Глава 1

Определение и проблемы языков программирования

Критерии эффективности языков программирования

Язык программирования можно определить множеством показателей, характеризующих отдельные его свойства. Возникает задача введения меры для оценки степени приспособленности ЯП к выполнению возложенных на него функций — меры эффективности.

Эффективность определяет степень соответствия ЯП своему назначению. Она измеряется либо количеством затрат, необходимых для получения определенного результата, либо результатом, полученным при определенных затратах. Произвести сравнительный анализ эффективности нескольких ЯП, принять решение на использование конкретного языка позволяет критерий эффективности.

Критерий эффективности — это правило, служащее для сравнительной оценки качества различных ЯП. Критерий эффективности можно назвать правилом предпочтения сравниваемых вариантов [28].

Строятся критерии эффективности на основе частных показателей эффективности (показателей качества) Способ связи между частными показателями определяет вид критерия эффективности.

В качестве частных показателей обычно выступают:

- читабельность (легкость чтения и понимания программ);
- легкость создания программ (удобство языка для создания программ в выбранной области);
- надежность (обеспечение минимума ошибок при работе программ);
- стоимость (суммарная стоимость всего жизненного цикла языка);
- переносимость программ (легкость переноса программ из одной операционной среды в другую);
- универсальность (применимость к широкому кругу задач);
- четкость (полнота и точность официального описания языка).

В свою очередь, каждый из частных показателей зависит от множества разнообразных характеристик.

Рассмотрим самые важные из показателей более подробно [81, 84–86].

Читабельность

Одним из важнейших показателей качества языка программирования является легкость чтения и понимания программ, написанных на нем. По современным представлениям самый длинный период времени из жизненного цикла программ приходится на сопровождение, в ходе которого программы часто модифицируются. Поскольку читабельность программ определяет легкость сопровождения, ее считают существенной характеристикой качества программ и языков программирования.

Читабельность (Readability) языка программирования должна способствовать легкому выделению основных понятий каждой части программы без обращения к документации.

Простота. Простота сильно влияет на читабельность языка программирования. Язык должен предоставить простой набор конструкций, которые могут быть использованы в качестве базисных элементов при создании программы. Желательно обеспечить минимальное количество различных понятий с простыми правилами их комбинирования. Этому мешает наличие в языке нескольких способов описания одного и того же действия. Например, в языке C добавление единицы к целому числу можно записать четырьмя способами. Сильное воздействие на простоту оказывает синтаксис языка: он должен прозрачно отражать семантику конструкций, исключать двусмысленность толкования. Предельно лаконичный синтаксис удобен при написании программы, однако усложняет ее модификацию, поскольку в программе нелегко разобраться. Здесь нужен разумный компромисс — простота не должна быть чрезмерной, не должна приводить к загадкам расшифровки.

Ортогональность. Ортогональность означает, что любые возможные комбинации различных языковых конструкций будут осмысленными, без непредвиденных ограничений или неожиданного поведения, возникающих в результате взаимодействия конструкций или контекста использования.

Например, предположим, что ЯП содержит три элементарных типа данных (целый, вещественный с плавающей точкой и символьный), а также две конструкции данных (массив и указатель). Если обе конструкции могут применяться к этим типам и самим себе, то говорят об их ортогональности, обеспечивающей создание большого количества структур данных.

Когда конструкции языка ортогональны, язык легче выучить и использовать для чтения и создания программ, ведь в нем меньше исключений и специальных случаев, требующих запоминания.

Приведем примеры недостатка ортогональности в языках:

- В языке Pascal функции могут возвращать значения только скалярного или указательного типов, а в языках C и C++ — значения всех типов, за исключением массивов (трактовка массивов в этих языках отличается от трактовки остальных типов). В функциональных языках, языках Ada и Python этот недостаток ортогональности устранен.
- В языке C локальные переменные могут быть определены только в начале блока (составного оператора), а в C++ переменные определяются в любом месте блока (но, конечно, перед использованием).

- В языке Java величины скалярных типов (символьного, целого, вещественного и т. д.) не являются объектами, а величины всех остальных типов считаются объектами. Скалярные типы называют примитивными типами, а типы объектов — ссылочными типами. Примитивные типы используют *семантику значения* (значение копируется во время присваивания). Ссылочные типы используют *семантику ссылки* (присваивание формирует две ссылки на один и тот же объект). Кроме того, в языке Java коллекции объектов и коллекции примитивных типов трактуются по-разному. Среди коллекций в Java только массивы могут содержать примитивные значения. Примитивные значения могут вставляться в коллекции других типов лишь в капсулах объектов-оболочек. В языках Smalltalk и Python, напротив, все величины являются объектами, а все типы — ссылочными типами. Таким образом, в этих языках применяется лишь семантика ссылки и все коллекции объектов создаются в ортогональном стиле.

Излишняя ортогональность может стать источником проблем. Например, из-за разрешения полной ортогональности в ходе компиляции программы не генерируются ошибки, даже при наличии комбинаций, которые логически не согласованы или крайне неэффективны при выполнении.

Следовательно, простота языка должна быть результатом комбинирования небольшого числа элементарных конструкций и ограниченного применения понятия ортогональности.

Структурированность потока управления в программе. Порядок передач управления между операторами программы должен быть удобен для чтения и понимания человеком. Речь идет об ограниченном использовании оператора безусловного перехода goto и применении специальных структур управления. Более детально эта тема раскрывается в главе 5.

Легкость создания программ

Легкость создания программ (Writability) отражает удобство языка для написания программ в конкретной предметной области. Поскольку в каждой предметной области программы имеют свою специфику, очень важно выбирать язык, который ее учитывает. Например, если речь идет об исследовании искусственного интеллекта, следует использовать язык Prolog или LISP, а при решении задач научных вычислений — язык Fortran.

Как и следовало ожидать, характеристики, от которых зависит этот показатель, во многом пересекаются с характеристиками читабельности, ведь чтение текста является неотъемлемым атрибутом как процесса создания, так и процесса понимания программы.

Концептуальная целостность языка. Концептуальная целостность языка включает в себя три взаимосвязанных аспекта: простоту, ортогональность и единообразие понятий. Простота языка предполагает использование минимального числа понятий. Ортогональность позволяет комбинировать любые языковые конструкции по определенным правилам. Единообразие понятий требует согласованного, единого подхода к описанию и использованию всех понятий.

Простота достигается не за счет отказа от сложных языковых конструкций, а путем устранения случайных ограничений на их использование. Так, при реализации массивов следует разрешить их объявление для любого типа данных, допускаемого языком. Если же, напротив, запретить массивы некоторого типа, то язык окажется скорее сложным, чем простым, поскольку основные правила языка изучить и запомнить проще, чем связанные с ними ограничения.

Простота уменьшает затраты на обучение программистов и вероятность ошибок, возникающих из-за неправильной интерпретации программистом языковых конструкций. Естественно, упрощать язык целесообразно лишь до определенного предела.

Если язык содержит большое количество разнообразных конструкций, то программисты могут просто не знать каждую из них. Это приводит к неправильному использованию одних возможностей и игнорированию других. В итоге программисты пишут не самые изящные и эффективные программы. Отсюда вывод: использовать небольшой набор элементарных конструкций и согласованных между собой правил их комбинирования намного удобнее, чем применять большое количество конструкций.

Еще раз напомним: ортогональность приемлема в разумных пределах. Чрезмерная степень ортогональности усложняет применение языка. Если разрешена любая комбинация элементарных конструкций, то логические ошибки программы могут просто остаться незамеченными, их не сможет выявить компилятор.

Естественность для приложений. Синтаксис языка должен способствовать легкому и прозрачному отображению в программах алгоритмических структур предметной области. Любой из типовых алгоритмов (последовательный, разветвляющийся, циклический, параллельный) имеет естественную структуру, которая должна поддерживаться программными операторами реализующего языка. Язык должен предоставлять структуры данных, операции и структуры управления, адекватные решаемой задаче. Естественность — это одна из важнейших причин популярности того или иного языка. Язык, ориентированный на характерные черты предметной области, может сильно упростить создание для нее программных приложений. Приведем примеры языков с очевидной направленностью на решение конкретных классов задач: Prolog (поддерживает дедуктивные рассуждения), Perl (предназначен для записи различных сценариев).

Поддержка абстракций. Абстракция является инструментом определения сложных структур данных и действий, при использовании которого гарантируется простота, а также игнорируются многие второстепенные детали. Абстракция устраняет пробел между структурами данных и операциями, характерными для решения задачи, и конкретными структурами данных и операциями, встроенными в язык программирования. Например, при написании информационной системы института требуются абстрактные структуры данных *студент*, *курс*, *профессор*, *расписание* и абстрактные операции *записать студента на курс* и *спланировать расписание для курса*. Программист должен создать программные реализации этих абстракций с использованием реального языка программирования, в котором изначально они отсутствуют. После этого программные абстракции можно использовать как новые элементы и в других частях программы, не вникая в их фактическую реализацию. Ясно, что язык должен обеспечивать такую возможность. Например, в языке C++ поддержка абстракций существенно выше, чем в языке C.

Выразительность. Выразительность языка может характеризовать две возможности. С одной стороны, она означает наличие очень мощных средств для представления структур данных и действий, описывающих большой объем вычислений с помощью очень маленькой программы (языки APL, Snobol, Icon, SETL). С другой стороны, выразительность позволяет записывать вычисления в более удобной и компактной форме. Например, в языке C запись `x++` удобнее и короче записи `x = x + 1`. Аналогично, булевы операции `and then` и `or else` в языке Ada позволяют указать сокращенное вычисление булевых выражений. В языке Pascal циклы с известным количеством повторений проще создавать с помощью оператора `for`, чем с помощью оператора `while`. Несомненно, что все эти возможности облегчают разработку программ.

Надежность

В общем случае, надежность — это способность программы выполнять требуемые функции при заданных условиях и в течение определенного периода времени. Обычно уровень надежности характеризуется степенью автоматического обнаружения ошибок, которую обеспечивают транслятор и операционная среда выполнения программы. Надежный язык позволяет выявлять большинство ошибок во время трансляции программы, а не во время ее выполнения, поскольку это минимизирует стоимость ошибок.

Опишем факторы, имеющие сильное влияние на надежность программ.

Проверка типов. Принципиальным средством достижения высокой надежности языка является система типизации данных. В ходе проверки типов анализируется совместимость типов в программе. Разные языки обеспечивают разную полноту проверки типов. Достаточно слабой считают проверку типов в языке C. Языки с динамической типизацией вообще относят эту проверку только к периоду выполнения программы. Наиболее полную проверку гарантирует языка Ada: в процессе компиляции программы проверяются типы практически всех переменных и выражений. Такой подход фактически устраняет ошибки типов при выполнении программы. В языках Pascal, Ada и Java диапазон изменения индексов является частью объявления массива и тоже подвергается проверке. Такая проверка очень важна для обеспечения надежности программы, поскольку индексы, выходящие за пределы допустимого диапазона, часто создают серьезные проблемы.

Обработка исключений. Исключением называют аварийное событие, которое обнаруживается во время выполнения программы (аппаратом исключений). В результате авария устраняется и программа продолжает работу. Подобный механизм значительно повышает надежность вычислений. Языки Ada, C++, C# и Java позволяют обрабатывать исключения, хотя во многих других языках этот механизм отсутствует.

Совмещение имен. Совмещением имен называют наличие нескольких разных имен у одной и той же ячейки памяти. Во многих языках переменным разрешается иметь по паре имен: обычное прямое имя и косвенное имя (на базе указателя). С одной стороны, совмещение имен может приводить к понижению надежности программы. С другой стороны, эта возможность повышает гибкость программирования и компенсирует недочеты жестких схем типизации.

В завершение отметим, что читабельность и легкость создания прямо влияют на надежность программы. Чем выше значения этих показателей, тем надежнее будет программа.

Стоимость

Суммарная стоимость языка программирования складывается из нескольких составляющих.

Стоимость выполнения программы. Она во многом зависит от структуры языка. Язык, требующий многочисленных проверок типов во время выполнения программы, будет препятствовать быстрой работе программы. Взгляды на этот фактор стоимости претерпели существенную эволюцию. В середине прошлого века фактор был решающим в силу высокой стоимости аппаратных средств компьютера и их низкой производительности. Большое значение придавалось применению оптимизирующих компиляторов, эффективному распределению регистров и механизмам эффективного выполнения программ. Сейчас считают, что стоимость (и скорость) выполнения программы существенна лишь для программного обеспечения систем реального времени. Программы реального времени должны обеспечивать быстрые вычисления управляющих воздействий на разнообразные управляемые объекты. Поскольку необходимо гарантировать определенное время реакции, следует избегать языковых конструкций, ведущих к непредсказуемым издержкам времени выполнения программы (например, при сборке мусора в схеме динамического распределения памяти). Для обычных приложений все снижающаяся стоимость аппаратуры и все возрастающая стоимость разработки программ позволяют считать, что скорость выполнения программ уже не столь критична.

Стоимость трансляции программы. Размер этой стоимости зависит от возможностей используемого компилятора. Чем совершеннее методы оптимизации, тем дороже стоит трансляция. В итоге создается эффективный код: резко сокращается размер программы и/или возрастает скорость ее работы.

Стоимость создания, тестирования и использования программы. Этот фактор стоимости удобно проиллюстрировать на примере среды для языка Smalltalk. Данная среда состоит из окон, меню, механизма ввода данных с помощью мыши и набора средств, позволяющих свободно оперировать со Smalltalk-программой. Здесь программное решение может быть спроектировано, закодировано, протестировано, изменено и использовано с минимальными затратами времени и сил программиста. По современным представлениям, наличие в языке развитых конструкций и структур является лишь одним аргументом, влияющим на широту его использования. Наличие же подходящей среды программирования существенно усиливает применимость слабого языка. Прежде всего, в среде программирования должна присутствовать надежная, эффективная и хорошо документированная реализация языка программирования. Специализированные текстовые редакторы, средства моделирования и управления конфигурацией, а также утилиты тестирования, отражающие особенности как самого языка, так и порядка его использования, — это мощные ускорители всех этапов разработки программ. В итоге минимизируются время и затраты, требуемые программисту на решение какой-либо задачи.

Стоимость сопровождения программы. Многочисленные исследования показали, что значительную часть стоимости используемой программы составляет не стоимость разработки, а стоимость сопровождения программы. Сопровождение — это процесс изменения программы после ее поставки заказчику. Сопровождение включает в себя:

- исправление ошибок (17% времени и стоимости);
- изменения, связанные с обновлением операционного окружения (18% времени и стоимости);
- усовершенствование и расширение функций программы (65% времени и стоимости).

В [87] утверждается, что соотношение между стоимостью сопровождения и стоимостью начальной разработки может быть разным в зависимости от предметной области, где эксплуатируется программа. Для прикладных программ, работающих в деловой сфере, стоимость затрат на сопровождение в основном сравнима со стоимостью разработки. Для программного обеспечения встроенных систем реального времени затраты на сопровождение могут в четыре раза превышать стоимость самой разработки. Высокие требования в отношении производительности и надежности таких программ предполагают их жесткую структуру, которая труднее поддается модификации.

Связывая сопровождение программ с характеристиками языка программирования, следует выделить, прежде всего, зависимость от читабельности. Обычно сопровождение выполняется не авторами программы, а другими лицами. В силу этого, плохая читабельность может крайне усложнить задачу усовершенствования и расширения функций программы.

Способы построения критериев эффективности

Возможны следующие способы построения критериев из частных показателей.

Выделение главного показателя. Из совокупности частных показателей $A_1, A_2, ..., A_n$ выделяется один, например A_1, который принимается за главный. На остальные показатели накладываются ограничения:

$$A_i \le A_{i\text{доп}} \ (i = 2, 3, ..., n),$$

где $A_{i\text{доп}}$ — допустимое значение i-го показателя. Например, если в качестве A_1 выбирается легкость создания программ W (*Writability*), а на показатели надежности P и стоимости S накладываются ограничения, то критерий эффективности ЯП принимает вид:

$$W \to \max, P \le P_{\text{доп}}, S \le S_{\text{доп}}.$$

Способ последовательных уступок. Все частные показатели нумеруются в порядке их важности: наиболее существенным считается показатель A_1, а наименее важным — A_n. Находится минимальное значение показателя A_1 — $\min A_1$ (если нужно найти максимум, то достаточно изменить знак показателя). Затем делается «уступка» первому показателю ΔA_1, и получается ограничение $\min A_1 + \Delta A_1$.

На втором шаге отыскивается $\min A_2$ при ограничении $A_1 \le \min A_1 + \Delta A_1$. После этого выбирается «уступка» для A_2: $\min A_2 + \Delta A_2$. На третьем шаге отыскивается $\min A_3$ при ограничениях $A_1 \le \min A_1 + \Delta A_1$; $A_2 \le \min A_2 + \Delta A_2$ и т. д. На последнем шаге ищут $\min A_n$ при ограничениях

$$A_1 \le \min A_1 + \Delta A_1;$$

$$A_2 \le \min A_2 + \Delta A_2;$$

$$\ldots$$

$$A_{n-1} \le \min A_{n-1} + \Delta A_{n-1}.$$

Полученный на этом шаге вариант языка программирования и значения его показателей $A_1, A_2, ..., A_n$ считаются окончательными. Недостатком данного способа (критерия) является неоднозначность выбора ΔA_i.

Отношение частных показателей. В этом случае критерий эффективности получают в виде

$$K_1 = \frac{A_1, A_2, ..., A_n}{B_1, B_2, ..., B_m} \to \max, \tag{1.1}$$

или в виде

$$K_2 = \frac{B_1, B_2, ..., B_m}{A_1, A_2, ..., A_n} \to \min, \tag{1.2}$$

где A_i ($i = 1, 2, ..., n$) — частные показатели, для которых желательно увеличение численных значений, а B_i ($i = 1, 2, ..., m$) — частные показатели, численные значения которых нужно уменьшить. В частном случае критерий может быть представлен в виде

$$K_3 = \frac{B_1}{A_1} \to \min. \tag{1.3}$$

Возможной формой выражения (1.3) является критерий цены создания программы

$$K_4 = \frac{S}{W} \to \min, \tag{1.4}$$

где S — стоимость, W — легкость создания программы ЯП. Формула критерия K_4 характеризует затраты стоимости, приходящиеся на единицу легкости создания программы.

Аддитивная форма. Критерий эффективности имеет вид

$$K_5 = \sum_{i=1}^{n} \alpha_i A_i \to \max, \tag{1.5}$$

где $\alpha_1, \alpha_2, ..., \alpha_n$ — положительные и отрицательные весовые коэффициенты частных показателей. Положительные коэффициенты ставятся при тех показателях, которые желательно максимизировать, а отрицательные — при тех, которые желательно минимизировать.

Весовые коэффициенты могут быть определены методом экспертных оценок. Обычно они удовлетворяют условиям

$$0 \le \alpha_i < 1, \ \sum_{i=1}^{n} \alpha_i = 1. \tag{1.6}$$

Основной недостаток критерия заключается в возможности взаимной компенсации частных показателей.

Мультипликативная форма. Критерий эффективности имеет вид

$$K_6 = \prod_{i=1}^{n} A_i^{\alpha_i} \to \max, \tag{1.7}$$

где, в частном случае, коэффициенты α_i полагают равными единице.

От мультипликативной формы можно перейти к аддитивной, используя выражение:

$$\lg K_6 = \sum_{i=1}^{n} \alpha_i \lg A_i. \tag{1.8}$$

Критерий K_6 имеет тот же недостаток, что и критерий K_5.

Максиминная форма. Критерий эффективности описывается выражением:

$$K_6 = \prod_{i=1}^{n} A_i^{\alpha_i} \to \max, \tag{1.9}$$

Здесь реализована идея равномерного повышения уровня всех показателей за счет максимального «подтягивания» наихудшего из показателей (имеющего минимальное значение).

У максиминного критерия нет того недостатка, который присущ мультипликативному и аддитивному критериям.

Нормализация частных показателей

Частные показатели качества обычно имеют различную физическую природу и различные масштабы измерений, из-за чего их простое сравнение становится практически невозможным. Поэтому появляется задача приведения частных показателей к единому масштабу измерений, то есть их нормализация.

Рассмотрим отдельные способы нормализации.

Использование отклонения частного показателя от максимального.

$$\Delta A_i = A_{\max_i} - A_i. \tag{1.10}$$

В данном случае переходят к отклонениям показателей, однако способ не устраняет различия масштабов отклонений.

Использование безразмерной величины $\overline{A_i}$.

$$\overline{A_i} = \frac{A_{\max_i} - A_i}{A_{\max_i}}, \tag{1.11}$$

$$\overline{A_i} = \frac{A_i}{A_{\max_i}}. \qquad (1.12)$$

Формула (1.11) применяется тогда, когда уменьшение A_i приводит к увеличению (улучшению) значения аддитивной формулы критерия. Выражение (1.12) используется, когда к увеличению значения аддитивной формулы критерия приводит увеличение A_i.

Учет приоритета частных показателей

Необходимость в учете приоритетов возникает в случае, когда частные показатели имеют различную степень важности.

Приоритет частных показателей задается с помощью ряда приоритета I, вектора приоритета $(b_1, ..., b_q, ..., b_n)$ и вектора весовых коэффициентов $(\alpha_1, \alpha_2, ..., \alpha_n)$.

Ряд приоритета представляет собой упорядоченное множество индексов частных показателей $I = (1, 2, ..., n)$. Он отражает чисто качественные отношения доминирования показателей, а именно отношения следующего типа: показатель A_1 важнее показателя A_2, а показатель A_2 важнее показателя A_3 и т. д.

Элемент b_q вектора приоритета показывает, во сколько раз показатель A_q важнее показателя A_{q+1} (здесь A_q — показатель, которому отведен номер q в ряду приоритета). Если A_q и A_{q+1} имеют одинаковый ранг, то $b_q = 1$. Для удобства принимают $b_n = 1$.

Компоненты векторов приоритета и весовых коэффициентов связаны между собой следующим отношением:

$$b_q = \frac{\alpha_q}{\alpha_{q+1}}.$$

Зависимость, позволяющая по известным значениям b_i определить величину α_q, имеет вид:

$$\alpha_q = \frac{\prod_{i=q}^{n} b_i}{\sum_{q=1}^{n}\prod_{i=q}^{n} b_i}.$$

Знание весовых коэффициентов позволяет учесть приоритет частных показателей.

Заключительные замечания

Когда собираются сторонники различных языков, обсуждение обычно принимает острую форму в стиле спора известных литературных героев Шуры Балаганова и Михаила Паниковского, страсти кипят, в выражениях не стесняются. Ведь известно, что профессионалы-программисты (в подавляющем большинстве) являются

однолюбами-консерваторами, строящими храм для любимого языка и нещадно критикующими все остальные языки. Может быть, по этой причине большинство частных показателей качества ЯП, в частности читабельность, легкость создания и надежность, в настоящее время не являются ни строго определенными, ни точно измеримыми. Тем не менее предложенные схемы построения критериев вполне реальны и позволяют перевести дискуссию из эмоциональной в количественную плоскость. Нужно лишь заручиться поддержкой экспертов.

Конечно, здесь есть определенные трудности. Отдельные показатели имеют разный вес при рассмотрении языка с различных точек зрения. Разработчиков трансляторов в первую очередь интересует сложность реализации конструкций и структур данных языка. Программистов прежде всего волнует легкость создания программных приложений, а инженеров сопровождения — читабельность эксплуатируемых программ. Авторы языков программирования придают особое значение элегантности и возможности широкого использования языка. Мало того, некоторые характеристики, формирующие показатели, входят в противоречия друг с другом. Но все эти препятствия преодолимы при наличии доброй воли.

Глава 3
Виды языков программирования

Гибридные языки разметки/программирования

Гибридный язык разметки/программирования является языком разметки, в котором некоторые элементы могут задавать действия по программированию, такие как управление потоком действий и вычисления.

Язык XSLT

Расширяемый язык разметки (XML) является языком мета-разметки. Такой язык используется для определения языков разметки. Производные от XML языки разметки используются для определения документов данных, называемых XML-документами. Хотя XML-документы читабельны для человека, они обрабатываются с помощью компьютеров. Иногда эта обработка состоит только из преобразований в формы, которые могут эффективно отображаться или распечатываться. Во многих случаях это преобразования в формы HTML, которые могут отображаться веб-браузером. В других случаях данные в документе обрабатываются так же, как и другие формы файлов с данными.

Преобразования XML-документов в XML- и HTML-документы определяются другим языком разметки, расширяемым языком стилей для преобразований XSLT. XSLT может указывать операции подобные программированию. Таким образом, XSLT является гибридным языком разметки/программирования [7, 35].

Спецификация XSLT гласит, что это язык для преобразования одних XML-документов в другие XML-документы. Именно такой и была изначальная идея XSLT. Тем не менее по мере разработки язык перерос ее и теперь, как утверждает редактор новой версии языка Майкл Кей (Michael Kay), XSLT — это язык для преобразования структуры документов.

В преобразовании участвуют три документа:

- входной документ, который подвергается преобразованию (XML-документ);
- преобразователь (документ XSLT, который описывает само преобразование);
- выходной документ, который является результатом преобразования.

Преобразователь задает правила трансформации входного документа в выходной.

Выполнением преобразований занимаются специальные программы — процессоры XSLT. Процессор получает входной документ и преобразователь и, применяя правила преобразования, генерирует выходной документ.

По сути процессор оперирует не самими документами, а древовидными моделями их структур — именно структурными преобразованиями занимается XSLT, оставляя за кадром синтаксис, который эти структуры выражает.

Несмотря на то что для XSLT совершенно неважно, в каком виде находятся документы изначально (главное — чтобы была структура, которую можно преобразовать), абсолютное большинство процессоров XSLT может работать с документами, которые физически записаны в файлах. В этом случае процесс обработки делится на три этапа.

- Этап разбора (парсинга) документа. Здесь XSLT-процессор разбирает входящий документ и документ-преобразователь, создавая для них древовидные структуры данных.
- Этап преобразования. К дереву входного документа применяются правила, описанные в преобразователе. В итоге процессор создает дерево выходного документа.
- Этап сериализации. Для созданного дерева генерируется сущность в определенной физической форме.

Процессоры выполняют каждый из трех этапов (получают входные документы и выдают результат их трансформации), но областью применения XSLT является лишь второй этап — этап преобразования.

Сериализация ориентирована на создание физической интерпретации конечного дерева. Текущая версия языка поддерживает три способа сериализации: XML, HTML и текст. Каждый из этих способов учитывает синтаксис целевого физического формата и позволяет получить документ требуемого вида.

Преобразователь может указывать способ сериализации, однако непосредственный контроль над синтаксисом физического документа сильно ограничен. Преобразования совершенно сознательно отделены от синтаксической обработки, ведь их задачей являются структурные трансформации, а не работа с физическим синтаксисом. Благодаря такому разделению многие процессоры производят разбор и сериализацию с помощью внешних приложений, что способствует универсальности XSLT. Ведь для каждого этапа преобразования можно применить наиболее подходящий инструмент.

С точки зрения языка XSLT задачу преобразования можно разделить на три подзадачи:

- обращение к преобразуемому объекту;
- создание результата преобразования;
- связывание первых двух действий для осуществления преобразования.

Первая подзадача обеспечивает получение информации о входном документе, его структуре. Здесь применяется язык XPath — язык путей в XML-документах.

Вторая и третья подзадачи решаются программой на XSLT, которую мы назвали преобразователем. Синтаксически преобразователь — это XML-документ.

В отличие от традиционных языков программирования, преобразование в XSLT не описывается последовательностью действий, выполняемых для достижения результата. Преобразование указывается в виде набора шаблонных правил, каждое из которых определяет процедуру обработки определенной части документа. Иначе говоря, преобразователь в XSLT объявляет, декларирует правила преобразования —

правила, применяя которые к входному документу, XSLT-процессор генерирует выходной документ.

Рассмотрим пример XSLT-преобразования. Поставим задачу преобразования входного документа

```
<msg>Hello, world!</msg>
```

в выходной документ вида:

```
<message>Hello, world!</message>
```

Суть преобразования: если в документе встретится элемент `msg`, создать в выходном документе элемент `message` и включить в него содержимое элемента `msg`.

Эта задача решается следующей программой на XSLT:

```
<xsl:stylesheet
    version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="msg">
        <message>
            <xsl:value-of select="."/>
        </message>
    </xsl:template>
</xsl:stylesheet>
```

Корневым элементом программы является элемент `xsl:stylesheet`, обозначающий преобразование. Атрибут `version` указывает на использованную версию языка XSLT; здесь же объявляется пространство имен с префиксом `xsl`, которому соответствует URL `"http://www.W3.org/1999/XSL/Transform"`. Все элементы преобразования из этого пространства имен будут восприняты процессором как принадлежащие языку XSLT.

Элемент `xsl:stylesheet` содержит единственный дочерний элемент `xsl:template`, задающий правило преобразования. Атрибут `match` указывает, что данное правило должно обрабатывать элемент `msg`. Содержимое `xsl:template` является телом шаблона. Тело выполняется тогда, когда сам шаблон применяется к некоторой части документа. В данном случае тело шаблона будет выполнено, когда само правило будет применяться к элементу `msg`.

В теле шаблона записан элемент `message`. Это простой литеральный элемент результата. Он не принадлежит пространству имен XSLT и поэтому при обработке будет просто скопирован в результирующий документ. Содержимое этого элемента будет обработано и включено в сгенерированную копию.

Содержимым элемента `message` является элемент `xsl:value-of`, который, в отличие от `message`, принадлежит XSLT. Элемент `xsl:value-of` вычисляет XPath-выражение, заданное в его атрибуте `select`, и возвращает результат этого вычисления. XPath-выражение, ".", указанное в `select`, возвращает ту часть узла дерева, которая обрабатывается в данный момент, иначе говоря элемент `msg`.

В отличие от синтаксиса языка XPath (представлен лаконичным выражением "."), синтаксис языка XSLT полностью воспроизводит синтаксис XML.

В описанном преобразовании участвовала и третья синтаксическая конструкция — паттерн XSLT. Паттерн `msg`, записанный в атрибуте `match` элемента `xsl:template`, задает ту часть XML-документа, которая должна быть обработана этим правилом. Синтаксически паттерны считаются XPath-выражениями (но не наоборот), однако смысл у них другой. XPath-выражения вычисляются и возвращают

результат, паттерны же просто устанавливают соответствие некоторому образцу. В нашем преобразовании паттерн `msg` указывает, что шаблон должен обрабатывать только элементы `msg` и никакие другие.

Каждое из шаблонных правил может вызывать другие шаблонные правила — в этом случае результат выполнения вызванных шаблонов включается в результат выполнения шаблона, который их вызывал. Для демонстрации этого приема модифицируем программу так, чтобы результат возвращался в виде HTML-документа:

```
<xsl:stylesheet
    version="1. 0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="/">
        <html>
            <head>
            <title>Message</title>
            </head>
            <body>
                <xsl:apply-templates select="msg"/>
            </body>
        </html>
    </xsl:template>
    <xsl:template match="msg">
            <b>
                <xsl:value-of select="."/>
            </b>
    </xsl:template>
</xsl:stylesheet>
```

В этот преобразователь добавлено еще одно шаблонное правило:

```
<xsl:template match="/">
    <html>
        <head>
            <title>Message</title>
        </head>
        <body>
            <xsl:apply-templates select="msg"/>
        </body>
    </html>
</xsl:template>
```

Данное правило задает обработку корневого узла — в атрибуте `match` указан паттерн `"/"`, что соответствует корню документа. Шаблон создает элементы `html`, `head`, `title`, `body` и в последний включает результат применения шаблонов к элементу `msg`.

Выходной документ, полученный в результате применения этого преобразователя, примет вид:

```
<html>
    <head>
        <title>Message</title>
    </head>
    <body>
        <b>Hello, world!</b>
    </body>
</html>
```

Отметим, что процессор скопировал все элементы, не принадлежащие XSLT, не изменяя их, а элемент `xsl:apply-templates` выполнил подстановку, применив шаблон к элементу `msg` и включив в `body` результат (выделен полужирным шрифтом).

Возможность вызова одних правил из других, а также наличие в XSLT таких управляющих конструкций, как `xsl:if`, `xsl:choose` и `xsl:for-each`, обеспечивает реализацию очень сложной логики преобразования. XSLT позволяет определить простые правила обработки отдельных частей преобразования, связав эти правила логикой взаимных вызовов и управляющих конструкций.

Таким образом, с каждым шаблоном в документе XSLT ассоциированы инструкции преобразования, которые определяют, как сопоставленные данные должны быть преобразованы перед их размещением в выходном документе. Можно сказать, что шаблоны (и ассоциированная с ними обработка) выступают в качестве подпрограмм, которые «выполняются», когда процессор XSLT находит сопоставление с шаблоном в данных документа XML.

Язык JSP

Основной частью стандартной библиотеки тегов серверных страниц Java (JSTL) является еще один гибридный язык разметки/программирования, хотя его форма и цели отличны от XSLT. Прежде чем обсуждать JSTL, необходимо ввести понятия сервлетов и Java Server Pages (JSP). Сервлет является экземпляром класса Java, который находится и выполняется на веб-серверной стороне. Выполнение сервлета, запрашиваемое документом разметки, отображается веб-браузером. Выводы сервлета возвращаются запрашивающему браузеру в форме документа HTML. Программа, которая выполняется веб-сервером, называется контейнером сервлетов и контролирует выполнение сервлетов. Сервлеты обычно используются для обработки форм и доступа к базе данных.

JSP представляет собой набор технологий, предназначенных для поддержки динамических веб-документов, а также обеспечивающих иные потребности в обработке веб-документов [59]. Если документ JSP, который часто является комбинацией кодов HTML и Java, запрашивается браузером, процессорная программа JSP, которая находится в веб-серверной системе, преобразует документ в сервлет. Встроенный в документ Java-код копируется в сервлет. Простой код HTML копируется в операторы печати Java, что рассматривается как вывод. Сервлет производится процессором JSP, который запускается контейнером сервлетов. JSTL определяет набор элементов действия для XML, которые управляют обработкой документа JSP на веб-сервере. Эти элементы имеют тот же вид, что и другие элементы HTML и XML. Наиболее часто используется такое управляющее действие JSTL, как if, определяющее в качестве атрибута логическое выражение. Содержанием элемента if (текстом между открывающим тегом `<if>` и закрывающим тегом `</if>`) является код HTML, который будет включен в выходной документ только в том случае, если логическое выражение будет истинным. Элемент if подобен препроцессорной команде #if для языков C/C++. Контейнер JSP обрабатывает JSTL-части JSP-документа подобно тому, как препроцессор C/C++ обрабатывает программы C и C++. Команды

препроцессора представляют собой инструкции для препроцессора, указывающие, как из входного файла строится выходной файл. Кроме того, элементы управления действиями JSTL являются инструкциями для процессора JSP, определяющими порядок создания XML-файла выходных данных на основе входного XML-файла.

Чаще всего элемент if используется для проверки формы данных, представляемых пользователем браузера. Форма данных доступна процессору JSP и может быть проверена элементом if на предмет корректности данных. Если проверка не проходит, элемент if может вставить в выходной документ сообщение для пользователя об ошибке.

Для управления множественным выбором в JSTL могут использоваться choose, when и другие элементы. JSTL также включает в себя элемент forEach, который выполняет итерации по коллекциям, формируемым клиентами. Для управления итерациями элемент forEach может содержать атрибуты begin, end, а также step.

Глава 16

Объектно-ориентированное и аспектно-ориентированное программирование

ООП на языке Ада

Расширяемые типы

Основная цель расширяемых типов — обеспечить повторное использование существующих программных элементов (без необходимости перекомпиляции и перепроверки). Они позволяют объявить новый тип, который уточняет существующий родительский тип наследованием, изменением или добавлением как существующих компонентов, так и операций родительского типа. В качестве расширяемых типов используются *теговые типы* (разновидность комбинированного типа).

Рассмотрим построение иерархии геометрических объектов. На вершине иерархии находится точка, имеющая два атрибута (координаты *X* и *Y*):

```
type Точка is tagged
    record
        x_Коорд : Float;
        y_Коорд : Float;
    end record;
```

Другие типы объектов можно произвести (прямо или косвенно) от этого типа.

Например, можно ввести новый тип, наследник точки:

```
type Окружность is new Точка with  -- новый теговый тип;
    record
        радиус : Float;
    end record;
```

Данный тип имеет три атрибута: два атрибута (координаты `x` и `y`) унаследованы от типа `Точка`, а третий атрибут (`радиус`) нами добавлен. Дочерний тип `Окружность` наследует все операции родительского типа `Точка`, причем некоторые операции могут быть переопределены. Кроме того, для дочернего типа могут быть введены новые операции.

Классы

В языке Ада нет синтаксической конструкции «класс». Для моделирования конструкции «класс» здесь применяют конструкцию «пакет». Пакет — это элегантный способ инкапсуляции программного кода и данных, взаимодействующих друг

с другом, в единый модуль. Пакет может экспортировать один или несколько пользовательских типов вместе с их примитивными операциями. Примитивная операция определяется для типа (в части аргументов и результата) и объявляется вместе с типом в одной спецификации пакета.

Правила моделирования класса

1. Имени класса придается префикс `Класс_`.
2. Пакет имеет единственный теговый приватный тип, который получает имя класса и используется для объявления экземпляров класса. Вследствие этого все экземпляры класса будут разделять одинаковую структуру и поведение.
3. Процедуры и функции используются для определения поведения класса. Первым параметром для процедуры или функции является экземпляр класса.
4. Реализация приватного типа определяется как комбинированный расширяемый тип. (Это дает возможность расширять количество полей данных в классах-наследниках, а также количество операций.) Компоненты комбинированного типа определяют структуру данных класса.

Например, класс `Счет` задается в виде следующего пакета:

```
package Класс_Счет is
    type Счет is tagged private; -- осн. тип класса
                -- используется для объявления экземпляров
    subtype Деньги is Float; -- вспомог. типы
    subtype РДеньги is Float range 0.0.. Float'LAST;
    -- используются в сообщениях,  посылаемых в экз.
    -- объявления методов класса:
    procedure заявить ( the : in Счет );
    procedure положить ( the : in out Счет;
                    Сумма : in РДеньги );
    procedure снять ( the : in out Счет; сумма : in РДеньги;
                    принимать : out РДеньги );
    function баланс ( the : Счет ) return Деньги;
    private    -- скрытое представление класса
        type Счет is tagged record
            остаток : Деньги := 0.00; -- сумма на счету
        end record;
end  Класс_Счет;
-- Тело пакета-класса включает реализацию его методов:
with Ada.Text_IO, Ada.Float_Text_IO;
use Ada.Text_IO, Ada.Float_Text_IO;
package body Класс_Счет is
    procedure заявить ( the : in Счет ) is
    begin
        Put ("Тек. состояние : Сумма на вкладе $");
        Put ( the.остаток, Aft => 2, Exp => 0);
        New_Line ( 2 );
    end заявить;
    procedure положить ( the : in out Счет;
                    сумма : in РДеньги ) is
    begin
        the.остаток := the.остаток + сумма;
    end положить;
    procedure снять ( the : in out Счет; сумма : in РДеньги;
                    принимать : out РДеньги ) is
    begin
        if the.остаток >= сумма then
```

```
            the.остаток := the.остаток - сумма;
            принимать := сумма;
        else принимать := 0.00;
        end if;
    end снять;
    function баланс ( the : Счет ) return Деньги is
    begin
        return  the.остаток;
    end баланс;
end Класс_Счет;
```

Видим, что для доступа к полю данных `остаток`, содержащемуся в экземпляре `Счета`, используется составное имя `the.остаток`. Например, в функции `баланс` результат возвращается с помощью оператора

```
return  the.остаток;
```

Составное имя используется для доступа к атрибуту экземпляра комбинированного типа. В этом случае экземпляром типа запись является объект `the`, а атрибутом объекта — поле `остаток`.

К объектам класса (типа) `Счет` применимы следующие операции:

- обработка (предвыполнение) при создании объекта;
- присвоение значения экземпляра другому экземпляру такого же типа;
- сравнение экземпляров класса на эквивалентность и неэквивалентность;
- заданные методы (чтение и изменение внутреннего состояния возможно только в результате действия методов класса).

Если мы объявили объект класса `Счет`

```
мой_Счет: Класс_Счет.Счет;
```

то можно использовать две формы обращения к его операциям:

```
1.    положить (мой_Счет, деньги);
2.    мой_Счет.положить (деньги);
```

В первой форме в качестве первого аргумента вызываемой операции указывается имя объекта (которому принадлежит эта операция). Во второй форме сообщение начинается с указания имени адресуемого объекта, поэтому из списка аргументов операции имя объекта изымается. Вторая форма соответствует традиционному стилю записи сообщений, принятому в объектно-ориентированном программировании.

Абстрактные классы и интерфейсы

Абстрактный класс — это описание (спецификация) будущих возможностей, которые будут обеспечены в дальнейшем производными классами. Абстрактный класс не имеет тела, то есть реализации.

Основным элементом абстрактного класса является абстрактный расширяемый тип. Этот тип задает только имя. Он не имеет полей, то есть экземплярных атрибутов, и поэтому записывается в публичной части спецификации в виде:

```
type Абстрактный_Счет is abstract tagged null record;
```

Примитивными операциями абстрактного типа (методами абстрактного класса) являются абстрактные процедуры и функции. Абстрактные процедуры и функции

также не имеют тел, их назначение — объявить спецификации будущих конкретных методов.

В качестве примера приведем абстрактный класс банковского счета:

```
package Класс_Абстрактный_Счет is
    type Абстрактный_Счет is abstract tagged null record;
    subtype Деньги is Float;  -- вспомогат. подтипы
    subtype Рденьги is Float range 0.0 .. Float'Last;
    procedure заявить ( the : in Абстрактный_Счет ) is abstract;
    procedure положить ( the : in out Абстрактный_Счет; сумма : in Рденьги ) is
abstract;
    procedure снять ( the : in out Абстрактный_Счет; сумма : in РДеньги;
        принимать : out РДеньги ) is abstract;
    function баланс ( the : in Абстрактный_Счет ) return Деньги is abstract;
end Класс_Абстрактный_Счет;
```

Сам по себе абстрактный класс и абстрактный тип нельзя использовать для объявления объектов. Однако абстрактный класс можно применить для производства (путем наследования) конкретных типов банковского счета, например:

```
with Класс_Абстрактный_Счет;
use Класс_Абстрактный_Счет;
package Класс_Счет is
    type Счет is new Абстрактный_Счет with private;
    subtype Деньги is Класс_Абстрактный_Счет.Деньги;
    subtype Рденьги is Класс_Абстрактный_Счет.РДеньги;
    procedure заявить ( the : in Счет );
    procedure положить ( the : in out Счет; сумма : in Рденьги );
    procedure снять ( the : in out Счет; сумма : in РДеньги; принимать : out
                                                                  РДеньги );
    function баланс ( the : in Счет ) return Деньги;
private
    type Счет is new Абстрактный_Счет with
        record
            остаток : Деньги := 0.00;  -- сумма на счету
        end record;
end Класс_Счет;
```

ПРИМЕЧАНИЕ

Подтипы Деньги и РДеньги объявлены для того, чтобы сделать их видимыми для клиентов класса. Если этого не сделать, то клиенты должны будут ссылаться на Класс_Абстрактный_Счет (с помощью with и use). Тело (реализация) этого класса аналогично телу класса Счет из предыдущего подраздела.

В свою очередь, конкретный класс `Счет` может использоваться как родительский класс в производстве нового класса. Например, может быть произведен счет, по которому операцию `снять` разрешено выполнять только три раза в неделю:

```
with Класс_Счет; use Класс_Счет;
package Класс_Огр_Счет is
    type Огр_Счет is new Счет with private;
    procedure снять ( the : in out Огр_Счет; сумма : in РДеньги;
                принимать : out РДеньги );
    procedure сброс ( the : in out Огр_Счет);
private
```

```
    снять_За_Неделю : Natural := 3;
    type Огр_Счет is new Счет with
        record
            снятия : Natural := снять_За_Неделю;
        end record;
end Класс_Огр_Счет;
```

ПРИМЕЧАНИЕ

В данном классе переопределяется родительский метод снять. Новый метод `сброс` применяется для сброса количества снятий, которые могли быть выполнены на текущей неделе.

Реализация этого класса записывается в виде:

```
package body Класс_Огр_Счет is
    procedure снять ( the : in out Огр_Счет; сумма : in РДеньги;
                      принимать : out РДеньги ) is
    begin
        if the.снятия > 0 then  -- проверка ограничения
            the.снятия := the.снятия - 1;
            снять (Счет(the), сумма, принимать);
                -- вызов родительского метода
        else
            принимать := 0.00;    -- извините
        end if;
    end снять;
    procedure сброс ( the : in out Огр_Счет) is
    begin
        the.снятия := снять_За_Неделю;
    end сброс;
end Класс_Огр_Счет;
```

Использование класса `Огр_Счет` проиллюстрируем следующей программой:

```
with Класс_Счет, Класс_Огр_Счет;
use Класс_Счет, Класс_Огр_Счет;
procedure Main is
    петр : Огр_Счет;
    получить : Деньги;
begin
    петр.положить (700.00);
    петр.заявить;   -- количество денег на счету
    петр.снять (150.00, получить); -- снятие денег
    петр.снять (70.00, получить); -- снятие денег
    петр.снять (20.00, получить); -- снятие денег
    петр.снять (15.00, получить);
            -- отказ - превышен лимит
    петр.заявить; -- количество денег на счету
end Main;
```

ПРИМЕЧАНИЕ

Спецификаторы with и use для Класс_Счет обеспечивают прямую видимость подтипа Деньги. Вспомним, что подтип Деньги объявлен в классе Счет и невидим в классе Огр_Счет. Можно отказаться от спецификатора use Класс_Счет, если в программе Main для переменной `получить` применить объявление получить:Класс_Счет.Деньги.

Если абстрактный класс вообще не имеет ни конкретных операций, ни компонентов данных, он может быть объявлен как интерфейс. Интерфейс — это предельно ограниченный вариант абстрактного типа. В нем нет объявлений полей данных, а операции объявлены или абстрактными, или нулевыми. Вместе с тем интерфейс — важнейший механизм обеспечения множественного наследования.

Объявление интерфейса записывается в публичной части спецификации в виде:

```
type My_Abstract is Interface;
```

Приведем ряд примеров. Множественное наследование можно обеспечить, объявив родителями конкретный тип `My_Parent` и два интерфейса `Int1` и `Int2`:

```
type My_Child is new My_Parent and Int1 and Int2 with …
```

Можно составить новый интерфейс из двух имеющихся:

```
type Int3 is Interface and Int1 and Int2;
```

Надклассовые типы

Для обеспечения полиморфизма в языке Ада введено понятие надклассового типа.

Для каждого тегового типа `T` автоматически объявляется надклассовый тип `T'Class`. Значения `T'Class` являются объединением значений самого типа `T` и всех производных от него типов. Сам тип `T` считается корневым типом для дерева наследования, которое представляет `T'Class`. Значение любого дочернего от `T` типа может быть неявно преобразовано к надклассовому типу `T'Class`.

Например, если наследниками тегового типа `Комната` являются типы `Кладовая` и `Офис`, а тип `Офис`, в свою очередь, имеет наследника `Канцелярия`, то все эти типы образуют надклассовый тип `Комната'Class`.

Значение любого типа помещения может быть неявно преобразовано в значение типа `Комната'Class`. В свою очередь, надклассовый тип `Офис'Class` включает в себя конкретные типы `Офис` и `Канцелярия`, а надклассовый тип `Кладовая'Class` — только конкретный тип `Кладовая`.

Каждый объект надклассового типа имеет свой тег, который задает его конкретный тип во множестве других типов дерева наследования. Единственность тега объясняется его происхождением: тег — это элемент конкретного тегового типа. Тег объекта может быть явно выделен с помощью атрибута `'Tag`. Например, если `b304` и `b306` — это объекты типа `Комната'Class`, то можно записать:

```
if b304'Tag = b306'Tag then
    Put ("Аудитории имеют один и тот же тип помещения");
    New_Line;
end if;
```

Надклассовый тип `T'Class` считается неограниченным типом, так как заранее нельзя предугадать размер объекта этого типа. Поэтому поступают следующим образом:

- объявляют объект надклассового типа;
- инициализируют объект, вследствие чего он ограничивается с помощью тега.

Говорят, что надклассовые типы являются полиморфными. Когда сообщение (например, `Описать`) посылается в объект надклассового типа `Комната'Class`, компилятор в период компиляции не знает, какой метод будет выполняться. Решение об исполняемом методе принимается в период выполнения (путем анализа тега

объекта). В терминологии языка Ада динамическое связывание между объектом и посылаемым сообщением называется *диспетчированием времени выполнения*.

Наследование от родового класса

В главе 15 обсуждался родовой АТД для стека и его родовое дочернее расширение. Для превращения их в классы достаточно заменить использованный там обычный комбинированный тип на расширяемый теговый тип. В результате такой замены получим:

```
generic
    type T is private;  -- любой нелимитир. тип
    Size : Positive := 4;
                -- указан тип и значение по умолчанию
package Class_Stack is
    type Stack is tagged private;
    Stack_Error : exception;
    procedure Reset ( The : in out Stack );
    procedure Push ( The : in out Stack; Item : in T );
    procedure Pop ( The : in out Stack; Item : out T );
private
    type Stack_Index is new Integer range 0 .. Size;
    subtype Stack_Range is Stack_Index
                range 1 .. Stack_Index ( Size );
    type Stack_Array is array ( Stack_Range ) of T;
    type Stack is tagged record
        Elements : Stack_Array;   -- массив элементов
        Tos : Stack_Index := 0;   -- указатель на вершину
    end record;
end Class_Stack;
```

Тело родового класса содержит реализацию его методов:

```
package body Class_Stack is
    procedure Reset ( The : in out Stack ) is
    begin
        The.Tos := 0;
                -- установить указатель  в нуль (нет элементов)
    end Reset;
    procedure Push ( The : in out Stack; Item : in T ) is
    begin
        if The.Tos /= Stack_Index ( Size ) then
                        -- проверка заполнения
            The.Tos := The.Tos + 1;  -- указатель вверх
            The.Elements (The.Tos) := Item; -- запись элемента
        else
            raise Stack_Error;    -- ошибка переполнения
        end if;
    end Push;
procedure Pop ( The : in out Stack; Item : out T ) is
    begin
        if The.Tos > 0 then
            Item := The.Elements (The.Tos ); --счит.верхн.элемен.
            The.Tos := The.Tos - 1;  -- указатель вниз
else
            raise Stack_Error;
                -- ошибка удаления из пустого стека
        end if;
```

```
    end Pop;
end Class_Stack;
generic
package Class_Stack.Additions is
    function Top ( The : in Stack ) return T;
    function Items ( The : in Stack ) return Natural;
private
end Class_Stack.Additions;
package body Class_Stack.Additions is
    function Top ( The : in Stack ) return T is
    begin
        return The.Elements ( The.Tos );
    end Top;
    function Items ( The : in Stack ) return Natural is
    begin
        return Natural ( The.Tos );
    end Items;
end Class_Stack.Additions;
```

Положим, что теперь к родовому классу `Class_Stack` и его родовой дочери `Class_Stack.Additions` нужно добавить метод.

Метод	Обязанность
Max_Depth	Вернуть максимальную глубину, которая достигнута стеком

Для решения этой задачи используем механизм наследования. Спецификацию наследника представим в виде:

```
with Class_Stack, Class_Stack.Additions;
generic
    type T is private;    -- любой нелимитирован. тип
    Size : Positive := 4; -- указан тип и значение по умолчанию
package Class_Best_Stack is
            package Class_Stack_D is new Class_Stack ( T, Size );
            package Class_Stack_D_Additions is
                    new Class_Stack_D.Additions;
        -- этот экземпляр становится независимым от родителя
            type Best_Stack is new Class_Stack_D.Stack with private;
        -- переопределяем метод
            procedure Push ( The : in out Best_Stack; Item : in T );
        -- вводим новый метод
            function Max_Depth ( The : in Best_Stack )
                        return Natural;
private
    type Best_Stack is new Class_Stack_D.Stack with
                record
                    Depth : Natural := 0;  -- новый атрибут
                end record;
end Class_Best_Stack;
```

ПРИМЕЧАНИЕ

1. Таким образом, конкретизацию базового класса Stack и его родовой дочери мы разместили в теле класса-наследника.
2. Метод Push переопределен для того, чтобы он мог фиксировать максимальную достигнутую глубину.

Реализация класса-наследника выглядит так:

```
package body Class_Best_Stack is
    procedure Push ( The : in out Best_Stack; Item : in T ) is
                Dep : Natural;
        begin
            Dep := Class_Stack_D_Additions.Items
                        (Class_Stack_D.Stack(The));
            if Dep > The.Depth then
                    The.Depth := Dep;
            end if;
            Class_Stack_D.Push (Class_Stack_D.Stack (The),Item);
        end Push;
    function Max_Depth ( The : in Best_Stack ) return Natural is
        begin
            return The.Depth;
        end Max_Depth;
end Class_Best_Stack;
```

Конкретизация класса `Class_Best_Stack` для чисел типа `Positive` выполняется по объявлению:

```
with Class_Best_Stack;
    package Class_Best_Stack_Pos is
            new Class_Best_Stack ( Positive, 15 );
```

Тестирование конкретизированного класса можно осуществить с помощью программы:

```
with Ada.Text_IO, Ada.Integer_Text_IO, Class_Best_Stack_Pos;
use Ada.Text_IO, Ada.Integer_Text_IO, Class_Best_Stack_Pos;
procedure Main is
    Numbers : Best_Stack;
    Res : Positive;
begin
    Put ("Максимальная глубина: ");
    Put_Line ( Max_Depth ( Numbers ));
    Push ( Numbers, 40 );
    Push ( Numbers, 20 );
    Put ("Максимальная глубина: ");
    Put_Line ( Max_Depth ( Numbers ));
    Push ( Numbers, 50 );
    Put ("Максимальная глубина: ");
    Put_Line ( Max_Depth ( Numbers ));
    Pop ( Numbers, Res );
    Put ("Максимальная глубина: ");
    Put_Line ( Max_Depth ( Numbers ));
end Main;
```

Данная программа выводит на экран следующие результаты:

```
Максимальная глубина: 0
Максимальная глубина: 2
Максимальная глубина: 3
Максимальная глубина: 3
```

Глава 18
Ввод-вывод и файлы

Пакеты ввода-вывода языка Ада

Ада — это язык для разработки программ реального мира, программ, которые могут быть очень большими (включают сотни тысяч операторов). При этом желательно, чтобы отдельный программный модуль имел разумно малый размер. Для обеспечения такого ограничения Ада построена на идее библиотек и пакетов. В библиотеку, составленную из пакетов, помещаются программные тексты, предназначенные для многократного использования.

Пакет подключается к программе с помощью указания (спецификатора) контекста, имеющего вид: `with < Имя_Пакета >;`

В частности, в пакетах размещены процедуры ввода-вывода для величин предопределенных типов (табл. 18.3).

Таблица 18.3. Пакеты ввода-вывода

Имя пакета	Тип вводимых-выводимых величин
Ada.Text_IO	Character, String
Ada.Integer_Text_IO	Integer
Ada.Float_Text_IO	Float

Для поддержки ввода-вывода величин других типов, определяемых пользователем, используются шаблоны (заготовки) пакетов — родовые пакеты. Родовой пакет перед использованием должен быть настроен на конкретный тип.

Родовые пакеты ввода-вывода всегда находятся внутри пакета `Ada.Text_IO` (табл. 18.4).

Таблица 18.4. Родовые пакеты ввода-вывода

Внутренние пакеты библиотеки Ada.Text_IO	
Имя родового пакета	**Нужна настройка на тип**
Integer_IO	Любой целый тип со знаком
Float_IO	Любой вещественный тип с плавающей точкой
Enumeration_IO	Любой перечисляемый тип
Fixed_IO	Любой двоичный вещественный тип с фиксированной точкой
Decimal_IO	Любой десятичный вещественный тип с фиксированной точкой
Modular_IO	Любой целый тип без знака

Для обращения к внутренним родовым пакетам используют составные имена. Например, `Ada.Text_IO.Modular_IO`, `Ada.Text_IO.Enumeration_IO`.

Процесс настройки родового пакета называют конкретизацией. В результате конкретизации образуется экземпляр родового пакета. Экземпляр помещается в библиотеку и может быть подключен к любой программе.

Конкретизация задается предложением следующего формата:

```
with <Полное имя родового пакета>;
        package <Имя пакета-экземпляра> is
            new <Имя родового пакета> ( <Имя типа настройки> );
```

Например, введем перечисляемый тип Type `Summer` is `( may, jun, jul, aug );`

Создадим экземпляр пакета для ввода-вывода величин этого типа:

```
with Ada.Text_IO.Enumeration_IO;
        package Summer_IO is new Ada.Text_IO.Enumeration_IO ( Summer );
```

Теперь пакет `Summer_IO` может использоваться в любой программе.

Процедуры ввода языка Ада

Формат вызова процедуры:

```
< ИмяПакета > . < ИмяПроцедуры > ( < ФактАрг > );
```

Перечислим (и охарактеризуем) возможные процедуры ввода.

Процедура ввода величины типа `Character.`

Оператор вызова: `Ada.Text_IO.Get ( Var );`

Описание: Переменной `Var` (типа `Character`) присваивается значение символа, введенного с клавиатуры. Пробел считается символом, нажатие на клавишу `Enter` не учитывается.

Процедура ввода величины типа `String.`

Оператор вызова: `Ada.Text_IO.Get ( Var );`

Описание: Переменная `Var` должна иметь тип `String (min..max)`, где `1<=min<=max`. С клавиатуры читается точно `max-min+1` символов. Нажатие на клавишу `Enter` не учитывается как ввод символа. Компьютер будет ждать ввода заданного количества символов (без учета нажатий на `Enter`).

Процедура ввода величины типа `String`.

Оператор вызова: `Ada.Text_IO.Get_Line ( Var, Number );`

Описание: Переменная `Var` должна иметь тип `String (min..max). Get_Line` пытается прочитать `max-min+1` символов. Чтение прекращается при нажатии `Enter`. После операции `Get_Line` переменная `Number` содержит реальное количество прочитанных символов. Если строковая переменная только частично заполнена операцией, то оставшиеся символы не определены.

Процедура ввода величины типа `Integer`.

Оператор вызова: `Ada.Integer_Text_IO.Get ( Var );`

Описание: В переменную `Var` заносится строка числовых символов. Все ведущие пробелы и нажатия на `Enter` игнорируются. Первым отличным от пробела символом может быть знак `( +, - )` или цифра. Строка данных прекращается при вводе нечислового символа или нажатии `Enter`.

Процедура ввода величины типа `Float`.

Оператор вызова: `Ada.Float_Text_IO.Get ( Var );`

Описание: В переменную `Var` заносится строка символов. Все ведущие пробелы и нажатия на `Enter` игнорируются. Первым отличным от пробела символом может быть знак `( +, - )` или цифра. Строка символов может содержать десятичную точку, окруженную числовыми символами. Строка данных прекращается при вводе символа, который не может быть частью числа с плавающей точкой, или нажатии `Enter`.

Процедура ввода величины перечисляемого типа.

Оператор вызова: `Instance.Get ( Var );`

Описание: Под `Instance` подразумевается экземпляр родового пакета `Enumeration_IO`, настроенный на конкретный перечисляемый тип. В переменную `Var` заносится строка символов. Все ведущие пробелы и нажатия на `Enter` игнорируются. Первый вводимый символ должен быть буквой, а символы должны формировать один из идентификаторов типа. Ввод прекращается при наборе символа, не принадлежащего идентификатору, или нажатии `Enter`. Если прочитанный символ не является ни одним из символов перечисляемого типа, то возбуждается `Data_Error`.

Процедура перехода на новую строку ввода.

Оператор вызова: `Ada.Text_IO.Skip_Line;`

Описание: Игнорируются оставшиеся символы текущей строки ввода (очищается буфер ввода). Выполняется принудительный переход к началу новой строки ввода.

Процедуры вывода языка Ада

Язык Ада позволяет указывать в операторе вызова не только фактические параметры, но и сопоставления формальных и фактических параметров в следующем формате:

```
< ИмяПроцедуры > ( < ФактПарам1 >,
        < ФормПарам3 > => < ФактПарам3 >, ... );
```

При такой форме порядок записи параметров безразличен. Перечислим (и охарактеризуем) возможные процедуры вывода.

Процедура вывода величины типа `Character`.

Оператор вызова: `Ada.Text_IO.Put ( Item => Var );`

Описание: Значение переменной `Var` (типа `Character`) отображается на дисплее, а курсор перемещается в следующую позицию.

Процедура вывода величины типа `String`.

Оператор вызова: `Ada.Text_IO.Put ( Item => Var );`

Описание: Переменная `Var` должна иметь тип `String (min..max)`, где `1<=min<=max`. На экране отображается точно `max-min+1` символов, а курсор перемещается в очередную позицию после конца строки.

Процедура вывода величины типа `Integer`.

Оператор вызова: `Ada.Integer_Text_IO.Put ( Var, Width => 4 )`

Описание: Отображается значение переменной `Var`, используются текущие `Width` позиций (в примере — 4) на экране. Если значение (включая знак) занимает меньше, чем `Width` позиций, ему предшествует соответствующее количество пробелов. Если

значение занимает больше, чем `Width` позиций, то используется реальное количество позиций. Если параметр `Width` пропущен, то используется ширина, заданная компилятором по умолчанию.

Процедура вывода величины типа `Float`.

Оператор вызова: `Ada.Float_Text_IO.Put (Var, Fore=>4, Aft=>2, Exp=>3);`

Описание: Значение переменной `Var` отображается на экране. Параметр `Fore` задает требуемое количество позиций для целой части числа (слева от десятичной точки); параметр `Aft` задает количество позиций для дробной части числа (справа от десятичной точки); параметр `Exp` задает количество позиций в степени (после `E`). Если реальная целая часть числа, включая знак, занимает меньше чем `Fore` позиций, слева добавляются пробелы. Если `Exp=0`, то степень не отображается.

Процедура вывода величины перечисляемого типа.

Оператор вызова: `Instance.Put ( Var, Width => 7 );`

Описание: Отображается значение переменной `Var` (перечисляемого типа), используются `Width` позиций на экране. Если значение будет занимать меньше, чем `Width` позиций, то за ним следует соответствующее количество пробелов. Если значение будет занимать больше, чем `Width` позиций, то используется реальное количество позиций. Если параметр `Width` пропущен, то используется ширина, заданная компилятором по умолчанию.

Процедура перехода на новую строку экрана.

Оператор вызова: `Ada.Text_IO.New_Line (Spacing => <число> );`

Описание: Если параметр `Spacing = 1`, то курсор перемещается в первую позицию следующей строки экрана. Если `Spacing > 1`, то это действие повторяется `Spacing` раз. Если параметр `Spacing` пропущен, то по умолчанию используется 1.

Процедура вывода величины типа `String` с переходом на новую строку экрана.

Оператор вызова: `Ada.Text_IO.Put_Line ( Var );`

Описание: Значение переменной `Var` отображается на экране, после чего курсор перемещается в начало следующей строки экрана.

Организация файлов и средства управления ими

Файл — это структура данных во внешней памяти, имеющая имя и специальную организацию. Во внешней памяти файлы только хранятся. Для обработки данные из файла вызываются в оперативную память. В каждый момент времени возможен доступ только к одному элементу файла. Этот элемент адресуется с помощью указателя файла. Данные из элемента файла заносятся в буферную переменную (рис. 18.1).

После чтения или записи указатель файла перемещается к следующему элементу и делает его доступным для обработки. Количество элементов в файле не фиксируется. Иными словами, файл имеет переменную длину.

Предусмотрены два способа доступа к элементам файла: последовательный и прямой (произвольный).

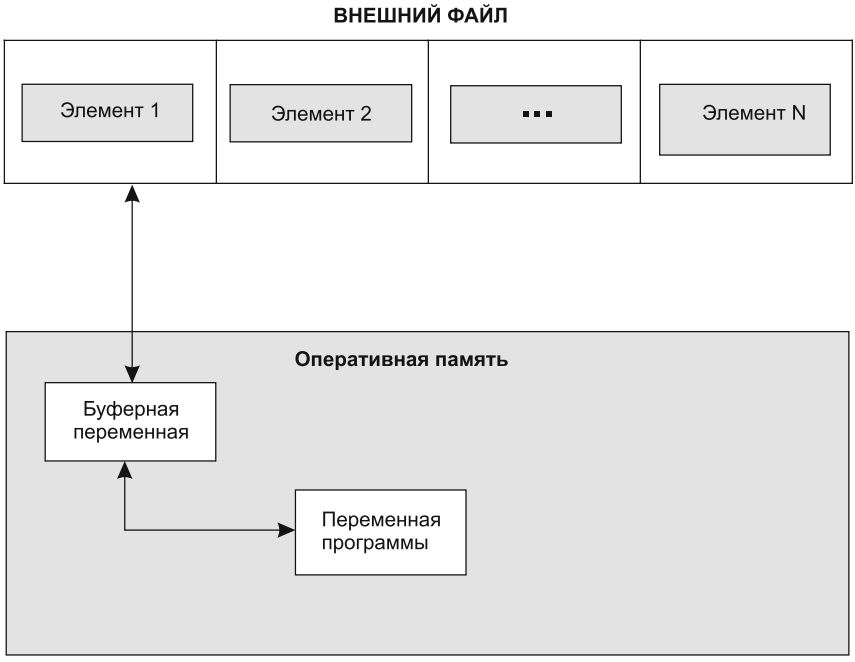

Рис. 18.1. Доступ к элементам файла

При последовательном доступе элементы файла доступны только в последовательном порядке (от первого к последнему). Таким образом, перед обращением к `n+1`-му элементу должно произойти обращение к предыдущим `n` элементам файла.

При прямом доступе возможно прямое обращение к любому элементу по его номеру в файле.

Различают понятия: внешний файл и внутренний файл.

Внешний (физический) *файл* — это реальный набор данных, существующий во внешней памяти. *Внутренний файл* — это абстракция, описываемая в программе и обеспечивающая взаимодействие с реальным внешним файлом (рис. 18.2).

Программа пользователя не имеет непосредственного доступа к внешнему файлу. Вся работа с внешними файлами описывается в ней в терминах внутренних файлов.

В языке Ада внутренний файл задается как объект файлового типа. Объявление файлового типа находится в стандартных пакетах ввода-вывода и имеет вид:

```
type File_Type is limited private;
```

Из этого объявления следует, что внутреннее содержание файловых объектов закрыто для пользователя. К файловым объектам применимы только те операции, которые определены в пакетах ввода-вывода.

Для управления файлами используются пять операций. Процедура `Create` служит для создания и открытия нового внешнего файла, связывания его с внутренним файлом. Процедура `Open` открывает уже существующий внешний файл и связывает

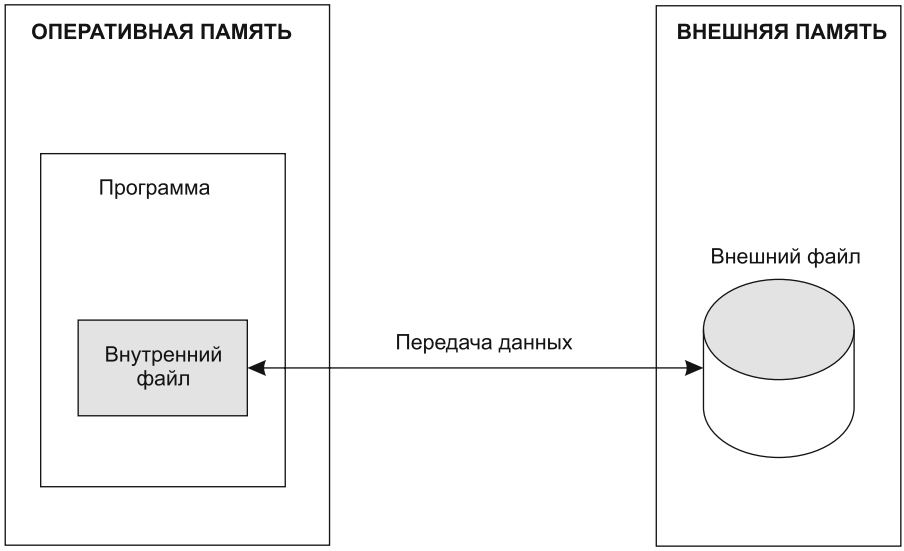

Рис. 18.2. Внешний и внутренний файлы

его с внутренним файлом. Процедура `Close` закрывает внешний файл, разрывая его связь с внутренним файлом. Процедура `Delete` удаляет внешний файл. Процедура `Reset` переводит указатель файла в начальную позицию, позволяя возобновить ввод или вывод элементов с самого начала.

Процедура `Create` имеет следующую спецификацию:

```
procedure Create ( File : in out File_Type;
            Mode : in File_Mode := режим_по_умолчанию;
            Name : in String := " ";
            Form : in String := " " );
```

где `File` — имя внутреннего файла, связываемого с создаваемым внешним файлом, `Mode` — режим работы (допустимы значения: `In_File` — ввод, `Out_File` — вывод, `InOut_File` — ввод-вывод, `Append_File` — добавление), `Name` — имя внешнего файла, `Form` — дополнительные характеристики внешнего файла.

Спецификация процедуры `Open` записывается в виде:

```
procedure Open ( File : in out File_Type;
            Mode : in File_Mode;
            Name : in String;
            Form : in String := " " );
```

Формальные параметры у процедуры `Open` те же, что и у процедуры `Create`. Однако параметры `Mode` и `Name` не имеют значений по умолчанию. Дело в том, что запрещается открывать уже открытый файл (возникает исключение `Status_Error`). Если не существует внешнего файла с указанным через `Name` именем, то генерируется исключение `Name_Error` (для процедуры `Create` оно генерируется, если имя внешнего файла недопустимо). Наконец, исключение `Use_Error` вырабатывается, если операционная среда не поддерживает открытие (или создание) внешнего файла по каким-либо другим причинам (например, на диске нет свободного места).

Фактически процедуры создания и открытия готовят внешний файл к сеансу ввода или вывода.

Процедура

```
procedure Close ( File : in out File_Type );
```

завершает сеанс работы с внешним файлом, ее параметром является внутренний файл, связь которого с внешним файлом уничтожается. Если внешний файл не был открыт, то возбуждается исключение `Status_Error`.

Процедура

```
procedure Delete ( File : in out File_Type );
```

уничтожает внешний файл, связанный с данным внутренним файлом.

Процедура `Reset` имеет две разновидности:

```
procedure Reset ( File : in out File_Type;
            Mode : in File_Mode );
procedure Reset ( File : in out File_Type );
```

С ее помощью внешний файл устанавливается в состояние, позволяющее возобновить ввод-вывод с начала файла или добавление в конец файла (для режима `Append_File`). Если задан параметр `Mode`, то в соответствии с ним устанавливается новый режим работы файла. Так, если надо создать новый файл, записать в него данные, а затем прочитать их и обработать, то необходимы следующие этапы работы:

1. Выполнение процедуры `Create`.
2. Запись данных в открытый файл.
3. Выполнение процедуры `Reset` с заданием режима `In_File`.
4. Чтение и обработка данных из файла.
5. Закрытие файла с помощью процедуры `Close`.

Для получения информации о внешнем файле определены следующие функции:

```
function Mode ( File : in File_Type ) return File_Mode;
-- возвращает характеристику текущего режима работы
function Name ( File : in File_Type ) return String;
-- возвращает имя файла
function Form ( File : in File_Type ) return String;
-- возвращает дополнит. характеристики файла
function Is_Open ( File : in File_Type ) return Boolean;
-- позволяет узнать, открыт ли файл
```

В заключение отметим, что описанные здесь процедуры и функции определены для всех разновидностей файлов Ады.

Текстовые файлы

Текстовый файл является файлом, состоящим из символов. Текстовые файлы считаются основной формой файлов, применяемых пользователем для ввода и вывода данных, так как они могут быть непосредственно распечатаны или введены с клавиатуры. Текстовые файлы — это всего лишь разновидность обычных файлов с последовательным доступом, и с ними можно производить те же действия, что и с «последовательными» файлами.

Средства для работы с текстовыми файлами содержатся в пакетах `Ada.Text_IO`, `Ada.Float_Text_IO`, `Ada.Integer_Text_IO`.

Данные в текстовом файле запоминаются в виде символов. Текстовый файл — это набор символов, группируемых в строки переменной длины. Поэтому способ доступа к элементам текстового файла — только последовательный. Текстовый файл не имеет фиксированного размера. Конец файла отмечается специальным символом `End_Of_File` (обозначается как <eof>). Точная форма представления символа-маркера <eof> зависит от операционной системы.

При создании текстового файла с помощью программы-редактора для разделения файла на строки нажимается клавиша `Enter`. Каждый раз при нажатии на `Enter` в файл помещается другой специальный символ-маркер `End_Of_Line` (обозначается <eoln>).

Обсудим содержание текстового файла, который состоит из двух строк букв, символов пробела и символов пунктуации:

```
Это текстовый файл! <eoln>
Он содержит две строки.<eoln><eof>
```

Каждая строка заканчивается маркером <eoln>. В последней строке за маркером <eoln> следует маркер <eof>. При просмотре содержимого файла каждая строка файла представляется как отдельная строка экрана. В реальном дисковом файле символы запоминаются в последовательности байтов памяти, каждый символ — в своем байте памяти. Байт первого символа второй строки (буква `O`) следует за байтом первого маркера <eoln>.

Текстовый файл может также содержать числовые данные или комбинацию числовых и буквенных данных. Следующий файл включает числовые данные и символы пробела:

```
12345 6789<eoln>
777 -11<eoln><eof>
```

Каждое число записывается на диске как последовательность цифровых символов. Символы пробелов разделяют числа, входящие в одну строку.

В диалоговой программе Ада рассматривает данные, вводимые с клавиатуры, так, как будто они читаются из предопределенного файла `Standard_Input`. Нажатие клавиши `Enter` вводит маркер <eoln> в этот файл. В диалоговом режиме для индикации окончания данных мы обычно используем сигнальную метку, а не пытаемся ввести маркер <eof> в системный файл `Standard_Input`. Впрочем, мы можем использовать маркер <eof>. Его «клавиатурное» представление зависит от операционной системы. Наиболее часто используются комбинации `Ctrl-Z` или Ctrl-D.

В свою очередь, отображение символов на экране эквивалентно записи символов в предопределенный файл `Standard_Output`. Маркер <eoln> помещает в этот файл процедура `New_Line`. В результате курсор перемещается в начало новой строки экрана. Стандартные файлы `Standard_Input` и `Standard_Output` являются текстовыми файлами, содержащими символы.

При работе со стандартными файлами процедуры `Create` (для `Standard_Output`) и `Open` (для `Standard_Input`) в программе не указываются. Эти процедуры вызываются и выполняются автоматически, по умолчанию. Не ссылаются на эти файлы

(с помощью файловых объектов) и в стандартных процедурах ввода-вывода типа `Get`, `Put`, `Skip_Line`, `New_Line`.

Маркеры `<eoln>` и `<eof>` отличны от других символов в текстовом файле, так как не являются символами данных. Фактически в стандарте для Ады они даже не определены, так как их представление зависит от операционной системы. Однако если программа попытается прочитать `<eof>`, генерируется исключение `End_Error`.

Для обработки этих маркеров в пакет `Ada.Text_IO` введены две функции. Эти функции позволяют определить — не является ли следующий символ маркером `<eoln>` или `<eof>`.

Функция

```
End_Of_Line ( <ИмяВнутреннегоФайла> )
```

возвращает значение *True*, если следующий символ является маркером <`eoln`>.

Функция

```
End_Of_File ( <ИмяВнутреннегоФайла> )
```

возвращает значение *True,* если следующий символ является маркером `<eof>`.

ПРИМЕЧАНИЕ

Для стандартных файлов ввода-вывода аргумент в этих функциях не указывается.

Алгоритм обработки файла данных на основе этих функций имеет вид:

```
while Not End_Of_File ( Имя ) loop
    while Not End_Of_Line ( Имя ) loop
            обработка каждого символа строки
    end loop;
        -- предполагается: следующий символ - <eoln>
        обработка символа <eoln>
end loop;
        -- предполагается: следующий символ - <eof>
```

Если файл данных не пуст, то начальный вызов `End_Of_File` вернет значение *False* и будет выполняться внутренний цикл. Этот цикл обрабатывает каждый символ в строке, за исключением `<eoln>`. Например, для представленного выше 2-строчного файла первое выполнение внутреннего цикла обработает первую строку:

```
Это текстовый файл!
```

Как только следующим символом станет `<eoln>`, функция `End_Of_Line` вернет значение *True* и внутренний цикл прекратится. Маркер `<eoln>` обрабатывается во внешнем цикле сразу после выхода из внутреннего цикла.

Каждое повторение внешнего цикла начинается с вызова функции `End_Of_File` для проверки — не является ли следующим символом маркер `<eof>`. Если это правда, функция возвращает *True* и внешний цикл прекращается. Если это неправда, то снова выполняется внутренний цикл. Он обрабатывает следующую строку данных (до `<eoln>`). Для нашего примера второе выполнение внутреннего цикла обработает вторую строку символов:

```
Он содержит две строки
```

После обработки второго <eoln> следующим символом будет <eof>, функция End_Of_File вернет *True,* и прекращается внешний цикл.

Для обработки произвольных текстовых файлов предусмотрены такие же процедуры, что и для стандартных файлов ввода-вывода. Единственное отличие — в список аргументов добавляется имя соответствующего внутреннего файла (указывается как первый аргумент).

Рассмотрим несколько примеров.

Если переменная NextCh имеет тип Character, то оператор вызова

```
Get ( Item => NextCh );
```

заносит следующий символ данных, набранный на клавиатуре, в переменную Next. В действительности это сокращенная форма следующего оператора:

```
Get ( File => Ada.Text_IO.Standard_Input, Item => NextCh );
```

Оператор

```
Get ( File => InData, Item => NextCh );
```

заносит следующий символ из файла InData в переменную NextCh. На этот «следующий» символ показывает указатель файла, который после каждого чтения автоматически передвигается.

Аналогичным образом операторы

```
Put ( Item => NextCh );
Put ( File => Ada.Text_IO.Standard_Input, Item => NextCh );
```

отображают значение NextCh на экране.

Оператор

```
Put ( File => OutData, Item => NextCh );
```

записывает значение NextCh в конец файла OutData.

Оператор New_Line (OutData) записывает в файл OutData маркер <eoln>.

Для текстовых файлов разрешены три режима работы:

```
In_File     -- только ввод из файла
            -- начало — с первого элемента
Out_File    -- только вывод в файл
            -- начало — с первого элемента
Append_File -- только вывод-добавление в файл
            -- начало — за последним элементом
```

Если операция обработки не соответствует режиму открытого файла, то возбуждается исключение Mode_Error. Элементы текстовых файлов могут иметь следующие типы: Character, String, числовые и перечисляемые типы.

Программа 18.1

Следующая программа копирует данные, набираемые на клавиатуре, в файл с именем Data.txt. Файловый объект OutData связывается с создаваемым файлом Data.txt, а затем используется для записи в этот файл. Если файл не может быть создан, генерируется исключение Name_Error.

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Main1 is
    OutData : Ada.Text_IO.File_Type;      -- внутренний файл
    File_Name : constant String := "Data.txt";       -- имя
    Ch : Character;                        -- читаемый символ
begin
```

```
    Create ( File => OutData, Mode => Out_File,
                Name => File_Name );
    while Not End_Of_File loop                -- для каждой строки
        while Not End_Of_Line loop            -- для каждого символа
            Get ( Ch ); Put ( OutData, Ch ); -- чтение/запись симв.
        end loop;
        Skip_Line; New_Line ( OutData );
                    -- след.строка / новая строка
    end loop;
    Close ( OutData );
    exception
        when Name_Error =>
            Put ("Не могу создать " & File_Name );
            New_Line;
end Main1;
```

Программа 18.2

Следующая программа выводит на экран содержимое дискового файла `Data.txt` (который уже существует).

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Main2 is
    InData : Ada.Text_IO.File_Type;               -- внутренний файл
    File_Name : constant String := "Data.txt";    -- имя
    Ch : Character;                               -- читаемый символ
begin
    Open ( File => InData, Mode => In_File,
                Name => File_Name );
    while Not End_Of_File ( InData ) loop
                        -- для каждой строки
        while Not End_Of_Line ( InData ) loop
                        -- для каждого символа
            Get ( InData, Ch ); Put ( Ch ); --чтение/вывод симв.
        end loop;
        Skip_Line ( InData ); New_Line;
                    -- след.строка/новая строка
    end loop;
    Close ( InData );
    exception
        when Name_Error =>
            Set_Col ( 10 );  -- сдвиг курсора в 10-ю колонку
            Put ("Не могу создать " & File_Name );
            New_Line;
end Main2;
```

Программа 18.3

Следующая программа добавляет экземпляры целого типа `Numbers` в существующий дисковый файл `Data.txt`.

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Main3 is
    type Numbers is range 1 .. 100;
    AppData : Ada.Text_IO.File_Type;           -- внутренний файл
    File_Name : constant String := "Data.txt"; -- имя
    package Numbers_IO is new
                Ada.Text_IO.Integer_IO ( Numbers );
begin
```

```
    Open ( File => AppData, Mode => Append_File,
                Name => File_Name );
    for I in Numbers loop
        Numbers_IO.Put ( AppData, I );
        New_Line ( AppData );
    end loop;
    Close ( AppData );
    exception
        when Name_Error =>
            Put ("Не могу создать " & File_Name );
            New_Line;
end Main3;
```

ПРИМЕЧАНИЕ

- Процедура Skip_Line (<Имя>) применяется при чтении данных из дискового файла. Она перемещает «стрелку» указателя текстового файла в начало новой строки.
- Процедура New_Line (<Имя>) применяется при записи данных в дисковый файл. Она записывает в конец файла маркер <eoln>. Если между операторами вызова Put (<Имя>, Ch) пропущен оператор New_Line (<Имя>), то в файле будет отсутствовать разбивка на строки, то есть данные файла образуют одну длинную строку.
- Если, например, в программе 18.1 пропустить оператор Skip_Line, то после записи в дисковый файл одной строки в него будут записываться только маркеры <eoln>.

Программа 18.4

Разработаем пакет для ввода/вывода календарных дат. Создадим его на основе стандартного пакета службы времени `Ada.Calendar`. Пакет `Ada.Calendar` содержит определение приватного типа `Time`. Текущее значение переменной этого типа возвращает функция `Clock`. В пакете имеются функции `Year`, `Month`, `Day`. Они возвращают числовые значения типов `Year_Number`, `Month_Number`, `Day_Number`.

```
with Ada.Text_IO, Ada.Calendar;
use Ada.Text_IO;
package Calendar_Dates is
    type Months is ( Jan, Feb, Mar, Apr, May, Jun,
                     Jul, Aug, Sep, Oct, Nov, Dec );
    type Date is
        record
            Day : Ada.Calendar.Day_Number;
            Month: Months;
            Year : Ada.Calendar.Year_Number;
        end record;
    procedure Get ( Item : out Date );
    procedure Put ( Item : in Date );
    function Today return Date;
end Calendar_Dates;
with Ada.Text_IO, Ada.Calendar, Ada.Integer_Text_IO;
use Ada.Integer_Text_IO;
package body Calendar_Dates is
    package Month_IO is new
```

```
                    Ada.Text_IO.Enumeration_IO ( Months );
    procedure Get ( Item : out Date ) is
    begin      -- вводит дату в формате DD MMM YYYY
        Get ( Item.Day );
        Month_IO.Get ( Item.Month );
        Get ( Item.Year );
    end Get;
    procedure Put ( Item : in Date ) is
    begin     -- отображает дату в формате DD MMM YYYY
        Put ( Item.Day, width => 1 );
        Ada.Text_IO.Put (' ');
        Month_IO.Put ( Item.Month, width => 1 );
        Ada.Text_IO.Put (' ');
        Put ( Item.Year, width => 4 );
    end Put;
    function Today return Date is
                    -- возвращает текущую дату
        Now : Ada.Calendar.Time;
        Tmp : Date;
    begin
        Now := Ada.Calendar.Clock; -- получение кода времени
        Tmp.Day := Ada.Calendar.Day ( Now );
                -- извлечение дня из кода времени
        Tmp.Month := Months'Val (Ada.Calendar.Month (Now)-1);
        Tmp.Year := Ada.Calendar.Year ( Now );
                -- извлечение года из кода времени
        return Tmp;
    end Today;
end Calendar_Dates;
```

Клиентом этого пакета может быть следующая программа:

```
with Ada.Text_IO, Calendar_Dates, Ada.Integer_Text_IO;
use Ada.Integer_Text_IO, Ada.Text_IO, Calendar_Dates;
procedure Test_Dates is
    D : Date;
begin
    D := Today;
    Put (" Today is ");
    Put ( D );    -- процедура из пакета Calendar_Dates;
    New_Line;
    Put (" Please, enter a date (DD MMM YYYY)> ");
    Get ( D );    -- процедура из пакета Calendar_Dates;
    New_Line;
    Put (" You entered: ");
    Put ( D );
end Test_Dates;
```

Двоичные файлы последовательного доступа

Файл последовательного доступа — это структура данных, состоящая из линейной последовательности компонентов одного типа, а также имеющая переменную длину (без ограничения на максимальный размер).

В отличие от текстовых файлов, двоичные файлы не могут непосредственно читаться человеком. Элементы двоичных файлов хранятся не в виде символов, а в том же формате, который принят для представления элементов данных в оперативной памяти. Например, элементы типа `Integer` запоминаются в виде 16-разрядных двоичных слов, элементы типа `Float` — в виде 32-разрядных двоичных слов. Иначе говоря, двоичные файлы можно считать фрагментами внутренней памяти, размещенными во внешней памяти.

Средства для работы с двоичными файлами последовательного доступа находятся в родовом пакете `Ada.Sequential_IO`.

Последовательные двоичные файлы состоят из элементов типа `Element_Type`, который является формальным родовым параметром пакета:

```
generic
    type Element_Type ( <> ) is private;
package Ada.Sequential_IO is
    ...
```

Из данного объявления следует, что элементы двоичного последовательного файла могут иметь числовой или перечисляемый тип, могут быть массивами и записями.

Разбивки на строки в двоичных файлах нет, поэтому маркер `<eoln>` не используется, а в пакете `Ada.Sequential_IO` отсутствует функция `End_Of_Line`.

Для работы с последовательными двоичными файлами разрешены три режима: `In_File`, `Out_File`, `Append_File`. Для ввода-вывода элементов файла используются процедуры `Read`, `Write`.

Процедура `Read` имеет следующую спецификацию:

```
procedure Read ( File : in File_Type;
            Item : out Element_Type );
```

Эта процедура читает из заданного файла один элемент и заносит его значение в параметр `Item`. При этом предполагается, что дисковый файл открыт для работы в режиме `In_File`. После выполнения процедуры «стрелка» указателя файла перемещается на следующий элемент.

Спецификация процедуры `Write` записывается в виде:

```
procedure Write ( File : in File_Type;
            Item : in Element_Type );
```

Эта процедура записывает в файл (после последнего элемента) значение нового элемента. При этом предполагается, что дисковый файл открыт для работы в режиме `Out_File` или `Append_File`.

В качестве примера рассмотрим программу, в которой:

1) создается экземпляр пакета `Ada.Sequential_IO` для работы с типом `Person`;
2) создается двоичный дисковый файл `People.txt`. Для него определяется режим работы `Out_File` (вывод), задается связывание с внутренним файлом `Data`;
3) в двоичный внешний файл `People.txt` записываются двоичные образы объектов типа `Person`;
4) меняется режим работы дискового файла. Задается режим ввода (`In_File`). Стрелка указателя файла перемещается на первый элемент;

5) читаются данные из дискового файла. Их содержимое выводится на экран;
6) внешний файл закрывается;
7) для обеспечения определения в отношении исключения `Name_Error` используется пакет `Ada.Text_IO`.

Программа 18.5

```
with Ada.Text_IO; Ada.Sequential_IO;
use Ada.Text_IO;
procedure Main4 is
    type T_Sex is ( Female, Male );
    subtype T_Height is Integer range 0 .. 250;
    type Person is record
            Name : String ( 1 .. 6 );
            Height : T_Height := 0;  -- рост в см.
            Sex : T_Sex;
        end record;
    procedure Output_Person ( One_Person : in Person ) is
    begin    -- начало процедуры
        Put ( One_Person.Name );
        Put ( " is " );
        Put ( Integer'Image ( One_Person.Height ));
        Put ("cm and is ");
        if One_Person.Sex = Female then Put ( "Female" );
                        else Put ( "Male" );
        end if;
        New_Line;
    end Output_Person;    -- конец процедуры
    type Person_Index is range 1 .. 2;
    subtype Person_Range is Person_Index;
    type Person_Array is array ( Person_Range ) of Person;
    File_Name : constant String := "People.txt";
    People : Person_Array;
    package Person_IO is new Ada.Sequential_IO ( Person );
        Data : Person_IO.File_Type;  -- внутр.файл
    begin   -- начало программы
        People ( 1 ) := ( Name => "Peter ", Height => 180,
                    Sex => Male );
        People ( 2 ) := ( Name => "Eva   ", Height => 165,
                    Sex => Female );
        Person_IO.Create ( File => Data,
                Mode => Person_IO.Out_File, Name => File_Name );
        for i in Person_Range loop
            Person_IO.Write ( Data, People ( i ));
        end loop;
        Reset ( File => Data, Mode => Person_IO.In_File );
        for i in Person_Range loop
            Person_IO.Read ( Data, People ( i ));
            Output_Person ( People ( i ));
        end loop;
        Person_IO.Close ( Data );
    exception
        when Name_Error =>
            Put ( "Can not create " & File_Name );
            New_Line;
end Main4;    -- конец программы
```

Двоичные файлы прямого доступа

Организация *файла прямого доступа* делает возможным доступ к произвольному компоненту файла. Индекс, используемый для доступа к компоненту, обычно называется *ключом*. Если ключ реализован в виде целого числа, то он очень похож на обычный индекс, используемый для обозначения компонентов массива. Однако в целом реализация файла прямого доступа и операций выбора его компонентов сильно отличается от реализации массива, так как файлы хранятся не в основной памяти, а на внешних запоминающих устройствах.

Средства для работы с двоичными файлами прямого доступа находятся в родовом пакете `Ada.Direct_IO`. Элементы этих файлов также должны принадлежать к типу `Element_Type`, который является формальным родовым параметром пакета.

Для работы с файлами прямого доступа разрешены три режима: `In_File` (ввод из файла), `InOut_File` (ввод из файла и вывод в файл), `Out_File` (вывод в файл).

Основное отличие файлов прямого доступа — обработка их элементов производится в произвольном порядке. На практике для работы с элементом такого файла надо указывать его индекс. Индекс отображает положение элемента в файле. Непосредственно после открытия файла значение индекса устанавливается равным 1.

Для обработки индекса в пакете `Ada.Direct_IO` имеются:

- процедура установки индекса `Set_Index`;
- функция определения индекса `Index`;
- функция определения максимального индекса `Size`.

Процедура

```
procedure Set_Index ( File : File_Type; To : Positive_Count );
```

устанавливает индекс, равный значению параметра `To`.

ПРИМЕЧАНИЕ

Диапазон значений подтипа Positive_Count — от 1 до зависящего от реализации числа Count'Last.

Функция

```
function Index ( File : File_Type ) return Positive_Count;
```

возвращает значение, равное текущему индексу файла.

Имеются две модификации процедуры чтения элемента из файла прямого доступа и занесения его в переменную `Item`:

```
procedure Read ( File : in File_Type; Item : out Element_Type;
           From : in Positive_Count );
procedure Read ( File : in File_Type; Item : out Element_Type);
```

Для первой модификации из файла читается элемент, индекс которого указывается параметром `From`. Для второй модификации индекс считываемого элемента равен текущему значению индекса файла. После выполнения операции ввода значение индекса увеличивается на единицу.

Аналогичным образом предусмотрены две разновидности процедуры записи из переменной `Item` в элемент файла:

```
procedure Write ( File : in File_Type; Item : in Element_Type;
           To : in Positive_Count );
procedure Write ( File : in File_Type; Item : in Element_Type );
```

Параметр `To` задает индекс элемента, в который записываются данные. Если параметр `To` не указан, то используется текущее значение индекса файла. После завершения вывода значение индекса увеличивается на 1.

Максимально возможное значение индекса файла возвращается с помощью функции

```
function Size ( File : File_Type ) return Count;
```

В качестве примера рассмотрим программу, в которой:

1) создан экземпляр пакета `Ada.Direct_IO` — для работы с типом `Book`;
2) объявлена процедура `Format` — для создания дискового файла прямого доступа (каталога книг) и последовательного заполнения его фиктивным содержимым — пробелами;
3) объявлена процедура `Load` — для загрузки в каталог фактической информации (по конкретному индексу — номеру);
4) объявлена процедура `View` — для просмотра на дисплее записи с конкретным номером;
5) приведены примеры использования этих процедур для решения конкретных задач.

Программа 18.6

```
with Ada.Text_IO, Ada.Direct_IO;
use Ada.Text_IO;
procedure Main5 is
    type Book is record
        Number : Positive;
        Author : String ( 1 .. 17 ) := ( others => ' ' );
        Name : String ( 1 .. 25 ) := ( others => ' ' );
    end record;
    package Book_IO is new Ada.Direct_IO ( Book );
    One_Book : Book;
    procedure Format ( N : Positive; Cat_Name : String ) is
        Inner : Book_IO.File_Type;  -- внутренний файл
    begin
        Book_IO.Create ( File => Inner,
                    Mode => Book_IO.Out_File,
                    Name => Cat_Name );
        for i in 1 .. N loop
            One_Book.Number := i;
            Book_IO.Write ( Inner, One_Book );
        end loop;
        Book_IO.Close ( Inner );
    end Format;
    procedure Load ( One : Book; I : Book_IO.Positive_Count;
                Cat_Name : String ) is
        Inner : Book_IO.File_Type;  -- внутренний файл
    begin
```

```
        Book_IO.Open ( File => Inner,
                       Mode => Book_IO.Out_File,
                       Name => Cat_Name );
        Book_IO.Write ( Inner, One, I );
        Book_IO.Close ( Inner );
    end Load;
    procedure View ( I : Book_IO.Positive_Count;
                Cat_Name : String ) is
        One : Book;
        Inner : Book_IO.File_Type;  -- внутренний файл
    begin
        Book_IO.Open ( File => Inner,
                       Mode => Book_IO.In_File,
                       Name => Cat_Name );
        Book_IO.Read ( Inner, One, I );
        Put ("Number is "); Put ( Positive'Image (One.Number));
        New_Line;
        Put ("Author is "); Put_Line ( One.Author );
        Put ("Name is "); Put_Line ( One.Name );
        Book_IO.Close ( Inner );
    end View;
    File_Name : constant String := "Catalog.txt";
begin     -- начало программы
    Format ( 10_000, File_Name ); -- форматирование каталога
    One_Book.Number := 10;
    One_Book.Author := "Н.Гоголь";
    One_Book.Name := "Ревизор";
    Load ( One_Book, 10, File_Name );   -- запись в каталог
    One_Book.Number := 20;
    One_Book.Author := "М.Лермонтов";
    One_Book.Name := "Демон";
    Load ( One_Book, 20, File_Name );   -- запись в каталог
    View ( 20, File_Name );  -- просмотр 20-й записи
end Main5;    -- конец программы
```

Потоки ввода-вывода

Пакеты последовательного и прямого ввода-вывода обрабатывают только однородные файлы (все элементы файла имеют один и тот же тип). Потоковый ввод-вывод позволяет обрабатывать неоднородные файлы. В языке Ada поток — это последовательность элементов, составленных из величин различных типов.

Основная идея — с любым файлом, объявленным с использованием пакета `Ada.Streams.Stream_IO`, связывается поток. Такой файл может быть обработан последовательно (с использованием потокового механизма) или с указанием позиции (как в пакете прямого ввода-вывода). Пакет `Ada.Streams.Stream_IO` позволяет создавать, открывать и закрывать файл. Дополнительно здесь имеется функция `Stream`, которая для заданного потокового файла возвращает ссылку на поток, связанный с файлом.

Спецификация этой функции имеет вид:

```
function Stream ( File : in File_Type ) return Stream_Access;
```

ПРИМЕЧАНИЕ

Все потоки являются производными от абстрактного типа Streams.Root_Stream_Type. Доступ к потоку осуществляется с помощью параметра-ссылки, указывающего на объект типа Streams.Root_Stream_Type'Class.

Последовательная обработка потоков выполняется с помощью атрибутных функций: `T'Read`, `T'Write`, `T'Input`, `T'Output`. Эти атрибуты определены для всех нелимитированных типов. Пользователь может заменить их, предусматривая спецификатор определения атрибута, может явно определить такие атрибуты для лимитированных типов. Параметры атрибутов `Read` и `Write` обозначают поток и элемент типа `T`:

```
procedure T'Write ( Stream : access
            Streams.Root_Stream_Type'Class;
        Item : in T );
procedure T'Read ( Stream : access
            Streams.Root_Stream_Type'Class;
        Item : out T );
```

Рассмотрим пример. Предположим, что надо записать смесь из целых, имен месяцев и дат, причем тип `Date` имеет вид

```
type Date is
    record
        Day : Integer;
        Month : Month_Name;
        Year : Integer;
    end record;
```

Решение

1. Обычным способом создается файл и получается доступ к связанному потоку:

```
use Streams. Stream_IO;
Mixed_File : File_Type;  -- внутренний файл
S : Stream_Access;       -- ссылочная переменная;
...
Create ( Mixed_File, "Data.doc" );
        -- создаем файл для хранения смеси
S := Stream ( Mixed_File );
        -- получаем ссылку на внутренний файл
```

2. Вызываются атрибуты для записи величин в поток:

```
Date'Write ( S, Some_Date );
Integer'Write ( S, Some_Integer );
Month_Name'Write ( S, This_Month );
...
```

ПРИМЕЧАНИЕ

Streams.Stream_IO — это не родовой пакет, он не должен конкретизироваться.

3. Все неоднородные файлы имеют один и тот же тип. Все они являются двоичными файлами.
4. Для чтения файла, записанного таким образом, должна использоваться симметричная последовательность атрибутов:

```
Date'Read ( ...
Integer'Read ( ...
Month_Name'Read ( ...
```

Если для чтения используются неподходящие подпрограммы, то будет получено странное значение или исключение `Data_Error`.

В случае простой записи, такой как `Date`, предопределенный атрибут `Write` просто вызывает атрибуты для ее компонентов. Иными словами, реализация этого атрибута имеет вид:

```
procedure Date'Write ( Stream : access
            Streams.Root_Stream_Type'Class;
            Item : in Date ) is
begin
    Integer'Write ( Stream, Item.Day );
    Month_Name'Write ( Stream, Item.Month );
    Integer'Write ( Stream, Item.Year );
end Date'Write;
```

Мы можем применить собственный вариант атрибута `Write`. Положим, что имя месяца в дате надо выводить как целое число. Тогда мы должны записать другую процедуру:

```
procedure Date_Write ( Stream : access
            Streams.Root_Stream_Type'Class;
            Item : in Date ) is
begin
    Integer'Write ( Stream, Item.Day );
    Integer'Write ( Stream, Month_Name'Pos (Item.Month) + 1 );
    Integer'Write ( Stream, Item.Year );
end Date_Write;
```

Для замены стандартной процедуры на построенную нам необходимо записать:

```
for Date'Write use Date_Write;
```

В этом случае оператор вызова `Date'Write ( S, Some_Date )` будет использовать новый формат для вывода дат.

Подобные возможности применимы и к вводу. Иначе говоря, мы должны будем объявить полную версию `Date'Read` для чтения месяца как целого числа и преобразования в соответствующее значение типа `Month_Name`.

ПРИМЕЧАНИЕ

- Мы изменяли формат вывода месяца только в датах. Если формат вывода месяца изменяется во всех случаях, то вместо переопределения `Date'Write` мы бы переопределили `Month_Name'Write`. Это оказало бы косвенное воздействие на изменение вывода дат.
- Предопределенные атрибуты `T'Read` и `T'Write` могут быть перекрыты только в той спецификации пакета, где объявлен тип `T`.

СЛЕДСТВИЕ

Для предопределенных типов эти атрибуты не могут быть изменены. Но они могут изменяться для типов, производных от них.

Ввод-вывод усложняется в случае работы с массивами и записями с дискриминантами, так как необходимо учесть закрытую информацию, представленную границами и дискриминантами. (В случае дискриминанта со значением по умолчанию он обрабатывается как обычный компонент.)

Учет закрытой информации выполняется при помощи дополнительных атрибутов `'Input` и `'Output`.

Основная идея — атрибуты обрабатывают закрытую информацию (если она есть), а затем вызывают процедуры `Read` и `Write` для обработки остатка значения.

Они имеют следующие спецификации:

```
procedure T'Output ( Stream : access
            Streams.Root_Stream_Type'Class;
            Item : in T );
function T'Input ( Stream : access
    Streams.Root_Stream_Type'Class ) return T;
```

Отметим, что `Input` реализован в виде функции, так как тип `T` может быть неопределенным и мы можем не знать ограничения при конкретном вызове.

В случае массива процедура `Output` выводит границы массива, а затем вызывает атрибут `Write` для вывода самого значения.

В случае комбинированного типа:

- Если дискриминант определен, то процедура `Output` просто вызывает атрибут `Write` (он обрабатывает дискриминант так же, как и другие компоненты).
- Если дискриминант не определен, то процедура `Output` вначале выводит дискриминант, а затем вызывает атрибут `Write` для обработки остатка записи.

Например, если задан подтип

```
subtype String_6 is String ( 1 .. 6 );
S : String_6 := "String";
```

то можно записать:

```
String_6'Output ( S );
            -- в файл выводятся и границы массива;
String_6'Write ( S ); -- границы в файл не выводятся.
```

Приведенное выше описание `T'Input`, `T'Output` относится к подразумеваемым атрибутам. Они могут быть переопределены для выполнения любых действий так, что не потребуется вызывать `T'Read`, `T'Write`. Более того, `Input` и `Output` существуют и для определенных подтипов — по умолчанию они просто вызывают `Read` и `Write`.

Для работы с надклассовыми типами имеются атрибуты: `T'Class'Output` и `T'Class'Input`.

При выводе: выводится внешнее представление тега, а затем вызывается процедура `Output` для конкретного типа (используется механизм диспетчеризации). Эта процедура выводит конкретное значение (вызовом процедуры `Write`).

Для ввода: вначале читается тег, а затем в соответствии с его значением вызывается соответствующая функция `Input` (диспетчеризацией).

Определены также атрибуты `T'Class'Write` и `T'Class'Read`. Они выполняют диспетчеризацию атрибутных подпрограмм `Write` и `Read` для конкретного типа на основе идентификации тега.

Основной принцип — все, что записано, может быть обратно прочитано с помощью соответствующей обратной операции.

Вернемся к рассмотрению основной структуры. Все потоки являются производными от абстрактного типа `Streams.Root_Stream_Type`, который имеет две абстрактные операции `Read` и `Write`:

```
procedure Read ( Stream : in out Root_Stream_Type;
            Item : out Stream_Element_Array;
            Last : out Stream_Element_Offset ) is abstract;
procedure Write ( Stream : in out Root_Stream_Type;
            Item : in Stream_Element_Array ) is abstract;
```

Они работают в терминах потоковых элементов, а не отдельных типизированных величин.

Предопределенные атрибуты `Read` и `Write` используют операции `Read` и `Write` ассоциированного потока. Пользователь может переопределить содержание атрибутов. Заметим, что в корневом типе параметр `Stream` принадлежит типу `Root_Stream_Type`, тогда как такой же параметр производного атрибута имеет ссылочный тип, обозначающий соответствующий класс. Таким образом, любой пользовательский атрибут будет выполнять разыменование (снятие косвенности):

```
procedure My_Write ( Stream : access
                Streams.Root_Stream_Type'Class; Item : T ) is
begin
        -- преобразование величины в потоковые элементы
    Streams.Write ( Stream.all, ... );  -- диспетчирование
end My_Write;
```

Пакет `Stream_IO` может использоваться и для индексированного доступа. Это возможно, так как файл является последовательностью потоковых элементов. Индексация для потоковых элементов работает так же, как и для типизированных элементов в пакете `Direct_IO`. Индекс может быть прочитан и сброшен. Предусмотрены варианты процедур `Read` и `Write` как для последовательной, так и для индексной обработки.