Глава 3
Управление контекстом компиляции и структура семантических таблиц

Понятие **области действия** (scope) является одним из фундаментальных понятий языка Си++; правила, описывающие это понятие, определяют важнейшие проектные решения при конструировании семантических таблиц компилятора. В настоящей главе рассматривается эволюция данного понятия от языка Си к языку Си++ и обсуждаются основные составляющие области действия, как они определены в Стандарте. На основе этого рассмотрения предлагается и обосновывается модель контекста компиляции, положенная в основу реализации компилятора переднего плана Си++.

Перед тем, как описать принципы построения таблиц компиляции, рассмотрим правила видимости имен в языках Си и Си++. Язык Си как объект анализа выбран по двум причинам: во-первых, он практически полностью сохраняет принципы структурного устройства, характерные для предшествующего поколения языков программирования, во-вторых, он послужил непосредственной основой для языка Си++ (вплоть до совместимости снизу вверх). В то же время, правила видимости имен в Си++ претерпели существенные изменения по сравнению с Си. Совместное существование в одном языке двух во многом противоречивых множеств правил обусловливает сложности выбора проектных решений по организации таблиц трансляции.

3.1. Правила видимости имен в языке Си

Видимость имен в Си основана на традиционной блочной структуре, характерной, в частности, для таких языков, как Algol 60, PL/I, Pascal. Имя считается определенным, начиная с точки, где впервые встретилось объявление программной сущности с данным именем. Данное имя считается известным вплоть до конца того блока, где это объявление имело место, и никакое другое имя с тем же идентификатором в этом блоке объявлено быть не может.

Если язык допускает вложенные блоки, то имя, объявленное в таком блоке, может совпадать с именем из объемлющего блока; в этом случае новое имя скрывает имя из объемлющего блока в пределах своей области видимости.

Эта простая и естественная структура дополняется в Си наличием независимых (параллельных) пространств имен, которые "накладываются" на блочную структуру. Так, в Си имеется пространство имен для тегов ("ярлыков") структур, объединений и перечислимых типов, пространство имен для каждой структуры или объединения, и пространство для всех остальных имен, исключая метки.

Соответственным образом уточняется правило видимости имен: идентификатор может быть объявлен в одном блоке неоднократно, если

он не повторяется в одном и том же пространстве имен. Указанное дополнение на самом деле не нарушает общую концепцию, характерную для блочной структуры, так как имена из разных пространств имен с одними теми же идентификаторами распознаются по занимаемой ими синтаксической позиции [14]. Например, конструкцию

```
struct S
```

можно рассматривать как единый идентификатор (отличный от идентификатора `S`) из пространства имен тегов структур; при этом использование идентификатора `S` (без служебного слова **`struct`**) для именования структуры `S` не допускается ни в каком контексте. Аналогично, выражение вида

```
instanceS.member          или
pointerS->member
```

обозначает объект с идентификатором `member` из текущей области действия; возможная неоднозначность снимается по типу объекта `instanceS` или базовому типу указателя `pointerS`. Здесь также невозможен доступ к члену структуры без использования префикса, однозначно специфицирующего структуру, которая содержит данный член.

Структуры в Си не образуют собственного пространства имен; так, задание "вложенных" структур вида

```
struct S  {
     struct InnerS { ... };
     enum InnerE { ... };
};
```

не локализует имя вложенной структуры и имя перечислимого типа в пределах структуры `S`: использование `InnerS` и `InnerE` допустимо вне структуры `S`, например:

```
struct InnerS  is;
enum InnerE  ie;
```

Принципиальным следствием описанного устройства пространств имен языка Си является возможность эффективной реализации таблиц трансляции в виде единой блочно-структурированной таблицы с минимально необходимыми оптимизирующими средствами (включая обычный дисплей [34], единую хеш-таблицу и т.д.). Вся таблица будет, по существу, представлять стек областей действия, в совокупности образующих текущий контекст компиляции. Различение одинаковых имен из различных пространств может быть реализовано с помощью незначительной модификации поискового алгоритма, а для простых случаев может быть эффективно обеспечено даже на уровне лексического анализа.

3.2. Правила видимости в Си++

Особенности организации пространств имен в Си++ отражают двойственность, присущую данному языку: с одной стороны, Си++

должен обеспечить совместимость с Си, а с другой стороны, обилие новых понятий и возможностей этого языка не может не привносить большого количества корректив в простую и естественную модель видимости языка-предшественника.

В основе правил видимости Си++ лежит традиционная блочно-структурированная видимость имен языка Си; в последующих подразделах описываются наиболее существенные модификации традиционной схемы.

3.2.1. Единое пространство имен

Все имена (исключая метки) считаются находящимися в одном пространстве имен [45, разд. 3.1c]. Тем самым поддерживается принцип полной унификации типов, определенных пользователем, и встроенных типов; иными словами, определив некоторый класс вида

```
class complex {
      // ...
};
```

программист может оперировать с новым именем типа `complex` в точности так же, как и с любым встроенным типом - **`int`**, **`float`** и т.д. Так, объявление объекта приведенного класса может быть задано следующим образом:

```
complex c1;
```

В случае сохранения раздельных пространств имен, характерного для Си, каждое объявление объекта типа `complex` необходимо было бы предварять служебным словом **`class`**:

```
class complex c1;
```

Это, далее, пришлось бы делать при любом упоминании имени `complex`: в операциях преобразования типов, спецификации параметров и результата функции, конструировании производных типов и т.д., что, как пишут авторы языка, "обесценило бы усилия по приближению употребления определенных пользователем типов, подобных `complex`, к работе со встроенными типами, вроде **`int`**" [45, разд. 3.1c].

Однако, необходимость обеспечения совместимости с языком Си вынудила разработчиков Си++ пойти на компромисс, согласно которому Си++ должен поддерживать все принятые в Си правила употребления отдельных пространств имен для структур (**`struct`**) и объединений (**`union`**).

(Заметим, что относительно перечислимых типов (**`enum`**) Си++ обеспечивает максимально возможную для сильно типизированных языков строгость: каждый перечислимый тип считается отдельным целочисленным типом, и имя этого типа может употребляться наравне в любыми другими типами без префикса **`enum`**; единственной уступкой совместимости является возможность все-таки задавать этот префикс перед именем типа).

В результате Си++ допускает одновременное наличие в одной и той же области действия имени типа и имени "не типа", обозначенных одинаковыми идентификаторами. В частности, характерное для Си описание вида

```
struct S { } S;
```

которое одновременно объявляет тип `S` и объект `S`, корректно и для Си++ и допустимо также для случая классов, объединений и перечислимых типов. Если в некоторой области действия имеются одноименные тип и "не тип", то, как и в Си, для различения этих сущностей необходимо использовать соответствующее служебное слово типа - **`class`**/**`struct`**/**`union`**/**`enum`** (то есть, в терминах Си++, задавать *уточненный-спецификатор-типа*).

Подобная двойственность правил для типов, определенных пользователем, вносит существенные сложности в проектирование таблиц компиляции и организацию поисковых алгоритмов. В качестве иллюстрации рассмотрим следующий пример:

```
struct S {  }; // описание класса S
S s;           // объявление переменной типа S
               // (стиль Си++)
int S ( int ); // объявление функции S

     // Теперь в текущей области действия присутствуют
     // две сущности с одним и тем же именем S

int x = S(1);  // вызов функции S в качестве
               // инициализатора
struct S s1;   // описание переменной типа S (стиль Си);
               // использование S в качестве имени типа
               // теперь возможно только вместе со
               // спецификатором struct:
S s2;          // ошибка
```

Для разбора этого примера компилятор должен содержать усовершенствованный поисковый алгоритм для распознавания вызова функции `S` в инициализаторе переменной `x`. Альтернативой могла бы явиться усложненная структура таблиц, поддерживающая раздельные пространства имен для типов и "не типов".

Дополнительные трудности обусловлены большим числом исключений из общих правил видимости. Иллюстрацией может служить конструкция *уточненного-спецификатора-типа*. Правила Си++ [Std, 3.3.1, §4] говорят, что если имя класса впервые вводится *уточненным-спецификатором-типа* и если он имеет форму

class-key `identifier ;`

то есть имя вводится обычным предварительным объявлением класса, то это имя считается *именем-класса* в области действия, содержащей данное объявление, то есть в текущей области действия. С другой стороны, если *уточненный-спецификатор-типа* имеет форму:

```
class-key identifier ...
```

то идентификатор, впервые введенный таким образом, объявляется как *имя-класса* в пределах минимальной неклассовой, нефункциональной области действия, содержащей данное объявление. Иными словами, если имеется описание класса, содержащего функцию-член:

```
class X  {
     void f ( class C* );
};
```

то имя `C` считается *именем-класса* в области действия, содержащей описание класса `X`; поэтому после вышеприведенного описания допустимо, например,

```
C* pc;  // C известно в данном контексте как имя-класса
```

Иными словами, вышеприведенный фрагмент считается эквивалентным следующему:

```
class C;  // предварительное объявление класса C
class X  {
     void f(C*);
};
C* pc;
```

С точки зрения разработчика компилятора приведенное правило, будучи примененным к последнему примеру, означает, что имя нового класса `C` должно добавляться не в текущий контекст (функции-члена `f`) и не в контекст класса `X`, а в контекст, объемлющий класс `X`. При этом для определения подходящего для `C` контекста необходим последовательный анализ **видов** объемлющих областей действия.

3.2.2. Область действия объявления класса

Объявление класса (в том числе, структуры или объединения) определяет собственную область действия, в которой существуют члены класса.

Таким образом, семантика доступа к членам классов отличается от принятой в языке Си (см. разд. 3.1). В Си++ члены класса помещаются в отдельную область видимости, а конструкция вида

```
instanceC.member          или
pointerC->member
```

означает доступ к объекту `member` из области видимости, которая вычисляется, исходя из типа объекта `instanceC` или базового типа указателя `pointerC`.

Указанная семантическая разница между трактовкой доступа к членам структур в Си и доступом к членам классов в Си++, необязательно влечет разную реализацию табличной операции выбора члена структуры, так как в простых случаях обе трактовки дают одинаковый эффект. Однако наличие в Си++ понятия наследования, в особенности множественного наследования принципиально меняет

картину, существенно усложняя семантику доступа к членам классов. Так, для нахождения члена, заданного в правой части операции "`.`" или "`->`" в общем случае необходимо просмотреть весь граф наследования того класса, который определяется левым операндом; при этом необходимо контролировать возможные неоднозначности, связанные с объявлениями одноименных членов в "соседних" по дереву наследования базовых классах. Наличие виртуальных базовых классов усложняет общую схему, требуя однократного (бесповторного) учета имен членов таких базовых классов.

Другой пример нерегулярности правил видимости Си++ связан с объединениями (**`unions`**). С одной стороны, объединения считаются частным случаем классов и в качестве таковых образуют собственную область действия. С другой стороны, неименованное объединение определяет не тип, а объект [Std, 9.5, §2] и потому не формирует новую область действия. Считается, что члены неименованного объединения содержатся в области действия, содержащей данное объединение, и используются в ней непосредственно, без обычных конструкций доступа к членам класса.

Так как неименованное объединение вводит сущности, формально описанные как члены объединения, в объемлющий контекст, то их имена должны отличаться от других имен из области действия, где это объединение объявлено. Тем самым, обработка неименованных объединений предполагает модификацию объемлющей области действия.

3.2.3. Лексическая вложенность и отношения областей действия

Лексическая (текстуальная) вложенность блоков не обязательно совпадает с семантическими отношениями блоков. Это накладывает дополнительные требования на организацию таблиц трансляции. Рассмотрим характерный пример, связанный с функциями-членами класса.

По определению [Std, 3.3.6, §1, п.1] тело функции-члена класса находится в области действия того класса, членом которого она является, так, как будто блок, содержащий объявления членов класса, окружает тело функции (а формальные параметры функции введены в самый внешний блок тела функции). Данное правило действует независимо от того, где задано тело функции,- в пределах описания класса или вне его. Так, следующие два примера совершенно идентичны с точки зрения семантики языка Си++ (с точностью до неявного спецификатора **`inline`** у функции-члена в первом варианте, что для нашего рассмотрения несущественно).

```
// а) Функция-член
//    внутри тела класса

int i;
class C {
     int i;
     void f()
     {
          i = 5;
     }
};
```

```
// б) Функция-член
//    вне тела класса

int i;
class C {
     int i;
     void f();
};
void C::f ()
{
     i = 5;
}
```

В обоих случаях член `i` класса `C` скрывает глобальное `i`, которое, тем самым, становится невидимым для функции-члена `f()`. Поэтому присваивание в теле функции в обоих случаях устанавливает новое значение члена `i`, а не глобального `i`. Однако если в первом примере традиционные правила видимости "срабатывают", как обычно (семантически зависимые блоки вложены друг в друга и текстуально), то необходимый эффект обработки тела функции во втором примере может быть достигнут специальным формированием контекста функции `f()` на период ее трансляции. Иными словами, перед началом компиляции функции-члена необходимо добавлять в текущий контекст фрагмент таблицы, соответствующий области действия класса `C` (а если `C` - вложенный класс, то и таблицы для областей действия всех объемлющих классов), а после завершения компиляции - удалять из текущего контекста эти фрагменты.

Аналогично следует поступать и в случае описания статического члена класса вне тела класса, так как инициализатор статического члена находится в области действия класса. Так, в следующем примере

```
int a;
class C  {
     static int a;
     static int b;
};
int C::a = 1;
int C::b = a;  // здесь a есть C::a, а не ::a
```

использование имени `a` в инициализаторе статического члена `C::b` означает доступ к члену класса с данным именем, а не к одноименной глобальной переменной.

(Заметим, что исходя из приведенных примеров, операцию разрешения области действия `::` компилятор мог бы интерпретировать как команду построения контекста из областей действия, указанных перед последним знаком `::`. Однако, семантика этой операции в выражениях (то есть, в использующих вхождениях имен вида `M::N::...::K::x`) не предполагает перестройки контекста, а просто задает область поиска сущности с именем `x`.)

Проблема несовпадения текстуальной вложенности и семантическими отношениями блоков, описанная выше, в общем виде может быть сформулирована так: объявление программной сущности, имя которой задается посредством *составного-спецификатора-имени*, рассматривается в контексте, определяемым ее *составным-спецификатором-имени*, независимо от текущего контекста.

Подчеркнем, что сформулированное правило распространяется на области действия классов и пространств имен, то есть, тех сущностей, из имен которых и формируется *составной-спецификатор-имени* [Std, 5.1, §7]. Следует иметь в виду также, что речь идет об объявлении в целом, включая инициализаторы объектов или тела функций (функций-членов), а также объявления параметров, включающих значения по умолчанию.

Характерно, что локальные и вложенные классы (см. разд. 3.2.5) и дружественные функции (см. разд. 3.2.6) дают противоположные примеры несовпадения лексической вложенности и семантических отношений областей действия: лексическая вложенность (например, локального класса в некоторую функцию или описания дружественной функции в класс), вообще говоря, не означает семантической подчиненности соответствующих блоков.

3.2.4. Механизм наследования

Правила, аналогичные изложенным в предыдущем подпункте, действуют и для производных классов. Производный класс, вместе со своими функциями-членами рассматривается как находящийся в области действия своего базового класса; тем самым, производный класс сохраняет возможность непосредственного доступа к членам базового класса. При этом, разумеется, описание производного класса не является текстуально вложенным в описание базового класса. В случае совпадения имен в базовом и производном классе действуют обычные правила блочности: член производного класса скрывает одноименный член базового класса в своей области действия.

(Заметим, что свойство совместной используемости функций не оказывает влияния на блочные правила, так как в качестве совместно используемых рассматриваются только функции из одной и той же области действия.)

Однако, ситуация существенно усложняется для множественного наследования. В данном случае для отображения контекстных зависимостей областей действия недостаточно простой стековой структуры таблиц трансляции, так как производный класс должен транслироваться в контексте всех своих базовых классов (а те, в свою очередь,- в контексте своих собственных непосредственных базовых классов, и т.д.). При этом необходимо убедиться в однозначности использования имени в контексте производного класса. Следующий простой пример показывает возникающие проблемы.

```
struct Base1  {
     int i;
};

struct Base2  {
     int i;
};

struct Derived : Base1, Base2  {
     int f(void);
}

int Derived::f(void)
{
     return i; // неоднозначность: Base1::i или Base2::i?
}
```

Для разрешения неоднозначности компилятор по определению не может опираться на порядок следования базовых классов в заголовке, так как он никак не влияет на видимость имен. Таким образом, в общем случае контекст класса включает контексты всех своих непосредственных и косвенных базовых классов, причем для корректного разрешения имен в общем случае необходимо просмотреть весь направленный ациклический граф наследования [Std, Глава 10, §3] данного класса.

3.2.5. Особенности вложенных и локальных классов

Хотя правила видимости Си++ и основаны на блочной структуре, они, как уже говорилось, существенно зависят от вида области действия. Так, для вложенных или локальных классов, а также для статических или дружественных функций, описанных в теле класса, видимость имен из области действия объемлющего класса (или, соответственно, из содержащей функции) ограничена именами статических переменных, функциями, именами типов и перечислителей из объемлющей области действия, например:

```
int x;
void f()
{
     int x;
     static int y;
     typedef int I;
     class Local {  // локальный класс
          int g()
          {
               I i;       // OK: int i;
               i = y;     // OK: y - статическая
               i = ::x;   // OK: глобальная x
               return x; // ошибка: x - нестатическая
          }
     };
}
```

Обработка ситуаций, аналогичных приведенному примеру, вызывает сложность, существо которой состоит в том, что на реализацию поискового алгоритма накладываются ограничения, не связанные с правилами видимости как таковыми, а определяемые соотношениями "вид области действия/вид искомой сущности". Заметим, что тот факт, что в примере имеется глобальная переменная `x`, не влияет на алгоритм поиска: локальная `x`, оставаясь недоступной для функции `Local::g`, тем не менее скрывает одноименную глобальную переменную в своей области действия.

Таким образом, для отношения областей действия функции и ее локального класса характерна двойственность: для некоторых сущностей действуют традиционные правила видимости имен (во вложенном блоке видимы имена из объемлющего), другие сущности из объемлющего блока (например, локальные переменные объемлющей функции) недоступны в локальном классе. То же правило действует и для дружественных функций (см. разд. 3.2.6).

3.2.6. Дружественные функции

Понятие дружественных функций, введенное для "обхода" правил доступа к приватным членам класса, также нуждается в специальных правилах видимости. Считается, что если дружественная функция целиком описана в пределах класса, то она находится в лексической области действия этого класса; таким образом, для этой функции непосредственно доступно определенное множество программмных сущностей из дружественного класса (статические члены, функции, типы и элементы перечислений). Этот случай полностью подпадает под действие правил из предыдущего пункта и не вызывает дополнительных сложностей.

С другой стороны, когда в качестве дружественной функции задается функция-член некоторого другого класса, ситуация не столь однозначна. Если в теле класса в качестве дружественной полностью описана функция-член некоторого другого класса, то тело этой

дружественной функции находится сразу в двух, в общем случае никак не связанных между собой областях действия: "своего" класса и того класса, для которого функция описана как дружественная.

Следующий пример иллюстрирует сказанное.

```
class C  {
     int i;
     typedef int I;
     int f(I);            // C::I
};

class B  {
     static int z;
     int i;
     typedef int I;
     friend int C::f(I)   // B::I или C::I?
     {
          I x;            // C::I или B::I?
          x = z;          // возможно ли обращение к z?
          return i;       // C::i или B::i?
     }
};
```

Данный случай правил видимости в Си++, видимо, является одним из наиболее сложных для компиляции с точки зрения организации таблиц трансляции. Существо подхода, определенного в Стандарте к разрешению "старшинства" (правил предпочтения) двух указанных областей действия, можно сформулировать следующим образом: заголовок дружественной функции-члена транслируется в контексте класса, в котором функция указана как дружественная, а ее тело - в контексте "собственного" класса. Так, из пар альтернатив, указанных в комментариях в вышеприведенном примере, выбираются предпочтения, написанные первыми. При этом область действия класса `B` не включается в контекст тела функции `C::f`: член `B::z` невидим в пределах тела этой функции. Заметим, что компиляторы Borland C++ и GNU C++ устанавливают приоритеты именно таким образом.

Из сказанного следует, что организация таблиц должна обладать необходимой гибкостью, чтобы в достаточно широких пределах производить перестроение текущего контекста согласно правилам видимости, приведенным выше.

3.2.7. Пространства имен

Помимо перечисленных аспектов, необходимо кратко рассмотреть также механизм явного управления пространствами имен (namespaces), семантика которого также предполагает отступления от исходной модели видимости имен.

Существо данного механизма заключается во введении особых именованных областей действия имен и в возможности явного управления этими областями действия. Так, имея некоторое

пространство имен N, содержащее объявления сущностей с именами a, b, c, можно задать явное включение нескольких (или всех) указанных сущностей в некоторую другую область действия, в которой известно (видимо) имя N. Включенные имена получают в точности те же права видимости и доступа, что и имена, непосредственно объявленные в области действия.

Сказанное проиллюстрируем примером:

```
int a;

namespace N  {
    int a;
    typedef int b;
    void c ( int );
}
    . . .

void f ()
{
    // В данном блоке сущности из N доступны
    // посредством квалифицированных имен
    int I1 = a;      // используется глобальная ::a;
                     // a из N недоступна по простому имени
    int I2 = N::a;   // используется a из N
    . . .
    using namespace N;
    // Теперь в данном блоке имена из N доступны
    // по простым (неквалифицированным именам

    c(1);         // вызов N::c()
    b x1;         // неоднозначность: b или N::b?
    N::b x2;      // неоднозначность снята использова-
                  // нием составного-спецификатора-имени
}
```

Следует добавить, что сущности, объявленные в пространстве имен, могут быть описаны вне этого пространства имен. В этом случае для них справедливы соображения, высказанные в разд. 3.2.3 относительно перестроения контекста трансляции.

Резюмируя сказанное, отметим, что механизм пространств имен приводит к необходимости дополнительных решений при проектировании семантических таблиц. Основная проблема, требующая решения, может быть сформулирована следующим образом.

В общем случае любая область действия (любой блок программы на Си++) характеризуется, помимо прочего, списком сущностей, объявленных вне этой области действия, но доступных в нем (посредством простых имен) согласно правилам включения пространств имен (*объявления-использования* и *директивы-использования*). Необходимо предложить достаточно эффективный способ реализации таких контекстных связей.

Отметим также уникальное свойство пространств имен, которое заключается в возможности их "разрывного" описания: в одной и той же области действия может существовать более одного описания пространств имен с одним и тем же именем; в этом случае такие одноименные описания "складываются", образуя единое пространство имен. Правда, с точки зрения организации таблиц это свойство не добавляет существенных затруднений и практически полностью вписывается в совокупность требований, вытекающих из предыдущих рассмотрений. К тому же для любой точки программы видимыми являются только те части пространства имен, которые расположены выше по тексту программы, например:

```
namespace N  {
    int a1;
}

int f ( )
{
    using namespace N;
    int b = a1;      // OK: N::a1
    int c = a2;      // ошибка: a2 не описана в N
}

namespace N  {
    int a2;
}
```

Обстоятельство, проиллюстрированное этим примером, дает возможность организовать трансляцию раздельных описаний одного пространства имен, не усложняя обычную однопроходную схему. Однако, "разрывность" пространств имен существенно повлияет на выбор реализации семантических таблиц (см. разд. 3.5 далее).

3.3. Проблемы и требования

Перед тем, как перейти к описанию принципов и проектных решений, предлагаемых для таблиц компиляции, суммируем основные **проблемы**, которые возникают в связи с отображением правил видимости языка Си++ на структуры данных компилятора.

1. Самое общее свойство семантических таблиц (таблиц трансляции) заключается в том, что таблицы должны содержать исчерпывающую информацию как обо всех сущностях, явно или неявно объявленных в компилируемой единице трансляции, так и о контекстных отношениях областей действия всех видов [Std, 3.3], допустимых в единице трансляции.

2. Таблицы трансляции должны адекватно и эффективно поддерживать все виды контекстных отношений между областями действия Си++.

   Контекстные отношения областей действия, как следует из предыдущих рассмотрений, можно разделить на две категории.

   Первая категория, характерная для "Си-части" языка, включает блочную иерархию с традиционными правилами видимости, которая при компиляции представляется стековой организацией таблиц.

   Вторая категория содержит более сложные и во многом специфичные для языка Си++ контекстные отношения, к числу которых относятся описания функций-членов классов вне тел классов, схемы наследования классов (в том числе, множественное наследование и виртуальные базовые классы), "разрывные" пространства имен, механизм дружественных функций (в том числе, функций-членов класса, дружественных некоторому другому классу) и т.п.

   Общим свойством указанных особенностей является явное задание контекста посредством *квалифицированных имен* - идентификаторов программных сущностей, квалифицированных именами содержащих областей действия (классов и/или пространств имен). Для реализации подобных отношений таблицы трансляции должны обеспечивать динамическое перестроение иерархии (порядка) областей видимости для корректной обработки областей, контекстная принадлежность которых не совпадает с их текстуальной позицией. При этом необходимо уметь восстанавливать исходный порядок областей видимости после завершения обработки "специфических" областей действия. Существенно, что алгоритм такого перестроения не подчиняется обычным стековым правилам, характерным для простой блочной структуры.

3. Необходимо обеспечить как единое пространство имен для всех программных сущностей (Си++), так и отдельные пространства имен для классов и перечислимых типов (Си).

   Проблема может решаться как усовершенствованным поисковым алгоритмом, учитывающим вид искомых сущностей, так и усложненной структурой таблиц, обеспечивающей раздельное хранение сущностей.

4. Необходимо обеспечить полный набор средств (доступ к элементам, модификацию, добавление и удаление элементов) не только для фрагмента таблиц, соответствующего текущей области видимости (что, как правило, бывает достаточно в случае традиционной блочной структуры), но и для фрагментов для областей действия, объемлющих текущую. Более того, в общем случае необходим полный доступ к произвольному фрагменту таблиц, соответствующему такой области действия, которая потенциально может оказаться в текущем контексте.

   Обе проблемы могут быть решены за счет раздельного представления фрагментов таблиц, соответствующих различным областям действия, с построением системы контекстных связей между фрагментами. Подробнее об этом см. ниже.

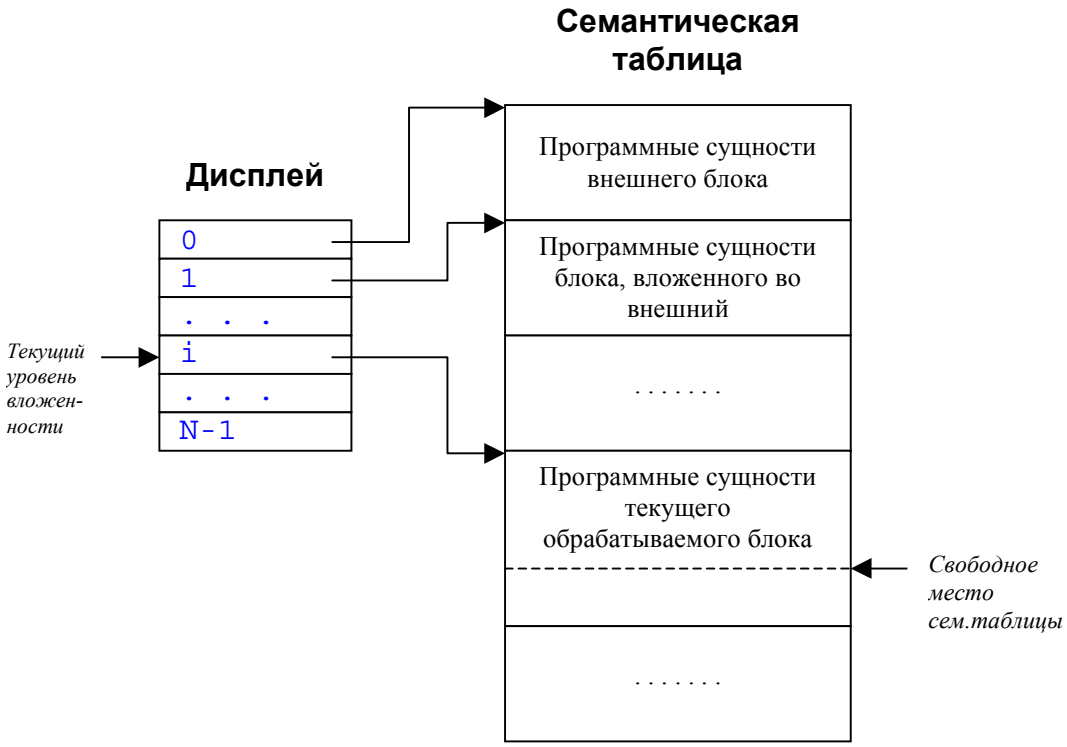
5. В пределах каждой отдельной области видимости необходимо обеспечить различение одноименных, но семантически отличных программных сущностей, допустимых правилами языка (см. разд. XX). Кроме того, следует обеспечить корректную обработку допустимых случаев повторных объявлений одной и той же сущности.

3.4 Традиционная организация семантических таблиц

Традиционная модель структуры семантических таблиц подробно описывается, например, в [34]; простой пример реализации таблиц для компилятора паскалеподобного языка имеется в [43]. В работе [42] описывается построение таблицы для языков класса Modula-2. В общем случае модель включает собственно **таблицу**, концептуально представляющую собой единый непрерывный неограниченный массив, элементы которого содержат семантические атрибуты отдельных программных сущностей, и **дисплей** - короткий массив фиксированного размера, содержащий информацию о текущем контексте компиляции. Размер дисплея фиксирует максимальную статическую вложенность контекстов в данной реализации.

Фрагменты таблицы, соответствующие областям видимости, компиляция которых началась, но еще не закончилась (т.е. вложенных друг в друга), организованы по стековому принципу. На практике это означает, что таблица заполняется последовательно, по мере поступления новых программных сущностей из исходного текста. При завершении обработки очередного блока программы достаточно просто переместить указатель свободной части таблицы в позицию начала соответствующего фрагмента и уменьшить на 1 значение текущего уровня вложенности.

Для поиска имен в такой единой таблице необходимо только иметь адреса (индексы) начал областей видимости. Для хранения адресов вложенных областей видимости используется дисплей. Схематически это можно представить следующим рисунком:



Для быстрого поиска имен по таблице используются различные техники; наиболее распространенной из них является хеширование имен [34], [35] со связыванием имен с одинаковыми хеш-значениями в цепочки. Удачный выбор направления такого связывания (от имен из текущего блока в именам в объемлющих блоках) может обеспечить дополнительное ускорение поиска, правда, за счет возрастания накладных расходов на операцию выхода из блока.

Такая организация таблиц исторически возникла для простых семантических отношений между областями действия, характерных для языков класса Algol-60, Pascal и Си; по существу, единственное существенное отношение заключается в доминировании сущностей текущего блока над одноименными сущностями объемлющих блоков. В предыдущих разделах были подробно рассмотрены особенности структурной семантики языка Си++, из которых следует недостаточность подхода, связанного с единой семантической таблицей.

К сказанному можно добавить несколько других существенных соображений, затрудняющих использование традиционной модели семантических таблиц.

Принципиальная особенность описанной схемы заключается в том, что в каждый момент в таблице содержатся те и только те программные сущности, которые образуют текущий контекст трансляции. Сама таблица работает при этом как обобщенный стек: при входе в некоторый блок в "вершине" стека формируется область для нового блока, в которую далее помещаются элементы, соответствующие программным сущностям этого блока, а при выходе из блока информация о его программных сущностях удаляется из таблицы. С одной стороны, это позволяет организовать весьма эффективный доступ к таблицам (в

частности, поиск имен) и обеспечивает компактность информационных структур компилятора. С другой стороны, фундаментальный недостаток такой организации состоит в потере семантической информации об исходной программе.

Последнее обстоятельство во многих практических случаях не является существенным. Так, если компилятор реализует однопроходную схему трансляции, то он может, например, формировать объектный код поблочно (такой подход использован в компиляторе Эль-76 [95]). Тогда при завершении обработки очередного блока (подпрограммы) вся необходимая информация, в частности, секция объектного кода и, быть может, соответствующая отладочная информация, оказываются полностью сформированными, и в дальнейшем сохранении фрагмента таблицы для этого блока необходимости нет.

Хотя в целом Си++ допускает однопроходную схему трансляции, некоторые его особенности не позволяют использовать подобный подход. Так, семантика шаблонов функций и пространств имен (namespaces) предопределяет необходимость сохранения информации об их локальных контекстах после завершения обработки их тел. В некоторой степени это относится к классам и встраиваемым (inline) функциям.

Однако, современные подходы к задачам трансляции ЯП и цели работы, сформулированные во Введении, определяют самые существенные причины, по которым потеря семантической информации в процессе трансляции является недопустимой.

Сказанное выше в принципе не означает безусловной невозможности адаптации традиционной схемы к новым задачам. Так, специфические для Си++ контекстные отношения можно было бы реализовать и на основе традиционной модели семантических таблиц. Это относится к таким особенностям языка, как локальные классы и "разрывные" описания пространств имен. Однако в целом подход к реализации семантических таблиц, основанный на единой таблице, реализованной на стековом принципе, может быть применен для компиляции языка Си++ только за счет введения дополнительных контекстных связей и семантических атрибутов, а это влечет существенное усложнение реализации и снижение эффективности базовых алгоритмов доступа к таблице.

Таким образом, при выборе схемы реализации семантических таблиц возникает принципиальная дилемма: либо пытаться использовать традиционную модель, отразив дополнительную контекстную семантику Си++ введением, например, соответствующих атрибутов элементов таблицы и операций над ними, либо предложить иной подход к реализации, максимально учитывающий особенности Си++ и требования, сформулированные во Введении.

В последующих разделах описывается модель организации семантических таблиц, основанная на децентрализованном подходе.

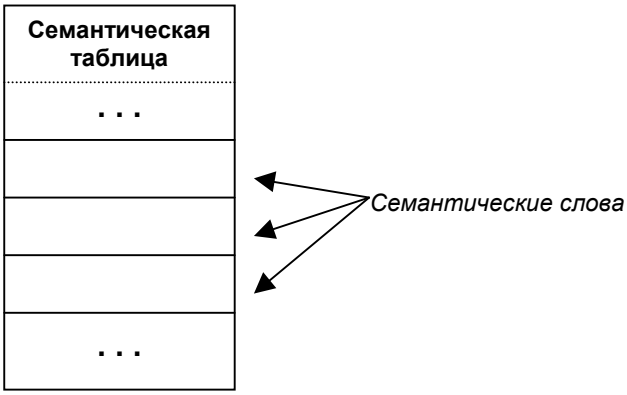
3.5 Модель организации семантических таблиц для Си++

Рассмотрения, проведенные в предыдущих разделах данной главы, приводят к следующей организации семантических таблиц в компиляторе Си++.

1. Каждая область действия отображается в виде отдельной структуры, называемой далее **семантической таблицей**. Каждая семантическая таблица создается динамически компилятором в момент начала обработки соответствующей области действия в тексте единицы компиляции. Каждая семантическая таблица существует *независимо* от подобных таблиц для других областей действия.

2. Основное содержание семантической таблицы образует информация обо всех (именованных и/или неименованных) сущностях, явно или неявно объявленных в соответствующей области действия. Элемент семантической таблицы, относящийся к одной сущности (называемый далее **семантическим словом**), можно представить как множество атрибутов, в совокупности составляющих необходимое для целей компилятора знание о данной сущности (включая информацию о месте ее объявления в единице трансляции).

   Два вышеприведенных тезиса можно наглядно представить в виде следующей схемы:



3. Считается, что между семантическим словом, содержащимся в некоторой семантической таблице, и самой этой таблицей установлено отношение принадлежности: семантическое слово ***O*** **принадлежит** семантической таблице ***S***, если и только если сущность, соответствующая семантическому слову, объявлена в области действия, которой соответствует семантическая таблица.

   Чтобы не перегружать графическое представление модели, отношение принадлежности будем изображать непосредственным вхождением семантического слова в семантическую таблицу. Например, если в некоторой области действия объявлен объект, то соответствующая этой области действия семантическая таблица ***S*** и соответствующее объекту семантическое слово ***O*** можно изобразить так:

4. Две семантические таблицы могут находиться в определенных **семантических отношениях** друг с другом. Эти отношения прямо соответствуют правилам Си++ для областей действия. Имеется три вида отношений: *вложенность*, *наследование* и *использование*.

5. Существо отношения вложенности и его интерпретацию в модели стоит рассмотреть несколько подробнее. Вложенность областей действия характерна для большинства языков программирования и сама по себе не нуждается в каких-либо дополнительных комментариях. Отметим только, что в Си++, как говорилось выше, вложенность областей действия не обязательно представляется как текстуальная вложенность: например, область действия функции-члена класса вложена в область действия своего класса, хотя текстуально может располагаться вне класса.

   Однако некоторым категориям программных сущностей свойствен определенный дуализм, который характерен практически для всех современных языков программирования. Этот дуализм наиболее ярко проявляется для такого фундаментального понятия Си++, как класс. По определению [Std, Глава 9] класс в Си++ - это *тип*. В этом качестве классы могут использоваться для объявления объектов, в операциях преобразования и приведения типов и т.д. С другой стороны, описание любого класса в Си++ образует отдельную *область действия*, для которой действуют правила видимости. Такой же дуализм свойствен и другим сущностям языка - функциям, пространствам имен и составным операторам.

   Эта двойственность поддерживается в модели следующим образом: такие программные сущности, как функции, классы, пространства имен и составные операторы, представляются в модели *и семантическим словом, и собственной семантической таблицей*. А отношение вложенности семантических таблиц опосредуется семантическим словом "двойственной" сущности. Иными словами, такое семантическое слово помещается в семантическую таблицу, которая соответствует области действия, содержащей данную сущность, а между семантическим словом сущности и его семантической таблицей устанавливается отношение **владения**.

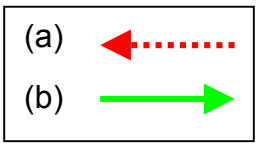
Таким образом, в модели имеется два, по существу симметричных, отношения: вложенность семантических таблиц и владение семантического слова семантической таблицей.

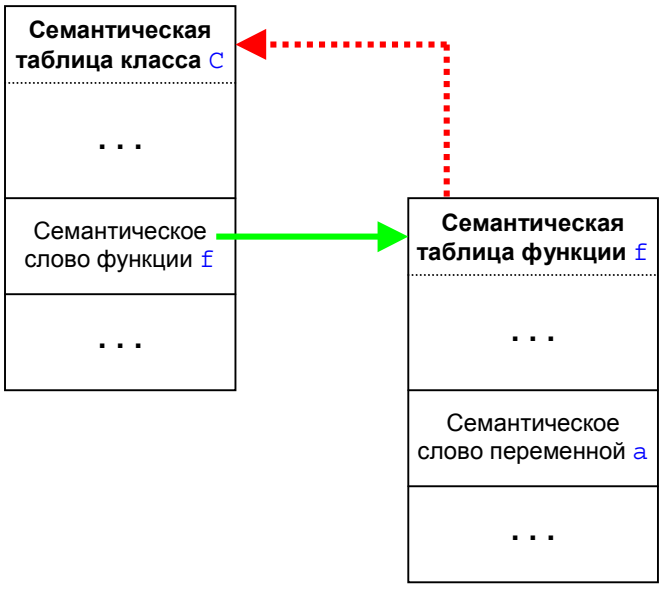
Сказанное можно проиллюстрировать следующим примером. Пусть имеется описание класса, содержащего функцию-член:

```
class C  {
        . . .
      int f ( void ) { int a; ... }
        . . .
};
```

Область действия функции `f` по определению вложена в область действия класса `C`. При этом в семантической таблице класса `C` содержится семантическое слово, соответствующее функции `f`. Далее, так как описание функции-члена `f` само образует область действия (в ней содержится, в частности локальный объект `a`), для него имеется собственная область действия; соответствующая `f`. Семантическое слово `f` **владеет** семантической таблицей `f`. Так как семантическое слово `f` содержится в семантической таблице `C`, то считается, что таблица `f` **вложена** в таблицу `C`.

Приведенный пример можно изобразить в виде следующей схемы. Здесь и в последующих рисунках пунктирная стрелка вида (a) отображает отношение вложенности, а сплошная стрелка вида (b) отображает отношение владения.





В заключение дадим определения введенных отношений.

Семантическая таблица ***S1*** **вложена** в таблицу ***S2***, если и только если соответствующая ***S1*** область действия в единице трансляции текстуально или логически вложена в область действия, соответствующую ***S2***.

Семантическое слово ***O*** **владеет** семантической таблицей ***S***, если и только если соответствующая таблице область действия вводится посредством объявления сущности, семантическое слово которой есть ***O***.

Для целей установки отношения владения считается, что составной оператор объявляется своим непосредственным вхождением; для него, так же, как и для других сущностей, образуется семантическое слово, а его имя генерируется компилятором как уникальное в пределах всей программы (в точности по правилам Си++ для неименованных пространств имен [Std, 7.3.1.1]).

6. Семантическая таблица ***S2*** **наследует** таблицу ***S1***, если и только если соответствующая ***S2*** область действия в единице трансляции относится к описанию производного класса, а область действия таблицы ***S1*** соответствует одному из его непосредственных базовых классов.

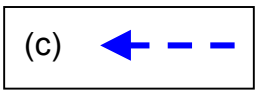
   Отношение наследования иллюстрируется следующим примером. Пусть имеется класс `D`, двумя непосредственными базовыми классами которого являются `B1` и `B2`:

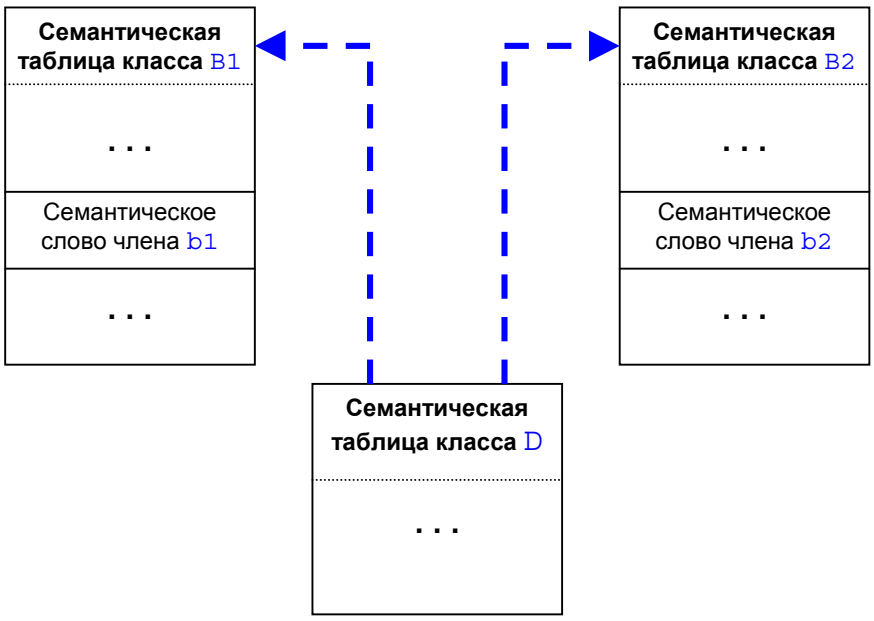
```
class B1 {
public:
        ...
      int b1;
        ...
};

class B2 {
public:
        ...
      int b2;
      ...
};

class D : public B1, private B2
{
      ...
};
```

   Соответствующий введенным сущностям фрагмент семантических таблиц представлен ниже. Пунктирная стрелка вида (c) отображает отношение наследования.





Заметим, что отношение наследования во многом аналогично отношению вложенности: по определению, сущность из базового класса видима в производном классе, а сущность, введенная в производном классе, скрывает одноименную сущность из базового класса. Однако отношение наследования в общем случае не может быть сведено к отношению вложенности, так как семантика наследования дополнительно включает *права доступа* имен (access rights, [Std, Глава 11]). Так, в приведенном примере публичный член `b2` из класса `B2`, будучи унаследованным классом `D`, считается в нем приватным (в силу приватной формы наследования `B2`) и, тем самым, недоступен через производный класс.

Кроме того, при множественном наследовании возможны неоднозначности в случае присутствия в базовых классах одноименных сущностей (так как в языке не определяются правила предпочтения базовых классов, а порядок их задания не принимается во внимание).

По этим причинам отношение наследования, кроме собственно связи между семантическим таблицами, включает и дополнительный атрибут - "характер" наследования; например, публичное или приватное наследование. Поисковые алгоритмы компилятора будут трактовать данное отношение аналогично отношению вложенности, с учетом этого дополнительного атрибута и возможных неоднозначностей.

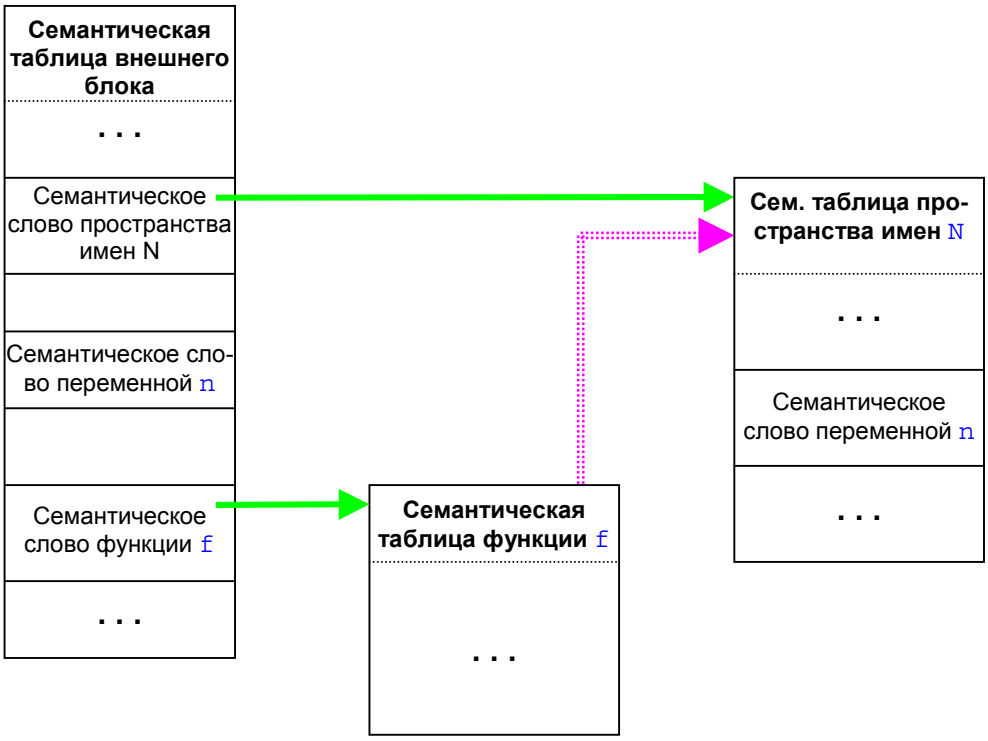
7. Семантическая таблица ***S1*** **использует** таблицу ***S2***, если и только если в соответствующей ***S1*** области действия имеется *директива-использования* (*using-directive*, [Std, 7.3.4]), в которой указано пространство имен, соответствующее таблице ***S2***.

   Рассмотрим следующий фрагмент программы:

```
namespace N  {
      int n;
}
  . . .
int n;
  . . .
int f ( void )
{
▶     n = 2;                  // глобальная ::n
      using namespace N;
▶▶    n = 1;                  // N::n
      return ::n+n;           // ::n + N::n
}
```

Отношения между областями действия в данном примере можно представить в виде схемы внизу (показаны только атрибуты владения и использования). Заметим, что глобальная область действия, согласно ее семантике, трактуется в модели как неименованное пространство имен (либо, для простоты, как составной оператор), охватывающее все единицы трансляции [3.3.5, §3].

Далее в примерах отношение использования будет обозначаться двойной пунктирной стрелкой:



Отношение использования, в отличие от отношений владения и наследования, распространяется не на всю использующую область действия, а на ее часть - начиная от точки вхождения *директивы-использования* и до конца области действия. Поэтому данное отношение (аналогично отношению наследования) включает дополнительный атрибут - семантическое слово использующей семантической таблицы, начиная с которого действует данное отношение. Заметим, что этот дополнительный атрибут несущественен для целей генерации объектного кода, так как использующие вхождения сущностей из пространства имен однозначно разрешаются компилятором; такой дополнительный атрибут вводится для сохранения полной информации об исходном тексте. С другой стороны, необходимость такого атрибута должна учитываться при выборе способа реализации семантических таблиц, чтобы его вычисление не приводило к снижению производительности. Чтобы быстро определить, действует ли отношение использования для некоторого семантического слова, необходимо, в терминологии Си++, чтобы структура, реализующая семантическую таблицу, содержала эффективную операцию "больше" над составляющими таблицу семантическими словами.

8. Чтобы завершить первую часть описания модели семантических таблиц, необходимо ввести ряд "операционных" терминов, непосредственно связанных с понятиями модели. Операции над понятиями в дальнейшем будут описываться в этих терминах. Базовым термином является понятие *собственного контекста*; остальные термины рекурсивно определяются из него.

   Собственный контекст:

   > Множество программных сущностей, объявленных в некоторой области действия, или, что то же самое, множество всех семантических слов из некоторой семантической таблицы. В большинстве случаев собственный контекст можно отождествлять с семантической таблицей как таковой.

   Контекст простого блока (составного оператора, функции, пространства имен):

   > Собственный контекст семантической таблицы, соответствующей данной области действия, и контексты всех семантических таблиц, непосредственно связанных с данной отношениями вложенности и/или использования.

   Контекст класса:

   > Собственный контекст класса, контекст объемлющей области действия и контексты семантических таблиц, непосредственно связанных с данной отношением наследования.

Контекст функции-члена

Собственный контекст функции-члена и контекст класса, членом которой является функция.

Контекст

Контекст простого блока, либо контекст класса, либо контекст функции-члена.

Текущий контекст

Контекст, соответствующий текущему состоянию разбора исходного текста. Текущий контекст включает известную в данный момент разбора часть собственного контекста обрабатываемой области действия и все контексты, связанные с ним отношениями вложенности и/или владения и/или использования.

3.6 Обсуждение, замечания и примеры

Прежде чем перейти к описанию второй составляющей модели семантических таблиц - набору операций над введенными выше понятиями - необходимо обсудить ряд существенных моментов, характерных для этой модели.

Независимость от реализаций. Представленная модель носит концептуальный характер и не навязывает какого-либо конкретного способа реализации. В частности, система семантических таблиц и семантических слов (как наборов атрибутов) с определенным множеством отношений между ними вполне адекватна понятийной базе реляционной алгебры, поэтому введенные понятия можно рассматривать как некоторую реляционную базу данных. Сказанное подтверждается успешным опытом реализации подобного подхода для языка Java компанией XDS (Новосибирск, www.xds.ru). В то же время данная модель допускает и более традиционную реализацию, в которой семантические таблицы логически имеют вид прямоугольных матриц (или, что тот же самое, массивов, каждый структурный элемент которых представляет одно семантическое слово), а семантические отношения представляются в виде указателей на таблицы.

Неполнота и открытость. Во-вторых, заметим, что описанная модель является неполной; она не включает ряд необходимых компонент, прежде всего, понятийную базу и логическую инфраструктуру поиска имен. С другой стороны, эту составляющую можно считать полностью зависящей от реализации и не привносящей в данную модель новых понятий. Например, в случае реализации модели в виде базы данных алгоритмы поиска полностью совпадают со стандартными алгоритмами, входящими в состав любой СУБД. С другой стороны, характерные для задач визуализации и обратной инженерии (reverse engineering) алгоритмы полного обхода некоторого подмножества семантических таблиц в своей основе тривиальны и не нуждаются в специальной спецификации.

Выбранные для компилятора Си++ методы реализации поиска имен полностью основываются на понятиях представленной модели и предусматривают некоторую логическую надстройку над ней в виде единой хеш-таблицы и дополнительных семантических атрибутов таблиц, связывающих их с семантическими словами, а также друг с другом в двоичные деревья. Подробное описание структур и алгоритмов поиска имен содержится в работе [59].

Представленная модель неполна еще в одном отношении: она не охватывает все возможности языка Си++. В первую очередь это касается шаблонов. Этот аспект модели имеет конкретные исторические причины и комментируется в главе 4. Там же содержится описание способа расширения модели для представления средств типовой параметризации Си++, причем показывается, что эти расширения, несмотря на существенные особенности механизма шаблонов, не вносят в модель концептуальных усложнений.

Таким образом, предложенная модель "выдерживает" как добавление отсутствующих в ней, но необходимых при реализации логических механизмов (например, поиск имен), так и допускает достаточно естественное расширение собственных возможностей при минимальном числе новых понятий. Сказанное подтверждается успешной реализацией данной модели в компиляторе переднего плана, поддерживающем полный набор правил поиска (look-up rules, [Std, 3.4]) и весь спектр возможностей механизма шаблонов [Std, Глава 14].

Система атрибутов. Семантические отношения между областями действия (вложенность, наследование, использование), а также между сущностями и областями действия отображаются в виде специальных атрибутов семантических таблиц. Таким образом, обе структурные составляющие модели - семантические таблицы и семантические слова - обладают собственными множествами атрибутов. Состав множества атрибутов специфичен как для различных видов семантических таблиц (блок, фунция, пространство имен и т.д.), так и для семантических слов, представляющих различные программные объекты (переменные, члены классов, typedef-имена и т.д.). Наряду с этим, для семантических таблиц и семантических слов имеется фиксированный набор *общих* атрибутов, характерных для всех сущностей данной категории. В число таких общих атрибутов входят, в частности:

- для семантических слов - вид программной сущности, атрибут принадлежности семантической таблице, имя сущности, представляемой данным словом, права доступа, признак используемости, а также ее тип. Кроме того, имеется ряд атрибутов реализационного характера, например, признак видимости сущности в текущем контексте.
- для семантических таблиц - вид области действия и атрибут принадлежности.

Симметрия отношений. Следует сделать несколько дополнительных комментариев относительно симметрии семантических отношений. Понятно, что отношения вложенности и владения являются в

определенном смысле симметричными (хотя и устроены несколько по-разному). Такая симметрия является, строго говоря, избыточной для традиционных задач компиляции и введена для поддержки требований, сформулированных во Введении. В самом деле, в случае традиционной компиляции представляет интерес только отношение

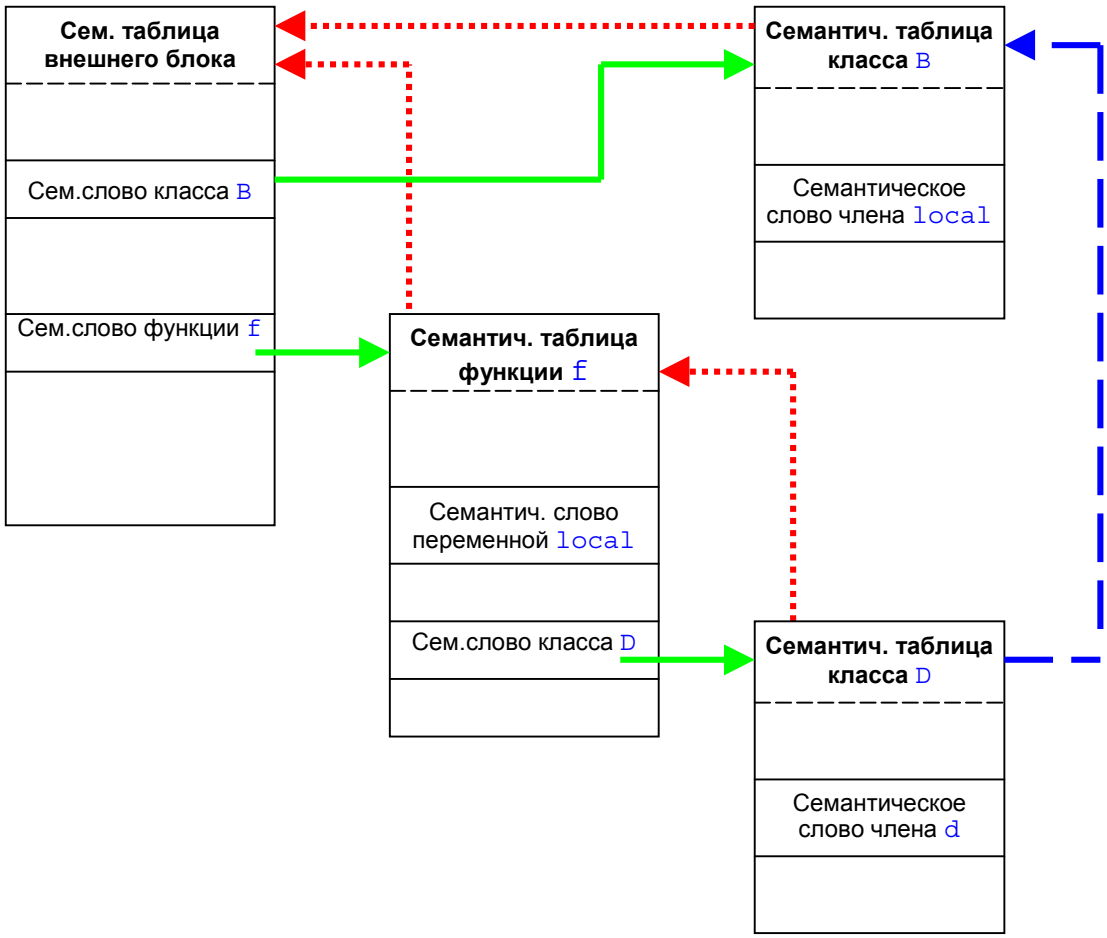
Возможна и аналогичная симметрия для отношений наследования и использования. Так, если некоторая область действия используется (посредством конструкции **using namespace**) в другой области действия, то, в принципе, может представлять интерес информация о том, в каких областях действия используется данная. Так же, как вложенность, отношения наследования и использования многозначны: один класс может выступать в качестве базового для нескольких призводных классов, а пространство имен может использоваться во многих областях действия. Однако поддержка в модели подобной многозначности для наследования и использования приводит к ее неоправданному усложнению, а ее реализация снижает общую эффективность системы. Поэтому симметрия здесь вводится в усеченном виде (например, информация о том, что данный класс используется как базовый, представляется в виде логического атрибута без указания конкретных производных классов). Такое решение обусловлено, помимо указанных причин, еще и отсутствием необходимости в такого рода знании для распространенных применений и, во-вторых, высокой сложностью поддержки такого рода отношений (в случае раздельной трансляции многомодульных программ эта поддержка оказывается невозможной).

Совмещение отношений. Семантика Си++ предопределяет возможность участия семантической таблицы одновременно в нескольких отношениях. Так, семантическая таблица локального класса находится в отношении вложенности с семантической таблицей содержащей функции или блока. Если же этот класс является производным от некоторого другого класса, то наряду с упомянутым отношением его семантическая таблица будет находиться в отношении наследования с таблицей базового класса.

Покажем совмещение отношений на примере.

```
class B  {
     int local;
};

void f ()
{
     int local;
     class D : public B  {
          int d;
     };
}
```



Приоритет между несколькими отношениями проявляется, в частности, при поиске имени в таблицах и определяется алгоритмически в зависимости от вида конкретной семантической таблицы и, в некоторых случаях, вида искомой сущности. Так, если в классе `D` содержится использующее вхождение имени `local` (на рисунке не показано), то по правилам языка сущность с этим именем из функции `f` считается недоступной из класса `D` (хотя и находится в его контексте); с другой стороны, в производном классе доступен член базового класса с таким же именем `local`.

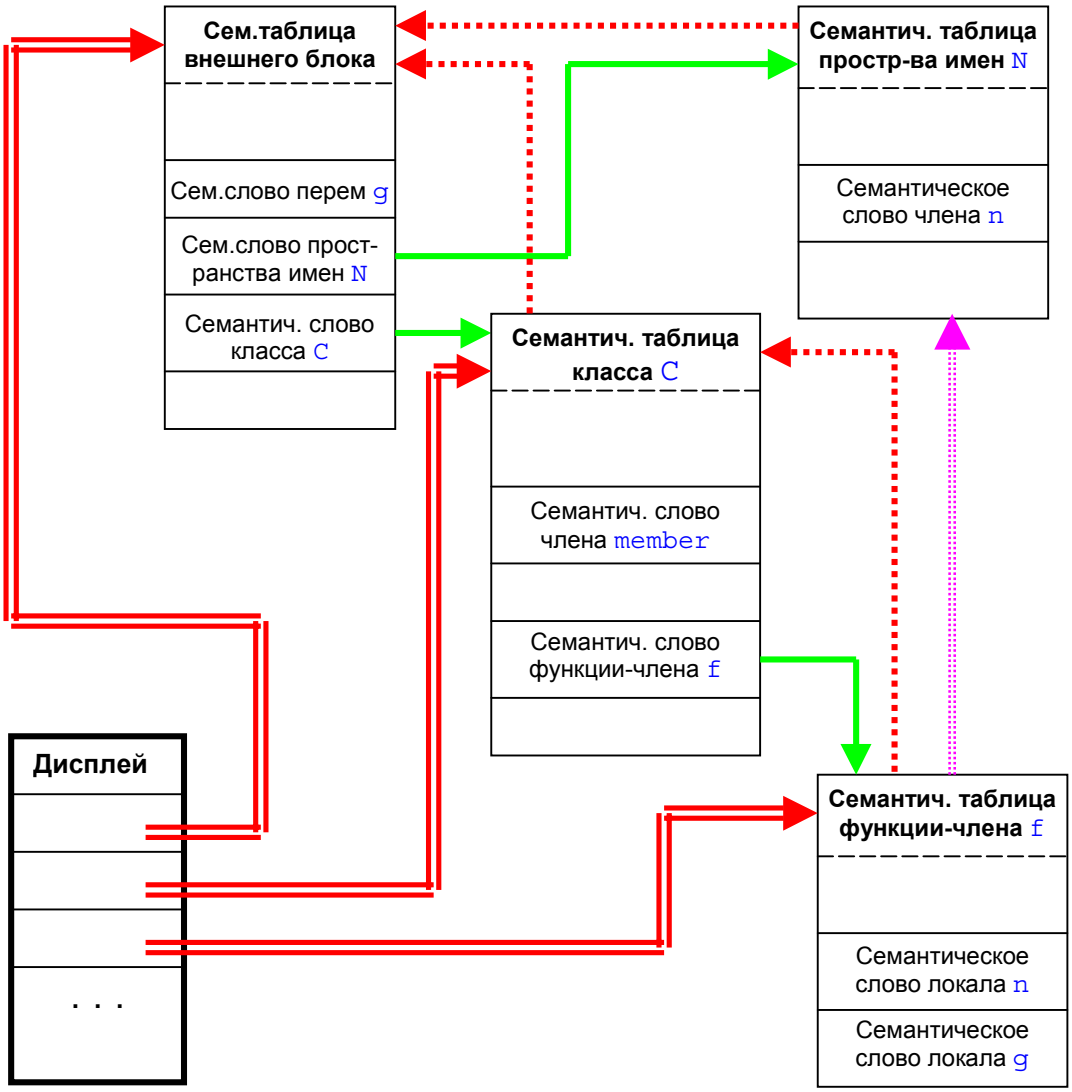
Необходимость учета видов сущностей и областей действия в алгоритмах над семантическими таблицами следует считать определенной слабостью модели; с другой стороны, этот недостаток непосредственно вытекает из нерегулярности правил Си++, которые опираются на отличия областей действия и формулируются по-разному применительно к сущностям различных видов.

Контекст и текущий контекст. Дисплей. Из данных в предыдущем разделе определений контекста и текущего контекста следует, что текущий контекст однозначно идентифицируется семантической таблицей, которая соответствует обрабатываемой в данный момент компилятором области действия. Однако реализация семантических таблиц включает дополнительное средство отображения текущего контекста, которое традиционно называется **дисплеем** [34].

Дисплей представляет собой структуру (массив), каждый элемент которого хранит информацию об одной семантической таблице из текущего контекста, причем порядок следования элементов соответствует отношению вложенности. Последний элемент дисплея содержит информацию о семантической таблице обрабатываемой в данный момент области действия. Состояние семантических таблиц и дисплея для следующего фрагмента:

```
int g;
namespace N { int n; }
class C  {
     int member;
     int f ()
     {
          int n, g;
          using namespace N;
          member = g;          // локальное g скрывает
                               // глобальное g
          n += member;         // неоднозначность
          return member;
     }
};
```

показано ниже:



Строго говоря, дисплей можно трактовать как альтернативную форму реализации отношений вложенности. Поэтому концептуально дисплей не является необходимым элементом модели, а выступает как ее реализационное усовершенствование. Тем не менее, в последующем изложении мы будем использовать это понятие, прежде всего, ввиду его высокой наглядности.

Дисплей является удобной основой для организации эффективного поиска имен. В частности, традиционные правила блочности и скрытия глобальных сущностей одноименными локальными естественно реализуются последовательным просмотром семантических таблиц, доступных из дисплея, начиная от последней (текущей) таблицы. Так, объявление имени `g` в функции `f` *скроет* глобальное имя `g`, так как семантическая таблица функции будет просмотрена первой.

Существенно, что отношение использования в дисплее не отображается. На рисунке выше семантическая таблица для пространства имен `N` не попадает в дисплей. Это отражает семантическую разницу отношений вложенности и использования. Так,

добавление в контекст функции `f` пространства имен `N` посредством директивы-использования `using namespace N` не означает скрытия сущностей из функции одноименными сущностями из пространства имен. Считается, что после этой директивы и до конца области действия функции в ней имеются *две* сущности с именем `n`, а любое использующее вхождение этого имени приводит к неоднозначности. Аналогичная же ситуация для функций будет означать их совместное использование (overloading).

Алгоритм поиска имен в некоторой отдельной области действия должен учитывать эти правила, распространяя сферу поиска неквалифицированного имени на все семантические таблицы, находящиеся в отношении использования с таблицей для данной области действия.

3.7 Модельные операции управления контекстом

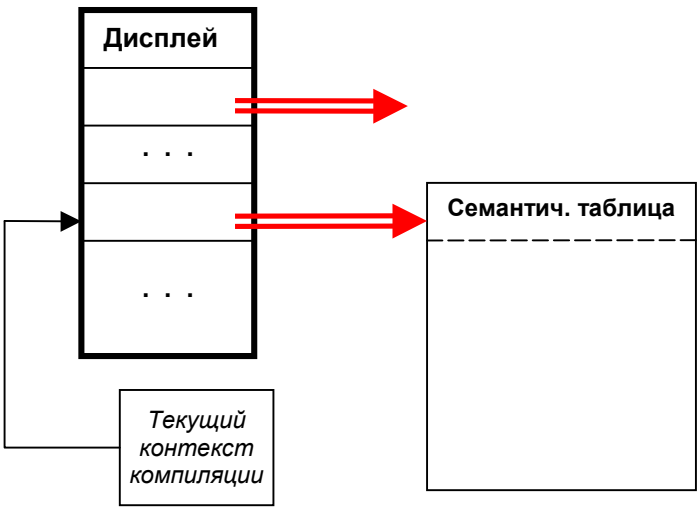
В данном разделе содержится общее описание важнейших **операций** над понятиями модели, описанной в предыдущих разделах. В совокупности эти операции образуют вторую часть модели; их можно считать обобщенными спецификациями компонент компилятора Си++, ответственных за управление контекстом трансляции.

NewScope*(Attributes...)*

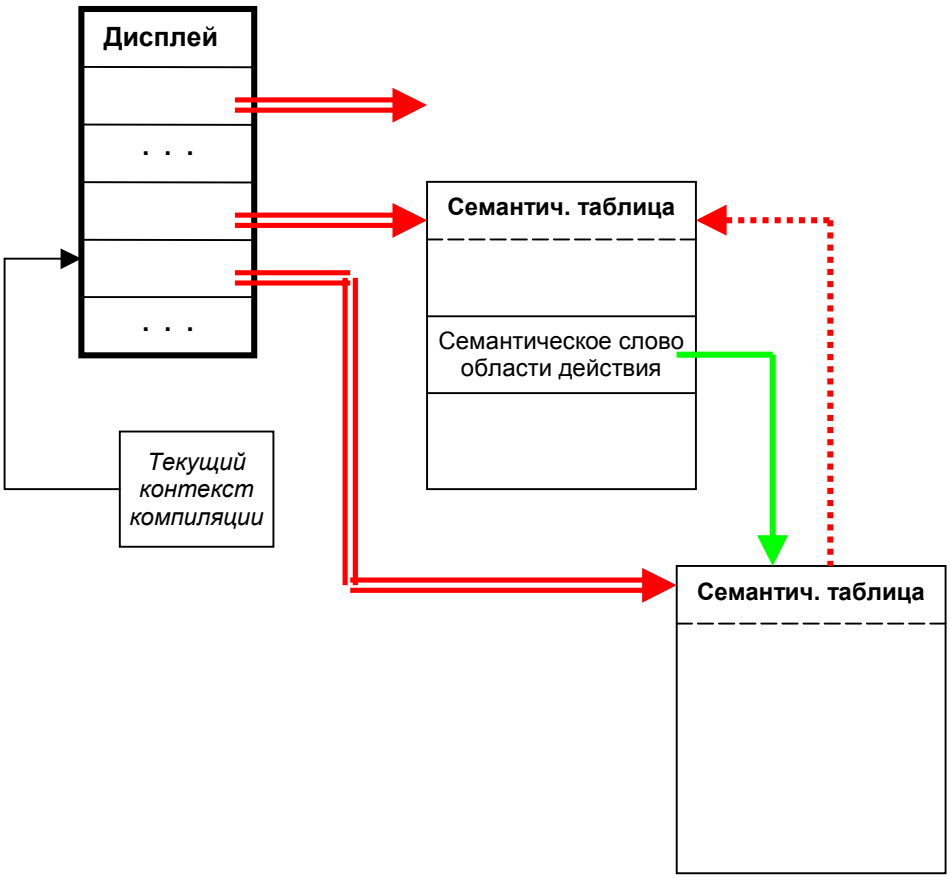
В системе таблиц возникает новая семантическая таблица (первоначально пустая). Между новой семантической таблицей и текущей таблицей устанавливается отношение вложенности. Более точно, в текущей семантической таблице образуется семантическое слово, представляющее новую область действия, и связывается с созданной семантической таблицей. Семантическое слово инициализируется набором атрибутов, переданных операции. Новая таблица помещается в текущий контекст (то есть добавляется в дисплей) и становится текущей.

На рисунках ниже схематически показано состояние таблиц до и после выполнения операции ***NewScope***:

а) До операции ***NewScope***:



б) После операции ***NewScope***:



Данная операция соответствует началу обработки простых блоков - составных операторов (в том числе, `try`- и `catch`-блоков), тел функций и пространств имен. Параметрами операции является набор атрибутов семантического слова для новой таблицы; одним из основных атрибутов служит вид программной сущности (составной оператор, функция, пространство имен и т.д.).

AddScope*(Entity)*

Данная операция носит более общий характер по сравнению с предыдущей. Она отличается от операции ***NewScope***, во-первых, тем, что не создает новую семантическую таблицу, а работает с уже существующей. Операция ***AddScope*** включает в текущий контекст семантическую таблицу, идентифицируемую семантическим словом, передаваемым ей в качестве параметра, действуя аналогично операции ***NewScope***. Второе отличие данной операции заключается в том, что между текущей областью действия и областью, добавляемой в текущий контекст, допускается как отношение вложенности, так и отношение наследования.

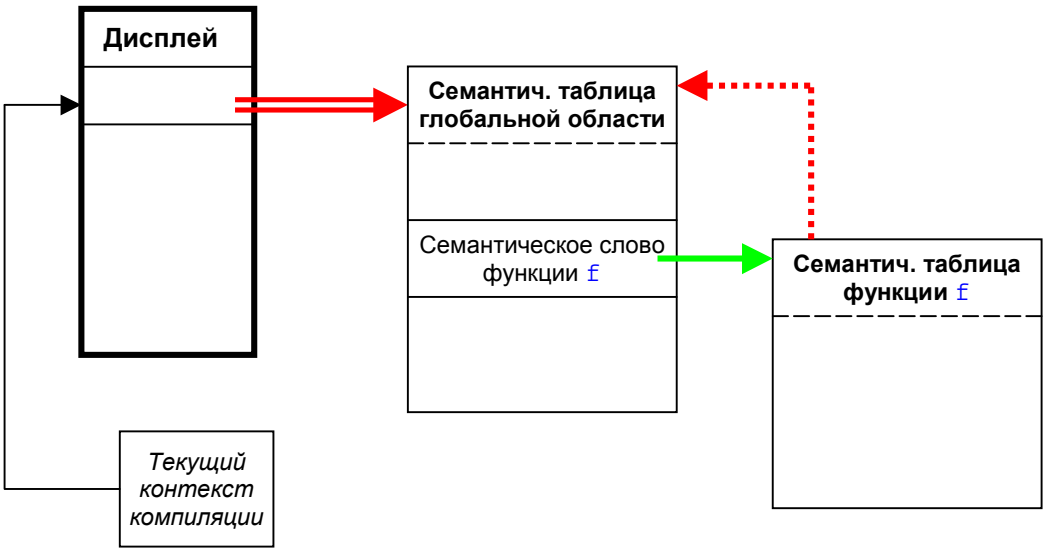
Простейшим примером использования данной операции служит начало компиляции полного описания функции, чье предварительное объявление было дано ранее:

```
      void f ( int a );
        . . .
►     void f ( int a )
      {
            . . .
      }
```

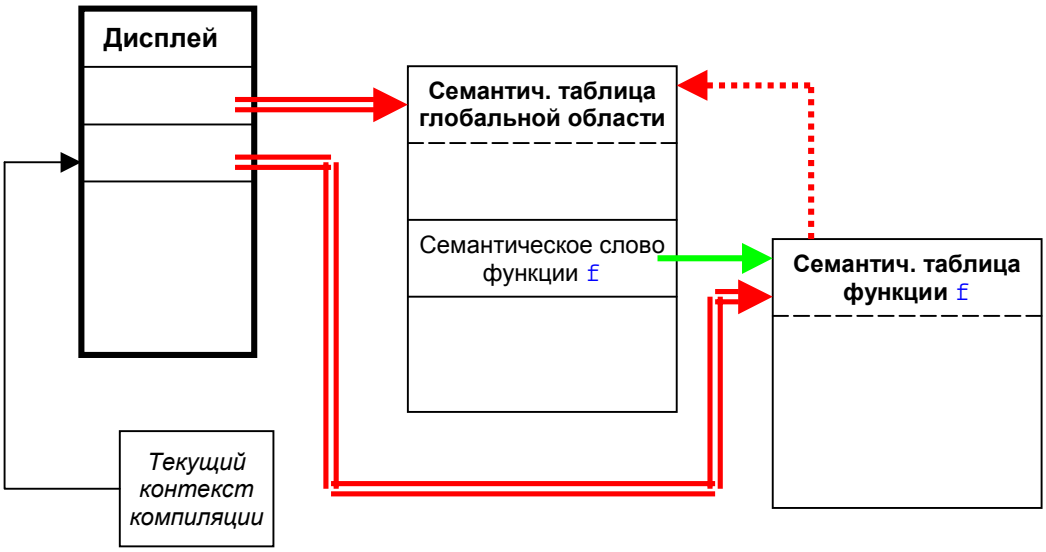
В этом случае в системе таблиц уже имеется информация о функции `f`: в таблице глобальной области действия содержится семантическое слово функции, которое связано отношением владения с таблицей этой функции; в этой таблице содержатся семантические слова для формальных параметров функции. В начале компиляции заголовка полного описания таблица функции добавляется в текущий контекст операцией ***AddScope***.

Данный пример можно иллюстрировать следующими схемами:

а) До начала обработки полного описания функции `f`:



б) При обработке полного описания функции `f` (после выполнения операции ***AddScope***):



Точно таким же образом данная операция используется при компиляции заголовка полного описания класса в случае наличия его предварительного объявления.

Кроме того, описываемая организация семантических таблиц и, в частности, операция ***AddScope*** позволяет естественно реализовать обработку "разрывных" описаний пространств имен, о которых говорилось в разд. 3.X. Упрощенно схему такой обработки можно описать следующим образом. Пусть компилятор получил из входного текста конструкцию `namespace N {`. Если имя `N` не известно в текущей области действия, то он выполняет операцию ***NewScope***. В результате возникает новая семантическая таблица для нового пространства имен, которая помещается в текущий контекст. Если же пространство имен с именем `N` уже имеется в текущей области действия, то операция ***AddScope*** просто добавит ее в текущий контекст. В результате семантические слова для всех последующих объявлений объектов (членов пространства имен) будут помещаться в семантическую таблицу этого пространства имен.

RemoveScope

Текущая семантическая таблица удаляется из текущего контекста. Текущей становится таблица, бывшая текущей до выполнения последней операции ***NewScope*** или ***AddScope***.

В результате имена, объявленные в данном блоке, становятся невидимыми в текущем контексте (хотя они могут быть доступны посредством некоторого семантического отношения).

Эта операция выполняется при завершении "простых" блоков - составных операторов и их смысловых эквивалентов - в условных, выбирающих, циклах, `try`-блоках и `catch`-обработчиках - и симметрична операциям ***NewScope*** и ***AddScope***.

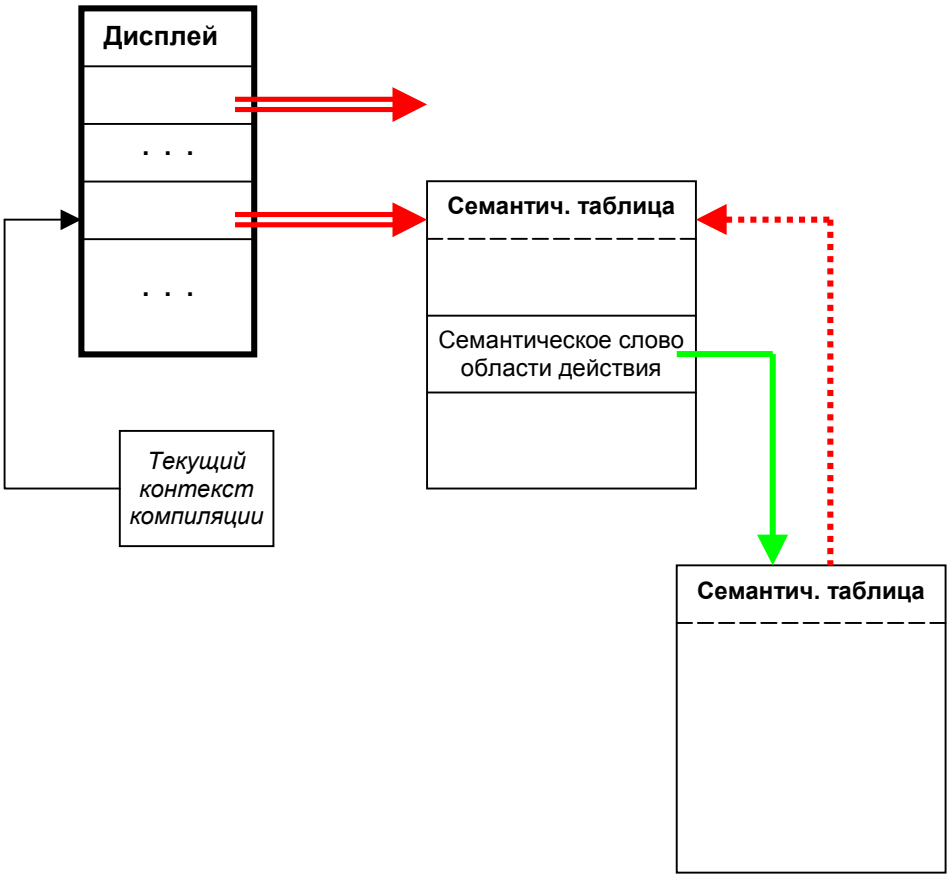
Реализация операции ***RemoveScope*** должна, помимо прочего, учитывать одну семантическую особенность "простых" областей действия: если такая область действия непосредственно или косвенно вложена в область действия функции, то метки, определяющее или использующее вхождение которых появилось впервые в пределах текущей области действия, считаются видимыми во всех областях действия в пределах области действия данной функции [Std, 3.3.4, §1]. Проще говоря, Си++, вслед за Си, допускает переходы внутрь блоков и вовне блоков (с некоторыми ограничениями). При этом в языке отсутствует понятие объявления метки, а область действия метки совпадает с областью действия функции, в пределах которой она появляется.

Такая явная нерегулярность языка, вообще говоря, может быть реализована двояко. Первый вариант предусматривает включение в операцию ***RemoveScope*** действий по переносу семантических слов меток в объемлющую область действия при удалении из контекста текущей области действия. Второй вариант заключается в том, что при обработке первого вхождения метки ее семантическое

слово помещается не в текущую обасть действия, а в ближайшую объемлющую область действия функции.

Выбор конкретного способа реализации, кроме соображений эффективности, должен учитывать ограничения Си++ на использование меток (в частности, запрет "обходов" объявлений с инициализацией, [Std, 6.7, §3]).

Типичное состояние семантических таблиц перед выполнением операции ***RemoveScope*** соответствует рисунку (б) для операции ***NewScope***. После выполнения операции ***RemoveScope*** конфигурация таблиц будет выглядеть следующим образом:



AddBaseScope*(Base)*

Операция устанавливает отношение наследования между текущей семантической таблицей и семантической таблицей, идентифицируемой семантическим словом *Base*. Операция соответствует началу компиляции класса, в заголовке которого заданы базовые классы:

```
class Derived : public Base1, public Base2 { ...
```

Последовательность действий с семантическими таблицами при обработке такого класса можно описать следующим образом:

а) Если класс *Derived* не был предварительно объявлен:

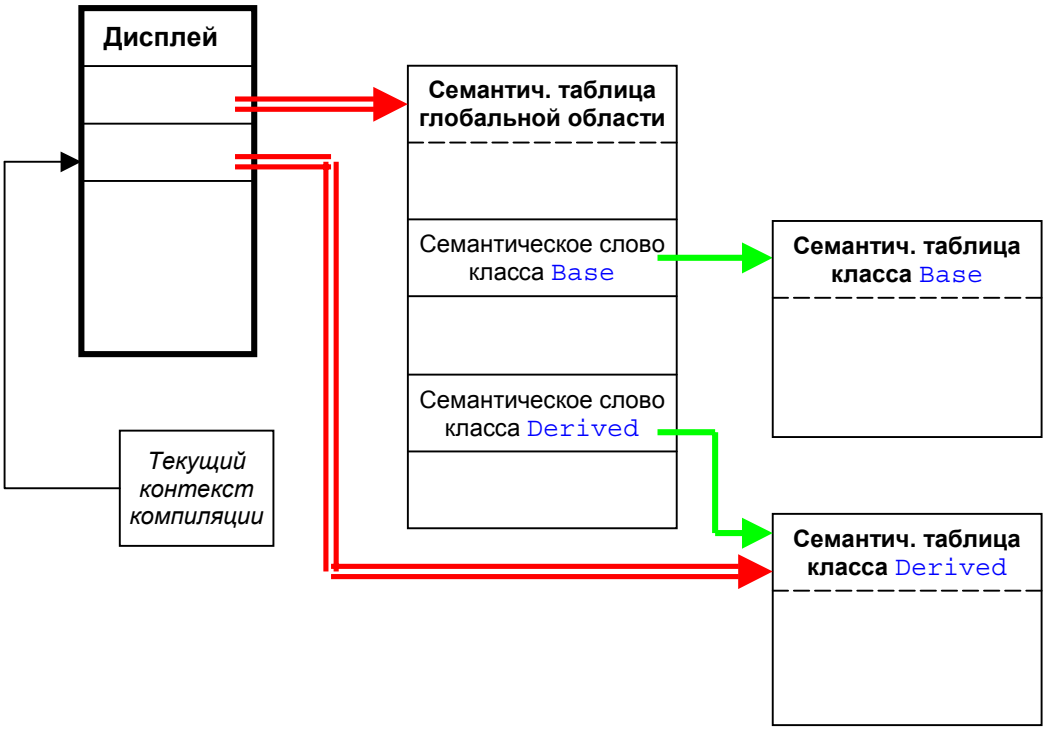
```
NewScope(CLASS)
AddBaseScope(Base1)
AddBaseScope(Base2)
  // Трансляция тела класса
RemoveScope
```

б) Если класс *Derived* был преварительно объявлен:

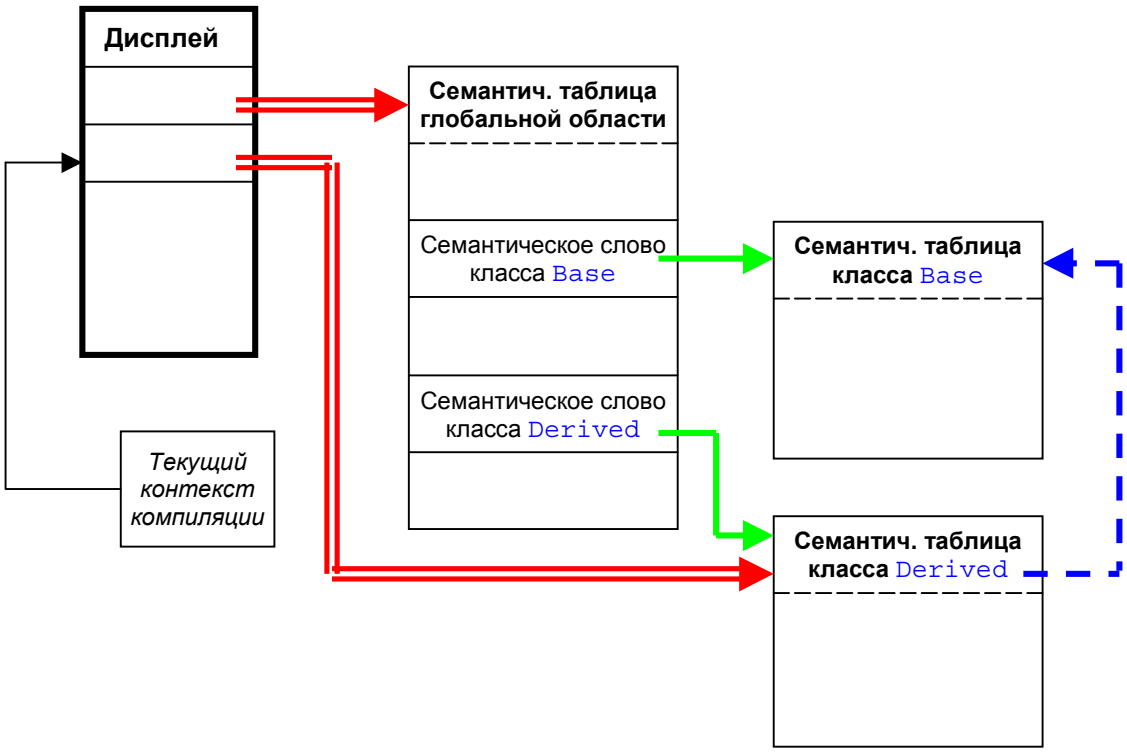
```
AddScope(Derived)
AddBaseScope(Base1)
AddBaseScope(Base2)
  // Трансляция тела класса
RemoveScope
```

Механизм действия операции ***AddBaseScope*** можно проиллюстрировать следующими схемами:

а) До выполнения операции ***AddBaseScope*** (для большей наглядности отношения вложенности не показаны):



б) После выполнения операции ***AddBaseScope***:



Обратим внимание на два важных обстоятельства. Во-первых, классы *Base* и *Derived* не обязательно должны находиться в одной и той же области действия. Классы в Си++ могут быть вложенными или объявляться в пространствах имен; отношение наследования может устанавливаться независимо от места их описания. Необходимо лишь, чтобы полные (квалифицированные) имена классов были видимы и доступны в момент выполнения операции ***AddBaseScope***. Следующий пример иллюстрирует сказанное:

```
class Outer  {
public:
          // описание вложенного класса
     class Inner  { . . . };
};

namespace N  {
          // описание производного класса
          // как члена пространства имен
     class Derived : public Outer::Inner { ... };
}

     // использование производного класса
     // из пространства имен
     // для объявления объекта
N::Derived d;
```

Легко видеть, что операция ***AddBaseScope*** не зависит от позиций, знимаемых в системе таблиц семантическими словами классов, между которыми устаналвивается отношение наследования. Тот факт, что на схемах выше эти семантические слова располагаются в глобальной семантической таблице (то есть оба этих класса описаны в самом внешнем блоке программы) - частный случай общего правила.

Второе обстоятельство заключается в том, что установка отношения наследования *не предполагает* добавление базового класса в дисплей. Согласно определению, данному в предыдущем разделе, базовый класс `Base` находится в текущем контексте, однако класс `Derived` не связан с ним отношением вложенности. Алгоритмы поиска имен при просмотре семантических таблиц текущего контекста должны учитывать вид очередной таблицы. Так, дисциплина просмотра базовых классов в процессе поиска имен отличается от традиционного поиска по вложенным блокам; в частности, наличие объявления искомого имени в одном из базовых классов не скрывает одноименное объявление в других базовых классах.

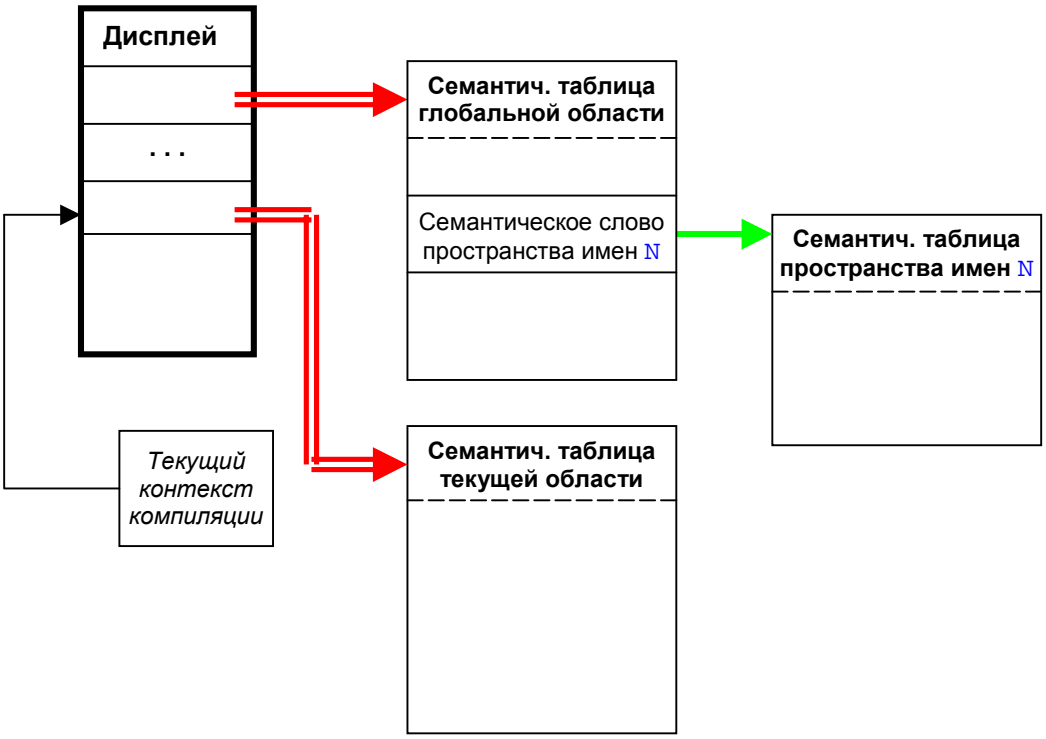
AddNamespaceScope*(Namespace)*

Операция устанавливает отношение использования между текущей семантической таблицей и семантической таблицей, идентифицируемой семантическим словом *Namespace*. Операция составляет основное содержание компиляции *директивы-использования*, встретившейся в некоторой области действия:

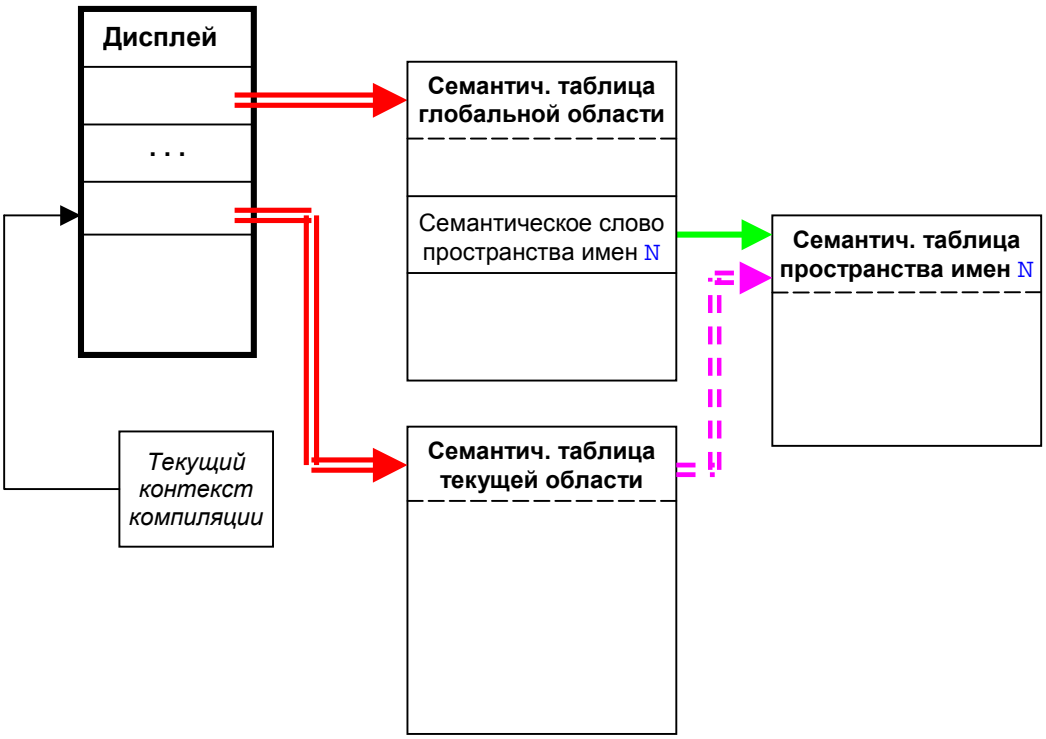
```
{
    . . .
    using namespace N;
     . . .
```

Последовательность действий с семантическими таблицами при обработке такого фрагмета можно описать следующим образом (для определенности предположим, что пространство имен `N` описано в глобальной области действия):

а) До операции ***AddNamespaceScope***:



б) После выполнения операции ***AddNamespaceScope***:



Заметим, что данная операция, так же, как и операция ***AddBaseScope***, не добавляет семантическую таблицу пространства имен в дисплей. По существу, операция только расширяет сферу действия поискового алгоритма на пространство

имен N: теперь при поиске имен таблица этого пространства имен будет просматриваться *вместе* с текущей семантической таблицей, что соответствует семантике языка.

***MakeContext**(Qualifiers...)*

Согласно правилам Си++, сущность, являющаяся членом некоторого класса или пространства имен, может быть полностью описана вне контекста своего объявления. Для этого используется нотация *спецификатора-вложенного-имени* (*nested-name-specifier*, [Std, 5.1, §7]), которая имеет следующий общий вид:

$$Q_1::Q_2::...Q_n::E$$

где Q_i ($i=1...n$) - имена классов или пространств имен, ***E*** - имя объявляемой сущности. Последовательность задания квалификаторов должна соответствовать иерархии вложенности классов и/или пространств имен (при этом пространство имен может быть вложено только в пространство имен, а класс - либо в класс, либо в пространство имен).

Простой пример использования описанного свойства языка приводится ниже:

```
      namespace N  {
            int n;
            class C  {
                  static int a;
                  static int b;
            public:
                  void f ( int );
            };
►           void C::f ( int i ) { b = i; }
            int C::a = 0;
      }
►►    int N::C::b = a+n;  // N::C::a + N::n
```

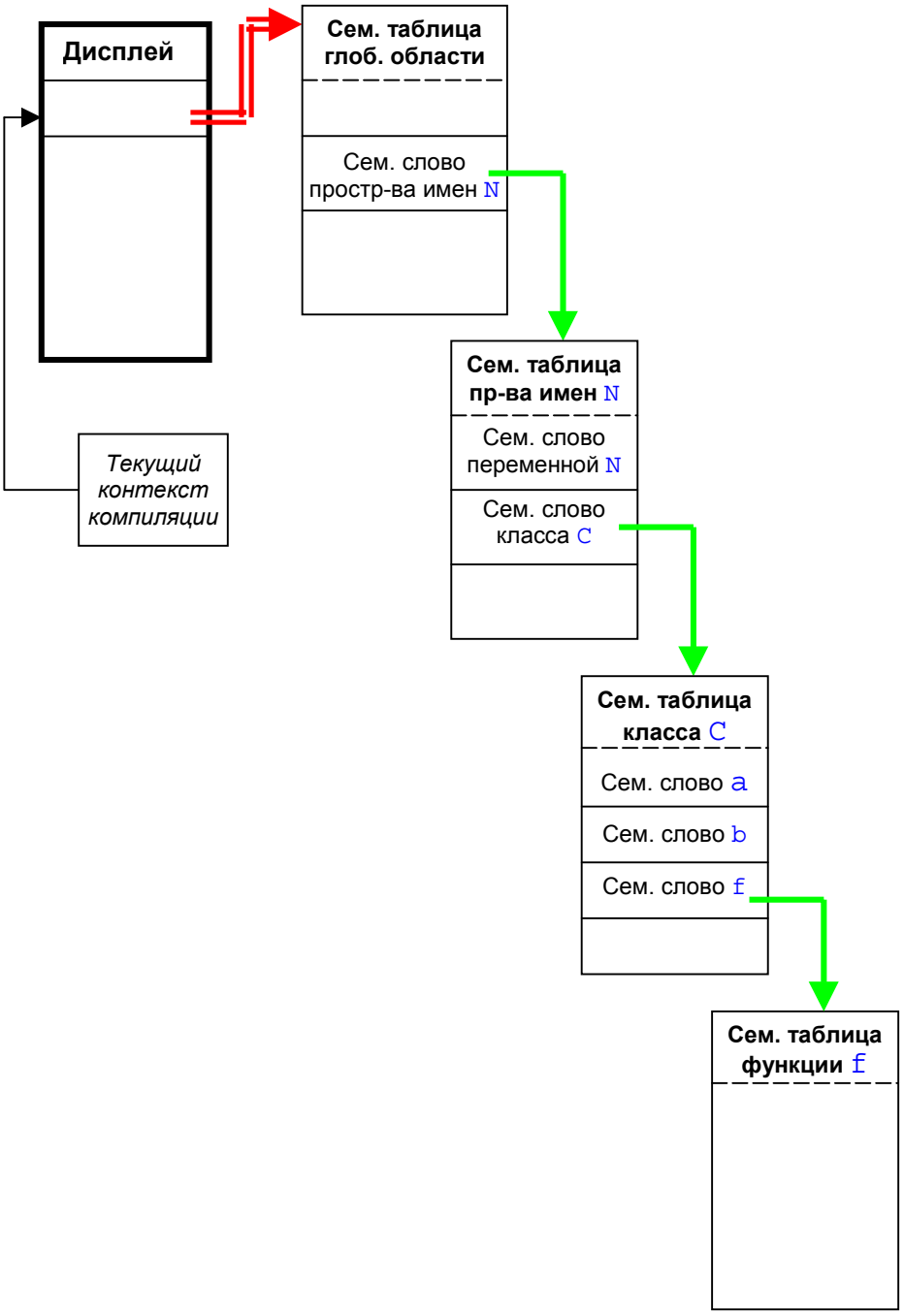
В этом примере класс C описывается в пространстве имен N; при этом два его статических члена a и b и функция-член f только объявляются. Полные описания a и f даются вне класса, в пределах объемлющего пространства имен, а член b - в глбальной области действия. Все три описания используют нотацию *специфкатора-вложенного-имени*.

Принципиально важно, что компиляция указанных описаний должна выполняться в контексте соответствующих объявлений. Так, описание функции f включает использующее вхождение члена b, а описание b - аналогичные вхождения члена a и члена пространства имен n. Однако лексически эти описания располагаются вне области действия, в которой эти переменные видимы. Поэтому перед обработкой описаний необходимо установить контекст соответствующего объявления.

Операция ***MakeContext*** и предназначена для корректной трансляции определяющих вхождений квалифицированных имен.

Операция устанавливает текущий контекст, формируя его из списка квалификаторов. Эти квалификаторы берутся непосредственно из *спецификатора-вложенного-имени*, посредством которого задается описываемая сущность.

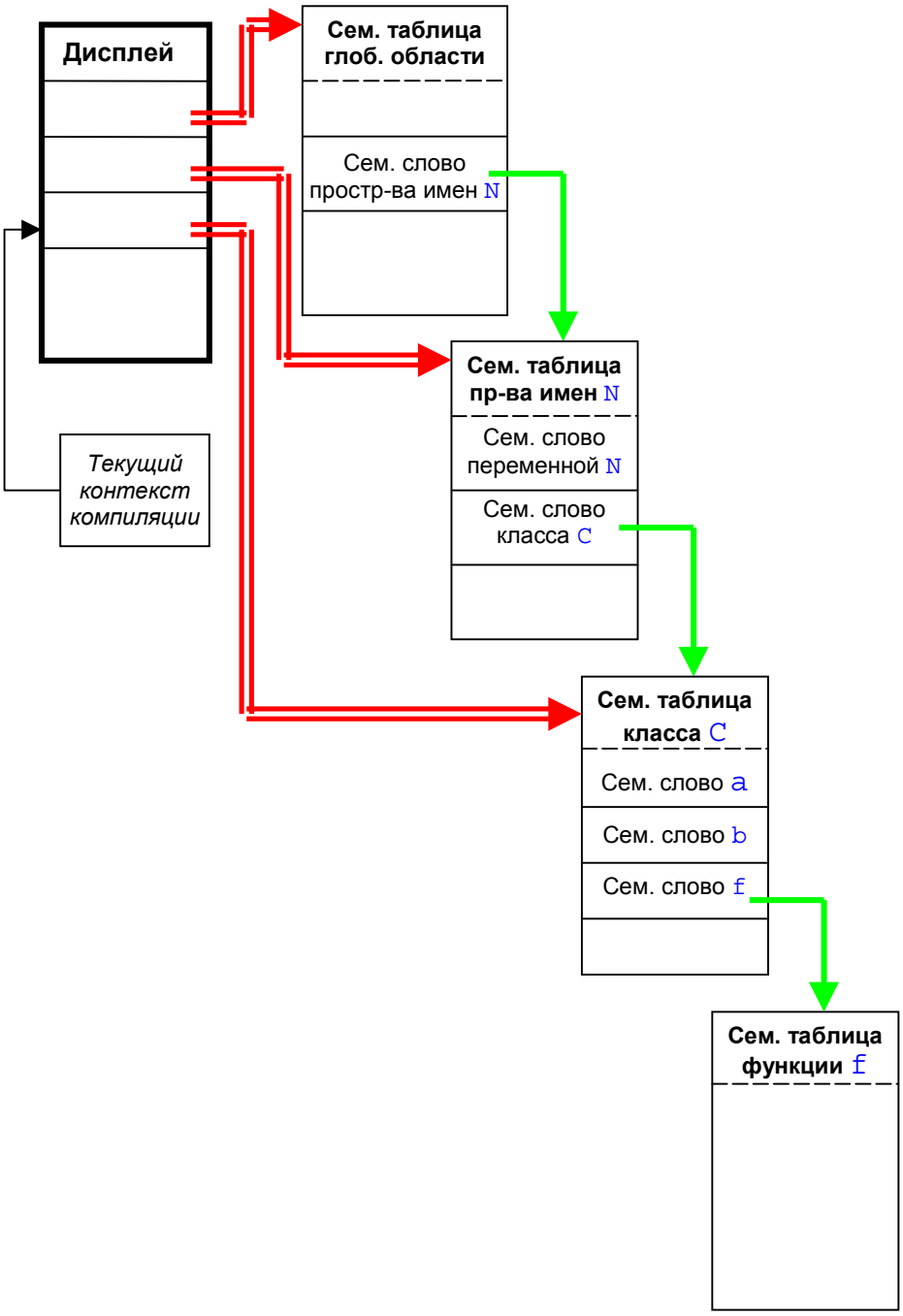
Рассмотрим конфигурацию семантических таблиц непосредственно перед трансляцией описания функции, отмеченного символами ▶▶:



Здесь текущий контекст компиляции образуется единственной семантической таблицей, которая соответствует глобальной области действия. Описание `b` - статического члена класса `C` - находится в этой глобальной области действия. Чтобы компилятор

корректно обработал это описание, необходимо, чтобы оно транслировалось в контексте класса `C`. На это явно указывает квалификация `N::C::`. Эта квалификация относится ко всему описанию в целом, включая и его выражение-инициализатор.

Таким образом, перед компиляцией данного описания необходимо выполнить операцию ***MakeContext***`(N,C)`. После ее выполнения конфигурация таблиц будет выглядеть так:



После выполнения операции ***MakeContext*** выражение-инициализатор рассматриваемого описания будет трактоваться в полном соответствии с семантикой Си++: входящая в это

выражение переменная `a` интепретируется как `N::C::a`, а переменная `n` - как член пространства имен `N`. Семантические таблицы этого пространства имен и класса `C` теперь будут просматриваться поисковым алгоритмом по обычным правилам блочности.

Аналогично будет обрабатываться описание функции-члена `C::f` (отмеченное символом ▶), которое находится в пространстве имен `N`. Перед обработкой этого описания текущий контекст образуется из двух семантических таблиц: таблицы глобальной области действия и таблицы пространства имен `N`. После выполнения операции ***MakeContext***`(C,f)` в текущий контекст будут помещены семантические таблицы класса `C` и функции-члена `f`. В результате члены класса `C` и пространства имен `N` становятся видимыми в текущем контексте, что и требуется для трансляции заголовка и тела функции `f` (в теле функции как раз и используется член `b`).

Кроме помещения в текущий контекст необходимых семантических слов, операция запоминает глубину сформированного контекста (число добавленных в контекст семантических таблиц). Это необходимо для восстановления первоначальной конфигурации семантических таблиц после завершения компиляции описания.

Реализация может хранить глубину контекста либо в последней в контекстной иерархии семантической таблице, либо в самом последнем элементе дисплея.

RemoveContext

Эта операция симметрична предыдущей операции. Она удаляет из текущего контекста семантические таблицы, помещенные в него операцией ***MakeContext***. Глубина удаляемого контекста (количество семантических таблиц) определяется по сохраненному операцией ***MakeContext*** значению.

Возвращаясь к схемам, иллюстрирующим операцию ***MakeContext***, можно сказать, что операция ***RemoveContext*** возвращает конфигурацию, изображенную на второй схеме, к ее первоначальному виду, показанному на первой схеме.

AddEntity*(Attributes)*

Эта операция может рассматриваться в двух вариантах, которые назовем "узким" и "широким". "Узкий" вариант формирует в текущей семантической таблице новое семантическое слово, которое соответствует объявлению некоторой программной сущности, и помещает в него атрибуты этой сущности, извлеченные из ее объявления.

"Широкий" вариант данной операции соответствует обработке одного описания, а также включает добавление в систему таблиц всех структур, представляющих это описание, в частности, новых семантических таблиц (например, для описания класса). В широком варианте эта операция включает все другие операции,

формирующие систему таблиц, и, по существу, является синонимом компиляции описания сущности. "Широкая" трактовка этой операции будет использоваться в главе 4.

Заключение. Выводы главы 3

Объем диссертационной работы не позволяет подробно обсудить все составляющие представленной модели семантических таблиц. Так, за рамками рассмотрений остались модельные операции, определяющие поиск имен в семантических таблицах, вычисление использующих вхождений квалифицированных имен, а также два специфических семантических отношения - отношение дружественности и отношение совместного использования, которые представляют собой абстракции одноименных свойств языка Си++.

Отношение ***дружественности*** связывает семантическое слово некоторой функции (в том числе и функции-члена) с классом, в котором данная функция объявлена как дружественная [Std, 11.4]. Это отношение, как и соответствующее языковое свойство, не является приниципиально важным для модели и влияет только на алгоритмы определения прав доступа к членам класса.

Отношение ***совместного использования*** связывает все одноименные функции в некоторой области действия (то есть, все одноименные функции из глобальной области действия, из некоторого пространства имен, а также одноименные функции-члены одного класса). Это отношение также не имеет определяющего значения для модели и используется алгоритмами поиска наилучшей подходящей функции ("best-matching") при обработке вызовов. Соответствующая модельная операция ***Select****(Function)* анализирует семантические слова всех функций, состоящих в отношении совместного использования с функцией, переданной в параметре, и выбирает среди них наиболее подходящую; критерием являются количество и типы фактических параметров вызова.

Основные **результаты** и **выводы** главы 2 состоят в следующем.

В данной главе на основе подробного обсуждения концепции области действия - одного из базовых понятий языка Си++ - предлагается логическая модель семантических таблиц компилятора переднего плана этого языка. Предложенный способ организации семантических таблиц основан на децентрализованном подходе, при котором каждая область действия моделируется в виде логически независимой структуры, находящейся в определенных семантических отношениях со структурами, представляющими другие области действия программы. Полный контекст программы на Си++ представляется в виде коллекции семантических таблиц, связанных семантическими отношениями.

Основные виды семантических отношений - вложенность, наследование и использование - прямо соответствуют базовым контекстным механизмам Си++. Фундаментальное свойство современных языков, которое заключается в двойственности их

сущностей (в частности, класс Си++ является одновременно типом и областью действия), естественно отображается посредством отношения владения между семантическим словом сущности и ее семантической таблицей.

Предложенная модель семантических таблиц компилятора совмещает обычную блочную иерархию, характерную для традиционных языков программирования, начиная с Algol-60, и продвинутые средства организации программ, присущие таким современным языкам, как Си++ и Ада 95.

Модель таблиц трансляции является достаточно общей и не ориентирована на какой-либо конкретный способ реализации. Вместе с тем, в своей "традиционной" части она допускает эффективную реализацию, в основе которой лежит дисплей, описывающий текущий контекст компиляции. Реализация нетрадиционных механизмов языка, адекватно поддержанных понятием семантических отношений, будет носить характер расширения функциональности традиционных поисковых алгоритмов.

Выполненная реализация представленной модели в действующем компиляторе переднего плана Си++, по мнению автора, подтверждает верность принципиальных решений, положенных в основу модели таблиц трансляции. Характеристики производительности компилятора сравнимы с аналогичными характеристиками компиляторов ведущих мировых фирм.

Дополнительным доводом в пользу выбранной модели служит ее успешное расширение, направленное на поддержку в значительной степени ортогонального прочим механизма языка - аппарата шаблонов (настраиваемых алгоритмов и типов). Обсуждение вопросов, связанных с таким расширением модели, содержится в главе 4.