Writing a C++ 20 Module

Steve Downey

C++ now

2021
MAY 2-7
Aspen, Colorado, USA

WRITING A C++ 20 MODULE

Steve Downey

ABSTRACT

This talk will walk through creating a C++ 20 module interface and the implementation of a simple data structure, a functional tree.

This will cover how to control export of types and inline code, hiding an implementation, and making sure that necessary un-exported definitions are still reachable.

OVERVIEW OF C++ 20 MODULES

NOT PACKAGES - HYGIENE

In the run up to C++ 20 there was a lot of hope that modules solved packaging.

NOT PACKAGES - HYGIENE

In the run up to C++ 20 there was a lot of hope that modules solved packaging.

- They don't.

NOT PACKAGES - HYGIENE

In the run up to C++ 20 there was a lot of hope that modules solved packaging.

- They don't.
- At All.

NOT PACKAGES - HYGIENE

In the run up to C++ 20 there was a lot of hope that modules solved packaging.

- They don't.
- At All.
- They add to the problem.

NOT PACKAGES - HYGIENE

In the run up to C++ 20 there was a lot of hope that modules solved packaging.

- They don't.
- At All.
- They add to the problem.
- Need to deliver source interfaces that consumers will compile.

MODULE UNITS

Modules extend the concept of translation unit.

`Module Unit`
: a TU that contains a module declaration.

`Named Module`
: the collection of `Module Units` with module name.

`Module Interface Unit`
: a module unit that **`export`**s.

`Module Implementation Unit`
: a module unit that does not.

`Primary Module Interface Unit`
: There will be exactly one MIU that is not a partition.

`Module Partition`
: Part of a module. MIU partitions must be exported by the PMIU.

EXAMPLE 1 (FROM THE IS)

```
// Translation unit #1: PMI
export module A;
export import :Foo;
export int baz();

// Translation unit #2: Partion A:Foo
export module A:Foo;
import :Internals;
export int foo() { return 2 * (bar() + 1); }

// Translation unit #3: Partition A:Internals
module A:Internals;
int    bar();

// Translation unit #4: an implementation unit
module A;
import :Internals;
int bar() { return baz() - 10; }
int baz() { return 30; }
```

THE MODEL IS RETROFITTING EXISTING TECH

The standard is complicated because it is trying not to describe an implementation.

A module interface TU produces an object file and a BMI.

A module TU is a TU and produces an object file.

The consumer of a module reads the BMI.

The program links the library or objects from the module.

EXPORTS

Exports make names from the module available to the consumers.

IMPORTS

Makes names from the module visible in the current TU.

```
export import M;
```

makes the module M's exported names visible to the importer of a module.

PRIVATE MODULE FRAGMENT

You can write `Java` style single file modules.

In the Primary Module Interface Unit you can write:

```
module :private;
```

And the names and definitions thereafter are not `reachable` from the importers.

INSTANTIATION CONTEXT

How we figure out what declarations are in play for ADL and which are reachable.

REACHABILITY

Reachable isn't the same as name availability.

> *"Whether a declaration is exported has no bearing on whether it is reachable."*

A TRANSLATION UNIT IS REACHABLE FROM P

- if the unit P is in has an interface dependency on U
- if the unit P is in imports U
- other unspecified reasons you should not depend on

A DECLARATION IS REACHABLE FROM P

- if it appears before P in the same TU
- it is not discarded, is in a unit reachable from P, not in a PMF.

THE THINGS YOU EXPORT MAKE MORE THINGS REACHABLE

This allows consumers to use the things you export, without having to export everything.

```
// Translation unit #1:
export module A;
struct X {};
export using Y = X;

// Translation unit #2:
module B;
import A;
Y y;                // OK, definition of X is reachable
X x;                // error: X not visible to unqualified lookup
```

REACHABILITY IS ABI

THE COMPONENT FRINGETREE TO BE MODULATED

PURE PERSISTENT FUNCTIONAL TREE

Fringe tree is an intentionally poor persistent functional binary tree implementation that grew out of wanting an example to work on the 'same fringe' problem.

Persistent, in this context, means updating the tree doesn't change observable tree and produces a new tree.

Functional implies immutability which means unchanging state can be shared.

Modeled after fingertrees, which are far more complicated.

Data is stored at the edges, the fringe of the tree, internal nodes have children.

USES STD::VARIANT<>, STD::SHARED_PTR<>, AND VISITORS

- Shared ptr isn't that bad.
- `std::variant` is terribly expensive.
- As are `std::variant` visitors.

  The actual interface of the tree does not need them.

  Poster child for modules.

CODE

The template parameter Value is the type held in the fringe.

The template parameter Tag is a monoidal type describing the tree.

The nodes of the tree are one of

Branch
: points to left and right Tree

Leaf
: holds data of types value and tag

Empty
: A nil value. Avoids having nulls.

Tree
: a variant of <Empty, Leaf, Branch>

BRANCH

```
template <typename Tag, typename Value>
class Branch {
    Tag                             tag_;
    std::shared_ptr<Tree<Tag, Value>> left_;
    std::shared_ptr<Tree<Tag, Value>> right_;
///
    };
```

LEAF

```
template <typename Tag, typename Value>
class Leaf {
    Tag   tag_;
    Value v_;
///
    };
```

EMPTY

```
template <typename Tag, typename Value>
class Empty {
  public:
    Empty(){};
    auto tag() const -> Tag { return {}; };
};
```

TREE

```
template <typename Tag, typename Value>
class Tree {
  private:
    std::variant<Empty_, Leaf_, Branch_> data_;

  public:
    Tree(Empty_ const& empty) : data_(empty) {}
    Tree(Leaf_ const& leaf) : data_(leaf) {}
    Tree(Branch_ const& branch) : data_(branch) {}
    ///
    template <typename Callable>
    auto visit(Callable&& c) const {
        return std::visit(c, data_);
    }
};
```

SHARED NODES

Operations on Trees produce Trees that share nodes with the original.

Tree exposes factory functions that return shared_ptr<Tree> constructing empty, leaf, and branch. A "smart constructor" idiom.

TAG

The tag of a branch is the plus operator of the tags of it's left and right. The tag type is required to be monoidal, that is have

- A binary operator+(Tag, Tag) -> Tag
- Have an identity element such that t + identity == t

Examples:

- The + operator on numbers.
- Concatentation on strings or linear containers.
- min and max.

Tags can be used for index lookups, priority, and other things.

EXPOSES FUNCTION OBJECTS AS INTERFACE

DEPTH

```
constexpr inline struct depth {
    template <typename T, typename V>
    auto operator()(Empty<T, V> const&) const -> T {
        return 0;
    }

    template <typename T, typename V>
    auto operator()(Leaf<T, V> const&) const -> T {
        return 1;
    }

    template <typename T, typename V>
    auto operator()(Branch<T, V> const& b) const -> T {
        auto leftDepth  = (b.left()->visit(*this)) + 1;
        auto rightDepth = (b.right()->visit(*this)) + 1;

        return (leftDepth > rightDepth) ? leftDepth : rightDepth;
    }
} depth_;

constexpr auto depth = [](auto tree) { return tree->visit(depth_); };
```

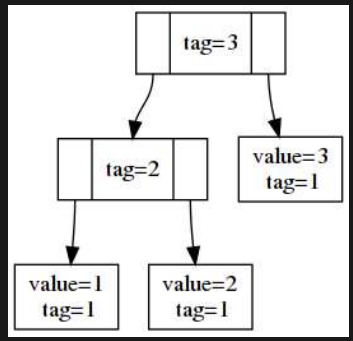
FLATTEN TO VECTOR

```
constexpr inline struct flatten {
    template <typename T, typename V>
    auto operator()(Empty<T, V> const&) const -> std::vector<V> {
        return std::vector<V>{};
    }

    template <typename T, typename V>
    auto operator()(Leaf<T, V> const& l) const -> std::vector<V> {
        std::vector<V> v;
        v.emplace_back(l.value());
        return v;
    }

    template <typename T, typename V>
    auto operator()(Branch<T, V> const& b) const -> std::vector<V> {
        auto leftFlatten  = b.left()->visit(*this);
        auto rightFlatten = b.right()->visit(*this);
        leftFlatten.insert(leftFlatten.end(), rightFlatten.begin(), rightFlatten.end());
        return leftFlatten;
    }
} flatten_;

constexpr auto flatten = [](auto tree) { return tree->visit(flatten_); };
```

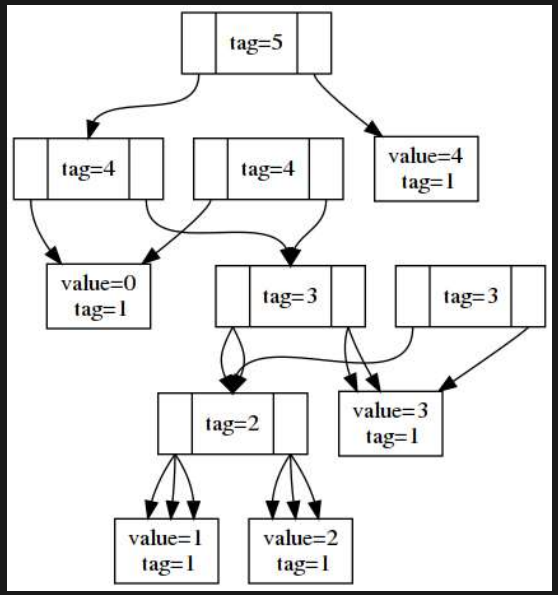
EXAMPLE

```
auto t = Tree::branch(
    Tree::branch(Tree::leaf(1), Tree::leaf(2)),
    Tree::leaf(3)
    );

auto t1 = prepend(0, t);
auto t2 = append(4, t1);

//    printer(std::cout, t_);

std::cout << "digraph G {\n";
printer_ p(std::cout);
t->visit(p);
t1->visit(p);
t2->visit(p);
std::cout << "}\n";
```

JUST T



OUTPUT FROM EXAMPLE



IS AN EXPERIMENTAL TOY

https://github.com/steve-downey/fringetree

Variant and visit can model sum type systems.

Convinced me that we need pattern matching.

CONSIDERATIONS FOR A MODULE

NOT NEW DECISIONS, BUT MORE CONTROL

Export has fine-grained control.

Can chose everything or just particular names.

WHAT TO EXPORT

Export what clients need to name.

WHAT NOT TO EXPORT

Implementation details and infrastructure.

EXPORTING CODE FOR INLINING

If you want to export code as part of your interface you must explicitly inline. Functions defined in the class declaration are not implicitly inline in a module. Inlines can not refer to anything with internal linkage.

ORGANIZATION IS NOT EXPOSED TO CUSTOMERS

You can use partitions, the PMF, module implementation units, and all of it looks the same to customers.

Re-exporting a name may not. Names are 'attached' to modules, and that may be part of the name.

HELLO, WORLD!

HELLO MODULE

```
module;
#include <iostream>
#include <string_view>

export module smd.hello;

export namespace hello {
void hello(std::string_view name)
{
  std::cout << "Hello, " << name << "! \n";
}
}  // namespace hello
```

MAIN

```
import smd.hello;

int main()
{
  hello::hello("Steve");
}
```

MAKEFILE

```
main : main.o hello.o
    g++-11 -o main main.o hello.o

main.o : main.cpp gcm.cache/smd.hello.gcm
    g++-11 -fPIC -fmodules-ts -x c++ -o main.o -c main.cpp

hello.o: hello.cpp
    g++-11 -fPIC -fmodules-ts -x c++ -o hello.o -c hello.cpp

gcm.cache/smd.hello.gcm:  hello.o
    @test -f $@ || rm -f hello.o
    @test -f $@ || $(MAKE) hello.o

clean:
    rm hello.o main.o gcm.cache/smd.hello.gcm

clean-gcm:
    rm gcm.cache/smd.hello.gcm

test:
    ./main
```

CODE

PRIMARY MODULE INTERFACE

```
module;
// global module fragment
#include <non_module.h>
export module foo;
export import  :part; // exports foo:part a module partition

import std; // <= maybe we can do better someday

import bar; // <= not exported, reachable

export namespace foo {
       // everything here is exported
       int theAnswer();
}
```

Every name that clients consume is exported through the primary module interface. Those may be rexported from module partitions or from other modules.

NOTE THAT MODULES COMPOSE

```
export module foo;
export import foo.bar;
export import foo.baz;
export import foo.quux;
```

As long as there is a strict dependency directed acyclic graph (DAG) between the more fine grained modules.

. is a convention. It has no hierarchical meaning to the compiler.

MODULE IMPLEMENTATION UNIT(S)

These are almost the same as a regular translation unit, except they have access to module linkage names.

```
module foo;

int foo::theAnswer() { // foo is the namespace, not the module.
    return 42;
}
```

MODULE PARTITIONS TO DECOMPOSE LARGE MODULES

```
export module foo:part;

export int quux_foos(int);
```

No one outside may access the partition. You can not `import foo:part`.

ACCESS TO NAMES WITH MODULE LINKAGE

Your module partitions have access to all of the names and definitions from the module interface.

PRIVATE FRAGMENT

A special partition that can appear in a primary module interface. They allow unexported and unreachable definitions to be included in the PMI.

From The Standard [module.private.frag]:

```
export module A;
export inline void fn_e();      // error: exported inline function fn_e not defined
                                // before private module fragment
inline void fn_m();             // OK, module-linkage inline function
static void fn_s();
export struct X;
export void g(X *x) {
  fn_s();                       // OK, call to static function in same translation unit
  fn_m();                       // OK, call to module-linkage inline function
}
export X *factory();            // OK

module :private;
struct X {};                    // definition not reachable from importers of A
X *factory() {
  return new X ();
}
void fn_e() {}
void fn_m() {}
void fn_s() {}
```

THE C++ STANDARD TRIES TO AVOID POLICY

The private fragment exists to allow single file modules.

Many people want them.

It's not clear this is the best, or a good, way to deploy modules at scale. Combining the interface and implementation is likely to convince build systems that implementation changes require rebuilds of all interface dependers.

Barring a mythical "smart enough" build system.

BUILDING MODULES

YOUR BUILD SYSTEM WILL NOT SURVIVE CONTACT

In particular the name of a module in code is not a filename.

Module dependencies require some parsing of C++, which is hard. The standard tries to make it almost regexpable.

MUST BUILD IN DAG ORDER

Before a module can imported, it has to be built.

If you are lucky, if you build in the wrong order, the module interface won't be available.

Stale module interfaces will seem to work but will be wrong.

BACK TO THE FUTURE: `MAKEDEPS`

The current model is the compiler emits dependency information as it compiles. This works because if deps don't exist, you need to compile, and deps can't change without files changing, causing a recompile.

Before that there was a tool `makedeps`. Because doing it by hand is impossible to keep correct.

`makedeps` runs before the main build.

Compiler vendors are working on tools to emit what modules are direct dependency and what module a translation unit produces.

PACKAGING MODULES IS AN OPEN QUESTION

We don't have a solution yet. We might be able to extend pkgconfig metadata. There's divergence in compiler flags.

CMI ARE FRAGILE - PLAN ON DELIVERING SOURCE

You will not be able to ship your compiled module interface. They depend on compiler internals.

I might be able to. We version our compiler along with libraries and packages.

We ship what's basically an OS distro with just userland every 15 minutes.

MODULATING FRINGETREE

SUCCESSFUL WITH VISUAL STUDIO

(yesterday)

ICED GCC

Segfaults in reporting the error it was reporting.

CLANG SOMEWHERE IN BETWEEN

Didn't ICE, but reachability issues.

ACTUAL WORK WAS STRAIGHTFORWARD

At least partly because I'm in the habit of Lakosian components which are always strictly DAG and have tests.

Convincing Visual Studio to export the interface was more frustrating. I'm not a fan of GUIs for this. "Simply open the X dialog and …" is not.

But it worked.

THE HEADER MOVED TO THE INTERFACE FILE (.IXX)

I don't know what our eventual style guide will be here.

```
// fringtree.ixx
module;
#include <memory>
#include <variant>
#include <vector>

export module smd.fringetree;
namespace fringetree {
```

1. I'm claiming the smd. module space. Find your own. 😈
2. I'm not exporting the fringetree namespace.

EXPORTING TREE TYPE

The node types are made reachable.

```
template <typename Tag, typename Value>
class Leaf {
    //
};
template <typename Tag, typename Value>
class Branch {
    //
};

template <typename Tag, typename Value>
class Empty {
    //
};

export // <=== Make the Tree template available
template <typename Tag, typename Value>
class Tree
    //
};
```

USING TREE

Clients interact with Tree, and want to be able to name Tree.

```
using namespace fringetree;

using Tree = Tree<int, int>;
auto t     = Tree::branch(Tree::branch(Tree::leaf(1), Tree::leaf(2)), Tree::leaf(3));
```

INLINE FUNCTION DEFINITIONS

inline isn't unless you say so.

```
export template <typename Tag, typename Value>
class Tree {
public:
  bool isEmpty() { return std::holds_alternative<Empty_>(data_); }
}
```

The definition of branch is not inlined in client code. Trade-offs with exposing implementation vs optimization opportunities.

EXPORTING FUNCTION OBJECTS

```
    constexpr inline struct breadth {
        template <typename T, typename V>
        auto operator()(Empty<T, V> const&) const -> T {
            return 0;
        }

        template <typename T, typename V>
        auto operator()(Leaf<T, V> const&) const -> T {
            return 1;
        }

        template <typename T, typename V>
        auto operator()(Branch<T, V> const& b) const -> T {
            return b.left()->visit(*this) + b.right()->visit(*this);
        }
    } breadth_;

export constexpr auto breadth = [](auto tree) { return tree->visit(breadth_); };
```

(Note Visual Studio disagrees that I have to export breadth)

breadth is a lambda that uses the breadth_ visitor object.

TESTS PASS

Having test coverage is very good. It helps you not fool yourself.

THANK YOU