# 名字查找（Name Lookup）

名字查找是*重载决议*的第一阶段，但本次不提及重载决议。

名字查找分为Qualified Name Lookup 有限定名字查找和Unqualified Name Lookup无限定名字查找，其中无限定名字查找又包括著名的实参依赖查找(ADL)

## 1 有限定名字查找

### 1.1 Class Member Lookup

类成员查找，实在访问类成员时进行名字查找的规则,成员分为变量和函数，同时也可以分为静态成员和动态成员，其中静态成员可以通过：：解析访问，而动态成员则可以通过是否为指针用'.'或'->'访问

示例：

```
#include <iostream>

struct A {};//在全局定义了一个结构体A
struct B {
    struct A {};//在B里面定义了一个结构体A，注意此时的A与上面的A不等价
    constexpr static int len = 50;
    static A arr[len];
};
//静态变量类外定义
A B::arr[len];//数组是A类型，arr是B作用域下的成员，len也会被当作成员在B作用域查找
//但此时会报错，因为最左边的A是全局里面的A，而arr的类型是B作用域下的A。

//修改为B::A B::arr[len]; 编译通过
```

### 1.2 Namespace Member Lookup

当存在：：限定符名字作用域下进行查找，否则在该成员所处作用域下查找

示例如下：

```
#include <iostream>

void foo() {
    std::cout << "this is ::foo()\n";
}

namespace test {
    void foo() {
        std::cout << "this is test::foo()\n";
    }

    void h() {
        ::foo();//加上了::限定符，在全局作用查找foo 输出 this is ::foo()
        foo();//在当前作用域下查找foo 输出 this is test::foo()
    }
}

int main() {
```

```
    test::h();
}
```

更复杂一点的例子：

```
int x;
namespace Y {
    void f(float);
    void h(int);
}

namespace Z {
    void h(double);
}

namespace A {
    using namespace Y;
    void f(int);
    void g(int);
    int i;
}

namespace B {
    using namespace Z;
    void f(char);
    int i;
}

namespace AB {
    using namespace A;
    using namespace B;
    void g();
}

void h() {
    AB::g();        // #1
    AB::f(1);       // #2
    AB::f('c');     // #3
    AB::x++;        // #4
    AB::i++;        // #5
    AB::h(16.8);    // #6
}
```

#1

Name Lookup 发现需要在AB命名空间下查找，到AB下后成功找到g编译通过。

#2

Name Lookup 同样在AB命名空间下进行查找，但是发现AB下并没有f这个名字，但同时他发现在AB中存在using namespace A和B，此时会在A和B中分别找到A::f(int)和B::f(char)，此时重载决议发生，发现A::f(int）是最佳匹配，所以选择调用A::f(int).

#3

流程与#2完全一致，但是在重载决议发生的时候会发现B::f(char)是最佳匹配，所以选择调用B::f(char).

#4

Name Lookup 首先在AB下寻找x，没有找到，于是在被using了的A和B下分别查找，依旧没找到A,B下又分别using了Y,Z最后还是没找到，所以程序在#4处会发生报错，虽然在全局定义了x,但是在有限定名字查找中，编译器不会跳出限定的作用域。

#5

Name Lookup 在AB下查找失败后，然后去A,B下分别查找，但是A,B下都定义了一个i，此时产生ambiguous（歧义）错误。

#6

与#2和#3类似，Name Lookup 最终会找到Y::h(int)和Z::h（double）,然后重载决议发生，Z::h（double）为最优匹配。

## 2 无限定名字查找

### 2.1Usual Unqualified Lookup

普通的函数都会进行常规的无限定名字查找

示例如下

```
void foo(const char *);

namespace A {
    void foo(int);

    void test() {
        foo(1);//无作用域限定，会在当前所处的作用域查找foo名字，匹配到了foo(int) OK
        foo("abc");//在当前查找到foo(int)但是参数不匹配，编译错误。
        //注意匹配过程只会在作用域查找名字而不会去匹配参数，也就是说匹配到foo(int)后就
        //停止匹配，并不会跳出A到全局查找
    }
}
```

```
#include <iostream>

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

namespace A {

    namespace B {

        namespace C {
            void test() {
                f(1);//输出 ::f() 如果在当前作用域没有找到名字的话就跳出当前作用域到上一
级，直到全局。
            }
```

```
        }
    }
}

int main() {
    A::B::C::test();
}
```

### 2.2 ADL实参依赖查找

实参依赖查找是通过参数的所在作用域来确定函数的作用域。比如有如下代码

```
#include <iostream>

struct A {
    int a;
    friend std::ostream &operator<<(std::ostream& os, const A &rhs) {
        return os << rhs.a;
    }
};

int main() {
    A a{1};
    std::cout << a;//此时该处等价调用 operator<<(std::cout, a),但是在全局中并没有operator这个函数的声明，但是根据ADL会在A下进行查找，最终找到声明。
}
```

```
#include <iostream>

struct B {};

namespace A {
    struct B {};
    void foo(B) {
        std::cout << "foo()\n";
    }
}

int main() {
    ::B x;
    foo(x);//全局未找到foo 匹配失败
    A::B y;
    foo(y);//输出 foo() ，虽然全局没有找到foo，但是可以根据参数所在作用域使得在A下查找foo
}
```

更多详细规则，查阅文档