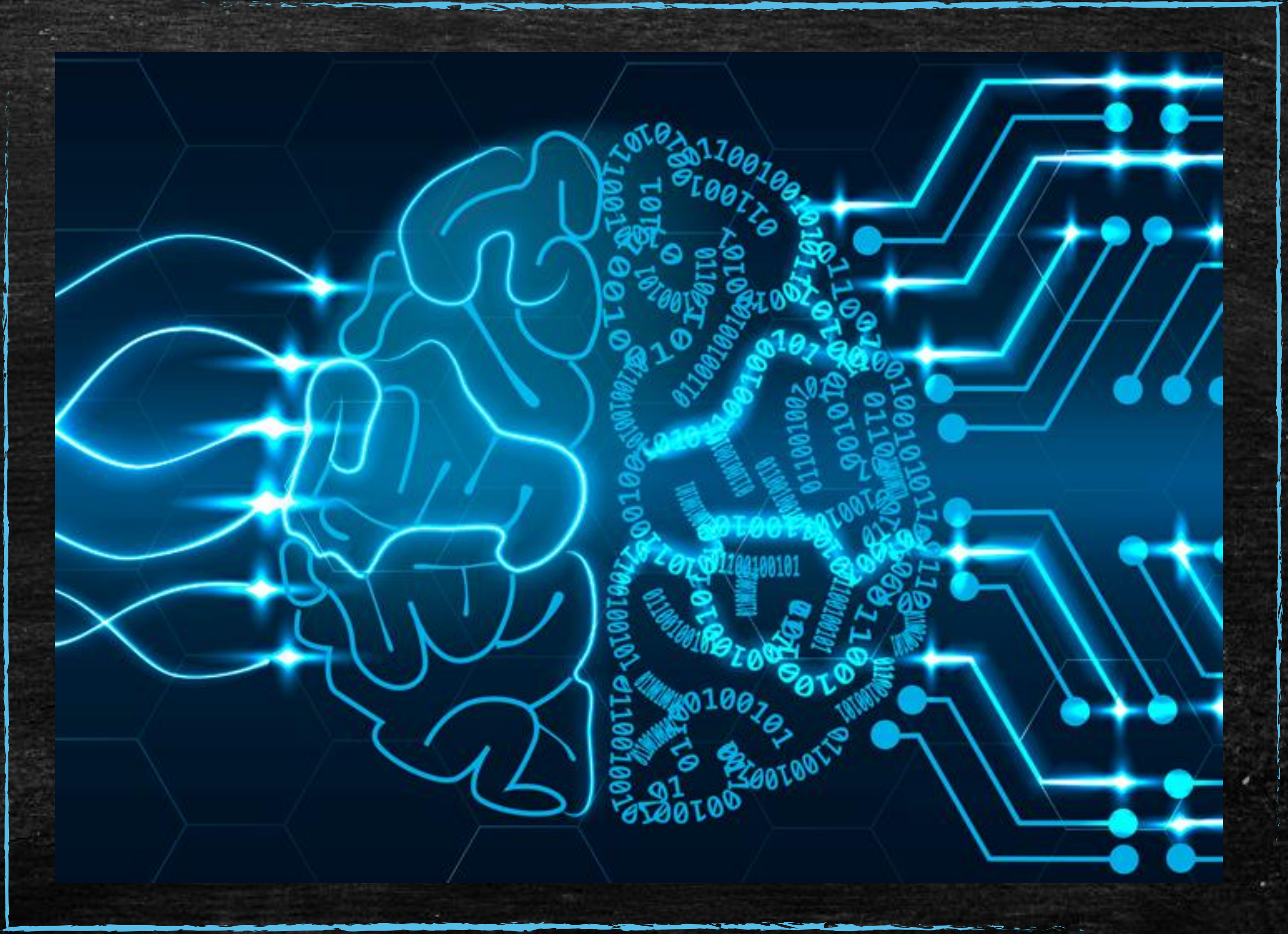


Write on disk

Read from disk

Proposal: In-memory and persistent representations based on same principles

Similar concepts used for the in-memory and the persistent representations, offering the same programming model



Early Design Choices

- Abstract away from concrete syntax
  - Provide semantic resolution mapping
- Focus on semantics, capturing the higher level of C++ abstractions
  - The essence of C++, before machine code generation
- Avoid mimicking ISO C++ tortuous irregularities
  - The semantics are encrypted in hundreds of pages of obscure standardese
- Provide framework for representing vendor extensions
  - Necessary for real world uses
  - Often, vendor extensions follow no discernable semantics pattern

ISO C++ Grammar: https://eel.is/c++draft/ as of 2021-10-25

- Annex A, Grammar summary: **347** non-terminals, and counting…

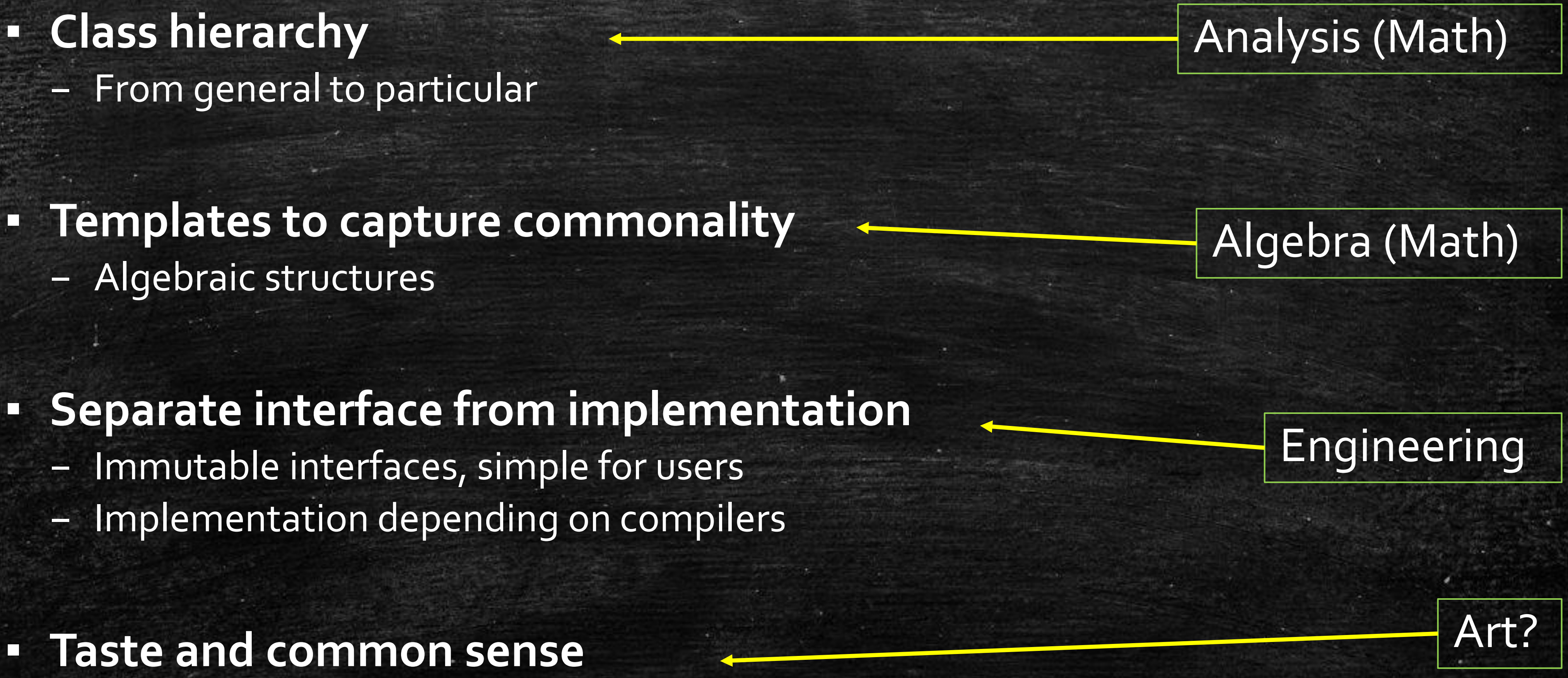
1. Keywords: 6
2. Lexical conventions: 76
3. Basics: 1
4. Expressions: 60
5. Statements: 14
6. Declarations: 97
7. Modules: 8
8. Classes: 25
9. Overloading: 3
10. Templates: 22
11. Exception handling: 6
12. Preprocessing directives: 29

IPR: In-memory Representation

Design Principles of IPR (2004)

- **Completeness**: represents all Standard C++ constructs, but not macros before expansions
- **Generality**: suitable for every kind of application, rather than targeted to a particular application area
- **Regularity**: does not mimic C++ language irregularities; general rules used, rather than long lists of special cases
- **Typefulness**: every expression has a type
- **Minimality**: has no redundant values, traversal involves no redundant indirections
- **Compiler neutrality**: not tied to a particular compiler
- **Scalability**: able to handle hundreds of thousands of lines of code on common machines

IPR Design and Implementation Methodology



- **Class hierarchy**
  - From general to particular

Analysis (Math)

- **Templates to capture commonality**
  - Algebraic structures

Algebra (Math)

- **Separate interface from implementation**
  - Immutable interfaces, simple for users
  - Implementation depending on compilers

Engineering

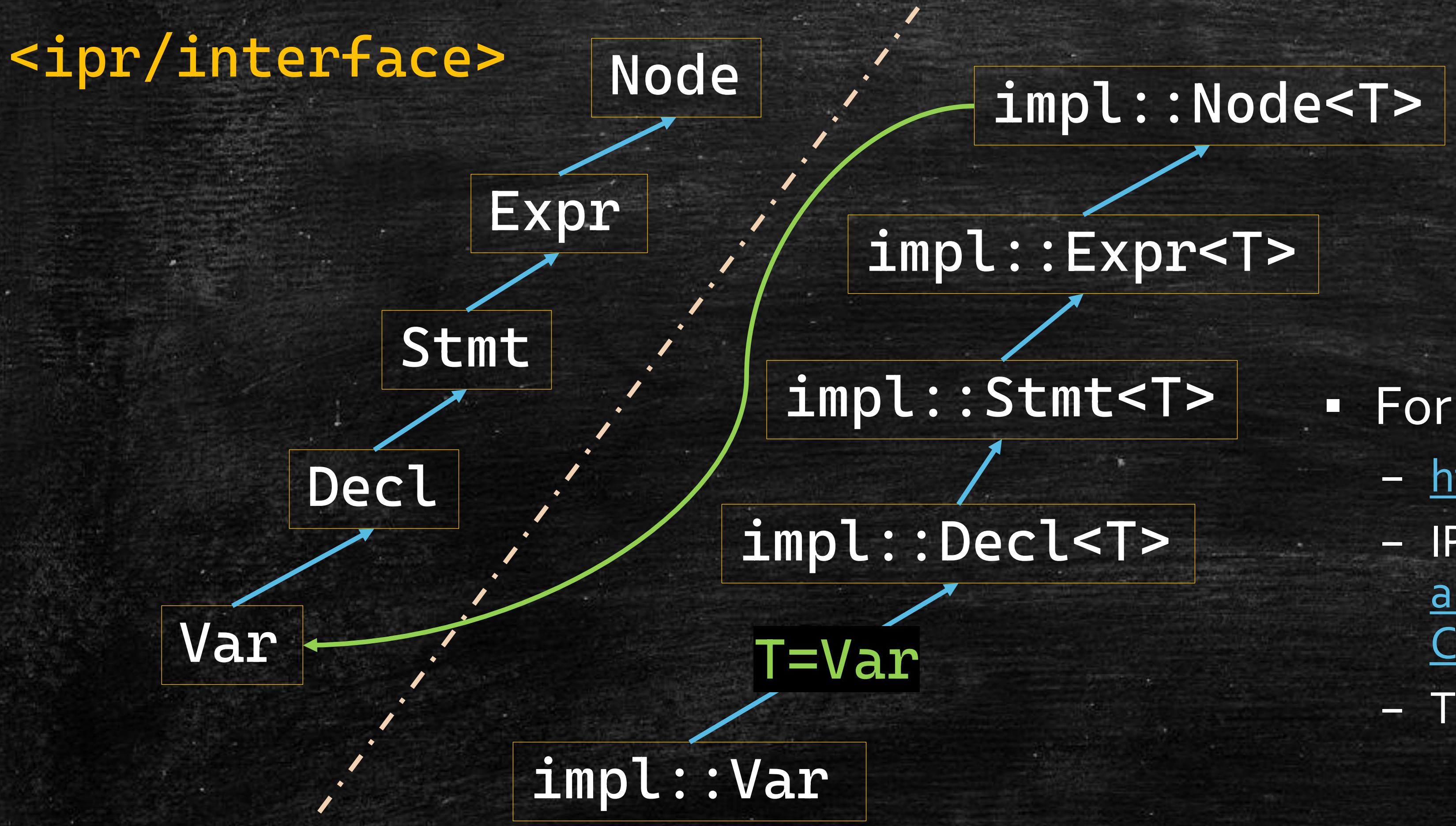
- **Taste and common sense**

Art?

IPR Implementation

- Open source library
  - MIT license (3-clause)
  - Available at https://github.com/GabrielDosReis/ipr
- Designed to be minimal in representation space and traversal time
- Uses Visitor Design Pattern for traversal
- Relatively small
- Need help to connect to other compilers
  - Originally generated from GCC (15+ years ago)
  - Generation from Clang and EDG lost

IPR Interface and Implementation



- For details, see
  - https://github/GabrielDosReis/ipr
  - IPR paper: "A Principled, Complete, and Efficient Representation of C++", MACIS 2009
  - TC++PL4

(C) Gabriel Dos Reis; CppCon 2021

Persistent Forms

Design Principles: Same as for the IPR

- Completeness
- Generality
- Regularity
- Typefulness
- Minimality
- Compiler neutrality
- Scalability

Early Design Choices

- Abstract away from concrete syntax
  - Provide semantic resolution mapping
- Focus on semantics, capturing the higher level of C++ abstractions
  - The essence of C++, before machine code generation
- Avoid mimicking ISO C++ tortuous irregularities
  - The semantics are encrypted in hundreds of pages of obscure standardese
- Provide framework for representing vendor extensions
  - Necessary for real world uses
  - Often, vendor extensions follow no discernable semantics pattern

Design and Implementation Methodology

IPR

- Class hierarchy
- Templates to capture commonality
- Interface separate from implementation
- Taste and common sense

Design and Implementation Methodology

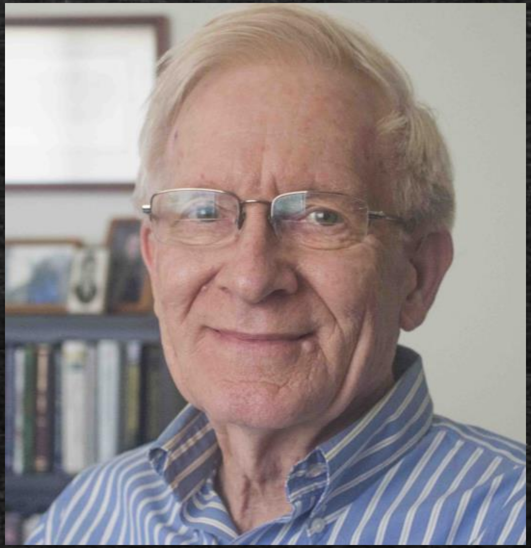
- **Principled description of data structures**
  - Universal Algebra
- **Multi-sorted algebras to surface commonality**
  - Universal Algebra
- **Gradual lifting**
  - Generic Programming at la Stepanov
- Taste and common sense

IPR

- Class hierarchy
- Templates to capture commonality
- Interface separate from implementation
- Taste and common sense

Generic Programming: Gradual Lifting

- *"Generic programming centers around the idea of **abstracting from concrete, efficient algorithms** to obtain generic algorithms that can be **combined with different data representations** to produce a wide variety of useful software."*
  -- Musser & Stepanov (ISSAC, 1988)



Gradual Lifting Illustration on find()

```
ListNode* find(ListNode* first, T t)
{
    for (ListNode* cur = first; cur != nullptr; cur = cur->next)
        if (cur->data == t) return cur;
    return nullptr;
}
```

```
int find(vector<T>& v, T t)
{
    for (int i = 0; i < v.size(); ++i)
        if (v[i] == t) return i;
    return -1;
}
```

Gradual Lifting [find]: when to stop

```
ListNode* find(ListNode* first, T t)
{
    for (ListNode* cur = first; cur != nullptr; cur = cur->next)
        if (cur->data == t) return cur;
    return nullptr;
}
```

```
int find(vector<T>& v, T t)
{
    for (int i = 0; i < v.size(); ++i)
        if (v[i] == t) return i;
    return -1;
}
```

Gradual Lifting [find]: Next!

```
ListNode* find(ListNode* first, T t)
{
    for (ListNode* cur = first; cur != nullptr; cur = cur->next)
        if (cur->data == t) return cur;
    return nullptr;
}
```

```
int find(vector<T>& v, T t)
{
    for (int i = 0; i < v.size(); ++i)
        if (v[i] == t) return i;
    return -1;
}
```

Gradual Lifting [find]: Peek

```
ListNode* find(ListNode* first, T t)
{
    for (ListNode* cur = first; cur != nullptr; cur = cur->next)
        if (cur->data == t) return cur;
    return nullptr;
}
```

```
int find(vector<T>& v, T t)
{
    for (int i = 0; i < v.size(); ++i)
        if (v[i] == t) return i;
    return -1;
}
```

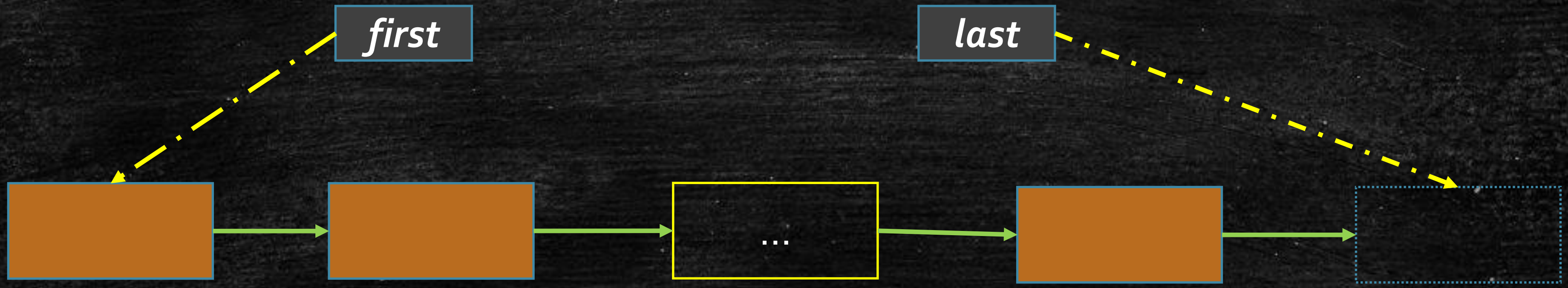
Gradual Lifting [find]: missing position

```
ListNode* find(ListNode* first, T t)
{
    for (ListNode* cur = first; cur != nullptr; cur = cur->next)
        if (cur->data == t) return cur;
    return nullptr;
}
```

```
int find(vector<T>& v, T t)
{
    for (int i = 0; i < v.size(); ++i)
        if (v[i] == t) return i;
    return -1;
}
```

Gradual Lifting [find]: Iterators

- Abstraction to delimit a ***range*** of “cells” in a container (e.g. list, vector, etc.)
- A range is conceptually a half open interval
  - Pair of iterators **[*first*, *last*)**
  - Starting iterator ***first***, and a number ***n*** of consecutive cells
  - Starting iterator ***first***, and a predicate ***p*** indicating the last



Gradual Lifting [find]: with iterators!

```
template<typename Iter>
Iter find(Iter first, Iter last, T t)
{
    for (auto cur = first; cur != last; ++cur)
        if (*cur == t) return cur;
    return last;
}
```

- Iterator is a *concept*!
  - Requirements: operations, algorithmic complexity, assumptions
    - **++, *, ->, ==**
- There are many different data structures that satisfy a given iterator concept

Apply Gradual Lifting to Persistent Form

1. Start with concrete efficient persistent representation
   - Build Module Interfaces (BMI) needed to implement C++ Modules
2. Abstract implementation details
   - Find good mapping for compiler internal specifics to general notions
3. Combine with different C++ compilers
   - Every toolset should provide one

- Help needed
  - For steps (2) and (3)
- The C++ algebras are the analogous of concepts

The Algebra of C++ Programs

1. **Decl** = ⑤ + ② + ③ + ⑦ + ① // algebra of declarations
2. **Type** = ② + ③ // algebra of types
3. **Expr** = ① + ② + ③ + ④ // algebra of expressions
4. **Stmt** = ① + ③ + ④ + ⑦ // algebra of statements
5. **Name** = ② + ⑧ // algebra of names
6. **Dir** = ③ + ⑤ + ⑦ // algebra of directives
7. **Attr** = ⑥ + ③ // algebra of attributes
8. **Token** // algebra of tokens

Module Interface Build Artifacts

What Are They?

- Produced by compilers when compiling a module interface unit or a header units
- Reused to avoid reparsing/recompiling the same thing over and over again
- Encapsulates the semantics of a module interface unit at the point of its processing
- Compiler independent

What Are They Useful for?

- Offer the same semantic information contained in an interface unit/header unit
  - But require NO C++ compiler to process – no name lookup, no overload resolution, etc.

- Offer "mass" direct access to higher level of C++ representation without intervention of C++ front-end

- Can be used for generating other devtools metadata
  - Safe runtime introspection
  - Typesafe distributed computation (marshalling)
  - Easy interoperability between the higher levels of C++ and other systems

Tooling Opportunities

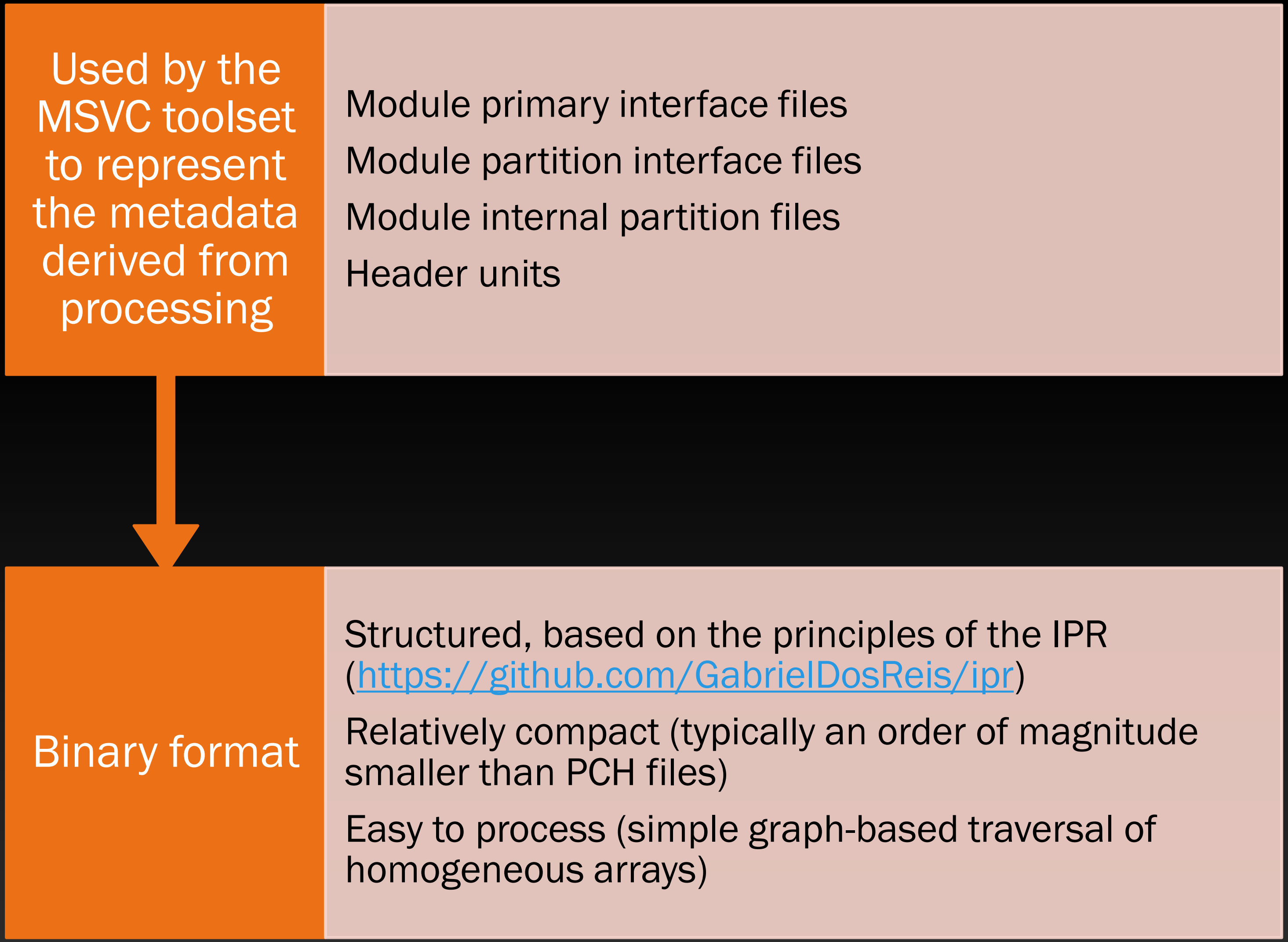
- See my WG21 paper: Modules Are a Tooling Opportunity (P0822R0)

Open IFC Specification

IFC Specification now open

- Open to the C++ community
  - License: Creative Commons Attribution 4.0 International (CC-BY-4.0)
- Available at https://github.com/microsoft/ifc-spec
- Use as you see fit
  - With credit
- Contribute!
  - Help evolve it

IFC file



Used by the MSVC toolset to represent the metadata derived from processing

- Module primary interface files
- Module partition interface files
- Module internal partition files
- Header units

Binary format

- Structured, based on the principles of the IPR (https://github.com/GabrielDosReis/ipr)
- Relatively compact (typically an order of magnitude smaller than PCH files)
- Easy to process (simple graph-based traversal of homogeneous arrays)

What Is in an IFC File?

1. Declarations
2. Types
3. Statements
4. Expressions
5. Names
6. Charts
7. Scopes

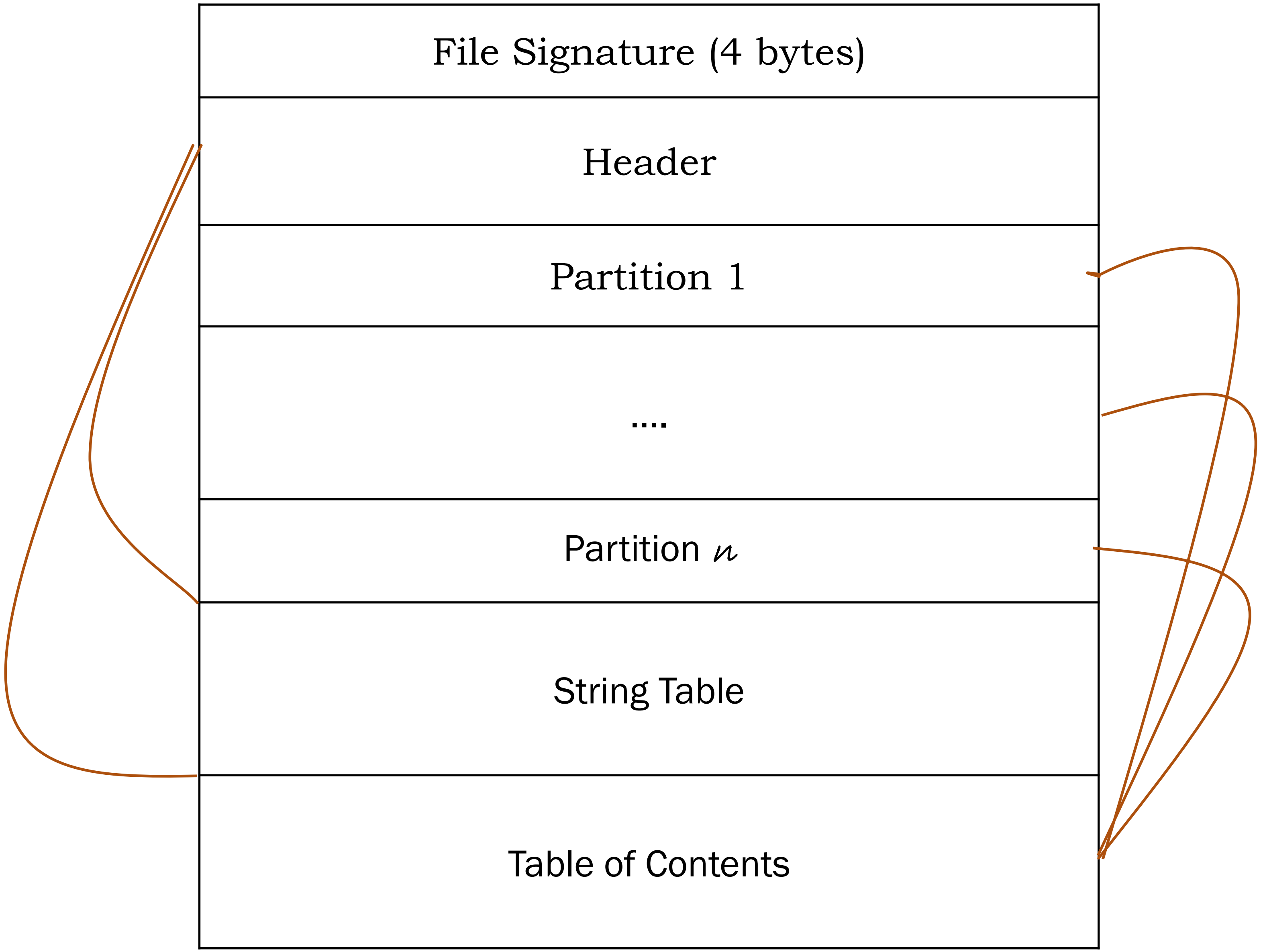
What Is in an IFC File?

A. String table
B. Source locations
C. Token streams
D. Preprocessing forms
E. Heaps
F. Lookaside tables
G. Other compiler data structures

Representation Design Choices

- All of ISO C++ (and beyond) is representable
- Entities are represented by structures in ***partitions*** (homogeneous arrays)
- Entities are referenced by ***abstract references*** (i.e. typed indices)
- An abstract reference is a 32-bit value with two components
  - ➢ A ***tag*** that designates the partition the reference points into
  - ➢ An ***index*** that designates the position of the entity in that partition
- Semantically-similar tags are grouped into ***sort***s
  - Example: `DeclSort`, `TypeSort`, `ExprSort`, `NameSort`, etc.

IFC File



File Signature (4 bytes)

Header

Partition 1

....

Partition *n*

String Table

Table of Contents

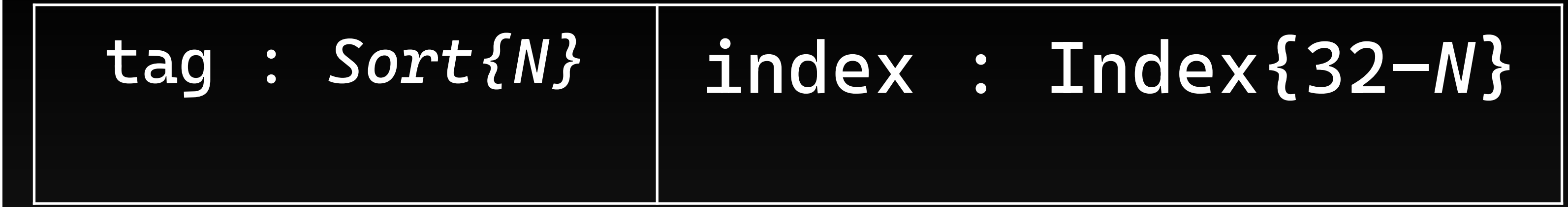
IFC Header

checksum : SHA256	u256
major_version : Version	u8
minor_version : Version	u8
abi : Abi	u8
arch : Architecture	u8
dialect : LanguageVersion	u32
string_table.bytes : ByteOffset	u32
string_table.size : Cardinality	u32
unit : UnitIndex	u32
src_path : TextOffset	u32
global_scope : ScopeIndex	u32
toc : ByteOffset	u32
partition_count : Cardinality	u32
internal : u8	u8

IFC Partition

`name : TextOffset`	u32
`offset : ByteOffset`	u32
`cardinality : Cardinality`	u32
`entry_size : EntitySize`	u32

Abstract Reference



`tag : `*`Sort{N}`*	`index : Index{32-`*`N`*`}`

A Zoo of References

- Declarations : `DeclIndex`
- Types: `TypeIndex`
- Expressions: `ExprIndex`
- Names: `NameIndex`
- Statements: `StmtIndex`
- Charts: `ChartIndex`
- Strings: `StringIndex`
- Preprocessing forms: `FormIndex`

- Scopes: `ScopeIndex`
- Text: `TextOffset`
- Words : `WordIndex`
- Sentences: `SentenceIndex`

Where To from Here?

- Draft specification released to the C++ community
  - The entire C++ community invited to collaborate *for* the C++ community
  - Inputs from C++ devtools implementers, WG21/SG15

- Development in the open of set of APIs over IFC files
  - Open source libraries constructing IPR from IFC files and vice versa

- IFC-powered devtools for semantics-based analysis and program transformations for C++
  - Community-driven

Summary

Call to Action

- **Get the IPR library, OSS under MIT (3-clause) license**

  ```
  git clone https://github.com/GabrielDosReis/ipr.git
  ```

  - Provide feedback, comments, contribute!

- **Get the IFC specification, open documentation under CC-BY-4.0**

  ```
  git clone https://github.com/microsoft/ifc-spec.git
  ```

  - Provide feedback, comments, contribute!

- **Help needed!!!**
  - Talk to your C++ devtools providers about the IFC format
  - Help me help the C++ community move to less token-oriented world

?