Глава 4
Механизм шаблонов и модель компиляции для Си++

Данная глава посвящена обсуждению одного из наиболее сложных, но в то же время весьма мощного и гибкого механизма языка Си++ - шаблонов функций и классов. После краткого обзора данного механизма, а также его аналогов в других языках программирования, в главе рассматриваются аспекты реализации шаблонов в компиляторе переднего плана. Основное внимание при этом уделяется адекватности модели семантических таблиц, предложенной в предыдущей главе, для существенно иного, в значительной степени ортогонального другим понятия языка.

Далее в данной главе рассматривается ряд проблем, связанных с использованием механизма шаблонов в языках с возможностью раздельной трансляции, и пути решения этих проблем как в рамках процесса компиляции, так и на уровне архитектуры системы программирования.

4.1 Типовая параметризация структур данных и алгоритмов

Концепция параметризуемых типов ("тип как объект") возникла, согласно [33], в середине 70-х гг. Однако в качестве конкретного механизма общецелевого ЯП эта концепция, по-видимому, впервые воплотилась в виде родовых пакетов и подпрограмм в Аде. Первый пример использования этого свойства языка был дан в самом Стандарте [1] при спецификации библиотек ввода-вывода.

К настоящему времени типовая параметризация зарекомендовала себя как эффективный механизм и реализована как в современных промышенных языках программирования - Ада95 и Си++ , так и в менее распространенных, но перспективных ЯП, например, Eiffel [92].

Существо данного механизма заключается в возможности задания фундаментальных языковых сущностей - структур данных (или типов, определяемых пользователем) и алгоритмов (подпрограмм) в обобщенном виде, который не предусматривает явного указания некоторых их характеристик и параметров. Типичным примером может служить такая известная структура данных, как стек. Основными атрибутами этой структуры выступают тип элементов стека и его максимальный размер. Понятно, что характер операций над стеком (поместить значение в вершину стека, извлечь значение из вершины) не зависит от этих атрибутов и, следовательно, может быть специфицирован без их учета. Тем самым, вместо конкретной структуры (типа) "стек" мы можем получить некий образец для генерации на его основе конкретных типов, например "стек из 100 элементов целого типа" или "стек из десяти указателей на строки".

Аналогичная идея используется при задании обобщенных алгоритмов. Распространенным примером является сортировка, алгоритм которой можно специфицировать без привлечения информации о природе (в частности, типе) сортируемых объектов. По существу, все, что необходимо знать об этих объектах,- это как получить доступ к очередному сортируемому объекту и как их сравнивать между собой. В терминах Си++ эти требования могут быть выражены так: чтобы описать алгоритм сортировки, достаточно, чтобы тип сортируемых объектов допускал применение к ним операций получения доступа к очередному объекту из некоторого агрегата (например, в виде операций разыменования `'*'` и инкрементации `'++'`) и нестрогого сравнения объектов (`'<='`).

Чтобы получить из образца конкретную структуру (тип) или функцию, используется специальный языковой механизм, называемой *настройкой родового модуля* (generic instantiation) в Аде и просто *настройкой* (instantiation) в Си++. В настройке задается имя образца (шаблона) и фактические параметры, необходимые для настройки,- конкретные типы и константы. Существенно, что во всех ЯП, содержащих средства типовой параметризации, механизм настройки реализуется на стадии компиляции, обеспечивая тем самым эффективность результирующего кода, не отличающуюся от обычной.

Поскольку настройки полностью разрешаются при компиляции, а в качестве их параметров задаются конкретные типы (а также константы и статически-вычисляемые указатели), этот механизм не нарушает общий принцип строгой типизации, характерный для современных промышленных ЯП.

Концептуально типовая параметризация в Аде и Си++ обеспечивают одинаковые возможности, однако имеется ряд значимых различий. В частности, при спецификации родового модуля в Аде на типовой параметр накладываются ограничения (то есть, задается так называмый type class - "сорт" или "класс" типа), определяющие диапазон фактических типов и, следовательно, ассортимент операций, допустимых над объектами этого типа. Тем самым можно выполнить более глубокую и потенциально более эффективную обработку родового модуля и заранее (до этапа настройки) выявить многие ошибки. В Си++ такие ограничения отсутствуют, что приводит к переносу основного объема работы по компиляции шаблона на стадию его настройки.

Более существенным недостатком представляется безусловное требование явной настройки любого родового модуля в Аде до первого обращения к нему. Так, чтобы использовать родовую функцию, например, для сортировки массива целых чисел, необходимо предварительно объявить подходящую настройку этой родовой функции, задав при этом ее уникальное имя. Такая избыточная статика на практике приводит к определенным неудобствам при разработке библиотечных пакетов, в которых, как правило, не известен заранее контекст использования родовых модулей. Это обстоятельство было одной из причин неудачи А.Степанова при попытке реализации шаблонной библиотеки для языка Ада (см. интервью Степанова [53]).

В Си++ настройка шаблона функции, как правило, не задается; компилятор автоматически производит настройку при вызове. Так как для любого вызова шаблона функции используется одно и то же имя, пользователь такой функции может даже не знать, что вызываемая им библиотечная функция на самом деле является шаблоном. Информация, необходимая для такой неявной настройки, извлекается компилятором из (статически известных) типов фактических парметров вызова. Процедура такого извлечения называется в Стандарте *выведением аргументов* (deducing, [Std, 14.8.2]) Явная настройка шаблона функции носит необязательный характер и используется, когда, например, параметры функции отсутствуют или информация, получаемая из них, недостаточна для разрешения возможных неоднозначностей, связанных с совместным использованием функций (overloading).

Значение развитой типовой параметризации, на наш взгляд, исключительно велико. В-первых, в настоящее время это единственный надежный и адекватный *языковой* механизм достижения повторной используемости (reusability) программных компонент; прочие способы предусматривают следование (неформально определенным, а потому потенциально нарушаемым) соглашениям об использовании тех или иных интерфейсов и базируются на специальных программных средствах. Примером служат спецификации пакета JavaBeans, встроенного в стандартное окружение языка Java.

Во-вторых, этот механизм, не нарушая строгой типизации ЯП, сам по себе не приводит к падению эффективности результирующих программ, так как реализуется полностью на этапе компиляции. Более того, систематическое и аккуратное использование парадигмы *обобщенного программирования*, основанной на механизме шаблонов, дает возможность строго оценивать и предсказывать эффективность получамых программ, которая гарантированно не будет превышать эффективность функционально идентичных программ, разработанных без использования типовой параметризации. Блестящим примером, подтверждающим сказанное, служит Стандартная Библиотека Шаблонов (Standard Template Library, STL) А.Степанова и М.Ли [51], [52], вошедшая в качестве составной части в стандартную библиотеку Си++ [Std, главы 24, 25, 26].

Заметим, что STL в ее нынешнем виде существенно использует аппарат шаблонов Си++; многие нововведения, внесенные в язык на завершающей стадии стандартизации (начиная с 1995 года) и заметно усилившие средства типовой параметризации Си++, внесены именно с целью более адекватной поддержки STL.

Таким образом, можно определенно утверждать, что наличие в современном компиляторе Си++ поддержки механизма шаблонов в полном объеме является абсолютно необходимым с точки зрения практического использования компилятора.

В то же время практика показывает, что для разработчиков компиляторов Си++ реализация шаблонов представляет собой наиболее (возможно,- самую) трудную и тяжелую задачу. Это подтверждается тем

фактом, что к настоящему времени ни один известный компилятор ведущих мировых компаний-производителей не поддерживает шаблоны в полном объеме Стандарта. И это несмотря на то, что эти компании давно производят компиляторы Си++, а их представители сами вырабатывали спецификации этого языкового свойства, принимая непосредственное участие в процессе стандартизации языка. Обзор состояния реализации шаблонов в 10 распространенных компиляторах Си++ на август 1997 г. содержится в материале [58].

Причина такого состояния дел заключается прежде всего в том, что язык Си++ в целом и его механизм шаблонов в частности крайне нетехнологичны для реализации. Стремление разрешить с помощью языка все известные проблемы, встречающиеся в прикладных программах, необходимость обеспечить совместимость с Си (в том числе, стилевую и синтаксическую совместимость) привели к громоздкому и неоднозначному синтаксису, запутанным конструкциям и чрезмерно сложной семантике с обилием исключений и особых случаев. Кроме того, было необходимо "гладко" совместить относительно новую и активно развивающуюся часть языка (механизм шаблонов) с уже устоявшимися его механизмами. Такую работу можно образно охарактеризовать как "обобщение" практически всех языковых конструкций на случай, когда та или иная семантическая информация об элементах этих конструкций отсутствует.

4.2 Основные требования к реализации шаблонов

Переходя к обсуждению принципов и подходов к реализации шаблонов в компиляторе переднего плана, следует определить исходные предпосылки и требования к такой реализации.

Во-первых, реализация должна соответствовать требованиям, зафиксированным во Введении; прежде всего, необходимо обеспечить сохранение информации из исходного текста и соответствие внутренних структур трансляции конструкциям входного языка. Далее, следует сделать так, чтобы внутреннее представление шаблонов предоставляло возможность трансляции их использующих вхождений в любой единице трансляции и, тем самым, естественно поддержать "экспортирование" шаблонов [Std, 14, §7].

Во-вторых, желательно выдержать общий подход к реализации таблиц трансляции, предложенный в предыдущей главе, избегая чрезмерного переусложнения модели семантических таблиц. Иными словами, хотелсь бы использовать для обработки шаблонов те же внутренние структуры, что и для других конструкций Си++ (классов и функций).

Последнее требование нуждается в кратком пояснении. При проектировании принципов реализации ЯП и, в частности, внутренних структур компилятора, необходимо учитывать все свойства и механизмы реализуемого языка. Поэтому, в принципе, вопрос о соответствии таких структур какому-либо конкретному языковому механизму должен решаться именно на стадии проектрования.

Однако в случае реализации Си++ складывалась несколько иная ситуация. Первоначально механизм шаблонов в том виде, как он был определен в [45], представлял собой относительно несложное в реализации свойство; по существу, речь шла о варианте "контролируемых макросов высокого уровня". Именно так трактовались шаблоны в первых реализациях; например, ранние версии компиляторов Си++ компании Borland рассматривали описание шаблона как макроопределение, которое подставлялось и транслировалось в точках его использования (при обработке настроек). В результате описание шаблона как таковое не транслировалось, а ошибки в таких описаниях выявлялись только при настройке шаблонов.

Но в процессе стандартизации Си++ механизм шаблонов претерпел весьма значительные изменения, которые относились как к возможностям типовой параметризаци как таковым, так и к проблемам интеграции этих средств с другими механизмами языка. На усиление "шаблонной" части Си++ повлияли, как уже говорилось, требования, предъявляемые со стороны Стандартной Библиотеки Шаблонов, ставшие частью стандарта языка. В результате реализация шаблонов в компиляторе переднего плана проводилась на основе той модели семантических таблиц, которая была разработана для исходной версии языка, предложенной в начале процесса стандартизации [45].

Наконец, реализация должна обеспечить максимально полный синтаксический и семантический анализ описания шаблона непосредственно в точке его объявления. Это требование основывается на том обстоятельстве, что *описание-шаблона* считается полноправной конструкцией языка. Альтернативным является подход, при котором шаблоны трактуются как своего рода макросы. В этом случае реализация могла бы сохранять их тексты в некотором буфере и пытаться текстуально подставлять их в места их использования.

Заметим, что хотя Стандарт не содержит явного требования такого рода, сама семантика шаблонов вполне недвусмысленно предполагает компиляционный подход. Иллюстрацией этого обстоятельства служит следующий пример. Рассмотрим шаблон функции:

```
template <class T>
void f ( void )
{
        . . .
      T::t* a;        // объявление или оператор?
        . . .
}
```

Отмеченный комментарием фрагмент тела функции в отсутствие информации о сущности `T::t` можно трактовать двояко: если при настройке данного шаблона член `t` в аргументе для типового параметра `T` окажется типом, то данный фрагмент будет представлять собой *объявление* локального объекта указательного типа. Если же этот член будет представлять собой, например, статическую переменную, то обсуждаемая конструкция является *оператором-выражением* (умножение `T::t` на `a`). Понятно, что при трансляции шаблона

информация о типе `T` отсутствует, поэтому компилятор не может принять какую-либо конкретную трактовку.

В случае "препроцессорного" подхода эта разница не играет существенной роли: весь шаблон текстуально (или полексемно) будет сохраняться в некотором буфере как макроопределение без какого-либо анализа, а операция настройки шаблона на некоторый фактический тип будет представлять собой макроподстановку с заменой формальных параметров на фактические. После такой подстановки компилятор может однозначно проинтерпретировать результирующий текст.

Однако стандарт языка, по существу, отвергает подобный подход: согласно [Std, 14.6, §2], в теле шаблона подобные неоднозначности не допускаются. При задании шаблона требуется явно указать способ интерпретации. Правило заключается в том, что если данная конструкция должна считаться объявлением (то есть, квалифицированное имя `T::t` должно обозначать тип), то это имя следует префиксировать специальным служебным словом **`typename`**. В противном случае компилятор обязан считать `T::t` *не типом*, а всю конструкцию - выражением.

Таким образом, конструкция в исходном примере всегда интерпретируется как выражение. В случае, когда необходимо использовать `T::t` как тип, шаблон должен иметь следующий вид:

```
template <class T>
void f ( void )
{
        . . .
     typename T::t* a;    // объявление
        . . .
}
```

Правило Си++, проиллюстрированное этим примером, разумеется, формально не означает безусловного требования компиляционного подхода (в конце концов, компилятор может и не проводить анализ тела шаблона, считая **`typename`** просто подсказкой, принимаемой во внимание только на этапе настройки). Однако сам дух этого и других подобных правил Стандарта ориентирует разработчиков именно на компиляцию шаблонов в местах их описаний.

4.3 Модель семантических таблиц и механизм шаблонов

Учитывая требования, сформулированные в предыдущем разделе, основной принцип реализации шаблонов в компиляторе переднего плана можно сформулировать так:

Реализация шаблонов выполняется на основе использования *тех же понятий* (прежде всего, *семантических таблиц*), *что и другие механизмы языка*. Модель семантических таблиц, введенная в предыдущей главе, должна быть расширена введением понятий, свойственных и адекватных семантике шаблонов. При этом необходимо проверить и показать пригодность этой модели для всех базисных

концепций, составляющих существо иного понятия языка, каким является механизм шаблонов.

Неформально этот основной принцип можно обосновать следующими соображениями.

Шаблон класса языка Си++ по определению [Std, 14, §1] представляет собой формализм для задания семейства классов, различающихся типовыми и константными атрибутами своего внутреннего устройства. Внешние свойства шаблона (то есть, в терминологии предыдущей главы, его семантические отношения), хотя и могут параметризоваться, но принципиально те же самые, что и соответствующие свойства классов. Иными словами, для шаблона класса можно задавать базовые классы (как шаблоны, так и обычные классы); шаблоны, так же, как и классы, могут быть вложенными или объявляться в пространствах имен, функцию можно объявить дружественной шаблону (то есть, всем классам из семейства, описывамого шаблоном). Единственное отличие заключается в том, что все эти семантические отношения для шаблона являются потенциальными; они актуализуются только для конкретных настроек этого шаблона. Возникающий же при настройке класс (в терминологии Стандарта - *класс-по-шаблону*) полностью идентичен по своим свойствам обычным нешаблонным классам и повторяет схему семантических отношений, заданную для его шаблона-прототипа.

Таким образом, для целей компиляции имеются достаточные основания трактовать шаблон класса как класс, обладающий определенным набором дополнительных свойств (атрибутов) и, быть может, характеризующийся специфическими семантическими отношениями с другими сущностями.

Принципиально так же обстоит дело с шаблонами функций. Аналогично шаблонам классов, *шаблон функции* в Си++ считается способом задания семейства одноименных совместно используемых (overloaded) функций, возможно, различающихся количеством и типами параметров и атрибутами внутреннего устройства. Поэтому в модели семантических таблиц шаблон функции трактуется как обычная функция, обладающая рядом дополнительных атрибутов.

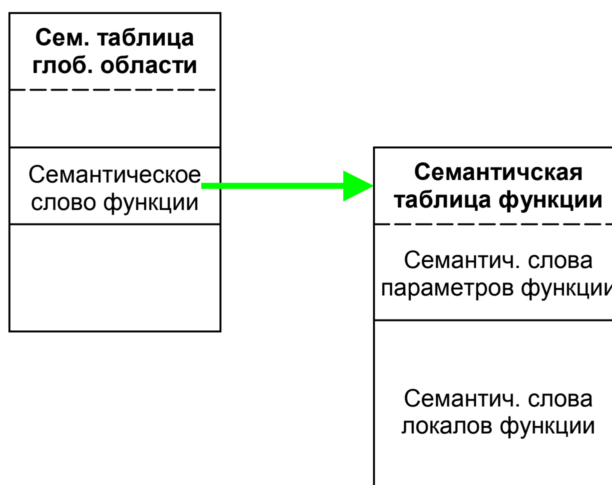
Наконец, отметим, что такие понятия механизма шаблонов, как явные и частичные специализации, не требуют введения новых сущностей в модель таблиц трансляции. Явная специализация естественно рассматривается как частный случай класса- или функции-по-шаблону, а частичная специализация - как вариант шаблона для определенного подмножества параметров. Подробнее эти понятия будут рассмотрены далее в этой главе.

4.4 Представление параметров шаблона

Обсуждение расширений модели семантических таблиц начнем со статуса параметров шаблона. Семантика формальных параметров шаблона существенно отличается от семантики параметров функций, и

следует решить, как они должны представляться в семантических таблицах.

Параметры функции очень близки по своему статусу к локальным объектам функции (их область действия совпадает), поэтому естественно помещать их в семантическую таблицу вместе с локалами функции. Схематически это можно представить так:



Что же касается параметров шаблона, то их сфера действия распространяется как на тело шаблона (в частности, шаблона функции), так и на заголовок шаблона. В отличие от параметров функции, параметры шаблона могут использоваться в объявлениях других параметров этого шаблона, например:

```
template<class T1,class T2 = T1*>
void f ( ...
```

Кроме того, синтаксис частичных специализаций шаблонов допускает задание "сорта типа" (в терминологии языка Ада) непосредственно после имени шаблона. Например, для шаблона класса C:

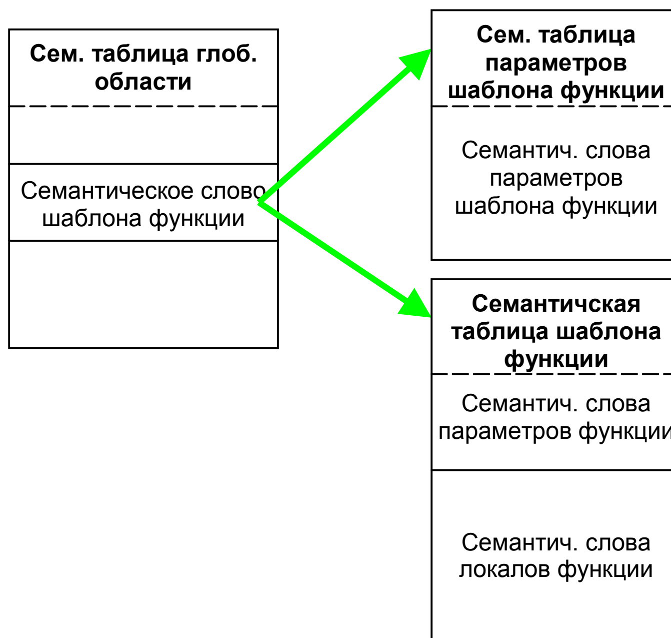
```
template<class T>
class C { ... };
```

его частичная специализация для указательных типов может выглядеть так:

```
template<class T>
class C<T*> { ... }
```

Наконец, основная операция над шаблонами в языке Си++ - их настройка на конкретные типы и константные значения - требует отдельного доступа к параметрам шаблонов вне контекста самого шаблона.

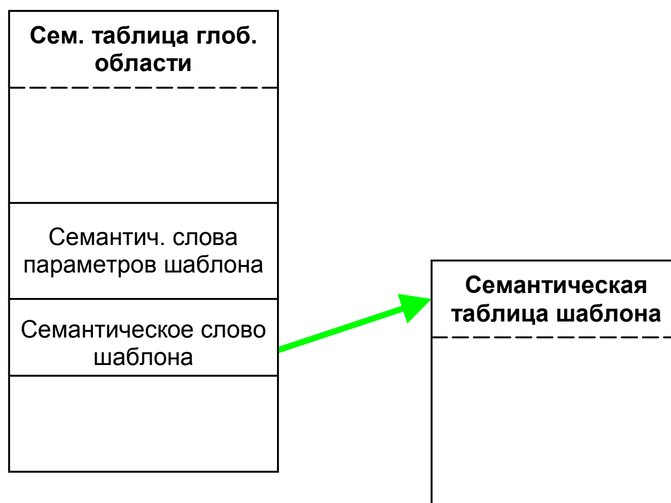
По этим причинам параметры шаблона следует рассматривать отдельно от (обычных) параметров. Можно было бы ввести для параметров шаблона специальную семантическую таблицу; тогда шаблон мог бы представляться следующей конфигурацией семантических таблиц:



Такое решение страдает несколькими недостатками. Во-первых, дополнительная семантическая таблица для параметров шаблона не соответствует никакой области действия, что нарушает концептуальную чистоту модели. Во-вторых, введение такой таблицы потребует внесения в модель дополнительных семантических отношений между семантическим словом шаблона и таблицей для параметров, а также между ней и таблицей самого шаблона, что также не соответствует никаким понятиям Си++ и тем самым отдаляя модель от языка, создавая определенный "семантический зазор" между ними. Наконец, операции управления контекстом трансляции должны предусматривать действия с дополнительной таблицей, что усложняет их семантику.

Внимательный анализ механизма шаблонов позволяет сделать выводу о том, что параметры шаблона по своему статусу ближе всего стоят к имени самого шаблона. Любое использование в программе имени шаблона явно или неявно предполагает либо некоторые конкретные значения параметров, либо их значения по умолчанию, либо их "сорт", задаваемый в частичной специализации этого шаблона

По этим причинам заслуживает внимания представление, согласно которому семантические слова параметров шаблона раполагаются *в той же области действия*, что и сам шаблон, и непосредственно перед его семантическим словом. Это решение, помимо прочего, облегчит компиляцию, так как порядок следования семантических слов будет в точности соответствовать синтаксису описания шаблона. Описанное представление иллюстрируется следующей схемой:



Однако, такая схема в общем случае нарушает принцип скрытия имен, так как сфера действия имен параметров шаблона распространяется только на сам шаблон (на заголовок и тело шаблона); в общем случае имена параметров недоступны в объемлющей области действия. Поэтому это решение нуждается в некоторых усовершенствованиях.

Чтобы обеспечить невидимость имен параметров шаблона вне самого шаблона, семантические слова параметров снабжаются специальным логическим атрибутом "скрытый". Поисковые операции, описанные в предыдущем разделе, должны дополнительно проверять значение этого атрибута, что усложнит их весьма незначительно. В начале компиляции шаблона эти атрибуты устанавливаются равными `false`, что делает их видимыми. Тем самым параметры шаблона автоматически попадают в текущий контекст компиляции. После завершения компиляции значением атрибутов становится `true`, что исключает параметры шаблона из текущего контекста. Действия по установке и сбросу значений видимости для параметров шаблона включаются в семантику операций ***NewScope***/***AddScope*** и ***RemoveScope***, что также не приведет к существенному усложнению этх операций.

В принципе, оба представленных решения концептуально близки. Был принят второй вариант, как более соответствующий духу модели таблиц и требующий меньших модификаций модели. Чтобы обеспечить связь параметров шаблона с шаблоном как таковым, в семантическое слово шаблона добавляется атрибут, ссылающийся на первый параметр. Ненулевое значение этого атрибута позволяет отличать семантическое слово класса (функции) от семантического слова шаблона класса (функции).

Чтобы завершить рассмотрение модификаций модели, связанных с параметрами шаблонов, кратко остановимся на атрибутах их семантических слов. Наиболее естественным является следующая трактовка параметров: типовые параметры шаблонов, как концептуально идентичные typedef-именам, и представляются в этом качестве.

Нетиповые параметры, в зависимости от их типа, представляются либо в виде констант (для параметров целочисленных типов), либо в виде переменных (для указательных и ссылочных типов). В семантические слова соответствующих видов, помимо уже имеющихся атрибутов "начальное значение" добавляются атрибуты "значение по умолчанию", аналогично одноименному атрибуту семантических слов для параметров функции.

После обработки описания шаблона атрибуты "начальное значение" не определены, а в атрибуты "значение по умолчанию" устанавливаются значения умолчаний, извлеченные их соответствующих снтаксических конструкций. При трансляции настройки или частичной специализации шаблона атрибуты "начальное значение" берутся либо из списка значений настройки, либо устанавливаются равными атрибуту "значение по умолчанию", либо извлекаются из заголовка частичной специализации. После завершения обработки настройки или частичной специализации атрибуты "начальное значение" сбрасываются.

4.5 Семантические отношения для механизма шаблонов

Введение в модель таблиц компилятора поддержки механизма шаблонов предполагает отражение в ней специфических семантических отношений, присущих этому механизму. Анализ семантики шаблонов приводит к выводу о наличии двух существенных семантических отношений, требующих подержки в модели: отношение настройки и отношение специализации.

Рассмотрим эти отношения подробнее.

4.5.1 Отношение настройки

Важнейшим понятием, связанным с механизмом шаблонов Си++, является ***настройка***, которую можно неформально определить как операцию получения из некоторого шаблона конкретного класса- или функции-по шаблону, который (которая) соответствует исходному шаблону для заданного набора фактических параметров. Синтаксическую конструкцию, задающую эту операцию, также будем называть настройкой.

Семантическая таблица ***S1*** **настроена** по семантической таблице ***S2***, если и только если соответствующий ***S1*** класс является шаблоном, а соответствующий ***S2*** класс является классом-по-шаблону, полученным в результате настройки (instantiation [Std, 14.7]) класса, соответствующего ***S1***.

В точности так же данное отношение определяется для шаблонов функций и функций-по-шаблону.

Заметим, что, аналогично отношению совместного использования (см. Главу 3), данное отношение действует в рамках одной области действия, так как по определению шаблон и класс-по-шаблону всегда находятся в одной области действия, независимо от позиции настройки.

Рассмотрим пример, приведенный на следующей странице. Пусть имеется шаблон класса C и несколько его настроек, причем настройки шаблона C задаются в различных областях действия: в глобальной области действия, в области действия функции f и в области действия блока, вложенного в тело функции f.

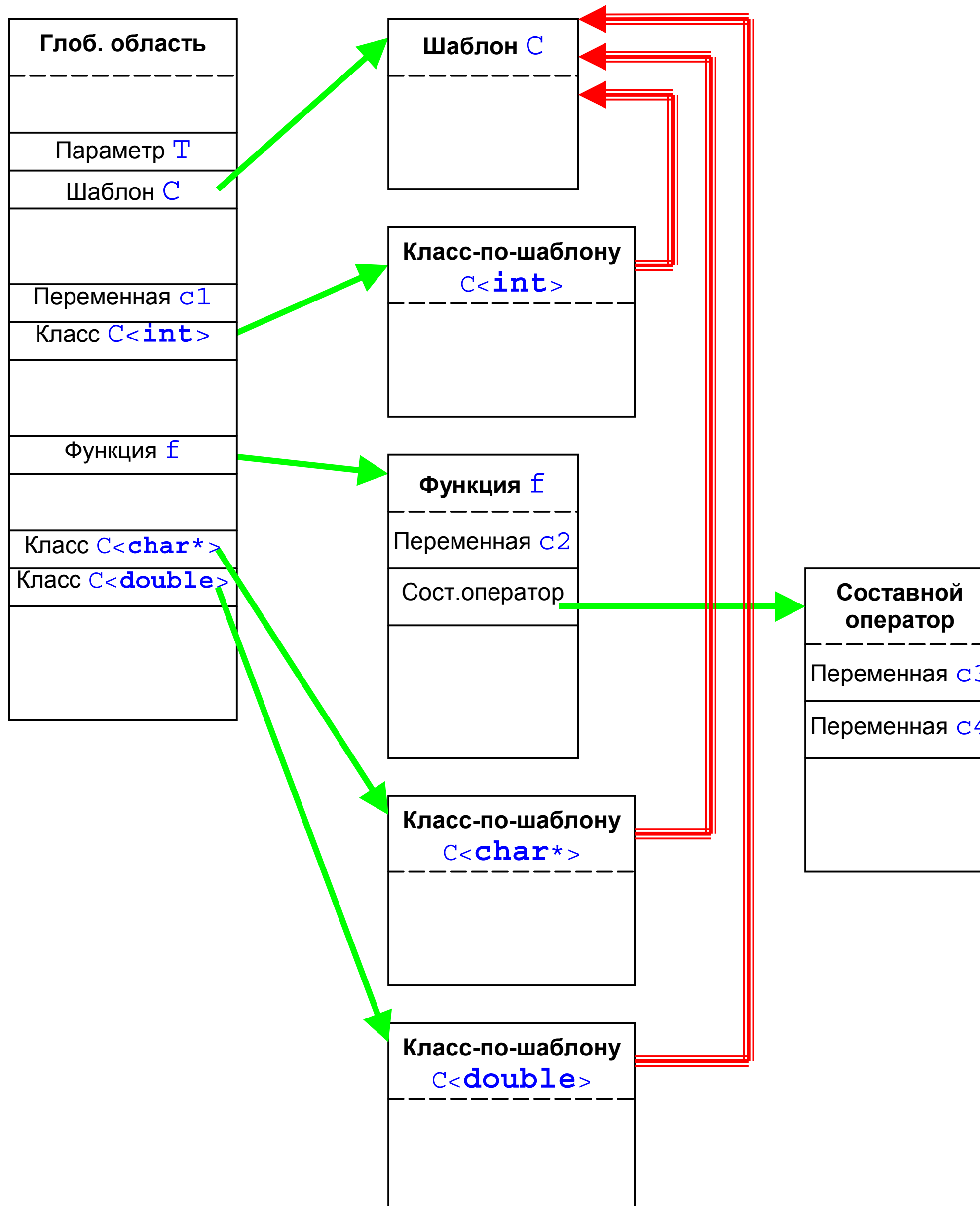
Исполнение компилятором каждой настройки приводит к появлению в программе (и, следовательно, в системе семантических таблиц) соответствующего этой настройке класса-по-шаблону. Имя этого класса-по-шаблону формально совпадает с синтаксисом самой настройки; например, C<**int**> считается именем класса-по-шаблону, полученного в результате настройки шаблона C, фактическое значение параметра T которого есть **int**. Если при обработке настройки выясняется, что класс-по-шаблону с таким "именем" уже существует, то повторная настройка не производится. Так, объявление C<**int**> c4 не приводит к образованию нового класса-по-шаблону C<**int**>, так как класс-по-шаблону с таким "именем" уже был сформирован при обработке объявления объекта c1. При этом тот факт, что объявления c1 и c4 располагаются в различных областях действия, не играет роли, так как классы-по-шаблону всегда помещаются в область действия шаблона-"прототипа".

```
template<class T>
class C  { . . . };
 . . .
C<int> c1;
 . . .
void f (  )
{
     C<char*> c2;
     {
          C<double> c3;
          C<int> c4;
     }
}
```

Фрагмент системы таблиц, соответствующий приведенному примеру, можно изобразить следующим образом.

Компоненты системы таблиц здесь и далее показываются в сокращенном виде: изображаются только существенные для данного рассмотрения семантические отношения Слова "семантическая таблица" и "семантичесое слово" для краткости опускаюся. Из атрибутов семантических слов показываются только их вид, имя и наиболее существенные отношения (например, отношение владения). Если конфигурация текущего контекста компиляции несущественная, то состояние дисплея не показывается.

Отношение настройки представляется тройной стрелкой вида



Заметим, что отношение настройки, вообще говоря, следовало бы сделать симметричным в том смысле, чтобы обеспечить для данного шаблона доступ ко всем его настройкам. Это необходимо для ответа на вопрос, имеется ли для данного шаблона класс-по-шаблону с данным списком фактических параметров. Однако наличие в модели такого "симметричного" отношения приведет к неоправданному усложнению реализации; в то же время выяснить наличие в известной области действия сущности с известным именем можно посредством стандартной операции поиска ***FindInScope***.

Тем не менее, проблема нахождения всех классов-по-шаблону для данного шаблона имеет практическое значение для "некомпиляционных" задач - обратной инженерии, статического анализа программ и т.п. Поэтому в реализацию семантических таблиц, возможно, целесообразно включить поддержку такого рода отношений.

В заключение отметим, что семантика отношения настройки естественно охватывает и такие возможности механизма шаблонов Си++, как явные специализации шаблонов и явные настройки шаблонов функций. Прокажем это на примере явной специализации шаблона (явная настройка шаблона функции концептуально представляет собой идентичный механизм).

Пусть имеется шаблон класса, реализующий обобщенный массив произвольной длины из элементов произвольного типа. Параметры шаблона задают тип элементов массива и его размер.

```
template<class T, int N>
class ARRAY  {
     T arr[N];
      . . .
};
```

Понятно, что для общего случая реализация такого массива будет достаточно сложной, поскольку должна быть рассчитана на произвольный тип элементов массива. В то же время, для определенных частных случаев можно выполнить значительно более эффективную реализацию. Так, для массива булевских элементов размером не более ширины аппаратно поддерживаемого машинного слова можно предложить версию, непосредственно использующую, в частности, логические операции и операции сдвигов. Такая реализция оформляется в виде явной специализации "общего" шаблона для конкретных значений параметров. Внутреннее устройство этой специализации может отличаться от внутреннего устройства "общего шаблона"; в частности, представление массива может быть выполнено в виде логической шкалы в пределах значения какого либо аппаратно поддерживаемого базового типа:

```
template<>
class ARRAY<bool,32>  {
     unsigned long Scale;
      . . .
};
```

С точки зрения организции таблиц такая явная специализация шаблона ничем не отличается от обычной настройки шаблона. Поэтому при ее обработке соответствующее семантическое слово получает статус класса-по-шаблону с "именем" `ARRAY<bool,32>`, а между ее семантической таблицей и семантической таблицей шаблона `ARRAY` устанавливается отношение настройки, в точности как при обработке какой-либо настройки этого шаблона, например, `ARRAY<int,100>`. Обработка явной специализации выполняется той же операцией ***Instantiate*** (см. разд. 4.X ниже), с тем отличием, что тело класса-по-шаблону получается не настройкой тела исходного шаблона, а берется непосредственно из тела специализации.

4.5.2 Отношение специализации

Второе важное средство, связанное с шаблонами, которое должно отражаться в модели таблиц,- частичные специализации.

Механизм частичных специализаций Си++ является средством, уникальным для языков программирования, в которых имеется типовая параметризация. Существо этого механизма заключается в возможности задания для некоторых подмножеств параметров шаблона реализации, отличной от реализации для "общего случая". Так, для некоторого шаблона

```
template<class T,int N>
class C  {
      . . .
};
```

можно задать отдельные, не связанные с данным, варианты для определенных подмножеств параметров. Например, можно определить версию шаблона `C` не для всех возможных типов, а только для указательных:

```
template<class T,int N>
class C<T*,N>  {
      . . .
};
```

Допускается фиксация одного параметра (то есть, задание для него конкретного значения):

```
template<class T>
class C<T,2>  {
      . . .
};
```

Для одного описания шаблона допускается неограниченное множество подобных частичных специализаций. Выбор конкретного варианта (либо первичного шаблона, либо одной из его частичных специализаций) для некоторой настройки производится согласно специальным правилам [Std, 14.5.5.2], которые аналогичны правилам выведения (deducing) параметров при выборе наиболее подходящего шаблона функции.

Например, для настройки `C<int*,10>` будет выбрана первая частичная специализация. По настройке `C<A,5>`, где `A` - некоторый класс, будет сгенерирован класс-по-шаблону из первичного шаблона, а настройка `C<int*,2>` приведет к неоднозначности, так как соответствует одновременно двум частичным специализациям.

Частичные специализации можно считать дальнейшим обобщением явных специализаций: если последние задают специфическую реализацию данного шаблона для конкретного набора аргументов шаблона, то частичные специализации определяют реализацию для подмножества наборов параметров.

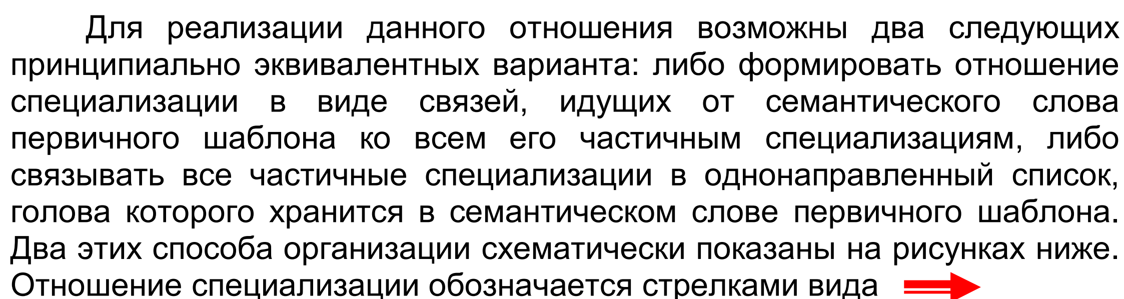
Переходя к представлению отношения специализации в модели таблиц, заметим, что аппарат частичных специализаций подобен

совместному использованию функций в том аспекте, что для одной настройки имеется несколько одноименных специализаций-кандидатов (вместе с первичным шаблоном), а критерием выбора является множество параметров, "наиболее подходящих" по своим типам. При разрешении совместного использования (overload resolution) применяются правила наилучшего соответствия (best matching, [Std, Глава 13]); при выборе подходящей частичной специализации используется так называемое частичное упорядочение (partial ordering) специализаций, основанное, как уже говорилось, на механизме выведения параметров (deducing).

Вместе с тем, необходимо указать на определенную разницу в семантике соответствующих отношений. Отношение совместной используемости (см. заключение главы 3) является полностью равноправным для всех функций, входящих в это отношение, независимо от порядка их задания в области действия. Если фактические параметры вызова не соответствуют ни одной из совместно используемых функций, вызов считается ошибочным (и, следовательно, некорректной является вся программа). В то же время, если фактические параметры настройки (типы, константы и указатели) не соответствуют ни одной частичной специализации, то для настройки выбирается первичный шаблон.

Таким образом, схема построения отношения совместной используемости - связывание совместно используемых функций в последовательную цепочку - для данного случая неприемлема. Отношение специализации следует строить, отталкиваясь от первичного шаблона (тем более, что он всегда должен предшествовать первой частичной специализации). Это позволяет дать следующее определение.

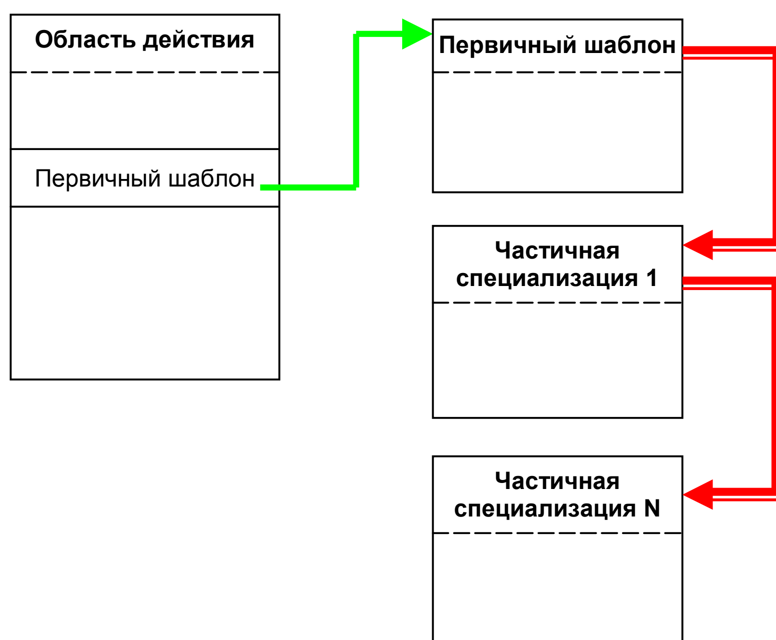
Семантическая таблица $\boldsymbol{S}$ связана с семантическими таблицами $\boldsymbol{S_1}$, **…**, $\boldsymbol{S_n}$ отношением **специализации**, если и только если таблице $\boldsymbol{S}$ соответствует шаблон, а таблицам $\boldsymbol{S_1}$, **…**, $\boldsymbol{S_n}$ - частичные специализации этого шаблона.

Для реализации данного отношения возможны два следующих принципиально эквивалентных варианта: либо формировать отношение специализации в виде связей, идущих от семантического слова первичного шаблона ко всем его частичным специализациям, либо связывать все частичные специализации в однонаправленный список, голова которого хранится в семантическом слове первичного шаблона. Два этих способа организации схематически показаны на рисунках ниже. Отношение специализации обозначается стрелками вида 

По соображениям относительной эффективности, а также простоты представления и его унификации с реализациями других отношений был выбран второй вариант

Первый вариант:

Второй вариант:



4.6 Модельные операции для механизма шаблонов

Обсуждение расширений модели таблиц трансляции для поддержки механизма шаблонов завершим описанием основных операций над сущностями модели.

AddSpecialization*(Template,Specialization)*

Операция добавляет специализацию, семантическая таблица которой задана во втором параметре, в отношение специализации для первичного шаблона из первого параметра.

Операция соответствует объявлению частичной специализации в исходном тексте программы.

SelectSpecialization*(TemplateName,Arguments…)*

TemplateName - имя первичного шаблона. *Arguments* - список фактических параметров настройки. Операция выбирает подходящую специализацию (из числа находящихся в отношении специализации с данным шаблоном), либо сам первичный шаблон.

Результатом операции является семантическое слово подходящей специализации, либо семантическое слово первичного шаблона.

Instantiate*(Template,Arguments…)*

Операция выполняет генерацию класса или функции по шаблону, используя набор фактических параметров. В системе таблиц возникает новая семантическая таблица для генерируемого класса. В области действия шаблона формируется семантическое слово для новой функции или класса. Между семантическим словом и новой таблицей устанавливается отношение владения; между новой таблицей и таблицей области действия шаблона устанавливается отношение принадлежности. Между таблицей шаблона и новой таблицей устанавливается отношение настройки.

Параметр *Template* представляет собой семантическое слово первичного шаблона, либо одной из его частичных специализаций (специализация должна быть выбрана до выполнения данной операции посредством операции ***SelectSpecialization***, см. ниже). Параметры *Arguments* - список фактических параметров настройки.

Операция не выполняет проверку наличия настройки шаблона для данного набора фактических параметров. Считается, что такая проверка производится операцией ***FindInScope***, а эта операция начинает работу, только если ***FindInScope*** дала отрицательный результат.

Операция ***Instantiate*** выполняется при обработке компилятором настройки шаблона. В качестве примера рассмотрим следующий фрагмент программы:

```
template<class T,int N>
class C  {
     . . .
};

. . .
C<int,10>
. . .
```

Типичная последовательность действий при обработке этого фрагмента может быть описана следующим образом (для описания используется некоторый модельный язык, смысл операторов которого очевиден):

```
if FindInScope(Scope(C),C<int,10>,findCLASS) = nil
then
     Template := SelectSpecialization(C,int,10);
     Instantiate(Template,int,10)
endif
```

Здесь конструкция ***Scope***`(C)` обозначает действие по актуализации отношения принадлежности шаблона с именем `C`; иными словами, эта операция идентифицирует семантическую таблицу, которой принадлежит шаблон. Второй аргумент операции ***FindInScope*** - имя искомой сущности - условно обозначен посредством оригинального синтаксиса настройки (в реализации в этом качестве традиционно используются "сигнатуры" - уникальные имена, конструируемые компилятором из имени шаблона и литеральных кодов фактических параметров).

Таким образом, мы завершили построение расширений модели таблиц, направленное на поддержку механизма шаблонов Си++. Эти расширения не выводят за рамки понятий, введенных ранее в главе 3. Семантические отношения, введенные для поддержки шаблонов, и соответствующие операции логически строятся так же, как и отношения принадлжености, наследования и использования. Структура введенных отношений достаточно проста и может быть эффективно реализована.

4.7 Модель программы Си++: раздельная компиляция или модульность

Вторая часть настоящей главы посвящена рассмотрению вопросов системной поддержки создания программ на Си++, активно использующих механизм шаблонов. Основная проблема, обсуждаемая в этом и последующих разделах, заключается в том, что структура Си++-программ вместе с исторически сложившимися правилами программирования, берущими свое начало с языка Си, входят в определенное противоречие с новыми возможностями языка, что приводит к значительным трудностям при разработке больших программ.

Сформулируем основные аспекты этих проблем.

Программа на Си++ представляет собой совокупность *единиц-трансляции* (*translation-units*, [Std, Глава 2, §1]), каждая из которых может храниться и обрабатываться компилятором независимо от других. Кодовые образы единиц трансляции, по существу, представляют собой заготовки, подлежащие последующему объединению. Формирование исполняемого образа всей программы осуществляется отдельным, независимым от компилятора процессором, который традиционно называется редактором связей.

Такой подход, который естественно именовать **раздельной компиляцией**, сам по себе не может вызывать возражений. Удачное и обоснованное разбиение (проекта, а в дальнейшем и текста) программы на отдельные компоненты дает возможность повысить ее надежность и организовать коллективную разработку. Кроме того, раздельная компиляция позволяет создавать многократно используемые компоненты.

Основная проблема и одновременно принципиальная слабость языков Си и Си++ заключается, на наш взгляд, в том, что, во-первых, их понятие раздельной компиляции практически не поддерживает естественное, надежное и безопасное *взаимодействие* единиц трансляции. Это относится как к информационному, так и к управляющему взаимодействию. Во-вторых, Стандарт допускает отсутствие контроля за нарушениями правил межмодульного взаимодействия Остановимся на этих обстоятельствах несколько подробнее, привлекая для обсуждения соответствующие положения Стандарта Си++.

Несколько упрощая [Std, 3.2], можно сказать, что любой объект с внешним связыванием, описанный в некоторой единице трансляции, считается потенциально доступным в пределах всей программы, то есть, во всех других единицах трансляции. Чтобы актуализовать доступность объекта в некоторой другой единице трансляции, нет необходимости явно указывать единицу-"поставщик" этого объекта (для этого нет и соответствующего языкового средства); достаточно объявить объект в этой единице как внешний (то есть, со спецификатором `extern`). То же относится и к функциям с внешним связыванием.

Согласно правилу одного описания (one definition rule, [Std, 3.2, §3]), указанные сущности могут встречаться в программе не более одного раза. Однако, чтобы контролировать выполнение этого правила компилятор вынужден был бы анализировать все единицы трансляции, образующие программу, что вступает в противоречие с принципом раздельной компиляции в том виде, в котором этот принцип заимствован из языка Си. Поэтому разработчики Стандарта вынуждены были сопроводить правило одного описания оговоркой, что при его нарушении "диагностика не требуется" [Std, там же]. Такое допущение, по существу, *отменяет* правило одного описания, так как на практике оно означает, что в нескольких единицах трансляции *можно* задать одноименные объекты; такая ошибка диагностирована не будет, а эффект использования таких объектов в пределах всей программы никак не определяется языком и его компилятором и полностью зависит от "интеллекта" и особенностей реализации редактора связей.

Ситуация с другими программными сущностями - классовыми и перечислимыми типами, шаблонами и т.д.- по существу, аналогична. В программе допускается [Std, 3.2, §5] несколько описаний, например, одноименных классов (по одному в каждой единице трансляции), но на такие описания накладываются ограничения, которые, несколько упрощая, можно свести к требованию, чтобы все такие классы были статически идентичными (вплоть до полексемного совпадения). И самое существенное, что при нарушении этого правила "поведение программы не определено", то есть, опять-таки, проблема реального (надежного и автоматически проверяемого) согласования единиц трансляции выводится вовне языка и его компилятора.

Единица трансляции Си++ представляет собой (по существу, произвольную) коллекцию объявлений и описаний, в пределах которой информация о ее внешних связях никак не выявлена. Такое построение програмы не позволяет компилятору определить, какие ресурсы других единиц трансляции использует данная единица. Иными словами, в Си++ нет средства задания спецификаций единицы трансляции; единица трансляции не является **модулем** в современном понимании этого понятия [2], [96]. Это обстоятельство, на наш взгляд, является фундаментальным недостатком Си и Си++.

Как частичная компенсация указанного недостатка, для организации межмодульного взаимодействия на практике используются так называемые заголовочные файлы. Для каждой единицы трансляции создается специальный текст ("интерфейс"), в который помещаются объявления сущностей из этой единицы трансляции, имеющих внешнее связывание. Такой текст посредством механизма препроцессирования включается в другие единицы трансляции, использующие эти сущности.

Этот подход нельзя считать полноценным решением; он, скорее является некоторым суррогатом модульности. Его принципиальный недостаток заключается все в том же обстоятельстве: язык не содержит механизма для поддержания соответствия между единицей трансляции и ее заголовочным файлом (строго говоря, в языке отсутствует само понятие интерфейса), и компилятор не может проконтролировать ни

корректность заголовочного файла в смысле его соответствия единице трансляции, ни само наличие такого "интерфейса". Механизм препроцессирования (директива `#include`), посредством которого используются заголовочные файлы, по своей семантике является чисто текстовым и не обеспечивает никакого (даже синтаксического) контроля.

Введение в Стандарт языка понятия пространства имен (namespace [Std, 7.3]) представляет собой попытку частичного исправления ситуации; однако его практическое использование возможно только на основе тех же заголовочных файлов и способно лишь несколько уменьшить объем неконтролируемых межмодульных связей. Кроме того, механизм пространств имен сосуществует в языке вместе с описанными выше механизмами и не отменяет их.

4.8 Раздельная компиляция и механизм шаблонов

Описанная в предыдущем разделе проблема приобретает особое значение для аппарата шаблонов. Дело в том, что, помимо описаний шаблонов как таковых, эта проблема распространяется и на неявно присутствующие в программе сущности (классы- и функции-по-шаблону, статические члены классов-по-шаблону), автоматически генерируемые компилятором.

Рассмотрим данное обстоятельство на простом примере. Пусть имеется описание функции:

```
void f ( char* t ) { . . . }
```

и пусть необходимо необходимо сделать эту функцию доступной вне той единицы трансляции, в которой она описана; такая возможность обеспечивает связь компонент по управлению и является типичной в реальном программировании.

Стандартное решение для обычных функций выглядит следующим образом. Описание функции располагается в одной единице трансляции, а ее объявление помещается в заголовочный файл, который посредством директивы `#include` вводится во все единицы трансляции, использующие эту функцию. Тем самым соблюдается правило одного описания, а компилятор получает возможность транслировать вызовы этой функции в других компонентах программы, так как все, что необходимо для компиляции вызова - это наличие объявления функции. Связывание использующих и определяющего вхождений функции осуществляется редактором связей на основе (и при условии) идентичности объявлений и описания.

Теперь рассмотрим такую же ситуацию для шаблона функции:

```
template<class T>
void f ( T t ) { . . . }
```

Если использование такого шаблона предполагается в нескольких единицах трансляции, то прием, описанный для функций, не даст желаемого результата. Если в некоторую единицу трансляции поместить

объявление шаблона (например, посредством включения соответствующего заголовочного файла):

```
template<class T>
void f ( T t );
```

то для обработки вызовов функции такой информации будет недостаточно. В самом деле, компиляция вызова такого шаблона, например:

```
... f(1) ...
```

подразумевает, во-первых, настройку шаблона функции фактическим типовым аргументом, извлеченным из фактического параметра вызова (в данном примере - `int`) с порождением функции-по-шаблону вида

```
void f ( int t )
```

и, во-вторых, формирование собственно вызова порожденной функции. Понятно, что для генерации функции-по-шаблону компилятору необходимо полное описание шаблона, которое в данном случае отсутствует.

Прямое решение этой проблемы достаточно очевидно: вынести в заголовочный файл полное описание шаблона и в таком виде включать его во все использующие этот шаблон единицы компиляции. Именно такой подход (назовем его "включающим") широко используется на практике, формируя негативную тенденцию "разбухания" заголовочных файлов: по некоторым оценкам, активное использование механизма шаблонов приводит к тому, что до 40 процентов программного текста сосредотачивается в заголовках. В результате значительная часть программного текста, становясь составной частью многих компонент, подвергается многократным повторным трансляциям, то весьма существенно замедляет компиляцию больших проектов.

Помимо отмеченного недостатка "включающего" подхода, он, как легко видеть, противоречит правилу одного описания: если в нескольких единицах компиляции имеются вызовы с одинаковыми типами фактических параметров, то это приведет к появлению в них описания одной и той же функции-по-шаблону. Как уже говорилось, компилятор в общем случае не в состоянии диагностировать такое нарушение. В результате в исполняемый код могут попасть несколько идентичных копий одной и той же функции-по-шаблону, пришедших из различных единиц трансляции. Заметим, что такое положение дел характерно и для функций-членов шаблонов классов. Данная проблема активно обсуждается в литературе, где она получила название "разбухания кода" ("code bloat").

В процессе стандартизации языка Си++ проблема "встраивания" механизма шаблонов в принцип раздельной компиляции являлась предметов продолжительных и острых дискуссий. Долгое время членам комитета ANSI/ISO не удавалось прийти к решению, которое устраивало бы как авторов языка, так и разработчиков компиляторов. Наконец, в июле 1996 года были приняты предложения компании Silicon Graphics, которые были направлены на то, чтобы, с одной стороны, сделать

шаблоны более легкими для реализации и использования и, с другой стороны, сделать возможной реализацию для них раздельной компиляции.

Принятые решения были направлены на достижение консенсуса и потому выглядят несколько противоречиво. Их существо можно охарактеризовать следующим образом. Во-первых, "включающий" принцип был признан допустимым; с этой целью правило одного описания было соответствующим образом скорректировано (помимо классов и перечислимых типов, в различных единицах компиляции теперь могут входить идентичные "шаблонные" описания). Тем самым, была фактически легализована повсеместная практика переноса большой части программного текста в заголовочные файлы.

Во-вторых, в механизм шаблонов были внесены усовершенствования, направленные на поддержку раздельной компиляции объявлений и описаний шаблонов (такая возможность, в противоположность "включающему", иногда называется "раздельным" подходом). Основная идея заключается во введении служебного слова **`export`**, которое может использоваться, чтобы сделать описание шаблона доступным вне той единицы трансляции, в которой оно появляется.

В общих чертах механизм "экспортирования" выглядит следующим образом. Стандарт устанавливает, что если некоторое объявление шаблона предваряется служебным словом **`export`**, то этого достаточно для обработки той единицы трансляции, в которой это объявление встречается. Например, в следующем фрагменте

```
export template <class T>
void f ( T );
  . . .
f(1);
  . . .
f('a');
  . . .
```

наличие слова **`export`** перед объявлением шаблона делает возможным трансляцию данной единицы, включая обработку настроек данного шаблона. Так как для таких настроек требуется (полное) описание шаблона `f`, компилятор должен каким-либо образом получить его. Конкретные способы поиска в Стандарте не фиксируются.

В результате Стандарт поддерживает обе формы организации исходных текстов - "раздельную" и "включающую", например:

Заголовочный файл `Templ.h`

```
export template<class T>
void f(T t);

template<class T>
void g(T t) { ... }
```

Единица трансляции `Usage.cpp`

```
#include <Templ.h>
f(1);
 . . .
f('a');
 . . .
```

Единица трансляции `Templ.cpp`

```
#include <Templ.h>
template<class T>
void f(T t) { ... }
```

Описание шаблона `g` , будучи помещенным в заголовочный файл, включается в каждую единицу трансляции, в которой есть на него ссылки, и отдельно не компилируется. Шаблон `f`, напротив, описан только в одной единице трансляции, но может быть вызван в любой единице трансляции, которая включает только его объявление. Чтобы выполнить настройку шаблона `f`, которая требуется для обработки вызовов, компилятор должен каким-то образом найти файл `Templ.cpp` и получить описание `f`.

Как уже отмечалось, способ получения компилятором текста полного описания экспортируемого шаблона в Стандарте не специфицируется. Единственное указание на возможный способ решения этой проблемы содержится в [Std, 14, §9], где говорится, что реализация может потребовать, чтобы единица трансляции с описанием экспортируемого шаблона компилировалась перед компиляцией любой единицы компиляции, содержащей настройку этого шаблона.

Таким образом, наряду со сравнительно приемлемым механизмом экспортирования шаблонов в языке, фактически, в полном объеме сохраняется "включающий" подход, практически не контролируемый комплятором, приводящий к увеличению результирующего кода и к появлению трудноуловимых ошибок.

С другой стороны, проблемы, связанные с преодолением негативных последствий "включающего" подхода, так же, как и адекватная поддержка "раздельного" подхода не могут быть реализованы только на уровне компилятора и потому требуют существенной поддержки от других компонент системы программирования. В следующем разделе кратко рассматриваются некоторые пути решения описанных проблем, реализованные в известных системах компиляции.

4.9 Поддержка механизма шаблонов на уровне системы программирования

Компилятор cfront фирм AT&T/USL/Novell/SCO. Этот компилятор сохраняет информацию относительно каждой единицы трансляции, которую он компилирует, в специальном каталоге, называемом `ptrepository`. Во время обычной компиляции никакие настройки не производятся. Когда в процессе разрешения ссылок редактор связей

обнаруживает сущности, которые использовались, но не определены, и если эти сущности являются настройками шаблонов (об этом говорят их внутренние "испорченные" имена, сгенерированные компилятором), то по информации из `ptrepository` редактор связей находит единицу трансляции, содержащую соответствующий шаблон. Далее формируется так называемый синтетический констекст компиляции (synthetic compilation context), состоящий из всех всех необходимых заголовочных файлов и единицы трансляции с шаблоном. Затем компилятор вызывается повторно, транслируя построенный синтетический контекст. В результате генерируется объектный код для необходимой настройки. Этот объектный код затем объединяется с "нормальным" объектным кодом. Этот механизм дает возможность не включать какие-либо заголовочные файлы в единицы трансляции с описаниями шаблонов и, тем самым, описания шаблонов могут быть откомпилированы независимо.

Разработчик, использующий cfront, должен составлять программу, следуя конкретным соглашениям: все шаблоны должны быть объявлены в заголовочных файлах, имеющих расширение `.h`, и для каждого такого файла должна присутствовать единица трансляции (файл с тем же именем и с расширением `.C`), содержащая соответствующие описания. Эти единицы трансляции явно никогда не подаются на компиляцию, а отыскиваются и компилируются при необходимости, согласно описанной схеме.

Эта схема имеет тот недостаток, что она компилирует каждую настраиваемую функцию отдельно (или, в лучшем случае, все функции-члены каждого класса). Даже хотя сама функция часто совсем маленькая, она должна компилироваться вместе с объявлениями типов, на которых основана данная настройка, а эти объявления могут легко превратиться во многие тысячи строк. Для больших систем эти компиляции могут отнять очень длительное время. В принципе, этап связывания можно сделать более "интеллектуальным" и вызывать компилятор для перекомпиляции настроек только когда это необходимо. Но репозиторий cfront не сохраняет никакой "тонкой" информацию о зависимостях, поэтому он часто вынужден делать глобальную перетрансляцию ("перекомпилировать весь мир") из-за незначительного изменения в некотором заголовочном файле. Кроме того, cfront не имеет никакого способа обеспечить, чтобы при компиляции настроек макроопредения препроцессора были установлены правильно (то есть, в точности так, как они были определены в тексте "первичной" единицы компиляции), если они задаются не в командной строке.

Компилятор Си++ фирмы Borland. Этот компилятор, в противоположность cfront, настраивает все сущности-по-шаблонам, на которые есть ссылки в единице трансляции, а затем для удаления повторных описаний настроенных функций использует специальный редактор связей.

Программист, использующий компилятор Borland, должен удостовериться, что каждая единица трансляции видит весь исходный текст, необходимый для настроек всех шаблонных сущностей, на

которые есть ссылки в данной единице трансляции. Это означает, что нельзя ссылаться на некоторую шаблонную сущность в исходном файле, если описание этой сущности не включено в этот исходный файл. Практически это приводит к тому, что либо весь код с описаниями должен быть помещен непосредственно в заголовочный файл, либо каждый заголовочный файл включает соответствующую единицу компиляции (файл с расширением `.CPP`).

Эта простая схема, и она вполне приемлема для небольших программ. Для больших систем, однако, ее использование приводит к формированию очень больших объектных файлов, так как каждый объектный файл должен содержать объектный код (и символьную информацию для отладки) для каждой шаблонной сущности, на которую есть ссылка.

Компилятор переднего плана компании Edison Design Group (далее - EDG). При первой компиляции исходных файлов программы никакие шаблонные сущности не настраиваются, но формируются дополнительные файлы (по умолчанию, с суффиксом `.ti`), которые содержат информацию относительно тех сущностей, которые могли бы быть настроены в каждой единице трансляции.

После связывания объектных файлов выполняется программа, называемая `prelinker`. Она анализирует объектные файлы, ища ссылки и описания шаблонных сущностей, а также добавленную информацию относительно сущностей, которые могли бы быть настроены. Схема работы этой программы такова.

Если `prelinker` находит ссылку на шаблонную сущность, для которой не имеется никакого описания в наборе объектных файлов, он ищет файл, который указывает, что он мог бы настроить эту шаблонную сущность. Когда он находит такой файл, то связывает с ним настройку. Набор настроек, связанных с данным файлом, записывается в некоторый связанный с ним файл запросов настройки (по умолчанию, с суффиксом `.ii`).

Затем `prelinker` повторно вызывает компилятор, чтобы перетранслировать каждую единицу трансляции, для которой файл запроса настройки был изменен. При перетрансляции используются первоначальные параметры командной строки (сохраненные в файле информации шаблона). В процессе перетрансляции компилятор читает файл запроса настроек для текущей единицы компиляции и следует запросам, записанным в нем. Он формирует новый объектный файл, содержащий (дополнительно к уже имеющемуся коду) запрошенные шаблонные сущности. Компилятор также получает файл со списком описаний, в котором перечислены все настройки, для которых описания уже существуют в наборе объектных файлов. Если в процессе компиляции компилятор имеет возможность настроить сущность, на которую имеются ссылки и которой нет в списке, он делает эту настройку. Он передает обратно `prelinker`'у (в файле списка описаний) список настроек, которые были "приняты" таким образом, так что `prelinker` может связать их с файлом. Этот процесс принятия

обеспечивает быструю настройку и связывание настроек, на которые есть ссылки из новых настроек, и уменьшает потребность в перекомпиляции данного файла более чем один раз в процессе `prelinking`'а.

`Prelinker` в цикле повторяет описанные действия, пока не останется настроек, для которых необходима корректировка. После этого обновленые объектные файлы окончательно связываются.

После успешного завершения этого процесса файлы запроса настроек содержат полный набор связываний настроек. Затем всякий раз, когда исходные файлы перекомпилируются, компилятор будет консультироваться с файлами запроса настроек и выполнять указанные настройки, как это делается при обычной компиляции. Это означает, что, за исключением случаев, когда набор требуемых настроек изменяется, `prelinker` обнаружит, что все необходимые настройки в объектных файлах присутствуют, и что нет необходимости ни в каких корректировках связывания настроек. Это истинно, даже если перекомпилируется вся программа.

Схема, реализованная EDG (ее описание заимствовано из [66]), является наиболее продвинутой и изощренной. С одной стороны, она избавляет программиста от ручной работы по комбинированию единиц трансляции и заголовочных файлов и, с другой стороны, сводит к минимуму объем необходимых перетрансляций. В то же время показательно, что реализация этой схемы основывается на генерации и поддержании в актуальном состоянии некоторой дополнительной информации о программе; последующие вызовы компилятора используют информацию, сформированную предыдущими сеансами ее работы. (Как будет показано немного далее, в качестве следующего шага естественной выглядела бы трактовка объектного кода и дополнительных структур, порождаемых компилятором, как единого репозитория, содержащего информацию обо всех единицах трансляции, образующих программу.)

На основе рассмотрений этого и предыдущего разделов данной главы можно сделать следующие выводы.

1. Модель программы, положенная в основу Си++, допускает раздельную компиляцию компонент и в то же время не содержит развитых модульных средств. Механизм шаблонов, являясь потенциально очень мощным инструментом программирования, входит в серьезное противоречие с архаичной моделью программы Си++. Не вводя в язык существенных нововведений (то есть, оставаясь в рамках определения языка, данного в его Стандарте), улучшить ситуацию принципиально невозможно. Следовательно (и это подтверждает мировая практика), решение проблемы следует искать в направлении совершенствования архитектуры систем (сред) программирования, поддерживающих процесс разработки программ на Си++.

2. Большинство систем программирования для языка Си++ базируется на традиционных компонентах (прежде всего, компиляторе и редакторе связей), работающих независимо друг от друга и

выполняющих строго определенные функции. Усовершенствования, связанные с новыми возможностями языка, оставляют неизменным низкий уровень промежуточного представления программ (объектные файлы), носят характер введения дополнительных компонент (драйверы, пре- и пост-линкеры), базируются на различного рода ухищрениях и принципиально не изменяют общую схему обработки, свойственную таким языкам, как ассемблер и Си. По существу, в таких системах используется два основных механизма: метод пробного связывания с последующей выборочной перетрансляцией, либо усовершенствованная схема редактирования связей, исключающая повторное вхождение фрагментов кода. Оба подхода (это отмечалось также во Введении) страдают специфическими недостатками и достигают результата либо за счет снижения эффективности работы, либо ценой неоправданных затрат ресурсов.

4.10 Семантические таблицы и продвинутая модель компиляции

В данном разделе схематически описывается путь решения проблем, обсуждавшихся в предыдущих разделах, который базируется на модели семантических таблиц. Предлагаемые решения снимают проблемы, связанные с межмодульными взаимодействиями и, в частности, с обработкой шаблонов, и не страдают большинством недостатков, свойственных известным схемам, основанным на традиционных компонентах системы программирования.

Как отмечалось во Введении, основная идея, положенная в основу предлагаемого подхода, заключается в том, чтобы при компиляции единицы трансляции использовать информацию об уже обработанных единицах. В качестве информационного базиса компиляции используется промежуточное представление откомпилированных единиц, ядро которого составляют семантические таблицы.

Сразу отметим, что такой подход в целом не нарушает принцип раздельной компиляции, так как при обработке некоторой единицы информация, привлекаемая из ранее построенного промежуточного представления, используется только для проверки корректности правила одного описания и в целях соблюдения требований Стандарта по настройкам шаблонов. Компиляция некоторой единицы гарантированно будет успешно проведена и без использования существующего ПП, однако при этом могут быть не диагностированы ошибки и/или могут быть повторно скомпилированы некоторые настройки.

Покажем, как могла бы выглядеть обычная последовательность формирования программы на Си++ с точки зрения модели семантических таблиц, предложенной в данной и предыдущих главах.

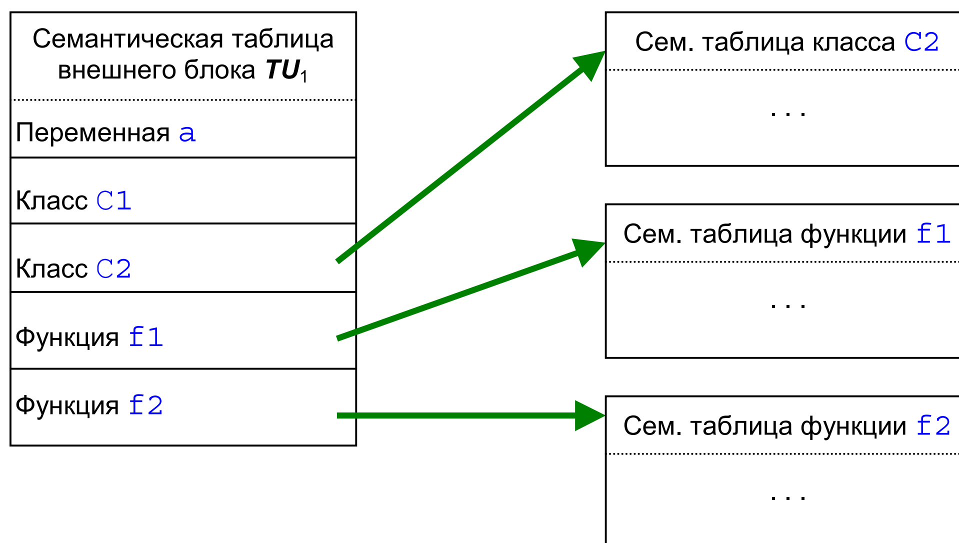
После завершения обработки компилятором переднего плана некоторой единицы трансляции (назовем ее $\boldsymbol{TU}_1$) образуется некоторая иерархия семантических таблиц, "корнем" которой является семантическая таблица, соответствующая внешнему блоку этой единицы. Построенная компилятором система таблиц полностью отражает контекстную структуру единицы трансляции и, что самое

главное в данном рассмотрении, содержит всю информацию о программных сущностях данной единицы, потенциально доступных в других единицах трансляции. Так как объявления таких сущностей могут находиться только во внешнем блоке, вся "интерфейсная" информация сосредоточена в "корневой" семантической таблице.

Пусть единица трансляции ***TU***$_1$ содержит следующие объявления:

```
int a;                       // описание переменной
                             // с внешним связыванием
class C1;                    // объявление класса
class C2  { . . . };         // описание класса
static void f1 ( int x ) { . . . }
     // описание функции с внутренним связыванием
void f2 ( int p ) { . . . }
     // описание функции с внешним связыванием
```

Семантическая таблица внешнего блока после завершения компиляции будет выглядеть так:



Переменная `a`, классы `C1` и `C2` и функция `f2` - сущности с внешним связыванием и, следовательно, потенциально доступны из других единиц трансляции. Функция `f1` локализована в ***TU***$_1$ и недоступна из других единиц. Чтобы другие единицы компиляции могли получить доступ к интерфейсным элементам ***TU***$_1$ обычно формируется заголовочный файл следующего вида:

```
extern int a;
class C1;
class C2  { . . . };
extern void f2 ( int p );
```

который включается в другие единицы компиляции, например, в ***TU***$_2$:

```
#include "TU1"
C1* c1;
C2  c2;

C2* f1;

void g()
{
      f1 = &c1;
      f2(a);
}
```

В результате при компиляции ***TU***$_2$ использующие вхождения сущностей из ***TU***$_1$ разрешаются как внешние из некоторой другой единицы компиляции.

Теперь предположим, что перед началом компиляции ***TU***$_2$ семантические таблицы будут установлены в то состояние, какое они имели *после* завершения компиляции ***TU***$_1$. Это допущение, по существу, эквивалентно ситуации, при которой обе рассматриваемые единицы образовывали бы единую единицу трансляции. Тот же эффект может быть достигнут, например, организацией циклической работы компилятора по нескольким единицам без повторной инициализации семантических таблиц, либо (что наиболее соответствует данным рассмотрениям) сохранением промежуточного представления ***TU***$_1$ после завершения ее компиляции и инициализацией таблиц из этого промежуточного представления перед началом компиляции ***TU***$_2$.

Любой из указанных вариантов означает, что компиляция второй единицы будет проводиться в контекстном окружении первой; тем самым сущности с внешним связыванием из ***TU***$_1$ естественным образом станут доступны для ***TU***$_2$. При этом для разрешения ссылок при обработке использующих вхождений в ***TU***$_2$ сущностей из ***TU***$_1$ не потребуется никаких дополнительных усилий: так как все сущности располагаются в семантической таблице, для этого можно использовать стандартные алгоритмы поиска имен. В этом смысле и устройство семантических таблиц, и основные алгоритмы компилятора будут в точности те же самые, что и для случая обработки одной единицы трансляции в ее собственном контексте.

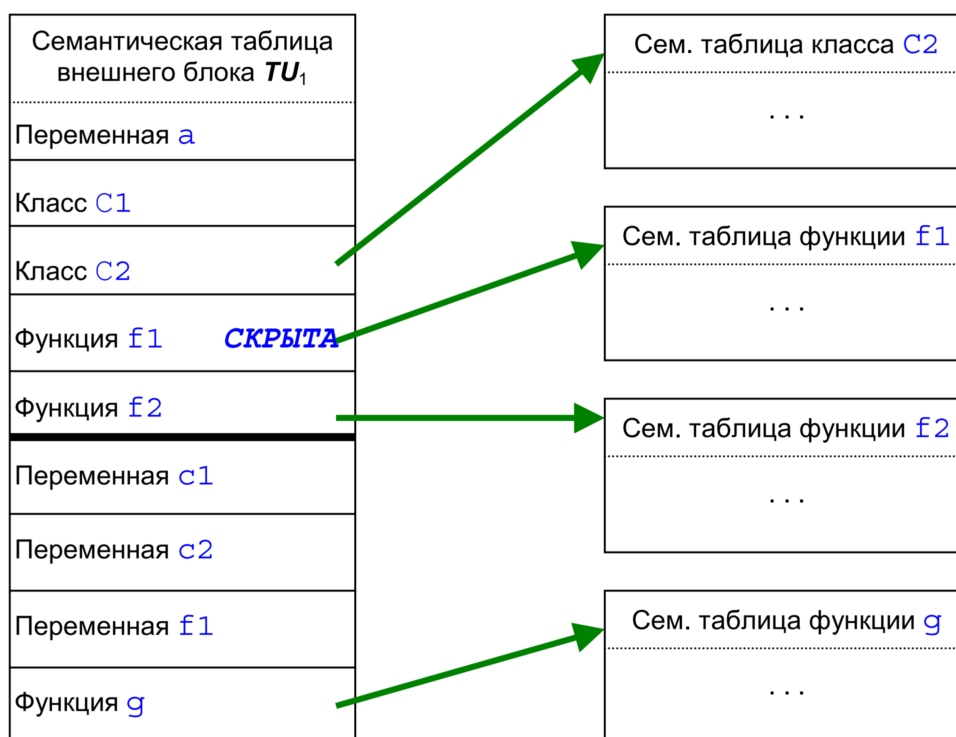
В результате глобальные области действия ***TU***$_1$ и ***TU***$_{12}$ будут совмещены. Это можно трактовать таким образом, что сущности из глобальных областей действия единиц трансляции программы образуют единую область действия, что полностью согласуется со Стандартом [Std, 3.3.5, §3].

Одна из проблем, возникающих при такой организации - возможные конфликты имен сущностей из глобальных областей действия, но имеющих внутреннее связывание. В рассматриваемом примере функция `f1` из ***TU***$_1$ описана как статическая и в этом качестве не должна быть доступна никаким другим единицам компиляции. В то же время, если единицу ***TU***$_2$ компилировать в контексте, построенном к моменту завершения компиляции ***TU***$_1$, то, во-первых, функция `f1` будет видима

для ***TU***$_2$ (что неправильно, но может и не привести к катастрофическим последствиям) и, во-вторых, она может конфликтовать с такими же именами из ***TU***$_2$, например, с именем переменной `f1`.

Решение этой проблемы, тем не менее, возможно без выхода за рамки описаннной модели семантических таблиц. Для этого достаточно учесть, что разрешение всех имен, имеющих внутреннее связывание, уже полностью выполнено в процессе обработки единицы трансляции. К моменту ее завершения все использующие вхождения таких имен однозначно связаны с соответствующими объявлениями. Поэтому после завершения компиляции данной единицы достаточно некоторым образом скрыть все имена из глобальной области, имеющие внутреннее связывание. Для этого хорошо подходит атрибут семантических слов "скрытый", введенный для параметров шаблонов (см. разд. 4.4). Если после завершения компиляции установить значение этого атрибута для всех глобальных сущностей с внутренним связыванием, то, как уже говорилось, алгоритмы поиска имен будут их игнорировать. Тем самым, они никогда не войдут в конфликт с такими же именами из других единиц.

Покажем, как могла бы выглядеть система таблиц после компиляции ***TU***$_2$ в контексте единицы ***TU***$_1$:



Прежде чем развивать предлагаемую модель компиляции, сделаем еще несколько замечаний в обоснование ее практичности и полезности.

Во-первых, достаточно очевидно, что такая модель исключает потребность в заголовочных файлах как системы организации программ. В самом деле, возвращаясь к приведенному примеру с единицами

трансляции TU_1 и TU_2, отметим, что, скажем, описание переменной `a` в единице TU_1 исключает необходимость дополнительного объявления этой же переменной в единице TU_2 и, следовательно, потребность во включении в единицы заголовочных файлов практически отпадает. Можно утверждать, что описываемая модель компиляции (без каких бы то ни было дополнительных накладных расходов!) реализует автоматическое формирование интерфейсов единиц трансляции: таким интерфейсом можно считать саму глобальную семантическую таблицу. Без всяких дополнительных операций и преобразований этот интерфейс доступен всем другим единицам трансляции программы. В этом отношении прослеживается некоторая аналогия с моделью модульности языка Oberon-2 [96], в котором интерфейсы модулей (правда, в текстовом виде), формируются как побочный продукт работы компилятора.

С другой стороны, безусловная необходимость поддерживать привычные традиции программирования систем (прежде всего, технологию заголовочных файлов) может несколько осложнить обработку. Простые ситуации не должны вызывать проблемы. Так, описание переменной `a` в единице TU_1 (эта переменная по умолчанию имеет внешнее связывание) по правилам Си++ не будет конфликтовать с объявлением *этой же* переменной с описателем **`extern`** из TU_2 (там оно появилось из заголовочного файла). Согласно Стандарту [Std, 3.1, §3], фрагмент вида

```
int a;
  . . .
extern int a;
```

представляет собой допустимую последовательность: первая конструкция считается описанием переменной, вторая - это (повторное) объявление той же самой переменной.

Несколько сложнее дело обстоит с теми сущностями, многократные *описания* которых в различных единицах трансляции в языке допускаются. Это относится [Std, 3.2, §5] к классовыми и перечислимым типам, шаблонам классов и функций, специализациям шаблонов и некоторым другим сущностям. Так например, в вышеприведенном примере описание класса `C2` встречается в обеих единица трансляции (в TU_1 - непосредственно, в TU_2 - посредством включения заголовочного файла). Согласно Стандарту, все такие вхождения должны удовлетворять нескольким жестким критериям (прежде всего требуется полексемное совпадение описаний), и все они обозначают одну и ту же сущность. На практике это совпадение и обеспечивается посредством включения во все единицы трансляции одного и того же заголовочного файла с такими описаниями. Однако ниже будет показано, что существо сложностей в обработке таких многократных вхождений в представляемой модели компиляции принципиально иное, нежели в обычных моделях.

В традиционных системах компиляции, когда связывание единиц производится неязыковым процессором - редактором связей - обеспечить контроль упомянутых требований крайне затруднительно

либо невозможно в принципе (например, редактор связей просто "не знает" таких понятий как классы и, тем более, шаблоны, которые полностью обрабатываются компилятором и вообще не присутствуют в объектных кодах). Практические эксперименты с известными системами программирования показывают, что отождествение описаний классов в различных единицах трансляции либо происходит только по их именам, без контроля соответствия их "содержимого", либо не происходит вовсе (одноименные, но несовпадающие классы из различных единиц по умолчанию считаются различными), без вывода какой бы то ни было диагностики.

В описываемой модели ситуация выглядит следующим образом. При попадании в глобальную семантическую таблицу описания некоторой сущности из числа перечисленных выполняется проверка, нет ли описания одноименной сущности среди уже имеющихся. Если такое описание есть, и оно пришло из другой единицы трансляции (в противном случае фиксируется ошибка - повторное описание), то компилятор может выбрать любую стратегию: либо безусловно совместить эти два описания без дополнительного контроля (как это делается в большинстве систем), либо выполнить их полное сравнение по критериям Стандарта. Собственно, вся сложность, связанная с обработкой многократных описаний, и заключается в сравнительно высоких накладных расходах на такого рода проверки (так как классы или шаблоны могут быть сколь угодно большими и сложно устроенными). Однако понятно, что такая стратегия дает безусловное соблюдение Стандарта, недостижимое в обычных системах. Кроме того, адекватность диагностических сообщений по результатам проверки на совпадение гарантируется тем, что компилятор всегда имеет полную семантическую информацию о сравниваемых описаниях. Такой информацией никогда не обладает редактор связей, имеющий дело с объектными кодами.

Наконец, очень важно, что алгоритм проверки на совпадение описаний на самом деле (в виде составных частей) уже присутствует в компиляторе и реально используется для аналогичных проверок по семантическим таблицам в пределах одной единицы трансляции. Таким образом, включение в модель описанной функциональности, опять-таки, не выводит за рамки модели таблиц и конкретной реализации компилятора.

В заключение данного раздела кратко коснемся вопросов, связанных с обработкой шаблонов в представленной модели компиляции. Из предыдущих рассуждений должно быть понятно, что само наличие некоторого общего контекста трансляции, образованного суммой табличных образов ранее обработанных единиц, снимает большинство проблем с раздельной компиляцией шаблонов. По существу, все, что требуется от разработчика в данной ситуации - это обеспечить опережающую трансляцию описания шаблона относительно его настроек либо его частичных или полных специализаций. Это как раз то требование, которое Стандарт допускает для реализаций компилятора Си++.

Механизм экспортирования объявлений шаблонов становится излишним, так как по своей сути он ориентирован на эффективное разнесение объявлений и описаний шаблона по различным единицам трансляции. При этом семантика всех соответствующих языковых конструкций полностью сохраняется.

4.11 Концепция непрерывной компиляции

Предлагаемая модель компиляции программ на Си++, основанная на семантических таблицах, удобна еще и в том отношении, что она делает излишним само наличие дополнительной компоненты, выполняющей связывание кодовых образов единиц трансляции. В самом деле, как было показано, вся работа по разрешению межмодульных ссылок выполняется самим компилятором по тем же алгоритмам, которые используются им для поиска имен и разрешения ссылок внутри одной единицы. Так как компилятор имеет доступ ко всей информации, извлеченной из исходного текста, он в состоянии давать адекватную диагностику ошибок в терминах исходного текста, а не во внутренних понятиях объектных файлов (например, "внешняя ссылка"), с неизбежным использованием "испорченных" имен (mangled names) и, как правило, без привязки к координатам в исходном тексте.

Итак, согласно данной модели, компилятор, добавляя табличные образы очередной единицы трансляции к системе таблиц, формирует общее промежуточное представление всей программы. На основе полученного таким образом интегрального промежуточного представления можно сгенерировать исполняемый код всей программы. Таким образом, традиционная модель "компилятор переднего плана - генератор объектного кода - редактор связей" заменяется на заметно более простую модель "компилятор переднего плана - генератор исполняемого кода". Во Введении говорилось о дополнительных преимуществах единого промежуточного представления, основным из которых является возможность глубоких глобальных оптимизаций на уровне программы в целом. Генератор кода, работающий по одной единице трансляции, в принципе не может проводить достаточно серьезный оптимизационный анализ межмодульных связей; с другой стороны, оптимизатор, который работал бы по исполняемому коду, полученному редактором связей, не имеет доступа к семантической информации об исходной программе, что существенно ограничивает его возможности.

Продолжим рассмотрение модели программы на Си++. Как уже говорилось, эта модель, основанная на понятии единицы трансляции и не содержащая развитого понятия модуля, представляется весьма архаичной и ограниченной. Однако сама примитивность понятия единицы трансляции дает возможность несколько по-иному взглянуть на принципы построения программ на Си++ и возможную организацию их разработки. По существу, единица трансляции - искусственное, неязыковое понятие, необходимое, скорее, для потребностей технологии разработки программ и отражающее особенности конкретного способа представления и хранения программных текстов.

Действительно важным базовым понятием языка является ***описание*** программной сущности. Единица трансляции и, следовательно, вся программа на Си++ конструктивно состоит из набора описаний, совокупные атрибуты и поведение которых и определяют ее существо. Каким образом (через каких посредников или носителей или агентов) описания доставляются компилятору,- это, по существу, особенности реализации, не имеющие отношения к семантике программы. Будут ли описания каким-то образом сгруппированы (или, наоборот, разнесены) по единицам трансляции и представлены там в текстовом виде, либо они будут храниться все вместе в некотором репозитории во внутреннем представлении, которое при необходимости можно визуализировать в привычном для программиста виде, либо они будут представлены в виде иерархии групп, отражающей процесс коллективной разработки программы - в любом случае существо этой программы определяется именно набором описаний ее программных сущностей - переменных, классов и других типов, функций, шаблонов и пространств имен.

Поэтому естественно было бы взамен механизма, основанного на манипуляциях с (текстовыми) единицами трансляции, предложить иную, более адекватную и гибкую схему, согласно которой программист или группа программистов рассматривают программу на Си++ как *коллекцию программных сущностей* и в процессе ее разработки манипулируют только понятиями объявлений и описаний сущностей. Понятие единицы трансляции в такой модели, тем самым, концептуально становится излишним, либо (что то же самое) становится тождественным понятию глобального описания. Сложные и нерегулярные правила Си++, регламентирующие механизмы межмодульного связывания, заменяются на, по существу, единственное требование опережающего добавления в систему таблиц описания или объявления сущности по сравнению с использующими вхождениями этой сущности в другие описания.

Описание сущности (класса, типа, функции, шаблона) становится, вместо единицы трансляции, основным "квантом" программной информации, обрабатываемой компилятором. Все новые описания компилируются в контексте уже существующих в промежуточном представлении описаний. Саму систему таблиц, модель которой обсуждается в работе, можно было бы назвать ядром *репозитория* программы.

Модель компиляции, ставящую во главу угла понятие описания, естественно назвать **непрерывной компиляцией**, имея в виду существенное сближение (отсутствие разрыва) между этапом разработки некоторой программной сущности и ее включением в контекст создаваемой программы: компилятор воспринимает не единицу трансляции, а описание, и в самом процессе его трансляции (без выполнения традиционного связывания) включает его в общий контекст.

Разумеется, здесь представлена лишь общая концепция непрерывной компиляции; ее реальное применение подразумевает конкретизацию ряда дополнительных аспектов технологического,

организационного и реализационного характера, что выходит за рамки настоящей работы.

Кроме того, язык Си++ представляет собой весьма большой конгломерат понятий и механизмов, и не все они одинаково естественно "ложатся" в эту схему. Однако показателен статус большинства "неудобных" механизмов: практически все они унаследованы Си++ от языка-предшественника. Примером могут служить глобальные статические объекты, область действия которых ограничена единицей трансляции, в пределах которой они объявлены. Понятно, что с исчезновением единицы трансляции (что предлагается в модели) становится неопределенной семантика описаний таких объектов: проще говоря, непонятно, где (в каких других программных сущностях) такие объекты могут использоваться, а в каких - нет, если описание сущности больше не связано с какой-либо конкретной единицей трансляции.

Однако, в Стандарте [Std, 7.3.1.1, §2] механизм глобальных статических объектов прямо признается устаревшим, а их использование не рекомендуется. Взамен предлагается намного более мощный и полезный для организации программ механизм пространств имен (namespaces, [Std, 7.3]), который без каких-либо оговорок соответствует продвинутой модели компиляции.

4.12 Расширенное множество операций модели семантических таблиц

Чтобы адекватно поддержать концепцию непрерывной компиляции на уровне системы таблиц, недостаточно того набора отношений и операций над моделью семантических таблиц, которые были определены выше и которые ориентированы только на компиляцию. В данном разделе рассматриваются расширения модели, минимально необходимые для реализации предавленной выше модели компиляции.

Проведенные в предыдущих разделах расширения возможностей модели семантических таблиц не потребовали сколько-нибудь серьезных модификаций этой модели. Единственное, что следует добавить в нее - операции ***Save*** и ***Restore***. Первая операция сохраняет систему таблиц, построенную компилятором, в некотором хранилище (например, в файле). Вторая операция инициализирует систему таблиц, считывая ее из такого хранилища. Расположение хранилища передается операции в качестве параметра. Операция ***Restore***, вызванная без параметра, могла бы формировать начальное (пустое) состояние системы таблиц перед компиляцией самой первой единицы трансляции программы.

Основное же действие, лежащее в основе модели компиляции, описанной в предыдущих разделах,- это выполнение операции ***AddEntity*** (см. разд. 3.4), добавляющей семантическое слово очередной сущности в семантическую таблицу.

Однако, чтобы сделать модель семантических таблиц полезным и мощным инструментом поддержки процесса разработки, такой функциональности недостаточно. Можно указать, по крайней мере, два

класса действий, помимо простого добавления сущностей, которые, в приницпе, не требуются для компиляции как таковой, однако необходимы для поддержки реального процесса разработки. Это *удаление* описания сущности и *изменение* описания сущности. Такие действия можно формально добавить в модель таблиц, введя следующие операции:

DeleteEntity Удалить сущность из семантической таблицы;

ReplaceEntity Изменить атрибуты сущности в семантической таблице.

Содержание отмеченых действий и составляет операционный базис концепции непрерывной компиляции, основанной на системе семантических таблиц.

Семантика введенных операций существенно отличается от семантики других операций, определенных над системой таблиц. Так, операция ***DeleteEntity*** должна не только удалить семантическое слово сущности из таблицы, но и соответствующим образом скорректировать все другие описания, содержащие использующие вхождения данной сущности. Такие использующие вхождения представляются как в виде семантических отношений (например, удаляемый класс является базовым для некоторого другого класса), так и встречаются в другой составляющей промежуточного представления, не рассматриваемой в данной работе,- дереве программы. Например, удаление семантического слова переменной потребует соответствующей корректировки фрагментов дерева, содержащих ссылки на его семантическое слово.

Заметим, что наличие дерева программы концептуально не меняет конфигурацию модели семантических таблиц, так как все "выполняемые" фрагменты программы на Си++ (обычно и представляемые в виде дерева) прямо или косвенно принадлежат некоторому описанию: переменной (если она описана с инициализатором), функции, функции-члена класса, шаблону функции или шаблону функции-члена. Таким образом, дерево программы реально образуется из деревьев перечисленных сущностей; каждое такое "поддерево" можно считать одним из атрибутов семантического слова соответствующей сущности.

Из сказанного следует, что для поддержки операции ***DeleteEntity*** потребуется ввести новое семантическое отношение, которое отражало бы наличие использующих вхождений сущности в структуры (семантические слова и семантические таблицы), контролируемые другой сущностью. Назовем такое семантическое отношение ***отношением вхождения***. Реализация отношения вхождения, которое по своему смыслу является отношением "один ко многим", и составит основные накладные расходы по введению в модель новых возможностей.

Другой особенностью рассматриваемых операций является действие по коррекции описаний, с которым данное описание находится в отношении вхождения. Вполне очевидно, что такая коррекция, по существу, представляет собой повторную компиляцию всех таких описаний. Таким образом, особенность данных операций заключается в

том, что при их выполнении может быть вызван компилятор. В этом заключается отличие рассматриваемых операций от других и родство этих операций с "широкой" версией операции ***AddEntity*** (см. разд 3.4).

Вторую операцию - изменение атрибутов сущности - можно было бы считать некоторой последовательностью операций ***DeleteEntity*** и ***AddEntity***. Однако удобнее ввести единую операцию, так как в данном случае семантика замены может быть реализована однократным вызовом компилятора.

Заметим, что выбор "кванта" модификаций для модели не играет существенной роли. Минимально необходимым является возможность удаления и замены глобальных сущностей; однако сферу действия этих операций можно было бы распространить и на другие (локальные) сущности, так как существо выполняемых ими действий будет тем же самым. Соответственно, можно варьировать степень детализации отношения вхождения, связывая этим отношением либо только глобальные сущности, без учета границ реального использования одной сущности в другой, либо используя более детальную спецификацию. Например, если некоторая глобальная переменная используется в классе, то можно связать отношением вхождения эту переменную со всем классом; тогда при модификации атрибутов переменной или ее удаления из программы придется перекомпилировать весь этот класс. В случае большей детализации отношения вхождения можно связать этим отношением не весь класс, а, например, только те его функции-члены, в пределах которых реально используется данная переменная. В этом случае при модификации переменной достаточно повторно откомпилировать только эти функции-члены. Реализация детального варианта отношения сложнее, однако в этом случае объем необходимых перекомпиляций можно свести к минимуму, что приведет к более гладкому (непрерывному) процессу разработки.

Следует отметить также ряд менее очевидных проблем, связанных с отношением вхождения. Вообще говоря, при определении объема перекомпиляций, необходимых для коррекции системы таблиц, следует учитывать ряд других отношений. Так, удаление класса должно привести к перетрансляции не только, скажем, описаний объектов этого класса, но и всех классов, для которых данный класс является базовым. В свою очередь, модификация этих производных классов приведет к необходимости коррекции (и, следовательно, перетрансляции) описаний объектов призводных классов, и т.д. Таким образом, отношение вхождения служит только основой (начальной информацией) для построения полного дерева зависимостей, которые необходимо принимать в расчет при модификации данного описания.

При этом перетрансляции гарантированно затронут только те описания, которые реально изменились; объем перетрансляций заведомо не будет превышать перетрансляции, необходимые в традиционной модели компиляции. Обычно при модификации некоторой единицы трансляции безусловно перетранслируется вся эта единица целиком, независимо от характера ее модификаций, даже если изменениям подверглось только одно описание. При модификации

заголовочного файла безусловно перетранслируются все единицы, в которые прямо или косвенно входит этот файл.

Наконец, последнее замечание. Введение указанных операций в общем случае означает необходимость некоторых дополнительных операций и над семантическими таблицами в целом. Например, замена в репозитории класса

```
class C : public A { . . . };
```

на класс с описанием

```
class C : public B1, public B2 { . . . };
```

будет означать не только модификацию семантического слова класса и замену его семантической таблицы, но и изменение отношений наследования этого класса с другими. В общем случае для всех операций из разд 3.4, устанавливающих те или иные семантические отношения, необходимо ввести "симметричные" операции, снимающие эти отношения. Однако такие "симметричные" операции не представляют самостоятельного интереса: они необходимы только при выполнении ***DeleteEntity*** и ***ReplaceEntity*** и, следовательно, могут трактоваться как реализационные подробности этих операций.

Выводы главы 4

В главе обсуждались принципы отображения механизма шаблонов Си++ на логическую модель таблиц трансляции. Для поддержки этого мощного и гибкого механизма в модель введены отношения, выражающие существо аппарата шаблонов,- отношение настройки и отношение специализации, а также соответствующие операции установки данных отношений.

Проведенные расширения не выводят за рамки базовых понятий модели, введенных в главе 3. Семантические отношения для поддержки шаблонов, и соответствующие операции логически строятся аналогично отношениям принадлежности, наследования и использования. Структура введенных отношений достаточно проста и может быть эффективно реализована.

Основной **вывод** из проведенных рассмотрений заключается в том, что предложенная модель семантических таблиц адекватна достаточно широкому спектру механизмов, характерных для современных промышленных языков программирования, и допускает достаточно эффективную реализацию в компиляторах.

Во второй части главы анализируется модель программы Си++ с точки зрения механизма шаблонов. Делается вывод о том, что понятие раздельной компиляции, унаследованное от языка Си, без развитого аппарата модульности недостаточно для адекватной поддержки новых возможностей Си++. Попытка преодоления этого недостатка за счет введения средств экспортирования шаблонов, а также аппарата пространств имен представляет собой частичное решение.

На основе анализа известных путей решения этих проблем в некоторых современных системах программирования предлагается более последовательное и всеобъемлющее решение, в основе которого лежит модель семантических таблиц. Основная идея заключается в компиляции единицы трансляции в контексте ранее построенных таблиц. Такое решение позволяет, с одной стороны, получить единое промежуточное представление программы без привлечения таких неязыковых процессоров, как редакторы связей и, с другой стороны, надежно и адекватно поддержать все виды межмодульных связей Си++.

Принцип раздельной компиляции с точки зрения разработчика программы перестает быть привязанным к тексту единицы трансляции и связывается с отдельным описанием в смысле языка Си++. Понятие единицы трансляции, представляющее собой абстракцию файла с текстами описаний программных сущностей, объективно становится излишним. В результате само понятие программы освобождается от неязыковых аспектов, не ассоциируясь с вопросами ее физических носителей.