Why would one use nested classes in C++?

Can someone please point me towards some nice resources for understanding and using nested classes? I have some material like Programming Principles and things like this:

http://publib.boulder.ibm.com/infocenter/lnxpcomp/v8v101/index.jsp?topic=%2Fcom.ibm.xlcpp8l.doc%2Flanguage%2Fref%2Fcplr061.htm

But I'm still having trouble understaning their purpose. Could someone please help me ?

c++ nested inner-classes

edited Jul 3 '14 at 6:34
Ronald Wildenberg 19.5k ● 3 ● 50 ● 87

asked Dec 31 '10 at 17:14
Supraja Jayakumar 378 ● 1 ● 8 ● 17

5 My advice for nested classes in C++ is simply to not use nested classes. – Billy ONeal Dec 31 '10 at 17:18

2 They're exactly like regular classes... except nested. Use them when a class's internal implementation is so complex that it can most easily be modeled by several smaller classes. – meagar Dec 31 '10 at 17:24

7 @Billy: Why? Seems overly broad to me. – John Dibling Dec 31 '10 at 17:29

15 I still haven't seen an argument why nested classes are bad by their nature. – John Dibling Dec 31 '10 at 17:55

4 @Billy: What about name scoping, as in my example below? Is not the expression `Field::match(...)` fairly natural and self-documenting? Of course this isn't *necesarry*, one could declare a namespace-scoped class like `match_field` just as easily, but that doesn't make the nested class **bad**, just another way to skin the cat. – John Dibling Dec 31 '10 at 17:58

3 Answers

Nested classes are cool for hiding implementation details

List:

```
class List
{
    public:
        List(): head(NULL), tail(NULL) {}
    private:
        class Node
        {
              public:
                  int   data;
                  Node* next;
                  Node* prev;
        };
    private:
        Node*     head;
        Node*     tail;
};
```

Here I don't want to expose Node as other people may decide to use the class and that would hinder me from updating my class as anything exposed is part of the public API and must be maintained **forever**. By making the class private, I not only hide the implementation I am also saying this is mine and I may change it at any time so you can not use it.

Look at std::list or std::map they all contain hidden classes (or do they?). The point is they may or may not, but because the implementation is private and hidden the builders of the STL were able to update the code without affecting how you used the code, or leaving a lot of old baggage laying around the STL because they need to maintain backwards compatibility with some fool who decided they wanted to use the Node class that was hidden inside <list>.

edited Aug 29 '14 at 5:57

answered Dec 31 '10 at 18:27

Stephan
2,519 • 7 • 25

Loki Astari
126k • 31 • 180 • 344

3 If you're doing this then `Node` shouldn't be exposed in the header file at all. – Billy ONeal Dec 31 '10 at 19:08

3 @Billy ONeal: What if I am doing a header file implementation like the STL or boost. – Loki Astari Dec 31 '10 at 19:29

5 @Billy ONeal: It protects it from accidental use. It also documents the fact that it is private and should not be used (can not be used unless you do something stupid). Thus you do not need to support it. Putting it in a namespace makes it part of the public API (something you keep missing in this conversation. Public API means you need to support it). – Loki Astari Dec 31 '10 at 22:36

1 @Martin York: Err... convention is that anything in a namespace `detail` is not public for clients to use. – Billy ONeal Jan 1 '11 at 4:21

1 @Billy ONeal: I like that convention and will start using it. But its the first time I have heard it. Is it a boost convention that is documented in their site. – Loki Astari Jan 1 '11 at 19:30

Nested classes are just like regular classes, but:

- they have additional access restriction (as all definitions inside a class definition do),
- they **don't pollute the given namespace**, e.g. global namespace. If you feel that class B is so deeply connected to class A, but the objects of A and B are not necessarily related, then you might want the class B to be only accessible via scoping the A class (it would be referred to as A::Class).

Some examples:

Publicly nesting class to put it in a scope of relevant class

Assume you want to have a class `SomeSpecificCollection` which would aggregate objects of class `Element`. You can then either:

1. declare two classes: `SomeSpecificCollection` and `Element` - bad, because the name "Element" is general enough in order to cause a possibly name clash
2. introduce a namespace `someSpecificCollection` and declare classes `someSpecificCollection::Collection` and `someSpecificCollection::Element`. No risk of name clash, but can it get any more verbose?
3. declare two global classes `SomeSpecificCollection` and `SomeSpecificCollectionElement` - which has minor drawbacks, but is probably OK.
4. declare global class `SomeSpecificCollection` and class `Element` as its nested class. Then:
   - you don't risk any name clases as Element is not in the global namespace,
   - in implementation of `SomeSpecificCollection` you refer to just `Element`, and everywhere else as `SomeSpecificCollection::Element` - which looks +- the same as 3., but more clear
   - it gets plain simple that it's "an element of a specific collection", not "a specific element of a collection"
   - it is visible that `SomeSpecificCollection` is also a class.

In my opinion, the last variant is definitely the most intuitive and hence best design.

Let me stress - It's not a big difference from making two global classes with more verbose names. It just a tiny little detail, but imho it makes the code more clear.

Introducing another scope inside a class scope

This is especially useful for introducing typedefs or enums. I'll just post a code example here:

```
class Product {
public:
    enum ProductType {
        FANCY, AWESOME, USEFUL
    };
    enum ProductBoxType {
        BOX, BAG, CRATE
    };
    Product(ProductType t, ProductBoxType b, String name);

    // the rest of the class: fields, methods
};
```

One then will call:

```
Product p(Product::FANCY, Product::BOX);
```

But when looking at code completion proposals for `Product::`, one will often get all the possible enum values (BOX, FANCY, CRATE) listed and it's easy to make a mistake here (C++0x's strongly typed enums kind of solve that, but never mind).

But if you introduce additional scope for those enums using nested classes, things could look like:

```
class Product {
public:
    struct ProductType {
        enum Enum { FANCY, AWESOME, USEFUL };
    };
    struct ProductBoxType {
        enum Enum { BOX, BAG, CRATE };
    };
    Product(ProductType::Enum t, ProductBoxType::Enum b, String name);

    // the rest of the class: fields, methods
};
```

Then the call looks like:

```
Product p(Product::ProductType::FANCY, Product::ProductBoxType::BOX);
```

Then by typing `Product::ProductType::` in an IDE, one will get only the enums from the desired scope suggested. This also reduces the risk of making a mistake.

Of course this may not be needed for small classes, but if one has a lot of enums, then it makes things easier for the client programmers.

In the same way, you could "organise" a big bunch of typedefs in a template, if you ever had the need to. It's an useful pattern sometimes.

The PIMPL idiom

The PIMPL (short for Private IMPLementation) is an idiom useful to remove the implementation details of a class from the header. This reduces the need of recompiling classes depending on the class' header whenever the "implementation" part of the header changes.

It's usually implemented using a nested class:

X.h:

```
class X {
public:
    X();
    virtual ~X();
    void publicInterface();
    void publicInterface2();
private:
    struct Impl;
    std::unique_ptr<Impl> impl;
}
```

X.cpp:

```
#include "X.h"
#include <windows.h>

struct X::Impl {
    HWND hWnd; // this field is a part of the class, but no need to include windows.h
in header
    // all private fields, methods go here

    void privateMethod(HWND wnd);
    void privateMethod();
};

X::X() : impl(new Impl()) {
    // ...
}
```

```
// and the rest of definitions go here
```

This is particularly useful if the full class definition needs the definition of types from some external library which has a heavy or just ugly header file (take WinAPI). If you use PIMPL, then you can enclose any WinAPI-specific functionality only in `.cpp` and never include it in `.h`.

edited Jul 4 '14 at 7:17

answered Dec 31 '10 at 19:12
Kos
31.8k ● 9 ● 69 ● 142

1 `struct Impl; std::auto_ptr<Impl> impl;` This error was popularized by Herb Sutter. Don't use auto_ptr on incomplete types, or at least take precautions to avoid wrong code being generated. – Gene Bushuyev Dec 31 '10 at 20:00

@Gene: What do you mean "wrong code"? I was not aware `auto_ptr` didn't work on incomplete types... – Billy ONeal Dec 31 '10 at 20:39

@Billy ONeal: As far as I am aware you can declare an `auto_ptr` of incomplete type in most implementations but technically it is UB unlike some of the templates in C++0x (e.g. `unique_ptr`) where it has been made explicit that the template parameter may be an incomplete type and where exactly the type must be complete. (e.g. use of `~unique_ptr`) – Charles Bailey Jan 1 '11 at 11:29

1 @Billy ONeal: In C++03 17.4.6.3 [lib.res.on.functions] says "In particular, the effects are undefined in the following cases: [...] if an incomplete type is used as a template argument when instantiating a template component." whereas in C++0x it says "if an incomplete type is used as a template argument when instantiating a template component, unless specifically allowed for that component." and later (e.g.): "The template parameter `T` of `unique_ptr` may be an incomplete type." – Charles Bailey Jan 1 '11 at 11:33

1 You can't force others not to inherit anyway, and you should never try to enforce a "no inheriting" rule. – Miles Rout Mar 16 '13 at 23:50

I don't use nested classes much, but I do use then now and then. Especially when I define some kind of data type, and then I want to define and STL functor designed for that data type.

For example, consider a generic `Field` class that has an ID number, a type code and a field name. If I want to search a `vector` of these `Field`s by either ID number or name, I might construct a functor to do so:

```
class Field
{
public:
  unsigned id_;
  string name_;
  unsigned type_;

  class match : public std::unary_function<bool, Field>
  {
  public:
    match(const string& name) : name_(name), has_name_(true) {};
    match(unsigned id) : id_(id), has_id_(true) {};
    bool operator()(const Field& rhs) const
    {
      bool ret = true;
      if( ret && has_id_ ) ret = id_ == rhs.id_;
      if( ret && has_name_ ) ret = name_ == rhs.name_;
      return ret;
    };
    private:
      unsigned id_;
      bool has_id_;
      string name_;
      bool has_name_;
  };
};
```

Then code that needs to search for these `Field`s can use the `match` scoped within the `Field` class itself:

```
vector<Field>::const_iterator it = find_if(fields.begin(), fields.end(),
Field::match("FieldName"));
```

edited Dec 31 '10 at 17:33

answered Dec 31 '10 at 17:26
John Dibling
60.9k ● 8 ● 90 ● 201

Fixed a couple missing bits. – John Dibling Dec 31 '10 at 17:34

2 +1 I'm not sure why this was downvoted, it is a good example. – Sam Miller Dec 31 '10 at 17:54

Even though I don't think this is a good example I'm not the downvoter. – Billy ONeal Dec 31 '10 at 19:07

1 @user: In the case of an STL functor, the constructor does need to be public. – John Dibling Dec 31 '10 at 19:41

1 @Billy: I *still* have yet to see any concrete reasoning why nested classes are bad. – John Dibling Dec 31 '10 at 19:55