Name Resolution with `typedef` and `using`

Table of Contents

Introduction

- **Type aliases** are synonyms for existing types. They improve readability by giving meaningful names to types (especially complex ones).
- Aliases can be declared in **global, block, class, or namespace** scope. This means you can have aliases inside functions, inside classes as member typedefs, or at namespace/global scope.
- In modern C++, `using` (since C++11) is preferred. It can be used exactly like `typedef` for simple aliases and also to create *alias templates* for generic types.

Name Resolution Rules

- **Uniform lookup rules:** The compiler treats alias names just like any other name. In other words, aliases obey the same C++ scope and hiding rules as class or namespace names.
- **Scope and hiding:** An alias's visibility depends on where it's declared. A local declaration can hide an alias from an outer scope. For example, if a typedef `X` exists globally, a local variable `X` inside a function will shadow that alias. To refer to an alias from an outer scope, you must qualify it (e.g. `Outer::AliasName`).
- **Dependent names in templates:** When using a type alias inside a template (as a dependent name), you still use `typename` as needed, just as with any nested type.
- **Argument-dependent lookup (ADL):** When calling functions, C++ considers the *underlying types* of arguments, not alias names. **Typedefs/`using` aliases do not contribute to ADL.**

typedef vs. using

- **Equivalence (for simple aliases):** For basic aliases, `typedef` and `using` are equivalent. Example: `typedef int T1;` and `using T2 = int;` both make a synonym for `int`. The choice is syntactic.
- **Readability:** `using` syntax is often clearer to read (aliases appear left-to-right). With `typedef`, the alias name is at the end of the declaration, which can be confusing for complex types.
- **Templates:** The key advantage of `using` is alias templates. You *cannot* create a templated alias with `typedef`; `using` can declare alias templates (e.g. `template<class T> using Vec = std::vector<T>;`).

Example 1: typedef alias for a simple type

```
#include <iostream>

// Example 1: typedef alias for a simple type
typedef int myint;

int main() {
    myint x = 5;
    std::cout << "myint x = " << x << "\n";
    return 0;
}
```

Explanation: `typedef int myint;` creates a new name `myint` that's exactly equivalent to `int`. In `main()`, `myint x = 5;` declares an integer variable. This always compiles because `typedef` does not introduce a new type, just an alias.

Example 2: using alias for a template

```cpp
#include <iostream>
#include <vector>

// Example 2: using alias for a template
using myvec = std::vector<int>;

int main() {
    myvec v = {1, 2, 3};
    std::cout << "myvec size = " << v.size() <<
        "\n";
    return 0;
}
```

Explanation: `using myvec = std::vector<int>;` defines an alias `myvec` for `std::vector<int>`. It's more readable than a `typedef` with templates. The code compiles and prints the vector's size because `myvec` behaves exactly like the instantiated template.

Example 3: Name conflict with a using-declaration

```
#include <iostream>

namespace A { int x; }
using A::x; // brings A::x into the global scope

int main() {
    int x = 1;
    std::cout << x;
    return 0;
}
```

Explanation: `using A::x;` imports `A::x` into the global namespace. Declaring your own `x` causes ambiguity between `A::x` and your local `x`, leading to a compile-time error.

Example 4: Ambiguity with `using namespace`

```
#include <iostream>

namespace B { void f(); }
using namespace B;

void f(int) { std::cout << "f(int)\n"; }

int main() {
    f(); // error: ambiguous between B::f and ::
        f(int)
    return 0;
}
```

Explanation: `using namespace B;` brings all of `B` into the global scope. Since `B` has `void f()` and you define `void f(int)`, the call `f()` is ambiguous.

Example 5: Template alias via typedef (fails)

```
template<typename T>
typedef T* PtrTypedef; // Error
```

Explanation: We try to use template alias with typedef. Typedef cannot be used with templates.

Example 6: Template alias via using (works)

```
template<typename T>
using PtrUsing = T*;

int main() {
    PtrUsing<int> p = new int(123); // OK
    delete p;
}
```

Explanation: However, it is possible with using.

Example 7: Qualified name lookup

```
// Define a function inside namespace M
namespace M {
    void g() { std::cout << "M::g called\n"; }
}

int main() {
    M::g();     // call the function with its
        full qualification
    return 0;
}
```

Explanation: Using the qualified name ‘M::g()‘ tells the compiler exactly which ‘g‘ to invoke—namely the one defined in namespace ‘M‘.

Example 8: typedef vs using for a complex type

```
#include <iostream>
#include <map>
#include <vector>
#include <string>

// Example 8: typedef vs using for a complex
    type
typedef std::map<std::string, std::vector<int>>
    MapVec;
using M2 = std::map<std::string, std::vector<int
    >>;

int main() {
     MapVec mv;
     M2 m2;
     std::cout << "Complex aliases created." << "
         \n";
     return 0;
}
```

Example 8: `typedef` vs `using` for a complex type

Explanation: Both `MapVec` and `M2` alias the same complex type. `using` syntax is generally clearer as type complexity grows.

Example 9: `using`-declaration for a type

```
#include <iostream>

namespace C { typedef double real; }
using C::real;

void use_real(real r) { std::cout << "real: " <<
    r << "\n"; }

int main() {
    use_real(3.14);
    return 0;
}
```

Explanation: `using C::real;` imports `C::real` into the global scope as double.

Example 10: Implicit conflicts with `using namespace`

```cpp
#include <iostream>

namespace D {
    struct Foo {};
    void Foo() { }
}
using namespace D;

int main() {
    Foo foo_obj; // ambiguous: type vs. function
        ( most  vexing parse )
    return 0;
}
```

Explanation: Importing both a type and a function named `Foo` causes parsing ambiguity and potential "most vexing parse" issues.

Grammar update

```
DTypedef.    Def ::= "typedef" Type Id ";" ;
             -- typedef int MyInt;
DUsing.      Def ::= "using" Id "=" Type ";" ;
           -- using MyInt = int;
DUsingNs.    Def ::= "using" "namespace" QualId "
    ;" ; -- using namespace std;
DUsingSymbol. Def ::= "using" Id ; -- using std
    ::cout;
DNamespace. Def ::= "namespace" Id "{" [Def] "}"
     ;   -- namespace foo { ... }
```

Visit functions

```
void visitSimpleId(SimpleId *p);
void visitQualIdNs(QualIdNs *p);
void visitDTypedef(DTypedef *p); // use AddAlias
     or AddLocalAlias to add the alias to the
   appropriate scope
void visitDUsing(DUsing *p); // use AddAlias or
   AddLocalAlias to add the alias to the
   appropriate scope
void visitDUsingNs(DUsingNs *p);
void visitDUsingSymbol(DUsingSymbol *p); //
   Handle using declarations for symbols
void visitDNamespace(DNamespace *p); //add
   namespace to globals
void visitTypeQualId(TypeQualId *p); // Checks
   if it's a struct/class or if it's in aliases
```

Helper structures

```
struct AliasType {
    const Type* aliasedType;
    Id name;
};
using AliasTable = SymbolTable<AliasType>;

struct Globals {
    FnTable fns;
    StTable sts;
    AliasTable aliases; // Store aliases for
        both typedefs and using
    std::map<Id, NamespaceInfo> namespaces;
    std::vector<NamespaceInfo> usedNamespaces;
        // Namespaces included with using
};
```

Helper structures

```
struct Context {
    Globals globals;
    std::vector<Scope> vars;
    AliasTable localAliases; // Local aliases
        for the current scope
    std::vector<NamespaceInfo> localNamespaces;
        // Local namespaces for the current scope
};
```

visitTypeQualId

```
void visitTypeQualId(TypeId *p)
{
    //Existing logic for visiting TypeQualId
    auto alias = findAlias(p->id);
    if(alias) {
        lastType = alias->aliasedType;
    }
}
```

Aliases

```
void AddAlias(const Id& id, const Type* type,
    AliasTable& aliasTable) { // called whenever
    typedef or using is used
     if (aliasTable.find(id) != aliasTable.end())
          {
          std::cerr << "TYPE ERROR: Alias '" << id
               << "' already exists." << std::endl;
          exit(1);
     }
     if (context.globals.aliases.find(id) !=
         context.globals.aliases.end()) {
          std::cerr << "TYPE ERROR: Alias '" << id
               << "' already exists." << std::endl;
          exit(1);
     }
     aliasTable[id] = AliasType{type, id};
}
```

Aliases

```
const AliasType* findAlias(const QualifiedId&
    qualId) {
     // Check local aliases first (simple names
         only)
     if (qualId.find("::") == std::string::npos)
         {  // Simple name
          auto it = context.localAliases.find(
              qualId);
          if (it != context.localAliases.end()) {
               return &it->second;
          }
     }
     // Check global aliases
     auto it = context.globals.aliases.find(
         qualId);
     if (it != context.globals.aliases.end()) {
          return &it->second;
     }
     return nullptr; // Not found
}
```

Namespaces

```
namespace std {
    // ... many declarations
}
using std::string; // Using declaration for a
   specific type

struct NamespaceInfo {
    std::string name; // Name of the namespace
    FnTable functions;
    StTable structs;
    VaTable variables;
    AliasTable aliases;
    std::vector<NamespaceInfo> nestedNamespaces;
        // Nested namespaces within this
       namespace
};
```

Using namespaces

```
struct Globals {
    FnTable fns;
    StTable sts;
    AliasTable aliases; // Store aliases for
        both typedefs and using
    std::map<Id, NamespaceInfo> namespaces;
    std::vector<NamespaceInfo> usedNamespaces;
        // Local namespaces for the current scope
};

struct Context {
    Globals globals;
    std::vector<Scope> vars;
    AliasTable localAliases; // Local aliases
        for the current scope
    std::vector<NamespaceInfo> localNamespaces;
        // Local namespaces for the current scope
};
```

Using namespaces

```
void AddUsingNameSpaceLocal(const Id& id) {
    //Find namespace in the globals and add it to
        the current context or if symbols from that
         namespace were included, update the values
}

void AddUsingNameSpaceGlobal(const Id& id) {
    //Find namespace in the globals and add it to
        the global context or if symbols from that
        namespace were included, update the values
}
void AddSymbol(const qualId&, const std::vector<
   NamespaceInfo>& namespaces) {
    //Check if symbol exists in specific namespace
        , inside local or global used namespace
        list, create a dummy namespace with nested
        dummy namespaces but without any values and
         add the symbol to that namespace
}
```

Using namespaces

```
Type* findTypeInNamespace(const Id& id, const std::vector<NamespaceInfo
    >& namespaces) {
    for (const auto& ns : namespaces) {
        auto fnIt = ns.functions.find(id);
        if (fnIt != ns.functions.end()) {
            return fnIt->second.ret; // Return the function's return
                type
        }
        auto stIt = ns.structs.find(id);
        if (stIt != ns.structs.end()) {
            return stIt->second; // Return the struct's type
        }
        auto vaIt = ns.variables.find(id);
        if (vaIt != ns.variables.end()) {
            return vaIt->second; // Return the variable's type
        }
        auto aliasIt = ns.aliases.find(id);
        if (aliasIt != ns.aliases.end()) {
            return aliasIt->second.aliasedType; // Return the aliased
                type
        }
        for (const auto& nestedNs : ns.nestedNamespaces) {
            // Recursively search in nested namespaces
            auto type = findTypeInNamespace(id, {nestedNs});
            if (type) {
                return type; // Found in nested namespace
            }
        }
    }
    return nullptr;
}
```

Summary

- Both 'using' and 'typedef' create aliases, which are stored as their name and actual type
- Namespaces can be used fully, or as dummies with only neccessary symbols
- After exiting scope, local aliases and namespaces are cleared
- Visitor functions for qualified identifiers can check a specific namespace defined in globals directly
- Helper functions like findTypeNamespace() can be used to search for non-qualified identifiers in used namespaces

Our Experiences