ADL（Argument-Dependent Lookup，Koenig Lookup）

- 下面例子解释了名称查找的基本规则

```
namespace A {
struct X;
struct Y;
void f(int);
void g(X);
}

namespace B {
void f(int i)
{
  f(i); // 调用自身，即B::f()，造成无限递归
}
void g(A::X x)
{
  g(x); // 通过x的类型A::X看到A，通过A看到A::g()，因为B::g()也可见，此处产生二义性调用错误
}
void h(A::Y x)
{
  h(x); // 通过x的类型A::Y看到A，但A中无A::h()，此处调用自身，即B::h()，造成无限递归
}
}
```

- 受限名称的名称查找在受限的作用域内进行

```
int x;

class B {
 public:
  int i;
};

class D : public B {};

void f(D* p)
{
  p->i = 3; // 找到B::i
  D::x = 2; // 错误：在受限作用域中找不到::x
}
```

- 非受限名称则是普通查找方式，先查找该类和基类的作用域再查找外围类的作用域

```
extern int x; // (1)
int f(int x) // (2)
{
  if (x < 0)
  {
    int x = 1; // (3)
    f(x); // 使用(3)
  }
  return x + ::x; // 分别使用(2)、(1)
}
```

- 对于一个类，其成员函数与使用了它的非成员函数，都是该类的逻辑组成部分

```
class A {};
void f(const A&) {}; // f()是A的逻辑组成部分
```

- ADL也就是依赖于实参的查找，如果传给函数一个class类型实参，为了查找这个函数名，编译器不仅要查找局部作用域，还要查找包含实参类型的命名空间

```
namespace N {
class A {};
void f(A);
}

N::A x;

int main()
{
  f(x); // 通过ADL找到N::f()，f()是N::X的逻辑组成部分
  // 如果没有ADL，就要写成N::f(x)
}
```

- 一个更明显的例子

```
std::string s = "hi";
std::cout << s; // std::operator<<()是std::string的逻辑组成部分
// 如果没有ADL，就必须写成std::operator<<(std::cout, s)
```

- 非受限名称除了普通查找，还可以使用ADL

```
template<typename T>
inline const T& max(const T& a, const T& b)
{
  return a < b ? b : a;
}

namespace N {
class X {
  ...
};
bool operator<(const X&, const X&);
...
}

using N::X; // ADL会忽略using声明

void g(const X& a, const X& b)
{
  ...
  X x = ::max(a, b); // 通过ADL找到X的operator<
  ...
}
```

- ADL会忽略using声明

```
#include <iostream>

namespace X {
template<typename T>
void f(T);
}

namespace N {
using namespace X; // 忽略using声明，不会调用X::f
enum E { e1 };
void f(E)
{
  std::cout << "N::f(N::E) called\n";
}
}

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

int main()
{
  ::f(N::e1); // 受限的函数名称，不使用ADL
  f(N::e1); // 使用ADL找到N::f()
}
```

- ADL会查找实参关联的命名空间和类，关联的命名空间和类组成的集合定义如下
  - 内置类型：集合为空
  - 指针和数组类型：所引用类型关联的命名空间和类
  - 枚举类型：关联枚举声明所在的命名空间
  - 类成员：关联成员所在的类
  - 函数类型：所有参数和返回类型关联的命名空间和类
  - 类成员指针类型：成员和类关联的命名空间和类

友元声明的ADL

- 标准规定友元声明在外围作用域不可见。如果可见的话，实例化类模板会使普通函数的声明可见，如果没有先实例化类就调用函数，将导致编译错误。但如果友元函数所在类属于ADL的关联类集合，则在外围类可以找到该友元声明

```
template<typename T>
class C {
  friend void f();
  friend void f(const C<T>&);
};

void g(C<int>* p)
{
  f(); // f()不可见
  f(*p); // f(const C<int>&)可见
}
```

- f()没有关联类或命名空间，因为没有参数而不能利用ADL，所以是一个无效调用
- `f(*p)` 具有关联类 `C<int>`，因此只要调用前实例化了类 `C<int>` 就能找到该友元，为了确保这点，如果涉及关联类中友元查找的调用，没被实例化的类会被实例化

注入类名（Injected Class Name）

- 为了便于查找，在类定义中出现自身的类名称是一个自身的类型别名的public成员，该名称称为注入类名

```
int X;
struct X {
  void f()
  {
    X* p; // OK：X是注入类名
    ::X* q; // 错误：查找到变量名X，它隐藏了struct X的名称
  }
};
```

- 类模板也可以有注入类名

```
template<template<typename> class TT>
class A {};

template<typename T>
class X {
  X* a; // OK：X被当作类型名，等价于X<T>* a
  X<void> b; // OK
  using c = A<X>; // OK：X被当作模板名
  A<::X> d; // OK：::X不是注入类名，所以总会被当作模板名
};
```

- 旧标准中，注入类名不会被看作模板名称，必须加上作用域限定符以不使用注入类名

```
template<template<typename> class TT>
class A {};

template<typename T>
class X {
  X* a; // OK：X被当作类型名，等价于X<T>* a
  X<void> b; // OK
  A<X> c; // 错误：X后没有实参列表，不被看作模板
  A<::X> d; // 错误：<:是[的另一种标记
  A< ::X> e; // OK：<和::之间必须要有空格，因为<:会被看作[
};
```

- 如果注入类名的类是可变参数模板，则注入类名将包含未扩展的模板参数包

```
template<int I, typename... T>
class X {
  X* a; // OK：等价于X<I, T...>* a
  X<0, void> b; // OK
};
```

当前实例化和未知特化

- 类模板（及其成员函数和嵌套类）的注入类名为当前实例化（current instantiation），依赖于一个模板参数但不是当前实例化的被称为一个未知特化（unknown specialization）

```
template<typename T>
class Node {
  using Type = T;
  Node* next; // Node是当前实例化
  Node<Type>* previous; // Node<Type>是当前实例化
  Node<T*>* parent; // Node<T*>是未知特化
};
```

- 在嵌套类和类模板存在的情况下，识别一个类型是否为当前实例化容易产生混淆

```
template<typename T>
class C {
  using Type = T;

  struct I {
    C* c; // C是当前实例化
    C<Type>* c2; // C<Type>是当前实例化
    I* i; // I是当前实例化
  };

  struct J {
    C* c; // C是当前实例化
    C<Type>* c2; // C<Type>是当前实例化
    I* i; // I是未知特化
    J* j; // J是当前实例化
  };
};
```

- 上例中 `C<int>::J` 的定义将用于实例化具体类型， `C<int>` 将由嵌套类的定义实例化，因此 `C<int>::J` 和 `C<int>` 实例化的J是相同的。但如果 `C<int>::I` 有显式特化，就会为 `C<T>::J` 提供一个完全不同的定义

```
template<>
struct C<int>::I {
  // definition of the specialization
};
```

- 当前实例化和未知特化允许在定义点，而非在实例化点检测某些错误

```
template<class T>
class A {
  using type = int;
  void f()
  {
    A<T>::type i; // OK：ype是当前实例化的成员
    typename A<T>::X j; // 错误：X不是当前实例化和未知特化的成员
  }
};
```

非模板中的上下文相关性

- 解析理论主要面向上下文无关语言，而C++是上下文相关语言，为了解决这个问题，编译器使用一张符号表结合扫描器和解析器
- 解析某个声明时会把它添加到表中，扫描器找到一个标识符时，会在符号表中查找，如果发现该符号是一个类型就会注释这个标记，如编译器看见

```
x*
```

- 扫描器会查找x，如果发现x是一个类型，解析器会看到

```
identifier, type, x
symbol, *
```

- 并认为这里进行了一个声明，如果x不是类型，则解析器从扫描器获得标记为

```
identifier, nontype, x
symbol, *
```

- 于是这个构造就被解析为乘积
- 下面的表达式是一个上下文相关的例子

```
X<1>(0)
```

- 如果X是类模板名称，则表达式是把0转换成 `X<1>` 类型。如果不是模板的话，表达式等价于

```
(X<1)>0
```

- 先让X和1比大小，然后把结果转换成1或0，再与0比较大小。因此解析器先查找 `<` 前的名称，是模板名称时才会把 `<` 看作左尖括号，其他情况则看作小于号

```
template<bool B>
class X {
 public:
  static const bool result = !B;
};

void f()
{
  bool test = X<(1>0)>::result; // 必须使用小括号
}
```

- 如果省略小括号，第一个 `>` 会被错误地看作模板实参列表的结束标记
- C++11之前，由于maximal munch，两个大于号会被组合成右移标记 `>>`，尖括号时遇到作用域运算符 `<::` 会被看作 `[:`

```
List<List<int>> a; // C++11前的右尖括号要用空格隔开：List<List<int> >

class X {};
List<::X> many_X; // C++11前<和::要用空格隔开：List< ::X>
```

依赖型类型名称与typename前缀

- 模板名称的问题主要是不能有效确定名称，模板中不能引用其他模板的名称，因为其他模板的内容可能由于显式特化使原来的名称失效

```
template<typename T>
class Trap {
 public:
  enum{x}; // (1) x is not a type here
};

template<typename T>
class Victim {
 public:
  int y;
  void poof()
  {
    Trap<T>::x * y; // (2) declaration or multiplication?
  }
};

template<>
class Trap<void> {
 public:
  using x = int; // (3) x is a type here
};

void boom(Trap<void>& bomb)
{
  bomb.poof();
}
```

- 编译器解析(2)时，要确定看到的是声明还是乘积，取决于依赖型受限名称 `Trap<T>::x` 是否为类型名称，编译器此时查找模板Trap，根据(1)，`Trap<T>::x` 不是类型，因此(2)被看作乘积。而特化中x变成了类型，完全违背了前面的代码，这种情况下类Victim中的 `Trap<T>::x` 实际上是一个int类型
- 为了解决这个冲突，依赖型受限名称默认不会被看作类型，如果要表明是类型则需要加上typename前缀

```
typename Trap<T>::x * y; // 这是声明，如果不加typename前缀则是乘法
```

- 只有出现在模板中的受限名称，且名称不位于指定基类的列表和成员初始化列表中，才能使用typename前缀。在此前提下，如果名称依赖于模板参数，则必须指定typename前缀

```
template<typename T>
struct S : typename X<T>::Base { // 错误：不能用于指定基类的列表
  S(): typename X<T>::Base // 错误：不能用于初始化列表
    (typename X<T>::Base(0)) {} // OK：必须使用typename前缀

  typename X<T> f() // 错误：X<T>不是受限名称
  {
    typename X<T>::C * p; // 必须加上typename表示声明指针
    X<T>::D * q; // 否则会被视为乘法
  }

  typename X<int>::C * s; // 名称不依赖于模板参数，typename前缀可有可无
};

struct U {
  typename X<int>::C * pc; // 错误：受限名称，但不是在模板中
};
```

依赖型类型名称与.template

- 编译器会把模板名称后的 `<` 看作模板参数列表的开始，要在名称前加template关键字，才能让编译器知道引用的依赖型名称是一个模板。下面的例子说明了如何在运算符（ `::`、`->`和 `.`）后使用template关键字

```
template<typename T>
class A {
 public:
  template<int N>
  class B {
   public:
    template<int M>
    class C {
     public:
      virtual void f();
    };
  };
};

template<typename T, int N>
class X {
 public:
  void case1(A<T>::template B<N>::template C<N>* p)
  {
    p->template C<N>::f(); // 不调用虚函数
  }
  void case2(A<T>::template B<T>::template C<T>& p)
  {
    p.template C<N>::f(); // 不调用虚函数
  }
};
```

- 如果运算符前的名称或表达式类型依赖于模板参数，紧接在运算符后的是一个template-id，就应该使用template关键字，否则编译器不会将C视为模板

```
p.template C<N>::f(); // p依赖于模板参数N
```

依赖型类型名称与using声明

- using声明从依赖型类中引入名称时，并不知道该名称是类型名称、模板名称还是其他名称。如果希望using声明引入的依赖型名称是一个类型，必须用typename关键字显式指定。如果希望引入的是模板，必须用别名模板来指定

```
template<typename T>
class B {
 public:
  using X = T;
  template<typename U>
  struct Y;
};

template<typename T>
class D : private B<T> {
 public:
  using typename B<T>::X;
  X* p; // 如果不是类型名称则产生语法错误

  template<typename U>
  using Y = typename B<T>::template Y<T>;
  // 直接using B<T>::template Y是错误的
  Y<T>* p;
};
```

ADL和显式模板实参

```
namespace N {
class X {};
template<int I> void f(X*);
}

void g(N::X* p)
{
  f<3>(p); // 错误：不会通过ADL找到函数模板f
}
```

- 编译器必须先知道 `<3>` 是模板实参列表才能知道p是函数实参，先知道f是模板才能知道 `<3>` 是模板实参列表，这就陷入了死循环

非依赖型基类

- 类模板中，非依赖型基类指不用知道模板实参就可以推断类型的基类。派生类中查找一个非受限名称会先查找非依赖型基类，然后才会查找模板参数列表，比如下面派生类中的T x，T来自基类而非自身的模板参数

```
template<typename X>
class B {
 public:
  int n;
  using T = int;
};

template<typename T>
class D : public B<double> { // 非依赖型基类
 public:
  void f() { n = 7; }
  T x; // T是B<double>::T，而非模板参数
};

void g(D<int*>& d, int* p)
{
  d.x = p; // 错误：x类型总会是Base<double>::T，即int
}
```

依赖型基类

- 对于模板中的非依赖型名称，编译器会在看到的第一时间查找

```
template<typename X>
class B {
 public:
  int n;
  using T = int;
};

template<typename T>
class D : public B<T> { // dependent B
 public:
  void f() { n = 0; } // (1) problem...
};

template<> // explicit specialization
class B<bool> {
 public:
  enum { n = 42 }; // (2) tricky!
};

void g(D<bool>& d)
{
  d.f(); // (3) oops?
}
```

- (1)发现非依赖型名称n，在模板B中找到，把n绑定到int变量，然而随后的特化改写了B的定义，改变了n的含义（但此时n已经确定绑定为一个int变量），在(3)实例化 `D::f` 定义时就会产生错误
- 为了解决这个问题，标准规定非依赖型名称不会在依赖型基类中查找，因此编译器会在(1)给出诊断信息，为了纠正可以让n也成为依赖型名称（实例化时才会查找），可以使用 `this->` 或作用域运算符来指定

```
template<typename T>
class D1 : public B<T> {
 public:
  void f() { this->n = 0; } // lookup delayed
};

template<typename T>
class D2 : public B<T> {
 public:
  void f() { B<T>::n = 0; }
};
```

- 或者使用using声明，这样只需要引入一次而不用每次都指定

```
template<typename T>
class D3 : public B<T> {
 public:
  using B<T>::n;
  void f() { n = 0; }
};
```

- 注意使用作用域运算符会禁止虚函数，如果原来使用了虚函数调用会改变程序含义，对于这种情况只能使用 `this->`

```
template<typename T>
class B {
 public:
  enum E { e1 = 6, e2 = 28, e3 = 496 };
  virtual void f(E e = e1);
  virtual void g(E&);
};

template<typename T>
class D : public B<T> {
 public:
  void h()
  {
    typename D<T>::E e; // this->E是一个无效的语法
    this->f(); // D<T>::f()会禁止虚函数调用
    g(e); // g是依赖型名称，因为实参e类型（D<T>::E）是依赖型
  }
};
```