15.3 — Nested types (member types)

ALEX[1] FEBRUARY 27, 2025

Consider the following short program:

```
#include <iostream>

enum class FruitType
{
    apple,
    banana,
    cherry
};

class Fruit
{
private:
    FruitType m_type { };
    int m_percentageEaten { 0 };

public:
    Fruit(FruitType type) :
        m_type { type }
    {
    }

    FruitType getType() { return m_type; }
    int getPercentageEaten() { return m_percentageEaten; }

    bool isCherry() { return m_type == FruitType::cherry; }

};

int main()
{
    Fruit apple { FruitType::apple };

    if (apple.getType() == FruitType::apple)
        std::cout << "I am an apple";
    else
        std::cout << "I am not an apple";

    return 0;
}
```

There's nothing wrong with this program. But because `enum class FruitType` is meant to be used in conjunction with the `Fruit` class, having it exist independently of the class leaves us to infer how they are connected.

Nested types (member types)

So far, we've seen class types with two different kinds of members: data members and member functions. Our `Fruit` class in the example above has both of these.

Class types support another kind of member: **nested types** (also called **member types**). To create a nested type, you simply define the type inside the class, under the appropriate access specifier.

Here's the same program as above, rewritten to use a nested type defined inside the `Fruit` class:

```cpp
#include <iostream>

class Fruit
{
public:
    // FruitType has been moved inside the class, under the public access specifier
        // We've also renamed it Type and made it an enum rather than an enum class
    enum Type
    {
        apple,
        banana,
        cherry
    };

private:
    Type m_type {};
    int m_percentageEaten { 0 };

public:
    Fruit(Type type) :
        m_type { type }
    {
    }

    Type getType() { return m_type;  }
    int getPercentageEaten() { return m_percentageEaten;  }

    bool isCherry() { return m_type == cherry; } // Inside members of Fruit, we no longer need to prefix enumerators with FruitType::
};

int main()
{
    // Note: Outside the class, we access the enumerators via the Fruit:: prefix now
    Fruit apple { Fruit::apple };

    if (apple.getType() == Fruit::apple)
        std::cout << "I am an apple";
    else
        std::cout << "I am not an apple";

    return 0;
}
```

There are a few things worth pointing out here.

First, note that `FruitType` is now defined inside the class, where it has been renamed `Type` for reasons that we will discuss shortly.

Second, nested type `Type` has been defined at the top of the class. Nested type names must be fully defined before they can be used, so they are usually defined first.

Best practice

Define any nested types at the top of your class type.

Third, nested types follow normal access rules. `Type` is defined under the `public` access specifier, so that the type name and enumerators can be directly accessed by the public.

Fourth, class types act as a scope region for names declared within, just as namespaces do. Therefore the fully qualified name of `Type` is `Fruit::Type`, and the fully qualified name of the `apple` enumerator is `Fruit::apple`.

Within the members of the class, we do not need to use the fully qualified name. For example, in member function `isCherry()` we access the `cherry` enumerator without the `Fruit::` scope qualifier.

Outside the class, we must use the fully qualified name (e.g. `Fruit::apple`). We renamed `FruitType` to `Type` so we can access it as `Fruit::Type` (rather than the more redundant `Fruit::FruitType`).

Finally, we changed our enumerated type from scoped to unscoped. Since the class itself is now acting as a scope region, it's somewhat redundant to use a scoped enumerator as well. Changing to an unscoped enum means we can access enumerators as `Fruit::apple` rather than the longer `Fruit::Type::apple` we'd have to use if the enumerator were scoped.

Nested typedefs and type aliases

Class types can also contain nested typedefs or type aliases:

```cpp
#include <iostream>
#include <string>
#include <string_view>

class Employee
{
public:
    using IDType = int;

private:
    std::string m_name{};
    IDType m_id{};
    double m_wage{};

public:
    Employee(std::string_view name, IDType id, double wage)
        : m_name { name }
        , m_id { id }
        , m_wage { wage }
    {
    }

    const std::string& getName() { return m_name; }
    IDType getId() { return m_id; } // can use unqualified name within class
};

int main()
{
    Employee john { "John", 1, 45000 };
    Employee::IDType id { john.getId() }; // must use fully qualified name outside
class

    std::cout << john.getName() << " has id: " << id << '\n';

    return 0;
}
```

This prints:

```
John has id: 1
```

Note that inside the class we can just use `IDType`, but outside the class we must use the fully qualified name `Employee::IDType`.

We discuss the benefits of type aliases in lesson [10.7 -- Typedefs and type aliases (https://www.learncpp.com/cpp-tutorial/typedefs-and-type-aliases/)](https://www.learncpp.com/cpp-tutorial/typedefs-and-type-aliases/)[2], and they serve the same purpose here. It is very common for classes in the C++ standard library to make use of nested typedefs. As of the time of writing, `std::string` defines ten nested typedefs!

Nested classes and access to outer class members

It is fairly uncommon for classes to have other classes as a nested type, but it is possible. In C++, a nested class does not have access to the `this` pointer of the outer (containing) class, so nested classes can not directly access the members of the outer class. This is because a nested class can be instantiated independently of the outer class (and in such a case, there would be no outer class members to access!)

However, because nested classes are members of the outer class, they can access any private members of the outer class that are in scope.

Let’s illustrate with an example:

```
#include <iostream>
#include <string>
#include <string_view>

class Employee
{
public:
    using IDType = int;

    class Printer
    {
    public:
        void print(const Employee& e) const
        {
            // Printer can't access Employee's `this` pointer
            // so we can't print m_name and m_id directly
            // Instead, we have to pass in an Employee object to use
            // Because Printer is a member of Employee,
            // we can access private members e.m_name and e.m_id directly
            std::cout << e.m_name << " has id: " << e.m_id << '\n';
        }
    };

private:
    std::string m_name{};
    IDType m_id{};
    double m_wage{};

public:
    Employee(std::string_view name, IDType id, double wage)
        : m_name{ name }
        , m_id{ id }
        , m_wage{ wage }
    {
    }

    // removed the access functions in this example (since they aren't used)
};

int main()
{
    const Employee john{ "John", 1, 45000 };
    const Employee::Printer p{}; // instantiate an object of the inner class
    p.print(john);

    return 0;
}
```

This prints:

```
John has id: 1
```

There is one case where nested classes are more commonly used. In the standard library, most iterator classes are implemented as nested classes of the container they are designed to iterate over. For example, `std::string::iterator` is implemented as a nested class of `std::string`. We'll cover iterators in a future chapter.

Nested types and forward declarations

A nested type can be forward declared within the class that encloses it. The nested type can then be defined later, either within the enclosing class, or outside of it. For example:

```cpp
#include <iostream>

class outer
{
public:
    class inner1;   // okay: forward declaration inside the enclosing class okay
    class inner1{}; // okay: definition of forward declared type inside the enclosing class
    class inner2;   // okay: forward declaration inside the enclosing class okay
};

class inner2 // okay: definition of forward declared type outside the enclosing class
{
};

int main()
{
    return 0;
}
```

However, a nested type cannot be forward declared prior to the definition of the enclosing class.

```cpp
#include <iostream>

class outer;         // okay: can forward declare non-nested type
class outer::inner1; // error: can't forward declare nested type prior to outer class definition

class outer
{
public:
    class inner1{}; // note: nested type declared here
};

class outer::inner1; // okay (but redundant) since nested type has already been declared as part of outer class definition

int main()
{
    return 0;
}
```

While you can forward declare a nested type after the definition of the enclosing class, since the enclosing class will already contain a declaration for the nested type, doing so is redundant.

Next lesson
15.4 Introduction to destructors

3

Back to table of contents

4

Previous lesson
15.2 Classes and header files

5

B U URL INLINE CODE C++ CODE BLOCK HELP!

Leave a comment...

Name*

Email*

Notify me about replies:

POST COMMENT

Find a mistake? Leave a comment above!

Avatars from https://gravatar.com/[9] are connected to your provided email address.

86 COMMENTS Newest

vitrums

April 23, 2025 1:35 pm PDT

I made a follow up on the sample in the end of this chapter and discovered that `inner2` forward declaration can mean different things depending on how many types we define later. If we defined only `class inner2 {};`, then the declaration inside `outer` would be about some non-inner class named `inner2`. But in the following example:

```cpp
class outer {
public:
    class inner2;   // okay: forward declaration inside the enclosing class okay
};

class inner2 {};
class outer::inner2 {};
```

because we also defined `outer::inner2` the compiler relates `inner2` forward declaration to `outer::inner2`. It's easy to demonstrate by adding a member pointer and trying to point to lvalues of different types as follows:

```
class outer {
public:
    class inner2;   // okay: forward declaration inside the enclosing class okay
    inner2* i2_ptr{};
};

class inner2 {};
class outer::inner2 {};

int main() {
    outer o{};

    inner2 i{};
    outer::inner2 oi{};

    o.i2_ptr = &oi; // okay: we can point at lvalue instance of type outer::inner2
    o.i2_ptr = &i;  // error: but not at objects of type inner2
    return 0;
}
```

Hence, the definition of `outer::inner2` "overshadows" the definition of `inner2` in the context of forward declaration `class inner2;` inside `outer`.

2 Reply

Hari

Reply to vitrums 10 April 25, 2025 10:50 pm PDT

Thanks! This helps make it clear what happens when you put the class type definition in a header file, and put all the members in a separate class file. You'd use `Classname::` to define external member functions, and now, you would do the same for the forward declared nested types

Last edited 2 months ago by Hari

0 Reply

joddy

March 19, 2025 5:30 am PDT

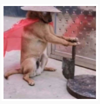
Dear Alex,

I didn´t get the idea why (at least for a brazilian student) it´s not necessary to use the fully qualified name for an enum.

"Changing to an unscoped enum means we can access enumerators as `Fruit::apple` rather than the longer `Fruit::Type::apple` we'd have to use if the enumerator were scoped."

It´s not clear to me, especially outside the class.

0 Reply



LechugaPlayer

Reply to joddy 11 April 12, 2025 12:06 pm PDT

If you check from 13.2 to 13.6 it should become much clearer

1 Reply

Kyutae Lee
February 22, 2025 11:02 pm PST

> Nested types can't be forward declared

> There is one other limitation of nested types that is worth mentioning -- nested types can't be forward declared. This limitation may be lifted in a future version of C++.

I think the statement above contradicts the following cppreference content (source: https://en.cppreference.com/w/cpp/language/nested_types (https://en.cppreference.com/w/cpp/language/nested_types)[12]).

> Nested classes can be forward-declared and later defined, either within the same enclosing class body, or outside of it:

```
class enclose
{
    class nested1;    // forward declaration
    class nested2;    // forward declaration
    class nested1 {}; // definition of nested class
};

class enclose::nested2 {}; // definition of nested class
```

Last edited 4 months ago by Kyutae Lee

0 Reply

Alex Author
Reply to Kyutae Lee [13] February 27, 2025 12:56 am PST

Thanks for the feedback! I revised the wording in the lesson, and provided both this case and an example of the case that does not work.

2 Reply

DANIEL S MAIN
December 4, 2024 5:39 am PST

I have a style question. The nested {} are really helpful in lots of places to determine the scope of an operation or function. Why are the {} not used when declaring whether the members or public or private? Thanks

0 Reply

Alex Author
Reply to DANIEL S MAIN [14] December 5, 2024 10:21 pm PST

Not totally sure. {} tends to imply some sort of nested scope region. Access specifiers don't create a scope region. They're just a cascading property.

2 Reply

Ali
September 28, 2024 2:02 pm PDT

How much nested classes are recommended? Do C++ developers use it?

1 Reply

Alex Author
Reply to Ali 15 October 1, 2024 11:23 am PDT

Nested classes aren't common, but they are used here and there, when the nested class is highly correlated to the outer class and isn't something that makes sense to live independently. They are probably most common in containers (e.g. a list class might have a nested node class to facilitate implementation, and an iterator implementation that is specific to a container might be implemented as a nested class).

I'd recommend using them if and when they make your implementation less complex or when you want to limit/prevent access to the class because it's meant only as a helper.

4 Reply

Baker
May 3, 2024 7:52 pm PDT

Why can't nested types be forward declared? What's the difference between forward declaring a class and a nested type?

1 Reply

Alex Author
Reply to Baker 16 May 4, 2024 8:20 pm PDT

> Why can't nested types be forward declared?

Because the language doesn't support it. I don't see why it couldn't. Maybe nobody thought it was worth advocating for.

> What's the difference between forward declaring a class and a nested type?

Classes are usually declared in namespace scope (either in the global namespace or a user-defined namespace). Forward declaring a type defined in namespace scope is allowed.

Nested types, by definition, are declared in class scope. Forward declaring a type declared in class scope is disallowed.

3 Reply

Buckley
April 18, 2024 1:06 am PDT

Modifying the 'Employee' example a bit, as well as split it into separated files.
This is what I landed on:

`Employee.h`

```cpp
#pragma once
#include <iostream>
#include <string>

class Employee
{
public:
    using ID = int;
    using Wage = double;
    using Name = std::string;
    using Gender = std::string;

private:
    ID m_id{ 0 };
    Wage m_wage{ 0.0 };
    Name m_name{ "John" };
    Gender m_gender{ "M" };
    const bool m_descriptor{ m_gender == "male" || m_gender == "Male" || m_gender == "M" };

    constexpr Gender descriptor() const;
    void getEmployeeSheet() const;

public:
    Employee(std::string_view name = "John", ID id = 0, Wage wage = 0.0, Gender g = "M");

    ID id() const { return m_id; }
    Wage wage() const { return m_wage; }
    const Name& name() const { return m_name; }
    void printEmployeeSheet() const { getEmployeeSheet(); }
};
```

`Employee.cpp`

```cpp
#include "Employee.h"

constexpr Employee::Gender Employee::descriptor() const { return (m_descriptor) ? "his" : "her"; }
void Employee::getEmployeeSheet() const
{
    std::cout
        << "\nThis employee's name is: " << m_name << ".\n"
        << descriptor() << " yearly salary is $" << m_wage << ", and "
        << descriptor() << " employee id is " << m_id << ".\n";
}

Employee::Employee(std::string_view name, ID id, Wage wage, Gender g)
    : m_name{ name }
    , m_id{ id }
    , m_wage{ wage }
    , m_gender{ g }
{   }
```

`main.cpp`

```cpp
#include "Employee.h"

int main()
{
    Employee john{ "John", 1, 45'000 , "M" };
    Employee jane{ "Jane", 2, 45'000 , "F" };

    john.printEmployeeSheet();
    jane.printEmployeeSheet();

    return 0;
}
```

This is the output for the program:

====================

This employee's name is: John.
his yearly salary is $45000, and his employee id is 1.

This employee's name is: Jane.
her yearly salary is $45000, and her employee id is 2.

====================

If anyone sees any glaring issues, or has suggestions for me… Please let me know! Thanks in advance!

Edit: I realized shortly after posting this that 'printEmployeeSheet' & 'getEmployeeSheet' could've probably just been merged into one public 'printEmployeeSheet'
function with all of the functionality publicly accessible.

My rationale for separating the functions into two initially was to separate the classes internal 'implementation' from it's public 'interface'… Upon further reflection, I am not sure this simple of a class would warrant 'data hiding' if it makes the code more cluttered and doesn't add much beyond that…am I overthinking this?

I want to establish good habits now at the foundational level, not spend effort tearing bad ones down later, so any advice helps.

Last edited 1 year ago by Buckley

5 Reply

Thesheshadesgucci
April 4, 2024 12:23 pm PDT

if the first program if i remove enum class and make it just enum why i just do Fruit apple{apple} since it's now unscoped enum ?

0 Reply

Alex Author
Reply to Thesheshadesgucci 17 April 4, 2024 2:15 pm PDT

`Fruit apple{apple}` leaves `apple` uninitialized since you're trying to initialize it with itself. This `apple` shadows the enumerator.

1 Reply

Jacob
March 14, 2024 3:42 pm PDT

Hi Alex.

Like others in this forum I am experimenting with my own version of your Fruit class, using my own nomenclature just to be less tempted to copy too much of your code. Also preferring to define the class in a Fruit.h file. Hence, I define the two "get" functions as:

```
public:
 inline fruiType getFruit {return m_fruit;}
 inline int getPctEaten   {return m_pctEaten;}
```

I also want to declare a static array of little structures to map the enum value to the name of the fruit. I've done this in C and Perl a gazillion times but never in the .h file for the C work.

In the private section:

```
struct fruitMap         // Arguably could be a nested **class**  but later..
{
  fruitType ftype{};    // An enum value
  std::string fname{};  // A string to be filled shortly
};
```

Now I want to instantiate an array of these at compile time:

```
fruitMap fruitList[]
= {
    {apple,  "apple"},
    {banana, "banana"},
    {cherry, "cherry"},
    {daiquiri, "daiquiri"},  //(Yum! ;-)
    // other fruits I like..
  };
```

I would also write a stringview function name getFruitName that returns the name of the fruit object. The loop required does not seem to be appropriate to an inline function. Hence my two questions, which I think are both Yes.

1. Can I initialize such an array in the .h file? I see no reason why not; I did this at compile time on C programs with far more complicated structures.
2. Can I write the non-inline function in the .h file? Looking back to lesson 14.2:

`Member functions defined inside the class definition are implicitly inline.`
It would seem that I indeed can. I fact, the `inline` designations I have already used seem unnecessary.

I really should experiment first but I'm trying to avoid getting into a situation that forces me back to start from scratch.

Thank you for guidance. (Dare I call you Old One? <grin>)

Last edited 1 year ago by Jacob

0 Reply

Alex Author
Reply to Jacob 18 March 16, 2024 4:50 pm PDT

1. Yes, if you make the array `inline` (or `constexpr`, which is implicitly inline).
2. No. A function defined in a .h should be inline (either explicitly or implicitly) to avoid ODR violations if the header is included into multiple translation units.

0 Reply

Jacob
Reply to Alex 19 March 17, 2024 3:54 pm PDT

Bottom line first, so you see where's I'm headed:

In C, I can initilize an array without giving the array size;
`int abclist[] = {1,2,3,4 5, 17,88};`
How do I do this in C++

Now the long version :
I'll get back to the inline business for those 1-liner functions later. I've commented out all functions to isolate only the array definition. I'm also avoiding duplication of definitions with #IFDEF. So here is my current .h file:

```
#ifndef FRUIT_H
#define FRUIT_H
#include <string>
#include <string_view>
class Fruit
{
 public:                // Declared first because a private member will
use it
   enum fruitType
   {
     apple,
     banana,
     cherry,
     daiquiri,
     mango,
     peach,
     _nomore_            // Mark end of enum list
   };
 private:
   fruitType m_fruit {};
   int      m_pctEaten {0};
   struct fruitMap
   {
     fruitType ftype{};
     std::string fruitName{};
   };
   fruitMap fruitList[]        //**NOTE** No inline or constexpr,  ___AND
NO ARRAY COUNT___
   {
     {apple,    "apple"},
     {banana,   "banana"},
     {cherry,   "cherry"},
     {daiquiri, "daiquiri"},
     {mango,    "mango"},
     {peach,    "peach"},
     {_nomore_, ""}    // End of list
   };
};  //(End of class definition)
#endif
```

The main() is a total dummy for now:

```
#include "Fruit.h"

main()
{
  return 0;
}
```

Without an inline or constexpr designation my compile yields:

```
$ c++ -c fruits.cpp
In file included from fruits.cpp:1:
Fruit.h:28:12: error: initializer for flexible array member ‘Fruit::fruitMap
Fruit::fruitList []’
28 | fruitMap fruitList[]
| ^~~~~~~~~
```

OK! OK! I'll add the constexpr:

```
constexpr fruitMap fruitList[]
...
```

Well, now I have 2 error messages:

```
In file included from fruits.cpp:1:
Fruit.h:28:3: error: non-static data member ‘fruitList’ declared ‘constexpr’
28 | constexpr fruitMap fruitList[]
| ^~~~~~~~~
Fruit.h:28:22: error: initializer for flexible array member ‘const
Fruit::fruitMap Fruit::fruitList []’
28 | constexpr fruitMap fruitList[]
```

I wonder if this is a problem with the "flexible array member" business, since I did not specify an array size. In fact, I just tried specifying [7] for array size and it complied without the constexpr designation (giving me an **error when I *do* specify constexpr**). This goes against the grain of my C experience. (Not to mention Perl's array-size anarchy!)

Concise is my middle name... NOT!

0 Reply

Jacob

Reply to Jacob ^{20} March 18, 2024 5:54 am PDT

It occurred to me this morning that the whole idea of putting that pre-initialized array inside the class definition is wrong-headed. Even when I got it to compile (by reluctantly giving an explicit array count) I suspect there would be a new copy of that array in every instance of a Fruit object.

I **think** I need to make the fruitMap structure public, then create & initialize the fruitMap[] array in the main.cpp, though not necessarily inside main() function. (I also think I've gone beyond the scope of the lesson. Gadzooks Alex! Have you created a monster? ;-)

0 Reply

Alex Author

Reply to Jacob ^{21} March 18, 2024 11:02 am PDT

If you want `fruitList` to be shared by all `Fruit` type, you can make it a static (or inline static) member. This also allows you to keep it (and `FruitMap`) private.

We cover static members later in this chapter.

0 Reply

Jacob

Reply to Alex ^{22} March 18, 2024 8:03 pm PDT

Hi Alex.

I actually did try prefacing the array definition with static but that was rejected. But, as I mentioned above, trying to put that inside the class definition seems wrong-headed. But being one tenacious szamár, I pulled the array out of the class, as well as function fruitName(), though I kept them inside the .h file. I think this is where you might suggest moving those two items out of the .h and into Fruits.cpp. But this works now. I hope this will be instructive to others and not just taking up eye-bandwidth.

Fruit.h

```cpp
#ifndef FRUIT_H
#define FRUIT_H
#include <iostream>
#include <string>
#include <string_view>
class Fruit
{
 public:                // Declared first because a
private member will use it
  enum fruitType
  {
    apple,
    banana,
    cherry,
    daiquiri,
    mango,
    peach,
    _nomore_           // Mark end of enum list
  };
  struct fruitMap
  {
    fruitType ftype{};
    std::string fruitName{};
  };
 private:
  fruitType m_fruit {};
  int      m_pctEaten {0};
 public:                // Constructor, given a fruit
type
  Fruit(fruitType ft)
       : m_fruit{ft}
  {
  }
  fruitType getFruit() {return m_fruit;}    // Return
enum value to caller
  int getPctEaten()    {return m_pctEaten;}
  void eatFruit(int pctEaten)
  {
    m_pctEaten += pctEaten;
    if (m_pctEaten > 100)
    {
      std::cout << "Cannot eat " << m_pctEaten << " %
of a fruit\n";
      m_pctEaten = 100;       // Set it back to what's
possible
    }
  }

};

// Arguably, this array and the fruitName() function
should be moved into
// Fruits.cpp but we'll jump off that bridge when we
get to it.
//
const Fruit::fruitMap fruitList[]
{
  {Fruit::fruitType::apple,    "apple"},
  {Fruit::fruitType::banana,   "banana"},
  {Fruit::fruitType::cherry,   "cherry"},
  {Fruit::fruitType::daiquiri, "daiquiri"},
  {Fruit::fruitType::mango,    "mango"},
  {Fruit::fruitType::peach,    "peach"},
  {Fruit::fruitType::_nomore_, ""}    // End of list
};

std::string_view fruitName(Fruit::fruitType ft)
{
  int lc;                     // Loop counter
  std::string_view rval{""};  // Eventual return value
```

```
    for (lc = 0; fruitList[lc].ftype !=
Fruit::fruitType::_nomore_; lc++)
    {
      if (fruitList[lc].ftype == ft)
      {
        rval = fruitList[lc].fruitName;
        break;
      }
    }
    return rval;
}

#endif
```

Now the main, fruits.cpp :

```
#include "Fruit.h"
#include <iostream>

main()
{
    Fruit fruit1{Fruit::fruitType::banana};
    fruit1.eatFruit(17);
    std::string_view frname =
fruitName(fruit1.getFruit());
    int eaten{fruit1.getPctEaten()};
    std::cout << "The " << frname << " has been " <<
eaten << "% gobbled\n";
    Fruit fruit2{Fruit::fruitType::daiquiri};
    fruit2.eatFruit(85);
    std::cout << "The " << fruitName(fruit2.getFruit())
              << " has been " << fruit2.getPctEaten() <<
"% drunk\n";
    fruit2.eatFruit(49);
    std::cout << "The " << fruitName(fruit2.getFruit())
              << " has been " << fruit2.getPctEaten() <<
"% drunk\n";

    return 0;
}
```

And some output:

```
$ ./fruits
The banana has been 17% gobbled
The daiquiri has been 85% drunk
Cannot eat 134 % of a fruit
The daiquiri has been 100% drunk
```

WHEW!

0 Reply

Jacob

Reply to Jacob[23] March 18, 2024 10:22 pm PDT

Epilogue, unless someone wants to see all that code again..
I did break out the fruitMap array and function fruitName() into a separate Fruit.cpp file. The only thing I had to add to Fruit.h was a forward declaration:

```
std::string_view fruitName(Fruit::fruitType ft);
```

OK, on to lesson 15.4

0 Reply

Marian
November 10, 2023 2:07 pm PST

What does the last affirmation mean? "nested types can't be forward declared."

1 Reply

Alex Author
Reply to Marian 24 November 11, 2023 1:55 pm PST

It means if you have something like this:

```
class Foo
{
};

enum Goo
{
};

class Boo
{
    enum Moo
    {
    };
};
```

You can forward declare `Foo`, `Goo`, and `Boo`, but not `Boo::Moo`.

5 Reply

Phargelm
Reply to Alex 25 April 23, 2024 9:16 am PDT

What is actually class forward declaration? From the last lesson I see that class can only be fully defined as compiler needs to see the full definition of a class:

> Unlike functions, which only need a forward declaration to be used, the compiler typically needs to see the full definition of a class (or any program-defined type) in order for the type to be used. This is because the compiler needs to understand how members are declared in order to ensure they are used properly, and it needs to be able to calculate how large objects of that type are in order to instantiate them. So our header files usually contain the full definition of a class rather than just a forward declaration of the class.

Is there exists an example of syntax and how class forward declaration can be used?

Last edited 1 year ago by Phargelm

0 Reply

Alex Author

Reply to Phargelm [26] April 25, 2024 8:14 pm PDT

Yes, you can forward declare a class:

```
class Foo; // forward declaration of class Foo
```

This tells the compiler that `Foo` is a class. However, until a full definition has been provided, `Foo` is an incomplete type, so the things you can actually do with this is really limited. The most common thing you can do is declare a pointer or reference to Foo.

We show a full example of this in lesson https://www.learncpp.com/cpp-tutorial/friend-classes-and-friend-member-functions/

0 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/typedefs-and-type-aliases/
3. https://www.learncpp.com/cpp-tutorial/introduction-to-destructors/
4. https://www.learncpp.com/
5. https://www.learncpp.com/cpp-tutorial/classes-and-header-files/
6. https://www.learncpp.com/nested-types-member-types/
7. https://www.learncpp.com/cpp-tutorial/chapter-26-summary-and-quiz/
8. https://www.learncpp.com/cpp-tutorial/rethrowing-exceptions/
9. https://gravatar.com/
10. https://www.learncpp.com/cpp-tutorial/nested-types-member-types/#comment-609509
11. https://www.learncpp.com/cpp-tutorial/nested-types-member-types/#comment-608665
12. https://en.cppreference.com/w/cpp/language/nested_types
13. https://www.learncpp.com/cpp-tutorial/nested-types-member-types/#comment-607985
14. https://www.learncpp.com/cpp-tutorial/nested-types-member-types/#comment-604857
15. https://www.learncpp.com/cpp-tutorial/nested-types-member-types/#comment-602455
16. https://www.learncpp.com/cpp-tutorial/nested-types-member-types/#comment-596628
17. https://www.learncpp.com/cpp-tutorial/nested-types-member-types/#comment-595432
18. https://www.learncpp.com/cpp-tutorial/nested-types-member-types/#comment-594700
19. https://www.learncpp.com/cpp-tutorial/nested-types-member-types/#comment-594764
20. https://www.learncpp.com/cpp-tutorial/nested-types-member-types/#comment-594789
21. https://www.learncpp.com/cpp-tutorial/nested-types-member-types/#comment-594817
22. https://www.learncpp.com/cpp-tutorial/nested-types-member-types/#comment-594829
23. https://www.learncpp.com/cpp-tutorial/nested-types-member-types/#comment-594848
24. https://www.learncpp.com/cpp-tutorial/nested-types-member-types/#comment-589599

25. https://www.learncpp.com/cpp-tutorial/nested-types-member-types/#comment-589661
26. https://www.learncpp.com/cpp-tutorial/nested-types-member-types/#comment-596111
27. https://g.ezoic.net/privacy/learncpp.com