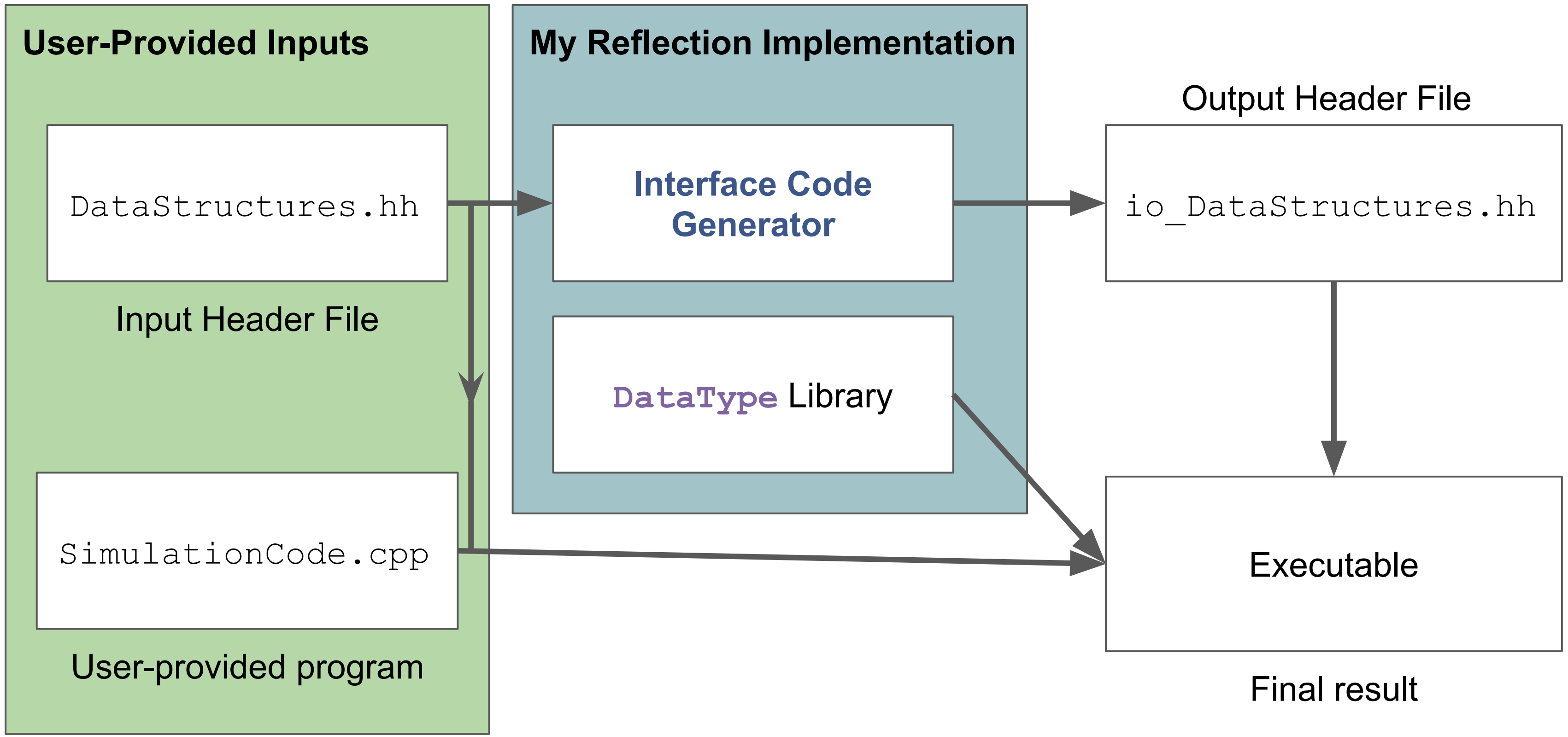
ICG 2

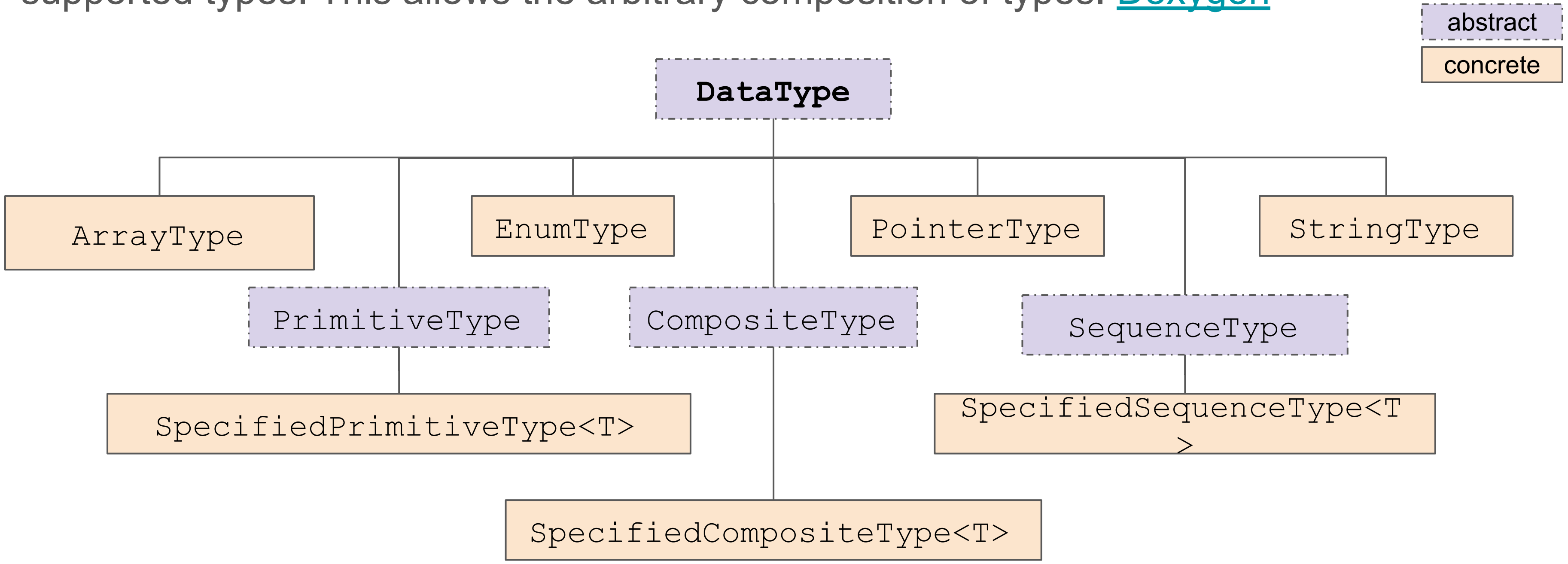
Electric Boogaloo
Design Review

High Level Architecture



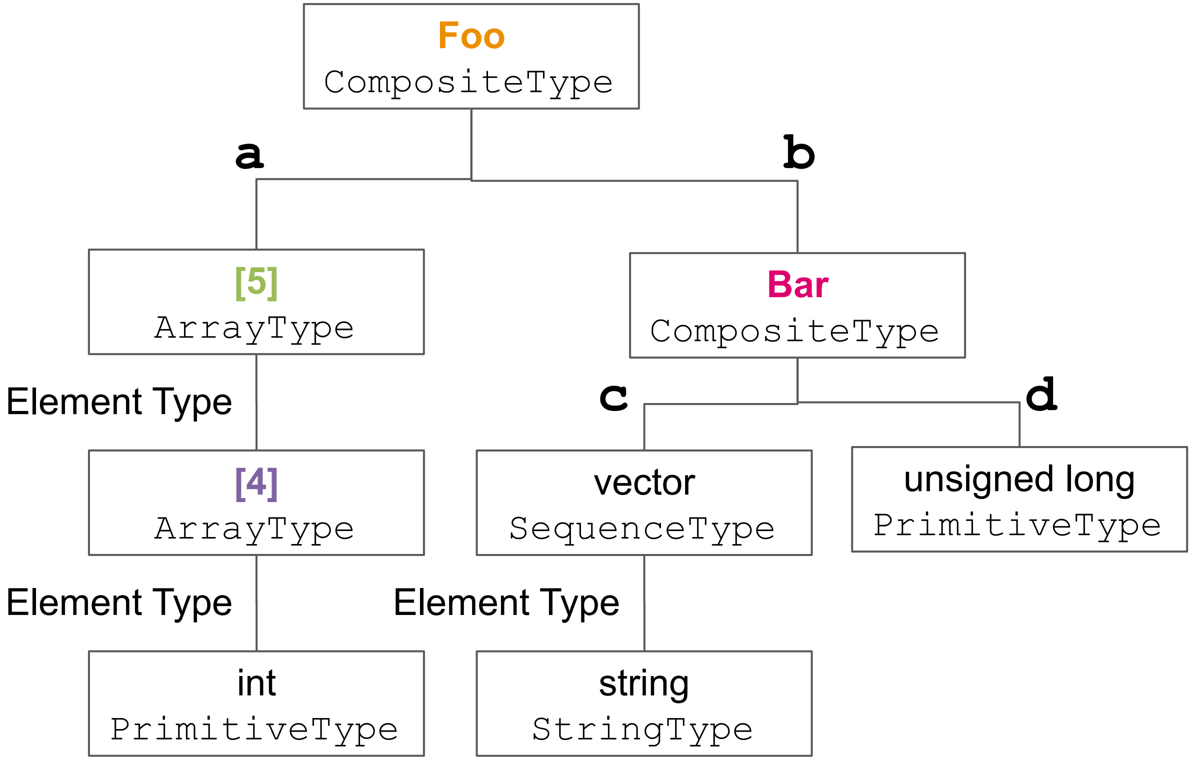
Type Representation

DataType is the base class of a hierarchy of types that can be used to represent all supported types. This allows the arbitrary composition of types. Doxygen



Type representation with composition

```
class Foo {
    int a[5][4];
    Bar b;
};

class Bar {
    vector<string> c;
    unsigned long d;
};
```



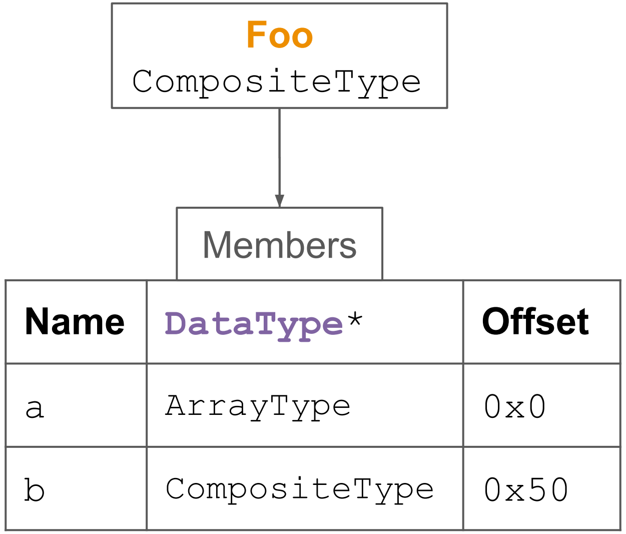
What’s in a `DataType`?

Each `DataType` class has:

- Total size of type in bytes
- String representation of the type name
- Allocator and deallocator

Subclasses hold specific information about their own structure

- `CompositeDataType` has a map of its members
- `ArrayDataType` has information about its elements
- Types that represent a literal (primitives, strings) have the ability to get or set a value at a given address



Name	DataType*	Offset
a	ArrayType	0x0
b	CompositeType	0x50

Managing `DataTypes` at runtime

The `DataTypeInator` keeps track of types at runtime

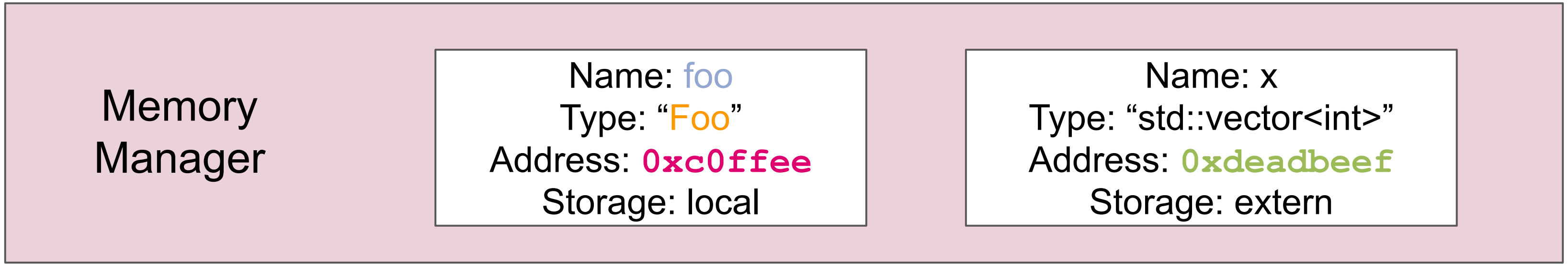
- `DataTypes` are registered and validated at initialization time
  - New pointer and array types can be created during runtime
- provide the transformation from `string -> DataType*` at runtime
- Handles parsing, namespaces, typedef statements

“int”	`SpecifiedPrimitiveType<int>`
“unsigned long”	`SpecifiedPrimitiveType<unsigned long>`
“std::string”	`StringType`
“Foo”	`SpecifiedCompositeType<Foo>`
“Bar[5]”	`ArrayType`
“vector<string>”	`SpecifiedSequenceType<vector<string>>`

Memory Management

- This is as similar to the current Trick interface that I can manage
- Keep track of all allocated data structures' location, type, and name
  - Use the `DataTypeInator` to get `DataType` from user provided string
- Delete local allocs in destructor, leave extern allocs alone

```
Foo * f_ptr = memoryManager->declare_var(“Foo foo”);

memoryManager->declare_var(“std::vector<int> x”, &x);
```



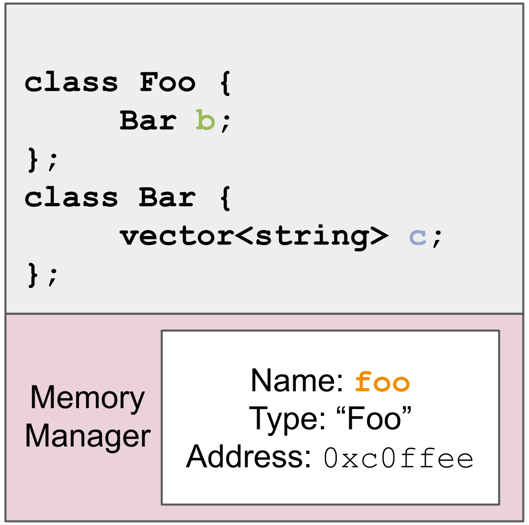
Name: foo
Type: “Foo”
Address: 0xc0ffee
Storage: local

Name: x
Type: “std::vector<int>”
Address: 0xdeadbeef
Storage: extern

doxygen code

Assignment from string using `DataTypes`

```
"foo.b.c[5] = \"Hello Trick!\""
```



```
class Foo {
      Bar b;
};
class Bar {
      vector<string> c;
};
```

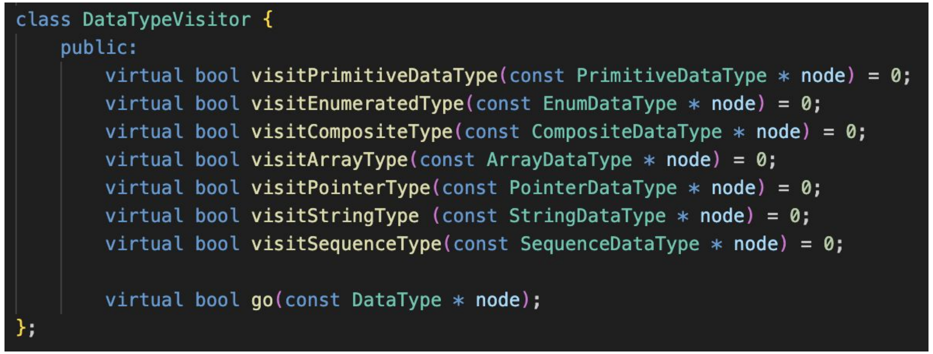
Steps: code

1. Parse assignment string code
2. Get the allocation named "`foo`" from the memory code manager, along with its starting address and `DataType`
3. Traverse the composite tree of the **`Foo`** `DataType` using remaining name parts `"b"`, `"c"`, `"[5]"` to calculate the address of `foo.b.c[5]` code
4. Assign the string literal **`"Hello Trick!"`** to the string object at that calculated address code

Traversing `DataTypes` without `dynamic_cast`

Use a visitor pattern with double dispatch to:

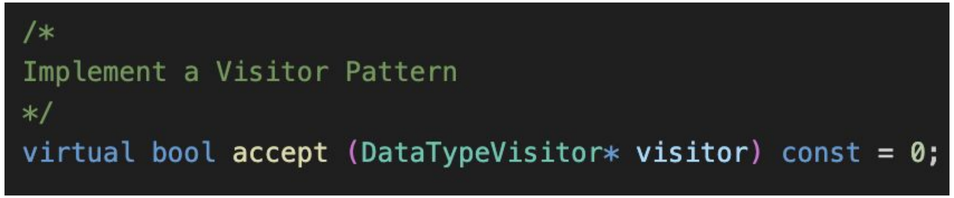
- Avoid bloating the `DataType` classes and/or using slow dynamic casting
- Provide a way to write arbitrary algorithms with awareness of which subclass of `DataType` they are operating on



```
class DataTypeVisitor {
    public:
        virtual bool visitPrimitiveDataType(const PrimitiveDataType * node) = 0;
        virtual bool visitEnumeratedType(const EnumDataType * node) = 0;
        virtual bool visitCompositeType(const CompositeDataType * node) = 0;
        virtual bool visitArrayType(const ArrayDataType * node) = 0;
        virtual bool visitPointerType(const PointerDataType * node) = 0;
        virtual bool visitStringType (const StringDataType * node) = 0;
        virtual bool visitSequenceType(const SequenceDataType * node) = 0;

        virtual bool go(const DataType * node);
};
```

Doxygen, code

In `DataType` base class



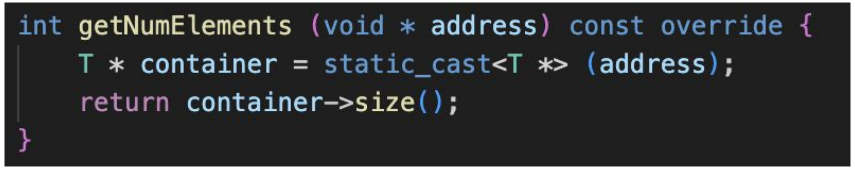
```
/*
Implement a Visitor Pattern
*/
virtual bool accept (DataTypeVisitor* visitor) const = 0;
```

Implementing `DataType` Algorithms

- Entry points are in `DataTypeAlgorithms.hpp` doxygen code
- Visitor algorithms can do many things -
  - Search the tree
    - LookupNameByAddress
    - LookupAddressByName
  - Traverse the entire tree
    - FindLeaves
    - PrintValue
  - Apply an operation to target type(s) and ignore all others
    - GetValue
    - AssignValue
    - ResizeSequence

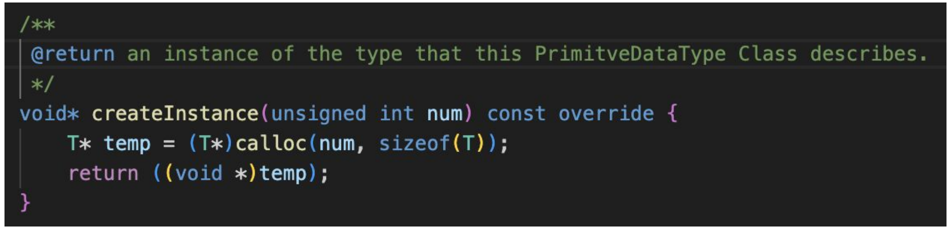
Type representation with Templates

Some **DataTypes** must take the class that is being represented as a template parameter to implement some operations. C++ does not allow for a virtual templated function, so they must inherit from a non-templated class that defines the specialized interface.



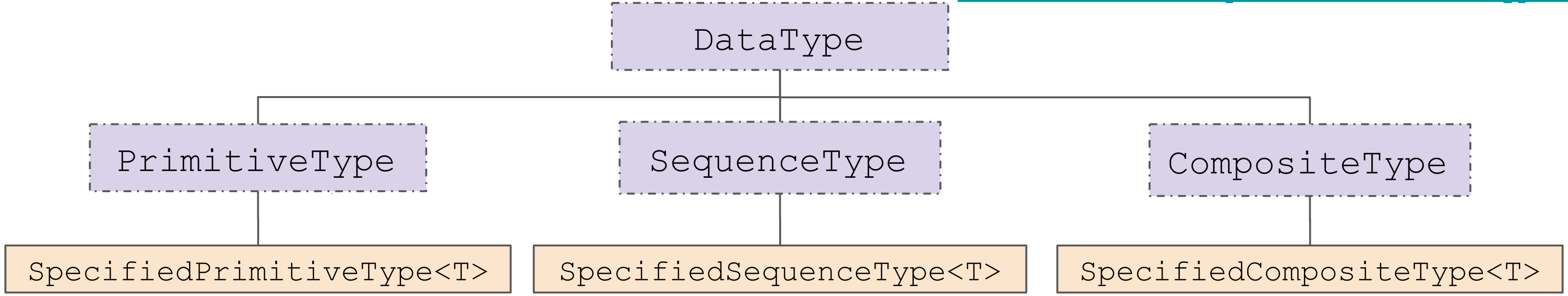
```
int getNumElements (void * address) const override {
    T * container = static_cast<T *> (address);
    return container->size();
}
```

getNumElements in SpecifiedSequenceType<T>



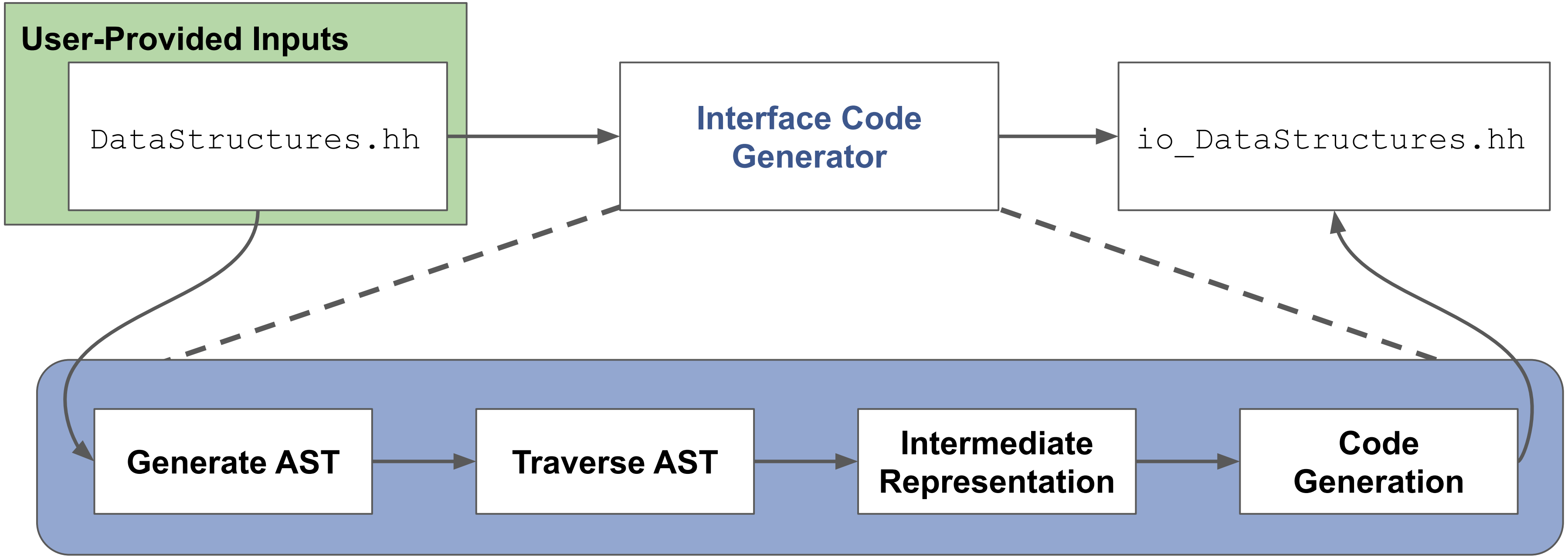
```
/**
 @return an instance of the type that this PrimitveDataType Class describes.
 */
void* createInstance(unsigned int num) const override {
    T* temp = (T*)calloc(num, sizeof(T));
    return ((void *)temp);
}
```

createInstance in SpecifiedPrimitiveType<T>



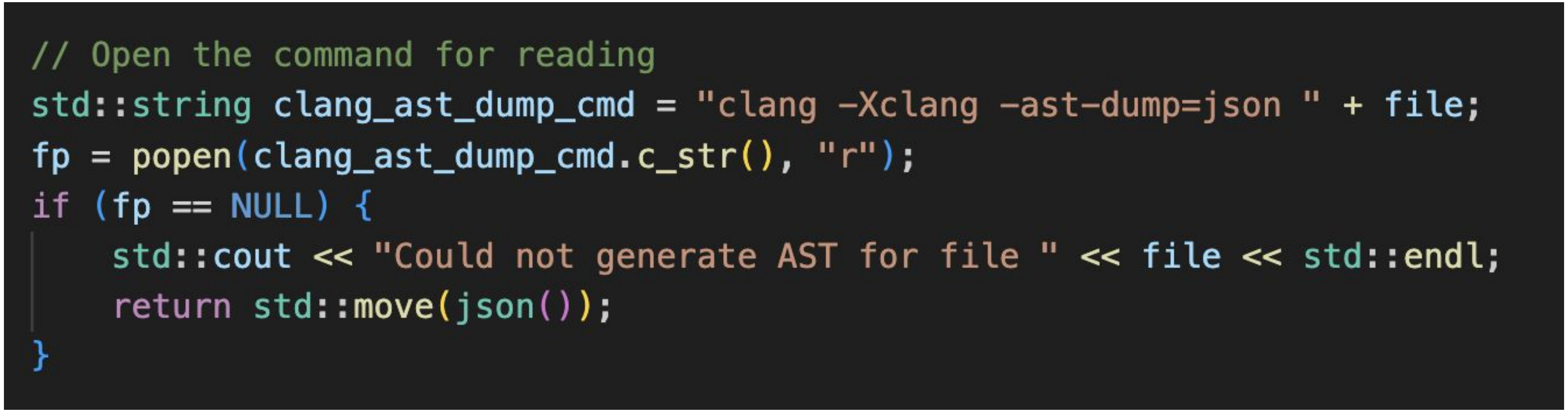
Interface Code Generator

This overarching structure is the same as Alex’s current ICG



AST Generation

- LibClang has been ditched!
- Instead, use normal Clang to dump an AST to json
- This is somewhat experimental and hacky



```
// Open the command for reading
std::string clang_ast_dump_cmd = "clang -Xclang -ast-dump=json " + file;
fp = popen(clang_ast_dump_cmd.c_str(), "r");
if (fp == NULL) {
    std::cout << "Could not generate AST for file " << file << std::endl;
    return std::move(json());
}
```

Code

AST Filtering

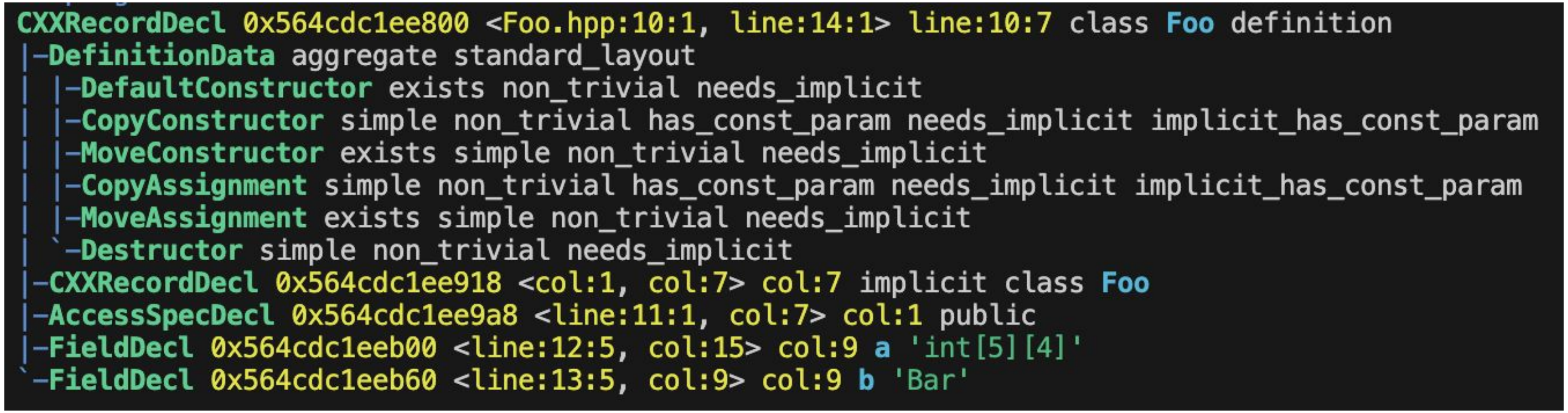
- This AST includes all system headers
- Filter it manually
  - Get the system header paths by running **`clang -### <filename>`** code
  - Prune AST json tree by chopping off branches that are located in those paths code

Note - the existence of the SourceManager class means that maybe instead of doing the filtering upfront, we could just prune the traversal like LibClang and Alex’s code does. I don’t think it matters much.

Filtering utility is available as a standalone tool here. It is useful for debugging.

Traversing the Abstract Syntax Tree

At this stage of compilation, the compiler has already done things that are hard, like pulled in all includes, resolved macros, figured out fully qualified names, and generated class template instantiations

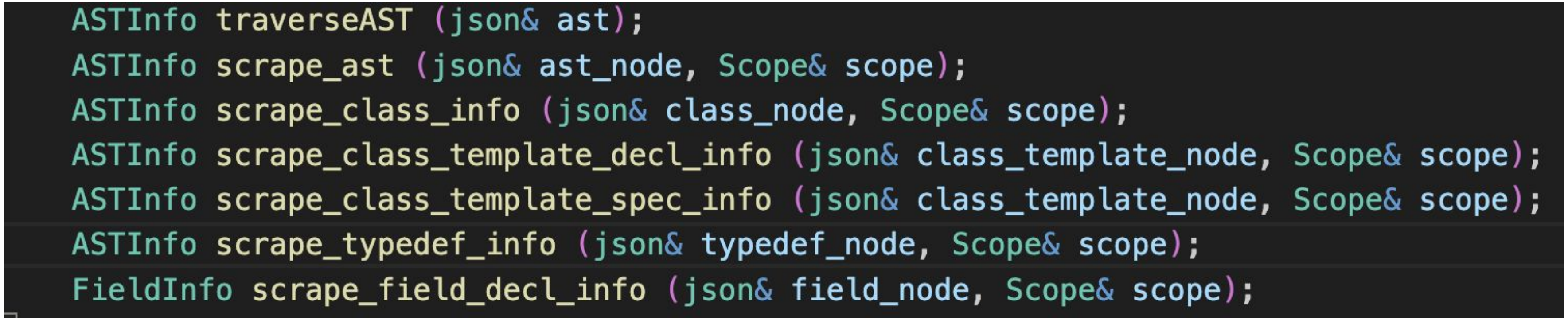


```
CXXRecordDecl 0x564cdc1ee800 <Foo.hpp:10:1, line:14:1> line:10:7 class Foo definition
|-DefinitionData aggregate standard_layout
| |-DefaultConstructor exists non_trivial needs_implicit
| |-CopyConstructor simple non_trivial has_const_param needs_implicit implicit_has_const_param
| |-MoveConstructor exists simple non_trivial needs_implicit
| |-CopyAssignment simple non_trivial has_const_param needs_implicit implicit_has_const_param
| |-MoveAssignment exists simple non_trivial needs_implicit
| `-Destructor simple non_trivial needs_implicit
|-CXXRecordDecl 0x564cdc1ee918 <col:1, col:7> col:7 implicit class Foo
|-AccessSpecDecl 0x564cdc1ee9a8 <line:11:1, col:7> col:1 public
|-FieldDecl 0x564cdc1eeb00 <line:12:5, col:15> col:9 a 'int[5][4]'
`-FieldDecl 0x564cdc1eeb60 <line:13:5, col:9> col:9 b 'Bar'
```

AST Traversal

JClang doxygen code

- I’ve written some utilities to make interacting with the Json AST easier
- I want to ensure that the entire Json AST is exposed, to prevent issues like the LibClang and template specialization problem from happening
- Implements a traversal model close to what LibClang supports

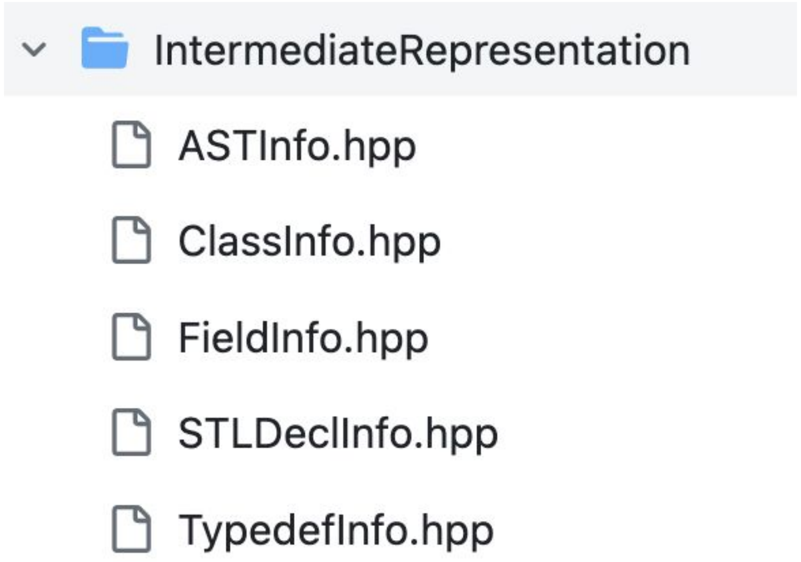


```
ASTInfo traverseAST (json& ast);
ASTInfo scrape_ast (json& ast_node, Scope& scope);
ASTInfo scrape_class_info (json& class_node, Scope& scope);
ASTInfo scrape_class_template_decl_info (json& class_template_node, Scope& scope);
ASTInfo scrape_class_template_spec_info (json& class_template_node, Scope& scope);
ASTInfo scrape_typedef_info (json& typedef_node, Scope& scope);
FieldInfo scrape_field_decl_info (json& field_node, Scope& scope);
```

Intermediate Representation

Traverse the AST to scrape intermediate representations of:

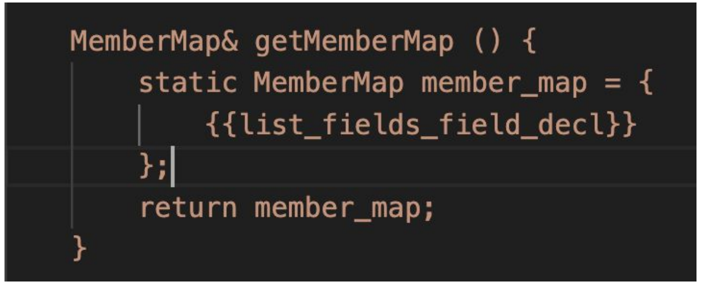
- Classes
  - Includes field and base classes
- Field
  - Save name and type
- Class Templates
  - Only pay attention to specializations, so really just treat these as normal classes with funky names
- TypeDefs
  - Need alias name -> full canonical name
- STL
  - STLs must be defined specially, so we need to scrape all types of STLs used



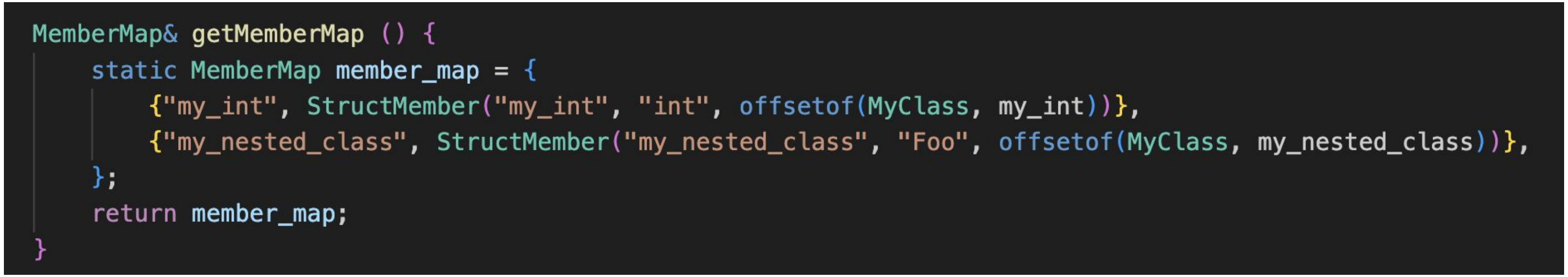
IntermediateRepresentation
- ASTInfo.hpp
- ClassInfo.hpp
- FieldInfo.hpp
- STLDeclInfo.hpp
- TypedefInfo.hpp

Code generation method

Use template engine (similar in concept to mustache, not C++ templates) to perform source code generation. Hardcoded strings have the expected structure, and they are filled in by the information scraped from the AST



```
MemberMap& getMemberMap () {
    static MemberMap member_map = {
        {{list_fields_field_decl}}
    };
    return member_map;
}
```



```
MemberMap& getMemberMap () {
    static MemberMap member_map = {
        {"my_int", StructMember("my_int", "int", offsetof(MyClass, my_int))},
        {"my_nested_class", StructMember("my_nested_class", "Foo", offsetof(MyClass, my_nested_class))},
    };
    return member_map;
}
```

Special Considerations

CompositeTypes

Composite types use a new design that takes advantage of template specialization!

Class Definition (user provided)

```
class Foo {
    int a[5][4];
    Bar b;
};
```

Foo
MemberMap

Name	DataType	Offset
a	int[5][4]	0x0
b	Bar	0x50

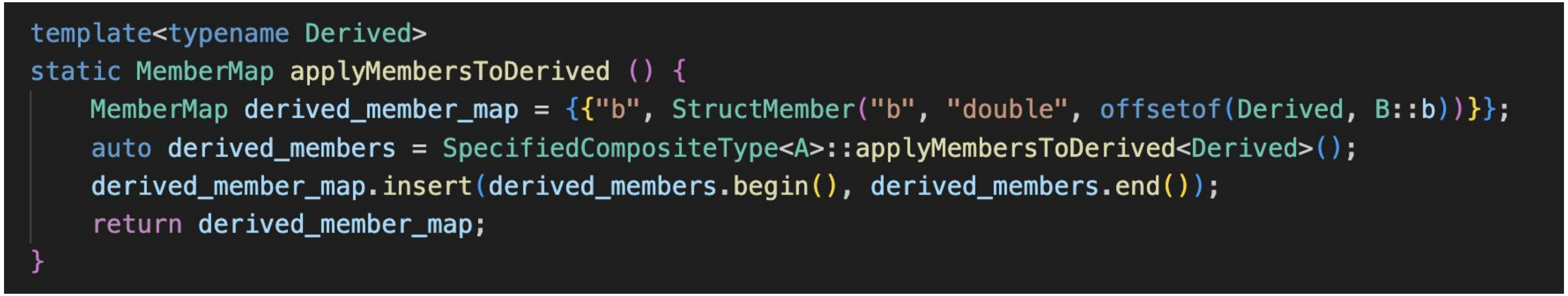
Type Definition (ICG generated)

```
class SpecifiedCompositeType<Foo> : public CompositeType {
     MemberMap& getMemberMap () {
         static MemberMap member_map = {
             {"a", StructMember("a", "int[5][4]", offsetof(Foo, a))},
             {"b", StructMember("b", "Bar", offsetof(Foo, b))} };
         return member_map;
      }
};
```

Inheritance

Accessible inherited members are included in the MemberMap as a normal member. They are assembled using the **`applyMembersToDerived()`** function, which is generated as part of the **`SpecifiedCompositeType<T>`** class.

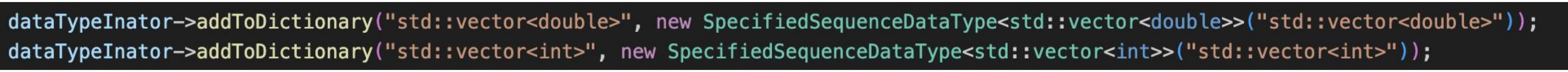
Note - this is complex, I'll write better docs for it. Look at the inheritance test to see a working example.



```cpp
template<typename Derived>
static MemberMap applyMembersToDerived () {
    MemberMap derived_member_map = {{"b", StructMember("b", "double", offsetof(Derived, B::b))}};
    auto derived_members = SpecifiedCompositeType<A>::applyMembersToDerived<Derived>();
    derived_member_map.insert(derived_members.begin(), derived_members.end());
    return derived_member_map;
}
```

STL Types

STL sequence types (vector, deque, list) are supported. They must be specifically registered with the **`DataTypeInator`** by ICG generated code.

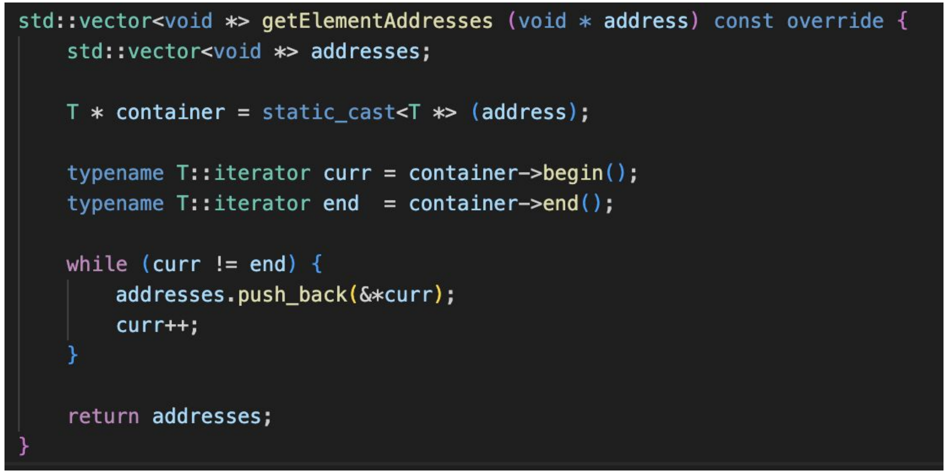


```
dataTypeInator->addToDictionary("std::vector<double>", new SpecifiedSequenceDataType<std::vector<double>>("std::vector<double>"));
dataTypeInator->addToDictionary("std::vector<int>", new SpecifiedSequenceDataType<std::vector<int>>("std::vector<int>"));
```

The key method that makes this work is the [**`getElementAddresses()`**]() method in **`SpecifiedPrimitiveType<T>`**.

Since STLs can reallocate themselves, this method must be called EVERY TIME a traversal is done, and pointers to elements should never be stored.

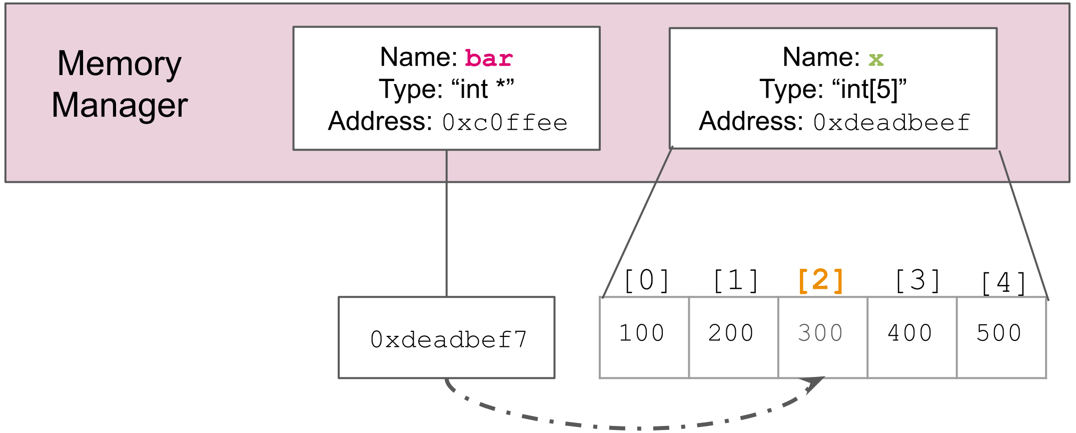
This is used by visitors.



```
std::vector<void *> getElementAddresses (void * address) const override {
    std::vector<void *> addresses;

    T * container = static_cast<T *> (address);

    typename T::iterator curr = container->begin();
    typename T::iterator end  = container->end();

    while (curr != end) {
        addresses.push_back(&*curr);
        curr++;
    }

    return addresses;
}
```

Dealing with Pointers

Data structures may have pointers to one another! These connections should be included in the state checkpoint functionality.



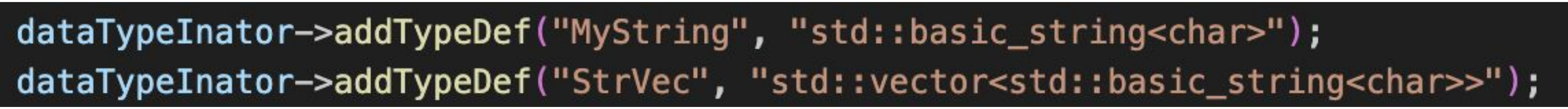
Checkpoint String:

```
bar = &x[2];
```

We cannot save literal addresses in a checkpoint, since the memory may not always be allocated in the same way. Instead, save the name of the variable that is being pointed to, which can be looked up at restore time as long as it is in managed memory.

TypeDef Statements

TypeDef statements are scraped by ICG and registered with the DataTypeInator.



```
dataTypeInator->addTypeDef("MyString", "std::basic_string<char>");
dataTypeInator->addTypeDef("StrVec", "std::vector<std::basic_string<char>>");
```

The DataTypeInator uses a `TypeDefDictionary` (code, doxygen) to track these things. One step in the `resolve` function is to look up the type name in this dictionary. This allows the user to use their typedef'd names with the memory manager.

```
memoryManager.declare_var("MyString s");
```

Namespaces/Nested classes

Namespaces and nested classes are transparent to DataTypes, just treated as funky names.

They are handled by ICG. Types of fields are fully qualified in the AST, and tracking the scope allows class definitions to be correctly qualified.