Name Lookup and the Interface Principle

```
namespace A {
    struct X;
    void g( X );
}
namespace B {
    void g( A::X x ) {
        g( x ); // which g()?
    }
}
```

Krishna Kumar

Name Lookup

- When you call a function, which function do you call? The answer is determined by name lookup, but you're almost certain to find some of the details surprising.
- In the following code, which functions are called? Why?

- namespace A {
  - struct X;
  - struct Y;
  - void f(int);
  - void g(X);
- }
- namespace B {
  - void f(int i) { f(i);} // which f()?
  - void g(A::X x) { g(x); }
    // which g()?
  - void h(A::Y y) { h(y); }
    // which h()?
- }

Name lookup

- **Let's start simple**.
- namespace A {
  - struct X;
  - void f(int);
- }
- namespace B {
  - void f(int i) { f(i); } // which f()?
- }
- This f() calls itself, with infinite recursion. The reason is that the only visible f() is B::f() itself.
- There is another function with signature f(int) , namely the one in namespace A . If B had written "using namespace A; " or " using A::f; ", then A::f(int) would have been visible as a candidate when looking up f(int) , and the f(i) call would have been ambiguous between A::f(int) and B::f(int) . Since B did not bring A::f(int) into scope, however, only B::f(int) can be considered, so the call unambiguously resolves to B::f(int) .

Koenig Lookup

- A good knowledge of C++'s name lookup rules—in particular, **Koenig lookup (argument-dependent lookup – ADL)** is required**.**
- void g(A::X x) {
  - g(x); // which g()?
- }
- This call is ambiguous between A::g(X) and B::g(X) . The programmer must explicitly qualify the call with the appropriate namespace name to get the g() they want.
- You may at first wonder why this should be so. After all, as with f() , **B** hasn't written " using " anywhere to bring A::g(X) into its scope, so you'd think that only B::g(X) would be visible, right? Well, this would true except for an extra rule that C++ uses when looking up names.

Koenig Lookup (cont...)

- Koenig Lookup (simplified):
  - *“If you supply a function argument of class type (here x , of type A::X), then to look up the correct function name the compiler considers matching names in the namespace (here A) containing the argument's type.”*
- Here is an example:
- namespace std {
  - class string { };
  - operator<<(T);
- }
- int main() {
  - std::string parm = “Hello world!”;
  - std::cout << parm ; // OK, calls std::string's <<
- }

Name lookup (cont)

- Now back to a simple example:
- void h(A::Y y) {
  - h(y); // which h()?
- }
- There is no other h(A::Y) , so this h() calls itself with infinite recursion.
- Although B::h() 's signature mentions a type found in namespace A , this doesn't affect name lookup because there are no functions in A matching the name and signature h(A::Y) .
- So, what does it mean? In short, the meaning of code in namespace B is being affected by a function declared in the completely separate namespace A , even though B has done nothing but simply mention a type found in A and there's no " using " in sight.
- What this means is that namespaces aren't quite as independent as people originally thought. Don't stardenouncing namespaces just yet, though; namespaces are still pretty independent, and they fit their intended uses to a T.

Class interface

- A traditional definition of a class:
  - A class describes a set of data, along with the functions that operate on that data.
- Question: What functions are "part of " a class, or make up the interface of a class?
  - Hint #1: Clearly nonstatic member functions are tightly coupled to the class, and public nonstatic member functions make up part of the class's interface. What about static member functions? What about free functions?
  - Hint #2: consider the implications of name lookup.

Class interface

A class describes a set of data, along with the functions that operate on that data.

- Programmers often unconsciously misinterpret this definition, saying instead: "Oh yeah, a class, that's what appears in the class definition—the member data and the member functions." But that's not the same thing, because it limits the word functions to mean just member functions. Consider:
- ***class X { /*...*/ };***
- ***void f(const X&);***
- The question is: Is **f** part of **X** ? Some people will automatically say "No" because **f** is a nonmember function (or free function). Others might realize something fundamentally important: If the code appears together in one header file, it is not significantly different from:
- **class X {**
  - ***/*...*/***
  - **public:**

    void f() const;
- **};**
- Besides access rights, **f** is still the same, taking a pointer/reference to **X** . The *this* parameter is just implicit in the second version, that's all.

Class interface principle

- So, if example code all appears in the same header, we're already starting to see that even though **f** is not a member of ***X***, it's nonetheless strongly related to **X** .
- On the other hand, if **X** and **f** do not appear together in the same header file, then f is just some old client function, not a part of **X** (even if **f** is intended to augment **X**). We routinely write functions with parameters whose types come from library headers, and clearly our custom functions aren't part of those library classes.
- With that example in mind, Interface Principle is:
  - For a class **X**, all functions, including free functions, that both
    - "Mention" **X**
    - Are "supplied with" **X**
  - are logically part of X, because they form part of the interface of X.
- By definition, every member function is "part of" X :
  - Every member function must "mention" X - a nonstatic member function has an implicit this parameter of type X* const or const X* const ; a static member function is in the scope of X
  - Every member function must be "supplied with" X (in X 's definition).

Class interface principle (cont...)

- Applying the Interface Principle to the example code gives the same result as our original analysis. Clearly, **f** mentions **X** If **f** is also "supplied with" **X** (same header file and/or namespace), then according to the Interface Principle, **f** is logically part of **X** because it forms part of the interface of **X**.
- Interface Principle behaves in exactly the same way that Koenig lookup does.
- So the Interface Principle is a useful touchstone to determine what is really "part of" a class. Is it unintuitive that a free function should be considered part of a class? Then let's give real weight to this example by giving a more common name to **f** .
- class X { /*...*/ };
- /*...*/
- ostream& operator<<(ostream&, const X&);
- Here the Interface Principle's rationale is perfectly clear, because we understand how this particular freefunction works. If operator**<<** is "supplied with" X (for example, in the same header and/or namespace), then **operator<<** is logically part of X because it forms part of the interface of **X** . That makes sense even though the function is a nonmember, because we know that it's common practice for a class's author to provide **operator<<** . If, instead, **operator<<** comes, not from **X** 's author, but from client code, then it's not part of **X** because it's not "supplied with" **X** .

OO Interface principle

- // Example 1
- struct _iobuf { /*...data goes here...*/ };
- typedef struct _iobuf FILE;
- FILE* fopen (const char* filename, const char* mode);
- Int fclose(FILE* stream);
- Int fseek (FILE* stream,
  - long offset, int origin);
- long ftell (FILE* stream);
- /* etc. */

- // Example 2
- class FILE {
- public:
  - FILE(const char* filename, const char* mode);
  - ~FILE();
  - int fseek(long offset, int origin);
  - long ftell();
- private:
  - /*...data goes here...*/
- };

FILE* parameters have just become implicit this parameters. Here it's clear that fseek is part of FILE , just as it was in Example 1, even though there it was a nonmember. We can even merrily make some functions members and some not.

More Koenig Lookup

- namespace NS {
  - class T { }; // some header
- }
- void f(NS::T);
- int main() {
  - NS::T parm;
  - f(parm); // OK: calls global f
- }

- **namespace NS {**
  - **class T { };**
  - **void f(T); // <-- new function**
- **}**
- **void f(NS::T);**
- **int main() {**
  - **NS::T parm;**
  - **f(parm);**
  - **// ambiguous: NS::f or Global f?**
- **}**

Adding a function in a namespace scope "broke" code outside the namespace, even though the client code didn't write using to bring NS 's names into its scope!

More Koenig lookup (Myer's example)

- Namespace A {
  - class X {};
- }
- namespace B {
  - void f(A::X);
  - void g(A::X parm) {
    - f(parm);
    - // OK: calls B::f
  - }
- }

- namespace A {
  - class X { };
  - void f(X); // <-- new function
- }
- namespace B {
  - void f(A::X);
  - void g(A::X parm) {
    - f(parm);
    - // ambiguous: A::f or B::f?
  - }
- }

More Koenig loopup (Myer's)

- "The whole point of namespaces is to prevent name collisions, isn't it? But adding a function in one namespace actually seems to 'break' code in a completely separate namespace."
- True, namespace **B** 's code seems to break merely because it mentions a type from **A** . **B** 's code didn't write a using namespace **A**; anywhere. It didn't even write using **A::X**;
- This is not a problem, and **B** is not broken. This is in fact exactly what should happen. If there's a function **f(X)** in the same namespace as **X** , then, according to the Interface Principle, f is part of the interface of **X** .
- It doesn't matter that **f** happens to be a free function; to see clearly that it's nonetheless logically part of **X** , just give it another name.

Interface principle

- **In general, if A and B are classes and A::g(B) is a member function of A :**
- Because A::g(B) exists, clearly A always depends on B . No surprises so far.
- If A and B are supplied together, then of course A::g(B) and B are supplied together. Therefore, because A::g(B) both "mentions" B and is "supplied with" B , then according to the Interface Principle, it follows that A::g(B) is part of B , and because A::g(B) uses an (implicit) A* parameter, B depends on A . Because A also depends on B , this means that A and B are interdependent.
- At first, it might seem like a stretch to consider a member function of one class as also part of another class, but this is true only if A and B are also supplied together.
- **Unlike classes, namespaces don't need to be declared all at once, and what's "supplied together" depends on what parts of the namespace are visible.**
- //---file a.h---
- namespace N { class B; }// forward decl
- namespace N { class A; }// forward decl
- class N::A { public: void g(B); }; In a.h: A and B are supplied together and are interdependent.
- //---file b.h---
- namespace N { class B { /*...*/ }; } Clients of B include b.h , A and B are not supplied together.

Name Hiding

- *// from a base class*
- **struct B {**
  - **int f(int);**
  - **int f(double);**
  - **int g(int);**
- **};**
- **struct D : public B {**
  private:
  - **int g(std::string, bool);**
- **};**
- **D d;**
- **int i;**
- **d.f(i);** **// ok, means B::f(int)**
- **d.g(i);** **// error: g takes 2 args**

- In short, when we declare a function named **g** in the derived **class D** , it automatically hides all functions with the same name in all direct and indirect base classes. It
- doesn't matter a whit that **D::g** "obviously" can't be the function that the programmer meant to call (not only does **D::g** have the wrong signature, but it's private and, therefore, inaccessible to boot), because **B::g** is hidden and can't be considered by name lookup.

Name Hiding

- To see what's really going on, let's look in a little more detail at what the compiler does when it encounters the function call **d.g(i)**.
- First, it looks in the immediate scope, in this case the scope of class **D**, and makes a list of all functions it can find that are named **g** (regardless of whether they're accessible or even take the right number of parameters). Only if it doesn't find any at all does it then continue "outward" into the next enclosing scope and repeat—in this case, the scope of the base class **B** — until it eventually either runs out of scopes without having found a function with the right name or else finds a scope that contains at least one candidate function.
- If a scope is found that has one or more candidate functions, the compiler then stops searching and works with the candidates that it's found, performing overload resolution and then applying access rules.
- It makes intuitive sense that a member function that's a near-exact match ought to be preferred over a global function that would have been a perfect match had we considered the parameter types only.

References

- Exceptional C++ - Herb Sutter
-