8.15 — Nested types in classes

BY ALEX ON DECEMBER 21ST, 2016 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

Consider the following short program:

```
#include <iostream>

enum 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;  }
};

int main()
{
    Fruit apple(APPLE);

    if (apple.getType() == 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 FruitType is meant to be used in conjunction with the Fruit class, it's a little weird to have it exist independently from the class itself.

Nesting types

Unlike functions, which can't be nested inside each other, in C++, types can be defined (nested) inside of a class. To do this, you simply define the type inside the class, under the appropriate access specifier.

Here's the same program as above, with FruitType defined inside the class:

```
#include <iostream>

class Fruit
{
public:
    // Note: we've moved FruitType inside the class, under the public access specifier
    enum FruitType
    {
        APPLE,
        BANANA,
        CHERRY
    };

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;  }
};

int main()
{
    // Note: we access the FruitType via Fruit 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;
}
```

First, note that FruitType is now defined inside the class. Second, note that we've defined it under the public access specifier, so the type definition can be accessed from outside the class.

Classes essentially act as a namespace for any nested types. In the prior example, we were able to access enumerator APPLE directly, because the APPLE enumerator was placed into the global scope (we could have prevented this by using an enum class instead of an enum, in which case we'd have accessed APPLE via FruitType::APPLE instead). Now, because FruitType is considered to be part of the class, we access the APPLE enumerator by prefixing it with the class name: Fruit::APPLE.

Note that because enum classes also act like namespaces, if we'd nested FruitType inside Fruit as an enum class instead of an enum, we'd access the APPLE enumerator via Fruit::FruitType::APPLE.

Other types can be nested too

Although enumerations are probably the most common type that is nested inside a class, C++ will let you define other types within a class, such as typedefs, type aliases, and even other classes!

Like any normal member of a class, nested classes have the same access to members of the enclosing class that the enclosing class does. However, the nested class does not have any special access to the "this" pointer of the enclosing class.

One other limitation of nested types -- they can't be forward declared. This limitation may be lifted in a future version of C++.

Defining nested classes isn't very common, but the C++ standard library does do so in some cases, such as with iterator classes. We'll cover iterators in a future lesson.

8.16 -- Timing your code

Index

8.14 -- Anonymous objects

C++ TUTORIAL | PRINT THIS POST

24 comments to 8.15 — Nested types in classes

nascardriver
August 28, 2019 at 4:26 am · Reply

Hello Alex!

A trap I just stumbled into: Nested types cannot be forward declared. eg.

```
// a.hpp
#include "b.hpp"

class C
{
public:
  enum class E{};
};
```

```
// b.hpp
// can't include a.hpp
// can't forward declare C::E
```

I think it's worth mentioning.

Alex
September 1, 2019 at 1:34 pm · Reply

I wasn't aware of that, but it makes sense, since the forward declared outer type is an incomplete type. Thanks! I learned something new today.

Brandon
August 10, 2019 at 9:14 am · Reply

If a class is divided into a header and .cpp file and you want to add an enum, which file should it go in?

nascardriver
August 10, 2019 at 9:27 am · Reply

Depends on the class and enum.
If the enum is only used in the source file, you can declare it inside the source file.
If the enum is used elsewhere, declare it in the header.
If the enum is considered a part of the class, declare it inside the class.

Abhijit
July 22, 2019 at 6:20 am · Reply

Can we put enum inside interface class?

nascardriver
July 22, 2019 at 6:40 am · Reply

yes

Atas
July 4, 2019 at 2:41 am · Reply

If my VS compiler is anything to go by then if we write

```
class Outer {
public:
    class Inner {
    } innerObject;
};
```

then innerObject becomes a member variable of any Outer object, but if we do this

```
class Outer {
public:
    class Inner {
    }
};
```

then we're just declaring a class in the Outer:: namespace that has access to Outer's innards, correct? Would it make sense to do the first variation and is it done often?

nascardriver
July 4, 2019 at 5:52 am · Reply

Both versions create an inner class that has access to the outer class. The first version immediately instantiates `Inner`. It's the same as

```
class Outer
{
public:

  class Inner
  {
  };

  Inner innerObject;
};
```

You'll find it a lot in C, especially with omitted type names, not so much in C++. Type and object declarations are easier to read when they're separate.

Atas
July 4, 2019 at 6:49 am · Reply

Thanks! It's just that the idea of a class being a member of another class weirds me out, I gotta have an object of the said class be the member as well :-) Thankfully, from what I've googled, nested classes don't seem to be that popular of a feature in C++.

Simon
July 29, 2018 at 5:05 pm · Reply

You should mention that you can forward declare enums. It could look like this:

```
class Fruit
{
    enum FruitType : int; //forward declaration

private:
    FruitType m_type;
    int m_percentageEaten{ 0 };

public:
    enum FruitType
    {
        APPLE,
        BANANA,
        CHERRY,
    };

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

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

int main()
{
    if (Fruit(Fruit::APPLE).getType() == Fruit::APPLE)
        std::cout << "I am an apple\n";
    else
        std::cout << "I am not an apple\n";
}
```

Yiu Chung WONG

January 17, 2019 at 1:11 am · Reply

I get a "Enumeration previously declared with fixed underlying type" error

nascardriver

January 17, 2019 at 8:59 am · Reply

The forward declaration of @FruitType has to have the same access specifier as it's definition (In this case, public), and line 10 has to specify the type.

```
#include <iostream>

class Fruit
{
public:
  enum FruitType : int;

private:
  FruitType m_type{ FruitType::NONE };
  int m_percentageEaten{ 0 };

public:
  enum FruitType : int
  {
    NONE,

    APPLE,
    BANANA,
    CHERRY,
  };

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

  FruitType getType(void) const
  {
    return m_type;
  }

  int getPercentageEaten(void) const
  {
    return m_percentageEaten;
  }
};

int main(void)
{
  if (Fruit{ Fruit::APPLE }.getType() == Fruit::APPLE)
  {
    std::cout << "I am an apple\n";
  }
  else
  {
    std::cout << "I am not an apple\n";
  }

  return 0;
}
```

Rohde Fischer
December 13, 2017 at 8:29 am · Reply

A common quite useful example of using nested classes (at least in the Java world) is builders. The basic idea is that some classes can get a lot of constructor arguments, and if you got a constructor signature looking like: Foo(string, string, string, string, string) it will be really hard knowing which string is which. If I'm not much mistaken in C++ it would look roughly like this:

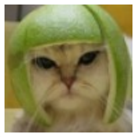
```
class Foo {
private:
  const std::string host;
  const std::string method;
  const std::string body;
  const std::string parameters;
  const std::string username;

  Foo(const std::string host,
      const std::string method,
      const std::string body,
      const std::string parameters,
      const std::string username)
    : host(host), method(method), body(body), parameters(parameters), username(username) {
  }

public:
  void doSomething() const {
    std::cout << host << " // " << method << " :: " << body << " ~~ " << parameters << " ==
  }

  class Builder {
  public:
    std::string host;
    std::string method;
    std::string body;
    std::string parameters;
    std::string username;

    Builder& withHost(std::string host) {
      this->host = host;
      return *this;
    }

    Builder& withMethod(std::string method) {
      this->method = method;
      return *this;
    }

    Builder& withBody(std::string body) {
      this->body = body;
      return *this;
    }

    Builder& withParameters(std::string parameters) {
      this->parameters = parameters;
```

```cpp
      return *this;
    }

    Builder& withUsername(std::string username) {
      this->username = username;
      return *this;
    }

    Foo build() {
      return Foo(host, method, body, parameters, username);
    }
  };
};
```

This is a lot of work when creating the class, but it tends to pay off in the long run, and especially if one is creating an api this is a lot nicer to use, since now the user can just:

```cpp
const Foo foo = Foo::Builder()
  .withHost("learncpp.com")
  .withHethod("GET")
  .withBody("Alex's work is fantastic, we love it!")
  .withParameters("kudos=100&receiver=Alex")
  .withUsername("rohdef")
  .build();

foo.doSomething();
```

Making this a lot more readable, it's independent of order and it makes creating factories a lot easier. Of course as in one of my other comments, it's worth discussing the tradeoffs. Because on any modern machine using this paradigm is of no consequence, unless one is doing something specifically intensive. However, on things like embedded devices this might be a very bad design. It is also an example utilizing the trick you present at: http://www.learncpp.com/cpp-tutorial/89-class-code-and-header-files/ with returning the class itself for creating a fluent API.



Alex
December 13, 2017 at 10:46 am · Reply

This is a great example. Thanks for sharing!

Rohde Fischer
December 14, 2017 at 7:54 am · Reply

You're welcome, thanks for a great tutorial. Of course as everything else this is matter of taste :) I've also seen people make containers for every singly type in question, basically simulating a typedef, but in a typesafe and quite bloated way. Not sure I'm a fan, but I can see the idea, as long as one use an IDE that helps with the types.

Udit
October 7, 2018 at 10:54 am · Reply

Hi sir,
I dont understand, how this example is working

```cpp
#include <iostream>

using namespace std;

```

```
 5  class test{
 6  public:
 7      test(){};
 8      void get_info(){
 9          cout << "Inside test::get_info\n";
10      }
11  };
12
13  int main(int argc, char const *argv[]){
14      test().get_info();
15      return 0;
16  }
```

According to my understanding there is only one way to call member function without creating an object, i.e. using 'static'. But how in this case we can explain that we are able to call 'get_info' without using object.

nascardriver
October 8, 2018 at 7:31 am · Reply

You're calling @test::test in line 14, instantiating an object of type @test.
@test::get_info is called on that object.

Udit
October 8, 2018 at 9:33 am · Reply

Ok, Its like anonymous object

John Halfyard
October 27, 2017 at 8:03 am · Reply

Hi Alex,

I'm not the best on the ol' "Passing by Reference, Const" and all that jazz (will have to review 7.1-7.4 a few times I guess). But in the code above:

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

why do you not place 'const' after the () for each function?

John

Alex
October 30, 2017 at 3:33 pm · Reply

Because it wasn't needed for the example. But you're right, these should be flagged as const functions so they could be called with a const object.

James
April 29, 2017 at 5:10 am · Reply

Good job Alex, but here am wondering how none static const private variable of the class Fruit "int m_percentageEaten = 0;" get to be initalized directly inside the class while you thought me in the previous lesson that only static const int and enum variable can be initialized inside the class that way. please

Teacher tell me how this work.
thanks

Alex
April 29, 2017 at 8:23 pm · Reply

Initialization for static and non-static members works differently. Static members can only be initialized in the class if they are ints or enums. Non-static members can be initialized inside the class regardless of the type.

john
March 19, 2017 at 4:20 am · Reply

Hi Alex! In the lesson on enumerators you mentioned we should prefer enum classes over enumerators if we have a C++11 compatible compiler since they are strongly typed and strongly scoped. What is the preferred way of using enumerations inside the class (using enum class complicates the access and we already limit the scope by having it inside the class)?

Alex
March 20, 2017 at 8:41 am · Reply

Personally, I tend to use normal enumerations inside classes, since the double-namespacing from the class and the enum class seems overkill. But it's really up to you. Enum classes have some additional advantages (e.g. you don't have to worry about enumerator naming collisions if you have multiple nested enum classes).