C++ Foundation

Program Organization and Dependency Management

Program Organization and Dependency Management

- namespace, “packages”
- using directives and declarations
- explicit qualification
- header files and source files
- unnecessary #includes
- Koenig lookup, argument dependent lookup
- forward declarations
- dependency injection

Namespaces

- A class is not a useful unit of design!
- Collaborating classes are, and can live inside a package, a named scope

grammar_lib/

- non_terminal.hpp
- production.hpp
- production_entry.hpp
- terminal.hpp
- ...

```
namespace grammar_lib
{
    class non_terminal
    {   ...
    };
}
```

```
namespace grammar_lib
{
    class production
    {   ...
    };
}
```

Header Guards

- Always use macro guards to ensure header files are idempotent (beware copy & paste)
- Make header macro guards reflect the folder/namespace name *and* the file/class name

grammar_lib/non_terminal.hpp

```
#ifndef GRAMMAR_LIB_NON_TERMINAL_HPP_INCLUDED
#define GRAMMAR_LIB_NON_TERMINAL_HPP_INCLUDED

...

#endif
```

Using Directives

- Pulls in *all* names into the current scope

```
namespace std
{
    class ostream { ... };
    ostream & endl(ostream &);
    extern ostream cout;
}
```

iostream

```
#include <iostream>

using namespace std;

void eg()
{
   cout << "Hello" << endl;
}
```

using directive

Using Declaration

- Pulls a *specific* name into the current scope

```
namespace std
{
    class ostream { ... };
    ostream & endl(ostream &);
    extern ostream cout;
}
```

iostream

```
#include <iostream>

using std::cout;

void eg()
{
   cout << "Hello" << std::endl;
}
```

using declaration

Explicit::Qualification

- A fully scope qualified name

iostream

```
namespace std
{
    class ostream { ... };
    ostream & endl(ostream &);
    extern ostream cout;
}
```

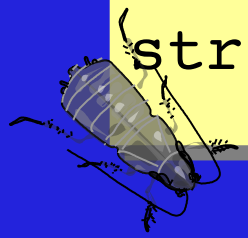
```
#include <iostream>

void eg()
{
   std::cout << “Hello” << std::endl;
}
```

Headers are meant to be included

- Using directives/declarations in a header will have an unknown span of effect - which entirely defeats the purpose of namespaces!
- In a header file use *only* explicit qualification

header.hpp

```
#include <string>

using namespace std;

string bad();
```



header.hpp

```
#include <string>

std::string good();
```

A Puzzle...

- (a << b) is syntactic sugar for operator<<(a, b)

```
std::operator<<(std::cout, “Hello\n”);
```

But we *aren't* required to qualify operator<< with std:: ?

```
#include <iostream>

void eg()
{
    std::cout << “Hello\n”;
}
```

Argument Dependent Lookup

- The compiler can look for the function in the namespaces of its arguments
- ADL - also known as Koenig lookup

```
namespace std
{
    ... operator<<(std::ostream &, const char *);
}
```

```
std::cout << “Hello, world\n”
```

Quiz

- Will this compile?
- If not, why not?

```
#ifndef GRAMMAR_LIB_GRAMMAR_HPP_INCLUDED
#define GRAMMAR_LIB_GRAMMAR_HPP_INCLUDED

namespace grammar_lib
{
    class grammar
    {
    public:
        //...
        void insert(non_terminal *);
    };
}

#endif
```

Answer

- No - the compiler does not magically somehow know an identifier is the name of a type

```
#ifndef GRAMMAR_LIB_GRAMMAR_HPP_INCLUDED
#define GRAMMAR_LIB_GRAMMAR_HPP_INCLUDED

namespace grammar_lib
{
    class grammar
    {
    public:
        //...
        void insert(non_terminal *);
    };
}

#endif
```

One Solution

- Given this header file...
- ...add a #include

non_terminal.hpp

```
namespace grammar_lib
{
    class non_terminal
    {
    public:
        //...
    };
}
```

```
#include "non_terminal.hpp"

namespace grammar_lib
{
    class grammar
    {
    public:
        ...
        void insert(non_terminal *);
    };
}
```

Another Solution

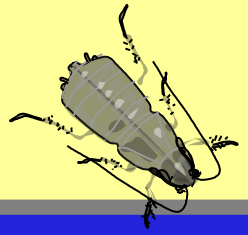
- Use a *forward declaration* - tell the compiler *only* that the identifier is the name of a class
- Reduces/exposes include dependencies :-)

```
#include ...

namespace grammar_lib
{
    class non_terminal;

    class grammar
    {
    public:
        ...
        void insert(non_terminal *);
    };
}
```

However...

- What you promise in a forward declaration has to *exactly* match the definition

```
namespace std
{
    class string;
}

namespace grammar_lib
{
    class grammar
    {
    public:
        grammar(const std::string & name);
        ...
    };
}
```



And sometimes...it doesn't

- You can't forward declare string
  - You have to #include <string>
- You can't forward declare stream classes
  - You can #include <iosfwd> though

```
namespace std
{
    template<typename CharType, ...>
    class basic_string
    {
        ...
    };

    typedef basic_string<char,...> string;
}
```

Quiz - 9 cases

- Which need a #include “wibble.hpp” and which only need a forward declaration of wibble?

```
class nine_cases
{
    void   one(wibble  );
    void   two(wibble *);
    void three(wibble &);

    wibble   four();
    wibble * five();
    wibble & six();

    wibble   seven;
    wibble * eight;
    wibble & nine;
};
```

Please discuss in your groups

Answer

- Only case seven requires a #include!!

  A forward declaration is sufficient for all others

```
class nine_cases
{
    void   one(wibble  );
    void   two(wibble *);
    void three(wibble &);

    wibble   four();
    wibble * five();
    wibble & six();

    wibble   seven;
    wibble * eight;
    wibble & nine;
};
```

one-six are declarations
not definitions

Self-Contained Header Files

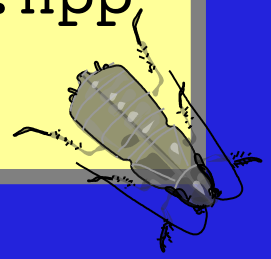
- To include one header you should *never* need to include another header
- Make the first #include in each source file its *own* header

wibble.hpp

```
class fubar;

class wibble
{
public:
    void (fubar *);
    ...
};
```

wibble.cpp

```
#include "fubar.hpp"
#include "wibble.hpp"
...
```



wibble.cpp

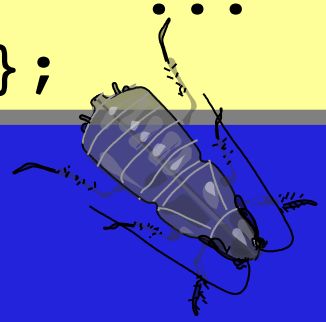
```
#include "wibble.hpp"
#include "fubar.hpp"
...
```

Unself-Contained Header Files

- To include some header files you need to include other header files first... :-(

wibble.hpp

```
class wibble
{
public:
    void (fubar *);
    ...
};
```



this header file does not forward declare fubar or #include fubar.hpp

...

but the problem is avoided like this...

```
#include "fubar.hpp"
#include "wibble.hpp"
...
```

source.cpp

Design is about being...

- Easy to use
  - clean abstractions that hide unimportant details
  - tests are examples of use - they shepherd design
- Easy to test
  - if you don't write tests you *will* end up with software that is hard to test - and that is *not* surprising
  - testability is a *key* criteria of design
- Easy to maintain
  - you are constantly battling against entropy
  - tests act as a safety net and encourage refactoring

Testing

- System test
  - all external dependencies in place
- Integration test
  - some external dependencies in place
  - some external dependencies mocked out
- Unit-test
  - all external dependencies mocked out
  - *Reliability* - no false positive/negative passes/fails
  - *Speed* - running unit-tests becomes the driver of how you program

Header File Summary

- Mirror the folder/namespace and file/class names in the macro guards
- Always use explicit qualification
- Use forward declarations when you can
- Use #include's only when you have to
- Ensure every header is self-contained; compilable in its own right
- Header files and tests represent the design
  source files are somewhat incidental!
  the tests tell us if they work
  and we don't have a choice about that!