"Адское" программирование.
Ada-95.
Компилятор GNAT.

май 2004
V-0.4

Данный материал содержит множество различных примеров программного кода. Автор не несет ответственность если использование представленных примеров каким-либо образом повлечет за собой частичную или полную потерю Ваших данных, или приведет к частичной или полной неработоспособности Вашего оборудования.

Глава 1

Введение

Современное общество все больше зависит от программного обеспечения. Стремительное падение цен на оборудование позволяет, в настоящее время, осуществлять разработку больших программных комплексов, целевое предназначение которых весьма разнообразно. Таким образом, транспортные, финансовые, медицинские и военные системы во все возрастающей степени зависят от программного обеспечения. В результате этого, все больше возрастают требования к надежности разрабатываемого программного обеспечения.

Характерной особенностью языка программирования Ада является то, что он специально проектировался как инструмент разработки больших программных комплексов реального времени для встроенных компьютерных систем, к которым предъявляются высокие требования надежности. В первую очередь, такие требования предъявляются к системам военного предназначения. Однако, это не исключает применение языка Ада для решения всевозможных задач вычислительного характера, параллельной обработки, моделирования промышленных и технологических процессов в реальном масштабе времени, системного программирования и т.д. Более того, язык Ада часто рассматривается как язык общего назначения.

К сожалению, существующая на русском языке литература, посвященная языку программирования Ада, в большинстве случаев, относится к старому стандарту языка Ада (Ada-83). Таким образом, основной целью данной книги является попытка ликвидировать образовавшийся информационный пробел. Данная работа рассматривает средства языка программирования Ада в соответствии с действующим в настоящее время стандартом Ada-95.

Представленный материал может быть полезен и интересен широкому кругу специалистов, чья работа связана с вычислительной техникой и программированием.

Основное содержание материала логически разделено на четыре части:

- Первая часть посвящена обзору средств и возможностей языка Ада, в соответствии с действующим в настоящее время стандартом Ada-95.
- Вторая часть служит дополнением к первой части, и ее цель - дать некоторое представление об идеологии программирования на языке Ада.
- Третья часть посвящена инструментальным средствам, и она рассматривает поставку компилятора *GNAT*, поскольку он является свободно доступным (включая исходные тексты).
- Четвертую часть составляют приложения, которые содержат некоторый справочный и информационный материал.

Следует заметить, что представленный материал не можт расцениваться как исчерпывающий. Так, ввиду ограниченности объема книги, опущено обсуждение целого ряда самостоятельных тем, примерами которых могут служить: "Спецификация семантического интерфейса Ады"(*ASIS - **A**da **S**emantic **I**nterface **S**pecification*), программирование распределенных систем (*Distributed Systems*), программирование систем реального времени (*Real-Time Systems*)...

Необходимо также заметить, что данная работа не может расцениваться как перевод стандарта, поэтому, в случае возникновения каких-либо конфликтных ситуаций, которые могут возникнуть при реальной работе, необходимо непосредственно обращаться к *Ada-95 Reference Manual*, который всегда является истиной в последней инстанции.

1.1 Благодарности

Хочу выразить благодарность всем энтузиастам: Д.Анисимков, И.Васильченко, В.Годунко, Е.Зуев, А.Литвинов, М.Резник, С.Рыбин, К.Сазонов, Г.Сисюк, О.Цирюлик, - которые помогли мне конструктивными замечаниями, советами и подсказками во время написания этой книги.

Кроме того, с благодарностью будут приняты любые дополнительные замечания по содержанию книги, которые можно направлять по адресу: alex@lviv.bank.gov.ua

1.2 Некоторые исторические сведения

Сведения об истории разработки и появлении на свет языка программирования Ада, а также понимание целей его разработки не только интересны сами по себе, но они также дают некоторое представление как о самом языке, так и об его свойствах.

1.2.1 История языка программирования Ада

В 1974 году в Министерстве Обороны США (*US Department of Defence / US DoD*) осознали, что они теряют много времени, усилий и денег на разработку и сопровождение встроенных компьютерных систем (например, систем наведения ракет).

В это время использовалось около 450 различных языков программирования и/или их расширений. Это увеличивало затраты времени и средств на разработку новых систем и на постоянную техническую переподготовку персонала для обслуживания уже созданных систем. Кроме того, сопровождению существующих систем мешало отсутствие стандартизации в инструментах поддержки (редакторы, компиляторы и т.д.). Все эти факторы привели к тому, что в Министерстве Обороны США осознали необходимость в едином мощном языке программирования, который должен был бы использоваться всеми поставщиками встроенных компьютерных систем.

Работы по разработке были начаты в 1975 году после того как в Министерстве Обороны США был разработан список требований к языку, который был широко распространен. Однако, ни один из существовавших на тот момент времени языков программирования не соответствовал выдвинутым требованиям. В итоге, в 1977 году Министерство Обороны США выдвигает предложение создать новый язык. В отличие от "комитетных языков", таких как КОБОЛ, новый язык был предметом конкурсного пересмотра в широких индустриальных и академических кругах.

Из большого числа предложений было отобрано четыре, для последующего пересмотра и доработки. Позже, для дальнейшего уточнения, из них отобрали два, и в финале выбрали проект представленный компанией Cii-Honeywell Bull. Этому языку и было дано имя Ада, в честь Августы Ады Байрон, графини Лавлейс, дочери английского поэта лорда Байрона. Она была сотрудницей Чарльза Беббиджа, изобретателя аналитической машины, и написала для этой машины программу вычисления чисел Бернулли - Августа Ада по праву считается первым в мире программистом. Разработкой данного проекта руководил Jean Ichbiah.

В 1983 году язык становится стандартом ANSI/MIL-STD-1815A-1983, а затем и международным стандартом ISO 8652:1987. Язык описывается в справочном руководстве по языку (*Language Reference Manual*), часто называемом *LRM*. Ссылки на это справочное руководство часто встречаются в книгах, посвященных языку программирования Ада, а также во многих сообщениях об ошибках компилятора. Эта книга часто рекомендуется для прочтения. Хотя читать ее довольно сложно, она является исчерпывающим авторитетным источником в вопросах по языку программирования Ада (была сформирована специальная постоянная группа для выявления противоречий в языке).

Язык претерпел пересмотр при введении нового ISO стандарта в начале 1995 года (ISO/IEC 8652:1995). Этот стандарт исправляет многие упущения и недостатки оригинального языка, и дополняет его многими новыми полезными свойствами.

Для предотвращения размножения множества различных версий языка Ада, в *Ada Joint Program Office* (*AJPO*) заняли довольно оригинальную позицию - они зарегистрировали имя "Ada"как торговую марку. Таким образом, вы не имеете права распространять компиляторы языка программирования Ада до тех пор, пока они не пройдут тестирование на совместимость. Позднее эти требования были ослаблены, и теперь защищенное название звучит как *Validated Ada*.

Результирующий сертификат ратификации (валидированности) ограничен по времени и имеет дату срока истечения. После истечения срока сертификации компилятор не может больше распространятся как ратифицированный, - *Validated Ada*, - компилятор языка программирования Ада. Таким образом *AJPO* убежден в том, что все, в текуший момент распространяемые компиляторы, соответствуют требованиям текущего стандарта.

Это помогает убедиться в том, что любая программа на языке Ада может быть скомпилирована на любой системе - с этой точки зрения деятельность *AJPO* более успешна чем любые другие языковые группы.

1.2.2 Цели разработки

Возможно, что наилучшей характеристикой целей разработки послужит цитата из руководства по языку программирования Ада:

> *"Язык Ада был разработан учитывая три взаимно перекликающиеся концепции: надежность программирования и сопровождения, программирование как человеческая деятельность и эффективность"*

И как примечание к этой фразе, также из руководства по языку программирования Ада:

> *"Таким образом, читабельности программ придавалось большее значение, чем легкости их написания"*

Результаты таких целей разработки можно увидеть в получившемся языке. Он имеет строгую типизацию и принуждает к абстрагированию, что влечет за собой улучшение читабельности и облегчение сопровождения.

Ада избавлена от использования криптованного синтаксиса и обладает более "разговорным"стилем английского языка что значительно улучшает читабельность программ (читабельность - программирование как человеческая деятельность). К тому же, почти все конструкции языка могут быть эффективно реализованы.

1.3 Применение языка программирования Ада

Исходя из того, что Ада разрабатывалась по заказу Министерства Обороны США, может создаться впечатление, что Ада используется только в военных проектах. Отчасти это так. Ада действительно считается единым языком программирования как для вооруженных сил США, так и для НАТО. Однако это не исключает использование языка Ада в промышленности и образовании.

Прежде всего, Ада используется для построения больших систем к которым предъявляются достаточно высокие требования по надежности. Как правило, к таким системам относятся:

- Управляющие компьютерные системы для авиации (в том числе и гражданской).
- Управляющие компьютерные системы для скоростных железных дорог.
- Банковские системы.
- Промышленная автоматика и робототехника.
- Медицинская техника.
- Телекоммуникационные системы.

Не составит труда догадаться почему от подобных систем требуется высокая надежность.

Следует обратить внимание на то, что в настоящее время Ада достаточно активно используется в различных высших учебных заведениях США и Западной Европы, как основа для изучения программирования. Кроме этого, Ада часто используется в различных научно-исследовательских разработках. Хорошей иллюстрацией сказанного может служить разработка Ада компилятора *GNAT*, которая была начата в стенах Нью-Йоркского университета, и разработка сопутствующей этому компилятору библиотеки времени выполнения, которая велась в университете штата Флорида.

Часть 1.

Обзор средств языка Ада

Глава 2

Элементарные понятия.

2.1 "Сюрпризы"переводной терминологии

Прежде чем приступить к непосредственному знакомству с Адой, есть необходимость заметить, что в англоязычной литературе, посвященной вычислительной технике в целом и программированию в частности, достаточно часто встречаются такие английские термины как *operator*, *operation* и *statement*. При сложившейся практике перевода такой документации на русский язык, эти термины, как правило, переводят следующим образом:

operation — операция
operator — операция, оператор
statement — оператор

На первый взгляд, проблема перевода данных терминов достаточно безобидна. Однако, при таком традиционном подходе, добившись приемлемой благозвучности изложения материала, достаточно сложно избежать неоднозначностей и даже путаницы при обсуждении некоторых основополагающих вопросов. Также стоит особо отметить, что стандарт языка Ада строго различает такие понятия.

В действительности, смысловое значение данных терминов, при их употреблении в англоязычной документации, как правило, имеет следующее русскоязычное толкование:

operation — непосредственно обозначает понятие операции, функции или какого-либо действия, в совокупности составляющего выполнение команды (или набора команд) процессора

operator — обозначает понятие *знака* или *обозначения* операции, функции или какого-либо действия, что подразумевает не столько само действие для выполнения операции, сколько обозначение операции в тексте программы

statement — элемент текста программы, выражающий целостное законченное действие (или набор действий)

Таким образом, исходя из всего выше сказанного, в данной работе, для достижения однозначности, принято следующее соответствие терминов:

operation — операция
operator — знак операции
statement — инструкция

Хотя такое решение выглядит несколько не привычно, оно не должно вызвать трудности при рассмотрении материала представленного в этой работе.

2.2 Первая программа

Для того, чтобы дать "почувствовать", что представляет из себя программа написанная на языке Ада рассмотрим простую программу. Традиционно, первая программа — это программа которая выводит на экран приветствие: "Hello World!". Не будем нарушать традицию. Итак, на Аде такая программа будет иметь следующий вид:

```
with Ada.Text_IO;
use  Ada.Text_IO;

procedure Hello is
begin
    Put_Line("Hello World!");
end Hello;
```

Давайте детально рассмотрим из каких частей состоит текст этой программы. Строка "**procedure** Hello **is**" является заголовком процедуры и она указывает имя нашей процедуры. Далее, между зарезервированными словами "**begin**" и "**end**", располагается тело процедуры "Hello". В этом примере тело процедуры очень простое и состоит из единственной инструкции "Put_Line("Hello World!");". Эта инструкция осуществляет вывод приветствия на экран, вызывая процедуру "Put_Line". Процедура "Put_Line" располагается в пакете текстового ввода/вывода *Ada.Text_IO*, и становится доступной благодаря спецификации контекста в инструкциях "**with** Ada.Text_IO;" и "**use** Ada.Text_IO;" (спецификация контекста необходима для указания используемых библиотечных модулей). Здесь, спецификатор контекста состоит из двух спецификаторов: спецификатора совместности "**with**"и спецификатора использования "**use**". Спецификатор совместности "**with**" указывает компоненты которые будут использоваться в данном компилируемом модуле. Спецификатор использования "**use**" делает имена используемых объектов непосредственно доступными в данном компилируемом модуле.

Программа "Hello" настолько проста, что в ней нет ни переменных, ни какой-либо обработки данных, поэтому, несколько забегая вперед, приведем общий вид процедуры.

with ... ; **use** ... ;	спецификаторы контекста, указывающие используемые модули (могут отсутствовать)
procedure *Имя_Процедуры* ... **is**	спецификация процедуры, определяющая имя процедуры и ее параметры (если они есть)
. . .	описательная (или декларативная) часть, которая может содержать описания типов, переменных, констант и подпрограмм
begin . . .	исполняемая часть процедуры, которая описывает алгоритм работы процедуры
end *Имя_Процедуры* ;	здесь, указание имени процедуры не является обязательным

Необходимо заметить, что в отличие от языков C/C++, которые имеет функцию "main", и языка Паскаль, который имеет "program", в Аде, любая процедура без параметров может быть подпрограммой "main" (другими словами — головной программой). Таким образом, процедура без параметров может быть выбрана как головная программа во время линковки.

Теперь, приведем еще один простой пример, в котором, для выдачи сообщения приветствия, используется ранее рассмотренная процедура "Hello":

```
with Hello;     -- указывает на использование показанной ранее
                -- процедуры Hello

procedure Use_Hello is
begin
    Hello;              -- вызов процедуры Hello
end Use_Hello;
```

2.3 Библиотека и компилируемые модули

В общем случае, программа на языке Ада представляет собой один или несколько программных модулей, которые могут компилироваться как совместно, так и раздельно. Кроме того, программные модули являются основой построения библиотек Ады, поэтому их также называют библиотечными модулями. Программные модули бывают четырех видов:

Подпрограммы — Являются основным средством описания алгоритмов. Различают два вида подпрограмм: процедуры и функции. Процедура — это логический аналог некоторой именованной последовательности действий. Функция — логический аналог математической функции — используется для вычисления какого-либо значения.

Пакет — Основное средство для определения набора логически взаимосвязанных понятий. В простейшем случае в пакете специфицируются описания типов и общих объектов. В более общем случае в нем могут специфицироваться группы взаимосвязанных понятий, включающих подпрограммы, причем, некоторые описываемые в пакете сущности могут быть "скрыты"от пользователя, что дает возможность предоставления доступа только к тем ресурсам пакета, которые необходимы пользователю и, следовательно, должны быть для него доступны.

Задача или **задачный модуль** — Средство для описания последовательности действий, причем, при наличии нескольких таких последовательностей они могут выполняться параллельно. Задачи могут быть реализованы на многомашинной или многопроцессорной вычислительной конфигурации, либо на единственном процессоре в режиме разделения времени. Синхронизация достигается путем обращения ко входам, которые подобно подпрограммам могут иметь параметры, с помощью которых осуществляется передача данных между задачами.

Настраиваемые модули — Средство для параметризации подпрограмм или пакетов. В ряде случаев возникает необходимость обрабатывать объекты, которые отличаются друг от друга количеством данных, типами или какими-либо другими количественными или качественными характеристиками. Если все эти изменяемые характеристики вынести из подпрограммы или пакета, то получится некоторая заготовка (или шаблон), которую можно настроить на конкретное выполнение. Непосредственно выполнить настраиваемый модуль нельзя. Но из него можно получить экземпляр настроенного модуля (подпрограмму или пакет), который пригоден для выполнения.

Каждый программный модуль обычно состоит из двух частей: спецификации и тела. Спецификация описывает интерфейс к модулю, а тело — его реализацию. Примечательно, что спецификация и тело программного модуля являются самостоятельными компилируемыми модулями, то есть, они могут компилироваться раздельно. Разбиение модуля на спецификацию и тело, а также возможность раздельной компиляции позволяют разрабатывать, кодировать и тестировать любую программу или систему как набор достаточно независимых компонентов. Такой подход полезен при разработке больших программных систем.

2.4 Лексические соглашения

Лексические соглашения описывают допустимые символы и последовательности символов, которые используются при обозначении идентификаторов, определяющих имена переменных и констант, подпрограмм и пакетов, указывают правила написания числовых значений, а также описывают некоторые специальные последовательности символов, используемые языком программирования. Согласно требований стандарта, реализация должна поддерживать как минимум 200-символьные строки исходного текста (максимальная длина лексических элементов — не определена).

2.4.1 Комментарии

Начнем с комментариев. Для облегчения понимания алгоритма работы программы, в текст программы могут, и должны помещаться комментарии. Комментарий начинается с двух символов дефиса "−−" и продолжается до конца строки. Пример:

```
-- это комментарий
--- это тоже комментарий
```

2.4.2 Идентификаторы

Теоретически, согласно требований стандарта, идентификаторы могут быть любой длины, однако, длина может быть ограничена реализацией конкретного компилятора. Общие правила таковы:

1. Идентификатор может состоять из букв, цифр и символов подчеркивания.

2. Идентификатор **обязан** начинаться с символа.

3. В идентификаторе **нельзя** использовать несколько символов подчеркивания подряд.

4. Символ подчеркивания **не** может быть первым и последним символом в идентификаторе.

5. Все идентификаторы в ADA **не** зависят от регистра символов.

Например:

```
Apple, apple, APPLE      -- один и тот же идентификатор
Max_Velocity_Attained
Minor_Number_            -- недопустимо, завершающий символ - подчеркивание
Minor__Revision          -- недопустимо, последовательность подчеркиваний
```

2.4.3 Литералы

Литералы служат для явного указания значения некоторого типа, сохраняемого в программе. Различают числовые, символьные и строковые литералы.

2.4.3.1 Числовые литералы

Числовые литералы, как не трудно догадаться, используются для представления численных значений. Они могут содержать в себе символы подчеркивания (для удобочитаемости), однако, они **не** могут начинаться или заканчиваться символом подчеркивания, или содержать более одного символа подчеркивания подряд. Различают числовые литералы для представления целочисленных и вещественных значений.

Примеры целочисленных литералов, представляющих значение числа 2000:

```
2000
2_000
2E3          -- для целочисленных литералов разрешена экспоненциальная форма
2E+3
```

Возможно представление чисел в разных системах счисления, например, представление десятичного числа 12 может быть задано следующими литералами:

```
2#1100#    -- двоичная система счисления
8#14#      -- восьмеричная
10#12#     -- десятичная (здесь, указана явно)
16#C#      -- шестнадцатиричная
7#15#      -- семиричная
```

Литералы, описывающие вещественные значения, содержат точку и обязаны иметь хотя бы по одной цифре до и после точки. Примеры:

```
3.14
100.0
0.0
```

2.4.3.2 Символьные литералы

Символьные литералы обозначают одиночные символы, и для их обозначения используются одинарные кавычки. Например:

```
'a'
'b'
```

Примечание:
В отличие от языка Паскаль, в Аде **нельзя** производить присваивание символьного литерала строковой переменной (подробности о работе со строками в Аде мы пока отложим на потом).

2.4.3.3 Строковые литералы

Строковые литералы предназначены для обозначения строковых значений, и для их обозначения используются двойные кавычки. Например:

```
"это строковый литерал"
"a"                     -- это тоже строковый литерал,
                        -- хотя и односимвольный
```

Примечание:
Строковый литерал не может непосредственно содержать символ табуляции, хотя значение строковой переменной или константы — может. Это достигается путем конкатенации строки и символа, или вставкой символа в строку.

2.4.4 Зарезервированные слова

Некоторые слова, такие как "**with**", "**procedure**", "**is**", "**begin**", "**end**" и т.д., являются частью самого языка программирования. Такие слова называют зарезервированными (или ключевыми) и они не могут быть использованы в программе в качестве имен идентификаторов. Полный список зарезервированных слов Ады приводится ниже:

abort	**else**	**new**	**return**
abs	**elsif**	**not**	**reverse**
* **abstract**	**end**	**null**	
accept	**entry**		**select**
access	**exception**		**separate**
* **aliased**	**exit**	**of**	**subtype**
all		**or**	
and	**for**	**others**	* **tagged**
array	**function**	**out**	**task**
at			**terminate**
	generic	**package**	**then**
begin	**goto**	**pragma**	**type**
body		**private**	
	if	**procedure**	
case	**in**	* **protected**	* **until**
constant	**is**		**use**
		raise	
declare		**range**	**when**
delay	**limited**	**record**	**while**
delta	**loop**	**record**	**while**
digits		**renames**	
do	**mod**	* **requeue**	**xor**

Примечание:
Зарезервированные слова помеченные звездочкой введены стандартом Ada95.

2.5 Методы Ады: подпрограммы, операции и знаки операций

Методами Ады являются подпрограммы (процедуры и функции), а также операции и знаки операций (возможно более корректно будет звучать: с помощью подпрограмм осуществляется реализация действий, выполняемых операциями и знаками операций). Необходимо отметить, что стандарт Ады строго различает понятия знаков операций (*operators*) и операций (*operations*).

Знаки операций представляются следующими символами (или комбинациями символов): "=", "/=", "<", ">", "<=", ">=", "\&", "+", "–", "/", "*". Другие знаки операций выражаются зарезервированными словами: "**and**", "**or**", "**xor**", "**not**", "**abs**", "**rem**", "**mod**", — или могут состоят из нескольких зарезервированных слов: "**and then**", "**or else**". Ада позволяет осуществлять программисту совмещение (*overloading*) знаков операций (в современной литературе по Си++ это часто называется как "перегрузка операторов").

В общем случае, **совмещением** (*overloading*) называют механизм, который позволяет различным сущностям использовать одинаковые имена.

Использование "**use type**" делает знаки операций именованных типов локально видимыми. Кроме того, их можно сделать локально видимыми используя локальное переименование.

Операции включают в себя присваивание, проверку принадлежности диапазону и любые другие именованные операции. Операции, также как и знаки операций, допускают совмещение.

Следует заметить, что Ада накладывает некоторые ограничения на использование совмещений: совмещения не допускаются для операций присваивания и проверки принадлежности диапазону, а также для знаков операций "**and then**" и "**or else**".

Операция присваивания обозначается комбинацией символов ":=". Она предопределена для всех нелимитированных типов. Операция присваивания не может быть совмещена или переименована. Присваивание запрещено для лимитированных типов. Необходимо подчеркнуть, что операция присваивания в Аде, в отличие от языков C/C++, не возвращает значение и не обладает побочными эффектами.

Еще одной разновидностью операций является операция проверки принадлежности диапазону, которая обозначается с помощью зарезервированного слова "**in**". Для выполнения проверки на не принадлежность "**in**" может комбинироваться с "**not**" — "**not in**". Проверка на принадлежность диапазону разрешена для всех типов Ады, включая лимитированные.

Другие операции могут быть описаны программистом. Как правило, описания таких операций выполняются в спецификации пакета, а реализация операций выполняется с помощью соответствующих подпрограмм.

2.6 Инструкции, выражения и элаборация

Очевидно, что исполнение инструкций осуществляется во время выполнения программы с целью выполнить какие-либо действия. Также, во время выполнения программы осуществляются вычисления различных выражений для получения значений каких-либо типов. Кроме того, во время выполнения программы происходит вычисление различных имен, которые указывают на соответствующие объекты (содержащие какие-либо значения) или другие сущности (такие как подпрограммы и типы).

Некоторые конструкции языка содержат описательные части, сопровождаемые последовательностями инструкций. Например, тело процедуры может иметь следующий вид:

```
procedure P ( ... ) is
    I: Integer := 1;     -- описательная часть
    . . .
begin
    . . .                -- последовательность инструкций
    I := I * 2;
    . . .
end P;
```

Перед выполнением тела процедуры происходит элаборация (*elaboration*) всех описаний, которые указаны в описательной части. Последовательность элаборации описаний определяется порядком их следования в описательной части. Эффект от элаборации описаний заключается в создании сущностей, определенных в описаниях, и в выполнении прочих действий, которые специфичны для описаний. Например, элаборация описания переменной может осуществить инициализацию этой переменной значением, которое определяется результатом вычисления какого-либо выражения. Достаточно часто значения подобных выражений могут быть вычислены в процессе компиляции программы.

После завершения элаборации, осуществляется исполнение последовательности инструкций, в порядке их следования (за исключением случаев, когда осуществляется передача управления в какое-либо другое место, отличное от последующей инструкции). Инструкция присваивания позволяет заменить значение переменной результатом вычисления выражения того же самого типа. Обычно, присваивание осуществляется простым побитовым копированием значения, которое получено в результате вычисления выражения. Однако, в случае нелимитированных контролируемых типов, после осуществления побитового копирования, пользователь (при необходимости) может определить дополнительную последовательность действий. Инструкции "**case**" и "**if**" позволяют осуществлять выбор выполнения определенной последовательности инструкций, полагаясь на результат вычисления какого-нибудь выражения. Инструкция "**loop**" позволяет повторять выполнение

последовательности каких-либо инструкций, согласно выбранной схемы итерации, или до обнаружения инструкции "**exit**". Инструкция "**goto**" осуществляет передачу управления в место отмеченное соответствующей меткой.

Выражения могут быть использованы в различном контексте, как в описаниях, так и в инструкциях. Используемые в языке Ада выражения, во многом подобны выражениям, которые используются в большинстве современных языков программирования. Они могут содержать обращения к переменным, константам и литералам, кроме того они могут использовать любые операции, которые возвращают значения. Результатом вычисления любого выражения является значение. Каждое выражение имеет определенный тип, который известен на этапе компиляции.

Во многих случаях результаты вычисления выражений и ограничения подтипов определяются статически (существует возможность отключения некоторых динамических проверок ограничений подтипов с помощью использования соответствующих опций компилятора). Более того, достаточно часто компилятор Ады требует чтобы вычисление некоторых выражений и подтипов осуществлялось на этапе компиляции программы. Например, в общем случае вся информация об описании известна во время компиляции, следовательно, элаборация во время выполнения программы не потребует выполнения какого-либо машинного кода. Язык определяет механизмы согласно которых Ада-компиляторы могут осуществлять предварительную элаборацию некоторых модулей, то есть, реальные действия, которые необходимы для осуществления элаборации, выполняются однократно, на этапе компиляции программы, вместо того, чтобы выполнять их при каждом запуске программы.

2.7 Директивы компилятора

Бывают случаи, когда в исходном тексте необходимо указывать какую-либо дополнительную информацию, которая предназначена сугубо для компилятора. Например, такая информация может предоставлять компилятору дополнительные сведения о режимах трансляции программного модуля, который компилируется в текущий момент времени (оптимизация генерируемого двоичного кода, вставка отладочного кода и т.д.) или управлять распечаткой листинга трансляции.

Для Ады, как и для многих других современных языков и сред программирования, такими средствами передачи дополнительной информации компилятору являются директивы компилятора. При указании директивы компилятору Ада использует зарезервированное слово "**pragma**". Общий вид указания директивы компилятору следующий:

pragma *имя_директивы* (*параметры_директивы*) ;

Стандарт языка Ада определяет 39 директив, список которых представлен в приложении L (*Annex L*) руководства по языку программирования Ада (*RM-95*). Кроме того, конкретная реализация компилятора может обеспечивать дополнительные директивы компилятора которые, как правило, описываются в сопроводительной документации компилятора.

Выражаясь не строго, можно сказать, что директивы компилятора не изменяют общий смысл программы.

Глава 3

Скалярные типы данных языка Ада.

К скалярным типам относятся типы данных которые определяют соответствующие упорядоченные множества значений. Эта глава описывает скалярные типы данных Ады, а также атрибуты и операции допустимые для этих типов.

Предопределенный пакет *Standard* содержит описания стандартных типов, таких как "Integer", "Float", "Boolean", "Character" и "Wide_Character", а также определяет операции, которые допускается производить над этими типами.

Следующие знаки операций допустимы для всех скалярных типов:

```
=, /=            -- проверка на равенство/не равенство
<, <=, >, >=     -- меньше, меньше или равно, больше, больше или равно
in, not in       -- проверка принадлежности к диапазону
```

Перед тем как приступить к непосредственному детальному обсуждению скалярных типов Ады, необходимо сделать некоторое общее введение в систему типов языка Ада

3.1 Введение в систему типов языка Ада

Данные — это то, что обрабатывает программа. Практически, любой современный язык программирования определяет свои соглашения и механизмы для разделения данных на разные типы.
Известно, что Ада — это язык со строгой типизацией. И хотя это не единственное свойство Ады, но, пожалуй, это свойство наиболее широко известно.

Понятие типа данных Ады подразумевает, что:

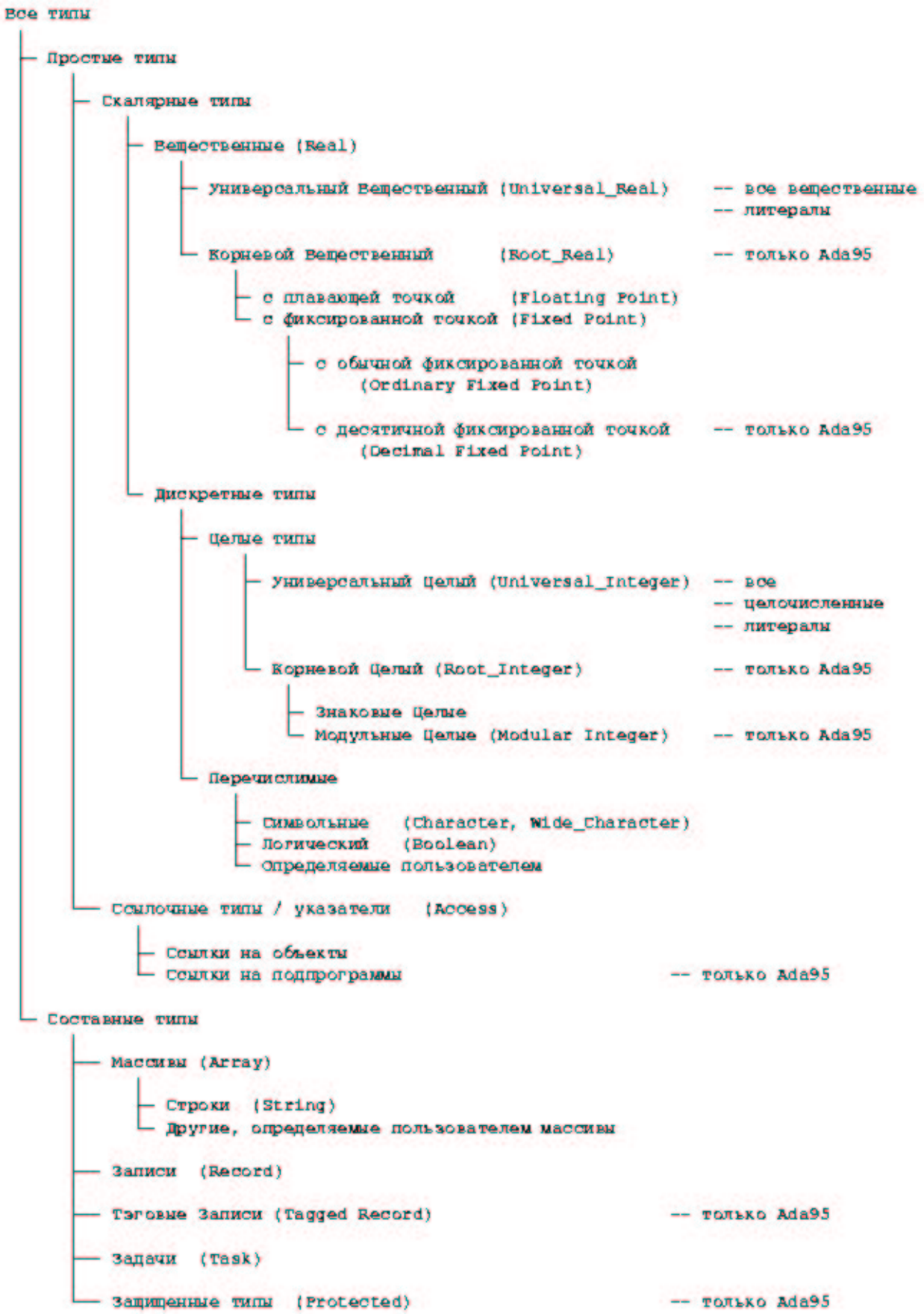
1. каждый тип данных имеет свое имя
2. каждый тип данных имеет свое множество допустимых значений
3. для каждого типа данных определено свое множество допустимых операций и знаков операций
4. строгое разделение объектов одного типа данных от объектов любого другого типа данных

Исходя из сказанного, в отличие от других языков программирования, Ада **не допускает** прямого присваивания значений одного типа данных другому типу данных и/или любого **неявного** преобразования значений одного типа в значения другого типа. В случае необходимости преобразования значений одного типа в значения другого типа можно указать (или описать) требуемое правило преобразования, но такое правило преобразования должно быть указано явно.

На первый взгляд, строгое требование явного указания или описания всех правил преобразования выглядит раздражающе (особенно для тех кто привык к C) - без явного указания таких правил компилятор просто отказывается компилировать вашу программу. Однако, в последствии это предохраняет от долгих и мучительных поисков, в процессе отладки, тех "сюрпризов"неявного преобразования и/или некорректного присваивания значений различных типов, которые допускаются в других языках программирования. Таким образом, подобный подход является одним из основополагающих факторов обеспечения надежности программного обеспечения.

Ада обладает достаточным набором предопределенных встроенных типов, а также предоставляет богатые возможности для описания новых типов данных и их подтипов (*subtype*). Кроме того, в языке представлены средства, которые позволяют гибко управлять внутренним представлением вновь создаваемых типов данных.

Приводимая ниже диаграмма демонстрирует общую организацию системы типов Ады.



Примечательно, что, в отличие от языка Паскаль, Ада не предоставляет предопределенного механизма

поддержки множественных типов. Однако, богатые средства описания типов Ады, при необходимости, позволяют программисту конструировать подобные типы без значительных усилий.

3.2 Целочисленные типы

3.2.1 Предопределенный тип `Integer`

Предопределенный целочисленный тип "Integer" описан в пакете *Standard* (пакет *Standard* не нужно указывать в инструкциях спецификации контекста "**with**" и "**use**"). Точный диапазон целочисленных значений, предоставляемых этим типом, зависит от конкретной реализации компилятора и/или оборудования. Однако, стандарт определяет минимально допустимый диапазон значений для этого типа от -(2 ** 15) до +(2 ** 15 - 1) (например, в случае 32-битных систем, таких как *Windows* или *Linux*, для реализации компилятора GNAT диапазон значений типа "Integer" будет от -(2 ** 31) до +(2 ** 31 - 1)).

Кроме типа "Integer", в пакете *Standard* предопределены два его подтипа (понятие подтипа будет рассмотрено позже) "Natural" и "Positive", которые, соответственно, описывают множества не отрицательных (натуральных) и положительных целых чисел.

3.2.2 Тип `Universal_Integer`

Для предотвращения необходимости явного преобразования типов при описании целочисленных констант, Ада предлагает понятие универсального целого типа — "Universal_Integer". Все целочисленные литералы принадлежат типу "Universal_Integer". Многие атрибуты языка (обсуждаемые позже) также возвращают значения универсального целого типа. Установлено, что тип "Universal_Integer" совместим с любым другим целочисленным типом, поэтому, в арифметических выражениях, компилятор будет автоматически преобразовывать значения универсального целого типа в значения соответствующего целочисленного типа.

3.2.3 Описание целочисленных констант

При хорошем стиле программирования, принято присваивать целочисленным значениям символьные имена. Это способствует улучшению читабельности программы и, при необходимости, позволяет легко вносить в нее изменения.

Приведем простой пример описания константы (именованного числа):

```
Max_Width : constant := 10_000; -- 10_000 - имеет тип Universal_Integer
                                -- это не тип Integer
```

3.2.4 Тип `Root_Integer`

Модель целочисленной арифметики Ады базируется на понятии неявного типа "Root_Integer". Этот тип используется как базовый тип для всех целочисленных типов Ады. Другими словами — все целочисленные типы являются производными от типа "Root_Integer" (см. Производные типы). Диапазон значений типа "Root_Integer" определяется как "System.Min_Int..System.Max_Int". Все знаки арифметических операций описаны так, чтобы они могли выполняться над этим типом.

При описании нового целочисленного типа возможны два различных подхода, которые можно проиллюстрировать на следующем примере:

```
type X is new Integer range 0 .. 100;
type Y is range 0 .. 100;
```

Здесь, тип "X" описывается как производный от типа "Integer"с допустимым диапазоном значений от 0 до 100. Исходя из этого, для типа "X" базовым типом будет тип "Integer".

Тип "Y" описывается как тип с допустимым диапазоном значений от 0 до 100, и при его описании не указан тип-предок. В таком случае, он будет производным от типа "Root_Integer", но его базовый диапазон не обязательно должен быть таким же как у "Root_Integer". В результате, некоторые системы могут размещать экземпляры объектов такого типа и его базового типа в одном байте. Другими словами, определение размера распределяемого места под объекты такого типа возлагается на компилятор.

3.2.5 Примеры целочисленных описаний

Ниже приводятся примеры различных целочисленных описаний Ады.

```
-- описания целочисленных статических переменных

Count           : Integer;
X, Y, Z         : Integer;
Amount          : Integer := 0;


-- описания целочисленных констант (иначе – именованных чисел)

Unity           : constant Integer := 1;
Speed_Of_Light  : constant := 300_000; -- тип Universal_Integer
A_Month         : Integer range 1..12;


-- описания целочисленных типов и подтипов
-- ( см. разделы "Подтипы" и "Производные типы" )

subtype Months is Integer range 1..12;        -- огранниченный тип Integer
-- подтипы – совместимы с их базовым типом (здесь – Integer)
-- например, переменная типа Month может быть "смешана" с переменными
-- типа Integer

type File_Id is new Integer; -- новый целочисленный тип, производный
                             -- от типа Integer

type Result_Range is new Integer range 1..20_000;
-- производный тип с объявлением ограничения

type Other_Result_Range is range 1..100_000;
-- тип производный от Root_Integer
-- при этом, компилятор будет выбирать подходящий размер целочисленного значения
-- для удовлетворения требований задаваемого диапазона
```

3.2.6 Предопределенные знаки операций для целочисленных типов

Следующие знаки операций предопределены для каждого целочисленного типа:

```
+, –            -- унарные плюс и минус
+, –, *, /      -- сложить, вычесть, умножить и разделить
**              -- возведение в степень (только целые значения степени)
mod             -- модуль
rem             -- остаток
abs             -- абсолютное значение
```

3.2.7 Модульные типы

Все целочисленные типы, которые мы рассматривали ранее, известны как целые числа со знаком. Для таких типов соблюдается правило — если в случае вычисления результат выходит за диапазон допустимых значений типа, то генерируется ошибка переполнения. Такие целочисленные типы были представлены стандартом Ada83.

Стандарт Ada95 разделяет целочисленные типы на целые числа со знаком и модульные типы. По существу, модульные типы являются целыми числами без знака. Характерной особенностью таких типов является свойство цикличности арифметических операций. Таким образом, модульные типы соответствуют целочисленным беззнаковым типам в других языках программирования (например: "**Byte**", "Word"... — в реализациях Паскаля; "unsigned_short", "**unsigned**"... — в C/C++).

В качестве простого примера рассмотрим следующий фрагмент кода:

```
    . . .
type  Byte  is mod 2 ** 8;     -- (2 ** 8) = 256
    Count : Byte := 255;
begin
    Count := Count + 1;
    . . .
```

Здесь не производится генерация ошибки в результате выполнения сложения. Вместо этого, переменная "Count", после выполнения сложения, будет содержать 0.

Кроме этого, с модульными типами удобно использовать знаки битовых операций "**and**", "**or**", "**xor**" и "**not**". Такие операции трактуют значения модульного типа как битовый шаблон. Например:

```
type  Byte  is mod 2 ** 8;     -- (2 ** 8) = 256
    Some_Var_1  : Byte;
    Some_Var_2  : Byte;
    Mask        : constant := 16#0F#
begin
    . . .
    Some_Var_2 := Some_Var_1 and Mask;
    . . .
```

Поскольку модульные типы не имеют отрицательных значений, для них допускается смешивание знаков битовых операций со знаками арифметических операций в одном выражении.

Следует заметить, что хотя при описании модульных типов зачастую используют степень двойки, использование двоичной системы счисления при описании модульных типов не является обязательным требованием.

Ада допускает выполнение преобразований беззнаковых чисел модульных типов в числа со знаком и обратно. При этом, производится проверка результата преобразования на допустимость диапазону значений типа назначения. В случае неудачи будет сгенерировано исключение *Constraint_Error*. Например:

```
type  Unsigned_Byte  is mod 2 ** 8;     -- (2 ** 8) = 256
type  Signed_Byte    is range -128 .. +127;

U : Unsigned_Byte := 150;
S : Signed_Byte   := Signed_Byte (U);  -- здесь будет сгенерировано исключение
                                       -- Constraint_Error
```

Этот код будет вызывать генерацию исключения *Constraint_Error*.

В следующем параграфе приводятся описания предопределенных модульных типов представленных в пакете *Interfaces*. Заметим, что для этих модульных типов (они описаны с использованием двоичной системы счисления) предопределены операции побитного сдвига и вращения.

3.2.8 Дополнительные целочисленные типы системы компилятора *GNAT*

Стандарт языка Ада допускает определять в реализации Ада-системы собственные дополнительные целочисленные типы. Таким образом, в пакете *Standard* системы компилятора *GNAT* определены дополнительные целочисленные типы:

```
type  Short_Short_Integer  is range -(2 ** 7) .. +(2 ** 7 - 1);
type  Short_Integer        is range -(2 ** 15) .. +(2 ** 15 - 1);
type  Long_Integer         is range -(2 ** 31) .. +(2 ** 31 - 1);
type  Long_Long_Integer    is range -(2 ** 63) .. +(2 ** 63 - 1);
```

Кроме этого, стандарт требует наличия определения дополнительных 8-, 16-, 32- и 64-битных целочисленных типов в пакете *Interfaces*:

```
type  Integer_8     is range   -2 **  7 .. 2 **  7 - 1;
type  Integer_16    is range   -2 ** 15 .. 2 ** 15 - 1;
type  Integer_32    is range   -2 ** 31 .. 2 ** 31 - 1;
type  Integer_64    is range   -2 ** 63 .. 2 ** 63 - 1;

type  Unsigned_8    is mod   2 **  8;
type  Unsigned_16   is mod   2 ** 16;
type  Unsigned_32   is mod   2 ** 32;
type  Unsigned_64   is mod   2 ** 64;
```

3.3 Вещественные типы

Ада предусматривает два способа представления вещественных чисел: представление вещественных величин с плавающей точкой и представление вещественных величин с фиксированной точкой. Кроме этого вы можете использовать типы вещественных величин с десятичной фиксированной точкой.

3.3.1 Вещественные типы с плавающей точкой, тип `Float`

Вещественные типы с плавающей точкой имеют неограниченный диапазон значений и точность, определяемую количеством десятичных цифр после запятой. Представление чисел с плавающей точкой имеет фиксированную относительную погрешность.

Пакет *Standard* предоставляет предопределенный вещественный тип с плавающей точкой "Float", который обеспечивает точность в шесть десятичных цифр после запятой:

```
type  Float  is digits 6 range -16#0.FFFF_FF#E+32 .. 16#0.FFFF_FF#E+32;
                          --     -3.40282E+38 ..  3.40282E+38
```

В пакете *Standard* компилятора *GNAT*, для 32-битных систем Linux и Windows, дополнительно представлены еще несколько вещественных типов с плавающей точкой (фактические значения констант для различных платформ отличаются):

```
type  Short_Float      is digits 6
  range -16#0.FFFF_FF#E+32 .. 16#0.FFFF_FF#E+32;
  --    -3.40282E+38 ..  3.40282E+38

type  Long_Float       is digits 15
  range -16#0.FFFF_FFFF_FFFF_F8#E+256 .. 16#0.FFFF_FFFF_FFFF_F8#E+256;
  --    -1.79769313486232E+308 ..  1.79769313486232E+308

type  Long_Long_Float  is digits 18
  range -16#0.FFFF_FFFF_FFFF_FFFF#E+4096 .. 16#0.FFFF_FFFF_FFFF_FFFF#E+4096;
  --    -1.18973149535723177E+4932 ..  1.18973149535723177E+4932
```

Ниже следуют примеры описаний вещественных величин с плавающей точкой.

```
X         : Float;
A, B, C   : Float;
Pi        : constant Float := 3.14_2;
Avogadro  : constant := 6.027E23;          -- тип Universal_Float

subtype Temperatures is Float range 0.0..100.0;
type Result is new Float range 0.0..20_000.0;

type Velocity is new Float;
type Height is new Float;
-- нельзя случайно смешивать Velocity и Height
-- без явного преобразования типов.

type Time is digits 6 range 0.0..10_000.0;
-- в этом диапазоне требуемая точность - шесть десятичных цифр
-- после точки

type Degrees is digits 2 range -20.00..100.00;
-- требуемая точность - две десятичных цифры после точки
```

Следующие знаки операций предопределены для каждого вещественного типа с плавающей точкой.

```
+, -, *, /
**                -- возведение в степень (только целые значения степени)
abs               -- абсолютное значение
```

3.3.2 Вещественные типы с фиксированной точкой, тип `Duration`

Представление чисел с фиксированной точкой имеет более ограниченный диапазон значений и указанную абсолютную погрешность, которая задается как "**delta**" этого типа.

В пакете *Standard* предоставлен предопределенный вещественный тип с фиксированной точкой "Duration", который используется для представления времени и обеспечивает точность измерения времени в 50 микросекунд:

```
type Duration is delta 0.000000001
     range -((2 ** 63 - 1) * 0.000000001) ..
           +((2 ** 63 - 1) * 0.000000001);
```

Ниже следуют примеры описаний вещественных типов с фиксированной точкой.

```
type Volt is delta 0.125 range 0.0 .. 255.0;

type Fraction is delta System.Fine_Delta range -1.0..1.0;          -- Ada95
        -- Fraction'Last = 1.0 - System.Fine_Delta
```

Последний пример показывает полезную способность вещественных типов с фиксированной точкой - четкое определение насколько тип должен быть точным. Например, это позволяет контролировать ошибки, возникающие при округлении.

3.3.3 Вещественные типы с десятичной фиксированной точкой

Следует учитывать, что поскольку от реализации компилятора Ады не требуется обязательное обеспечение поддержки вещественных величин с десятичной фиксированной точкой, то это может вызвать трудности при переносе программного обеспечения на другую систему или при использовании разных компиляторов.

Примером описания вещественного типа с десятичной фиксированной точкой может служить следующее:

```
type Money is delta 0.01 digits 15; -- десятичная фиксированная точка,
                                   -- здесь величина, задаваемая в delta,
                                   -- должна быть степенью 10
subtype Salary is Money digits 10;
```

3.3.4 Типы `Universal_Float` и `Root_Real`

Подобно тому как все целочисленные литералы принадлежат универсальному классу "Universal_Integer", все вещественные численные литералы принадлежат универсальному классу "Universal_Float". Этот универсальный тип совместим с любым вещественным типом с плавающей точкой и любым вещественным типом с фиксированной точкой. Такой подход избавляет от необходимости выполнения различных преобразований типов при описании вещественных констант и переменных.

Модель вещественной арифметики Ады основывается на анонимном типе "Root_Real". Этот анонимный тип используется как базовый тип для всех вещественных типов. Тип "Root_Real" имеет точность, которая определяется значением константы "Max_Base_Digits" пакета *System* ("System.Max_Base_Digits"). Такой подход использован для облегчения переносимости программ.

3.3.5 Пакеты для численной обработки

Полное обсуждение поддержки численной обработки в Аде — весьма обширная тема. Поэтому здесь, чтобы указать "куда бежать дальше", мы только приведем список пакетов для численной обработки, которые предоставляются поставкой компиляора *GNAT*:

```
Ada.Numerics
Ada.Numerics.Aux
Ada.Numerics.Float_Random
Ada.Numerics.Discrete_Random

Ada.Numerics.Complex_Types
Ada.Numerics.Complex_Elementary_Functions
```

```
Ada.Numerics.Elementary_Functions

Ada.Numerics.Generic_Complex_Types
Ada.Numerics.Generic_Complex_Elementary_Functions
Ada.Numerics.Generic_Elementary_Functions

Ada.Numerics.Long_Complex_Types
Ada.Numerics.Long_Complex_Elementary_Functions
Ada.Numerics.Long_Elementary_Functions

Ada.Numerics.Long_Long_Complex_Types
Ada.Numerics.Long_Long_Complex_Elementary_Functions
Ada.Numerics.Long_Long_Elementary_Functions

Ada.Numerics.Short_Complex_Types
Ada.Numerics.Short_Complex_Elementary_Functions
Ada.Numerics.Short_Elementary_Functions
```

Следует заметить, что пакет *Ada.Numerics.Aux*, который указан выше, не предназначен для непосредственного использования в программах пользователя, и упоминается только с целью полноты показанного выше списка.

3.4 Преобразование численных типов

Поскольку тип "Float" и тип "**Integer**" — различные типы, то Ада не допускает смешивания величин этих типов в одном выражении. Однако, встречаются случаи, когда нам необходимо комбинировать значения этих типов. В таких ситуациях нам необходимо производить преобразование значений одного численного типа в значения другого численного типа.

Ада позволяет явно указывать необходимость преобразования значения типа "Float"в значение типа "**Integer**", и наоборот. Для выполнения такого преобразования используется синтаксис подобный синтаксису вызова функции:

```
X : Integer := 4;
Y : Float;

Y := Float(X);

   . . .

X := Integer(Y);
```

В этом случае компилятор, во время трансляции, добавит необходимый код для преобразования типов. Такое преобразование типов всегда возможно, и будет успешным если значение результата не будет нарушать границ допустимого диапазона значений.

Следует заметить, что при преобразовании вещественного значения "Float" в целое значение "**Integer**"Ада использует традиционные для математики правила округления, то есть:

Значение "Float"	Округленное значение "**Integer**"
1.5	2
1.3	1
-1.5	-2
-1.3	-1

3.5 Перечислимые типы

До настоящего момента были рассмотрены типы данных которые представляли численные значения (целые и вещественные числа). Однако, при решении многих практических задач, важное значение имеет понятие перечислимого типа. Перечислимыми типами называют такие типы данных значения которых перечислены, то есть представлены списком некоторых значений. Перечислимый тип полезен когда необходимо представить фиксированное множество значений, которые не являются числами. Примером может служить представление дней недели или месяцев в году.

3.5.1 Описание перечислимого типа

Перечислимый тип описывается путем предоставления списка всех возможных значений данного типа в виде идентификаторов (другими словами, перечислением всех возможных значений данного типа). Например:

```
type Computer_Language is (Assembler, Cobol, Lisp, Pascal, Ada);
type C_Letter_Languages is (Cobol, C);
```

После такого описания типов, константы и переменные этих типов могут быть описаны следующим образом:

```
A_Language      : Computer_Language;
Early_Language  : Computer_Language := Cobol;
First_Language  : constant Computer_Language := Assembler;
Example         : C_Letter_Language := Cobol;
```

Необходимо заметить, что порядок перечисления значений типа, при описании перечислимого типа, имеет самостоятельное значение — он устанавливает отношение порядка следования значений перечислимого типа, которое используется атрибутами "'First", "'Last", "'Pos", "'Val", "'Pred", "'Succ" (см "Атрибуты типов"), а также при сравнении величин перечислимого типа. Так, для типа "Computer_Language", описанного в примере выше, значение "Assembler" будет меньше чем значение "Cobol".

В одной программе допускается использование одного и того же перечислимого литерала в описаниях различных перечислимых типов. Так, в показанном выше примере, литерал "Cobol" встречается при описании двух разных типов: "Computer_Language" и "C_Letter_Languages". Такие литералы называют **совмещенными** (или перегруженными). При этом, Ада, в большинстве случаев, распознает одинаковые перечислимые литералы различных перечислимых типов. В случаях, когда это не возможно — необходимо использовать квалификацию типов.

Рассмотрим следующий пример:

```
type Primary is (Red, Green, Blue);
type Rainbow is (Red, Yellow, Green, Blue, Violet);
...
for I in Red..Blue loop ...        -- это двусмысленно
```

Здесь, компилятор не может самостоятельно определить к какому из двух типов ("Primary" или "Rainbow") принадлежит диапазон значений "Red..Blue" переменной цикла "I"(литералы "Red" и "Blue" — совмещены). Поэтому, в подобных случаях, нам необходимо точно указывать требуемый тип, используя квалификацию типа:

```
for I in Rainbow'(Red)..Rainbow'(Blue) loop ...
for I in Rainbow'(Red)..Blue loop ...      -- необходима только одна квалификация
for I in Primary'(Red)..Blue loop ...
```

Квалификация типа не изменяет значения типа и не выполняет никаких преобразований типа. Она только информирует компиляор о том какой тип, в данном случае, подразумевает программист.

При описании перичислимых типов, для указания значений перечислимого типа также как и символические имена, допускается использовать символьные константы. В этом случае перечислимый тип будет символьным (см. "Символьные типы Ады").

В заключение обсуждения описания перечислимых типов, заметим, что перечислимые и целочисленные типы называют **дискретными типами**, потому что они описывают множества упорядоченных дискретных значений. Вещественные числа не могут считаться дискретными поскольку между двумя любыми вещественными числами располагается бесконечное множество вещественных чисел (естественно, теоретически).

3.5.2 Предопределенный логический тип `Boolean`

В Аде, предопределенный логический тип "**Boolean**" описывается как перечислимый тип в пакете *Standard*:

```
type Boolean is (False, True);
```

Таким образом, переменные логического типа "**Boolean**" могут принимать только одно из двух значений: "**True**" (истина) или "**False**" (ложь).

Примечательно, что предопределеннный логический тип "**Boolean**" имеет специальное предназначение. Значения этого типа используются в условных инструкциях языка Ада ("**if** ... ", "**exit when** ... ", . . .). Это подразумевает, что если вы пытаетесь описать свой собственный логический тип "**Boolean**", и описываете его точно также как и предопределенный тип "**Boolean**" (полное имя предопределенного логического типа — "Standard.**Boolean**"), то вы получаете абсолютно самостоятельный тип(!). В результате, вы не можете использовать значения, описанного вами типа "**Boolean**", в условных инструкциях языка Ада, которые ожидают только тип "Standard.**Boolean**".

Значения предопределенного логического типа "Standard.**Boolean**"возвращают знаки операций сравнения:

```
=     -- равно
/=    -- не равно
<     -- меньше
<=    -- меньше или равно
>     -- больше
>=    -- больше или равно
```

Для обработки значений типа "**Boolean**" могут быть использованы следующие знаки операций:

```
and          -- логическое И:    вычисляются и левая, и правая часть выражения
and then     -- логическое И:    правая часть выражения вычисляется, если
             --                  результат вычисления левой части — True

or           -- логическое ИЛИ:  вычисляются и левая, и правая часть выражения
or else      -- логическое ИЛИ:  правая часть выражения вычисляется, если
             --                  результат вычисления левой части — False

xor          -- исключающее ИЛИ
not          -- отрицание (инверсия); унарная операция
```

Обычно, при вычислении значений логических выражений, компилятор сам определяет последовательность вычисления каждого логического значения. При этом, производится вычисление всех логических переменных, указанных в логическом выражении.
Тем кто знаком с языком Паскаль следует заметить, что такой подход отличается от правил принятых в современных диалектах Паскаля, где, для повышения производительности, определение значения результата всего логического выражения может быть выполнено сокращенно, в зависимости от предварительных результатов обработки выражения.
Например, при определении значения результата следующего логического выражения

```
(B /= 0) and (A/B > 0)
```

в случае, когда значение "B" равно нулю будет возникать ошибка деления на ноль. Причина в том, что значение части выражения, расположенной справа от "**and**", вычисляется всегда, не зависимо от значения результата полученного при вычислении "(B /= 0)".

Чтобы избежать подобных ошибок, а также в целях увеличения производительности, необходимо производить вычисление значений логических переменных в определенной последовательности и прекращать ее как только результат всего выражения уже определен. Для этого можно использовать "**and then**" вместо "**and**", и "**or else**" вместо "**else**", указывая порядок обработки логического выражения явно:

```
(B /= 0) and then (A/B > 0)
```

В этом случае, обработка выражения справа от "**and then**" будет производиться только в случае когда "B" не равно нулю, т.е. результат слева от "**and then**" — "**True**".

Можно переписать предыдущий пример с использованием "**or else**". Тогда, обработка логического выражения будет завершена в случае если значение слева от "**or else**" вычислено как "**True**":

```
(B = 0) or else (A/B <= 0)
```

В заключение обсуждения логического типа отметим, что Ада не позволяет одновременное использование "**and**" ("**and**" или "**and then**"), "**or**" ("**or**" или "**or else**") и "**xor**"в одном выражении не разделенном скобками. Это уменьшает вероятность разночтения содержимого сложного логического выражения. Например:

```
(A < B) and (B > C) or (D < E)      -- запрещено
((A < B) and (B > C)) or (D < E)    -- разрешено
```

3.5.3 Символьные типы Ады (`Character`, `Wide_Character`)

Ада имеет два предопределенных перечислимый типа, "Character" и "Wide_Character", для поддержки обработки символьных значений. Согласно стандарта, любой перичислимый тип который содержит хотя бы один символьный литерал является символьным типом.

Оригинальный стандарт Ada83 описывал 7-битный тип "Character". Еще до появления стандарта Ada95, это ограничение было ослаблено, но оставалось принудительным для старых компиляторов (например таких как компилятор *Meridian Ada*). Это создавало трудности при попытках отобразить графические символы на PC, поскольку для отображения символов с кодами большими чем ASCII-127 приходилось использовать целые числа. Такая поддержка обеспечивалась за счет специальных подпрограмм предоставляемых разработчиками соответствующего компилятора.

В настоящее время, предопределенный символьный тип "Character" предусматривает 256 различных символьных значений (то есть, является 8-битным), и основывается на стандарте ISO-8859-1 (Latin-1).

Некоторые символы не имеют непосредственно печатаемого значения (первые 32 символа). Такие символы используются в качестве управляющих (примером может служить символ CR — возврат каретки). Для обращения к таким символам можно использовать пакет *ASCII*, который является дочерним пакетом пакета *Standard* (благодаря этому, нет необходимости указывать пакет *ASCII* в спецификаторах контекста "**with**" и/или "**use**"). Например, для обращения к символу возврат каретки можно использовать: "ASCII.CR". Однако, пакет *ASCII* содержит только первые 128 символов и считается устаревшим, и возможно, что в будущем он будет удален. Поэтому, вместо старого пакета *ASCII* рекомендуется использовать пакет *Ada.Characters.Latin_1*, который предоставляет 256 символов. Следовательно, используя пакет *Ada.Characters.Latin_1*, к символу возврата каретки можно обратиться следующим образом: "Ada.Characters.Latin_1.CR".

Предопределенный символьный тип "Wide_Character"основывается на стандарте ISO-10646 Basic Multilingual Plane (BMP) и предусматривает 65336 различных символьных значений (использует 16 бит).

Также, Ада предоставляет пакет *Ada.Characters.Handling*, предлагающий набор полезных подпрограмм символьной обработки.

Система компилятора *GNAT* предоставляет дополнительный пакет *Ada.Characters.Wide_Latin_1*, который описывает символьные значения типа "Wide_Character" соответствующие кодировке Latin_1.

Таким образом, для работы с символьными значениями, в Аде представлены следующие пакеты:

```
Standard.ASCII                -- предоставляет только первые 128 символов
                              -- (считается устаревшим)

Ada.Characters                --
Ada.Characters.Latin_1        -- предоставляет 256 символов ISO-8859-1 (Latin-1)
Ada.Characters.Handling       -- предоставляет подпрограммы символьной обработки

Ada.Characters.Wide_Latin_1   -- дополнительный пакет из поставки
                              -- системы компилятора GNAT
```

Следует заметить, что пакет *ASCII* считается устаревшим и его не рекомендуется использовать при написании новых программ. Вместо него необходимо использовать стандартный пакет *Ada.Characters.Latin_1*.

Также следует заметить, что стандарт не требует полноценного обеспечения поддержки национальных кодировок, которые отличны от кодировки Latin-1. Возможность предоставления такой поддержки возлагается на разработчиков соответствующего компилятора. Другими словами, в настоящее время, обеспечение поддержки кириллических кодировок не регламентируется стандартом.

3.6 Типы и подтипы

Как уже говорилось, концепция типа является в Аде основополагающим фактором создания надежного программного обеспечения. Предположим, что у нас есть два целочисленных типа, которые как-то характеризуют "звоночки"и "свисточки", и мы никак не хотим смешивать эти понятия. Нам необходимо, чтобы компилятор имел возможность предупредить нас: "Ба, да вы пытаетесь смешать выши "звоночки"и "свисточки"! Что Вы в действительности подразумеваете?". Это выглядит излишними осложнениями для людей,

которые используют другие языки программирования со слабой типизацией данных. Размышления над тем какие данные необходимо представить, описание различных типов данных и правил конвертирования типов требуют некоторых усилий. Однако, такие усилия оправдываются тем, что как только это сделано, компилятор сможет помочь отыскать разные глупые ошибки.

Бывают случаи, когда нам необходимо указать, что какие-то переменные могут хранить только какое-то ограниченное число значений предоставляемых определенным типом данных. При этом, мы не хотим описывать самостоятельный новый тип данных, поскольку это потребует явного указания правила преобразования типа при взаимодействии со значениями, которые имеют оригинальный тип данных, то есть нам необходима возможность смешивать различные типы, которые, по своей сути, являются одинаковыми сущностями, но при этом имеют некоторые различия в своих характеристиках. К тому же, мы хотим сохранить преимущества проверки корректности наших действий, которые предоставляет компилятор. В таких случаях, Ада позволяет нам описывать подтипы ("**subtype**") существующих типов данных (это соответствует типам определяемым пользователем в Паскале).

Общий синтаксис объявления подтипа имеет вид:

```
subtype Name_1 is Base_Type;  -- в данном случае, Name_1 является
                              -- синонимом типа Base_Type

subtype Name_2 is Base_Type range Lowerbound..Upperbound;
```

Примеры объявления подтипов приводятся ниже:

```
type Processors is (M68000, i8086, i80386, M68030, Pentium, PowerPC);
subtype Old_Processors is Processors range M68000..i8086;
subtype New_Processors is Processors range Pentium..PowerPC;

subtype Data is Integer;
subtype Age is Data range 0..140;
subtype Temperatures is Float range -50.0..200.0;
subtype Upper_Chars is Character range 'A'..'Z';
```

Подтип, по своей сути, является той же самой сущностью, что и оригинальный тип, но при этом он может иметь ограниченный диапазон значений оригинального типа. Значения подтипа могут использоваться везде, где могут использоваться значения оригинального типа, а также значения других подтипов этого оригинального типа При этом, любая попытка присваивания переменной такого подтипа значения выходящего за границы указанного для этого подтипа диапазона допустимых значений будет приводить к исключительной ситуации *Constraint_Error* (проще говоря — ошибке программы). Такой подход облегчает обнаружение ошибок, а также, позволяет отделить обработку ошибок от основного алгоритма программы.

```
My_Age  : Age;
Height  : Integer;

Height := My_Age;   -- глупо, но никогда не вызывает проблем

My_Age := Height;   -- может вызвать проблемы, когда значение типа Height
                    -- будет за пределами диапазона значений My_Age (0..140),
                    -- но при этом остается совметимым
```

Чтобы избежать генерацию исключительной ситуации, можно использовать проверки принадлежности диапазону ("**in**" и/или "**not in**"). Например:

```
I : Integer;
N : Natural;

    ...

if  I in Natural  then
    N := I
else
    Put_Line ("I can't be assigned to N!");
end if;

    ...
```

Реально, все типы Ады являются подтипами анонимных типов, рассматриваемых как их **базовые типы**. Поскольку базовые типы анонимны, то на них нельзя ссылаться по имени. При этом, для получения базового типа можно использовать атрибут "'Base". Например, "**Integer**'Base" — это базовый тип для "**Integer**". Базовые типы могут иметь или могут не иметь диапазон значений больший чем их подтипы. Это имеет значение только в выражениях вида "A * B / C" которые, при вычислении промежуточных значений, используют базовый тип. То есть, результат "A * B" может выходить за пределы значений типа не приводя к генерации исключительной ситуации если общий результат вычисленного значения всего выражения будет находиться в допустимом диапазоне значений для данного типа.

Таким образом, необходимо заметить, что проверка допустимости диапазона производится только для значений подтипов, а для значений базовых анонимных типов такая проверка не производится. При этом, чтобы результат вычислений был математически корректен, всегда производится проверка на переполнение.

3.7 Производные типы

Мы уже рассмотрели, что подтипы остаются совместимыми со своими базовыми типами. Однако, нам может понадобиться создать абсолютно новый тип, который не будет ассоциироваться ни с каким другим типом вообще, в том числе и с оригинальным типом. Такая концепция типа сильно отличается от того, что принято в других языках программирования, и, даже, в прародителе Ады — Паскале.

Для выполнения поставленной задачи, мы производим новый тип от другого типа, используя подобный синтаксис:

```
type Child_Type is new Parent_Type;
```

Такое описание создает новый тип данных — "Child_Type", при этом "Parent_Type" — это тип-предок для типа "Child_Type", а тип "Child_Type" — это производный тип от типа "Parent_Type".

В данном случае, тип "Child_Type" будет обладать такими же характеристиками что и его тип-предок "Parent_Type": у него такой же диапазон допустимых значений как и у типа-предка, для него допустимы те же операции, которые допустимы для типа-предка (говорят, что тип "Child_Type" унаследовал операции от типа-предка "Parent_Type").

Однако, в этом случае, производный тип "Child_Type" — это абсолютно самостоятельный и независимый тип данных. Он не совместим ни с каким другим типом, включая тип "Parent_Type", от которого он был произведен, и другими типами, производными от типа "Parent_Type". Это подразумевает, что при необходимости комбинирования значений типа "Child_Type"со значениями типа "Parent_Type" требуется выполнять преобразование типов. В Аде, разрешается выполнять преобразование значений производного типа в значения типа-предка и наоборот.

Для того, чтобы преобразовать значение одного типа в значение другого типа необходимо указать имя типа к которому требуется выполнить преобразование:

```
Parent  : Parent_Type;
Child   : Child_Type := Child_Type (Parent);  -- конвертирует значение Parent,
                                              -- имеющее тип Parent_Type
                                              -- в значение типа Child_Type
```

Описание производного типа может указывать ограничение диапазона значений типа-предка, например:

```
type Child_Type is new Parent_Type range Lowerbound..Upperbound;
```

В этом случае диапазон значений производного типа "Child_Type" будет ограничен значениями нижней границы диапазона ("Lowerbound") и верхней границы диапазона ("Upperbound").

Механизм производства новых типов данных из уже существующих типов позволяет создавать целые семейства родственных типов, которые обычно называют классами. Так, например, тип "**Integer**" принадлежит к целому классу целочисленных типов. Класс целочисленных типов является, в свою очередь, подмножеством более обширного класса дискретных типов.

Основной смысл использования производных типов заключается в том, что для определенного типа данных уже существует определенный набор примитивных операций и этот набор операций можно унаследовать от такого типа при производстве от него нового типа данных.

Можно создать новый тип данных и затем описать для него идентичный набор операций. Однако, при производстве нового типа данных от уже существующего типа, производный тип автоматически наследует версии

примитивных операций, описанные для типа-предка. Таким образом, нет необходимости переписывать код заново.

Например, класс дискретных типов предусматривает атрибут "'First", который наследуется всеми дискретными типами. Класс целочисленных типов добавляет к унаследованным от класса дискретных типов операциям знак операции арифметического сложения "+". Эти механизмы более полно рассматриваются при обсуждении тэговых типов.

Производные типы используются в случаях когда моделирование определенного объекта предполагает, что тип-предок не соответствует нашим потребностям, или тогда, когда мы желаем разделить объекты в различные, несмешиваемые классы.

```
type Employee_No is new Integer;
type Account_No  is new Integer range 0..999_999;
```

Здесь "Employee_No" и "Account_No" различные и не смешиваемые типы, которые нельзя комбинировать между собой без явного использования преобразования типов. Производные типы наследуют все операции объявленные для базового типа. Например, если была объявлена запись которая имела процедуры "Push" и "Pop", то производный тип автоматически унаследует эти процедуры.

Другое важное использование производных типов — это создание переносимого кода. Ада разрешает нам создавать новые уровни абстракции, на один уровень выше чем, скажем, абстракция целого числа на последовательности битов.

Это определяется использованием производных типов, без типа-предка.

```
type Name is range некоторый_диапазон_значений;
```

Например,

```
type Data is range 0..2_000_000;
```

В этом случае ответственность, за выбор подходящего размера для целого, ложится на компилятор. Таким образом, для PC, этот размер может быть 32 бита, что эквивалентно типу "Long_Integer", а для рабочей станции Unix, этот размер можт остаться равным 32-битному целому, но это будет эквивалентно типу "**Integer**". Такое предоставление компилятору возможности выбирать освобождает программиста от обязанности выбирать. Поэтому перекомпиляция программы на другой машине не требует изменения исходного текста.

3.8 Атрибуты

Ада предусматривает специальный класс предопределенных операций, которые позволяют в удобной форме определять и использовать различные характеристики типов и экземпляров объектов. Они называются атрибутами.

Указание имени требуемого атрибута производится в конце имени экземпляра объекта (переменной, константы) или имени типа с использованием символа одинарных кавычек.

имя_типа'***имя_атрибута***
имя_экземпляра_объекта'***имя_атрибута***

Некоторыми из атрибутов для дискретных типов являются:

```
type Processors is (M68000, i8086, i80386, M68030, Pentium, PowerPC);

Integer'First              -- наименьшее целое Integer
Integer'Last               -- наибольшее целое Integer
Processors'Succ(M68000)    -- последующее за M68000 в типе
Upper_Chars'Pred('C')      -- предшествующее перед 'C' в типе ('B')
Integer'Image(67)          -- строка " 67"
                           -- пробел для '-'
Integer'Value("67")        -- целое значение 67
Processors'Pos(M68030)     -- позиция M68030 в типе
                           -- (3, первая позиция - 0)
```

Простым примером использования атрибутов, — для улучшения переносимости программы, — может служить следующий пример описания подтипа "Positive", предоставляющего положительные целые числа:

```
subtype Positive is Integer range 1..Integer'Last;
```

Здесь мы получаем размер максимального положительного целого не зависимо от любых системно зависимых свойств.

В Ada83 для не дискретных типов, — таких как "Float", "Fixed", всех их подтипов и производных от них типов, — концепции "'Pred" и "'Succ" - не имеют смысла. В Ada95 — они присутствуют. Все другие атрибуты скалярных типов также справедливы и для вещественных типов.

Глава 4

Управляющие структуры

Управляющие структуры любого языка программирования предназначены для описания алгоритмов программ, или, другими словами, для описания последовательности тех действий которые выполняет программа во время своей работы.

Управляющие структуры языка Ада по своему стилю и назначению подобны тем конструкциям, которые встречаются в большинстве других современных языках программирования. Однако, при этом присутствуют и некоторые отличия.

Как и язык в целом, управляющие структуры языка Ада, были разработаны так, чтобы обеспечить максимальную читабельность. Все управляющие структуры языка Ада можно разделить на простые и составные инструкции. Все инструкции языка Ада должны завершаться символом точки с запятой. Все составные инструкции завершаются конструкцией вида:

end *что-нибудь* ;

4.1 Пустая инструкция

В отличие от многих других языков программирования, в Аде явно присутствует инструкция, которая не вызывает каких-либо действий. Такая инструкция называется пустой, и она имеет следующий вид:

```
null;
```

Пустая инструкция используется в тех случаях когда не требуется выполнение каких-либо действий, но согласно синтаксиса языка должна быть записана хотя бы одна инструкция.

4.2 Инструкция присваивания

Инструкция присваивания используется в Аде для установки и изменения значений переменных. Она использует операцию присваивания, и в общем случае имеет следующий вид:

```
result := expression;
```

Операция присваивания ":=" разделяет инструкцию присваивания на левую и правую части (между символами двоеточия и знак равенства, пробелы — не допустимы!). В левой части записывается имя переменной (*result*) содержимому которой будет производится присваивание нового значения. Следует заметить, что в левой части может располагаться имя только одной переменной. В правой части записывается выражение (*expression*) результат вычисления которого становится новым значением переменной *result*. Выражение *expression*, расположенное в правой части, может быть одиночной переменной или константой, может содержать переменные, константы, знаки операций и вызовы функций. Тип переменной определяемой как *result* должен быть совместим по присваиванию с типом результата вычисления выражения *expression*.

Выполнение присваивания разделяется на несколько действий. Сначала производится вычисление имени переменной *result* и результата выражения *expression* (порядок следования этих действий не регламентирован стандартом языка). После этого, в случае успеха, для переменных скалярных типов проверяется принадлежность значения результата вычисления выражения *expression* подтипу переменной. Если проверка успешна,

то значение результата вычисления выражения *expression* становится новым значением содержимого переменной *result*. При этом старое значение содержимого *result* — теряется. Иначе, в случае какой-либо неудачи, возбуждается исключение ошибки ограничения или, проще говоря, — ошибка, а значение переменной *result* остается без изменений.

Приведем несколько примеров инструкций присваивания:

```
A := B + C
X := Y
```

Следует заметить, что в Аде, в результате выполнения присваивания производится изменение только содержимого *result* (значение содержимого переменной указанной в левой части). Необходимо также подчеркнуть, что операция присваивания в Аде, в отличие от языков C/C++, не возвращает значение и не обладает побочными эффектами. Кроме того, напомним, что операция присваивания, в Аде, не допускает совмещение, переименование и использование псевдонимов, а также она запрещена для лимитированных типов.

4.3 Блоки

Блок содержит последовательность инструкций, перед которой может располагаться раздел описаний (все описания локальны для блока и не доступны вне блока). За последовательностью инструкций могут следовать обработчики исключений (обработка исключений в Аде рассматривается позже). В общем случае инструкция блока Ады имеет вид:

```
declare

    -- локальные описания

begin

    -- последовательность инструкций

exception

    -- обработчики исключений

end;
```

Блок может иметь имя. Для этого перед инструкцией блока записывается идентификатор, за которым ставится двоеточие. При именовании блока, имя блока должно указываться после "**end**" завершающего блок:

```
Some_Block:
declare
    X : Integer;
begin
    X := 222 * 333;
    Put (X);
end Some_Block;
```

4.4 Условные инструкции `if`

Для организации условного выполнения последовательностей алгоритмических действий (то есть, построения разветвляющихся алгоритмов), в Аде могут использоваться условные инструкции "**if**".

Каждая инструкция "**if**" заканчивается конструкцией "**end if**".

```
if логическое_выражение then

    -- последовательность инструкций

end if;
```

```
if логическое_выражение then

    -- последовательность инструкций 1

else

    -- другая последовательность инструкций 2

end if;
```

В первом примере, приведенном выше, последовательность инструкций, описывающая алгоритмические действия, будет выполнена только в случае когда результат вычисления логического выражения будет иметь значение "**True**". Во втором примере, в случае когда результат вычисления логического выражения — "**True**"будет выполняться "*последовательность инструкций 1*", в противном случае — "*последовательность инструкций 2*".

Для сокращения инструкций вида "**else if** ... ", и в целях улучшения читабельности, введена конструкция "**elsif**", которая может быть использована столько раз, сколько это будет необходимо.

```
if логическое_выражение then

    -- последовательность инструкций 1

elsif логическое_выражение then

    -- последовательность инструкций 2

elsif логическое_выражение then

    -- последовательность инструкций 3

else

    -- последовательность инструкций

end if;
```

В этой форме инструкции "**if**", заключительная конструкция "**else**" — опциональна.

Необходимо также заметить, что результат вычисления логического выражения всегда должен иметь предопределенный тип "Standard.**Boolean**".

4.5 Инструкция выбора case

Еще одним средством позволяющим строить разветвляющиеся алгоритмы является инструкция выбора "**case**".

Инструкция выбора "**case**" должна предусматривать определенное действие для каждого возможного значения переменной селектора (переключателя). В случаях, когда невозможно перечислить все значения переменной селектора, нужно использовать метку "**others**".

Каждое значение выбора может быть представлено как одиночное значение (например, 5), как диапазон значений (например, 1..20), или как комбинация, состоящая из одиночных значений и/или диапазонов значений, разделенных символом "'|".

Каждое значение выбора должно быть статическим значением, то есть оно должно быть определено компилятором во время компиляции программы.

```
case выражение is
    when значение_выбора => действия
    when значение_выбора => действия
      . . .
    when others => действия
end case;
```

Важные примечания:

- "*выражение*", в инструкции "**case**", должно быть дискретного типа
- метка "**others**" обязательна в инструкции "**case**" тогда, когда инструкции "**when**" не перечисляют всех возможных значений селектора.

```
case Letter is
    when 'a'..'z'| 'A'..'Z' => Put ("letter");
    when '0'..'9'           => Put ("digit! Value is"); Put (letter);
    when ''' | '"' | '`'    => Put ("quote mark");
    when '&'                => Put ("ampersand");
    when others             => Put ("something else");
end case;
```

В некоторых случаях, в качестве действий, указываемых для метки "**others**", может использоваться пустая инструкция "**null**":

```
. . .
when others  => null;  -- ничего не делать
. . .
```

4.6 Организация циклических вычислений

При решении реальных задач часто возникает необходимость в организации циклических вычислений. Все конструкции организации циклических вычислений в Аде имеют форму "**loop** ... **end loop**" с некоторыми вариациями. Для выхода из цикла может быть использована инструкция "**exit**".

4.6.1 Простые циклы (`loop`)

Примером простейшего цикла может служить бесконечный цикл. Обычно он используется совместно с инструкцией "**exit**", рассматриваемой позже.

```
loop

    -- инструкции тела цикла

end loop;
```

4.6.2 Цикл `while`

Во многих случаях, прежде чем выполнять действия которые описываются инструкциями тела цикла, необходимо проверить какое-либо условие. Для таких случаев Ада предусматривает конструкцию цикла "**while**".

Цикл "**while**" идентичен циклу "**while**" в языке Паскаль. Проверка условия выполнения цикла производится до входа в блок инструкций составляющих тело цикла. При этом, если результат вычисления логического выражения будет "**True**", то будет выполнен блок инструкций тела цикла. В противном случае, тело цикла - не выполняется.

```
while логическое_выражение loop

    -- инструкции тела цикла

end loop;
```

Необходимо заметить, что результат вычисления логического выражения должен иметь предопределенный тип "Standard.**Boolean**".

4.6.3 Цикл `for`

Еще одним распространенным случаем является ситуация когда необходимо выполнить некоторые действия заданное количество раз, то есть организовать счетный цикл. Для этого Ада предусматривает конструкцию цикла "**for**".

Конструкция цикла "**for**" Ады аналогична конструкции цикла "**for**", представленной в языке Паскаль.

Существует несколько правил использования цикла "**for**":

- тип переменной-счетчика цикла "**for**" определяется типом указываемого диапазона значений счетчика, и должен быть дискретного типа, вещественные значения — недопустимы
- счетчик не может быть модифицирован в теле цикла, другими словами — счетчик доступен только по чтению
- область действия переменной-счетчика распространяется только на тело цикла

Примечательно также, что тело цикла не будет выполняться если при указании диапазона значений переменной-счетчика величина значения "нижней границы"будет больше чем величина значения "верхней границы".

```
for счетчик in диапазон_значений_счетчика loop

    -- инструкции тела цикла

end loop;
```

```
for  Count in 1..20 loop
    Put (Count);
end loop;
```

Возможен перебор значений диапазона в обратном порядке:

```
for счетчик in reverse диапазон_значений_счетчика loop

    -- инструкции тела цикла

end loop;
```

```
for Count in reverse 1..20 loop
    Put (Count);
end loop;
```

Любой дискретный тип может использоваться для указания диапазона значений переменной-счетчика.

```
declare
    subtype List is Integer range 1..10;

begin
    for Count in List loop
        Put (Count);
    end loop;
end;
```

Здесь, тип "List" был использован для указания диапазона значений переменной-счетчика "Count". Подобным образом также можно использовать любой перечислимый тип.

4.6.4 Инструкции `exit` и `exit when`

Инструкции "**exit**" и "**exit when**" могут быть использованы для преждевременного выхода из цикла. При этом, выполнение программы будет продолжено в точке непосредственно следующей за циклом. Два варианта, показанных ниже, имеют одинаковый эффект:

```
loop

    -- инструкции тела цикла

    if логическое_выражение then
        exit;
    end if;
end loop;


loop

    -- инструкции тела цикла

    exit when логическое_выражение;
end loop;
```

4.6.5 Именованые циклы

Инструкции преждевременного выхода из цикла "**exit**" и "**exit when**", обычно, осуществляют выход из того цикла, который непосредственно содержит данную инструкцию. Однако, мы можем именовать циклы и модифицировать инструкцию выхода из цикла так, чтобы осуществлять выход сразу из всех вложенных циклов. Во всех случаях, следующая выполняемая инструкция будет следовать сразу за циклом из которого был осуществлен выход.

```
outer_loop:
loop

    -- инструкции

    loop

        -- инструкции

        exit outer_loop when логическое_выражение;
    end loop;
end loop outer_loop;
```

Примечательно, что в случае именованого цикла "**end loop**" также необходимо именовать меткой.

4.7 Инструкция перехода `goto`

Инструкция перехода "**goto**" предусмотрена для использования в языке Ада, в исключительных ситуациях, и имеет следующий вид:

```
goto Label;

<<Label>>
```

Использование инструкции "**goto**" очень ограничено и четко осмысленно. Вы не можете выполнить переход внутрь условной инструкции "**if**", внутрь цикла ("**loop**"), или, как в языке Паскаль, за пределы подпрограммы.

Вообще, при таком богатстве алгоритмических средств Ады, использование "**goto**" едва-ли можно считать оправданным.

Глава 5

Массивы (*array*)

Понятие массива подразумевает один из механизмов структурирования данных и является одним из способов построения составных типов данных. В сущности, массив — это набор данных идентичного типа. Как правило, массиву дается какое-то имя, которое будет обозначать весь набор данных, и механизм индексации, позволяющий обращаться к индивидуальным элементам набора. На сегодняшний день, большинство языков программирования предусматривают возможность работы с различного типа массивами.

Ада предоставляет массивы подобно языку Паскаль, и, при этом, добавляет некоторые новые полезные особенности. Неограниченные и динамические массивы, атрибуты массивов — вот некоторые из предлагаемых расширений.

5.1 Простые массивы

5.1.1 Описание простого массива

В общем случае, при объявлении массива, сначала производится описание соответствующего типа. Затем, экземпляр массива может быть создан используя описание этого типа.

```
type Stack is array (1..50) of Integer;
Calculator_Workspace : Stack;

type Stock_Level is Integer range 0..20_000;
type Pet is (Dog, Budgie, Rabbit);
type Pet_Stock is array (Pet) of Stock_Level;

Store_1_Stock : Pet_Stock;
Store_2_Stock : Pet_Stock;
```

В приведенном выше примере, тип "Stack" — это массив из 50-ти целочисленных элементов типа "**Integer**", а "Calculator_Workspace" — это переменная типа "Stack". Еще одним описанием массива является тип "Pet_Stock". При этом, тип "Pet_Stock" — это массив элементов типа "Stock_Level", а для индексирования элементов массива "Stock_Level" используется перечислимый тип "Pet". Переменные "Store_1_Stock" и "Store_2_Stock" - это переменные типа "Pet_Stock".

Общая форма описания массива имеет следующий вид:

type *имя_массива* **is array** (*спецификация_индекса*) **of** *тип_элементов_массива*;

Необходимо заметить:

- спецификация индекса может быть типом (например, "Pet")
- спецификация индекса может быть диапазоном (например, "1..50")
- значения индекса должны быть дискретного типа

5.1.2 Анонимные массивы

Массив может быть объявлен непосредственно, без использования предопределенного типа:

```
No_Of_Desks : array (1..No_Of_Divisions) of Integer;
```

В этом случае массив будет называться анонимным (поскольку он не имеет явного типа) и он будет несовместим с другими массивами — даже такими, которые описаны таким же самым образом. Кроме того, такие массивы не могут быть использованы как параметры подпрограмм. В общем случае рекомендуется избегать использования анонимных массивов.

5.1.3 Организация доступа к отдельным элементам массива

Организацию доступа к отдельным элементам массива проще всего продемонстрировать на простых примерах. Так для обращения к значению элемента массива "Store_1_Stock", описанного ранее, можно использовать:

```
if Store_1_Stock (Dog) > 10 then ...
```

В приведенном примере производится чтение значения элемента массива "Store_1_Stock"(доступ по чтению).

Для сохранения значения в элементе массива "Store_2_Stock" (доступ по записи) можно использовать:

```
Store_2_Stock (Rabbit) := 200;
```

Необходимо отметить, что в обоих случаях доступ к элементу массива в Аде внешне никак не отличается от вызова функции.

5.1.4 Агрегаты для массивов

В общем случае, агрегат массива — это совокупность значений для каждого элемента массива. Использование агрегатов позволяет выполнять одновременное присваивание значений всем элементам массива в эффективной и элегантной форме.

Рассмотрим следующий пример:

```
Store_1_Stock := (5, 4, 300);
```

В данном случае, присваивание значений элементам массива "Store_1_Stock"выполняется с помощью агрегата. Следует учесть, что в этом примере значения в агрегате присваиваются в порядке соответствующем следованию элементов в массиве. Такая нотация называется **позиционной** или **неименованой**, а такой агрегат - **позиционный** или **неименованый агрегат**.

Кроме позиционной нотации, возможно использование **именованой** нотации. В этом случае именуется каждый индивидуальный элемент массива. Используя именованую нотацию, предыдущий пример можно переписать следующим образом:

```
Store_1_Stock := (Dog => 5, Budgie => 4, Rabbit => 300);
```

Такой вид агрегата называют **именованым агрегатом**.

Приведем еще один пример именованого агрегата:

```
Store_1_Stock := (Dog | Budgie => 0, Rabbit => 100);
```

В пределах одного агрегата, Ада допускает использовать только один вид нотации. Это означает, что комбинирование позиционной и именованой нотации в одном агрегате — не допустимо и будет вызывать ошибку компиляции. Например:

```
Store_1_Stock := (5, 4, Rabbit => 300);     -- это недопустимо!
```

В агрегате может указываться диапазон дискретных значений:

```
Store_1_Stock := (Dog..Rabbit => 0);
```

Агрегаты обоих видов удобно использовать в описаниях:

```
Store_1_Stock:  Pet_Stock := (5, 4, 300);
```

С агрегатами массивов разрешается использование опции "**others**", которая практически полезна при установке всех элементов массива в какое-либо предопределенное значение. Стоит учесть, что в таких случаях часто требуется квалификация типа.

```
New_Shop_Stock  : Pet_Stock := (others := 0);
```

Рассмотрим следующие описания:

```
declare
    type Numbers1 is array (1..10) of Integer;
    type Numbers2 is array (1..20) of Integer;
    A : Numbers1;
    B : Numbers2;
begin
    A := (1, 2, 3, 4, others => 5);
end;
```

Заметьте, что в данном случае опция "**others**" используется вместе с позиционной нотацией. Поэтому Ада потребует указать квалификацию типа:

```
A : = Numbers1'(1, 2, 3, 4, others => 5);
```

В общем случае, при использовании опции "**others**" совместно с любой из двух нотаций, позиционной или именованой, требуется указывать квалификацию типа.

5.1.5 Отрезки (*array slices*)

Для одномерных массивов Ада предусматривает удобную возможность указания нескольких последовательных компонент массива. Такая последовательность компонент массива называется отрезком массива (*array slice*). В общем случае, отрезок массива может быть задан следующим образом:

имя_массива (*диапазон_значений_индекса*)

Таким образом, для переменной "Calculator_Workspace" типа "Stack", рассмотренных ранее, можно указать отрезок, содержащий элементы с 5-го по 10-й, следующим образом:

```
Calculator_Workspace (5..10) := (5, 6, 7, 8, 9, 10);
```

В данном примере выполняется присваивание значений элементам массива "Calculator_Workspace", которые попадают в указанный отрезок, с использованием позиционного агрегата массива.

Приведем еще один простой пример:

```
Calculator_Workspace (25..30) := Calculator_Workspace (5 .. 10);
```

Напомним что использование отрезков допускается только для одномерных массивов.

5.1.6 Массивы-константы

Ада допускает использование массивов-констант. В таких случаях, при описании массива-константы, необходимо инициализировать значения всех его элементов. Такая инициализация, как правило, осуществляется с помощью агрегатов массивов. Например:

```
type Months is (Jan, Feb, Mar, .... , Dec);
subtype Month_Days is Integer range 1..31;
type Month_Length is array (Jan..Dec) of Month_Days;

Days_In_Month : constant Month_Length := (31, 28, 31, 30, ... , 31);
```

5.1.7 Атрибуты массивов

С массивами ассоциируются следующие атрибуты:

```
имя_массива'First      -- нижняя граница массива
имя_массива'Last       -- верхняя граница массива

имя_массива'Length     -- количество элементов в массиве
                          -- имя_массива'Last - имя_массива'First + 1

имя_массива'Range      -- подтип объявленный как
                          -- имя_массива'First..имя_массива'Last
```

Эти средства очень полезны для организации перебора элементов массивов.

```
for Count in имя_массива'Range loop

    . . .

end loop
```

В приведенном выше примере, каждый элемент массива будет гарантированно обработан.

5.2 Многомерные массивы

Ада позволяет использовать многомерные массивы. В качестве простого примера многомерного массива рассмотрим двухмерный массив целых чисел "Square":

```
Square_Size : constant := 5;
subtype Square_Index  is Integer range 1..Square_Size;
type  Square  is array (Square_Index, Square_Index) of Integer;

Square_Var  : Square := ( others => (others => 0) );
```

Здесь, агрегат, который инициализирует переменную "Square_Var" типа "Square" в нуль, построен как агрегат массива массивов, поэтому требуется двойное использование скобок (опции "**others**" использованы для упрощения примера).

Более сложный пример инициализации этой переменной, использующий агрегат с позиционной нотацией, может иметь следующий вид:

```
------------------столбцы   1   2   3   4   5

Square_Var  : Square := ( ( 1,  2,  3,  4,  5),     -- строка 1
                          ( 6,  7,  8,  9, 10),     -- строка 2
                          (11, 12, 13, 14, 15),     -- строка 3
                          (16, 17, 18, 19, 20),     -- строка 4
                          (21, 22, 23, 24, 25) );   -- строка 5
```

Доступ к элементам такого массива можно организовать следующим образом:

```
Square_Var (1, 5) := 5;
Square_Var (5, 5) := 25;
```

Возможно использование альтернативного способа для описания подобного двумерного массива. Его можно описать как массив рядов (иначе — строк), где каждый ряд является одномерным массивом целых чисел "Square_Row".

```
Square_Size : constant := 5;
subtype Square_Index  is Integer range 1..Square_Size;

type  Square_Row is array (Square_Index) of Integer;
type  Square     is array (Square_Index) of Square_Row;

Square_Var  : Square := ( others => (others => 0) );
```

Примечательно, что инициализация переменных в обоих вариантах реализации двумерного массива выполняется одинаково.

В этом случае, доступ к элементам массива можно организовать следующим образом:

```
Square_Var (1)(5) := 5;
Square_Var (5)(5) := 25;
```

С многомерными массивами можно использовать те же атрибуты, которые допустимы для простых одномерных массивов. При этом несколько изменяется форма указания атрибутов:

```
имя_массива'First (N)
имя_массива'Last (N)
имя_массива'Length (N)
имя_массива'Range (N)
```

В данном случае, значение определяемое, например, как "*имя_массива*'Range (N)" будет возвращать диапазон "N"-мерного индекса.

5.3 Типы неограниченных массивов (*unconstrained array*), предопределенный тип `String`

До настоящего момента мы рассматривали только такие массивы у которых диапазон значений индекса был заведомо известен. Такие массивы называют ограниченными массивами, и они могут быть использованы для создания экземпляров объектов с четко определенными во время описания типа границами.

В дополнение к таким типам, Ада позволяет описывать типы массивов, которые не имеют четко определенного диапазона для индексных значений, когда описание типа не определяет границ массива. Поэтому такие массивы называют неограниченными массивами (*unconstrained array*).

Типы неограниченных массивов позволяют описывать массивы, которые во всех отношениях идентичны обычным массивам, за исключением того, что их размер не указан. Ограничение массива (указание диапазона для индексных значений) производится при создании экземпляра объекта такого типа.

Таким образом, описание неограниченого массива предоставляет целый класс массивов, которые содержат элементы одинакового типа, имеют одинаковый тип индекса и одинаковое количество индексов, но при этом разные экземпляры объеков такого типа будут иметь разные размеры.

Этот механизм удобно использовать в случаях когда массив произвольного размера необходимо передать в подпрограмму как параметр.

Примером описания неограниченного массива целых чисел может служить следующее:

```
type Numbers_Array is array (Positive range <>) of Integer;
```

Символы "<>" указывают, что диапазон значений индекса должен быть указан при описании объектов типа "Numbers_Array". Переменная такого типа может быть описана следующим образом:

```
Numbers : Numbers_Array (1..5) := (1, 2, 3, 4, 5);
```

Здесь, при описании переменной "Numbers" предусматривается ограничение (*constraint*) размеров массива — указывается диапазон значений индекса — "(1..5)".

Пакет *Standard* предоставляет предопределенный тип "String", который описывается как неограниченный массив символов:

```
type String is array (Positive range <>) of Character;
```

Таким образом, тип "String" может быть использован для описания обширного класса символьных массивов, которые идентичны во всем, кроме количества элементов в массиве.

Также как и в предыдущем примере описания переменной "Numbers", для создания фактического массива типа "String", мы должны предусмотреть ограничение диапазона возможных значений индекса:

```
My_Name : String (1..20);
```

Здесь, ограничение диапазона индексных значений находится в диапазоне "1..20". Преимущество такого подхода в том, что все описанные строки имеют один и тот же тип, и могут, таким образом, использоваться как параметры подпрограмм. Такой дополнительный уровень абстракции позволяет более общее использование подпрограмм обработки строк.

Необходимо заметить, что для инициализации объектов типа "String", можно использовать агрегаты, поскольку тип "String", по сути, является массивом символов. Однако, более цивилизованным способом будет использование строковых литералов. Так, вышеописанную переменную "My_Name", можно инициализировать следующим образом:

```
My_Name := "Alex                    ";
```

Следует учесть, что в приведенном примере, строковый литерал, указывающий имя, необходимо дополнить пробелами, чтобы его размер совпадал с размером описанной переменной. В противном случае, компилятор может выдать предупреждение о возбуждении исключения *Constraint_Error* во время выполнения программы.

При описании строк, которым присваиваются начальные значения, границы диапазона можно не указывать:

```
Some_Name   : String := "Vasya Pupkin";
Some_Saying : constant String := "Beer without vodka is money to wind!";
```

Для обработки каждого элемента переменной, которая порождается при использовании типа неограниченного массива, требуется использование таких атрибутов типа массив, как "A'**Range**", "A'First" и т.д., поскольку не известно какие индексные значения будет иметь обрабатываемый массив.

```
My_Name    : String (1..20);
My_Surname : String (21..50);
```

Обычно, неограниченные массивы реализуются с объектом который хранит значения границ диапазона индекса и указатель на массив.

5.4 Стандартные операции для массивов

Существует несколько операций, которые могут выполняться не только над отдельными элементами массива, но и над целым массивом.

5.4.1 Присваивание

Целому массиву может присваиваться значение другого массива. Оба массива должны быть одного и того же типа. Если оба массива одного и того же неограниченного типа, то они должны содержать одинаковое количество элементов.

```
declare
    My_Name   : String (1..10)  := "Dale      ";
    Your_Name : String (1..10)  := "Russell   ";
    Her_Name  : String (21..30) := "Liz       ";
    His_Name  : String (1..5)   := "Tim  ";
begin
    Your_Name := My_Name;      -- это корректно, поскольку в обоих случаях
    Your_Name := Her_Name;     -- оба массива имеют одинаковое количество
                               -- элементов
    His_Name  := Your_Name;    -- это приведет к возбуждению исключения:
                               -- хотя обе переменные одного и того же типа,
                               -- но они имеют различную длину (число элементов)
end;
```

5.4.2 Проверки на равенство и на неравенство

Проверки на равенство и на неравенство доступны почти для всех типов Ады. Два массива считаются равными если каждый элемент первого массива равен соответствующему элементу второго массива.

```
if Array1 = Array2 then ...
```

5.4.3 Конкатенация

Символ "&" может быть использован как знак операции конкатенации двух массивов.

```
declare
    type Vector is array (Positive range <>) of Integer;

    A : Vector (1..10);
    B : Vector (1..5) := (1, 2, 3, 4, 5);
    C : Vector (1..5) := (6, 7, 8, 9, 10);
begin
    A := B & C;
    Put_Line ("hello" & " " & "world");
end;
```

5.4.4 Сравнение массивов

Для сравнения одномерных массивов могут быть использованы следующие знаки операций "<", "<=", ">" и ">=". Они наиболее полезны при сравнении массивов символов (строк).

```
"hello" < "world"          -- возвращает результат "истина" (TRUE)
```

В общем случае, можно сравнивать только те массивы у которых можно сравнивать между собой индивидуальные компоненты. Таким образом, например, нельзя сравнивать массивы записей для которых не определена операция сравнения. (то есть, чтобы обеспечить возможность сравнения массивов записей, необходимо, чтобы была определена операция сравнения для записи компонента массива).

5.4.5 Логические операции

Если знаки логических операций "**and**", "**or**", "**xor**", "**not**"допускается использовать для индивидуальных компонентов массива, то использование знаков логических операций для такого массива также будет допустимым (например, в случае массива логических значений типа "**Boolean**").

5.5 Динамические массивы

Ада позволяет не указывать размеры массива при написании программы. В этом случае размеры массива не фиксируются во время компиляции программы, а определяются во время ее выполнения, что во многих случаях более предпочтительно. Массивы подобного вида известны как динамические массивы. Кроме того, в отличие от многих других языков программирования, Ада позволяет использование динамических массивов в качестве значения результата, возвращаемого функцией.

```
declare
    X       : Integer := Y    -- значение Y описано где-то в другом месте
    A       : array (1..X) of Integer;
begin
    for I in A'Range loop

        . . .

    end loop;
end;


procedure Demo (Item : String) is
    Copy    : String (Item'First..Item'Last) := Item;
    Double  : String (1..2 * Item'Length) := Item & Item;
begin
    . . .
```

Следует заметить, что не стоит позволять вводу пользователя устанавливать размер массива, и приведенный пример (с декларативным блоком) не должен использоваться как способ решения этой задачи. Использование второго примера наиболее типично.

Глава 6

Записи (*record*)

Понятие записи, также как и понятие массива, является механизмом структурирования данных. Однако, в отличие от массива, запись позволяет сгруппировать в одном объекте набор объектов которые могут принадлежать различным типам. При этом объекты из которых состоит запись часто называют компонентами или полями записи.

Для работы с записями, Ада предлагает средства подобные тем, которые предоставляют другие современные языки программирования, а также дополняет их некоторыми своими особенностями. Также как и для массивов, для записей предусматривается использование агрегатов. Использование дискриминантов позволяет создавать вариантные записи, указывать размер для записи переменного размера и выполнять инициализацию компонентов записи.

6.1 Простые записи

6.1.1 Описание простой записи

Как уже было сказано, запись — это структура данных состоящая из набора различных компонентов. В Аде, для описания такой структуры данных, необходимо описать тип записи. В общем случае, описание типа записи имеет следующий вид:

```
type имя_записи is
    record
        имя_поля_1 : тип_поля_1;
        имя_поля_2 : тип_поля_2;
            . . .
        имя_поля_N : тип_поля_N;
    end record;
```

Например:

```
type Bicycle is
    record
        Frame      : Construction;
        Maker      : Manufacturer;
        Front_Brake : Brake_Type;
        Rear_Brake  : Brake_Type;
    end record;

My_Bicycle  : Bicycle;
```

Примечательно, что описание индивидуальных компонентов записи выглядит как описание переменных. Также, следует заметить, что описание типа записи не создает экземпляр объекта записи. В приведенном выше примере, тип "Bicycle" описывает структуру записи, а переменная "My_Bicycle" типа "Bicycle" — является экземпляром записи.

В отличие от массивов, Ада не позволяет создавать анонимные записи. Таким образом, следующий пример описания будет неправильным:

```
My_Bicycle  : record                                  -- использование анонимных
                                                      -- записей - ЗАПРЕЩЕНО!!!
                  Frame       : Construction;
                  Maker       : Manufacturer;
                  Front_Brake : Brake_Type;
                  Rear_Brake  : Brake_Type;
              end record;
```

Из этого следует, что сначала необходимо описать тип записи, а затем описывать объекты этого типа.

6.1.2 Значения полей записи по умолчанию

При описании записи, полям записи могут быть назначены значения по умолчанию. Эти значения будут использоваться всякий раз при создании экземпляра записи данного типа (до тех пор пока они не будут инициализированы другими значениями).

```
type Bicycle is
    record
        Frame       : Construction := CromeMolyebdenum;
        Maker       : Manufacturer;
        Front_Brake : Brake_Type   := Cantilever;
        Rear_Brake  : Brake_Type   := Cantilever;
    end record;
```

6.1.3 Доступ к полям записи

В Аде организация доступа к индивидуальным полям записи осуществляется с помощью точечной нотации, за именем переменной-записи, которое сопровождается точкой, следует имя поля записи. Например:

```
Expensive_Bike  : Bicycle;

Expensive_Bike.Frame        := Aluminium;
Expensive_Bike.Manufacturer := Cannondale;
Expensive_Bike.Front_Brake  := Cantilever;
Expensive_Bike.Rear_Brake   := Cantilever;

if Expensive_Bike.Frame = Aluminium then ...
```

Это идентично организации доступа к полям записи в таких языках программирования как Паскаль или Си.

6.1.4 Агрегаты для записей

Так же как и в случае массива, все множество значений элементов записи может присваиваться с помощью агрегата. При этом агрегат должен предоставлять значения для всех компонентов записи даже в случаях когда некоторые компоненты обеспечены значениями по умолчанию. Для записей различают агрегаты использующие **позиционную**, **именованную** и **смешанную** нотацию.

Примером использования **позиционного** агрегата может служить следующее:

```
Expensive_Bike := (Aluminium, Cannondale, Cantilever, Cantilever);
```

При позиционной нотации порядок следования присваиваемых значений в агрегате соответствует порядку следования полей в описании типа записи.

Альтернативно позиционной нотации, показанной выше, для присваивания таких же значений полям переменной "Expensive_Bike"можно применить агрегат использующий **именованную** нотацию. В этом случае поля записи могут перечисляться в произвольном порядке:

```
Expensive_Bike := (Rear_Brake   => Cantilever,
                   Front_Brake  => Cantilever,
                   Manufacturer => Cannondale,
                   Frame        => Aluminium);
```

Для записей допускается смешивать в одном агрегате оба варианта нотации. При этом все позиционные значения должны предшествовать именованным значениям. Такой вариант нотации будет **смешанным**.

Также как и в случае агрегатов для массивов, в агрегатах для записей допускается использование опции "**others**". При этом, для опции "**others**" должен быть представлен хотя бы один компонент, а в случае когда для опции "**others**" предоставляется более одного компонента, все компоненты должны иметь один и тот же тип.

Агрегаты являются удобным средством указания значений полей при описании переменных и констант:

```
Expensive_Bike :  Bicycle := (Aluminium, Cannondale, Cantilever, Cantilever);
```

Одинаковые значения могут присваиваться разным полям записи при использовании символа '|'.

```
Expensive_Bike := (Frame                    => Aluminium,
                   Manufacturer             => Cannondale,
                   Front_Brake | Rear_Brake => Cantilever);
```

В заключение обсуждения применения агрегатов для записей рассмотрим следующий обобщающий пример:

```
type Summary  is
    record
      Field_1 : Boolean;
      Field_2 : Float;
      Field_3 : Integer;
      Field_4 : Integer;
    end record;

Variable_1  : Summary := (True, 10.0, 1, 1);   -- позиционная нотация

Variable_2  : Summary := (Field_4 => 1         -- именованная нотация
                          Field_3 => 1,
                          Field_2 => 10.0,
                          Field_1 => True);

Variable_2  : Summary := (True, 10.0,          -- смешанная нотация
                          Field_4 => 1,
                          Field_3 => 1);

-----------------------использование символа '|'

Variable_4  : Summary := (True, 10.0,
                          Field_3 | Field_4 => 1);

Variable_5  : Summary := (Field_1 => True,
                          Field_2 => 10.0,
                          Field_3 | Field_4 => 1);

-----------------------использование others

Variable_6  : Summary := (True, 10.0, others => 1);

Variable_7  : Summary := (Field_1 => True,
                          Field_2 => 10.0,
                          others => 1);
```

6.1.5 Записи-константы

Записи-константы могут быть созданы также как и обычные переменные. В этом случае значения всех полей записи должны быть инициализированы с помощью агрегата или значений определенных по умолчанию.

```
My_Bicycle : constant Bicycle := (Hi_Tensile_Steel,
                                  Unknown,
                                  Front_Brake => Side_Pull,
                                  Rear_Brake  => Side_Pull);
```

Присваивать новые значения полям записи-константы или самой записи-константе **нельзя**. Необходимо также заметить, что отдельные поля записи не могут быть описаны как константы.

6.1.6 Лимитированные записи

Тип записи может быть описан как лимитированная запись.
В качестве примера, рассмотрим следующие описания:

```
type  Person  is  limited
    record
        Name    : String (1..Max_Chs);  -- строка имени
        Height  : Height_Cm := 0;       -- рост в сантиметрах
        Sex     : Gender;               -- пол
    end record;

Mike    : Person;
Corrina : Person;
```

В случае, когда тип записи является лимитированной записью, компилятор не позволяет выполнять присваивание и сравнение экземпляров этого типа записи.

```
  . . .
Mike    := Corrina;        -- ОШИБКА КОМПИЛЯЦИИ!!!
                           -- для лимитированных записей присваивание запрещено
  . . .

if  Corrina = Mike  then   -- ОШИБКА КОМПИЛЯЦИИ!!!
                           -- для лимитированных записей сравнение запрещено
    Put_Line ("This is strange");
end if;
  . . .
```

В результате, при компиляции показанного выше кода, будут выдаваться сообщения об ошибке компиляции.

6.2 Вложенные структуры

В качестве компонентов записи можно использовать составные типы. Например, полем записи может быть массив или какая-либо другая запись. Таким образом, Ада предоставляет возможность построения описаний сложных структур данных. Однако, описания таких структур данных обладают некоторыми характерными особенностями на которые необходимо обратить внимание.

6.2.1 Поля типа массив

В случаях когда какой-либо компонент записи необходимо описать как массив необходимо учесть, что такой компонент не может быть указан как анонимный массив. Это означает, что тип массива для такого компонента записи должен быть предварительно описан.

```
type  Illegal is
    record
        Simple_Field_1: Boolean;
        Simple_Field_2: Integer;
        Array_Field   : array (1..10) of Float; -- использование
                                                -- анонимного массива
                                                -- ЗАПРЕЩЕНО!!!
    end record;


type  Some_Array  is array (1..10) of Float;    -- предварительно описанный
                                                -- тип массива
type  Legal is
    record
        Simple_Field_1: Boolean;
        Simple_Field_2: Integer;
        Array_Field   : Some_Array;             -- компонент предварительно
                                                -- описанного типа массив
    end record;
```

Также следует учесть, что в качестве компонентов записей не допускается использование неограниченных массивов. Рассмотрим следующий пример:

```
type  Some_Array  is array (Integer range <>) of Float; -- неограниченный
                                                        -- массив
type  Some_Record is
    record
        Field_1: Boolean;
        Field_2: Integer;
        Field_3: Some_Array (1..10);  -- описание компонента записи
                                      -- задает ограничение индекса
    end record;
```

Здесь, тип "Some_Array" — это неограниченный массив. Поэтому, при описании поля "Field_3" записи "Some_Record"указывается ограничение значений индекса для массива — "(1..10)". После этого, компонент "Field_3" становится ограниченным массивом.

Для доступа к индивидуальному компоненту поля "Field_3"какой-либо переменной типа "Some_Record" можно использовать:

```
. . .
Some_Var :  Some_Record;
. . .
Some_Var.Field_3 (1) := 1;
```

Для инициализации всех значений какой-либо переменной типа "Some_Record"можно использовать агрегаты. В качестве демонстрации, приведем несколько примеров.

```
Some_Var_1 :  Some_Record := (False, 0, (1, 2, 3, 4, 5, 6, 7, 8, 9, 10));

Some_Var_2 :  Some_Record := (Field_1 => False,
                              Field_2 => 0,
                              Field_3 => (1, 2, 3, 4, 5, 6, 7, 8, 9, 10));

Some_Var_3 :  Some_Record := (Field_1 => True,
                              Field_2 => 10,
                              Field_3 => (others => 0));
```

Из приведенных примеров видно, что для инициализации простых полей "Field_1" и "Field_2"записей "Some_Var_1", "Some_Var_2", "Some_Var_3"типа "Some_Record" используются обычные литералы соответствующего типа, а для инициализации поля "Field_3", которое является массивом, используется агрегат массива. Таким образом, для инициализации подобных структур необходимо использовать вложенные агрегаты.

6.2.2 Поля записей типа `String`

Частным случаем использования массивов в качестве компонентов записей являются строки "String". Тип "String", по сути, является предопределенным неограниченным массивом символов, поэтому для строк, при описании полей записи, допускается как предварительное описание какого-либо строкового типа или подтипа, так и непосредственное использование типа "String". Например:

```
type    Name_String     is new String (1..10);
subtype Address_String  is String (1..20);

type  Person  is
    record
        First_Name: Name_String;
        Last_Name : Name_String;
        Address   : Address_String;
        Phone     : String (1..15);
    end record;
```

В приведенном выше примере описания типа записи "Person", для описания полей "First_Name" и "Last_Name" используется самостоятельный строковый тип "Name_String", производный от типа "String". Для описания поля "Address", используется подтип "Address_String". Следует заметить, что тип "Name_String" и подтип "Address_String", оба описывают ограниченные строки (или массивы символов). При описании поля "Phone"

непосредственно использован тип "String". В этом случае, для типа "String", указывается ограничение для значений индекса — " (1..15)".

Для строковых полей, вместо агрегатов массива допускается использование строковых литералов. Например:

```
Chief : Person := (First_Name  => "Matroskin ",
                   Last_Name   => "Kot       ",
                   Address     => "Prostokvashino      ",
                   Phone       => "8-9-222-333    ");
```

6.2.3 Вложенные записи

Бывают случаи, когда компонент записи сам является записью, или, говоря иначе, требуется использование вложенных записей. Например:

```
type  Point is record
    X : Integer;
    Y : Integer;
end record

type Rect is record
    Left_Hight_Corner : Point;
    Right_Low_Corner  : Point;
end record

P : Point := (100, 100);
R : Rect;
```

В этом случае, доступ к полям переменной "R" типа "Rect"может быть выполнен следующим образом:

```
R.Left_Hight_Corner.X := 0;
R.Left_Hight_Corner.Y := 0;

R.Right_Low_Corner := P;
```

Для указания всех значений можно использовать агрегаты.

```
R_1 : Rect := ( (0, 0), (100, 100) );

R_2 : Rect := (Left_Hight_Corner => (Y => 0, X => 0),
               Right_Low_Corner  => (100, 100));
```

Как видно из приведенных примеров, здесь используются вложенные агрегаты.

6.3 Дискриминанты

Особенность всех ранее рассмотренных записей в том, что они всегда имеют строго определенные структуры. Действительно, число полей таких записей и их размеры строго зафиксированы на этапе описания типа записи и никак не могут быть изменены впоследствии. Однако, в реальной жизни, достаточно часто возникают ситуации когда необходимо сделать структуру записи зависимой от каких-либо условий.

Для решения такого рода задач, Ада разрешает записям содержать дополнительные поля — дискриминанты. Такие дополнительные поля являются средством "параметризации"и помогают выполнять для записей требуемую настройку. В этом случае, разные экземпляры записей могут принадлежать одному и тому же типу, но при этом иметь разный размер и/или разное количество полей.

Следует заметить, что значения дискриминантов записей должны быть дискретного или ссылочного типа.

6.3.1 Вариантные записи

Использование дискриминантов позволяет конструировать вариантные записи (или, называемые иначе, записи с вариантами). В этом случае, значение дискриминанта определяет наличие или отсутствие некоторых полей записи.

Рассмотрим следующий пример:

```
type Vehicle is (Bicycle, Car, Truck, Scooter);

type Transport (Vehicle_Type : Vehicle) is
    record
        Owner       : String (1..10);       -- владелец
        Description : String (1..10);       -- описание
        case Vehicle_Type is
            when Car    =>
                            Petrol_Consumption : Float; -- расход бензина
            when Truck  =>
                            Diesel_Consumption : Float; -- расход солярки
                            Tare               : Real;  -- вес тары
                            Net                : Real;  -- вес нетто
            when others =>
                            null;
        end case;
  end record;
```

В представленном описании типа записи "Transport", поле "Vehicle_Type" (vehicle — транспортное средство) является дискриминантом.

В данном случае, значение дискриминанта должно быть указано при описании экземпляра записи. Таким образом, можно создавать различные объекты у которых значения дискриминантов будут отличаться. Заметим также, что при описании таких объектов допускается указывать только значение дискриминанта, а присваивание значений остальным полям записи можно выполнять позже. Агрегаты для записей с дискриминантами должны включать значение дискриминанта как первое значение. Рассмотрим следующие описания:

```
My_Car      : Transport (Car);
My_Bicycle  : Transport (Vehicle_Type => Bicycle);
His_Car     : Transport := (Car, "dale      ", "escort  ", 30.0);
```

Здесь, переменные "My_Car" и "His_Car", у которых дискриминант имеет значение "Car", содержат поля: "Owner", "Description" и "Petrol_Consumption". При этом, попытка обращения к таким полям этих переменных как "Tare" и/или "Net" будет не допустима. В результате, это приведет к генерации ошибки ограничения (*constraint error*), что может быть обнаружено и обработано при использовании механизма исключений (*exceptions*) язака Ада.

Следует особо отметить, что несмотря на то, что дискриминант является полем записи, непосредственное присваивание значения дискриминанту запрещено.

Также отметим, что в приведенных примерах тип записи, для простоты, имеет только один дискриминант. Реально, запись может содержать столько дискриминантов, сколько необходимо для решения конкретной задачи.

И в заключение укажем несколько общих особенностей описания записей с вариантами:

- вариантная часть всегда записывается последней
- запись может иметь только одну вариантную часть, однако внутри вариантной части разрешается указывать другой раздел вариантов
- для каждого значения дискриминанта должен быть указан свой список компонентов
- альтернатива "**others**" допустима только для последнего варианта, она задает все оставшиеся значения дискриминанта (возможно, и ни одного), которые не были упомянуты в предыдущих вариантах
- если список компонентов варианта задан как "**null**", то такой вариант не содержит никаких компонентов

6.3.2 Ограниченные записи (*constrained records*)

Записи "My_Car", "My_Bicycle" и "His_Car", которые мы рассматривали выше, называют ограниченными. Для таких записей значение дискриминанта определяется при описании экземпляра (переменной) записи. Таким образом, однажды определенный дискриминант в последствии никогда не может быть изменен. Так, запись "My_Bicycle" не имеет полей "Tare", "Net", "Petrol_Consumption", и т.д. При этом, компилятор Ады даже не будет распределять пространство для этих полей.

Из всего этого следует общее правило: любой экземпляр записи, который описан с указанием значения дискриминанта, будет называться **ограниченным** (*constrained*). Его дискриминант никогда не может принимать значение, отличное от заданного при его описании.

6.3.3 Неограниченные записи (*unconstrained records*)

Ада позволяет описывать экземпляры записей без указания начального значения для дискриминанта, тогда запись будет называться **неограниченной** (*unconstrained*). В этом случае дискриминант может принимать любое значение на протяжении всего времени существования записи, иначе говоря, дискриминант можно изменять. Очевидно, что в этом случае, компилятор должен будет распределить достаточное пространство для того, чтобы иметь возможность разместить наибольшую по размеру запись.

Однако, для выполнения вышесказанного, дискриминант записи обязан иметь значение по умолчанию, которое указывается при описании типа записи.

```
type Accounts is (Cheque, Savings);
type Account (Account_Type: Accounts := Savings) is
    record
        Account_No   : Positive;
        Title        : String (1..10);
        case Account_Type is
            when Savings  => Interest : Rate;
            when Cheque   => null;
        end case;
    end record;
```

Здесь, дискриминант записи "Account" имеет значение по умолчанию "Savings". Теперь, мы можем описать запись:

```
Household_Account : Account;
```

Такая запись будет создана с определенным по умолчанию значением дискриминанта. Но теперь, мы позже, при необходимости, можем изменить тип этой записи.

```
Household_Account := (Cheque, 123_456, "household ");
```

В общем следует заметить, что Ада требует чтобы при описании типа записи значения дискриминантов по умолчанию либо указывались для всех дискриминантов, либо не указывались вовсе. При этом, необходимо учесть, что если тип записи описан с указанием значений дискриминантов по умолчанию и, затем, при описании экземпляра записи было указано значение дискриминанта, то такой экземпляр записи, в результате, будет ограниченным.

Также необходимо особо отметить, что значение дискриминанта неограниченного объекта запрещается изменять независимо от изменения значений других полей, а непосредственное присваивание значений дискриминанта, как уже говорилось, вовсе запрещено. Поэтому, единственным способом изменения значения дискриминанта является присваивание значения всему объекту. Кроме того, присваивание значений всему объекту сразу является единственным способом изменения значений тех компонентов, у которых определение подтипа зависит от значения дискриминанта. Для пояснения последнего, рассмотрим пример:

```
type  Property  is array (Positive range <>) of Float;
type  Man (Number: Positive := 2; Size: Positive := 10) is
    record
        Name        : String (1..Size);
        Prop_Array  : Property (1..Number);
    end record;

The_Man : Man;
```

```
The_Man.Name       := "Ivanov I I";
The_Man.Prop_Array := (25.0, 50.0);

   . . .

The_Man := (Number     => 3,
            Size       => 8,
            Name       => "Pyle I C",
            Prop_Array => (25.0, 50.0, 160.5));
```

Здесь, первоначально объект "The_Man" описан как запись, значения дискриминантов которой устанавливаются по умолчанию. Затем, значения дискриминантов изменяются, но это изменение выполняется согласно требований Ады: осуществляется присваивание значения всей переменной.

6.3.4 Другие использования дискриминантов

Также как и при использовании в качестве селектора выбора варианта в записи, дискриминант может быть использован для спецификации длины массива, который является компонентом записи.

```
type Text (Length  : Positive := 20) is
    record
        Value : String (1..Length);
    end record;
```

В этом случае длина массива зависит от значения дискриминанта. Как указано выше, запись может быть описана как ограниченная (*constrained*) или как неограниченная (*unconstrained*).

Обычно, такие текстовые записи используются как строки переменной длины для текстовой обработки.

Еще одним возможным способом использования дискриминантов могут быть описания различных подтипов какого-либо базового типа. В данном случае смысл идеи состоит в том, что подтипы можно использовать для создания объектов с конкретными значениями дискриминантов. Такой подход удобен при работе с типами, которые содержат большое количество дискриминантов.

Глава 7

Подпрограммы

В Аде, также как и во всех современных языках программирования, подпрограммы позволяют программисту группировать инструкции в самостоятельные, логически законченные алгоритмические единицы, которые, в последствии, могут быть вызваны и выполнены в любом месте программы. Они являются элементарным базовым средством для повторного использования кода и разделения одной большой задачи на самостоятельные подзадачи меньшего размера (декомпозиция). Использование подпрограмм позволяет уменьшить как общий размер исходных текстов программы, так и общий размер результирующих исполняемых файлов. Таким образом, применение подпрограмм облегчает общее управление проектом и упрощает его сопровождение.

При этом, подпрограммы Ады обладают некоторыми свойствами, которые будут новыми для программистов использующих Паскаль и/или Си. Совмещение (*overloading*), именованные параметры, значение параметров по умолчанию, режимы передачи параметров и возврата значений — все это значительно отличает подпрограммы языка Ада.

7.1 Общие сведения о подпрограммах

Также как и в Паскале, подпрограммами Ады являются процедуры и функции.

Подпрограммы могут иметь параметры различных типов или не иметь ни одного параметра. При описании механизмов передачи параметров, как правило, используются следующие понятия:

формальный параметр — это параметр, который используется при описании подпрограммы (процедуры или функции).

фактический параметр — это объект, который передается в подпрограмму при вызове подпрограммы. В качестве фактического параметра подпрограммы может быть передано целое выражение, при этом режимом передачи параметра не может быть режим "**out**".

Каждый формальный параметр подпрограммы имеет имя, тип и режим передачи.

Подпрограммы могут содержать локальные описания типов, подтипов, переменных, констант, других подпрограмм и пакетов.

Подпрограмма Ады, как правило, состоит из двух частей: спецификации и тела. Спецификация описывает интерфейс обращения к подпрограмме, другими словами — "что" обеспечивает подпрограмма. Тело подпрограммы описывает детали реализации алгоритма работы подпрограммы, то есть, "как" подпрограмма устроена.

Разделение описания подпрограммы на спецификацию и тело не случайно, и имеет большое значение. Такой подход позволяет предоставить пользователю подпрограммы только ее спецификацию и оградить, и даже избавить его от деталей реализации подпрограммы.

Однажды скомпилированная и сохраненная в атрибутивном файле спецификация может быть проверена на совместимость с другими подпрограммами (и пакетами) когда они будут компилироваться. Таким образом, при проектировании, мы можем представить только спецификацию подпрограммы и передать ее компилятору. При предоставлении большого количества фиктивных подпрограмм-заглушек (*stubs*) мы можем

осуществлять предварительное тестирование проекта всей системы и обнаруживать любые ошибки проектирования до того как будет потрачено много усилий на реализацию конкретных решений (идеи данного подхода рассматриваются также при обсуждении концепции пакетов).

Необходимо заметить, что спецификации подпрограмм, как правило, помещаются в спецификации пакетов, а тела подпрограмм — в тела пакетов.

Кроме того, подпрограмма может быть самостоятельным независимым программным модулем. В этом случае, спецификация и тело подпрограммы помещаются в разные файлы (в файл спецификации и в файл тела, соответственно). Следует также заметить, что помещать спецификацию в отдельный файл, когда подпрограмма не является самостоятельной единицей компиляции не обязательно. В этом случае, спецификация подпрограммы может отсутствовать.

7.1.1 Процедуры

Процедуры Ады подобны процедурам Паскаля и используются для реализации самых разнообразных алгоритмов.

Общий вид описания процедуры выглядит следующим образом:

procedure *имя_процедуры* [(*формальные_параметры*)] ;	спецификация процедуры, определяющая имя процедуры и профиль ее формальных параметров (если они есть)

Общий вид тела процедуры:

procedure *имя_процедуры* [(*формальные_параметры*)] **is**	спецификация процедуры, определяющая имя процедуры и профиль ее формальных параметров (если они есть)
. . .	описательная (или декларативная) часть, которая может содержать локальные для этой процедуры описания типов, переменных, констант, подпрограмм. . .
begin . . .	исполняемая часть процедуры, которая описывает алгоритм работы процедуры; обязана содержать хотя бы одну инструкцию
end *имя_процедуры*;	здесь, указание имени процедуры не является обязательным

Таким образом, описание содержит только спецификацию процедуры и определяет правила ее вызова (иначе — интерфейс), а тело содержит спецификацию и последовательность инструкций, которые выполняются при вызове процедуры.

Примечательно требование Ады, чтобы исполняемая часть процедуры содержала хотя бы одну инструкцию. Поэтому, как правило на этапе проектирования, при написании процедур-заглушек используется пустая инструкция, например:

```
procedure Demo (X: Integer; Y: Float) is
begin
    null; -- пустая инструкция
end Demo;
```

Вызов процедуры производится также как и в языке Паскаль, например:

```
Demo (4, 5.0);
```

Необходимо также заметить, что Ада предоставляет программисту возможность, при необходимости, помещать в любых местах внутри исполнительной части процедуры инструкцию возврата из процедуры — "**return**".

7.1.2 Функции

Функции во многом подобны процедурам, за исключением того, что они возвращают значение в вызвавшую их подпрограмму. Также можно сказать, что функция — это подпрограмма которая осуществляет преобразование одного или нескольких входных значений в одно выходное значение.

Общий вид описания функции выглядит следующим образом:

function *имя_функции* [(*формальные_параметры*)] **return** *тип_возвращаемого_значения* ;	спецификация функции, определяющая имя функции, профиль ее формальных параметров (если они есть) и тип возвращаемого значения

Общий вид тела функции:

function *имя_функции* [(*формальные_параметры*)] **return** *тип_возвращаемого_значения* **is**	спецификация функции, определяющая имя функции, профиль ее формальных параметров (если они есть) и тип возвращаемого значения
. . .	описательная (или декларативная) часть, которая может содержать локальные для этой функции описания типов, переменных, констант, подпрограмм. . .
begin . . .	исполняемая часть функции, которая описывает алгоритм работы функции; обязана содержать хотя бы одну инструкцию возврата значения "**return**"
end *имя_функции*;	здесь, указание имени функции не является обязательным

Использование инструкции возврата значения — "**return**" очень похоже на то, что используется в языке Си, при этом, функция может иметь сколько угодно инструкций возврата значения.

Функция может быть вызвана как часть выражения в инструкции присваивания или как аргумент соответствующего типа при вызове другой функции или процедуры. Другими словами — функция, возвращающая значения заданного типа, может быть использована везде, где может быть использована переменная этого типа.

Хотя режимы передачи параметров в подпрограммы будут подробно рассмотрены несколько позже, здесь, необходимо сделать несколько важных замечаний, которые имеют значение для функций Ады.

Согласно традиций стандарта Ada83, для передачи параметров в функцию разрешается использовать только режим "**in**". Поэтому, функция, через свои параметры, может только импортировать данные из среды вызвавшей эту функцию. При этом, параметры функции не могут быть использованы для изменения значений переменных в среде вызвавшей функцию. Таким образом, в отличие от традиций языка Си, функции Ады не обладают побочными эффектами.

Стандарт Ada95 ввел новый режим для передачи параметров — "**access**". Этот режим разрешается использовать для передачи параметров в функции. Следует заметить, что использование этого режима допускает написание функций обладающих побочными эффектами.

7.1.3 Локальные переменные

Как уже говорилось, подпрограммы (и процедуры, и функции) могут содержать локальные переменные. Такие переменные доступны только внутри подпрограммы и не видимы извне этой подпрограммы. Время жизни (время существования) таких переменных определяется временем жизни (временем выполнения) подпрограммы.

Во время работы программы, при входе в подпрограмму, имеющую локальные переменные, в стеке времени выполнения происходит автоматическое распределение пространства для локальных переменных данной подпрограммы. При выходе из подпрограммы пространство стека времени выполнения, распределенное для локальных переменных данной подпрограммы, автоматически возвращается системе.

7.1.4 Локальные подпрограммы

Также как и Паскаль, и в отличие от Си, Ада позволяет встраивать одни подпрограммы в другие подпрограммы, конструируя один общий компилируемый модуль. Другими словами, подпрограмма Ады может содержать внутри себя вложенные подпрограммы, которые не будут видимы извне этой подпрограммы. К таким локальным подпрограммам можно обращаться только из подпрограммы которая их содержит.

```
with Ada.Text_IO;           use Ada.Text_IO;
with Ada.Integer_Text_IO;   use Ada.Integer_Text_IO;

procedure Ive_Got_A_Procedure is

    X : Integer := 6;
    Y : Integer := 5;

    procedure Display_Values (Number : Integer) is
    begin
        Put (Number);
        New_Line;
    end Display_Values;

begin
    Display_Values (X);
    Display_Values (Y);
end Ive_Got_A_Procedure;
```

В этом примере область видимости процедуры "Display_Values" ограничивается процедурой "Ive_Got_A_Procedure". Таким образом, процедура "Display_Values" "не видна" и не может быть вызвана из любого другого места.

7.1.5 Раздельная компиляция

В предыдущем примере, если будет произведено какое-нибудь изменение кода, то обе процедуры должны быть переданы компилятору (поскольку обе находятся в одном файле с исходным текстом). Мы можем разделить эти две компоненты, и поместить их в отдельные файлы, оставляя без изменения ограничения области видимости для процедуры "Display_Values". Это несколько похоже на директиву "**#include**", используемую в языке Си, но, в языке Ада, теперь оба файла становятся независимыми компилируемыми модулями.

В первом файле:

```
with Ada.Text_IO;           use Ada.Text_IO;
with Ada.Integer_Text_IO;   use Ada.Integer_Text_IO;

procedure Ive_Got_A_Procedure is

    X : Integer := 6;
    Y : Integer := 5;

    procedure Display_Values (Number : Integer) is separate;

begin
    Display_Values (X);
    Display_Values (Y);
end Ive_Got_A_Procedure;
```

Во втором файле:

```
separate (Ive_Got_A_Procedure)    -- примечание: нет завершайщего символа
                                  -- точки с запятой

procedure Display_Values (Number : Integer) is
begin
    Put (Number);
    New_Line;
end Display_Values;
```

Выделенный в самостоятельный файл (и ставший отдельно компилируемым модулем), код — идентичен тому, что было в предыдущей версии. Однако теперь, если будет изменена только внутренняя подпрограмма, то только она должна быть подвергнута перекомпиляции компилятором. Это также позволяет разделить программу на несколько частей, что может облегчить ее понимание.

7.1.6 Подпрограммы как библиотечные модули

Любая подпрограмма Ады, при необходимости, может быть оформлена как абсолютно самостоятельный независимый библиотечный подпрограммный модуль.

Рассмотрим как это делается на примере процедур "Ive_Got_A_Procedure" и "Display_Values", из предыдущего примера о раздельной компиляции. Теперь, процедура "Display_Values" будет оформлена как самостоятельный библиотечный подпрограммный модуль, а процедура "Ive_Got_A_Procedure"будет ее использовать.

В этом случае, полное описание процедуры "Display_Values"будет состоять из двух файлов: файла спецификации и файла тела процедуры.

Примечание:
В системе компилятора *GNAT* существует соглашение согласно которому файлы спецификаций имеют расширение "`ads`" (*ADa Specification*), а файлы тел имеют расширение "`adb`" (*ADa Body*).

Файл спецификации процедуры "Display_Values" (`display_values.ads`) будет иметь следующий вид:

```
procedure Display_Values (Number : Integer);
```

Файл тела процедуры "Display_Values" (`display_values.adb`) будет иметь следующий вид:

```
with Ada.Text_IO;          use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Display_Values (Number : Integer) is
begin
    Put (Number);
    New_Line;
end Display_Values;
```

Третий файл — это файл тела процедуры "Ive_Got_A_Procedure" (`ive_got_a_procedure.adb`). В этом случае он будет иметь следующий вид:

```
with Ada.Text_IO;          use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
with Display_Values;

procedure Ive_Got_A_Procedure is

    X : Integer := 6;
    Y : Integer := 5;

begin
    Display_Values (X);
    Display_Values (Y);
end Ive_Got_A_Procedure;
```

Примечательно, что теперь, в файле тела процедуры "Ive_Got_A_Procedure", процедуру "Display_Values", которая оформлена как самостоятельный библиотечный модуль, необходимо указать в спецификаторе совместности контекста "**with**".

Также, необходимо заметить, что подпрограммы, оформленные как самостоятельные библиотечные модули, не указываются в спецификаторе использования контекста "**use**".

7.2 Режимы передачи параметров

Стандарт Ada83 предусматривал три режима передачи параметров для подпрограмм:

in
in out
out

Стандарт Ada95 добавил еще один режим передачи параметров:

access

Все эти режимы не имеют непосредственных аналогов в других языках программирования. Необходимо также отметить следующее:

> по умолчанию, для передачи параметров подпрограммы, всегда устанавливается режим — "**in**" !!!

Для "**in**" / "**out**" скалярных значений используется механизм передачи параметров по копированию-"**in**" (*copy-in*), по копированию-"**out**" (*copy-out*). Стандарт специфицирует, что любые другие типы могут быть переданы по *copy-in/copy-out*, или по ссылке.

Ada95 указывает, что лимитированные приватные типы (*limited private types*), которые рассматриваются позднее, передаются по ссылке, для предотвращения проблем нарушения приватности.

7.2.1 Режим `in`

Параметры передаваемые в этом режиме подобны параметрам передаваемым по значению в языке Паскаль, и обычным параметрам языка Си, с тем исключением, что им не могут присваиваться значания внутри подпрограммы.

Это значит, что при входе в подпрограмму, формальный параметр инициализируется значением фактического параметра, при этом, внутри подпрограммы, он является константой и разрешает только чтение значения ассоциированного фактического параметра.

```
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Demo (X : in Integer; Y : in Integer) is
begin
    X := 5;    -- недопустимо, "in"-параметр доступен только по чтению
    Put (Y);
    Get (Y);   -- также недопустимо
end Demo;
```

Режим "**in**" разрешается использовать и в процедурах, и в функциях.

7.2.2 Режим `in out`

Этот режим непосредственно соответствует параметрам передаваемым по ссылке (подобно *var*-параметрам языка Паскаль).

Таким образом, при входе в подпрограмму, формальный параметр инициализируется значением фактического параметра. Внутри подпрограммы, формальный параметр, использующий этот режим, может быть использован как в левой, так и в правой части инструкций присваивания (другими словами: формальный параметр доступен как для чтения, так и для записи). При этом, если формальному параметру внутри подпрограммы произведено присваивание нового значения, то после выхода из подпрограммы значение фактического параметра заменяется на новое значение формального параметра.

```
procedure Demo (X : in out Integer;
                Y : in     Integer) is

    Z : constant Integer := X;

begin
    X := Y * Z; -- это допустимо!
end Demo;
```

Режим "**in out**" разрешается использовать только в процедурах.

7.2.3 Режим out

В этом режиме, при входе в подпрограмму, формальный параметр **не** инициализируется (**!!!**) значением фактического параметра. Согласно стандарта Ada95, внутри подпрограммы, формальный параметр, использующий этот режим, может быть использован как в левой, так и в правой части инструкций присваивания (другими словами: формальный параметр доступен как для чтения, так и для записи). Согласно стандарта Ada83, внутри подпрограммы, такой формальный параметр может быть использован только в левой части инструкций присваивания (другими словами: доступен только для записи). При этом, после выхода из подпрограммы, значение фактического параметра заменяется на значение формального параметра.

```
procedure Demo (X : out Integer;
                Y : in Integer) is

   -- при входе в подпрограмму X не инициализирован !!!

begin
   X := Y;
end Demo;
```

Режим "**out**" разрешается использовать только в процедурах.

7.2.4 Режим access

Поскольку значения ссылочного типа (указатели) часто используются в качестве параметров передаваемых подпрограммам, Ада предусматривает режим передачи параметров "**access**", который специально предназначен для передачи параметров ссылочного типа. Заметим, что подробному обсуждению ссылочных типов Ады далее посвящена самостоятельная глава — *"Ссылочные типы (указатели)"*. Необходимо также обратить внимание на то, что режим передачи параметров "**access**" был введен стандартом Ada95 и он отсутствует в стандарте Ada83.

При использовании режима "**access**", фактический параметр, который предоставляется при вызове подпрограммы, — это любое значение ссылочного типа, которое ссылается (указывает) на объект соответствующего типа. При входе в подпрограмму, формальный параметр инициализируется значением фактического параметра, при этом, Ада производит автоматическую проверку того, что значение параметра не равно "**null**". В случае когда значение параметра равно "**null**" генерируется исключительная ситуация *Constraint_Error* (проще говоря, — ошибка). Внутри подпрограммы, формальный параметр, использующий режим "**access**", является константой ссылочного типа и ему нельзя присваивать новое значение, поэтому такие формальные параметры несколько подобны формальным параметрам, использующим режим "**in**". Однако, поскольку параметр является значением ссылочного типа (указателем), то подпрограмма может изменить содержимое объекта на который данный параметр ссылается (указывает). Кроме того, внутри подпрограммы такой параметр принадлежит анонимному ссылочному типу, и поскольку у нас нет возможности определить имя этого ссылочного типа, то мы не можем описать ни одного дополнительного объекта этого типа. Любая попытка конвертирования значения такого параметра в значение именованого ссылочного типа будет проверяться на соответствие правилам области действия для ссылочных типов. При обнаружении нарушения этих правил генерируется исключительная ситуация *Programm_Error*.

```
   . . .

function Demo_Access (A : access Integer) return Integer is
begin
   return A.all;
end Demo_Access;

   . . .

type  Integer_Access  is access Integer;

Integer_Access_Var  : Integer_Access := new Integer'(1);
Aliased_Integer_Var : aliased Integer;

   . . .
```

```
X : Integer := Demo_Access (Integer_Access_Var);
Y : Integer := Demo_Access (Aliased_Integer_Var'Access);
Z : Integer := Demo_Access (new Integer);

   . . .
```

Режим "**access**" разрешается использовать и в процедурах, и в функциях.

При этом необходимо обратить внимание на то, что функции, использующие этот режим для передачи параметров, способны изменять состояние объектов на которые такие параметры ссылаются. То есть, такие функции могут обладать побочными эффектами.

7.3 Сопоставление формальных и фактических параметров

7.3.1 Позиционное сопоставление

Позиционное сопоставление формальных и фактических параметров при вызове подпрограммы достаточно традиционно, и используется во многих языках программирования. При таком сопоставлении, ассоциирование между формальными и фактическими параметрами производится один к одному позиционно, т.е. первый формальный параметр ассоциируется с первым фактическим параметром и т.д.

```
procedure Demo (X : Integer; Y : Integer);   -- спецификация процедуры

  . . .

Demo (1, 2);
```

В этом случае, фактический параметр "1" будет подставлен вместо первого формального параметра "X", а фактический параметр "2" будет подставлен вместо второго формального параметра "Y".

7.3.2 Именованное сопоставление

Для улучшения читабельности вызовов подпрограмм (а Ада разрабатывалась с учетом хорошей читабельности) Ада позволяет использовать именованное сопоставление формальных и фактических параметров. В этом случае мы можем ассоциировать имя формального параметра с фактическим параметром. Это свойство делает вызовы подпрограмм более читабельными.

```
procedure Demo (X : Integer; Y : Integer);   -- спецификация процедуры

  . . .

Demo (X => 5, Y => 3 * 45);  -- именованное сопоставление
                             -- формальных и фактических
                             -- параметров при вызове
```

Расположение списка параметров вертикально, также способствует улучшению читабельности.

```
Demo (X => 5,
      Y => 3 * 45);
```

Поскольку при именованом сопоставлении производится явное ассоциирование между формальными и фактическими параметрами (вместо неявного ассоциирования, используемого в случае позиционного сопоставления), то нет необходимости строго придерживаться того же самого порядка следования параметров, который указан в спецификации подпрограммы.

```
Demo (Y => 3 * 45,    -- при именованом сопоставлении
      X => 5);        -- порядок следования параметров
                      -- не имеет значения
```

7.3.3 Смешивание позиционного и именованного сопоставления

Ада позволяет смешивать позиционное и именованное сопоставление параметров. В этом случае должно соблюдаться следующее условие: позиционно-ассоциированные параметры должны предшествовать параметрам, которые ассоциируются по имени.

```
procedure Square (Result :    out Integer;
                  Number : in     Integer) is
begin
    Result := Number * Number;
end Square;
```

В результате, показанная выше процедура "Square" может быть вызвана следующими способами:

```
Square (X, 4);
Square (X, Number => 4);
Square (Result => X, Number => 4);
Square (Number => 4, Result => x);

Square (Number => 4, X); -- недопустимо, поскольку позиционно-ассоциируемый
                         -- параметр следует за параметром, ассоциируемым
                         -- по имени
```

7.4 Указание значения параметра по умолчанию

Для любых "**in**"-параметров ("**in**" или "**in out**"), в спецификации подпрограммы можно указать значение параметра по умолчанию. Синтаксис установки значения параметра по умолчанию подобен синтаксису определения инициализированных переменных и имеет следующий вид:

```
with Ada.Text_IO;          use Ada.Text_IO;

procedure Print_Lines (No_Of_Lines: Integer := 1) is

begin
    for Count in 1 .. No_Of_Lines loop
        New_Line;
    end loop;
end Print_Lines;
```

Такое описание устанавливает значение параметра "No_Of_Lines" для случаев когда процедура "Print_Lines" вызывается без указания значения этого параметра (позиционного или именованного).

Таким образом, вызов этой процедуры может иметь вид:

```
Print_Lines;        -- это печатает одну строку
Print_Lines (6);    -- переопределяет значение параметра
                    -- установленное по умолчанию
```

Подобно этому, если процедура "Write_Lines" была описана как:

```
with Ada.Text_IO;          use Ada.Text_IO;

procedure Write_Lines (Letter       : in Char := '*';
                       No_Of_Lines : in Integer := 1) is

begin
    for I in 1 .. No_Of_Lines loop
        for I in 1 .. 80 loop
            Put (Letter);
        end loop;
        New_Line;
    end loop;
end Write_Lines;
```

то она может быть вызвана следующими способами:

```
Write_Lines;                        -- для параметров Letter и No_Of_Lines
                                    -- используются значения устанавливаемые
                                    -- по умолчанию
Write_Lines ('-');                  -- значение по умолчанию - для No_Of_Lines
Write_Lines (no_of_lines => 5);     -- значение по умолчанию - для Letter
Write_Lines ('-', 5)                -- оба параметра определены
```

7.5 Совмещение (*overloading*)

Поиск новых имен для подпрограмм, которые выполняют одинаковые действия, но с переменными разных типов, всегда является большой проблемой при разработке программного обеспечения. Хорошим примером для иллюстрации такой проблемы является процедура "Insert".

В подобных случаях, для облегчения жизни программистам, Ада позволяет разным подпрограммам иметь одинаковые имена, предоставляя механизм который называется совмещением (*overloading*).

7.5.1 Совмещение подпрограмм (*subprogram overloading*)

Предоставляя механизм совмещения имен подпрограмм, Ада выдвигает единственное требование: подпрограммы должны быть распознаваемы (или различимы). Две подпрограммы, имеющие одинаковые имена, будут распознаваемы если они имеют разный "профиль". Профиль подпрограммы характеризуется количеством параметров и их типами, а также, если подпрограмма является функцией, — типом возвращаемого значения.

Таким образом, пока компилятор может однозначно определить к какой подпрограмме осуществляется вызов, анализируя совпадение профиля вызываемой подпрограммы со спецификациями представленных подпрограмм, — все будет в порядке. В противном случае, вы получите сообщение об ошибке, указывающее на двусмысленность обращения.

```
procedure Insert (Item : Integer);  -- две процедуры с одинаковыми именами,
procedure Insert (Item : Float);    -- но имеющие разный "профиль"
```

Примерами совмещенных подпрограмм могут служить процедуры "**Put**" и "**Get**" из стандартного пакета *Ada.Text_IO*.

7.5.2 Совмещение знаков операций (*operator overloading*)

В языках подобных Паскалю знак операции "+" — совмещен. Действительно, он может использоваться для сложения целых и вещественных чисел, и даже строк. Таким образом, очевидно что этот знак операции используется для представления кода который выполняет абсолютно разные действия.

Ада разрешает программистам выполнять совмещение предопределенных знаков операций с их собственным кодом. Следует заметить, в Аде, действие выполняемое знаком операции, реализуется путем вызова функции именем которой является знак операции заключенный в двойные кавычки. При этом, для обеспечения корректной работы механизма совмещения знаков операций, функция, которая реализует действие знака операции, должна соответствовать обычному требованию механизма совмещения подпрограмм Ады: она должна быть различима, то есть, ее профиль должен быть уникальным. В некоторых случаях, проблему двусмысленности знака операции можно преодолеть непосредственно специфицируя имя пакета (рассматривается далее).

Кроме того, к функциям которые реализуют действия знаков операций предъявляется дополнительное требование: они не могут быть выделены в самостоятельно компилируемые модули, а должны содержаться в другом модуле, таком как процедура, функция или пакет.

Рассмотрим простой пример в котором мы хотим предусмотреть возможность сложения двух векторов:

```
procedure Add_Demo is

   type Vector is array (Positive range <>) of Integer;
   A : Vector (1..5);
   B : Vector (1..5);
   C : Vector (1..5);
```

```
    function "+" (Left, Right : Vector) return Vector is

        Result   : Vector (Left'First..Left'Last);
        Offset   : constant Natural := Right'First - 1;

    begin
        if Left'Length /= Right'Length then
            raise Program_Error;          -- исключение,
                                          -- рассматриваются позже
        end if;

        for I in Left'Range loop
            Result (I) := Left (I) + Right (I - Offset);
        end loop;
        return Result;
    end "+";

begin
    A := (1, 2, 3, 4, 5);
    B := (1, 2, 3, 4, 5);
    C := A + B;
end Add_Demo;
```

В этом примере хорошо продемонстрированы многие ранее рассмотренные средства которые характерны для языка программирования Ада.

7.5.3 Спецификатор `use type`

В случае когда какой-либо пакет содержит описания знаков операций, может быть использована несколько модифицированная форма спецификатора использования "**use**", которая позволяет использовать описанные в этом пакете знаки операций без необходимости указания имени пакета в качестве префикса. При этом, в случае использования других компонентов, описанных в этом пакете, необходимость указания имени пакета в качестве префикса сохраняется. Эта модифицированная форма спецификатора использования имеет следующий вид:

use type *имя_типа*

Здесь, *имя_типа* указывает тип данных для которого знаки операций будут использоваться без указания имени пакета в качестве префикса.

Глава 8

Пакеты

Хотя корни Ады лежат в языке Паскаль, концепция пакетов была заимствована из других языков программирования и подверглась значительному влиянию последних наработок в области разработки программного обеспечения обсуждавшихся в 1970-х годах. Несомненно, что одной из главных инноваций в этот период была концепция программного модуля (или пакета).

Следует заметить, что концепция пакетов не рассматривает то, как программа будет выполняться. Идея данного подхода посвящена проблеме построения программного комплекса, облегчению понимания того как он устроен и, следовательно, упрощению его сопровождения.

8.1 Общие сведения о пакетах Ады

8.1.1 Идеология концепции пакетов

Прежде чем непосредственно рассматривать примеры языковых конструкций необходимо разобраться в чем заключается и что предлагает концепция пакетов.

Пакет — это средство, которое позволяет сгруппировать логически связанные вычислительные ресурсы и выделить их в единый самостоятельный программный модуль. Под вычислительными ресурсами в этом случае подразумеваются данные (типы данных, переменные, константы. . .) и подпрограммы которые манипулируют этими данными. Характерной особенностью данного подхода является разделение самого пакета на две части: спецификацию пакета и тело пакета. Причем, спецификацию имеет каждый пакет, а тело могут иметь не все пакеты.

Спецификация определяет интерфейс к вычислительным ресурсам (сервисам) пакета доступным для использования во внешней, по отношению к пакету, среде. Другими словами — спецификация показывает "что" доступно при использовании этого пакета.

Тело является приватной частью пакета и скрывает в себе все детали реализации предоставляемых для внешней среды ресурсов, то есть, тело хранит информацию о том "как" эти ресурсы устроены.

Необходимо заметить, что разбиение пакета на спецификацию и тело не случайно, и имеет очень важное значение. Это дает возможность по-разному взглянуть на пакет. Действительно, для использования ресурсов пакета достаточно знать только его спецификацию, в ней содержится вся необходимая информация о том как использовать ресурсы пакета. Необходимость в теле пакета возникает только тогда, когда нужно узнать или изменить реализацию чего-либо внутри самого пакета.

Средства построения такой конструкции как пакет дают программисту мощный и удобный инструмент абстракции данных, который позволяет объединить и выделить в логически законченное единое целое данные и код который манипулирует этими данными. При этом, пакет позволяет программисту скрыть все детали реализации сервисов за развитым функциональным интерфейсом. В результате, структурное представление программного комплекса в виде набора взаимодействующих между собой компонентов облегчает понимание работы комплекса в целом и, следовательно, позволяет облегчить его разработку и сопровождение.

Необходимо также заметить, что на этапе начального проектирования системы можно предоставлять компилятору только спецификации, обеспечивая детали реализации только самых необходимых элементов. Таким

образом проверка корректности общей структуры проекта осуществляется на ранней стадии, когда не потрачено много усилий на разработку реализации отдельных элементов, которые позже придется переделывать (что, к великому сожалению, в реальной жизни происходит достаточно часто).

Примечание:
В системе компилятора *GNAT* существует соглашение согласно которому файлы спецификаций имеют расширение `ads` (*ADa Specification*), а файлы тел имеют расширение `adb` (*ADa Body*).

8.1.2 Спецификация пакета

Как уже говорилось, спецификация пакета Ады определяет интерфейс доступа к вычислительным ресурсам (сервисам) пакета. Она может содержать описания типов, переменных, констант, спецификации подпрограмм, других пакетов, — все то, что должно быть доступно тому, кто будет использовать данный пакет. Простой пример спецификации пакета может иметь следующий вид:

```
package Odd_Demo is

    type A_String is array (Positive range <>) of Character;

    Pi   : constant Float := 3.14;

    X    : Integer;

    type A_Record is
        record
          Left  : Boolean;
          Right : Boolean;
        end record;

    -- примечательно, что дальше, для двух подпрограмм представлены только
    -- их спецификации, тела этих подпрограмм будут находиться в теле пакета

    procedure Insert (Item : in Integer; Success : out Boolean);
    function Is_Present (Item : in Integer) return Boolean;

end Odd_Demo;
```

Мы можем получить доступ к этим сервисам в нашем коде путем указания данного пакета в спецификаторе совместности контекста "**with**", а затем использовать полную точечную нотацию. Полная точечная нотация, в общем случае, имеет следующий вид:

имя_пакета . имя_используемого_ресурса

где *имя_используемого_ресурса* — это имя типа, подпрограммы, переменной и т.д.

Для демонстрации сказанного приведем схематический пример процедуры которая использует показанную выше спецификацию:

```
with Odd_Demo;

procedure Odder_Demo is

    My_Name : Odd_Demo.A_String;
    Radius  : Float;
    Success : Boolean;

begin
    Radius := 3.0 * Odd_Demo.Pi;
    Odd_Demo.Insert (4, Success);
    if Odd_Demo.Is_Present (34) then ...
        . . .
end Odder_Demo;
```

Не трудно заметить, что доступ к ресурсам пакета, текстуально подобен доступу к полям записи.

В случаях, когда использование полной точечной нотации для доступа к ресурсам пакета обременительно, можно использовать инструкцию спецификатора использования контекста "**use**". Это позволяет обращаться к ресурсам которые предоставляет данный пакет без использования полной точечной нотации, так, как будто они описаны непосредственно в этом же коде.

```
with Odd_Demo;          use Odd_Demo;

procedure Odder_Demo is

    My_Name : A_String;
    Radius  : Float;
    Success : Boolean;

begin
    Radius := 3.0 * Pi;
    Insert (4, Success);
    if Is_Present (34) then ...
        . . .
end Odder_Demo;
```

Если два пакета, указаны в инструкциях "**with**" и "**use**" в одном компилируемом модуле (например, в подпрограмме или другом пакете), то возможно возникновение коллизии имен используемых ресурсов, которые предоставляются двумя разными пакетами. В этом случае можно избавиться от двусмысленности путем возвращения к использованию полной точечной нотации для соответствующих ресурсов.

```
package No1 is
    A, B, C : Integer;
end No1;
```

```
package No2 is
    C, D, E : Integer;
end No2;
```

```
with No1;          use No1;
with No2;          use No2;

procedure Clash_Demo is
begin
    A := 1;
    B := 2;

    C := 3;          -- двусмысленность, мы ссылаемся
                     -- на No1.C или на No2.C?

    No1.C := 3;      -- избавление от двусмысленности путем возврата
    No2.C := 3;      -- к полной точечной нотации
end Clash_Demo;
```

Может возникнуть другая проблема — когда локально определенный ресурс "затеняет" ресурс пакета указанного в инструкции "**use**". В этом случае также можно избавиться от двусмысленности путем использования полной точечной нотации.

```
package No1 is
    A : Integer;
end No1;

with No1;          use No1;
procedure P is
    A : Integer;
begin
    A := 4;         -- это - двусмысленно
```

```
    P.A := 4;       -- удаление двусмысленности путем указания
                    -- имени процедуры в точечной нотации
    No1.A := 5;     -- точечная нотация для пакета
end P;
```

8.1.3 Тело пакета

Тело пакета содержит все детали реализации сервисов, указаных в спецификации пакета. Схематическим примером тела пакета, для показанной выше спецификации, может служить:

```
package body Odd_Demo is

    type List is array (1..10) of Integer;
    Storage_List : List;
    Upto         : Integer;

    procedure Insert (Item     : in  Integer;
                      Success  : out Boolean) is
    begin
        . . .
    end Insert;

    function Is_Present (Item : in Integer) return Boolean is

    begin
        . . .
    end Is_Present;


begin                   -- действия по инициализации пакета
                        -- это выполняется до запуска основной программы!
    for I in Storage_List'Range loop
        Storage_List (I) := 0;
    end loop;
    Upto := 0;
end Odd_Demo;
```

Все ресурсы, указанные в спецификации пакета, будут непосредственно доступны в теле пакета без использования дополнительных инструкций спецификации контекста "**with**" и/или "**use**".

Необходимо заметить, что тело пакета, также как и спецификация пакета, может содержать описания типов, переменных, подпрограмм и т.д. При этом, ресурсы, описанные в теле пакета, не доступны для использования в другом самостоятельном модуле (пакете или подпрограмме). Любая попытка обращения к ним из другого модуля будет приводить к ошибке компиляции.

Переменные, описанные в теле пакета, сохраняют свои значения между успешными вызовами публично доступных подпрограмм пакета. Таким образом, мы можем создавать пакеты, которые сохраняют информацию для более позднего использования (другими словами: сохранять информацию о состоянии).

Раздел "**begin** ... **end**", в конце тела пакета, содержит перечень инструкций инициализации для этого пакета. Инициализация пакета выполняется до запуска на выполнение главной подпрограммы. Это справедливо для всех пакетов. Следует заметить, что стандарт не определяет порядок выполнения инициализации различных пакетов.

8.2 Средства сокрытия деталей реализации внутреннего представления данных

Как уже указывалось, спецификация пакета определяет интерфейс, а его тело скрывает реализацию сервисов предоставляемых пакетом. Однако, в примерах, показанных ранее, от пользователей пакетов были скрыты только детали реализации подпрограмм, а все типы данных были открыто описаны в спецификации пакета. Следовательно, все детали реализации представления внутренних структур данных "видимы" пользователям. В результате, пользователи таких пакетов, полагаясь на открытость представления внутренних

структур данных и используя эти сведения, попадают в зависимость от деталей реализации структур данных. Это значит, что в случае какого-либо изменения во внутреннем представлении данных, как минимум, возникает необходимость в полной перекомпиляции всех программных модулей которые используют такую информацию. В худшем случае, зависимые модули придется не только перекомпилировать, но и переделать.

На первый взгляд, сама проблема выглядит достаточно безобидно, а идея скрыть реализацию внутреннего представления данных кажется попыткой ущемления здорового любопытства пользователя. Справедливо заметить, что такие мысли, как правило, возникают при виде простых структур данных представленных в учебных примерах. К сожалению, в реальной жизни все сложнее. Представьте себе последствия изменения внутреннего представления данных в реальном проекте когда зависимых модулей много и разрабатываются эти модули разными программистами.

Ада позволяет "закрыть" детали внутреннего представления данных и, таким образом, избавиться от проблемы массовых переделок. Такие возможности обеспечивает использование приватных типов (*private types*) и лимитированных приватных типов (*limited private types*).

8.2.1 Приватные типы (*private types*)

Рассмотрим пример пакета который управляет счетами в бухгалтерской книге.
При этом, нам необходимо осуществлять полный контроль над всеми манипуляциями которые выполняются с объектами, и мы обеспечиваем пользователям пакета только следующие возможности:

- изъятие средств ("Withdraw")
- размещение средств ("Deposit")
- создание счета ("Create")

Никакие другие пакеты не должны иметь представления о деталях реализации внутренней структуры объекта бухгалтерского счета ("Account") и, следовательно, иметь к ним доступ.

Для того чтобы выполнить поставленную задачу, мы описываем объект бухгалтерского счета "Account" как приватный тип:

```
package Accounts is
    type Account is private;          -- описание будет представлено позже

    procedure Withdraw (An_Account : in out Account;
                        Amount     : in     Money);

    procedure Deposit (An_Account : in out Account;
                       Amount     : in     Money);
    function Create (Initial_Balance : Money) return Account;
    function Balance (An_Account : in    Account) return Integer;

private                  -- эта часть спецификации пакета
                         -- содержит полные описания
    type Account is
        record
            Account_No : Positive;
            Balance    : Integer;
        end record;
end Accounts;
```

В результате такого описания, тип "Account" будет приватным. Следут заметить, что Ада разрешает использовать следующие предопределенные операции над объектами приватного типа вне этого пакета:

- присваивание
- проверка на равенство (не равенство)
- проверки принадлежности ("**in**", "**not in**")

Кроме предопределенных операций, над объектами приватного типа, вне этого пакета, разрешается использовать операции, которые объявлены как подпрограммы в спецификации пакета (обычные процедуры и функции, а также функции реализующие действия знаков операций).

Все детали реализации внутреннего представления приватного типа доступны в теле пакета, и любая подпрограмма в теле пакета имеет к ним доступ и может модифицировать приватный тип как обычный тип. Таким образом, приватность типа сохраняется только вне пакета.

В данном примере необходимо обратить внимание на то, что спецификация пакета разделена на две части. Все что находится до зарезервированного слова "**private**" — это общедоступная часть описаний, которая будет "видна" всем пользователям пакета. Все что находится после зарезервированного слова "**private**" — это приватная часть описаний, которая будет "видна" только внутри пакета (и в его дочерних модулях; см. "Дочерние модули").

Может показаться противоречивым размещение приватных описаний в спецификации пакета. Действительно, мы пытаемся скрыть детали реализации приватного объекта, и размещаем их в спецификации пакета, которая доступна. Однако, это необходимо для программ, которые размещают экземпляры объектов приватного типа поскольку компилятор, благодаря такой информации, знает сколько необходимо зарезервировать места для размещения экземпляра объекта приватного типа.

Хотя читатель спецификации пакета видит как устроено реальное внутреннее представление реализации приватного типа, это не обеспечивает его возможностью явным и допустимым способом использовать эту информацию. При этом, примечательным является то, что экземпляры объектов приватного типа могут быть созданы вне пакета. Например:

```
with Accounts;          use Accounts;

procedure Demo_Accounts is

    Home_Account : Account;
    Mortgage     : Account;
    This_Account : Account;

begin
    Mortgage := Accounts.Create (Initial_Balance => 500.00);
    Withdraw (Home_Account, 50);

        . . .

    This_Account := Mortgage;          -- присваивание приватного типа - разрешено

    -- сравнение приватных типов
    if This_Account = Home_Account then

        . . .

end Demo_Accounts;
```

8.2.2 Лимитированные приватные типы (*limited private types*)

Хотя приватные типы позволяют разработчику получить значительный контроль над действиями пользователя, ограничив способности пользователя в манипулировании объектами, бывают случаи когда необходимо запретить пользователю приватного типа использовать даже такие предопределенные операции как сравнение и присваивание.

В качестве демонстрации сказанного, рассмотрим следующий пример:

```
package Compare_Demo is

    type Our_Text is private;

    . . .

private
```

```
        type Our_Text (Maximum_Length : Positive := 20) is
            record
                Length : Index := 0;
                Value  : String (1..Maximum_Length);
            end record;

        . . .

    end Compare_Demo;
```

Здесь, тип "Our_Text" описан как приватный и представляет из себя запись. В данной записи, поле длины "Length" определяет число символов которое содержит поле "Value" (другими словами — число символов которые имеют смысл). Любые символы, находящиеся в позиции от "Length + 1" до "Maximum_Length" будут нами игнорироваться при использовании этой записи. Однако, если мы попросим компьютер сравнить две записи этого типа, то он, в отличие от нас, не знает предназначения поля "Length". В результате, он будет последовательно сравнивать значения поля "Length" и значения всех остальных полей записи. Очевидно, что алгоритм предопределенной операции сравнения в данной ситуации не приемлем, и нам необходимо написать собственную функцию сравнения.

Для подобных случаев Ада предусматривает лимитированные приватные типы. Изменим рассмотренный выше пример следующим образом:

```
    package Compare_Demo is

        type Our_Text is limited private;

        . . .

        function "=" (Left, Right : in Our_Text) return Boolean;

        . . .

    private
        type Our_Text (Maximum_Length : Positive := 20) is
            record
                Length : Index := 0;
                Value  : String (1..Maximum_Length);
            end record;

        . . .

    end Compare_Demo;
```

Теперь, тип "Our_Text" описан как лимитированный приватный тип. Также, в спецификации пакета, описана функция знака операции сравнения на равенство "=". Реализация алгоритма этой функции должна быть помещена в тело пакета. Примечательно, что при совмещении знака операции равенства "=" автоматически производится неявное совмещение знака операции неравенства "/=". При этом следует учесть, что если функция реализующая действие знака операции равенства "=" возвращает значение тип которого отличается от предопределенного логического типа "**Boolean**" (полное имя — "Standard.**Boolean**"), то совмещение знака операции неравенства "/=" необходимо описать явно. Следует заметить, что Ада разрешает переопределять знак операции равенства для всех типов.

Для лимитированного приватного типа можно также создать процедуру для выполнения присваивания (или инициализации). Например, для показанного выше типа "Our_Text", спецификация такой процедуры может иметь следующий вид:

```
        . . .
        procedure Init (T : in out Our_Text;
                        S : in     String);
        . . .
```

Напомним, что спецификация такой процедуры должна быть размещена в спецификации пакета *Compare_Demo*, а ее тело (реализация) — в теле этого пакета.

8.2.3 Отложенные константы (*deferred constants*)

В некоторых спецификациях пакетов возникает необходимость описать константу приватного типа. Это можно выполнить таким же образом как и описание приватного типа (а также большинство опережающих ссылок). В общедоступной части спецификации пакета мы создаем неполное описание константы, после чего, компилятор ожидает получить полное описание константы в приватной части спецификации пакета. Например:

```
package Coords is

    type Coord is private;
    Home: constant Coord;  -- отложенная константа!

private
    type Coord is record
        X : Integer;
        Y : Integer;
    end record;

    Home : constant Coord := (0, 0);
end Coords;
```

В результате такого описания, пользователи пакета *Coords* "видят" константу "Home", которая имеет приватный тип "Coord", и могут использовать эту константу в своем коде. При этом, детали внутреннего представления этой константы им не доступны и они могут о них не заботиться.

8.3 Дочерние модули (*child units*) (Ada95)

Не редко, во время интенсивной разработки большой системы, возникают случаи когда один из пакетов разрастается необычайно быстро, а его спецификация подвергается постоянным изменениям. Таким образом, ввиду частого обновления спецификации пакета, резко возрастают затраты на перекомпиляцию зависимых модулей, а увеличение размеров самого пакета усложняет его дальнейшую разработку и последующее сопровождение.

Для преодоления подобных трудностей Ада предлагает концепцию дочерних модулей, которая является основой в построении иерархических библиотек. Такой подход позволяет разделить пакет большого размера на самостоятельные пакеты и подпрограммы меньшего размера, объединенные в общую иерархию. Кроме того, эта концепция позволяет расширять уже существующие пакеты. Она предлагает удобный инструмент для обеспечения множества реализаций одного абстрактного типа и дает возможность разработки самодостаточных подсистем при использовании приватных дочерних модулей.

Дочерние модули непосредственно используются в стандартной библиотеке Ады. В частности, дочерними модулями являются такие пакеты как *Ada.Text_IO*, *Ada.Integer_Text_IO*. Следует также заметить, что концепция дочерних модулей была введена стандартом Ada95. В стандарте Ada83 эта идея отсутствует.

8.3.1 Расширение существующего пакета

Рассмотрим случай когда возникает необходимость расширения пакета который уже содержит некоторое множество описаний. Например, в пакете *Stacks* может понадобиться дополнительный сервис просмотра "Peek". Если в текущий момент времени пакет *Stacks* используется многими модулями, то такая модификация, путем добавления нового сервиса, потребует значительных затрат на перекомпиляцию всех зависимых модулей, причем, включая и те модули которые не будут использовать новый сервис.

Следовательно, для логического расширения уже существующего пакета предпочтительнее использовать дочерний пакет который будет существовать абсолютно отдельно. Например:

```
package Stacks is

    type Stack is private;
    procedure Push (Onto : in out Stack; Item : Integer);
    procedure Pop (From : in out Stack; Item : out Integer);
    function Full (Item : Stack) return Boolean;
```

```
    function Empty (Item : Stack) return Boolean;

private
    -- скрытая реализация стека
    ...
    -- точка A

end Stacks;


package Stacks.More_Stuff is

    function Peek (Item : Stack) return Integer;

end Stacks.More_Stuff;
```

По правилам Ады, спецификация родительского пакета обеспечивает для дочернего пакета такую же самую область видимости, что и для своего тела. Следовательно, дочерний пакет видит всю приватную часть спецификации родительского пакета.

В показанном выше примере, пакет *Stacks.More_Stuff* является дочерним пакетом для пакета *Stacks*. Значит, дочерний пакет *Stacks.More_Stuff* "видит" все описания пакета *Stacks*, вплоть до точки *A*.

Необходимо заметить, что для сохранения приватности описаний родительского пакета, Ада не позволяет включать в спецификацию дочернего пакета приватную часть спецификации родительского пакета.

Примечание:
Согласно правил именования файлов, принятым в системе компилятора *GNAT*, спецификация и тело пакета *Stacks* должны быть помещены в файлы `stacks.ads` и `stacks.adb` соответственно, а спецификация и тело дочернего пакета *Stacks.More_Stuff* - в файлы `stacks-more_stuff.ads` и `stacks-more_stuff.adb`

Клиенту, которому необходимо использовать функцию "Peek", просто необходимо включить дочерний пакет в инструкцию спецификатора совместности контекста "**with**":

```
with Stacks.More_Stuff;

procedure Demo is

    X : Stacks.Stack;

begin
    Stacks.Push (X, 5);
    if Stacks.More_Stuff.Peek = 5 then
        . . .

end Demo;
```

Следует заметить, что включение дочернего пакета в инструкцию спецификатора совместности контекста "**with**" автоматически подразумевает включение в инструкцию "**with**" всех пакетов-родителей. Однако, инструкция спецификатора использования контекста "**use**" таким образом не работает. То есть, область видимости может быть получена пакетом только в базисе пакета.

```
with Stacks.More_Stuff; use Stacks; use More_Stuff;

procedure Demo is

    X : Stack;

begin
    Push (X, 5);
    if Peek (X) = 5 then
        . . .

end Demo;
```

Необходимо также заметить, что подпрограммы (процедуры и функции) могут быть дочерними модулями пакета (правила их использования достаточно очевидны). При этом, однако, сами подпрограммы не могут иметь дочерние модули.

8.3.2 Иерархия модулей как подсистема

Каждый программный модуль (процедура, функция пакет) должен иметь уникальное имя. Бывают случаи, когда происходит быстрое заполнение пространства имен. Например, при проектировании большой системы достаточно сложно обеспечить уникальность имен и при этом сохранить их смысловое значение, а в случаях когда разные части проекта разрабатываются разными программистами (или даже коллективами программистов) риск получения коллизии имен увеличивается еще больше.

В подобных ситуациях, используя концепцию дочерних модулей Ады, можно выделить каждую отдельно разрабатываемую подсистему в самостоятельную иерархию модулей. При этом, каждая подсистема будет иметь свой собственный корневой пакет с уникальным именем.

```
package Root is

   -- корневой пакет может быть пустым

end Root;
```

Поскольку имя корневого пакета уникально, то имена дочерних модулей иерархии также будут уникальны, что, в результате, минимизирует вероятность коллизии имен. Кроме того, такой прием является удобным средством разделения большого проекта на логически самостоятельные составные части.

Примером использования подобного подхода может служить стандартная библиотека Ады, которая предстявляется как набор дочерних модулей трех корневых пакетов: *Ada*, *Interfaces* и *System*.

8.3.3 Приватные дочерние модули (*private child units*)

Дочерние модули могут быть приватными. Такая концепция позволяет создавать дочерние модули, которые будут видимы только внутри иерархии родительского пакета. В этом случае сервисы для подсистемы могут быть инкапсулированы внутри приватных пакетов, используя наследуемые ими преимущества для компиляции и видимости.

В спецификацию приватных дочерних пакетов разрешается включать приватную часть спецификации их пакетов-родителей, поскольку такие дочерние модули — приватны. В обычном случае — это не разрешается, так как нарушает сокрытие деталей реализации родительской приватной части.

```
private package Stacks.Statistics is

   procedure Increment_Push_Count;

end Stacks.Statistics;
```

Процедура *Stacks.Statistics.Increment_Push_Count* могла бы быть вызвана внутри реализации пакета *Stacks*. Такая процедура не будет доступна ни одному внешнему, по отношению к этой иерархии модулей, клиенту.

Глава 9

Переименования

Ада предоставляет программисту возможность осуществлять переименования. Следует заметить, что переименование иногда вызывает споры в организациях программирующих на Аде. Некоторым людям переименование нравится, а другим — нет. Существует несколько важных вещей, которые необходимо понять:

- Переименование не создает нового пространства для данных. Оно просто создает новое имя для уже присутствующей сущности.
- Не следует постоянно переименовывать одно и то же. Этип можно запутать всех, включая самого себя.
- Переименование необходимо использовать для упрощения кода. Введение нового имени, в некоторых случаях, делает код более легко читаемым.

При использовании переименований следует учитывать, что частое переименование объектов и их значений может создать трудности в понимании исходного текста. Хотя каждое новое имя может иметь определенный смысл в контексте нового пакета, при большом количестве последующих переименований становиться трудно отследить имя оригинала.

9.1 Уменьшение длин имен

Переименование может быть полезно в случае наличия длинных имен пакетов:

```
with  Ada.Text_IO;
with  Ada.Integer_Text_IO;

procedure Gun_Aydin is

    package TIO renames  Ada.Text_IO;
    package IIO renames  Ada.Integer_Text_IO;

    . . .
```

В этом случае, для внутреннего использования, длинное имя "Ada.Text_IO" переименовано в короткое имя "TIO", а длинное имя "Ada.Integer_Text_IO" переименовано в короткое имя "IIO".

9.2 Переименование знаков операций

В некоторых случаях, знак операции для типа описанного в пакете, который указан в спецификаторе контекста "**with**", не является непосредственно видимым. В действительности, правила Ады заключаются в том, что сущность, находящаяся в контексте, не будет непосредственно видимой до тех пор, пока не будет явно указано, что она непосредственно видима. Спецификатор использования "**use**" для пакета всегда делает непосредственно видимыми знаки операций операции для типа описанного в пакете, однако, спецификатор использования "**use**" одновременно делает непосредственно видимыми все публично доступные ресурсы пакета, что может оказаться не желательным.

Переименование позволяет явно импортировать только те знаки операций, которые реально необходимы. При этом, видимость всех остальных ресурсов пакета остается не тронутой. Следующий пример показывает как это можно выполнить:

```
with  Ada.Text_IO;

procedure diamond1 is

    package TIO renames  Ada.Text_IO;

    function "+" (L, R: TIO.Count) return TIO.Count renames TIO."+";
    function "-" (L, R: TIO.Count) return TIO.Count renames TIO."-";

    . . .
```

Использование знаков операций облегчается при предварительном планировании использования знаков операций в процессе разработки пакета. В следующем примере знаки операций переименовываются во вложенном пакете, который, в последствии, может быть сделан непосредственно выдимым с помощью спецификатора использования "**use**":

```
package Nested  is

    type T1     is private;
    type Status is (Off, Low, Medium, Hight);

    package Operators is

        function ">=" (L, R: Status) return Boolean renames Nested.">=";
        function "="  (L, R: Status) return Boolean renames Nested."=";

    end Operators;

private
    type T1 is ...
    . . .
end Nested;
```

Показанный выше, вложенный пакет может быть сделан доступным путем указания спецификатора контекста "**with** Nested;", и последующего спецификатора использования "**use** Nested.Operators;" следующим образом:

```
with Nested;

procedure Test_Nested is

    use Nested.Operators;
    . . .
begin
    . . .
```

Возможно, что не все одобрительно встретят подобную технику, однако она упрощает использование инфиксной формы знаков операций, поскольку позволяет отказаться от локального переименования. Следует заметить, что такое решение будет лучше чем использование спецификатора использования типа "**use type**", поскольку делает видимым только ограниченное множество знаков операций. Однако, такой подход требует дополнительных усилий от разработчика пакета.

9.3 Переименование исключений

В некоторых случаях полезно осуществить локальное переименование исключения:

```
with  Ada.IO_Exceptions;

package My_IO is

    Data_Error: exception renames Ada.IO_Exceptions.Data_Error;
```

```
    . . .
end My_IO;
```

9.4 Переименование компонентов

Наиболее часто забываемым свойством переименования Ады является возможность предоставления собственного имени определенному компоненту составного типа:

```
with  Ada.Text_IO;

package Rename_A_Variable is

    Record_Count: renames Ada.Text_IO.Count;
    . . .
end Rename_A_Variable;
```

9.4.1 Переименование отрезка массива

Предположим, что у нас есть следующая строка:

```
Name  : String (1..60);
```

Причем, отрезок "(1..30)" — это фамилия (*last name*), отрезок "(31..59)" — имя (*first name*), символ в позиции "60" — это инициал отчества (*middle name*). Используя переименования мы можем выполнить следующее:

```
declare
    Last    : String renames Name (1..30);
    First   : String renames Name (31..59);
    Middle  : String renames Name (60..60);
begin
    Ada.Text_IO.Put_Line (Last);
    Ada.Text_IO.Put_Line (First);
    Ada.Text_IO.Put_Line (Middle);
end;
```

В результате, каждый вызов "Put_Line" будет обращаться к именованному объекту, а не к диапазону индексов. При этом не осуществляется распределение дополнительного пространства для данных, а обеспечивается новое имя для доступа к уже существующим данным. Примечательно также, что объект сохраняет те же самые индексы.

9.4.2 Переименование поля записи

Предположим, что у нас имеются следующие описания:

```
subtype Number_Symbol     is Character range '0' .. '9';
subtype Address_Character is Character range
    Ada.Characters.Latin_1.Space .. Ada.Characters.Latin_1.LC_Z;

type  Address_Data  is array (Positive range <>) of Address_Character;
type  Number_Data   is array (Positive range <>) of Number_Symbol;

type  Phone_Number  is
    record
        Country_Code  : Number_Data (1..2);
        Area_Code     : Number_Data (1..3);
        Prefix        : Number_Data (1..3);
        Last_Four     : Number_Data (1..4);
    end record;

type  Address_Record  is
    record
        The_Phone          : Phone_Number;
```

```
        Street_Address_1   : Address_Data (1..30);
        Street_Address_2   : Address_Data (1..20);
        City               : Address_Data (1..25);
        State              : Address_Data (1..2);
        Zip                : Number_Data (1..5);
        Plus_4             : Number_Data (1..4);
    end record;


One_Address_Record   : Address_Record;
```

Используя переименование, мы можем переименовать один из внутренних компонентов переменной записи "One_Address_Record" типа "Address_Record", для непосредственного использования в программе. Например, мы можем переименовать "Area_Code" в инструкции блока:

```
declare
    AC: Number_Data renames One_Address_Record.The_Phone.Area_Code;
begin
    . . .
end;
```

Описание "AC" не требует никакого распределения дополнительного пространства данных. Вместо этого, оно локализует имя для компонента, который вложен в запись. При наличии компонентов записей с большим уровнем вложения, такой подход может оказаться весьма удобным.

9.5 Переименование библиотечного модуля

Предположим, что в нашей библиотеке есть пакет который часто используется, и предположим, что этот пакет имеет довольно длинное имя. Пользуясь переименованием, мы можем указать этот пакет в спецификаторе контекста "**with**", после чего, переименовать пакет с длинным именем, и скомпилировать полученный модуль с более коротким именем обратно в библиотеку. Например:

```
with    Graphics.Common_Display_Types;
package CDT renames Graphics.Common_Display_Types;
```

Далее, мы можем использовать библиотечный модуль "CDT", с более коротким именем, также как и библиотечный модуль "Graphics.Common_Display_Types". При этом следует избегать переименований, когда новое имя очень сильно отличается от оригинала.

Глава 10

Настраиваемые модули в языке Ада (*generics*)

Долгие годы одной из самых больших надежд программистов была надежда в многократном повторном использовании однажды написанного кода (похоже, . . . и осталась). Хотя библиотеки математических подпрограмм используются достаточно широко, следует заметить, что математика оказалась весьма удачной предметной областью в которой функции хорошо описаны и стабильны. Попытки разработать подобные библиотеки в других предметных областях имели достаточно ограниченный успех из-за неминуемого изменения алгоритмов обработки и типов данных в подпрограммах. Ада пытается помочь в решении этой проблемы предоставляя возможность производства кода, который в большей степени зависит не от специфики используемого типа данных, а от общности алгоритма обработки.

Как описано Найдичем (*Naiditch*), настраиваемые модули во многом подобны шаблонам стандартных писем. Шаблоны стандартных писем, в основном, являются законченными письмами, но с несколькими пустыми местами в которых необходимо произвести подстановку (например, дописать имя и адрес). Таким образом, шаблоны стандартных писем не могут быть отосланы до тех по пока не будут заполнены пустые места, которые должны содержать эту недостающую информацию.

Подобным образом, настраиваемые модули не могут быть непосредственно использованы. Мы создаем новую подпрограмму или пакет путем конкретизации (*instantiating*) настраиваемого модуля, или, другими словами, создаем из настраиваемого модуля экземпляр настроенного модуля. При конкретизации настраиваемого модуля мы должны предоставить недостающую информацию, которая может быть информацией о типе, значении или даже подпрограмме.

10.1 Общие сведения о настраиваемых модулях

В языке Ада, любой программный модуль (подпрограмма или пакет) может быть настраиваемым модулем. Такой настраиваемый модуль используется для создания экземпляра кода, который будет работать с фактическим типом данных. Требуемый тип данных передается как параметр настройки при конкретизации настраиваемого модуля (создании экземпляра настроенного модуля). Как правило, описание настраиваемого модуля представлено двумя частями: спецификацией настраиваемого модуля и телом настраиваемого модуля. Спецификация описывает интерфейс настраиваемого модуля, а тело содержит детали его внутренней реализации.

Однажды скомпилированные настраиваемые модули помещаются в библиотеку Ады и могут быть указаны в инструкции спецификатора совместности контекста "**with**" в других компилируемых модулях. При этом, следует заметить, что настраиваемые модули не могут быть указаны в инструкции спецификатора использования контекста "**use**".

После указания настраиваемого модуля в спецификаторе контекста "**with**", программный модуль (подпрограмма, пакет или другой настраиваемый модуль) осуществляет конкретизацию настраиваемого модуля, то есть, создает экземпляр настроенного модуля из настраиваемого модуля. После этого, экземпляр настроенного модуля (конкретизированная подпрограмма или пакет) может быть сохранен в библиотеке Ады для последующего использования.

В качестве простого примера использования настраиваемого модуля, рассмотрим конкретизацию стандартного настраиваемого пакета *Integer_IO*:

```
with Ada.Text_IO;          use Ada.Text_IO;

package Int_IO is new Integer_IO (Integer);
```

Получившийся экземпляр настроенного модуля (пакет *Int_IO*), в последствии, может быть помещен в инструкции спецификации контекста "**with**" и "**use**" любого программного модуля.

Следует заметить, что стандартный настраиваемый пакет *Integer_IO* таким же образом может быть конкретизирован при использовании других типов.

```
with Ada.Text_IO;      use Ada.Text_IO;
with Accounts;         use Accounts;

package Account_No_IO is new Integer_IO (Account_No);
```

10.1.1 Настраиваемые подпрограммы

Рассмотрим простой пример описания настраиваемой подпрограммы. Однажды скомпилированная, она в последствии может быть помещена в инструкцию спецификатора совместности контекста "**with**" в других программных модулях. Напомним также, что настраиваемый модуль нельзя указывать в инструкции спецификатора использования контекста "**use**".

Спецификация модуля настраиваемой подпрограммы содержит ключевое слово "**generic**", после которого следует список формальных параметров настройки и далее — спецификация процедуры:

```
generic
    type Element is private;   -- Element – это параметр настраиваемой
                               -- подпрограммы

procedure Exchange (A, B : in out Element);
```

Тело настраиваемой процедуры описывает реализацию алгоритма работы процедуры и представляется как отдельно компилируемый модуль. Примечательно, что оно идентично не настраиваемой версии.

```
procedure Exchange (A, B  : in out Element) is

    Temp : Element;

begin
    T := Temp;
    T := B;
    B := Temp;
end Exchange;
```

Код, показанный выше, является простым шаблоном для процедуры, которая может быть реально создана. Следует обратить внимание на то, что этот код не может быть непосредственно вызван. Это подобно описанию типа, поскольку не производится никакого распределения пространства памяти или генерации машинного кода.

Для создания реальной процедуры, то есть, экземпляра процедуры которую можно вызвать, необходимо выполнить конкретизацию настраиваемой процедуры:

```
procedure Swap is new Exchange (Integer);
```

Следует заметить, что в показанном выше примере конкретизация настраиваемой процедуры осуществлена с использованием простого позиционного сопоставления формального и фактического параметров настройки. В дополнение к позиционному сопоставлению, Ада позволяет использовать именное сопоставление формальных и фактических параметров настройки (при большом количестве параметров настройки, именное сопоставление улучшает читабельность). Показанную выше конкретизацию настраиваемой процедуры можно осуществить с помощью использования именного сопоставления следующим образом:

```
procedure Swap is new Exchange (Element => Integer);
```

Теперь мы имеем процедуру "Swap" которая меняет местами переменные целого типа "**Integer**". Здесь, "**Integer**" является фактическим параметром настройки, а "Element" — формальным параметром настройки.

Процедура "Swap" может быть вызвана (и она будет вести себя) как будто она была описана следующим образом:

```
procedure Swap (A, B  : in out Integer) is

    Temp : Integer;

begin

    . . .

end Swap;
```

Таких процедур "Swap" можно создать столько, сколько необходимо.

```
procedure Swap is new Exchange (Character);
procedure Swap is new Exchange (Element => Account); -- ассоциация по имени
```

В этом случае будет нормально использоваться совмещение имен процедур. Компилятор будет определять к какой конкретно процедуре производится вызов используя информацию о типе параметра.

10.1.2 Настраиваемые пакеты

Пакеты также могут быть настраиваемыми. Следующая спецификация настраиваемого пакета достаточно традиционна:

```
generic
    type Element is private; -- примечательно, что это параметр
                             -- настройки
package Stacks is
    procedure Push (E : in Element);
    procedure Pop (E : out Element);
    function Empty return Boolean;
private
    The_Stack : array (1..200) of Element;
    top       : Integer range 0..200 := 0;
end Stacks;
```

Сопутствующее тело пакета может иметь подобный вид:

```
package body Stacks is

    procedure Push (E : in Element) is
        . . .

    procedure Pop (E : out Element) is
        . . .

    function Empty return Boolean is
        . . .

end Stacks;
```

В качестве элемента настройки, необходимо просто указать любой экземпляр типа данных.

10.1.3 Дочерние настраиваемые модули

Настраиваемые пакеты, подобно обычным пакетам Ады, могут иметь дочерние модули. При этом, следует заметить, что такие дочерние модули также должны быть настраиваемыми модулями.

В качестве примера, предположим, что нам необходимо расширить настраиваемый пакет *Stacks*, который был показан в примере выше (см. 9.1.2). Допустим, что нам необходимо добавить функцию "Top",

которая возвращает объект находящийся в вершине стека, но при этом не удаляет его из стека. Чтобы решить эту задачу, мы можем, для настраиваемого пакета *Stacks*, описать дочерний настраиваемый пакет *Stacks.Additions*. Спецификация *Stacks.Additions* может выглядеть следующим образом:

```
generic
package Stacks.Additions is

    function Top return Element;

end Stacks.Additions;
```

Примечательно, что дочерний настраиваемый модуль "видит" все компоненты своего родителя, включая все параметры настройки.

Тело дочернего настраиваемого модуля *Stacks.Additions* может иметь следующий вид:

```
package body Stacks.Additions is

    function Top return Element is
        . . .

end Stacks.Additions;
```

Ниже демонстрируется пример конкретизации настраиваемых модулей *Stacks* и *Stacks.Additions*.

Конкретизация модуля *Stacks* формирует пакет *Our_Stacks*, что имеет вид:

```
with  Stacks;

package Our_Stack is new Stack (Integer);
```

Конкретизация модуля *Stacks.Additions* формирует пакет *Our_Stack_Additions*, что имеет вид:

```
with  Our_Stack, Stacks.Additions;

package Our_Stack_Additions is new Stacks.Additions;
```

Примечательно, что настраиваемый дочерний модуль рассматривается как описанный внутри настраиваемого родителя.

10.2 Параметры настройки для настраиваемых модулей

Существует три типа параметров для настраиваемых модулей:

- параметры-типы
- параметры-значения
- параметры-подпрограммы

Необходимо заметить, что до настоящего момента в примерах мы рассматривали только параметры-типы.

10.2.1 Параметры-типы

Не смотря на привлекательные свойства настраиваемых модулей, при определении характера задач, решаемых с помощью настраиваемого модуля, необходимо иметь возможность накладывать некоторые ограничения. Допустим, что некоторые настраиваемые модули предусматривают возможность суммирования массива чисел. Очевидно, что это характерно только для чисел, и мы не можем производить суммирование записей. Следовательно, для того, чтобы защититься от конкретизации настраиваемых модулей с указанием не подходящих типов данных, требуется возможность в установке некоторых ограничений. Кроме того, желательно чтобы компилятор также мог осуществлять проверку того, что мы не выполняем какие-нибудь не допустимые действия с переменной внутри кода настраиваемого модуля, например, чтобы он не разрешал использовать атрибут "'Pred" для записей.

Решение таких задач обеспечивается механизмом который основан на виде спецификации формального параметра-типа. Таким образом, спецификация формального параметра-типа определяет категорию типов, которые могут быть использованы при конкретизации настраиваемого модуля, а также те действия, которые можно осуществлять над формальным параметром внутри настраиваемого модуля. Ниже показан общий список вариантов спецификаций формальных параметров-типов и различные ограничения, накладываемые на выбор фактических параметров-типов настраиваемых модулей.

```
type T is limited private     -- тип T – любой тип
type T is private             -- тип T – любой не лимитированный тип


type T is (<>)                -- тип T любой дискретный тип
                              --   (целочисленный или перечислимый)
type T is range <>            -- тип T любой целочисленный тип
type T is mod <>              -- тип T любой модульный целочисленный тип


type T is digits <>           -- тип T любой вещественный тип с плавающей точкой
type T is delta <>            -- тип T любой вещественный тип с фиксированной точкой
type T is delta <> digits <>  -- тип T любой вещественный децимальный тип


type T is access Y;           -- тип T любой ссылающийся на Y ссылочный тип
type T is access all Y;       -- тип T любой "access all Y" ссылочный тип
type T is access constant Y;  -- тип T любой "access constant Y" ссылочный тип
          -- примечание: тип Y может быть предварительно описанным
          --             настраиваемым параметром


type T is array (Y range <>) of Z; -- тип T любой неограниченный массив элементов типа Z
                                   --   у которого Y – подтип индекса
type T is array (Y) of Z;          -- тип T любой ограниченный массив элементов типа Z
                                   --   у которого Y – подтип индекса
          -- примечание: тип Z (тип компонента фактического массива)
          --             должен совпадать с типом формального массива.
          --             если они не являются скалярными типами,
          --             то они оба должны иметь тип
          --             ограниченного или неограниченного массива


type T is new Y;                          -- тип T любой производный от Y тип
type T is new Y with private;             -- тип T любой не абстрактный тэговый тип
                                          --   производный от Y
type T is abstract new Y with private;    -- тип T любой тэговый тип производный от Y
type T is tagged private;                 -- тип T любой не абстрактный не лимитированый
                                          --   тэговый тип
type T is tagged limited private;         -- тип T любой не абстрактный тэговый тип
type T is abstract tagged private;        -- тип T любой не лимитированый тэговый тип
type T is abstract tagged limited private; -- тип T любой тэговый тип
```

Для того чтобы лучше понять правила управляющие конкретизацией для параметров типа массив, рассмотрим следующий настраиваемый пакет:

```
generic
    type Item   is private;
    type Index  is (<>);
    type Vector is array (Index range <>) of Item;
    type Table  is array (Index) of Item;
package P is . . .
```

и типы:

```
type Color is (red, green, blue);
type Mix is array (Color range <> ) of Boolean;
type Option is array (Color) of Boolean;
```

тогда, "Mix" может соответствовать "Vector", а "Option" может соответствовать "Table".

```
package R is new P (Item   => Boolean,
                    Index  => Color,
                    Vector => Mix,
                    Table  => Option);
```

10.2.2 Параметры-значения

Параметры-значения позволяют указывать значения для переменных внутри настраиваемого модуля:

```
generic

    type Element is private;
    Size: Positive := 200;

package Stacks is

    procedure Push...
    procedure Pop...
    function Empty return Boolean;

end Stacks;

package body Stacks is

    Size : Integer;
    theStack : array (1..Size) of Element;

        . . .
```

Тогда, создать экземпляр настраиваемого модуля можно одним из следующих способов:

```
package Fred is new Stacks (Element => Integer, Size => 50);

package Fred is new Stacks (Integer, 1000);

package Fred is new Stacks (Integer);
```

Следует обратить внимание на то, что при конкретизации настраиваемого модуля фактический параметр-значение должен быть обязательно указан только в случаях когда для формального параметра-значения не представлено значение по умолчанию.

В качестве параметров-значений допускается использование строк.

```
generic

    type Element is private;
    File_Name : String;

package ....
```

Примечательно, что параметр "File_Name", имеющий строковый тип "String", — не ограничен (*not constrained*). Это идентично строковым параметрам для подпрограмм.

10.2.3 Параметры-подпрограммы

В качестве параметра настройки для настраиваемого модуля может быть передана подпрограмма. Необходимость в параметрах-подпрограммах чаще всего возникает когда какой-либо формальный параметр-тип настраиваемого модуля описан как приватный или лимитированный приватный тип. В таких случаях, Ада накладывает традиционные ограничения на использование операций над экземплярами данного типа внутри тела настраиваемого модуля. Однако, при этом может возникнуть необходимость в осуществлении сравнения или выполнения проверки на равенство значений данного типа, внутри тела настраиваемого модуля.

Следовательно, параметры-подпрограммы являются механизмом, который предоставляет для компилятора информацию о том как осуществлять эти действия.

В качестве примера рассмотрим типичный случай требующий применение параметров-подпрограмм. Предположим, что в качестве одного из параметров настраиваемого модуля используется лимитированый приватный тип. Тогда, для этого лимитированного приватного типа, с помощью параметров-подпрограмм, можно осуществить передачу операций: проверка на равенство и присваивание.

```
generic

    type Element is limited private;
    with function "=" (E1, E2 : Element) return Boolean;
    with procedure Assign (E1, E2 : Element);

package Stuff is . . .
```

Конкретизация такого настраиваемого модуля может иметь вид:

```
package Things is new Stuff (Person, Text."=", Text.Assign);
```

Для формального параметра-подпрограммы может быть указана подпрограмма используемая по умолчанию. Тогда, при конкретизации настраиваемого модуля, фактический параметр-подпрограмма может не указываться. Предположим, что у нас есть подпрограмма, спецификация которой имеет вид:

```
procedure My_Assign (E1, E2 : Person);
```

Тогда, при описании формального параметра-подпрограммы "Assign", мы можем указать процедуру "My_Assign" как подпрограмму которую необходимо использовать по умолчанию следующим образом:

```
generic

    type Element is limited private;
    with function "=" (E1, E2 : Element) return Boolean;
    with procedure Assign (E1, E2 : Element) is My_Assign (E1, E2 : Person);

package Stuff is . . .
```

В результате, конкретизация настраиваемого модуля, с использованием подпрограммы заданной по умолчанию, может иметь следующий вид:

```
package Things is new Stuff (Person, Text."=");
```

И наконец, при описании спецификации формального параметра-подпрограммы, можно указать, что выбор процедуры по умолчанию должен осуществляться согласно традиционных правил выбора подпрограмм. Для функции, реализующей действие знака проверки на равенство ("=") мы можем указать это следующим образом:

```
generic

    type Element is limited private;
    with function "=" (E1, E2 : Element) return Boolean is <>;
    . . .
```

Теперь, если при конкретизации настраиваемого модуля для функции "=" не будет представлена соответствующая функция, то будет использоваться функция проверки на равенство по умолчанию, выбранная в соответствии с фактическим типом "Element". Например, если фактический тип "Element" — тип "**Integer**", то будет использоваться обычная, для типа "**Integer**", функция "=".

10.3 Преимущества и недостатки настраиваемых модулей

В заключение обсуждения настраиваемых модулей Ады необходимо отметить преимущества и недостатки использования данной концепции.

Основным преимуществом использования настраиваемых модулей является то, что они оказывают значительное содействие в многократном повторном использовании ранее разработанных и отлаженных алгоритмов. Действительно, настраиваемые модули позволяют разработчику однажды написать и отладить алгоритмы подпрограмм для обработки объектов тип которых в последствии будет указываться пользователями этих подпрограмм.

Однако, применение настраиваемых модулей не лишено недостатков. Разработка алгоритмов для настраиваемых модулей требует более тщательного внимания, что в результате является дополнительной нагрузкой для их автора. Также необходимо заметить, что реализация настраиваемой процедуры, функции или пакета может оказаться не столь эффективной как непосредственная реализация. Компилятор может генерировать один и тот же код для всех экземпляров настроенных процедур, функций или пакетов, не зависимо от фактически обрабатываемых данных.

Глава 11

Исключения

Как это не печально, но процесс разработки и эксплуатации любого программного обеспечения всегда сочетается с процессом поиска и исправления ошибок. Все ошибки, возникающие в программах на языке Ада, можно разделить на два класса:

1. ошибки, которые обнаруживаются на этапе компиляции программы
2. ошибки, которые обнаруживаются во время выполнения программы

Таким образом, не смотря на то, что одной из целей при разработке Ады была задача максимально обеспечить возможность ранней диагностики и обнаружения ошибок, то есть обнаружение ошибок на стадии компиляции программы, бывают случаи когда ошибки возникают и во время выполнения программы.

В Аде, ошибочные или другие исключительные ситуации, возникающие в процессе выполнения программы, называются исключениями. Это значит, что при возникновении ошибочной ситуации, во время выполнения программы, вырабатывается сигнал о наличии исключения. Такое действие называют возбуждением (генерацией или порождением) исключения и понимают как приостановку нормального выполнения программы для обработки соответствующей ошибочной ситуации. В свою очередь, обработка исключения — это выполнение соответствующего кода для определения причины возникновения ошибочной ситуации которая привела к возбуждению исключения, а также, при возможности, устранение причины возникновения ошибки и/или выполнение других корректирующих действий.

Примечательно, что идея использования механизма исключений — это тема многих споров о том, что исключения являются или путем "ленивого" программирования, без достаточного анализа проблем и сопутствующих условий приводящих к возникновениям ошибок, или обычным видом управляющих структур, которые могут быть использованы для достижения некоторых эффектов. Тем не менее, хотя эти споры не прекращаются и в настоящее время, следует обратить внимание на то, что механизм исключений благополучно заимствован некоторыми современными реализациями других языков программирования (например, широко известная реализация языка *Object Pascal* фирмы *Borland*).

Все исключения языка программирования Ада можно разделить на стандартно предопределенные исключения и исключения определяемые пользователем.

11.1 Предопределенные исключения

Существует пять исключений которые стандартно предопределены в языке программирования Ада:

Constraint_Error — Ошибка ограничения

Numeric_Error — Ошибка числа

Program_Error — Ошибка программы

Storage_Error — Ошибка памяти

Tasking_Error — Ошибка задачи

11.1.1 Исключение *Constraint_Error*

Исключение *Constraint_Error* возбуждается в следующих случаях:

- при попытке нарушения ограничения диапазона, ограничения индекса или ограничения дискриминанта
- при попытке использования компонента записи, не существующего при текущем значении дискриминанта
- при попытке использования именуемого или индексируемого компонента, отрезка или атрибута объекта, обозначенных ссылочным значением, если этот объект не существует, поскольку ссылочное значение равно "**null**"

Рассмотрим пример:

```
procedure Constraint_Demo is

    X : Integer range 1..20;
    Y : Integer;

begin
    Put ("enter a number ");
    Get (Y);
    X := Y;
    Put ("thank you");
end Constraint_Demo;
```

Если пользователь вводит значение выходящее за диапазон значаний "1..20", то нарушается ограничение диапазона значений для "X", и происходит исключение *Constraint_Error*. Поскольку в этом примере не предусмотрен код, который будет обрабатывать это исключение, то выполнение программы будет завершено, и окружение времени выполнения Ады (Ада-система) проинформирует пользователя о возникшей ошибке. При этом, строка

```
    Put ("thank you");
```

выполнена не будет. Таким образом, при возникновении исключения, остаток, выполняющегося в текущий момент блока, будет отброшен.

Рассмотрим пример в котором выполняется нарушение ограничения диапазона индексных значений для массива:

```
procedure Constraint_Demo2 is

    X : array (1..5) of Integer := (1, 2, 3, 4, 5);
    Y : Integer := 6;

begin
    X (Y) := 37;
end Constraint_Demo2;
```

Здесь, исключение *Constraint_Error* будет генерироваться когда мы будем пытаться обратиться к несуществующему индексу массива.

11.1.2 Исключение *Numeric_Error*

Исключение *Numeric_Error* возбуждается в случае когда предопределенная численная операция не может предоставить математически корректный результат Это может произойти при арифметическом переполнении, делении на нуль, а также не возможности обеспечить требуемую точность при выполнении операций с плавающей точкой. Следует заметить, что в Ada95 *Numeric_Error* переопределена таким образом, что является тем же самым, что и *Constraint_Error*.

```
procedure Numeric_Demo is

    X : Integer;
    Y : Integer;

begin
    X := Integer'Last;
    Y := X + X;              -- вызывает Numeric_Error
end Numeric_Demo;
```

11.1.3 Исключение *Program_Error*

Исключение *Program_Error* возбуждается в следующих случаях:

- при попытке вызова подпрограммы, активизации задачи или конкретизации настройки, если тело соответствующего модуля еще не обработано.
- если выполнение функции достигло завершающего "**end**" так и не встретив инструкцию возврата ("**return** ... ")
- при межзадачном взаимодействии во время выполнения инструкции отбора с ожиданием ("**select** ... "), когда все альтернативы закрыты и отсутствует раздел "**else**"

Кроме того, это исключение может возбуждаться в случае возникновения ошибки элаборации.

```
procedure Program_Demo is

    Z : Integer;

    function Y (X : Integer) return Integer is
    begin
        if X < 10 then
            return X;
        elsif  X < 20 then
            return X
        end if;
    end Y;                  -- если мы попали в эту точку, то это значит,
                            -- что return не был выполнен

begin
    Z := Y (30);
end Program_Demo;
```

11.1.4 Исключение *Storage_Error*

Исключение *Storage_Error* возбуждается в следующих случаях:

- при попытке размещения динамического объекта обнаруживается, что нет достаточного пространства в динамической памяти (куче) которая выделена для задачи
- при исчерпании памяти выделенной для набора (коллекции) динамически размещаемых объектов
- при обращении к подпрограмме, когда израсходовано пространство стека

11.1.5 Исключение *Tasking_Error*

Исключение *Tasking_Error* возбуждается в случаях межзадачного взаимодействия. Оно может быть возбуждено при возникновении какого-либо исключения внутри задачи которая в текущий момент времени принимает участие в межзадачном взаимодействии или когда задача пытается организовать рандеву с абортированной задачей.

11.2 Исключения определяемые пользователем

Механизм исключений Ады был бы не столь полным если бы он позволял использовать только стандартно предопределенные исключения. Поэтому, в дополнение к стандартно предопределенным исключениям, Ада дает программисту возможность описывать свои собственные исключения и, в случае необходимости, выполнять их возбуждение.

11.2.1 Описание исключения пользователя

Описания пользовательских исключений должны размещаться в декларативной части кода, то есть там где обычно размещаются описания (например, в спецификации пакета). Форма описания исключений достаточно тривиальна и имеет следующий вид:

```
My_Very_Own_Exception : exception;
Another_Exception     : exception;
```

11.2.2 Возбуждение исключений

Указание возбуждения исключения достаточно простое. Для этого используется инструкция "**raise**". Например:

```
raise Numeric_Error;
```

Сгенерированное таким образом исключение не будет ничем отличаться от "истинного" исключения *Numeric_Error*.

11.3 Обработка исключений

В настоящий момент мы уже знаем стандартно предопределенные исключения Ады, знаем как описывать исключения и знаем как их возбуждать (и свои, и предопределенные). Однако, весь код примеров, которые мы рассматривали, не выполнял никакой обработки исключений. Во всех рассмотренных случаях, после возникновения исключения, происходило простое завершение выполнения программы кодом библиотеки времени выполнения Ады. Таким образом, для того, чтобы извлечь некоторую пользу из механизма исключений, необходимо знать как анализировать ситуацию в случае возникновения исключения. Проще говоря — необходимо рассмотреть как и где писать обработчики исключений.

11.3.1 Обработчики исключений

Обработчик исключения может размещаться в конце тела подпрограммы, пакета или настраиваемого модуля, в конце тела задачи или входа, а также в конце инструкции блока или инструкции принятия ("**accept**"). Заметим, что обработчик исключения не является обязательной частью этих конструкций.

Рассмотрим следующий пример:

```
declare

    X : Integer range 1..20;

begin
    Put ("please enter a number ");
    Get (X);
    Put ("thank you");

exception
    when Constraint_Error =>
        Put ("that number should be between 1 and 20");
    when others =>
        Put ("some other error occurred");
end;
```

Здесь описаны два обработчика. В одном выполняется обработка только исключений ограничения (*Constraint_Error*). Второй обработчик выполняет обработку всех остальных исключений ("**others**"). Таким образом, если пользователь вводит число в диапазоне от 1 до 20, то ошибки не происходит и появляется сообщение "`thank you`". В противном случае, перед завершением выполнения появляется сообщение обработчика исключения *Constraint_Error*: "`that number should be between 1 and 20`". В случае возникновения какого-либо другого исключения появится сообщение от второго обработчика: "`some other error occurred`".

Можно описать обработчик исключений так, чтобы он обрабатывал несколько указанных исключений. Для выполнения этого, исключения должны разделяться символом "|":

```
    . . .
exception
    . . .
    when Constraint_Error | Storage_Error =>
        . . .
```

Также следует заметить, что обработчик "**when others**" всегда должен быть последним в списке обработчиков исключений.

Если мы хотим чтобы пользователь продолжал ввод чисел до тех пор пока не пропадет ошибка ограничения, мы можем переписать предыдущий пример подобным образом:

```
loop
    declare
        . . .

    begin
        . . .

        Get (X);
        exit;

    exception
        when Constraint_Error =>
          Put ("that number ...
    end;

    . . .           -- здесь будет продолжено выполнение
                    -- после возникновения исключения
                    -- и обработки его обработчиком
end loop;
```

Кроме того, этот пример показывает точку в которой будет продолжено выполнение инструкций после возникновения исключения и его обработки обработчиком.

11.3.2 Распространение исключений

Для того, чтобы точно знать в каком месте должен быть расположен соответствующий обработчик исключения, необходимо понимать как при возникновении исключения, во время работы программы, происходит поиск обработчика. Этот процесс поиска называется распространением исключений.

Если исключение не обрабатывается в подпрограмме в которой это исключение возникло, то оно распространяется в подпрограмму которая вызвала текущую подпрорамму (на уровень выше). После чего, обработчик исключения ищется в вызвавшей подпрограмме. Если обработчик исключения не найден, то исключение распространяется дальше (еще на уровень выше). Это продолжается до тех пор пока не будет найден обработчик возникшего исключения или не будет достигнут уровень выполнения текущей задачи (наивысший уровень). Если в случае достижения уровня выполнения текущей задачи (то есть наивысшего уровня) обработчик исключения не будет найден, то текущая задача будет аварийно завершена (другими словами, абортирована). Если выполняется только одна задача, то библиотека времени выполнения Ады выполняет обработку возникшего исключения и аварийно завершает выполнение всей программы (другими словами, абортирует выполнение программы).

```
procedure Exception_Demo is

   ------------------------------------
   procedure Level_2 is
      -- здесь нет обработчика исключений
   begin
      raise Constraint_Error;
   end Level_2;

   ------------------------------------
   procedure Level_1 is
   begin
      Level_2;
   exception
      when Constraint_Error =>
         Put ("exception caught in Level_1");
   end Level_1;

begin
   Level_1;

exception
   when Constraint_Error =>
      Put ("exception caught in Exception_Demo");
end Exception_Demo;
```

После запуска этой программы будет выдано только сообщение "`exception caught in Level_1`". Следовательно, обработанное исключение не распространяется дальше.

Модифицируем процедуру "Level_1" поместив инструкцию "**raise**" в ее обработчик исключения. Наш предыдущий пример будет иметь следующий вид:

```
procedure Exception_Demo is

   ------------------------------------
   procedure Level_2 is
      -- здесь нет обработчика исключений
   begin
      raise Constraint_Error;
   end Level_2;

   ------------------------------------
   procedure Level_1 is
   begin
      Level_2;
   exception
      when Constraint_Error =>
         Put ("exception caught in Level_1");
         raise;         -- регенерация текущего исключения;
                        -- дает возможность другим подпрограммам
                        -- произвести обработку возникшего
                        -- исключения
   end Level_1;

begin
   Level_1;

exception
   when Constraint_Error =>
      Put ("exception caught in Exception_Demo");
end Exception_Demo;
```

Теперь, инструкция "**raise**", помещенная в обработчик исключения, вызывает распространение исключения *Constraint_Error* на один уровень выше, то есть, к вызвавшей подпрограмме. Таким образом, исключение может быть получено и соответствующим образом обработано в каждой подпрограмме иерархии вызовов.

Инструкцию "**raise**" очень удобно использовать в секции "**others**" обработчика исключений:

```
    . . .
exception
    . . .
    when others =>
        raise;          -- регенерация текущего исключения;
                        -- дает возможность другим подпрограммам
                        -- произвести обработку возникшего
                        -- исключения
end;
```

В этом случае, соответствующее исключение будет продолжать генерироваться и распространяться до тех пор, пока не будет обработано надлежащим образом.

11.3.3 Проблемы с областью видимости при обработке исключений определяемых пользователем

Во всех предыдущих примерах, посвященных обработке исключений, были использованы стандартно определенные исключения Ады. Обработка исключений определяемых пользователем идентична обработке предопределенных исключений, однако, при этом могут возникать некоторые проблемы с областью видимости. Рассмотрим следующий пример:

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Demo is

    procedure Problem_In_Scope is
        Cant_Be_Seen : exception;
    begin
        raise Cant_Be_Seen;
    end Problem_In_Scope;

begin
    Problem_In_Scope;

exception
    when Cant_Be_Seen =>
        Put ("just handled an_exception");
end Demo;
```

Этот пример не корректен. Проблема в том, что область видимости исключения "Cant_Be_Seen" ограничивается процедурой "Problem_In_Scope", которая, собственно и является источником этого исключения. То есть, исключение "Cant_Be_Seen" не видимо и о нем ничего не известно за пределами процедуры "Problem_In_Scope". Поэтому, это исключение не может быть точно обработано процедурой "Demo".

Решить эту проблему можно использованием опции "**others**" в обработчике исключений внешней процедуры "Demo":

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Demo is

    procedure Problem_In_Scope is
            Cant_Be_Seen : exception;
    begin
            raise Cant_Be_Seen;
    end Problem_In_Scope;

begin
    Problem_In_Scope;

exception
    when others =>
        Put ("just handled some exception");
end Demo;
```

Другая проблема возникает тогда, когда в соответствии с правилами области видимости, исключение, описываемое в одной процедуре, перекрывает (или прячет) исключение, описываемое в другой процедуре:

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Demo is

    Fred : exception;             -- глобальное исключение

    ------------------------------------
    procedure P1 is
    begin
        raise Fred;
    end P1;

    ------------------------------------
    procedure P2 is

        Fred : exception;         -- локальное исключение

    begin
        P1;

    exception
        when Fred =>
            Put ("wow, a Fred exception");
    end P2;
    ------------------------------------

begin
    P2;

exception
    when Fred =>
        Put ("just handled a Fred exception");
end Demo;
```

Выводом такой процедуры будет "`just handled a Fred exception`". Исключение, обрабатываемое в процедуре "P2", будет локально описанным исключением. Такое поведение подобно ситуации с областью видимости обычных переменных.

Для решения этой проблемы, процедуру "P2" можно переписать следующим образом:

```
------------------------------------
procedure P2 is

    Fred : exception;

begin
    P1;

exception
    when Fred =>
        -- локальное исключение
        Put ("wow, an_exception");

    when Demo.Fred =>
        -- "более глобальное" исключение
        Put ("handeled Demo.Fred exception");
        raise;
end P2;
```

Теперь, обработчик исключения процедуры "P2" выдаст сообщение "`handeled Demo.Fred exception`" и, с помощью инструкции "**raise**", осуществит передачу исключения "Demo.Fred" в обработчик исключения процедуры "Demo", который, в свою очередь, выдаст сообщение "`just handled a Fred exception`".

11.3.4 Пакет *Ada.Exceptions*

Стандартный пакет *Ada.Exceptions* предоставляет некоторые дополнительные средства, которые могут быть использованы при обработке исключений.

Описанный в нем объект:

```
Event : Exception_Occurence;
```

и подпрограммы:

функция "Exception_Name (Event)" — возвращает строку имени исключения, начиная от корневого библиотечного модуля.

функция "Exception_Information (Event)" — возвращает строку детальной информации о возникшем исключении.

функция "Exception_Message (Event)" — возвращает строку краткого объяснения исключения.

процедура "Reraise_Occurence (Event)" — выполняет повторное возбуждение исключения "Event".

процедура "Reraise_Exception (E, Msg)" — выполняет возбуждение исключения "E" с сообщением "Msg".

Могут быть весьма полезны при необходимости обработки неожиданных исключений. В таких случаях можно использовать код который подобен следующему:

```
    . . .
exception
    . . .
    when The_Event: others =>
        Put ("Unexpected exception is ";
        Put (Exeption_Name(The_Event));
        New_Line;
```

11.4 Подавление исключений

11.4.1 Принципы подавления исключений

Как правило, существует два источника которые выполняют возбуждение исключений при обнаружении некоторых ошибочных условий. Один из них — это механизмы аппаратной проверки, которые зависят от конкретно используемого оборудования. Второй источник для нас более интересен, поскольку этим источником является дополнительный машинный код, который генерирует компилятор.

Поскольку генерация дополнительного кода выполняется компилятором, а все компиляторы языка Ада должны соответствовать стандартным требованиям, то должно быть обеспечено стандартное средство управления вставкой подобных проверок в результирующий машинный код. Таким средством Ады является директива компилятора "Supress" (подавление проверок).

Эта директива может быть размещена в том месте, где не требуется производить проверки. Подавление проверок будет распространяться до конца текущего блока (при этом используются обычные правила области видимости).

Директива "Supress" имеет большое количество опций, позволяющих подавлять проверки различного вида на уровне типа, объекта или на функциональном уровне. Следует заметить, что многогранность директивы "Supress" сильно зависит от реализации конкретного компилятора, и различные реализации компиляторов свободны в предоставлении (или игнорировании) любых свойств этой директивы.

Исключение *Constraint_Error* имеет несколько подавляемых проверок:

```
pragma Suppress (Access_Check);
pragma Suppress (Discriminant_Check);
pragma Suppress (Idex_Check);
pragma Suppress (Length_Check);
pragma Suppress (Range_Check);
pragma Suppress (Division_Check);
pragma Suppress (Owerflow_Check);
```

Исключение *Program_Error* имеет только одну подавляемую проверку:

```
pragma Suppress (Elaboration_Check);
```

Исключение *Storage_Error* также имеет только одну подавляемую проверку:

```
pragma Suppress (Storage_Check);
```

11.4.2 Выполнение подавления исключений

Мы можем подавить проверку исключений для индивидуального объекта:

```
pragma Suppress (Idex_Check, on => table);
```

Подавление проверки исключений также может относиться к какому-то определенному типу:

```
type Employee_Id is new Integer;
pragma Suppress (Range_Check, Employee_Id);
```

Более полным примером использования директивы "Supress" может служить код показанный ниже. В этом случае область действия директивы распространяется до конца блока.

```
declare

    pragma Suppress (Range_Check);
    subtype Small_Integer is Integer range 1..10;

    A : Small_Integer;
    X : Integer := 50;

begin
    A := X;
end;
```

Этот код не будет генерировать ошибок ограничения (*Constraint_Error*).

Глава 12

Организация ввода/вывода

В отличие от своего прародителя Паскаля, Ада не содержит жестко встроенных средств ввода/вывода. Вместо этого, для обеспечения поддержки ввода/вывода, Ада предусматривает простую, но эффективную коллекцию стандартных пакетов, средства которых концептуально подобны средствам ввода/вывода большинства современных расширений Паскаля. Преимуществом такого решения является то, что стандартно предопределенные пакеты ввода/вывода, при необходимости, могут быть заменены. Таким образом, высокопроизводительные приложения или системы использующие специфическое оборудование могут не использовать большинство стандартно предлагаемых средств ввода/вывода непосредственно (при этом следует учитывать, что может быть уменьшена или вовсе утрачена переносимость конечных программ). В подобных случаях, стандартные средства ввода/вывода помогают формировать однородный базис для построения эффективной подсистемы файлового обмена.

Средства ввода/вывода Ады, определяемые стандартом Ada83, обеспечивают возможность работы с текстовыми и двоичными данными. Стандарт Ada95 расширил эти средства возможностью использования гетерогенных потоков.

Прежде чем приступить к непосредственному обсуждению, необходимо заметить, что использование средств ввода/вывода Ады может показаться сначала достаточно не привычным. Причина такой непривычности заключается в том, что природа строгой типизации данных Ады накладывает свой отпечаток и на средства ввода/вывода, а это значит, что весь ввод/вывод в Аде также подвержен строгой типизации данных.

12.1 Текстовый ввод/вывод

Заметим, что хотя средства поддержки текстового ввода/вывода Ады не рассматривались до настоящего момента, некоторые ранее рассмотренные примеры уже применяли эти средства, осуществляя ввод/вывод с использованием стандартного устройства ввода/вывода. Теперь, необходимо рассмотреть эти механизмы более детально.

Как известно, текстовыми файлами являются файлы которые содержат символы. Как мы уже знаем, согласно стандарта Ada95, Ада обеспечивает поддержку двух символьных типов: "Character" и "Wide_Character". Поэтому, средства поддержки текстового ввода/вывода Ады подразделяются на средства поддержки ввода/вывода для символов типа "Character" и средства поддержки ввода/вывода для символов типа "Wide_Character".

Следует заметить, что стандартные пакеты поддержки текстового ввода/вывода для символов типа "Character" содержат в своем названии строку "Text_IO", а стандартные пакеты поддержки текстового ввода/вывода для символов типа "Wide_Character" содержат в своем названии строку "Wide_Text_IO".

Поскольку логика обработки текстового ввода/вывода для символов типа "Character" соответствует логике обработки текстового ввода/вывода для символов типа "Wide_Character", то для понимания организации поддержки текстового ввода/вывода Ады достаточно рассмотреть эти механизмы в контексте символов типа "Character".

12.1.1 Пакет *Ada.Text_IO*

Основой организации текстового ввода/вывода Ады является пакет *Ada.Text_IO* и коллекция его дочерних пакетов. Этот пакет обеспечивает средства, которые позволяют манипулировать текстовыми файлами. Примерами таких средств могут служить подпрограммы "Close", "Delete", "**Reset**", "Open", "Create"...

Главный тип данных пакета *Ada.Text_IO* — это лимитированный приватный тип "File_Type". Он является внутренним представлением файла. Стандарт Ada95 добавил тип "File_Access", как ссылку на тип "File_Type" (объекты имеющие тип "File_Access" часто называют дескрипторами файлов). При открытии или создании файла, производится ассоциирование между именем файла в системе и объектом типа "File_Type". Кроме того, при открытии или создании файла, необходимо указывать режим доступа к файлу:

In_File	— чтение файла
Out_File	— запись в файл
Append_File	— запись в конец существующего файла (Ada95)

Заметим, что эти значения имеют тип "File_Mode", который также описывается в пакете *Ada.Text_IO*. После этого, объект типа "File_Type" может быть использован для выполнения непосредственных обращений к файлу.

В приведенном ниже примере, процедура "Create" создает файл "*first_file.dat*", после чего, в этот файл процедурами "**Put**" и "New_Line" производится запись строки "It **is** my first **text file**!". В завершение, процедура "Close" закрывает ранее открытый файл.

```
with Ada.Text_IO;  use Ada.Text_IO;

procedure Demo_File_IO is

    My_File : Ada.Text_IO.File_type;

begin
    Create (File => My_File,
            Mode => Out_File,
            Name => "first_file.dat");

    Put(File => My_File,
        Item => "It is my first text file!");

    New_Line (My_File);

    Close (My_File);        -- требуется! Ада может не закрыть
                            -- открытый вами файл
end Demo_File_IO;
```

Программа, представленная в следующем примере, выполняет посимвольное чтение данных из одного файла ("`input.dat`") и посимвольную запись этих же данных в другой файл ("`output.dat`").

```
with Ada.Text_IO;          use Ada.Text_IO;

procedure Read_Write is

    Input_File  : File_type;
    Output_File : File_type;
    Char        : Character;

begin
    Open (Input_File, In_File, "input.dat");
    Create (Output_File, Out_File, "output.dat");

    while not End_Of_File (Input_File) loop
        while not End_Of_Line (Input_File) loop
            Get (Input_File, Char);
            Put (Output_File, Char);
        end loop;
        Skip_Line (Input_File);
        New_Line (Output_File);
    end loop;

    Close (Input_File);
    Close (Output_File);
end Read_Write;
```

Необходимо обратить внимание на то, что в таких языках программирования как Ада и Паскаль существует концепция терминатора строки, который не является обычным символом файла. Это значит, что понятие "конец строки" ("*End Of Line*", или сокращенно — *EOF*) Ады отличается от того, что принято в системах DOS, Windows и UNIX. В этих системах для обозначения конца строки используется обычный символ (символ "CR" — для UNIX, и символы: "CR", "LF" — для DOS и Windows), который может быть обработан обычными средствами символьной обработки.

Для того чтобы, при чтении из файла, процедура "**Read**" "прошла" этот терминатор, необходимо использовать процедуру "Skip_Line". Подобным образом, для того чтобы осуществить построчную запись выходного файла, должна быть использована процедура "New_Line".

Пакет *Ada.Text_IO* обеспечивает большое число процедур для выполнения различных файловых манипуляций. В качестве примера наиболее часто используемых процедур можно перечислить следующие процедуры:

Create	Создает файл с указанным именем и режимом использования. Примечательно, что если файл имеет строку *null*, то файл является временным и позднее будет удален.
Open	Открывает файл с указанным именем и режимом использования.
Delete	Удаляет указанный файл. При попытке удалить открытый файл происходит ошибка.
Reset	Возвращает позицию чтения (или записи) в начало файла.

К наиболее часто используемым функциям пакета *Ada.Text_IO*, которые возвращают статус системы файлового обмена, можно отнести следующие функции:

End_of_File	Возвращает истину если мы находимся в конце текущего файла.
End_of_Line	Возвращает истину если мы находимся в конце текущей строки текста.
Is_Open	Возвращает истину если текущий файл открыт.
Mode	Возвращает режим использования текущего файла.
Name	Возвращает строку имени текущего файла.

Заметим, что список перечисленных здесь подпрограмм далеко не полон и имеет лишь ознакомительный характер. Поэтому, для получения более подробных сведений лучше всего непосредственно обратиться к спецификации пакета *Ada.Text_IO*.

12.1.2 Искючения ввода/вывода

Все средства организации ввода/вывода Ады полагаются на механизм исключений для сообщения об ошибках, которые возникли в процессе выполнения какой-либо операции ввода/вывода данных. Описания исключений ввода/вывода находятся в пакете *Ada.IO_Exceptions* который имеет следующий вид:

```
package Ada.IO_Exceptions is
   pragma Pure (IO_Exceptions);

   Status_Error : exception; -- попытка обращения к файлу который не был открыт
                             --   или повторное открытие файла
   Mode_Error   : exception; -- попытка чтения из файла который открыт для записи
   Name_Error   : exception; -- попытка открытия не существующего файла
                             --   или попытка попытка создания файла
                             --   с не допустимым именем
   Use_Error    : exception; -- зависит от характеристик внешней среды;
                             --   например, при обращении к файлу
                             --   при отсутствии соответствующих
                             --   привелегий доступа
   Device_Error : exception; -- запрошенная операция ввода/вывода не может
                             --   быть выполнена по причине неисправности
                             --   оборудования
```

```
   End_Error    : exception; -- попытка чтения за концом файла
   Data_Error   : exception; -- прочитанный элемент не соответствует
                             --   ожидаемому типу данных
   Layout_Error : exception; -- при текстовом вводе/выводе;
                             --   как правило, при выполении операции
                             --   ввода/вывода которая нарушает
                             --   установленные диапазоны значений
                             --   форматирования текста
end Ada.IO_Exceptions;
```

Таким образом, попытка создать файл, который уже существует, вызывает исключение, также как и попытка открыть файл которого нет.

Показанная ниже процедура "Robust_Open" является примером того, как можно попытаться избежать подобные неприятности. Сначала, она пытается открыть указанный файл, и если это приводит к ошибке говорящей об отсутствии этого файла, то тогда она пытается его создать:

```
with Ada.Text_IO;          use Ada.Text_IO;

procedure Robust_Open (The_File : in out File_type;
                       Mode     : in     File_mode;
                       Name     : in     String) is
begin
    Open(The_File, Mode, Name);

exception
    when Name_Error =>
        Create (The_File, Mode, Name);
end Robust_Open;
```

Следует обратить внимание на то, что на корректность работы этого примера может оказать влияние одновременное выполнение нескольких процессов. Действительно, выполнение программы может быть приостановлено после попытки открыть файл, но до того как она попытается его создать. Следовательно, есть вероятность того, что за время простоя программы этот файл будет создан другим процессом.

Примером еще одной простой утилиты может служить логическая функция "File_Exists", которая позволяет осуществить проверку существования файла:

```
with Ada.Text_IO;          use Ada.Text_IO;

function File_Exists (Name : String) return Boolean is

    The_File : Ada.Text_IO.File_Type;

begin
    Open (The_File, In_File, Name);

    -- файл открылся, закрываем его и возвращаем "истину"
    Close (The_File);
    return True;

exception
    when Name_Error =>
        -- файл не окрылся, значит его нет
        return False;
end File_Exists;
```

Эта функция пытается открыть файл и если при этом не возникает исключения *Name_Error*, то это значит что файл существует, и функция возвращает значение "истина"(если файл уже открыт, то генерируется исключение *Use_Error*).

12.1.3 Файлы ввода/вывода по умолчанию

В тех случаях, когда для осуществления ввода/вывода нет четкого указания внешних файлов используются файлы ввода/вывода по умолчанию. При этом, соответствующие файл ввода и файл вывода по умолчанию

всегда открыты перед началом выполнения программы и связаны с двумя, определяемыми реализацией системы, внешними файлами. Другие достаточно широко распространенные названия этих файлов: стандартный ввод и стандартный вывод, или стандартное устройство ввода и стандартное устройство вывода. Стандарт Ada95, кроме этого, добавил аналогичную концепцию стандартного файла ошибок, позаимствовав эту идею от Unix/C.

Как правило, файл стандартного ввода — это клавиатура, а файлы стандартного вывода и стандартного файла ошибок — это дисплей. Пакет *Ada.Text_IO* предоставляет средства которые позволяют программно определять и переопределять установленные по умолчанию файлы ввода/вывода.

Функции пакета *Ada.Text_IO*, которые позволяют определить стандартные файл ввода, файл вывода и файл ошибок, а также файл ввода, файл вывода и файл ошибок используемые в текущий момент имеют соответственные имена:

```
Standard_Input
Standard_Output
Standard_Error

Current_Input
Current_Output
Current_Error
```

Такие функции возвращают значения типа "File_Type" или "File_Access" (используется механизм совмещения имен функций).

Для переопределения используемых в текущий момент файлов ввода, вывода и ошибок, пакет *Ada.Text_IO* обеспечивает процедуры со следующими именами:

```
Set_Input
Set_Output
Set_Error
```

В качестве параметра, эти процедуры используют значения типа "File_Type", причем, указываемый файл должен быть предварительно открыт.

12.1.4 Настраиваемые пакеты текстового ввода/вывода

Для поддержки текстового ввода/вывода численных данных и данных перечислимых типов Ада предусматривает набор настраиваемых пакетов текстового ввода/вывода:

```
Ada.Text_IO.Integer_IO      -- для целочисленных типов
Ada.Text_IO.Modular_IO      -- для модульных типов
Ada.Text_IO.Float_IO        -- для вещественных типов с плавающей точкой
Ada.Text_IO.Fixed_IO        -- для вещественных типов с фиксированной точкой
Ada.Text_IO.Decimal_IO      -- для децимальных типов
Ada.Text_IO.Enumeration_IO  -- для перечислимых типов
```

Примечательно, что все эти настраиваемые пакеты являются дочерними модулями пакета *Ada.Text_IO*.

Хотя эти настраиваемые модули предназначены для организации ввода/вывода значений различных типов, спецификации этих модулей достаточно подобны. Таким образом, спецификация любого из этих модулей, используя механизм совмещения имен подпрограмм, предоставляет описания трех версий настраиваемых процедур "**Get**" и трех версий настраиваемых процедур "**Put**", которые можно разделить попарно. Первая пара процедур "**Get**" и "**Put**" соответственно предназначена для чтения и записи значений с устройств стандартного ввода/вывода, вторая — для аналогичного взаимодействия с внешними файлами, а третья для преобразования значений в строковое представление и обратно. Для получения исчерпывающей информации лучше непосредственно обратиться к спецификациям этих настраиваемых модулей.

Рассмотрим пример использования настраиваемого пакета *Ada.Text_IO.Integer_IO*, который предназначен для организации ввода/вывода целочисленных типов. Предварительно, необходимо выполнить конкретизацию настраиваемого пакета *Ada.Text_IO.Integer_IO* для соответствующего целочисленного типа (в нашем случае выбран тип "**Integer**", для простоты), и получить экземпляр настроенного модуля:

```
with Ada.Text_IO;          use Ada.Text_IO;

package Int_IO is new Integer_IO (Integer);
```

Теперь мы можем использовать полученный экземпляр настроенного модуля *Int_IO*, например, следующим образом:

```
with Ada.Text_IO;   use Ada.Text_IO;
with Int_IO;        use Int_IO;       -- указываем экземпляр настроенного модуля

procedure Demo_Int_IO is

    Data_File : Text_IO.File_type;

begin
    Create (File => Data_File,
            Mode => Out_File,
            Name => "data.dat");

    for I in 1..10 loop               -- цикл вывода в файл
        Put (Data_File, I);           -- чисел и их квадратов
        Put (Data_File, "    ");
        Put (Data_File, I * I);
        New_Line (Data_File);
    end loop;

    Close (Data_File);
end Demo_Int_IO;
```

Здесь, процедура "Create" создает файл данных "*data.dat*", после чего, в этот файл производится запись некоторых данных процедурами "**Put**" и "New_Line". Процедура "Close" закрывает файл.

Остальные вышеперечисленные настраиваемые пакеты могут быть использованы аналогичным образом.

Следует добавить, что в дополнение к настраиваемым пакетам, Ада также предоставляет набор настроенных пакетов:

```
Ada.Short_Short_Integer_Text_IO  -- для значений типа Short_Short_Integer
Ada.Short_Integer_Text_IO        -- для значений типа Short_Integer
Ada.Integer_Text_IO              -- для значений типа Integer
Ada.Long_Integer_Text_IO         -- для значений типа Long_Integer
Ada.Long_Long_Integer_Text_IO    -- для значений типа Long_Long_Integer

Ada.Short_Float_Text_IO          -- для значений типа Short_Float
Ada.Float_Text_IO                -- для значений типа Float
Ada.Long_Float_Text_IO           -- для значений типа Long_Float
Ada.Long_Long_Float_Text_IO      -- для значений типа Long_Long_Float
```

Первые пять перечисленных пакетов, являются результатами конкретизации настраиваемого пакета *Ada.Text_IO.Integer_IO* для использования со значениями соответствующих целочисленных типов, последние четыре — результатами соответствующих конкретизаций настраиваемого пакета *Ada.Text_IO.Float_IO*.

12.2 Ввод/вывод двоичных данных

Как правило, современные производительные системы обработки больших объемов данных не используют текст для сохранения информации в своих файлах. Это значит, что вместо текста, в целях повышения производительности, файлы данных таких систем обычно используют двоичное представление. В этом случае файлы состоят из композитных объектов, которыми чаще всего являются записи.

В качестве основы для организации ввода/вывода двоичных данных Ада предусматривает настраиваемые пакеты *Ada.Sequential_IO* и *Ada.Direct_IO*. С помощью этих пакетов можно построить работу с двоичными файлами, состоящими из однотипных объектов одинаковой длины (записи, массивы, числа с плавающей точкой...).

12.2.1 Пакет *Ada.Sequential_IO*

Стандартный настраиваемый пакет *Ada.Sequential_IO* позволяет нам создавать файлы, состоящие из компонентов любого типа. При этом, должно соблюдаться единственное условие: тип компонентов должен быть ограничен (*constrained*).

Базовое содержимое настраиваемого пакета *Ada.Sequential_IO* идентично пакету *Ada.Text_IO*, за исключением того, что процедуры "**Get**" и "**Put**" соответственно заменены процедурами "**Read**" и "**Write**", и эти процедуры будут работать с типом данных для которого была произведена конкретизация настраиваемого пакета. Кроме этого, отсутствует понятие строки текста, и, следовательно, нет функции "End_Of_Line" и процедур "Skip_Line", "New_Line".

Примером использования этого пакета может служить следующее:

```
with Ada.Sequential_IO;          -- настраиваемый пакет

with Personnel_Details;          -- имеет тип записи "Personnel"
use  Personnel_Details;


with Produce_Retirement_Letter;

procedure Sequential_Demo is

    package Person_IO is new Ada.Sequential_IO (Personnel);

    Data_File : Person_IO.File_type;
    A_Person  : Personnel;

begin
    Person_IO.Open (Data_File, In_File, "person.dat");

    while not Person_IO.End_Of_File (Data_File) loop
        Person_IO.Read (Data_File, A_Person);

        if A_Person.Age > 100 then
            Produce_Retirement_Letter (A_Person);
        end if;
    end loop;

    Close (Data_File);
end Sequential_Demo;
```

Заметим, что в данном примере мы не акцентируем внимание на содержимом пакета *Personnel_Details*, а только указываем в комментарии, что он описывает тип записи "Personnel".

После открытия файла, он последовательно обрабатывается от позиции начала файла, и до тех пор, пока не будет достигнут конец файла, или будет выполнена одна из процедур "**Reset**" или "Close". Прямой доступ к элементам файла не возможен (отсюда и название: *Ada.Sequential* — последовательный).

12.2.2 Пакет *Ada.Direct_IO*

Пакет *Ada.Direct_IO* построен поверх пакета *Ada.Sequential_IO*. Он предусматривает возможность прямого обращения к необходимой записи в файле, определения размера файла и определения текущего индекса. Кроме этого, он дополнительно позволяет открывать файлы в режиме — "Inout_File" (чтение/запись). Такие средства, в совокупности с подходящим индексирующим пакетом, должны позволять построение пакета файловой обработки очень высокого уровня.

Следующий пример демонстрирует использование файлов с прямым доступом:

```
with Ada.Integer_Text_IO;     use Ada.Integer_Text_IO;

with Ada.Direct_IO;           -- настраиваемый пакет

with Personnel_Details;       -- имеет:
use  Personnel_Details;       --   тип записи "Personnel",
                              --   процедуру  "Display_Personnel",
                              --   и т.д. ...

with Display_Menu;            -- внешняя процедура отображения меню
```

```
procedure Direct_Demo is

    package Person_IO is new Direct_IO (Personnel);

    Data_File   : Person_IO.File_type;
    A_Person    : Personnel;
    Option      : Integer;
    Employee_No : Integer;

begin
    Person_IO.Open (Data_File, Inout_File, "Person.dat");

    loop
        Display_Menu;
        Get_Option (Option);

        case Option is
            when 1 =>
                    Get (Employee_No);
                    Set_Index (Positive_Count (Employee_No));
                    Read (Data_File, A_Person);
                    Display_Person (A_Person);
            when 2 =>
                    Get (Employee_No);
                    Set_Index (Positive_Count (Employee_No));
                    Read (Data_File, A_Person);
                    Get_New_Details (A_Person);
                    Write (Data_File, A_Person);
            when 3 =>
                    exit;
            when others =>
                    Put ("not a great option!");
        end case;
    end loop;
    Close (Data_File);
end Direct_Demo;
```

Здесь, для краткости подразумевается, что записи о служащих сохраняются в порядке номеров служащих — "Employee_No".

Также заметим, что мы не акцентируем внимание на содержимом внешних модулей: пакете *Personnel_Details* и процедуре "Display_Menu".

12.3 Потоки ввода/вывода

Стандарт Ada95 обогатил средства ввода/вывода Ады возможностью использования гетерогенных (состоящих из различных по составу, свойствам, происхождению частей) потоков ввода/вывода. Основная идея этого подхода заключается в том, что существует поток данных который ассоциируется с каким-либо файлом. За счет использования потоковых механизмов, обработка такого файла может быть выполнена последовательно, подобно*Ada.Sequential_IO*, или позиционно, подобно *Ada.Direct_IO*. Причем, в отличие традиционных средств файлового ввода вывода которые обеспечиваются пакетами *Ada.Sequential_IO* и *Ada.Direct_IO*, один и тот же поток позволяет выполнять чтение/запись для данных различного типа. Для обеспечения поддержки механизмов потокового ввода/вывода Ада предоставляет стандартный пакет *Ada.Streams.Stream_IO*.

Прежде чем приступить к детальному рассмотрению потоковых механизмов, необходимо заметить, что они достаточно тесно связаны с тэговыми типами, которые до настоящего момента не были рассмотрены. Поэтому, при первом ознакомлении с Адой, рассмотрение поддержки ввода/вывода на потоках можно пропустить, и вернуться к нему после ознакомления с тэговыми типами.

Пакет *Ada.Streams.Stream_IO* предоставляет средства которые позволяют создавать, открывать и закрывать файлы обычным образом. Далее, функция "Stream", которая в качестве параметра принимает значение

типа "File_Type" (потоковый файл), позволяет получить доступ к потоку ассоциируемому с этим файлом. Схематически, начало спецификации этого пакета имеет следующий вид:

```
package Ada.Streams.Stream_IO is
    type Stream_Access is access all Root_Stream_Type'Class;
    type File_Type is limited private;
    -- Create, Open, ...
    function Stream (File: in File_Type) return Stream_Access;
        . . .
end Ada.Streams.Stream_IO;
```

Заметим, что все объекты потокового типа являются производными от абстрактного типа "Ada.Streams.Root_Stream_Type" и обычно получают доступ к потоку через параметр который ссылается на объект типа "Ada.Streams.Root_Stream_Type'Class".

Последовательная обработка потоков выполняется с помощью атрибутов "'Read", "'Write", "'Input" и "'Output". Эти атрибуты предопределены для каждого нелимитированного типа. Следует заметить, что Ада, с помощью инструкции описания атрибута, предоставляет программисту возможность переопределения этих атрибутов. Таким образом, при необходимости, мы можем переопределять атрибуты которые установлены по умолчанию и описывать атрибуты для лимитированных типов.

Атрибуты "T'**Read**" и "T'**Write**" принимают параметры, которые указывают используемый поток и элемент типа "T" следующим образом:

```
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);
```

В качестве простого примера, рассмотрим случай, когда нам необходимо выполнить запись в поток значения типа "Date", описание которого имеет вид:

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

Сначала, мы создаем файл (используя обычный подход) и получаем доступ к ассоциированному с ним потоку. Затем, мы можем вызвать процедуру атрибута для значения которое необходимо записать в поток:

```
use Streams.Stream_IO;

Mixed_File : File_Type;
S          : Stream_Access;
  . . .
Create (Mixed_File);
S := Stream (Mixed_File);
  . . .
Date'Write (S, Some_Date);
Integer'Write (S, Some_Integer);
Month_Name'Write (S, This_Month);
  . . .
```

Примечательно, что пакет "Streams.Stream_IO" не является настраиваемым пакетом и, таким образом, не нуждается в конкретизации. Все подобные гетерогенные файлы имеют один и тот же тип. Записанный таким образом файл, может быть прочитан аналогичным образом. Однако, необходимо заметить, что если мы попытаемся прочитать записанные данные используя не подходящую для этих данных подпрограмму, то мы получим ошибку *Data_Error*.

В случае простой записи, такой как "Date", предопределенный атрибут "Date'**Write**" будет последовательно вызывать атрибуты "'Write" для каждого компонента "Date". Это выглядит следующим образом:

```
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;
```

Мы можем написать свою собственную версию для "Date'**Write**". Предположим, что нам необходимо осуществлять вывод имени месяца в виде соответствующего целого значения:

```
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;

for Date'Write use Date_Write;
```

тогда, следующая инструкция

```
Date'Write (S, Some_Date);
```

будет использовать новый формат для вывода значений типа "Date".

Аналогичные возможности предусматриваются для осуществления ввода. Это значит, что если нам необходимо прочитать значение типа "Date", то теперь нам нужно описать дополнительную версию "Date'**Read**" для выполнения чтения целых значений как значений месяца с последующей конверсией этих значений в значения типа "Month_Name".

Примечательно, что мы изменили формат вывода "Month_Name" только для случая "Date". Если нам нужно изменить формат вывода "Month_Name" для всех случаев, то разумнее переопределить "Month_Name'**Write**" чем "Date'**Write**". Тогда, это произведет к косвенному изменению формата вывода для типа "Date".

Следует обратить внимание на то, что предопределенные атрибуты "T'**Read**" и "T'**Write**", могут быть переопределены инструкцией определения атрибута только в том же самом пакете (в спецификации или декларативной части) где описан тип "T" (как любое описание представления). В результате, как следствие, эти предопределенные атрибуты не могут быть изменены для стандартно предопределенных типов. Однако они могут быть изменены в их производных типах.

Ситуация несколько усложняется для массивов и записей с дискриминантами, поскольку необходимо принимать во внимание дополнительную информацию предоставляемую значениями границ массива и значениями дискриминантов. (В случае дискриминанта значение которого равно значению по умолчанию, дискриминант рассматривается как обычный компонент записи). Это выполняется с помощью использования дополнительных атрибутов "'Input" и "'Output". Основная идея заключается в том, что атрибуты "'Input" и "'Output" обрабатывают дополнительную информацию (если она есть) и затем вызывают "'Read" и "'Write" для обработки остальных значений. Их описание имеет следующий вид:

```
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;
```

Примечательно, что "'Input" — это функция, поскольку "T" может быть неопределенным и нам могут быть не известны ограничения которые установлены для конкретного типа.

Таким образом, в случае массива процедура "'Output" выводит значения границ и, после этого, вызывает "'Write" непосредственно для самого значения.

В случае типа записи с дискриминантами, если запись имеет дискриминанты значения которых равны значениям по умолчанию, то "'Output" просто вызывает "'Write", которая трактует дискриминант как простой компонент записи. Если значение дискриминанта не соответствует тому значению, которое указано как значение по умолчанию, то сначала "'Output" выводит дискриминанты записи, а затем вызывает "'Write" для обработки остальных компонентов записи. В качестве примера, рассмотрим случай определенного подтипа, чей тип — это первый подтип, который не определен:

```
subtype String_6 is String (1 .. 6);
S: String_6 := "String";
   . . .
String_6'Output (S);      -- выводит значения границ и "String"
String_6'Write (S);       -- не выводит значения границ
```

Примечательно, что атрибуты "'Output" и "'Write" принадлежат типам и, таким образом, не имеет значения или мы записываем "String_6'**Write**", или "String'**Write**".

Приведенное выше описание работы "T'Input" и "T'Output" относится к атрибутам которые заданы по умолчанию. Они могут быть переопределены для выполнения чего-либо другого, причем не обязятельно для вызова "T'**Read**" и "T'**Write**". Дополнительно отметим, что "Input" и "Output" существуют также для определенного подтипа, и их значения просто вызывают "**Read**" и "**Write**".

Для взаимодействия с надклассовыми типами предназначены атрибуты "T'Class'Output" и "T'Class'Input". Для вывода значения надклассового типа, сначала производится вывод внешнего представления тэга, после чего с помощью механизма диспетчеризации (перенаправления) вызывается процедура "'Output" которая соответствующим образом выводит специфические значения (вызывая "'Write"). Подобным образом, для ввода значения такого типа, сначала производится чтение тэга, а затем, в соответствии со значением тэга, с помощью механизма диспетчеризации (перенаправления) вызывается функция "Input". Для полноты, также описаны атрибуты "T'Class'**Write**" и "T'Class'**Read**", которые выполняют диспетчеризацию (перенаправление) вызовов к подпрограммам определяемых атрибутами "'Write" и "'Read" специфического типа идентифицируемого тэгом.

Из рассмотренных нами примеров следует основной принцип работы с потоками: то что сначала было записано, то и должно быть прочитано, при выполнении подходящей обратной операции.

Теперь можно продолжить рассмотрение структуры которая лежит в основе всех этих механизмов. Все потоки являются производными от абстрактного типа "Streams.Root_Stream_Type", который имеет две абстрактных операции "**Read**" и "**Write**" описанные следующим образом:

```
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;
```

Организацию работы этих механизмов лучше рассматривать в терминах потоковых элементов, а не значений какого-либо типа. Следует обратить внимание на разницу между потоковыми элементами (*stream elements*) и элементами памяти (*storage elements*) (элементы памяти будут рассмотрены при рассмотрении пулов памяти). Элементы памяти (*storage elements*) затрагивают внутреннюю память (*storage*) в то время как потоковые элементы (*stream elements*) затрагивают внешнюю информацию и, таким образом, подходят для распределенных систем.

Предопределенные атрибуты "'Read" и "'Write" используют операции "**Read**" и "**Write**" ассоциированного с ними потока, и пользователь может описывать атрибуты одинаковым образом. Примечательно, что параметр "Stream" для корневого типа имеет тип "Root_Stream_Type", тогда как атрибут — это ссылочный тип обозначающий соответствующий класс. Таким образом, такой атрибут, определяемый пользователем, должен будет выполнять подходящее разыменование:

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

В заключение рассмотрения потоков Ады заметим что *Ada.Stream_IO* может быть использован для осуществления индексированного доступа. Это возможно, поскольку файл структурирован как последовательность потоковых элементов. Таким образом, индексированный доступ работает в терминах потоковых элементов подобно тому как работает *Direct_IO* в терминах элементов типа. Это значит, что индекс может быть прочитан и переустановлен. Процедуры "**Read**" и "**Write**" выполняют обработку относительно текущего значения индекса, также существует альтернативная процедура "**Read**", которая стартует согласно указанного значения индекса. Процедуры "**Read**" и "**Write**" (которые используют файловый параметр) точно соответствуют диспетчеризуемым (перенаправляемым) операциям ассоциированного потока.

12.4 Взаимодействие с командной строкой и окружением

12.4.1 Параметры командной строки

Во время выполнения программы существует возможность получения доступа к аргументам, которые указаны в командной строке запуска программы на выполнение. Такая возможность обеспечивается средствами стандартного пакета *Ada.Command_Line*, спецификация которого имеет следующий вид:

```
package Ada.Command_Line is
pragma Preelaborate (Command_Line);

    function Argument_Count return Natural;

    function Argument (Number : in Positive) return String;

    function Command_Name return String;

    type Exit_Status is Определяемый_Реализацией_Целочисленный_Тип;

    Success : constant Exit_Status;
    Failure : constant Exit_Status;

    procedure Set_Exit_Status (Code : in Exit_Status);

private
    -- Стандартом языка не определено
end Ada.Command_Line;
```

В качестве простой демонстрации использования средств этого предопределенного стандартом пакета, рассмотрим следующий пример:

```
with  Ada.Text_IO;
with  Ada.Command_Line;

procedure Show_CMDLine is
begin

    Ada.Text_IO.Put_Line
        ("The program " &
         '"' & Ada.Command_Line.Command_Name & '"' &
         " has " &
         Ada.Command_Line.Argument_Count'Img &
         " argument(s):");

    for I in  1..Ada.Command_Line.Argument_Count  loop

        Ada.Text_IO.Put_Line
            ("   The argument " & I'Img & " is " &
            '"' & Ada.Command_Line.Argument (I) & '"');

    end loop;

end Show_CMDLine;
```

Данная программа отображает фактическое имя запущенной программы, количество переданных в командной строке аргументов, а затем показывает строковые значения переданных аргументов. Процедура "Set_Exit_Status", пакета *Ada.Command_Line*, может быть использована для возврата во внешню среду кода статуса завершения работы программы (иначе — кода ошибки).

12.4.2 Переменные окружения программы

Для поддержки организации взаимодействия с переменными окружения операционной системы, реализация компилятора GNAT содержит дополнительный пакет *Ada.Command_Line.Environment* спецификация которого имеет следующий вид:

```
package Ada.Command_Line.Environment is

   function Environment_Count return Natural;

   function Environment_Value (Number : in Positive) return String;

end Ada.Command_Line.Environment;
```

Функция "Environment_Count" возвращает общее количество переменных окружения, а функция "Environment_Value" возвращает строку в которой содержится имя переменной окружения и ее значение, разделенные символом равенства.

Глава 13

Ссылочные типы (указатели)

До настоящего момента мы предполагали, что структуры данных и их размещение в пространстве памяти строго фиксировано и не изменно, то есть статично. Поэтому, такие данные, как правило, называют статическими. При этом, следует заметить, что доступ к статическим данным всегда осуществляется непосредственно к месту их физического размещения в памяти, то есть, используется механизм непосредственного доступа к данным.

Для повышения эффективности в современных системах сбора и обработки информации часто требуется использование данных структура которых может изменяться в процессе работы программы (списки, деревья, графы. . .). Такие данные, как правило, называют динамическими данными или данными с динамической структурой. Для того, чтобы обеспечить эффективную работу с динамическими данными механизм непосредственного доступа как правило не приемлем. Для этого необходим механизм при котором доступ к реальным данным осуществляется через адресные значения, определяющие расположение реальных данных в пространстве памяти, и предоставляется возможность манипулирования такими адресными значениями. Такой механизм доступа к данным называют механизмом косвенного доступа.

Многие современные языки программирования, в качестве механизма косвенного доступа к данным, используют указатели, которые позволяют манипулировать адресами размещаемых в пространстве памяти объектов. Не смотря на то, что указатели являются достаточно эффективным средством работы с динамическими данными, непосредственная работа с адресами и адресной арифметикой часто является источником ошибок которые трудно обнаружить. Известным обладателем подобной проблемы является семейство языков C/C++.

В качестве механизма косвенного доступа к данным, Ада предлагает концепцию ссылочных типов, использование которых во многом подобно традиционному использованию указателей. При этом ссылочные типы Ады обладают следующими характерными особенностями:

- Ссылочные типы не могут быть использованы для доступа к произвольному адресу памяти.
- Значения ссылочного типа не рассматриваются как целочисленные зачения, а значит, они не поддерживают адресную арифметику.
- Значения ссылочного типа могут ссылаться только на значения того типа, который был указан при описании ссылочного типа.

Из этого следует, что использование ссылочных типов Ады более безопасно и повышает общую надежность создаваемого программного обеспечения.

Примечательно, что в литературе не редко встречаются случаи когда официальная терминология нарушается и ссылочные типы Ады называют указателями.
Как правило, это делается с целью обеспечить более традиционное изложение материала.

Все описания ссылочных типов Ады можно условно разделить на три вида:

- ссылочные типы для динамической памяти
- обобщенные ссылочные типы
- ссылочные типы для подпрограмм

При этом, следует заметить, что последние два вида введены стандартом Ada-95.

13.1 Ссылочные типы для динамической памяти

Ссылочные типы для динамической памяти известны со времен стандарта Ada-83 и благополучно поддерживаются в стандарте Ada-95. Они могут быть использованы для ссылок на объекты размещаемые в области динамической памяти, которую также часто называют пулом динамической памяти или кучей.

13.1.1 Элементарные сведения: описание, создание, инициализация

Предположим, что у нас есть следующие описания:

```
type Person_Name is new String (1 .. 4);
type Person_Age is Integer range 1 .. 150;

type Person is
    record
        Name : Person_Name;
        Age  : Person_Age;
    end record;
```

Предположим также, что нам необходимо разместить экземпляр объекта типа "Person" в области динамической памяти. Для этого нам надо описать ссылочный тип, значения которого будут ссылаться на значения (объекты) типа "Person" и переменную этого ссылочного типа (заметим, что все ссылочные типы Ады описываются с помощью использования зарезервированного слова "**access**"):

```
type Person_Ptr is access Person; -- описание ссылочного типа
Someone : Person_Ptr;             -- переменная (объект) ссылочного типа
```

Для индикации того, что переменная ссылочного типа (указатель) не указывает ни на один объект, Ада использует специальное значение "**null**" (это подобно тому, что используется в языке Паскаль). По умолчанию, объект ссылочного типа всегда инициализируется в "**null**". Таким образом, после приведенных выше описаний, переменная "Someone" имеет значение "**null**".

Теперь, чтобы создать объект типа "Person" в области динамической памяти и присвоить соответствующее ссылочное значение переменной "Someone" необходимо выполнить следующее:

```
Someone := new Person;
```

Примечательно, что созданный таким образом в области динамической памяти объект типа "Person" — никак не инициализирован. Таким образом производится простое распределение пространства, которое необходимо для размещения объекта типа "Person", в области динамической памяти.

После создания объекта в области динамической памяти, мы можем проинициализировать индивидуальные поля записи типа "Person", на которую ссылается переменная "Someone", следующим образом:

```
Someone.Name := "Fred";
Someone.Age  := 33;
```

Следует заметить, что при таком обращении к объекту в динамической области памяти производится неявная расшифровка ссылки, а внешний вид кода подобен тому, что используется при обращении к статическому объекту.

Для того, чтобы обратиться ко всему содержимому объекта, на который указывает значение ссылочного объекта (указателя), Ада использует зарезервированное слово "**all**". Кроме того, в подобных случаях, используются квалифицированные (указывающие имя типа) агрегаты:

```
Someone.all := Person'("Fred", 33);       -- вариант с позиционным агрегатом

Someone.all := Person'(Name => "Fred";    -- вариант с именованным агрегатом
                       Age  => 33);
```

Оба варианта, показанные в примере, выполняют абсолютно одинаковые действия, и их результат ничем не отличается от показанного ранее примера инициализации путем обращения к индивидуальным полям записи типа "Person". Однако, стоит обратить внимание на то, что использование агрегатов для инициализации всего объекта сразу — более удобно, а вариант с именованным агрегатом имеет лучшую читабельность (по сравнению с позиционным агрегатом), а это важно при работе со сложными структурами данных.

Ада позволяет одновременно создавать объект в динамической области памяти и выполнять его инициализацию требуемыми значениями. Например, вместо выполненных ранее раздельно создания и инициализации, мы можем использовать следующее:

```
Someone := new Person'("Fred", 33);        -- вариант с позиционным агрегатом

Someone := new Person'(Name => "Fred";     -- вариант с именованным агрегатом
                       Age  => 33);
```

Чтобы облегчить понимание синтаксиса, который используется в языке Ада при работе со ссылочными типами, для тех кто знаком с такими языками программирования как Си и Паскаль, предлагается следующая сравнительная таблица:

	Паскаль	C	Ада
Доступ к полям указываемого объекта	a^.fieldname	*a.fieldname a->fieldname	a.fieldname
Копирование указателя	b := a;	b = a;	b := a;
Копирование указываемого обекта	b^ := a^;	*b = *a;	b.**all** := a.**all**;

13.1.2 Структуры данных со ссылками на себя

В предыдущих примерах мы рассмотрели то как размещать обычные типы данных в динамической области памяти и выполнять их инициализацию. Однако, для того чтобы строить сложные динамические структуры данных (списки, деревья, графы...) необходима возможность описания структур данных которые могут ссылаться сами на себя, то есть требуется создание ссылки на структуру данных, которая еще не существует (не описана).

Ада позволяет решить подобную проблему осуществляя неполное описание типа. При неполном описании просто указывается имя типа, который еще не описан. После этого компилятор ожидает, что полное описание типа будет дано до завершения файла с исходным текстом:

```
type Element;                      -- неполное описание типа

type Element_Ptr is access Element;

type Element is                    -- полное описание типа
    record
        Value : Integer;
        Next  : Element_Ptr;
    end record;

Head_Element : Element_Ptr;        -- переменная которая будет началом списка
```

Теперь мы можем построить простой связанный список, началом которого будет переменная "Head_Element", следующим образом:

```
Head_Element := new Element'(
                      Value => 1,
                      Next  => (new Element'(
                                    Value => 2,
                                    Next  => (new Element'(
                                                  Value => 3,
                                                  Next  => null)))));
```

Данный список содержит всего три элемента (узла). Следует обратить внимание на то, что в последнем элементе списка ссылка "Next" имеет значение "**null**".

13.1.3 Освобождение пространства динамической памяти

После того как мы рассмотрели как распределять пространство динамической памяти и выполнять инициализацию объектов, осталось рассмотреть как осуществляется освобождение более не используемого пространства динамической памяти.

Существует два способа освобождения пространства, которое было распределено в области динамической памяти, для его последующего повторного использования:

- библиотека времени выполнения выполняет неявное освобождение распределеного пространства когда использованый для распределения пространства динамической памяти тип выходит из области видимости (для этого случая примечательно то, что если тип описан на уровне библиотеки, то освобождение памяти не произойдет вплоть до завершения работы программы).
- выполнение явного освобождения пространства динамической памяти в программе

Следует заметить, что стандарт языка Ада не определяет более четких требований и правил для алгоритмов библиотеки времени выполнения. Поэтому, реальные алгоритмы будут определяться реализацией конкретного компилятора и его библиотек поддержки. Следовательно, для уточнения этих сведений необходимо обратиться к документации на используемый компилятор, и здесь этот способ рассматриваться не будет.

Если вам необходимо освободить память (подобно тому как это делает системный вызов "free" в UNIX), то вы можете конкретизировать настраиваемую процедуру "Ada.Unchecked_Deallocation". Эта процедура называется непроверяемой (*unchecked*) поскольку компилятор не осуществляет проверку отсутствия ссылок на освобождаемый объект. Таким образом, выполнение этой процедуры может привести к появлению "висячих" ссылок.

```
generic
    type Object (<>) is limited private;
    type Name is access Object;

procedure Ada.Unchecked_Deallocation (X : in out Name);
pragma Convention (Intrinsic, Ada.Unchecked_Deallocatoin);
```

Для показанного ранее ссылочного типа "Element_Ptr", значения которого ссылаются на объекты типа "Element", это может быть конкретизировано следующим образом:

```
procedure Free is new Ada.Unchecked_Deallocation (Object => Element,
                                                  Name   => Element_Ptr);
```

Теперь, можно написать рекурсивную процедуру "Free_List", которая будет удалять список, состоящий из объектов типа "Element". Начало списка будет указываться параметром процедуры "Free_List", а для непосредственного удаления объекта типа "Element" процедура "Free_List" будет использовать процедуру "Free":

```
with  Free;

procedure Free_List (List_Head: in out Element) is
begin
    if  List_Head.Next /= null
        Free_List (List_Head.Next);  -- рекурсивный вызов
    end if;

    Free (List_Head);
end Free_List;
```

В результате, для удаления показанного ранее списка объектов типа "Element", начало которого указывается переменной "Head_Element", можно выполнить следующее:

```
    . . .
Free_List (Head_Element);
    . . .
```

Следует напомнить, что в рассмотренном нами списке для последнего элемента (узла) значение ссылки "Next" было равно "**null**".

При описании ссылочного типа "Element_Ptr", может быть использована директива компилятора "Controlled":

```
    . . .
type Element_Ptr is access Element;
pragma Controlled (Element_Ptr);
    . . .
```

В этом случае, компилятор будет проинформирован о том, что задачу освобождения распределенного в пуле динамической памяти пространства, для объектов на которые ссылаются значения ссылочного типа "Element_Ptr", взял на себя программист.

13.1.4 Пулы динамической памяти

Обычно, все объекты ссылочного типа определенные пользователем, используют один, общий для всей программы, пул динамической памяти. Однако, согласно стандарта Ada95, допускается определение пользовательского пула динамической памяти из которого будет производиться распределение памяти для динамически создаваемых объектов. Такой пул могут использовать различные объекты ссылочного типа.

Пользователь может описать свой собственный тип пула динамической памяти, используя абстрактный тип "Root_Storage_Pool" описанный в пакете *System.Storage_Pools*, и после этого ассоциировать его с объектом ссылочного типа используя атрибут "'Storage_Pool". Например:

```
Pool_Object : Some_Storage_Pool_Type;

type T is access какой-нибудь_тип;
for T'Storage_Pool use Pool_Object;
```

Применение пулов динамической памяти определяемых пользователем обусловлено, как правило, достаточно специфическими требованиями решаемой задачи.

Например, достаточно распространенными являются случаи, когда, при использовании единого пула, интенсивное распределение/удаление динамических объектов с различными размерами приводит к чрезмерной фрагментации единого пула динамической памяти. В результате этого, может возникнуть ситуация когда невозможно распределить непрерывный блок динамической памяти требуемого размера, для размещения какого-либо динамического объекта, при фактическом наличии достаточного общего объема свободной динамической памяти. Чтобы предотвратить возникновение подобной ситуации, можно использовать различные пулы динамической памяти для объектов с различными размерами, исключая, таким образом, возможность фрагментации пулов динамической памяти.

Другими примерами случаев использования пулов динамической памяти определяемых пользователем могут служить: необходимость распределения динамической памяти для объектов размер которых значительно меньше чем минимальный размер памяти, который распределяется для размещения объекта в общем пуле динамической памяти программы (иначе — в пуле по умолчанию); требование приложения реального времени обеспечить фиксированное время выполнения операций распределения/освобождения пространства динамической памяти.

13.1.5 Проблемы обусловленные применением ссылочных типов

Несмотря на то, что в сравнении с традиционными указателями ссылочные типы Ады обладают свойствами которые помогают обеспечить надежность разрабатываемого программного обеспечения, нельзя не упомянуть о проблемах которые возникают при использовании как указателей, так и ссылочных типов.

Рассмотрим следующий пример:

```
declare

    type Person_Name is new String (1 .. 4);
    type Person_Age is Integer range 1 .. 150;

    type Person is
        record
            Name : Person_Name;
            Age  : Person_Age;
        end record;

    X : Person_Ptr := new Person'("Fred", 27);
    Y : Person_Ptr := new Person'("Anna", 20);

begin

    X := Y;                          -- *****
    Y.all := Person'("Sue ", 34);
    Put (X.Name);

end;
```

В этом примере, основной интерес для нас представляет строка помеченная звездочками. В ней ссылочному объекту "X" присваивается значение ссылочного объекта "Y", а не значение объекта, на который ссылается "Y". После этого оба ссылочных объекта, — и "X", и "Y", — ссылаются на один и тот же объект, размещенный в области динамической памяти.

Первым интересным моментом является то, что теперь изменение объекта на который ссылается переменная "Y" будет неявно изменять объект на который ссылается переменная "X" (такое неявное изменение часто называют "побочным эффектом"). Поэтому в результате выполнения кода этого примера будет выводиться строка "Sue ". Следует заметить, что при разработке реальных программ, работающих со ссылочными типами, необходимо уделять должное внимание эффектам подобного вида, поскольку они могут быть первопричиной странного поведения программы, а в некоторых случаях могут вызвать аварийное завершение работы программы.

Вторым интересным моментом является то, что после выполнения присваивания ссылочному объекту "X" значения ссылочного объекта "Y", теряется ссылка на объект, на который до присваивания ссылался ссылочный объект "X". При этом, сам объект продожает благополучно располагаться в области динамической памяти (такой эффект называют "утечкой памяти"). Следует заметить, что при интенсивном использовании ссылочных типов, утечка памяти может привести к тому, что все доступное пространство области динамической памяти будет исчерпано. После этого, любая попытка разместить какой-либо объект в области динамической памяти приведет к ошибке *Storage_Error*.

13.2 Обобщенные ссылочные типы

Рассмотренные нами ссылочные типы могут быть использованы только как средство косвенного доступа к объектам которые размещаются в области динамической памяти. Однако, бывают случаи когда необходимо использование механизма косвенного доступа для статического объекта, то есть, необходимо иметь возможность обратиться к содержимому статического объекта используя значение ссылочного типа.

Ада (согласно стандарта Ada95) предоставляет возможность описывать статические объекты (переменные или константы) так, чтобы они являлись косвенно доступными, а также предоставляет возможность описывать ссылочные типы, так, чтобы они могли ссылаться не только на объекты размещенные в пуле динамической памяти, но и на статические объекты (которые были описаны как косвенно доступные).

В этом случае, статические объекты (переменные или константы) описываются с использованием зарезервированного слова "**aliased**", которое указывает, что данный статический объект является косвенно доступным. Например:

```
Int_Var   : aliased Integer;
Int_Const : aliased constant Integer := 0;
```

Чтобы получить ссылочные значения для статических объектов которые допускают использование косвенного доступа необходимо использовать атрибут "'Access".

Чтобы предоставить возможность ссылочному типу ссылаться на косвенно доступные статические объекты, ссылочный тип описывается с использованием зарезервированного слова "**all**" или зарезервированного слова "**constant**". Заметим, что такие ссылочные типы называют обобщенными ссылочными типами (*general access types*), и они также могут быть использованы для доступа к объектам которые размещены в пуле динамической памяти. Например:

```
type Int_Var_Ptr    is  access all Integer;
type Int_Const_Ptr  is  access constant Integer;
```

При этом, если при описании ссылочного типа использовано зарезервированное слово "**all**", то такой ссылочный тип предоставляет доступ который позволяет осуществлять как чтение, так и запись содержимого объекта на который ссылается значение этого ссылочного типа. Следовательно, такое описание используется для получения ссылок на статические переменные.

Если при описании ссылочного типа использовано зарезервированное слово "**constant**", то такой ссылочный тип предоставляет доступ который позволяет осуществлять только чтение содержимого объекта на который ссылается значение этого ссылочного типа. Следовательно, такое описание чаще всего используется для получения ссылок на константы и в более редких случаях для получения ссылок на переменные.

Рассмотрим следующий пример:

```
procedure General_Access_Demo is

    type Int_Var_Ptr_Type is access all Integer;
    A : aliased Integer;
    X, Y : Int_Var_Ptr_Type;

begin
    X := A'Access;
    Y := new Integer;
end General_Access_Demo;
```

Здесь, переменная "A" имеет тип "**Integer**" и является косвенно доступной. Переменные "X" и "Y" имеют тип "Int_Var_Ptr_Type" который является обобщенным ссылочным типом. В результате, внутри процедуры, переменная "X" ссылается на статическую переменную "A", ссылка на которую получена с помощью атрибута "'Access". Переменная "Y" ссылается на объект типа "**Integer**" который размещен в области динамической памяти.

Ада позволяет использовать обобщенные ссылочные типы для формирования ссылок на отдельные элементы внутри составных типов данных (таких как массив и/или запись), например:

```
type  Array_Type  is array (Positive range <>) of aliased Integer;

type  Record_Type is
    record
        A_Int_Var : aliased Integer;
        Int_Var   : Integer;
    end record;

type Int_Var_Ptr_Type is access all Integer;
```

В данном случае тип "Array_Type" — это массив, состоящий из косвенно доступных элементов типа "**Integer**", а тип "Record_Type" — это запись, у которой поля "A_Int_Var" и "Int_Var" имеют тип "**Integer**", причем поле "A_Int_Var" косвенно доступно, а поле "Int_Var" — нет. Таким образом, значения типа "Int_Var_Ptr_Type" могут ссылаться на индивидуальные элементы массивов, принадлежащих к типу "Array_Type", и поле "A_Int_Var" записей, принадлежащих к типу "Record_Type".

Интересным примером использования обобщенных ссылочных типов может служить следующее схематическое описание пакета:

```
package Message_Services is
    type Message_Code_Type is range 0..100;

    subtype Message is String;

    function Get_Message (Message_Code: Message_Code_Type) return Message;

    pragma Inline (Get_Message);
end Message_Services;



package body Message_Services is
    type Message_Handle is access constant Message;

    Message_0: aliased constant Message := "OK";
    Message_1: aliased constant Message := "Up";
    Message_2: aliased constant Message := "Shutdown";
    Message_3: aliased constant Message := "Shutup";
        . . .

    Message_Table: array (Message_Code_Type) of
        Message_Handle :=
            (0 => Message_0'Access,
             1 => Message_1'Access,
             2 => Message_2'Access,
             3 => Message_3'Access,
```

```
            . . .
        );

    function Get_Message (Message_Code: Message_Code_Type)
        return Message is
    begin
        return Message_Table (Message_Code).all;
    end Get_Message;
end Message_Services;
```

В данном случае, достаточно элегантным приемом является использование массива "Message_Table", который представляет таблицу ссылок на строковые константы переменной длины. Следует также заметить, что при этом не производится никакого распределения пространства в области динамической памяти.

13.2.1 Правила области видимости для обобщенных ссылочных типов

В целях обеспечения надежности, на использование обобщенных ссылочных типов накладываются некоторые дополнительные ограничения. Так, область видимости косвенно доступного объекта, на который будет ссылаться значение переменной обобщенного ссылочного типа "T", не должна быть глубже чем область видимости переменной обобщенного ссылочного типа "T". Это подразумевает, что следующий пример — не корректен, и, следовательно, будет отвергнут компилятором:

```
procedure Illegal is                -- внешняя область видимости описаний

    type Integer_Access is access all Integer;
    Integer_Ptr : Integer_Access;

begin
        . . .
    declare                         -- внутренняя область видимости описаний
        Integer_Variable : aliased Integer;
    begin
        Integer_Ptr := Integer_Variable'Access;         -- это не корректно!!!
    end;                            -- завершение области видимости
                                    -- переменной Integer_Variable

    Integer_Ptr.all := Integer_Ptr.all + 1;             -- сюрприз!
                                    -- переменная Integer_Variable
                                    --   больше не существует!

end Illegal;                        -- завершение области видимости
                                    --   для Integer_Access
```

Смысл примера заключается в следующем. Во внутреннем блоке, переменной "IA" ссылочного типа "Integer_Access" присваивается значение, которое ссылается на переменную "IVar". При завершении внутреннего блока, переменная "IVar" прекращает свое существование. Следовательно, в следующей инструкции присваивания, переменная "IA" ссылается на не существующую переменную. Такая ситуация известна как "проблема висячих указателей".

Такое ограничение выглядит достаточно строго, но оно гарантирует, что любой объект на который могут ссылаться значения типа "Integer_Access" будет существовать на протяжении всего времени существования переменных типа "Integer_Access". В частности, если тип "Integer_Access" описан на уровне библиотеки, то область видимости "Integer_Access" определяется всей программой, а это значит, что с типом "Integer_Access" могут быть использованы только те переменные, которые описаны на уровне библиотеки.

Бывают случаи когда необходимо нарушить строгость данного ограничения. Тогда, для получения ссылочного значения, вместо атрибута "'Access" можно использовать атрибут "'Unchecked_Access", который позволяет получить ссылочное значение без осуществления проверки правил доступа:

```
procedure Legal_But_Stupid is

    type Integer_Access is access all Integer;
    IA : Integer_Access;
```

```
begin
        . . .
    declare
        IVar : aliased Integer;
    begin
        IA := IVar'Unchecked_Access;   -- это не надежно!!!
    end;
    IA.all := IA.all + 1;              -- теперь это будет только ВАША ошибка!!!
end Legal_But_Stupid;
```

Следует заметить, что применение атрибута "'Unchecked_Access" - не рекомендуется. Это подразумевает, что при его использовании вы должны быть полностью уверены в своих действиях.

Еще одним способом, который позволяет обходить эти ограничения, является описание обобщенных ссылочных типов внутри настраиваемых модулей. Идея такого подхода основана на том, что область видимости для типа который описан внутри настраиваемого модуля будет ограничена областью видимости места конкретизации этого настраиваемого модуля.

13.3 Ссылочные типы для подпрограмм

Также как и обобщенные ссылочные типы, ссылочные типы для подпрограмм не доступны в стандарте Ada83. Они являются нововведением, которое определил выход стандарта Ada95.

Ссылочные типы для подпрограмм позволяют использовать подпрограммы как параметры передаваемые другим подпрограммам (например, так как это делается в математических библиотеках Фортрана), а также осуществлять позднее связывание.

При описании ссылочных типов для подпрограмм следует придерживаться следующих соглашений:

- роописание типа должно определять такой список параметров, который соответствует списку параметров подпрограммы на которую будут ссылаться значения этого типа
- при описании ссылочного типа для функции, тип возвращаемого значения должен соответствовать типу возвращаемого значения функции на которую будут ссылаться значения этого типа
- переменные ссылочного типа для подпрограмм могут быть использованы для доступа к любой подпрограмме профиль которой соответствует профилю определеному при описании этого ссылочного типа

Простейшим примером использования функционального ссылочного типа как параметра подпрограммы может служить следующий фрагмент кода:

```
type  Access_Function is access  function (Item: in      Float) return Float;

type  Vector  is array (Integer range <>) of Float;

procedure For_Each (F : in      Access_Function;
                    To: in out Vector) is
begin
    for I in To'Range  loop
        To (I) := F (To (I));
    end loop;
end For_Each;
```

Здесь, процедура "For_Each" принимает в качестве параметра "F" значение ссылочного типа "Access_Function" указывающее на функцию, которую необходимо вызывать при обработке каждого элемента массива типа "Vector", передаваемого ей как параметр "**To**".

Примечательно, что при вызове функции "F" расшифровка ссылки производится автоматически. Следует также заметить, что вместо "F(**To**(I))" можно было бы написать "F.**all**(**To**(I))", что в подобных случаях — не обязательно. Использование ".**all**" требуется когда вызываемая подпрограмма (процедура или функция) не имеет параметров.

Описание переменной массива чисел, передаваемой как параметр "**To**" для процедуры "For_Each", и описание функции, ссылка на которую могла бы быть использовна как параметр "F" процедуры "For_Each", могут иметь следующий вид:

```
Data_Vector : Vector (1..3) := (1.0, 2.0, 3.0);

function Square (Val: in      Float) return Float is
begin
    return Val * Val;
end Square;
```

Таким образом, вызов процедуры "For_Each" (с учетом приведенных ранее описаний) будет иметь вид:

```
For_Each (F  => Square'Access,
          To => Data_Vector);
```

Примечательно, что для получения ссылочного значения которое указывает на функцию "Square" используется атрибут "'Access".

Ссылочные типы для подпрограмм могут быть использованы для построения таблиц вызова подпрограмм. Рассмотрим следующий схематический пример:

```
type Action_Operation is access procedure;

procedure Add;
procedure List;
procedure Delete;

Action : constant array (1..3) of Action_Operation := (
                              Add'Access,
                              List'Access,
                              Delete'Access
);



type Math_Function is access function (I : in      Float) return Float;

function Sin (F : in      Float) return Float;
function Cos (F : in      Float) return Float;
function Tan (F : in      Float) return Float;

Math_Operation : constant array (1..3) of Math_Function := (
                              Sin'Access,
                              Cos'Access,
                              Tan'Access
);
```

Здесь формируются две таблицы вызовов подпрограмм: первая таблица вызовов представлена массивом "Action", который содержит значения ссылочного типа "Action_Operation", а вторая таблица вызовов представлена массивом "Math_Operation", который содержит значения ссылочного типа "Math_Function" (заметим, что таблицами вызовов, как правило, являются массивы).

Примером вызова "I"-той подпрограммы в таблице (где "I" — это индекс в таблице) может служить следующее:

```
F:  Float;
   . . .

Action (I).all;                -- вызов I-той процедуры из таблицы
                               --   Action
F := Math_Operation (I) (F);   -- вызов I-той функции из таблицы
                               --   Math_Operation с параметром F
```

Напомним, что для ссылочных значений, которые указывают на подпрограммы без параметров, при вызове подпрограммы требуется использование ".**all**".

13.3.1 Правила области видимости ссылочных типов для подпрограмм

Ссылочные типы для подпрограмм используют те же самые правила области видимости, что и обобщенные ссылочные типы. Таким образом, ссылочные типы для подпрограмм, которые описаны на уровне библиотеки, могут быть использованы только для ссылки на библиотечные подпрограммы. Дополнительным ограничением является то, что со ссылочными типами для подпрограмм нельзя использовать атрибут "'Unchecked_Access".

Единственный способ, который позволяет обходить такие строгие ограничения, является описание ссылочных типов для подпрограмм внутри настраиваемых модулей. Идея такого подхода заключается в том, что область видимости для типа который описан внутри настраиваемого модуля будет ограничена областью видимости места конкретизации этого настраиваемого модуля.

13.4 Низкоуровневая средства работы со ссылочными типами и физическими адресами памяти

Для организации работы с динамическими данными и динамической памятью в большинстве приложений достаточно тех механизмов, которые предоставляют ссылочные типы Ады. Однако, при решении специфических системных задач, может возникнуть необходимость непосредственного взаимодействия с физическими адресами памяти используемой аппаратной платформы.

Для таких случаев Ада предусматривает стандартный набор низкоуровневых средств. Так, в стандартном пакете *System* представлены:

- приватный тип "Address", внутреннее представление которого соответствует внутреннему представлению физического адреса памяти используемой системы.
- константа "Storage_Unit" — предоставляет битовый размер минимального адресуемого элемента памяти системы
- константа "Memory_Size" — максимально возможный размер памяти системы

При решении таких задач, важно учитывать тот факт, что реализация внутреннего представления значений ссылочных типов не обязана соответствовать представлению физических адресов системы. Стандарт Ады этого не требует. Для выполнения преобразований значений ссылочных типов в значение физического адреса и наоборот, следует использовать настраиваемый пакет *System.Address_To_Access_Conversions*, спецификация которого имеет следующий вид:

```
generic
   type Object (<>) is limited private;

package System.Address_To_Access_Conversions is

   type Object_Pointer is access all Object;
   for Object_Pointer'Size use Standard'Address_Size;

   function To_Pointer (Value : Address)        return Object_Pointer;
   function To_Address (Value : Object_Pointer) return Address;

end System.Address_To_Access_Conversions;
```

И в заключение, в стандартном пакете *System.Storage_Elements* предоставлены операции адресной арифметики и некоторые дополнительные описания (за более подробными сведениями следует обратиться к файлу спецификации этого пакета).

Глава 14

Тэговые типы (*tagged types*)

Тэговые типы являются нововведением стандарта Ada95. Они дополняют традиционную систему типов языка Ада, и позволяют обеспечить полную поддержку объектно-ориентированного программирования. Концептуально новыми особенностями тэговых типов являются возможность расширения структуры данных типа при наследовании, способствующая программированию посредством расширения, и динамическая диспетчеризация вызовов примитивных операций, являющаяся основой полиморфизма.

Чтобы в последствии не породить терминологической путаницы, необходимо сразу сделать одно важное замечание которое специально предназначено для знатоков ООП, активно использующих другие языки программирования (например, *C++* или какой-либо современный диалект Паскаля, поддерживающий объектно-ориентированное расширение). В традиционном понимании, слово "класс" трактуется как спецификация типа данных и множество методов (операций) этого типа данных. В отличие от этого, Ада трактует понятие "класс"как набор типов которые объединены иерархией наследования.

14.1 Механизмы наследования

Согласно концепции производных типов Ады, которая известна со времен стандарта Ada-83, производный тип наследует структуру данных и операции типа-предка. Позже, для работы с производным типом, можно изменить унаследованные от типа-предка операции и/или добавить к ним новые операции. Хотя такая модель наследования является достаточно мощным инструментом, она не позволяет нам дополнить унаследованную структуру данных новыми элементами, то есть, мы не можем выполнять расширение типа при наследовании.

14.1.1 Расширение существующего типа данных

Как было сказано ранее, тэговые типы, в отличие от обычных не тэговых типов, позволяют осуществлять расширение структуры данных типа предка при наследовании. Рассмотрим следующий пример описания:

```
type Object_1 is tagged
    record
        Field_1 : Integer;
    end record;
```

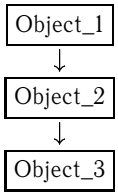
В данном случае тип "Object_1" содержит всего одно поле "Field_1" типа "**Integer**", и описан как тэговый тип. Не трудно заметить, что внешний вид такого описания подобен описанию обычной записи, и отличается только наличием зарезервированного слова "**tagged**", которое, собственно, и указывает на то, что описываемый тип является тэговым типом.

Теперь, в качестве примера, мы можем описать производные от "Object_1" типы, расширяя их новыми компонентами следующим образом:

```
type Object_2 is new  Object_1  with
    record
        Field_2 : Integer;
    end record;
```

```
type Object_3 is new Object_2 with
    record
        Field_3 : Integer;
    end record;
```

В данном примере, тип "Object_2" является производным типом от типа "Object_1", а тип "Object_3" — производным типом от типа "Object_2". Таким образом, в результате показанных описаний, получилась следующая иерархия типов:



Следует заметить, что тип, производный от тэгового типа, также будет являться тэговым типом. Поэтому все описанные выше типы: "Object_1", "Object_2" и "Object_3", - являются тэговыми типами, и, следовательно, обладают способностью расширения.

В результате показанных выше описаний, типы "Object_1", "Object_2" и "Object_3" будут содержать следующие поля:

Object_1	**Object_2**	**Object_3**
Field_1	Field_1	Field_1
	Field_2	Field_2
		Field_3

Примечательным фактом является то, что описание типа "Object_2" не указывает явно наличие поля "Field_1". Это поле наследуется от типа "Object_1". Также, описание типа "Object_3" не указывает явно наличие полей "Field_1" и "Field_2". Эти поля наследуются от типа "Object_2".

14.1.2 Описание переменных и преобразование типов

В отличие от других объектно-ориентированных языков программирования, Ада не использует каких либо специальных конструкторов, поэтому объекты (иначе переменные) тэговых типов могут быть описаны обычным образом. Инициализация индивидуальных полей может быть выполнена отдельно или с помощью агрегатов. Например:

```
declare
    Instance_1  : Object_1;
    Instance_2  : Object_2;
    Instance_3  : Object_3;
begin
    Instance_1.Field_1 := 1;
    Instance_2 := (1, 2 );
    Instance_3 := (Field_1 => 1,
                   Field_2 => 2,
                   Field_3 => 3);
    . . .
end;
```

Ада позволяет выполнять явное преобразование типов при преобразовании типа потомка к типу предка (снизу-вверх). Так, используя приведенные в предыдущем примере описания, можно выполнить следующие преобразования:

```
    . . .
    Instance_1 := Object_1 (Instance_3);
    Instance_2 := Object_2 (Instance_3);
    . . .
```

При таком преобразовании типов значение дополнительных полей потомка просто отбрасываются.

Кроме того, допускается выполнение преобразования типов от предка к потомку (сверху-вниз). Этот случай несколько сложнее, поскольку потомок может содержать дополнительные поля, которые отсутствуют у типа предка. В таких случаях используются расширяющие агрегаты, например:

```
. . .
Instance_2 := (Instance_1 with 2);
Instance_3 := (Instance_1 with Field_2 => 2, Field_3 => 3);
. . .
```

Следует заметить, что клиентские программы не часто используют эти средства, поскольку тэговые типы, как правило, описываются как приватные типы.

14.1.3 Примитивные и не примитивные операции над тэговыми типами. Наследование операций

Концепция примитивной операции типа имеет важное значение в Аде.
В общем случае, примитивная операция над типом "T" - это или предопределенный знак операции (например, знаки операций для выполнения действий над целыми числами типа "**Integer**"), или операция (подпрограмма или знак операции) которая описывается в том же самом пакете в котором описывается тип "T" вслед за описанием типа "T", и принимает параметр типа "T" или возвращает значение типа "T" (в случае функции). При этом, необходимо обратить особое внимание на тот факт, что все примитивные операции являются наследуемыми.

Концепция примитивной операции приобретает дополнительную важность в случае с тэговыми типами, поскольку обращения к примитивным операциям над тэговыми типами являются потенциально диспетчеризуемыми вызовами, а это, в свою очередь, является одним из базовых механизмов обеспечения динамической полиморфности объектов тэговых типов.

Следует также заметить, что если тэговый тип должен иметь какие-либо примитивные операции, то он должен быть описан в спецификации пакета. Описание тэгового типа в подпрограмме, с последующим описанием операций над этим тэговым типом не подразумевает того, что такие операции будут примитивными. Также, если в пакете описывается другой тэговый тип, производный от первого описанного в этом пакете тэгового типа, то после описания второго тэгового типа становится невозможным описание примитивных операций для первого тэгового типа (обратное будет подразумевать, что второй теговый тип способен наследовать примитивные операции которые еще не описаны).

Для уточнения сказанного рассмотрим следующий пример:

```
package Simple_Objects  is

    type Object_1 is tagged
        record
            Field_1 : Integer;
        end record;

    -- примитивные операции типа Object_1
    procedure Method_1 (Self: in out Object_1);


    type Object_2 is new  Object_1  with
        record
            Field_2 : Integer;
        end record;

    -- примитивные операции типа Object_2
    procedure Method_2 (Self: in out Object_2);
    procedure Method_2 (Self: in out Object_1); -- НЕДОПУСТИМО!!!
                -- должна быть примитивной операцией для Object_1,
                -- но недопустима поскольку следует за описанием типа Object_2
                -- который является производным от типа Object_1

end Simple_Objects;
```

В подобных случаях говорят, что описание типа "Object_1" становится "замороженным" при обнаружении описания типа "Object_2", производного от типа "Object_1". Подобное "замораживание" осуществляется также в случаях когда обнаруживается описание какого-либо объекта (переменной) типа "Object_1". Как только описание типа "заморожено", описание примитивных операций этого типа становится невозможным.

Следует также обратить внимание на то, как выполняется наследование примитивных операций, - подобных подпрограмме "Method_1" для типа "Object_1", в показанном выше примере, — для производного типа. Подразумевается, что такие операции идентичным образом описываются неявно сразу за описанием производного типа. Причем, тип парамета, где тип параметра соответствует типу предка, заменяется на производный тип. Таким образом, для приведенного выше примера, в случае типа "Object_2", выполняется неявное описание операции "Method_1", наследуемой от типа-предка "Object_1". Такое неявное описание будет иметь следующий вид:

```
. . .
procedure Method_1 (Self: in out Object_2);
. . .
```

Бывают случаи, когда необходимо описать подпрограмму которая не будет наследоваться потомками тэгового типа, то есть, описать подпрограмму так, чтобы она не являлась примитивной операцией типа. Типичными примерами таких подпрограмм могут служить функция, которая создает экземпляр объекта тэгового типа и возвращает его в качестве результата, или процедура инициализации полей объекта тэгового типа (назначение таких подпрограмм подобно конструкторам в других языках программирования). Опасность наследования подобных подпрограмм заключается в том, что подпрограмма предка не имеет никаких сведений о дополнительных полях которые описаны в объекте-потомке как расширение типа-предка. Следовательно, результат работы таких подпрограмм, в случае производного типа, будет не корректным (точнее — не полным).

Решением подобной проблемы, при описании пакета содержащего тэговый тип, может служить размещение описаний таких подпрограмм во внутреннем пакете. В качестве примера, рассмотрим следующее описание:

```
package Simple_Objects  is

    type Object_1 is tagged
        record
            Field_1 : Integer;
        end record;

    -- примитивные операции типа Object_1
    procedure Method_1 (Self: in out Object_1);
    . . .


    package Constructors  is     -- внутренний пакет содержащий не наследуемые
                                 --   операции типа Object_1

        function  Create (Field_1_Value: in Integer) return Object_1;
        . . .

    end Constructors;
    . . .

end Simple_Objects;
```

Здесь, функция "Create", которая возвращает значение типа "Object_1", расположена во внутреннем пакете *Constructors*. В результате такого описания, функция "Create" (а также другие подпрограммы для типа "Object_1", расположенные во внутреннем пакете *Constructors*) не будет наследоваться потомками типа "Object_1" (типами, производными от типа "Object_1").

В заключение обсуждения наследования операций над тэговыми типами, следует сделать одно важное замечание. Поскольку производные типы явно не отображают операции которые они наследуют от своих типов-предков, то могут возникнуть трудности в понимании того, какие операции реально выполняются над объектами конкретного производного типа. Действительно, для получения таких сведений необходимо изучить исходные тексты с описаниями всех типов-предков требуемого производного типа. Осуществление этого может быть достаточно трудоемким процессом для сложной иерархии типов. Для справедливости, следует заметить, что такая проблема характерна не только для Ады, но и для остальных объектно ориентированных языков программирования.

14.1.4 Пустые записи (*null record*) и расширения

Достаточно часто возникает необходимость создания типа который не имеет полей (атрибутов), а имеет только операции (методы). Как правило, подобный тип используется для последующего построения целой иерархии типов как базовый (или корневой) тип. Это может быть достигнуто описанием "пустой" записи при описании типа:

```
type Root is tagged
    record
        null;
    end record;
```

Для таких случаев, Ада обеспечивает специальный синтаксис описания "пустых" записей:

```
type Root is tagged null record;
```

Описание операций над таким типом традиционно, и может быть выполнено в спецификации пакета:

```
procedure Do_Something (Item : in out Root);
```

Также, бывают случаи, когда описание нового производного типа не нуждается в расширении (нет добавления новых полей), но при этом добавляются новые подпрограммы или переопределяются старые. Вид описания такого расширения следующий:

```
type Child is new Root with null record;

procedure Child_Method (Item : in out Child);
```

В результате такого описания получается новый тип "Child", производный от типа "Root", который не будет иметь дополнительных компонентов, но имеет дополнительный метод — "Child_Method".

14.1.5 Абстрактные типы и подпрограммы

В большинстве случаев, типы которые разрабатываются как родоначальники иерархий производных типов не предназначены для непосредственного описания объектов (переменных). Такие типы называют абстрактными, а для их описания используется зарезервированное слово "**abstract**". Например:

```
type Empty_Root is abstract tagged null record;

type Simple_Root is abstract tagged
    record
        Simple_Field: Integer;
    end record;
```

Здесь, в первом случае, тип "Empty_Root" — это абстрактный тип является "пустой" записью которая не содержит никаких полей. В свою очередь, тип "Simple_Root", который также описан как абстрактный, содержит единственное поле "Simple_Field" типа "**Integer**".

Абстрактный тип может иметь абстрактные подпрограммы. Абстрактными называют подпрограммы которые фактически не имеют тела (то есть реализации), а это значит, что они обязательно должны быть переопределены в производных типах. Смысл использования описания абстрактного типа с абстрактными подпрограммами заключается в том, что все производные типы, в последствии, будут вынуждены поддерживать общую логическую функциональность.

Описание абстрактного типа имеющего абстрактные подпрограммы может иметь следующий вид:

```
package Sets is
    type Set is abstract tagged null record;

    function Empty return Set is abstract;
    function Empty (Element : Set) return Boolean is abstract;
    function Union (Left, Right : Set) return Set is abstract;
    function Intersection (Left, Right : Set) return Set is abstract;
    procedure Insert (Element : Natural; Into : Set) is abstract;
end Sets;
```

Это описание множества натуральных чисел взято из справочного руководства по языку Ада. Примечательно то, что компиляор не позволит описать переменную типа "**Set**", поскольку тип "**Set**" — это абстрактный тип. Таким образом, следующий пример не будет компилироваться:

```
with Sets;          use Sets;

procedure Wont_Compile is

    My_Set : Set;          -- НЕДОПУСТИМО!!! абстрактный тип

begin
    null;
end Wont_Compile;
```

Как правило, тип, производный от абстрактного, обеспечивает реализацию функциональности которая была задана в абстрактном типе предке. Например:

```
with Sets;

package Quick_Sets is

    type Bit_Vector is array (0..255) of Boolean;
    pragma Pack (Bit_Vector);

    type Quick_Set is new Sets.Set with
        record
            Bits : Bit_Vector := (others => False);
        end record;

    -- объявление конкретной реализации
    function Empty return Quick_Set;
    function Empty (Element : Quick_Set) return Boolean;

    . . .
```

Показанный выше схематический пример описывает тип "Quick_Set", производный от абстрактного типа "**Set**". В этом примере, тип "Quick_Set" осуществляет реализацию функциональности которая была задана в абстрактном типе "**Set**". В результате, после описания типа "Quick_Set", мы можем описывать и использовать объекты типа "Quick_Set" логическая функциональность которых будет соответствовать функциональности абстрактного типа "**Set**".

Необходимо также заметить, что Ада позволяет чтобы вновь описываемый тип, производный от абстрактного типа, также был абстрактным. Используя такие возможности можно разработать иерархию абстрактных типов, которая будет определять логическую функциональность программной подсистемы или всего программного проекта.

14.2 Динамическое связывание и полиморфизм

Обеспечение полиморфности поведения объектов полагается на механизм который осуществляет связывание места вызова подпрограммы с конкретной реализацией (телом) подпрограммы.

Следует заметить, что механизм совмещения имен подпрограмм и знаков операций Ады, при использовании традиционных не тэговых типов, обладает свойством полиморфизма. Действительно, одно и то же имя может быть использовано для целого множества различных реализаций подпрограмм (что, собственно, и является полиморфностью). Однако, как мы знаем, все совмещаемые операции должны быть различимы по профилю. Таким образом, до настоящего момента предполагалось, что реализация подпрограммы, которую необходимо вызвать для обработки объекта, всегда точно определяется на этапе компиляции программы. В результате, в подобных случаях, и полиморфизм, предоставляемый механизмом совмещения, и связывание называют статическими, поскольку типы всех обрабатываемых объектов известны на этапе компиляции.

При использовании тэговых типов, некоторые объекты могут принадлежать одному иерархическому образованию класса и обладать некоторыми общими логическими свойствами (напомним, что понятие "класс" Ады отличается от понятия "класс", которое используется в других объектно-ориентированных языках программирования). Следовательно, выполнение обработки таких объектов по единым логическим правилам

будет более предпочтительным. Однако, в такой ситуации фактический тип индивидуального объекта не может быть определен вплоть до начала выполнения программы. А это значит, что связывание места вызова подпрограммы с конкретной реализацией подпрограммы должно осуществляться в процессе выполнения программы на основе информации о фактическом типе объекта во время выполнения программы. В таких ситуациях, и связывание, и полиморфизм называют динамическими.

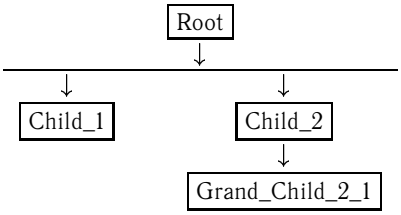
14.2.1 Надклассовые типы (*wide class types*)

Все типы, производные и расширенные (прямо или косвенно) от определенного тэгового типа "T", принадлежат одному иерархическому образованию класса, корнем которого будет тэговый тип "T". Тэговый тип "T" неявно ассоциирован с надклассовым типом который обозначает целый класс типов — иерархию включающую данный тэговый тип "T" и все производные от него типы. Надклассовый тип не имеет явного имени и обозначается как атрибут "T'Class", где "T" — имя соответствующего тэгового типа. В таком контексте, явно описанные типы удобно называть индивидуальными типами, чтобы подчеркнуть их отличие от надклассовых типов.

В иерархическом образовании класса каждый индивидуальный тип идентифицируется тэгом (*tag*), который скрыто присутствует в каждом тэговом типе (отсюда, собственно, и название "тэговые" типы). Наличие такой информации позволяет осуществлять идентификацию индивидуального типа объекта во время выполнения программы (то есть, динамически) и, используя результат такой идентификации, находить для этого объекта соответствующую реализацию операций.

Следует также заметить, что такой подход является некоторым ослаблением традиционной модели строгой типизации Ады, поскольку любой тип, производный от типа "T", может быть неявно преобразован в надклассовый тип "T'Class".

Для демонстрации сказанного, предположим, что у нас имеется следующая иерархия типов:



Предположим также, что спецификация пакета, описывающего показанную выше иерархию типов, имеет следующий вид (заметим, что пустые записи и расширения использованы для упрощения примера):

```
package Simple_Objects  is

   -- тип Root  и его примитивные операции

   type Root is tagged null record;

   function The_Name (Self: in Root) return String;
   procedure Show (Self: in Root'Class);


   -- тип Child_1  и его примитивные операции

   type Child_1 is new  Root  with null record;

   function The_Name (Self: in Child_1) return String;


   -- тип Child_2  и его примитивные операции

   type Child_2 is new  Root  with null record;

   function The_Name (Self: in Child_2) return String;
```

```
   -- тип Grand_Child_2_1  и его примитивные операции

   type Grand_Child_2_1 is new  Child_2  with null record;

   function The_Name (Self: in Grand_Child_2_1) return String;

end Simple_Objects;
```

В этом случае, все типы показанной иерархии ("Root", "Child_1", "Child_2" и "Grand_Child_2_1") будут принадлежать к надклассовому типу "Root'Class".

Надклассовые типы могут быть использованы при описании переменных, параметров подпрограмм и объектов ссылочного типа.

При описании любой переменной надклассового типа, следует учитывать, что любой надклассовый тип "T'Class" является неограниченным, а это значит, что компилятору заранее не известен размер резервируемого для размещения такой переменной пространства. Следовательно, при описании переменной надклассового типа, необходимо обязательно предусматривать инициализацию такой переменной начальным значением:

```
V : T'Class := значение_инициализации ;
```

В результате выполнения инициализации, для значения переменной надклассового типа, будет осуществляться фиксирование индивидуального типа переменной, которое будет сохраняется на протяжении всего времени жизни данной переменной. Следует заметить, что *значение_инициализации*, в свою очередь, может быть динамически вычисляемым выражением.

С учетом приведенной ранее иерархии типов, рассмотрим простой пример следующего описания переменной:

```
Instance : Root'Class := Child_1'(Root with null record);
```

Здесь, "Instance" — это переменная надклассового типа "Root'Class", а ее индивидуальный тип указывается значением инициализации как тип "Child_1".

Формальный параметр подпрограммы также может иметь надклассовый тип. В таком случае, каждый фактический параметр, принадлежащий к иерархическому образованию класса (или, короче, — к классу), будет совместим с формальным параметром. Такие подпрограммы не будут ограничены использованием только одного специфического типа и могут принимать парараметр тип которого принадлежит указанному классу. Как правило, такие подпрограммы называют надклассовыми подпрограммами. Например:

```
procedure Show (Self : in Root'Class);
```

В результате, любой фактический параметр, который принадлежит надклассовому типу "Root'Class" тэгового типа "Root" (напомним, что это все типы, производные от типа "Root") будет совместим с формальным параметром "Self". Например:

```
declare

   Root_Instance              : Root;
   Child_1_Instance           : Child_1;
   Child_2_Instance           : Child_2;
   GRand_Child_2_1_Instance   : GRand_Child_2_1;

   Instance : Root'Class := Child_1'(Root with null record);

begin

   Show (Root_Instance);
   Show (Child_1_Instance);
   Show (Child_2_Instance);
   Show (GRand_Child_2_1_Instance);
   Show (Instance);

end;
```

Надклассовые типы могут быть использованы при описании ссылочных типов:

```
type Root_Ref is access Root'Class;
```

Как правило, описанные подобным образом типы называют типами надклассовых ссылок, а использование подобных типов в программе позволяет создавать переменные которые, во время выполнения программы, будут способны хранить значения с типами принадлежащими любому типу класса:

```
declare

    Any_Instance:  Root_Ref;

begin
    Any_Instance := new Child_1'(Root with null record);

        . . .

    Any_Instance := new Child_2'(Root with null record);
        . . .
end;
```

В показанном выше примере, переменная "Any_Instance" имеет тип надклассовой ссылки "Root_Ref" и может обозначать любой объект который принадлежит классу "Root'Class". Таким образом, как показано в примере, индивидуальным типом объекта обозначаемого переменной "Any_Instance" сначала будет тип "Child_1", а затем — "Child_2".

14.2.2 Проверка типа объекта во время выполнения программы

В процессе выполнения программы можно осуществить проверку объекта на принадлежность его к какому-либо индивидуальному типу путем использования атрибута "'Tag".

Чтобы продемонстрировать это, предположим, что реализация процедуры "Show" описывается следующим образом:

```
procedure Show (Self: in Root'Class) is
begin

    if     Self'Tag = Root'Tag  then
        Ada.Text_IO.Put_Line ("Root");
    elsif Self'Tag = Child_1'Tag  then
        Ada.Text_IO.Put_Line ("Child_1");
    elsif Self'Tag = Clild_2'Tag  then
        Ada.Text_IO.Put_Line ("Clild_2");
    elsif Self'Tag = Grand_Child_2_1'Tag  then
        Ada.Text_IO.Put_Line ("Grand_Child_2_1");
    else
        Ada.Text_IO.Put_Line ("Unknown type");
    end if;

end Show;
```

Кроме того, в процессе выполнения программы, возможно осуществление проверки принадлежности (или не принадлежности) типа объекта к какому-либо классу. Для выполнения таких проверок используются операции проверки диапазона "**in**" и "**not in**":

```
    . . .
    if  Some_Instance in Child_1'Class    then
        . . .
    end if;
    . . .
```

В данном примере выполняется проверка принадлежности переменной "Some_Instance" к иерархии типов, корнем которой будет тип "Child_1", причем, предполагается, что переменная "Some_Instance" является переменной надклассового или тэгового типа.

Следует также обратить внимание на то, что для выполнения этой проверки используется "Some_Instance **in** Child_1'Class", а не "Some_Instance **in** Child_1".

14.2.3 Динамическая диспетчеризация

При внимательном рассмотрении показанной ранее реализации процедуры "Show" не трудно заметить, что хотя она и использует средства тэговых типов, позволяющие определить фактический тип объекта во время выполнения программы, аналогичный результат мог бы быть получен путем использования вместо тэговых типов обычных записей с дискриминантами.

В данном случае, подразумевается, что если мы решим описать новый тип, производный от любого типа входящего в показанную иерархию ("Root", "Child_1", "Child_2" и "Grand_Child_2_1"), то результатом работы такой реализации процедуры "Show" всегда будет сообщение "`Unknown type`", извещающее о том, что фактический тип параметра "Self" — не известен. Например, такая ситуация может возникнуть когда мы опишем новый тип "Grand_Child_1_1", производный от типа "Child_1", в каком-либо другом пакете.

С первого взгляда может показаться, что будет достаточно переписать процедуру "Show" с учетом нового типа "Grand_Child_1_1". Действительно, такое решение будет работать, но это значит, что при описании каждого нового типа, входящего в эту иерархию, мы вынуждены вносить изменения в уже написанные и отлаженные модули.

Для того, чтобы избавиться от таких трудностей, реализацию процедуры "Show" надо действительно переписать, но так, чтобы задействовать механизм динамической диспетчеризации вызовов подпрограмм, который предоставляет использование тэговых типов.

Вспомним, что при демонстрации примера иерархии типов, был также приведен пример спецификации пакета *Simple_Objects*, в котором эта иерархия типов описывается. Теперь, перед непосредственным обсуждением механизма динамической диспетчеризации вызовов подпрограмм, предположим, что описание тела этого пакета имеет следующий вид:

```
with  Ada.Text_IO;

package body Simple_Objects  is

   -- примитивные операции типа Root

   function The_Name (Self: in Root) return String is
   begin
      return ("Root");
   end The_Name;

   procedure Show (Self: in Root'Class) is
   begin
      Ada.Text_IO.Put_Line ( The_Name(Self) );
   end Show;


   -- примитивные операции типа Child_1

   function The_Name (Self: in Child_1) return String is
   begin
      return ("Child_1");
   end The_Name;


   -- примитивные операции типа Child_2

   function The_Name (Self: in Child_2) return String is
   begin
      return ("Child_2");
   end The_Name;


   -- примитивные операции типа Grand_Child_2_1

   function The_Name (Self: in Grand_Child_2_1) return String is
   begin
      return ("Grand_Child_2_1");
```

```
    end The_Name;

end Simple_Objects;
```

Не сложно догадаться, что особое внимание следует обратить на реализацию процедуры "Show", которая теперь, перед вызовом "Ada.Text_IO.Put_Line", выдающим строку сообщения, вызывает функцию "The_Name", возвращающую строку которая, собственно, содержит текст сообщения. Заметим также, что важным моментом в этой реализации является то, что процедура "Show" имеет формальный параметр надклассового типа, что позволяет ей принимать в качестве фактического параметра любые объекты тип которых принадлежит данной иерархии типов.

При дальнейшем внимательном рассмотрении спецификации и тела пакета *Simple_Objects*, следует обратить внимание на то, что функция "The_Name" описана для всех типов иерархии ("Root", "Child_1", "Child_2" и "Grand_Child_2_1") и является примитивной операцией для этих типов. Напомним, что в случае тэговых типов, примитивные операции являются потенциально диспетчеризуемыми операциями.

В результате, процедура "Show", принимая во время выполнения программы фактический параметр, осуществляет диспетчеризацию вызова соответствующей реализации функции "The_Name" на основании информации о тэге фактического параметра. Это значит, что в данном случае присутствует динамическое связывание которое обеспечивает динамическую полиморфность поведения объектов.

Таким образом, для выполнения диспетчеризации вызова подпрограммы во время выполнения программы должно соблюдаться два условия:

- подпрограмма должна иметь формальный параметр тэгового типа
- при вызове подпрограммы, фактический параметр, соответствующий параметру тэгового типа должен быть объектом надклассового типа

Следовательно, если фактический надклассовый тип представляет объекты двух или более индивидуальных типов, которые имеют подпрограммы с одинаковыми параметрами (исключая параметр надклассового типа), то определение реализации подпрограммы на этапе компиляции (иначе, статическое связывание) не возможно. Вместо этого, осуществляется динамическое связывание во время выполнения программы.

Возвращаясь к проблеме с которой мы начинали обсуждение диспетчеризации, следует заметить, что использование этого механизма в процедуре "Show", позволяет теперь описывать новые типы, производные от любого типа входящего в иерархию, без необходимости изменения реализации процедуры "Show". Действительно, при описании нового производного типа, достаточно описать соответствующую реализацию функции "The_Name" для нового типа, не внося никаких изменений в ранее написанные модули. Заметим также, что если при описании нового производного типа не будет предусмотрена реализация функции "The_Name", то для этого типа процедура "Show", используя диспетчеризацию, будет вызывать реализацию функции "The_Name" унаследованную от типа предка.

Мы рассмотрели случай динамической диспетчеризации в подпрограмме при описании которой используется формальный параметр надклассового типа. Следует заметить, что вызов подпрограммы с параметром надклассового ссылочного типа также является диспетчеризуемым:

```
... The_Name (Any_Instance.all)...
    -- Any_Instance может обозначать объект,
    -- который принадлежит любому типу
    -- класса Root'Class
```

Еще одним случаем диспетчеризации является использование переменной надклассового типа. В этом случае нужно обратить внимание на отличие диспетчеризуемого вызова от обычного вызова:

```
declare
    Instance_1 : Root;
    Instance_2 : Root'Class ... ;
begin
    ... The_Name (Instance_1)...
        -- статическое связывание: компилятор знает индивидуальный тип
        -- Instance_1, поэтому он может определить реализацию

    ... The_Name (Instance_2)...
        -- динамическое связывание: Instance_2 может принадлежать любому
```

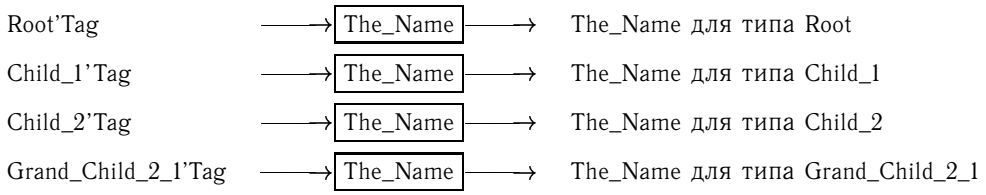
```
        -- типу класса Root'Class
        -- поэтому компилятор не может точно определить реализацию
        -- подпрограммы, она будет выбрана во время выполнения программы
end ;
```

14.2.4 Модель механизма диспетчеризации

Для лучшего понимания особенностей использования тэговых типов и свойств диспетчеризации вызовов операций, может оказаться полезным рассмотрение возможной модели реализации механизма диспетчеризации. Следует подчеркнуть, что не смотря на достаточную реалистичность нашей модели, мы рассматриваем только возможную модель, сознательно опуская множество нюансов, а это значит, что реализация для какого-либо конкретного компилятора может не соответствовать этой модели.

Согласно нашей модели, подразумевается, что тэг — это указатель на таблицу диспетчеризации вызовов. Каждый элемент такой таблицы содержит указатель на реализацию подпрограммы которая является примитивной операцией. Диспетчеризация осуществляется путем выполнения косвенного вызова с помощью таблицы диспетчеризации, а необходимая примитивная операция выбирается как соответствующий индекс элемента в таблице диспетчеризации.

Таким образом, наша модель для рассмотренной ранее иерархии типов ("Root", "Child_1", "Child_2" и "Grand_Child_2_1") будет иметь следующий вид:



Следует заметить, что наша иллюстрация максимально упрощена и не содержит множества предопределенных примитивных операций, кроме того, описания типов иерархии также были очень простыми.

Базовый адрес каждой таблицы соответствует значению тэга и содержится в значении надклассового объекта. Примечательно также, что в таблицах диспетчеризации не содержатся указатели на надклассовые операции (подобные процедуре "Show"), поскольку они не являются диспетчеризуемыми операциями.

Таким образом, при динамическом связывании во время выполнения программы, на основе информации о тэге объекта осуществляется выбор соответствующей таблицы диспетчеризации. Далее, по выбранной таблице, осуществляется косвенный вызов подпрограммы, для которой индекс в таблице соответствует требуемой примитивной операции.

Для лучшей демонстрации этой идеологии, предположим теперь, что мы решили описать новый тип "Grand_Child_1_1", производный от типа "Child_1", следующим образом:

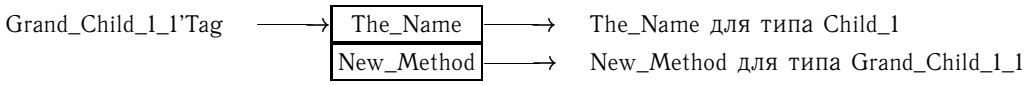
```
with Simple_Objects;   use Simple_Objects;

package Simple_Objects_New is

    type Grand_Child_1_1 is new Child_1 with null record;

    procedure New_Method (Self: in Grand_Child_1_1);
    procedure New_Method_Call (Self: in Grand_Child_1_1'Class);
        . . .

end Simple_Objects_New;
```

Дополнительно предположим, что далее в пакете *Simple_Objects_New* описываются типы, производные от типа "Grand_Child_1_1", и то, что реализация надклассовой процедуры "New_Method_Call" выполняет диспетчеризуемый вызов процедуры "New_Method". Тогда, в этом описании содержится два примечательных факта:

- Мы сознательно не определили для типа "Grand_Child_1_1" реализацию функции "The_Name". Следовательно, она будет унаследована от типа "Child_1".

- Описание типа "Grand_Child_1_1" определяет новую примитивную диспетчеризуемую операцию — процедуру "New_Method".

Согласно нашей модели, вид тэга для типа "Grand_Child_1_1" будет следующий:



Из данной иллюстрации видно, что поскольку тип "Grand_Child_1_1" не имеет собственной реализации функции "The_Name", то таблица диспетчеризации, которая соответствует его тэгу, содержит указатель на реализацию функции "The_Name" для его предка, типа "Child_1". Таким образом, вызов процедуры "Show", для типа "Grand_Child_1_1", осуществит вызов унаследованной от типа "Child_1" реализации функции "The_Name" (в соответствии с индексом в таблице диспетчеризации).

Примечательно также, что описание новой примитивной операции (процедура "New_Method") обусловило появление нового элемента в таблице диспетчеризации для типа "Grand_Child_1_1".

14.2.5 Вызов переопределенной операции предка

Достаточно часто, реализация примитивной операции производного типа нуждается в вызове переопределенной примитивной операции предка. Предположим, что для типов "Root" и "Child_1" существуют описания операции "Display" имеющие следующий вид:

```
    . . .
procedure Display (Self: in Root);
procedure Display (Self: in Child_1);
    . . .
```

Поскольку такие операции совмещены, то можно сказать, что реализация операции "Display" типа "Root" (предка для типа "Child_1") является "затененной", в виду переопределения реализации в производном типе "Child_1".

В таком случае, для вызова "затененной" операции предка можно использовать следующее:

```
procedure Display (Self: in Child_1) is
begin
    Display (Root (Self)); -- вызов "затененной" операции предка
    . . .
end Display;
```

Здесь, для вызова "затененной" реализации операции предка, используется явное преобразование представления параметра "Self" к типу "Root". За счет этого, в подобных случаях, всегда осуществляется статическое связывание, а также отсутствует опасность получения бесконечного рекурсивного вызова.

14.2.6 Динамическая передиспетчеризация

Известно, что производный тип всегда может переопределить примитивную операцию своего предка, то есть, предусмотреть свою собственную реализацию примитивной операции. В случае использования классов, необходимо заботиться в правильном распознавании вызова реализации для потомка от вызова реализации для предка. Рассмотрим следующий схематический вариант реализации процедуры "Display" для типа "Root":

```
procedure Display (Self: in Root) is
begin
    Put (The_Name (Self) & ... ); -- ???
    . . .
end Display;
```

В данном случае, при вызове реализации "Display" для типа "Child_1", которая была показана ранее, первая же инструкция выполняет вызов затененной реализации предка, то есть, реализации для типа "Root": "Display (Root(Self))".

Показанный пример реализации процедуры "Display" для типа "Root" всегда будет выполнять вызов функции "The_Name" полагая, что индивидуальным типом параметра "Self" является тип "Root". Таким образом, этот вызов будет статически связан с реализацией функции "The_Name" для типа "Root".

Следовательно, при вызове реализации процедуры "Display" для типа "Child_1", в результате вызова затененной реализации предка будет получена та же строка имени, что и для типа "Root".

Для того, чтобы процедура "Display" для типа "Root" правильно осуществила вызов той реализации функции "The_Name", которая соответствует фактическому индивидуальному типу параметра "Self" ее необходимо переписать следующим образом:

```
procedure Display (Self: in Root) is
begin
    Put (The_Name (Root'Class (Self)) & ... ); -- передиспетчеризация
    . . .
end Display;
```

В данном случае, в реализации процедуры "Display" для типа "Root" выполняется преобразование параметра "Self" к типу "Root'Class" предусматривая передиспетчеризацию вызова к корректной реализации "The_Name", в зависимости от тэга параметра "Self".

14.2.7 Двойная диспетчеризация

Бывают случаи, когда при описании в одном пакете двух различных тэговых типов возникает желание описать подпрограмму которая принимает параметры обоих тэговых типов. В подобной ситуации получается, что такая подпрограмма будет считаться примитивной операцией для обоих типов. Однако, компилятор должен иметь какой-нибудь способ определения того, какой параметр использовать для диспетчеризации. Чтобы, яснее представить суть подобной проблемы, рассмотрим следующий (не корректный!!!) пример

```
type Message_Type is
    tagged record
        . . .
    end record;

type Output_Device is
    tagged record
        . . .
    end record;

procedure Put (M : in Message_Type; D : in Output_Device);
            -- записать сообщение M в устройство вывода D (не допустимо!!!)

    . . .
```

На первый взгляд, идея выглядит достаточно логично, при наличии различных типов устройств вывода иметь процедуру, которая позволяет выдать определенного типа сообщение в устройство вывода определенного типа. Каждый тип сообщения может переопределить соответствующим образом процедуру вывода "**Put**" для того чтобы осуществить свой вывод необходимым образом. Каждый тип устройства вывода переопределяет процедуру "**Put**" с целью использования средств предоставляемых устройством вывода. Однако, при вызове процедуры "**Put**" становится не понятным какую версию этой процедуры необходимо использовать, ту которая определена в типе, производном от типа "Message_Type", или ту которая определена в типе, производном от типа "Output_Device".

Ада решает эту проблему очень просто: использование подпрограмм которые являются примитивными операциями для двух (и более) тэговых типов, подобно процедуре "**Put**", — не допускается. Следовательно, необходимо изменить описание процедуры таким образом, чтобы она была примитивной операцией только для одного типа. Примером такой модификации может служить следующее:

```
procedure Put (M : in Message_Type;
               D : in Output_Device'Class);
```

Теперь, параметр "D" больше не имеет тип "Output_Device", а значит, процедура "**Put**" больше не является примитивной операцией для типа "Output_Device". Однако, внутри процедуры, значение "Output_Device'Class" приведет к осуществлению диспетчеризации в случае вызова примитивной операции для типа "Output_Device". Подобный прием называют двойной диспетчеризацией. Например, предположим, что тип "Output_Device" имеет следующую примитивную операцию:

```
procedure Write_Output (D : in Output_Device; S : in String);
```

В этом случае, тип "Message_Child_Type", производный от типа "Message_Type", может переопределить реализацию процедуры "**Put**" приблизительно следующим образом:

```
procedure Put (M : in Message_Child_Type;
               D : in Output_Device'Class) is
begin
    Put (Message_Type (M), D);     -- вызов версии Put предка
    . . .
    Write_Output (D, ... );        -- отображение данных сообщения
    . . .
end Put;
```

Вызов процедуры "**Put**", осуществленный с вовлечением параметра надклассового типа, который соответствует типу "Message_Child_Type", и параметром с определенным типом устройства вывода приведет к вызову показанной выше версии процедуры "**Put**". Это приведет к тому, что отображение данных сообщения будет осуществлено с помощью процедуры "Write_Output", реализация которой будет выбрана в результате диспетчеризации.

14.3 Стандартные низкоуровневые средства, пакет *Ada.Tags*

Стандартным низкоуровневым средством работы с тэговыми типами является пакет *Ada.Tags*. Спецификация этого пакета, согласно стандарта, имеет следующий вид:

```
package Ada.Tags is

    type Tag is private;

    function Expanded_Name (T : Tag) return String;
    function External_Tag (T : Tag) return String;
    function Internal_Tag (External : String) return Tag;

    Tag_Error : exception;

private

    . . .   -- стандартом не определено

end Ada.Tags;
```

Функция "Expanded_Name" возвращает полное расширенное имя типа, индефицируемого значением тэга, в виде строки (в верхнем регистре). Результат будет зависеть от реализации компилятора, если тип описан внутри неименованного блока инструкций.

Функция "External_Tag" возвращает строку, которая может быть использована для внешнего представления указанного тэга. Вызов "External_Tag(S'Tag)" эквивалентен обращению к атрибуту "S'External_Tag".

Функция "Internal_Tag" возвращает тэг, который соответствует указанному внешнему представлению тэга, или возбуждает исключение "Tag_Error", если ни для одного из типов, в пределах раздела программы, указанная строка не является внешним представлением тэга.

Глава 15

Контролируемые типы (*controlled types*)

15.1 Общие сведения

Для того чтобы обеспечить сохранение абстракции, при автоматическом распределении и освобождении системных ресурсов, Ada95 предусматривает контролируемые типы. Контролируемыми типами называют типы, производные от типа "Controlled" или от лимитированного типа "Limited_Controlled". Оба этих типа описаны в стандартном пакете *Ada.Finalization*:

```
package Ada.Finalization is
pragma Preelaborate (Finalization);

    type Controlled is abstract tagged null record;

    procedure Initialize (Object : in out Controlled);
    procedure Adjust (Object : in out Controlled);
    procedure Finalize (Object : in out Controlled);


    type Limited_Controlled is abstract tagged limited null record;

    procedure Initialize (Object : in out Limited_Controlled);
    procedure Finalize (Object : in out Limited_Controlled);


private
    ... -- стандартом не определено

end Ada.Finalization;
```

Основная идея использования объектов контролируемых и лимитированных контролируемых типов заключана в том, что вызов управляющих операций " Initialize", "Adjust" и "Finalize" осуществляется автоматически:

- " Initialize" — "инициализация", вызывается сразу после создания объекта.
- "Finalize" — "очистка", вызывается непосредственно перед разрушением объекта (включая переопределение содержимого объекта).
- "Adjust" — осуществляет "подгонку" содержимого объекта после выполнения присваивания.

Необходимо заметить, что объекты типа "Controlled" или "Limited_Controlled" не возможно использовать непосредственно, поскольку оба типа являются абстрактными. Вместо этого, для использования контролируемых объектов, необходимо описать тип, который будет производным от типа "Controlled" или от типа "Limited_Controlled".

Примечательным также является то, что предопределенные управляющие операции " Initialize", "Adjust" и "Finalize" не являются абстрактными (они могут быть нормально унаследованы производным типом), однако, реализация этих предопределенных операций не выполняет никаких действий. Поэтому предполагается, что эти операции будут соответствующим образом переопределены для определяемого пользователем контролируемого типа, производного от типа "Controlled" или "Limited_Controlled".

Чтобы более наглядно продемонстрировать идею использования контролируемых типов, рассмотрим следующее:

```
declare
    A: T;     -- создание объекта A и вызов Initialize (A)
    B: T;     -- создание объекта B и вызов Initialize (B)
begin
    . . .
    A := B;   -- вызов Finalize (A), копирование значения B в A и вызов Adjust (A)
    . . .
end;   -- вызов Finalize (A) и Finalize (B)
```

В данном случае, предполагается, что тип "T" является производным от типа "Controlled". Отличие использования типов, производных от типа "Limited_Controlled", заключается только в отсутствии операции "Adjust".

При использовании объектов контролируемых типов, следует обратить внимание на то, что в случаях когда при описании объекта контролируемого типа указывается значение инициализации:

```
. . .
C: T := значение_инициализации;  -- вызов Initialize (C) не выполняется!!!
. . .
```

то автоматический вызов операции " Initialize" выполняться не будет.

15.2 Управление динамическими объектами

Одним из широко распространенных случев использования объектов контролируемых типов являестя решение проблемы "утечки" памяти при работе с динамическими данными.

Предположим, что у нас есть пакет *Lists* который содержит описание связанного списка "List", и спецификация этого пакета выглядит следующим образом:

```
package Lists is

    type List is private;

    Underflow : exception;

    procedure Insert_At_Head (Item : in out List; Value : in Integer);
    procedure Remove_From_Head (Item : in out List; Value : out Integer);

    procedure Insert_At_Tail (Item : in out List; Value : in Integer);
    procedure Remove_From_Tail (Item : in out List; Value : out Integer);

    function Full (Item : List) return Boolean;
    function Empty (Item : List) return Boolean;

private
    type List is ... -- полное описание типа
    . . .

end Lists;
```

Заметим, что тип связанного списка "List", который описан в этом пакете, не является контролируемым типом. Теперь, рассмотрим следующий пример, использующий описание пакета *Lists*:

```
with Lists;          use Lists;

procedure Memory_Leaks_Demo is

    A, B   : List;
    Result : Integer;

    procedure Loose_Memory is
        C : List;
    begin
```

```
        Insert_At_Head (C, 1);
        Insert_At_Head (C, 2);
    end Loose_Memory; -- при попадании в эту точку,
                      -- C выходит за пределы области видимости
                      -- и теряется память ассоциируемая с этим узлом

begin
    Insert_At_Head (A, 1);
    Insert_At_Head (A, 2);

    Insert_At_Head (B, 3);
    Insert_At_Head (B, 4);

    B := A;
    -- B и A указывают на один и тот же список
    -- все узлы "старого" списка B - теряются

    Remove_From_Tail (A, Result);
    -- изменяет как список A, так и список B

end Memory_Leaks_Demo;
```

В данном примере наглядно демонстрируются проблемы "утечки" памяти при работе с объектами связанного списка "List":

- когда объект связанного списка выходит за пределы области видимости, пространство динамической памяти, выделенное для размещения этого объекта, не восстанавливается
- в случае выполнения присваивания, осуществляется копирование только указателя на "голову" списка (начало списка), а копирование остальной части списка не выполняется

Рассмотрим вариант модификации пакета *Lists* в котором тип связанного списка "List" является контролируемым. Спецификация пакета будет иметь следующий вид:

```
with Ada.Finalization;

package Lists is

    type List is private;

    Underflow : exception;

    procedure Insert_At_Head (Item : in out List; Value : in Integer);
    procedure Remove_From_Head (Item : in out List; Value : out Integer);

    procedure Insert_At_Tail (Item : in out List; Value : in Integer);
    procedure Remove_From_Tail (Item : in out List; Value : out Integer);

    function Full (Item: List) return Boolean;
    function Empty (Item: List) return Boolean;

private

    -- обычные описания для списка

    type Node;
    type Ptr is access Node;

    type Node is
        record
            Value : Integer;
            Next  : Ptr;
        end record;
```

```
    -- только "голова" списка - "специальная"

    type List is new Ada.Finalization.Controlled with
        record
            Head : Ptr;
        end record;


    -- Initialize не нужна (указатели автоматически устанавливаются в null)

    -- procedure Initialize (Item : in out List);
    procedure Adjust (Item : in out List);
    procedure Finalize (Item : in out List);

end Lists;
```

Примечательным фактом является описание подпрограмм "Adjust" и "Finalize" в приватной части спецификации пакета. Это предотвращает от непосредственной возможности их неуместного вызова клиентами. Тело пакета *Lists* (с подробными комментариями) будет иметь вид:

```
with Unchecked_Deallocation;

package body Lists is

    -- подпрограмма освобождения памяти занимаемой узлом
    procedure Free is new Unchecked_Deallocation (Node, Ptr);

    ------------------------------------------------------------
    -- реализация остальных внутренностей (Insert_At_Head, Remove_From_Head...)

        . . .

    ------------------------------------------------------------
    -- дан указатель на список, подпрограмма будет размещать
    -- в памяти новый, идентичный первому, список
    function Copy_List (Item : Ptr) return Ptr is
    begin
        if Item = null then
            return null;
        else
            return new Node'(Item.Value, Copy_List (Item.Next));
        end if;
    end Copy_List;

    ------------------------------------------------------------
    -- при присваивании B := A, B будет только переписано содержимым A.
    -- для связанного списка это подразумевает, что оба, A и B,
    -- указывают на один и тот же объект.
    -- теперь, необходимо сделать физическую копию узлов, на которые указывает B,
    -- и переставить указатель начала списка на начало копии списка, который мы
    -- только что сделали
    procedure Adjust (Item : in out List) is
    begin
            Item.Head := Copy_List (Item.Head);
    end Adjust;

    ------------------------------------------------------------
    -- освободить всю память, занимаемую узлами списка,
    -- при разрушении списка
    procedure Finalize (Item : in out List) is
        Upto : Ptr := Item.Head;
        Temp : Ptr;
    begin
        while Upto /= null loop
            Temp := Upto;
```

```
            Upto := Upto.Next;
            Free (Temp);
        end loop;

        Item.Head := null;
    end Finalize;

end Lists;
```

Ниже представлен пример программы которая использует модифицированную версию пакета *Lists* (где тип "List" является контролируемым).

```
with Lists;         use Lists;

procedure Controlled_Demo is

    A : List;                   -- автоматический вызов Initialize (A)
    B : List;                   -- автоматический вызов Initialize (B)

begin
    Insert_At_Head (A, 1);
    Insert_At_Head (A, 2);

    Insert_At_Head (B, 3);
    Insert_At_Head (B, 4);
        ------------------------------------------
        --
        --    A --> | 2 |-->| 1 |--> null
        --    B --> | 4 |-->| 3 |--> null
        --
        ------------------------------------------

    B := A;
        ------------------------------------------
        --
        --    Finalize (B);
        --    освобождение узлов B, до перезаписи
        --
        --    A --> | 2 |-->| 1 |--> null
        --    B --> null
        --
        --    копирование A в B
        --    теперь они _оба_ указывают на один и тот же список
        --
        --    A --> | 2 |-->| 1 |--> null
        --    B ----^
        --
        --    Adjust (B);
        --    B копирует список, на который он в текущий момент
        --    указывает, и после этого указывает на новый список
        --
        --    A --> | 2 |-->| 1 |--> null
        --    B --> | 2 |-->| 1 |--> null
        --
        ------------------------------------------

end Controlled_Demo;
        ------------------------------------------
        --
        --    Finalize (A), Finalize (B).
        --    освобождение памяти ассоциируемой с A и B
        --
        --    A --> null
        --    B --> null
        --
```

Таким образом, использование контролируемых типов предоставляет удобное средство управления жизненным циклом динамических объектов, которое позволяет избавить клиентов типа от необходимости непосредственного управления динамическим распределением памяти.

15.3 Счетчик использования

Бывают случаи, когда при обработке множества объектов какого-либо типа данных необходимо использование определенного ресурса системы. Примером подобной ситуации может служить необходимость протоколирования состояния различных объектов одного и того же типа в определенном файле, который специально для этого предназначен.

Для большей ясности, представим себе случай, когда тип данных описывает пользователя, получающего доступ к базе данных. Объектов-пользователей базы данных может быть множество. При этом, необходимо протоколировать действия каждого пользователя, осуществляемые при работе с базой данных, в едином файле протокола.

Подобная ситуация подразумевает, что при создании хотя бы одного объекта указанного типа данных, файл протокола состояний должен быть открыт. Далее, при последующем создании объектов этого типа, открытие новых файлов для протоколирования состояний не выполняется. При удалении не используемых объектов, файл протокола должен быть закрыт только тогда, когда удалены все объекты указанного типа данных. Таким образом, нам необходимо осуществлять подсчет количества существующих объектов определенного типа данных, которые совместно используют ресурс системы.

Для решения таких задач удобно использовать контролируемые типы. Рассмотрим следующий пример спецификации пакета:

```
with  Ada.Text_IO;
with  Ada.Finalization;      use   Ada.Finalization;

package Log is

    type Item   is private;

    procedure Put_To_Log (Self: in out Item; S: in String);

private

    type Item   is new  Limited_Controlled with
        record

            . . .   -- описание полей расширения

        end record;

    procedure Initialize (Self: in out Item);
    procedure Finalize (Self: in out Item);

    The_Log_File: Ada.Text_IO.File_Type;
    The_Counter:  Natural := 0;

end Log;
```

Здесь тип "Item" описывает тип объектов при обработке которых используется общий файл протокола "The_Log_File". Для вывода информации о состоянии объекта типа "Item" в файл протокола "The_Log_File" используется процедура "Put_To_Log". Для подсчета количества существующих в текущий момент объектов типа "Item" используется переменная "The_Counter".

Тело данного пакета может быть описано следующим образом:

```
package body Log is

    procedure Initialize (Self: in out Item) is
```

```
        begin
            The_Counter := The_Counter + 1;

            if  The_Counter = 1   then
                Ada.Text_IO.Open (File => The_Log_File,
                                  Mode => Ada.Text_IO.Append_File,
                                  Name => "log.txt");
            end if;
        end Initialize;

        procedure Finalize (Self: in out Item) is
        begin
            if  The_Counter = 1   then
                Ada.Text_IO.Close (The_Log_File);
            end if;

            The_Counter := The_Counter - 1;
        end Finalize;

        procedure Put_To_Log (Self: in out Item; S: in String) is
        begin

            . . .   -- вывод необходимых данных в файл The_Log_File

        end Put_To_Log;

    end Log;
```

Как видно из примера, открытие файла протокола "The_Log_File", при создании первого объекта типа "Item", и инкремент количества существующих в текущий момент объектов типа "Item" в переменной "The_Counter" выполняется автоматически вызываемой процедурой " Initialize". Декремент количества существующих в текущий момент объектов типа "Item" в переменной "The_Counter" и закрытие файла протокола "The_Log_File" выполняется автоматически вызываемой процедурой "Finalize".

Следует заметить, что при рассмотрении данного примера мы не заостряли внимание на структуре типа "Item" и реализации процедуры "Put_To_Log", которая выводит информацию о состоянии объекта в файл протокола, поскольку в данном случае это не имеет принципиального значения.

15.4 Блокировка ресурса

Каноническим примером использования инициализации (" Initialize") и очистки ("Finalize") совместно со ссылочным дискриминантом может служить следующий пример организации монопольного доступа к общему ресурсу:

```
type  Handle (Resource: access Some_Thing) is
        new Finalization.Limited_Controlled with null record;

procedure Initialize (H: in out Handle) is
begin
    Lock (H.Resource);
end Initialize;

procedure Finalize (H: in out Handle) is
begin
    Unlock (H.Resource);
end Finalize;

    . . .

procedure P (T: access Some_Thing) is
    H:  Handle (T);
begin
    . . . -- монопольная обработка T
```

```
end P;
```

В данном случае, суть идеи заключается в том, что описание "H" внутри процедуры "P" приводит к автоматическому вызову операции " Initialize" которая, в свою очередь, вызывает операцию блокировки ресурса "Lock", после чего, внутри процедуры "P", ресурс используется монопольно. Операция очистки "Finalize", которая вызывает операцию освобождения ресурса "Unlock", будет гарантированно вызвана, вне зависимости от того как произошло завершение работы процедуры "P" (нормальным образом, в результате исключения или в результате абортирования обработки). Подобный прием удобен для гарантированной организации работы парных операций, подобных операциям открытия и закрытия файлов.

15.5 Отладка контролируемых типов. Некоторые рекомендации

Сложность отладки и проверки контролируемых типов заключается в том, что процедуры " Initialize", "Finalize" и "Adjust" вызываются автоматически, без какого-либо явного указания в программе. В таком случае, можно поместить в тела соответствующих процедур " Initialize", "Finalize" и "Adjust" инструкции, которые отображают диагностические сообщения, например, следующим образом:

```
procedure Initialize (Self: in out Controlled_Type) is
begin
    . . .   -- код инициализации

    Ada.Text_IO.Put_Line ("Initialize called for Controlled_Type");
end Initialize;

procedure Finalize (Self: in out Controlled_Type) is
begin
    . . .   -- код очистки

    Ada.Text_IO.Put_Line ("Finalize called for Controlled_Type");
end Finalize;

procedure Adjust (Self: in out Controlled_Type) is
begin
    . . .   -- код подгонки

    Ada.Text_IO.Put_Line ("Adjust called for Controlled_Type");
end Adjust;
```

Не смотря на простоту данного подхода, он является достаточно эффективным способом проверки корректности выполняемых действий.

Перечислим также некоторые рекомендации которые помогут избежать некоторых ошибок при написании процедур " Initialize", "Finalize" и "Adjust":

- Первой инструкцией подпрограмм "Finalize" и "Adjust" должна быть проверка "**if**", которая проверяет, что объект не "**null**". Следует остерегаться любого декларативного кода, который может быть выполнен до проверки на "**null**".
- Подпрограммы "Finalize" и "Adjust" должны быть симметричны и инверсны по отношению друг к другу.
- Если контролируемый тип является производным от контролируемого родителя, то процедура " Initialize" производного типа всегда должна вызывать реализацию " Initialize" родителя перед выполнением инициализации необходимой для части расширения.
- Если контролируемый тип является производным от контролируемого родителя, то "Finalize" и "Adjust" производного типа должны всегда вызывать реализацию "Finalize" и "Adjust" родителя после выполнения действий, необходимых для части расширения.
- При тестировании подпрограммы "Finalize", необходимо проверить, что значение объекта после очистки действительно будет "**null**".
- Для агрегата или вызова функции, реализация конкретного компилятора может создавать, а может и не создавать отдельный анонимный объект. Следовательно, подпрограммы "Finalize" и "Adjust" должны поддерживать создание временных анонимных объектов (следует остерегаться любых ограничений на число существующих объектов).

- Следует помнить, что при программировании контролируемых типов, любое присваивание, описание константы или динамическое размещение которое использует инициализационный агрегат, в результате, может привести к вызову "Finalize" и/или "Adjust".

  В частности, не следует выполнять подобных операций при реализации процедур "Finalize" и "Adjust" (это может привести к бесконечно рекурсивным вызовам).

Глава 16

Многозадачность

Встроенная поддержка многозадачности является уникальной и широко известной особенностью языка программирования Ада, которая выгодно отличает его от большинства современных языков программирования. Следует особо подчеркнуть, что поддержка многозадачности обеспечивается не с помощью каких-либо расширений или внешних библиотек, а с помощью строго стандартизированных средств, которые встроены непосредственно в язык программирования.

Обеспечение поддержки многозадачности на уровне языка программирования имеет важное значение. Благодаря этому достигается стабильность семантики, которая избавляет разработчика программного обеспечения как от необходимости использования разнородных внешних библиотек, так и от необходимости разработки своих собственных решений для обеспечения поддержки многозадачности. В результате, облегчается общее понимание исходных текстов многозадачных программ и обеспечивается лучшая переносимость программного обеспечения.

В качестве стандартных средств поддержки многозадачности Ады используются задачи (*tasks*), которые хорошо известны со времен стандарта Ada83 и описывают последовательности действий, которые способны выполняться одновременно, а также объекты защищенных типов (*protected types*), которые являются нововведением стандарта Ada95.

Программа на языке Ада состоит как минимум из одной, а возможно и множества задач, выполняющихся одновременно. Каждая задача выполняется независимо от остальных задач. Механизмы межзадачного обмена данными и синхронизации основаны на высокоуровневых концепцях рандеву и использовании защищенных объектов. Следует заметить, что рандеву и защищенные объекты обладают более высоким уровнем абстракции по сравнению с семафорами. Они предоставляют средства защитной блокировки и таймаутов, а также средства для выполнения выборочного перестроения очередей клиентов и аварийного завершения.

"Гибель" одной задачи не обязательно оказывает влияние на выполнение других задач, за исключением случаев, когда они пытаются с ней взаимодействовать. Специализированные задачи отсутствуют. Возможны ситуации, когда основная процедура Ада-программы может завершиться, а остальные задачи продолжат свое выполнение.

16.1 Задачи

16.1.1 Типы и объекты задач

Конструкция задачи обладает свойствами, которые характерны для пакетов, процедур и структур данных:

- Подобно пакетам, задача имеет спецификацию и тело, однако она не может быть самостоятельно компилируемой единицей, помещенной в свой собственный файл с исходным текстом. Вместо этого, задача должна быть помещена в другую структуру (например, пакет или процедуру)
- Подобно процедуре, задача содержит раздел описаний и исполнительную часть, однако она не вызывается как процедура. Вместо этого, она начинает выполняться автоматически, как часть структуры, в которой она описана.
- Подобно структуре данных, задача имеет тип и может существовать как переменная этого типа. Кроме того, подобно записи, задача может иметь дискриминанты.

Спецификация задачи, начинающаяся зарезервированными словами "**task type**", определяет тип задачи (или задачный тип). Значение объекта (переменная) типа задачи представляет собой задачу. Спецификация задачи, не содержащаяя зарезервированного слова "**type**", определяет одиночную задачу, а описание задачи с такой спецификацией равносильно описанию анонимного типа задачи с одновременным описанием объекта этого типа.

Простым примером многозадачной программы может служить следующая программа:

```
with  Ada.Text_IO;

procedure Multitasking_Demo is

    -- спецификация анонимной задачи
    task Anonimous_Task;

    -- тело анонимной задачи
    task body Anonimous_Task is
    begin -- для Anonimous_Task

        for Count in 1..5 loop
            Ada.Text_IO.Put_Line ("Hello from Anonimous_Task");
        end loop;

    end Anonimous_Task;



    -- спецификация типа задачи
    task type Simple_Task (Message: Character);

    -- тип задачи имеет тело
    task body Simple_Task is
    begin -- для Simple_Task

        for Count in 1..5 loop
            Ada.Text_IO.Put_Line ("Hello from Simple_Task " & Message);
        end loop;

    end Simple_Task;

    -- переменная задачного типа
    Simple_Task_Variable: Simple_Task (Message => 'A');

begin -- для Multitasking_Demo

    -- в отличие от процедур, задачи не вызываются,
    -- а активируются автоматически

    -- выполнение обоих задач начинается как только
    -- управление достигнет этой точки, то есть, сразу
    -- после "begin", но перед выполнением первой инструкции
    -- головной процедуры Multitasking_Demo

    null;

end Multitasking_Demo;
```

В данном примере описана одиночная задача анонимного типа "Anonimous_Task", тип задачи "Simple_Task" и переменная задачи "Simple_Task_Variable", имеющая тип "Simple_Task". Примечательно, что описание типа задачи "Simple_Task" содержит дискриминант, значение которого используется как параметр задачи и указывается при описании переменной задачи "Simple_Task_Variable". Следует также обратить внимание на то, что любой тип задачи является лимитированным и, таким образом, для него не существует предопределенных операций присваивания или сравнения. Алгоритмы работы обоих задач просты и подобны — каждая задача выводит пять приветственных сообщений и завершает свою работу.

Рассмотрим еще один простой пример, который демонстрирует использование задачного типа для описания более одного экземпляра объекта задачи.

```
with  Ada.Text_IO;

procedure Multitasking_Demo_2 is

    -- спецификация типа задачи
    task type Simple_Task (Message: Character; How_Many: Positive);

    -- тело типа задачи
    task body Simple_Task is
    begin -- для Simple_Task

        for Count in 1..How_Many loop
            Ada.Text_IO.Put_Line ("Hello from Simple_Task " & Message);
            delay 0.1;
        end loop;

    end Simple_Task;

    -- переменные задачного типа
    Simple_Task_A: Simple_Task (Message => 'A', How_Many => 5);
    Simple_Task_B: Simple_Task (Message => 'B', How_Many => 10);

begin -- для Multitasking_Demo_2

    null;

end Multitasking_Demo_2;
```

В этом примере, тип задачи "Simple_Task" содержит два дискриминанта: дискриминант "Message" типа "Character" используется при выдаче приветственного сообщения, как и ранее, а дискриминант "How_Many" используется для указания количества выдаваемых сообщений. Таким образом, переменная задачи "Simple_Task_A" выдаст пять приветствий, а переменная "Simple_Task_B" — десять.

16.1.2 Инструкции задержки выполнения (*delay statements*)

При внимательном рассмотрении исходного текста последнего примера можно обнаружить, что после инструкции вывода сообщения приветствия располагается инструкция "**delay** 0.1;", с помощью которой в данном случае осуществляется задержка выполнения задачи на 0.1 секунды.

Инструкции задержки выполнения могут быть использованы для приостановки выполнения тела задачи (или программы) на некоторое время. Рзличают два вида инструкций задержки выполнения: относительная задержка выполнения и абсолютная задержка выполнения.

Общий вид инструкции относительной задержки выполнения следующий:

```
delay время_задержки;
```

Здесь результат выражения *время_задержки*, имеющего предопределенный вещественный тип с фиксированной точкой "Duration" (описан в пакете *Standard*), указывает длительность задержки выполнения в секундах, на которую необходимо задержать выполнения задачи (или программы). При этом отсчет времени задержки выполняется относительно текущего момента времени. Наглядным примером использования инструкции относительной задержки, для задержки выполнения задачи на одну секунду, может служить следующее:

```
delay 1.0;
```

Общий вид инструкции абсолютной задержки выполнения следующий:

```
delay until время_задержки;
```

В этом случае результат выражения *время_задержки*, любого неограниченного типа, указывает момент времени, до наступления которого необходимо задержать выполнение задачи (или программы). Следующий пример демонстрирует использование инструкции абсолютной задержки выполнения задачи:

```
delay until Time_Of (2010, 1, 1, 0.0);  -- задержка выполнения до 1 января 2010 года
```

Следует также заметить, что описание предопределенного типа времени "Time" и ассоциируемых с ним операций предусматривается в стандартном пакете *Ada.Calendar*. Кроме того, пакет *Ada.Calendar* предоставляет функцию "Clock", которая возвращает значение текущего момента времени (для получения более полной информации следует обратиться к спецификации этого пакета).

Отметим также, что при построении систем, которые должны работать в реальном масштабе времени, вместо типа времени "Time", описанного в стандартном пакете *Ada.Calendar*, следует использовать тип времени "Time", который описывается в пакете *Ada.Real_Time* (для получения более полной информации следует обратиться к спецификации пакета *Ada.Real_Time*).

16.1.3 Динамическое создание объектов задач

Ада предоставляет возможность динамически порождать объекты задач с помощью "**new**", что может оказаться полезным, когда необходимо множество объектов задач одного типа. Предположим, что у нас есть следующая спецификация типа задачи:

```
task type Counting_Task is
    entry Add           (Amount : in     Integer);
    entry Current_Total (Total  :    out Integer);
end Counting_Task;
```

Мы можем описать ссылочный тип "Counting_Task_Ptr", который позволяет ссылаться на объекты задач типа "Counting_Task". Затем мы можем описывать переменные ссылочного типа "Counting_Task_Ptr" и с помощью "**new**" создавать динамические объекты задач, как показано в следующем примере:

```
declare
    . . .
    type Counting_Task_Ptr is access Counting_Task;

    Task_Ptr :  Counting_Task_Ptr;   -- переменная ссылающаяся на объект задачи
    . . .
begin
    . . .
    Task_Ptr := new Counting_Task;   -- динамическое создание объекта задачи
    . . .
end;
```

Подобным образом можно динамически создать любое необходимое количество объектов задач. Поскольку реализация Ада-системы не обязана освобождать память, занятую при динамическом создании объекта задачи, то это необходимо выполнять самостоятельно, используя предопределенную настраиваемую процедуру "Ada.Unchecked_Deallocation".

16.1.4 Принудительное завершение `abort`

Ада позволяет принудительно завершать выполнения объекта задачи. Это может быть выполнено с помощью инструкции прекращения, которая может иметь следующий вид:

```
abort Some_Task_Name;
```

Здесь "Some_Task_Name" — это имя какого-либо объекта задачи. Считается, что принудительно прекращенная задача находится в "ненормальном" (*abnormal*) состоянии и не может взаимодействовать с другими задачами. После того, как состояние задачи отмечено как "ненормальное", выполнение ее тела прекращается. Это подразумевает, что прекращается выполнение любых инструкций, расположенных в теле задачи, за исключением тех, которые вызывают операции, отложенные до принудительного прекращения (*abort-deffered operations*).

Следует заметить, что использование принудительного прекращения выполнения задачи является "аварийным" действием, и должно применяться только в тех случаях, когда принудительное прекращение выполнения задачи действительно необходимо (например, когда задача "зависла").

Следует понимать, что использование такого "сильнодействующего" средства остановки для задач, которые выполняют сохранение каких-либо дисковых данных, может повлечь за собой не только потерю самих данных, но и повреждение логической структуры дискового накопителя.

16.1.5 Приоритеты задач

Каждая задача Ады может обладать своим собственным приоритетом выполнения, который задается с помощью директивы компилятора "Priority":

```
pragma Priority ( expression );
```

Непосредственное использование этой директивы компилятора допускается:

- внутри спецификации задачи,
- внутри спецификации защищенного типа или объекта,
- в описательной части тела подпрограммы

Значение результата выражения *expression*, используемого для непосредственного указания приоритета, должно принадлежать целочисленному типу "**Integer**", причем при указании директивы "Priority" в описательной части тела подпрограммы выражение *expression* должно быть статическим, а его значение должно принадлежать диапазону значений подтипа "Priority", который описан в пакете *System*. Например:

```
task Some_Task is
    pragma Priority (5);
    . . .
end Some_Task;
```

Приоритет выполнения задачи определяет ее привилегии на обладание системными ресурсами (например, процессором). В простейшем случае, если две задачи с разными приоритетами готовы к выполнению, то к выполнению будет допущена та задача, приоритет которой выше. При равенстве приоритетов порядок выполнения задач не определен.

В Ada83 приоритет задачи строго фиксировался при ее описании. Согласно стандарту Ada95, приоритет задачи может быть изменен в процессе существования задачи (кроме приоритета, указанного для подпрограммы), то есть задачи могут иметь динамически изменяемые приоритеты.

Средства динамического определения и изменения текущего приоритета задачи предоставляются предопределенным стандартным пакетом *Ada.Dynamic_Priorities*. Спецификация этого пакета проста и имеет следующий вид:

```
with System;
with Ada.Task_Identification; -- спецификация этого пакета обсуждается в 15.2.7,
                              -- при рассмотрении вопроса идентификации задач
package Ada.Dynamic_Priorities is

    procedure Set_Priority
        (Priority : in System.Any_Priority;
         T        : in Ada.Task_Identification.Task_ID :=
             Ada.Task_Identification.Current_Task);

    function Get_Priority
        (T : Ada.Task_Identification.Task_ID :=
             Ada.Task_Identification.Current_Task)
        return System.Any_Priority;

end Ada.Dynamic_Priorities;
```

Следует заметить, что правила планирования выполнения задач на основе статических и динамических приоритетов рассматриваются в приложении *D* (*Annex D*) стандарта Ada95, в котором указываются требования для систем реального времени.

16.2 Взаимодействие задач

Показанные ранее примеры многозадачных программ демонстрировали только способность задач выполняться одновременно. При этом организация взаимодействия различных задач между собой не рассматривалась. В реальности обеспечение такого взаимодействия между одновременно выполняющимися задачами с целью обмена данными или взаимной синхронизации работы имеет важное значение.

16.2.1 Концепция рандеву

Как один из механизмов обеспечения надежного межзадачного обмена данными и взаимной синхронизации работы задач, Ада предоставляет механизм рандеву. Основополагающая идея механизма рандеву достаточно проста. В спецификации задачи публикуются различные входы (*entry*) в задачу, в которых она готова ожидать обращения к ней от других задач. Далее, в теле задачи указываются инструкции принятия обращений к соответствующим входам, указанным в спецификации этой задачи.

Необходимо обратить внимание на несимметричность такого механизма взаимодействия. Это означает, что в процессе взаимодействия одна из задач рассматривается как сервер, а вторая — как клиент, причем задача-сервер не может быть инициатором начала взаимодействия.

В простейшем случае, когда рассматривается взаимодействие только двух задач, задача-клиент, желающая обратиться к другой задаче (задаче-серверу), инициирует обращение к входу задачи-сервера. После этого задача-сервер откликается на обращение задачи-клиента, принимая обращение к этому входу. Таким образом, взаимодействие двух задач осуществляется в ситуации, когда задача-клиент обратилась к входу, а задача-сервер готова принять это обращение. Этот способ взаимодействия двух задач называется рандеву.

Поскольку задача-клиент и задача-сервер выполняются независимо друг от друга, то нет никакой гарантии, что обе задачи окажутся в точке осуществления рандеву одновременно. Поэтому, если задача-сервер оказалась в точке рандеву, но при этом нет ни одного обращения к входу (запроса на взаимодействие), то она должна ждать появления такого обращения. Аналогично, если задача-клиент обращается к входу, а задача-сервер не готова обслужить такое обращение, то задача-клиент должна ждать, пока задача-сервер обслужит это обращение. В процессе ожидания как задача-клиент, так и задача-сервер не занимают ресурсы процессора, находясь в состоянии, которое называют приостановленным или состоянием блокировки.

В случаях, когда вызовы к входу задачи-сервера осуществляют сразу несколько задач-клиентов, эти вызовы ставятся в очередь. Порядок обслуживания такой очереди зависит от соответствия конкретной реализации Ада-системы требованиям приложения *D* (*Annex D*) стандарта Ada95, в котором указываются требования для систем реального времени. Если реализация Ада-системы не обеспечивает соответствия этим требованиям, то очередь обслуживается в порядке поступления вызовов (*FIFO — First-In-First-Out*).

16.2.2 Описание входов

Чтобы некоторая задача-клиент могла инициировать рандеву с задачей-сервером, описание спецификации задачи-сервера должно содержать описание соответствующего входа. Следует заметить, что описания входов могут быть помещены только в описание спецификации задачи, кроме того, задача может иметь приватные входы. Таким образом, описание входа в спецификации задачи-сервера может рассматриваться как декларация сервиса, предоставляемого задачей-сервером для задач-клиентов.

Для описания входов задачи-сервера используется зарезервированное слово "**entry**", а семантика входов задач очень похожа на семантику процедур:

- Так же, как и процедуры, входы задач имеют имена и могут иметь различные параметры. Для имен входов допускается совмещение имен, что подразумевает наличие у одной задачи нескольких входов с одинаковыми именами, но различными параметрами.
- Параметры входов задачи, так же, как и параметры процедур, могут использоваться в режимах "**in**", "**in out**" и "**out**", и могут иметь значения по умолчанию.

При описании параметров входа задачи-сервера следует учитывать, что, в отличие от процедур, для входов задач не допускаются ссылочные параметры, хотя допускаются параметры ссылочного типа. Кроме того, при описании входа может быть опционально указан дискретный тип, который будет использоваться для целого семейства входов в качестве типа индекса, значения которого применяются для определения индивидуальных входов в семействе. Рассмотрим примеры следующих описаний:

```
task Anonimous_Task is
   entry Start;
end Anonimous_Task;


type Level is (Low, Middle, Hight);
```

```
task type Simple_Task is
    entry Read (Value: out Integer);
    entry Request (Level) (Item: in out Integer);
end  Simple_Task;
```

Здесь описание спецификации объекта задачи "Anonimous_Task" содержит описание единственного входа "Start", который не имеет ни одного параметра.

Спецификация типа задачи "Simple_Task" содержит описания двух входов. Вход "**Read**" имеет один "**out**"-параметр "Value" типа "**Integer**". Описание входа "Request" имеет один "**in out**"-параметр "Item" типа "**Integer**" и использует указание дискретного типа индекса (перечислимый тип "Level") представляя, таким образом, целое семейство входов.

16.2.3 Простое принятие обращений к входам

Для организации взаимодействия задач, кроме описаний входов, в спецификации задачи-сервера, тело задачи-сервера должно содержать инструкции принятия рандеву, которые описывают действия, выполняемые задачей-сервером в случае вызова рандеву соответствующими входами задачи-сервера.

Для описания инструкций принятия рандеву используется зарезервированное слово "**accept**", при этом следует учесть, что инструкции принятия рандеву не могут располагаться в подпрограммах, вызываемых в теле задачи, и не могут быть вложены в другие инструкции принятия рандеву этого же тела задачи. Таким образом, тело задачи может содержать инструкции принятия рандеву только для тех входов, которые указываются в спецификации задачи.

Рассмотрим схематические тела задач, демонстрирующие примеры описания инструкций принятия рандеву (тела задач соответствуют спецификациям задач, которые были показаны при обсуждении описания входов):

```
task body  Anonimous_Task  is
begin
    accept Start;
    . . .
end Anonimous_Task;


task body Simple_Task is
begin
    . . .
    accept Read (Value: out Integer) do
        Value := Some_Value;
    end Read;
    . . .
    accept Request (Low) (Item: in out Integer) do
        Some_Low_Item := Item;
        . . .
        Item := Some_Low_Item;
    end Request;

    accept Request (Middle) (Item: in out Integer) do
        Some_Middle_Item := Item;
        . . .
        Item := Some_Middle_Item;
    end Request;

    accept Request (Hight) (Item: in out Integer) do
        Some_Hight_Item := Item;
        . . .
        Item := Some_Hight_Item;
    end Request;
    . . .
end  Simple_Task;
```

Тело задачи "Anonimous_Task" содержит единственную инструкцию принятия рандеву, которая соответствует входу "Start", у которого нет параметров. Данная инструкция принятия очень проста и не содержит никакой последовательности инструкций, выполняемой в процессе принятия рандеву.

Следует заметить, что инструкция принятия "**accept**" может не содержать в себе последовательность инструкций, даже когда соответствующий вход имеет параметры, и наоборот, она может содержать последовательность каких-либо инструкций, если параметры у соответствующего входа отсутствуют. Кроме того, следует заметить, что последовательность инструкций, вложенная в инструкцию принятия рандеву, может содержать инструкцию выхода "**return**" (это подобно использованию "**return**" в теле процедуры). Также следует учесть, что формальные параметры, указанные в инструкции принятия рандеву для соответствующего входа, будут локальными для этой инструкции принятия.

Тело задачи "Simple_Task" более интересно. Первая инструкция принятия рандеву соответствует входу "**Read**" с "**out**"-параметром "Value". Внутри этой инструкции принятия рандеву параметру "Value" присваивается значение переменной "Some_Value" (предполагается, что эта и другие переменные были где-либо описаны).

Далее следуют три инструкции принятия рандеву, образующие "семейство". Все они соответствуют описанию входа "Request", в спецификации задачи "Simple_Task", которая описывалась с указанием типа "Level", значения которого ("Low", "Middle" и "Hight") используются в качестве индекса.

16.2.4 Простой вызов входа

Задача-клиент осуществляет вызов входа задачи-сервера, идентифицируя как объект задачи-сервера, так и необходимый вход задачи-сервера. Для демонстрации описаний простых вызовов входов задачи-сервера, заданных в задаче-клиенте, рассмотрим следующий пример:

```
declare
    . . .
    Simple_Task_Variable : Simple_Task;
    . . .
    Read_Value    : Integer;
    Request_Item  : Integer;
    . . .
begin
    . . .
    Anonimous_Task.Start;
    . . .
    Simple_Task_Variable.Read (Read_Value);
    Simple_Task_Variable.Request (Middle) (Request_Item);
    . . .
end;
```

Как видно из этого примера, простые вызовы входов очень похожи на вызовы процедур. Вызов входа может иметь параметры, значения которых могут передаваться в обоих направлениях между задачей-клиентом и задачей-сервером.

При непосредственной обработке рандеву происходит передача любых параметров "**in**" и "**in out**" в задачу-сервер. Затем задача-сервер выполняет последовательность инструкций, которая расположена внутри инструкции принятия (если такая последовательность существует), а задача-клиент остается в приостановленном состоянии. Далее, после завершения этой последовательности инструкций, происходит передача любых параметров "**out**" и "**in out**" к задаче-клиенту, и обработка рандеву завершается. После этого обе задачи продолжают свое выполнение независимо друг от друга.

Рассмотрим простой пример программы, которая использует свойства рандеву для своеобразного управления последовательностью запуска задач. Следует заметить, что стандарт Ады никак не определяет последовательность запуска множества задач одной и той же программы, и язык не предоставляет стандартных средств для непосредственного управления процессом активации множества задач. Следовательно, разные реализации Ада-систем могут использовать различные правила для определения последовательности активации множества задач, а это означает, что последовательность активации множества задач одной и той же программы может отличаться в различных реализациях Ада-системы. Однако, используя свойства рандеву, можно программно осуществлять взамную синхронизацию выполнения задач. Таким образом, выполнение тела задачи может быть приостановлено в самом начале задачи, то есть до того, как задача начнет выполнять свою непосредственную работу. Такой способ использования рандеву демонстрируется в следующем примере программы:

```
with  Ada.Text_IO;
```

```
procedure Multitasking_Demo_3 is

    -- спецификация типа задачи
    task type Simple_Task (Message: Character; How_Many: Positive) is

        entry Start; -- этот вход будет использоваться для реального
                     -- запуска задачи на выполнение

    end Simple_Task;


    -- тело задачи
    task body Simple_Task is
    begin -- для Simple_Task

        accept Start;  -- в этом месте, выполнение задачи будет заблокировано
                       -- до поступления вызова входа

        for Count in 1..How_Many loop
            Ada.Text_IO.Put_Line ("Hello from Simple_Task " & Message);
            delay 0.1;
        end loop;

    end Simple_Task;


    -- переменные задачного типа
    Simple_Task_A: Simple_Task (Message => 'A', How_Many => 5);
    Simple_Task_B: Simple_Task (Message => 'B', How_Many => 3);

begin -- для Multitasking_Demo_3

    -- в момент, когда управление достигнет этой точки,
    -- все задачи начнут свое выполнение,
    -- но будут приостановлены в инструкциях принятия рандеву

    Simple_Task_B.Start;
    Simple_Task_A.Start;

end Multitasking_Demo_3;
```

Как видно из исходного текста этого примера, здесь описаны два объекта задач: "Simple_Task_A" и "Simple_Task_B". Каждая задача имеет вход "Start", который используется для управления очередностью запуска задач. В результате сначала запускается задача "Simple_Task_B", а затем "Simple_Task_A".

Рассмотрим еще один простой пример программы, в котором демонстрируется использование входов с параметрами.

```
procedure Demo is

    X : Duration := Duration (Random (100));
    Y : Duration;

    task Single_Entry is
        entry Handshake (Me_Wait : in Duration; You_Wait : out Duration);
    end task;

    task body Single_Entry is
        A : Duration := Duration (Random (100));
        B : Duration;
    begin
        delay A;

        accept Handshake (Me_Wait : in Duration; You_Wait : out Duration) do
            B := Me_Wait;
```

```
            You_Wait := A;
        end Handshake;

        delay B;
    end;

begin
    delay (X);
    Handshake (X, Y);
    delay (Y);
end Demo;
```

В этом примере, две задачи обмениваются значением длительности задержки выполнения, используя для этого рандеву на входе "Handshake" задачи "Single_Entry".

Следует заметить, что возможна ситуация, когда задача ожидает рандеву, но при этом нет ни одной задачи, с которой она может его осуществить. В этом случае задача будет аварийно завершена с исключением *Tasking_Error*.

16.2.5 Селекция принятия рандеву

Предположим, что нам необходима задача-сервер, которая будет обслуживать множество задач-клиентов. Предположим также, что задача-сервер должна иметь не один, а целое множество входов для предоставления различных сервисов задачам-клиентам, а один из входов будет указывать на необходимость завершения работы задачи-сервера, то есть задача-сервер должна выполняться до тех пор, пока не получит явной команды на завершение своей работы.

Вполне резонно на основе полученных ранее сведений, попытаться реализовать что-нибудь подобное следующему:

```
task Server_Task is
    entry Service_1 [ параметры для Service_1 ] ;
    entry Service_2 [ параметры для Service_2 ] ;
    . . .
    entry Service_N [ параметры для Service_N ] ;

    entry Stop;
end task;

task body Server_Task is
    . . .
begin
    loop
        accept Service_1 [ параметры для Service_1 ] do
            . . .
        end Service_1;

        accept Service_2 [ параметры для Service_2 ] do
            . . .
        end Service_2;

        . . .

        accept Service_N [ параметры для Service_N ] do
            . . .
        end Service_N;

        accept Stop do
            exit ;      -- выход из цикла, и, следовательно,
                        -- завершение задачи
        end Stop;
    end loop;
end Server_Task;
```

Однако при внимательном рассмотрении логики работы данного примера оказывается, что задача-сервер будет блокироваться (приостанавливаться) в каждой инструкции принятия рандеву, ожидая поступления вызова на соответствующем входе от какой-либо задачи-клиента. Причем, находясь в состоянии ожидания, задача-сервер никак не будет реагировать на наличие и поступление вызовов от задач-клиентов на других входах. Следовательно, для ситуаций, которые подобны показанной в этом примере, необходима возможность селекции (или выбора) принятия рандеву на входах задачи-сервера.

Для обеспечения селекции принятия рандеву, Ада предоставляет различные варианты инструкции отбора с ожиданием, которая задается с помощью зарезервированного слова "**select**". Использование инструкции отбора в теле задачи-сервера позволяет:

- одновременно ожидать более одного рандеву
- выполнять таймаут, если за указанный период времени не поступило ни одного вызова рандеву
- осуществлять рандеву только в случае наличия вызова рандеву
- завершать выполнение задачи при отсутствии задач-клиентов, которые потенциально могут вызвать рандеву

Рассмотрим следующий пример использования инструкции отбора в теле задачи-сервера:

```
task Server_Task is
    entry Service_1 [ параметры для Service_1 ] ;
    entry Service_2 [ параметры для Service_2 ] ;
    . . .
    entry Service_N [ параметры для Service_N ] ;

    entry Stop;
end task;

task body Server_Task is
    . . .
begin
    loop
        select
            accept Service_1 [ параметры для Service_1 ] do
                . . .
            end Service_1;
            . . .       -- дополнительная последовательность инструкций,
                        -- которая выполняется после принятия рандеву
                        -- на входе Service_1
        or
            accept Service_2 [ параметры для Service_2 ] do
                . . .
            end Service_2;
        or
            . . .
        or
            accept Service_N [ параметры для Service_N ] do
                . . .
            end Service_N;
        or
            accept Stop;
            exit ;      -- выход из цикла, и, следовательно,
                        -- завершение задачи
        end select
    end loop;
end Server_Task;
```

Как видно из исходного текста примера, инструкции принятия рандеву указываются как альтернативы выбора инструкции отбора (это несколько подобно инструкции "**case**").

В данном случае при выполнении инструкции отбора задача-сервер циклически "опрашивает" свои входы на наличие вызова рандеву от задач-клиентов (без блокировки в состоянии ожидания вызова рандеву на каком-либо входе). Опрос продолжается до тех пор, пока не будет обнаружен вызов рандеву на каком-либо входе,

который соответствует одной из перечисленных инструкций принятия рандеву. После обнаружения вызова выполняется соответствующая альтернатива (примечание: завершение обработки альтернативы приводит к завершению инструкции отбора).

Следует обратить внимание, что если в процессе опроса, выполняемого инструкцией отбора, одновременно появятся два и более вызова рандеву, то инструкция отбора выберет для обработки только один из поступивших вызовов, причем правила выбора альтернативы для такого случая не определяются стандартом. Другими словами, когда инструкция отбора помещена в тело цикла, при одновременном появлении двух и более вызовов рандеву, инструкция отбора будет осуществлять обработку только одного вызова рандеву в течение одного "витка" цикла, а одновременно поступившие вызовы рандеву будут поставлены в очередь порядок которой не определяется стандартом.

В данном примере интересную ситуацию представляет альтернатива принятия рандеву на входе "Stop". В этом случае происходит выход из бесконечного цикла выполнения инструкции отбора, что, в свою очередь, приводит к завершению работы задачи-сервера. Если в процессе завершения работы задачи-сервера поступит вызов рандеву на какой-либо из входов "Service_1" — "Service_N", то задача-клиент, вызывающая рандеву, скорее всего получит исключение *Tasking_Error*. Недостаток такого подхода состоит в том, что завершение работы задачи-сервера требует явного указания вызова рандеву на входе "Stop".

Чтобы избавиться от необходимости явного указания вызова рандеву на входе "Stop", можно использовать инструкцию отбора, в которой указывается альтернатива завершения задачи:

```
task Server_Task is
    entry Service_1 [ параметры для Service_1 ] ;
    . . .
    entry Service_N [ параметры для Service_N ] ;
end task;

task body Server_Task is
    . . .
begin
    loop
        select
            accept Service_1 [ параметры для Service_1 ] do
                . . .
            end Service_1;
        or
            . . .
        or
            accept Service_N [ параметры для Service_N ] do
                . . .
            end Service_N;
        or
            terminate;    -- завершение работы задачи
        end select
    end loop;
end Server_Task;
```

В таком случае альтернатива завершения задачи будет завершать работу задачи-сервера, когда отсутствуют вызовы рандеву и отсутствуют задачи-клиенты, которые потенциально способны вызвать рандеву с задачей-сервером. Таким образом, для завершения работы задачи-сервера не требуется явного выполнения никаких специальных действий. Следует также учитывать, что альтернатива завершения работы задачи должна указываться последней в списке альтернатив инструкции отбора.

Может возникнуть ситуация, когда необходимо, чтобы в процессе ожидания вызова рандеву от задач-клиентов задача-сервер выполняла какие-либо дополнительные действия. Для этого можно использовать инструкцию отбора, в которой вместо альтернативы завершения работы используется раздел "**else**". Использование такого варианта инструкции отбора может иметь следующий вид:

```
loop
    select
        accept Service_1 [ параметры для Service_1 ] do
            . . .
        end Service_1;
    or
```

```
        . . .
    or
        accept Service_N [ параметры для Service_N ] do
            . . .
        end Service_N;
    else
        . . .       -- последовательность инструкций, которая выполняется
                    -- если нет ни одного вызова рандеву
    end select
end loop;
```

Подобным образом последовательность инструкций, указанная в разделе "**else**", может быть использована для организации "фоновой" работы задачи-сервера.

Еще одной разновидностью инструкции отбора служит инструкция отбора, использующая альтернативу таймаута (или задержки). Такая инструкция отбора может иметь следующий вид:

```
loop
    select
        accept Service_1 [ параметры для Service_1 ] do
            . . .
        end Service_1;
    or
        . . .
    or
        accept Service_N [ параметры для Service_N ] do
            . . .
        end Service_N;
    or
        delay 1.0;
        . . .       -- последовательность инструкций таймаута,
                    -- которая выполняется в случае
                    -- когда нет ни одного вызова рандеву
                    -- в течение одной секунды
    end select
end loop;
```

Подход, демонстрируемый этим примером, удобно использовать для организации "сторожевого таймера", сигнализирующего о некорректной работе программного обеспечения.

Альтернатива таймаута, указанная в инструкций отбора, позволяет задаче-серверу выполнять определенную последовательность действий, если в течение указанного интервала времени не поступило ни одного вызова рандеву. Интервал времени указывается в альтернативах таймаута аналогично интервалу времени в инструкциях задержки выполнения (необходимо однако заметить, что несмотря на внешнее подобие, не следует путать альтернативы таймаута инструкций отбора с инструкциями задержки выполнения). Таким образом, может быть указан относительный (как в примере выше) или абсолютный интервал времени. Если выражение, указывающее относительный интервал времени, имеет отрицательное или нулевое значение или значение абсолютного интервала времени оценивается как прошедшее, то альтернатива таймаута инструкции отбора может быть расценена как эквивалент раздела "**else**".

В одной инструкции отбора, допускается наличие более одной альтернативы таймаута. При этом, будет обрабатываться та альтернатива таймаута, которая имеет наименьший интервал времени. Отметим также, что в пределах одной инструкции отбора не допускается совместное использование различных альтернатив таймаута для которых одновременно заданы относительные и абсолютные интервалы времени.

При применении инструкций отбора следует учитывать, что использование альтернативы завершения работы задачи ("**terminate**"), альтернативы таймаута ("**delay**") и раздела "**else**" в пределах одной инструкции отбора является взаимно исключающим. Кроме того, любой вариант инструкции отбора обязан содержать хотя бы одну альтернативу выбора, в которой указана инструкция принятия рандеву "**accept**".

Еще одной особенностью инструкций отбора является опциональная возможность указания дополнительной проверки какого-либо условия для альтернатив принятия рандеву, завершения работы задачи и таймаута. Для указания такой проверки используется конструкция вида:

```
when условие => ...
```

где проверяемое *условие* описывается с помощью логического выражения, результат вычисления которого должен иметь значение предопределенного логического типа "Standard.**Boolean**". Как правило, такую проверку называют защитной или охранной (*guard*), а ее использование может иметь следующий вид:

```
declare
    . . .
    Service_1_Counter : Integer;
    . . .
    Service_N_Counter : Integer;
    . . .
begin
    . . .
    loop
        . . .
        select
            when (Service_1_Counter > 0) =>
                accept Service_1 [ параметры для Service_1 ] do
                    . . .
                end Service_1;
        or
            . . .
        or
            when (Service_N_Counter > 100) =>
                accept Service_N [ параметры для Service_N ] do
                    . . .
                end Service_N;
        end select
    end loop;
    . . .
end;
```

Вычисление значения логического выражения осуществляется в начале выполнения инструкции отбора. Если результат вычисления выражения "**True**", то соответствующая альтернатива имеет право быть выбранной инструкцией отбора, если результат "**False**", альтернатива выбрана не будет, причем даже в том случае, когда какая-либо задача-клиент осуществила вызов соответствующего входа и ждет обслуживания. Следует также учесть, что в случае, когда проверки условий используются для всех альтернатив инструкции отбора, и результаты проверок всех условий имеют значение "**False**", это приведет к возбуждению исключения *Program_Error*.

Альтернативу отбора можно назвать открытой, когда для нее не указана дополнительная проверка условия или когда значение результата проверки условия есть "**True**". В противном случае альтернативу отбора можно назвать закрытой.

16.2.6 Селекция вызова рандеву

Инструкции отбора "**select**" могут использоваться не только для селекции принятия рандеву в задаче-сервере, но и для селекции вызова рандеву в задаче-клиенте. В подобных случаях различные формы инструкций отбора позволяют задаче-клиенту выполнять условный вызов рандеву, временный вызов рандеву или асинхронную передачу управления.

Рассмотрим простой пример инструкции отбора для выполнения условного вызова рандеву на входе "Service_1" задачи-сервера "Server_Task":

```
select
    Server_Task.Service_1 [ параметры для Service_1 ] ;
else
    Put_Line ("Server_Task is busy!");
end select;
```

В этом случае, если задача-сервер "Server_Task" не готова к немедленному приему рандеву на входе "Service_1", то вызов рандеву будет отменен и выполнится последовательность инструкций в разделе "**else**", которая выводит сообщение "`Server_Task is busy!`".

Инструкция отбора, предназначенная для условного вызова рандеву на входе задачи-сервера, как правило, используется для многократного обращения к задаче-серверу. Таким образом, общий вид ее применения может быть следующм:

```
loop

    select
        Server_Task.Service_1 [ параметры для Service_1 ] ;
        . . .    -- опциональная последовательность инструкций,
        exit;    -- выполняемая после рандеву
    else
        . . .    -- последовательность инструкций,
                 -- выполняемая в случае невозможности
                 -- осуществления рандеву
    end select;

end loop;
```

Инструкция отбора для временного вызова рандеву позволяет задавать ожидание принятия рандеву на входе задачи-сервера в течение заданного интервала времени. Пример такой инструкции для выполнения временного вызова рандеву на входе "Service_1" задачи-сервера "Server_Task" может иметь следующий вид:

```
select
    Server_Task.Service_1 [ параметры для Service_1 ] ;
or
    delay 5.0;
    Put_Line ("Server_Task has been busy for 5 seconds!");
end select;
```

В данном случае, если в течение указанного интервала времени (здесь задан относительный интервал времени длительностью 5 секунд) задача-сервер "Server_Task" не приняла рандеву на входе "Service_1", то вызов рандеву отменяется и выполняется последовательность инструкций, заданная альтернативой задержки, которая в этом примере выдаст сообщение "`Server_Task has been busy for 5 seconds!`".

Вместо относительного интервала может быть указано абсолютное значение времени:

```
select
    Server_Task.Service_1 [ параметры для Service_1 ] ;
or
    delay until Christmas;
    Put_Line ("Server_Task has been busy for ages!");
end select;
```

Следует заметить, что когда, в случае относительного интервала времени, указанная задержка будет иметь нулевое или отрицательное значение, или заданное абсолютное значение времени расценивается как прошедшее, выполнение инструкции будет осуществляться подобно выполнению инструкции отбора для условного вызова рандеву.

Инструкция отбора для временного вызова рандеву удобно использовать в качестве сторожевого таймера. Таким образом, ее общий вид может быть следующм:

```
loop

    select
        Server_Task.Service_1 [ параметры для Service_1 ] ;
        . . .    -- опциональная последовательность инструкций,
        exit;    -- выполняемая после рандеву
    or
        delay интервал_времени

        . . .    -- последовательность инструкций,
                 -- выполняемая в случае невозможности
                 -- осуществления рандеву
    end select;

end loop;
```

Заметим, что вместо конструкции:

delay *интервал_времени*

которая указывает относительный интервал, может быть использована конструкция:

delay until *интервал_времени*

которая указывает абсолютное значение времени.

Инструкция отбора для асинхронной передачи управления (введена стандартом Ada95) позволяет задаче-клиенту продолжить выполнение какого-либо кода в то время, когда вызов рандеву ожидает обслуживания. Пример использования такой инструкции может иметь следующий вид:

```
select
    delay 5.0;
    Put_Line ("Server_Task is not serviced the Service_1 yet");
then abort
    Server_Task.Service_1 [ параметры для Service_1 ] ;
end select;
```

В данном случае начинается выполнение инструкций, расположенных между "**then abort**" и "**end select**", то есть вызов рандеву с задачей-сервером "Server_Task" на входе "Service_1". При этом, если истечение интервала времени из инструкции "**delay**" (или "**delay until**"), расположенной после "**select**", произойдет раньше завершения выполнения "Server_Task.Service_1", то выполнение "Server_Task.Service_1" принудительно прерывается и выполняется вывод сообщения "`Server_Task is not serviced the Service_1 yet`".

Примечательно, что использование асинхронной передачи управления не ограничено многозадачностью, например:

```
select
    delay 5.0;
    Put_Line ("So_Big_Calculation abandoned!");
then abort
    So_Big_Calculation;
end select;
```

В данном случае использование асинхронной передачи позволяет прервать неопределенно долго выполняющуюся или "зависшую" в бесконечном цикле процедуру "So_Big_Calculation".

При селекции вызова рандеву с помощью инструкций отбора (для осуществления условного вызова рандеву, временного вызова рандеву или асинхронной передачи управления) следует помнить, что если вызванная задача уже завершена, то выполнение инструкции отбора приведет к возбуждению исключению *Tasking_Error*.

16.2.7 Идентификация задач и атрибуты

Каждая задача (объект) Ады имеет свой собственный уникальный идентификатор, с помощью которого она может быть идентифицирована в других задачах. Механизм, с помощью которого задача может получить свой уникальный идентификатор, обеспечивается средствами стандартного пакета *Ada.Task_Identification*, описанного в приложении *C* (*Annex C*) стандарта Ada95, в котором указываются требования для системного программирования.

Спецификация пакета *Ada.Task_Identification* имеет следующий вид:

```
package Ada.Task_Identification is

    type Task_ID is private;
    Null_Task_ID : constant Task_ID;

    function  "=" (Left, Right : Task_ID) return Boolean;

    function  Image        (T : Task_ID) return String;
    function  Current_Task               return Task_ID;

    procedure Abort_Task   (T : in out Task_ID);
```

```
    function  Is_Terminated (T : Task_ID) return Boolean;
    function  Is_Callable   (T : Task_ID) return Boolean;

private

    . . .   -- Стандартом языка не определено

end Ada.Task_Identification;
```

Кроме этого пакета, в приложении *C* стандарта описываются атрибуты, которые могут быть использованы для идентификации задач:

T'Identity	где "T": любая задача. Возвращает значение типа "Task_ID" которое уникально идентифицирует "T".
E'Caller	где "E": имя любого входа задачи. Возвращает значение типа "Task_ID", которое идентифицирует обрабатываемую в текущий момент задачу, обратившуюся к входу задачи "E". Использование этого атрибута допустимо только внутри инструкции принятия (для задачи-сервера).

При использовании этих средств следует учитывать, что по истечении некоторого времени ничто не гарантирует активность задачи или ее присутствие в области видимости.

Кроме перечисленных средств идентификации, при работе с задачами могут быть также использованы следующие атрибуты:

T'Callable	Возвращает значение "**True**", если задача "T" может быть вызвана.
T'Terminated	Возвращает "**True**" если выполнение задачи "T" прекращено.

Следует учесть, что "T" — это любая задача а результат, возвращаемый этими атрибутами, имеет предопределенный логический тип "Standard.**Boolean**".

Для входов (и семейств входов) задач определен атрибут "'Count". Если "E" — имя любого входа задачи, то "E'Count" возвращает значение типа "Universal_Integer", показывающее число обращений к входу "E", которые находятся в очереди. Исползование этого атрибута допустимо только внутри тела задачи или внутри программного модуля, который расположен внутри тела задачи.

Следует заметить, что не рекомендуется, чтобы алгоритм обработки имел жесткую зависимость от значений атрибутов задач "'Callable" и "'Terminated", а также атрибута "'Count" для входов (и семейств входов).

16.2.8 Разделяемые (общие) переменные

Ада позволяет задачам совместно использовать разделяемые (общие) переменные (объекты, ресурсы), которые располагаются в общей памяти и могут быть необходимы как для организации взаимодействия задач, так и для хранения каких-либо общих данных. Следовательно, необходимо, чтобы такие переменные допускали возможность чтения/записи из различных задач программы.

Поскольку при выполнении программы возможны ситуации, когда несколько задач пытаются получить доступ к одной и той же разделяемой переменной, во избежание конфликтных ситуаций и поддержки общей корректности работы программы необходима синхронизация доступа к разделяемым переменным. Обеспечение синхронизации доступа к разделяемым переменным представляет собой основополагающую проблему многозадачного (параллельного) программирования, и называется взаимным исключением. Решение этой проблемы основывается на том, что в какой-либо момент времени право на доступ к разделяемой переменной предоставляется только одной задаче. Для обеспечения синхронизации доступа используются специальные программные (или аппаратные) средства: семафоры, критические секции, мониторы и т.д.

В стандарте Ada95 для указания разделяемых переменных используются следующие директивы компилятора:

```
pragma Atomic              ( Local_Name );
pragma Atomic_Components ( Local_Array_Name );

pragma Volatile              ( Local_Name );
pragma Volatile_Components ( Local_Array_Name );
```

Здесь *Local_Name* указывает локальное имя объекта или описание типа, а '*Local_Array_Name* указывает локальное имя объекта-массива или описание типа-массива.

Директивы компилятора "Atomic" и "Atomic_Components" обеспечивают непрерывность операций чтения/записи для всех указанных в них объектах. Такие объекты называют атомарными (*atomic*), а операции над ними выполняются только последовательно.

Директивы компилятора "Volatile" и "Volatile_Components" обеспечивают выполнение операций чтения/записи для всех указанных в них объектах непосредственно в памяти.

Примеры применения этих директив компилятора могут иметь следующий вид:

```
Array_Size  : Positive;
pragma Atomic   (Array_Size);
pragma Volatile (Array_Size);

Store_Array is array (1..Array_Size) of Integer;
pragma Atomic_Components   (Store_Array);
pragma Volatile_Components (Store_Array);
```

Разделяемые переменные и перечисленные для них директивы компилятора могут быть использованы для организации:

- взаимодействия задач
- взаимодействия Ада-программы с другими процессами
- управления устройствами из Ада-программ

Следует заметить, что описанные подобным образом разделяемые переменные не должны использоваться для синхронизации взаимодействия задач, для этого следует использовать средства механизма рандеву и/или защищенные объекты.

16.3 Защищенные модули (*protected units*)

16.3.1 Проблемы механизма рандеву

Несмотря на обилие возможностей, предоставляемых моделью механизма рандеву Ada83, который предусматривает развитый высокоуровневый подход для синхронизации задач, позволяющий избежать многочисленные методологические сложности, вызванные применением низкоуровневых примитивов, подобных семафорам и сигналам, этот механизм обладает некоторыми недостатками.

Представим себе классический пример, когда необходимо обеспечить взаимодействие двух задач, одна из которых выполняет чтение данных из какого-либо устройства, а вторая — осуществляет обработку этих данных. В литературе подобное взаимоотношение задач, как правило, называют "поставщик-потребитель". Как мы уже знаем, механизм рандеву является одновременно как механизмом синхронизации, так и механизмом межзадачного обмена данными. Необходимость передачи данных между двумя задачами, которые обычно выполняются асинхронно, требует приостановки выполнения задач для осуществления "переговоров". Следовательно, для эффективной организации подобного взаимодействия необходима буферизация данных, передаваемых от одной задачи к другой. Обычным решением в этой ситуации является создание промежуточной задачи, которая служит буфером и управляет потоком данных, а значит, всегда доступна для рандеву с обоими задачами, нуждающимися в обмене данными. При этом следует учесть, что рандеву является относительно длительной операцией, и необходимость наличия дополнительной задачи-буфера приводит к неоправданному снижению производительности.

Исходя из таких рассуждений, гораздо предпочтительнее выглядит возможность, чтобы задача-производитель (та задача, которая читает данные из устройства) могла просто оставлять данные в какой-либо переменной

для задачи-приемника (та задача, которая обрабатывает принятые данные), после чего задача-приемник могла бы так же просто эти данные прочитать. Однако, обычная переменная для этой цели не подходит, поскольку необходимо предотвратить модификацию содержимого переменной двумя одновременно выполняющимися задачами.

Еще одним недостатком рандеву является то, что в некоторых ситуациях может возникнуть инверсия абстракции, приводящая к замкнутой взаимной блокировке задач. Кроме того, с методологической точки зрения, идеология механизма рандеву строго ориентирована на управление и находится вне новейших объектно-ориентированных подходов.

Средством, которое позволяет организовать взаимно исключающий доступ к данным из разных, одновременно выполняющихся задач, являются защищенные модули, которые были введены стандартом Ada95. Характерной особенностью защищенных модулей является обеспечение ими синхронизированного доступа к приватным данным, однако в противоположность задачам, которые являются активными сущностями, защищенные модули — пассивны.

16.3.2 Защищенные типы и объекты. Защищенные подпрограммы

Защищенные модули (типы и объекты) Ады инкапсулируют данные и позволяют осуществлять доступ к ним с помощью защищенных подпрограмм или защищенных входов. Стандарт языка гарантирует, что в результате выполнения кода таких подпрограмм и входов изменение содержимого данных будет производиться в режиме взаимного исключения без необходимости создания дополнительной задачи.

Защищенный модуль может быть описан как защищенный тип или как одиночный защищенный объект, что аналогично одиночной задаче. В последнем случае предполагается, что защищенный объект имеет соответствующий анонимный тип. Следует учитывать, что защищенный тип является лимитированным и, тем самым, не обладает предопределенными операциями присваивания или сравнения.

Подобно задаче или пакету, защищенный модуль имеет спецификацию и тело. Спецификация описывает протокол доступа к защищенному модулю (интерфейс), и может содержать спецификации процедур, функций и входов защищенного модуля. Тело описывает детали реализации протокола доступа к данным защищенного модуля и, как правило, содержит тела защищенных подпрограмм и входов. Подобно задачам и записям, защищенный модуль может иметь дискриминанты дискретного или ссылочного типа.

В качестве простой иллюстрации рассмотрим пример следующго одиночного защищенного объекта:

```
-- спецификация защищенного объекта
protected Variable is

    function Read return Item;
    procedure Write (New_Value : Item);

private
    Data : Item;

end Variable;

-- тело защищенного объекта
protected body Variable is

    function Read return Item is
    begin
        return Item;
    end;

    procedure Write (New_Value : Item) is
    begin
        Data := New_Value;
    end;

end Variable;
```

Защищенный объект "Variable" предоставляет управляемый доступ к приватной переменной "Data" типа "Item". Функция "**Read**" позволяет читать, а процедура "**Write**" — обновлять текущее значение переменной "Data".

Защищаемые данные и данные о состоянии объекта должны быть помещены в приватную часть спецификации. Смысл этого заключается в том, что приватная часть не доступна клиенту непосредственно, а наличие всей информации в интерфейсе защищенного объекта необходимо компилятору для эффективного распределения памяти.

Защищенные процедуры предусматривают взаимно исключающий доступ к данным защищенного модуля по чтению и/или записи. Защищенные функции предоставляют одновременный доступ к данным защищенного модуля только по чтению, что подразумевает одновременное выполнение множества вызовов функций. Однако вызовы защищенных процедур и защищенных функций остаются взаимно исключающими. Порядок, в котором разные задачи ожидают выполнения защищенных процедур и защищенных функций, стандартом не определяется. Однако поддержка требований приложения *D* (*Annex D*) стандарта Ada95, в котором указаны требования для систем реального времени, позволяет сделать некоторые предположения о возможном порядке выполнения подпрограмм.

Для обращения к защищенным подпрограммам используется традиционная точечная нотация:

```
X := Variable.Read;
. . .
Variable.Write (New_Value => Y);
```

Внутри тела защищенного объекта допускается несколько подпрограмм, при этом реализация Ада-системы будет гарантированно осуществлять вызовы подпрограмм по принципу взаимного исключения (подобно монитору).

16.3.3 Защищенные входы и барьеры

По аналогии со входами задач, защищенный модуль может иметь защищенные входы. Действия, выполняемые при вызове защищенного входа, предусматриваются в его теле. Защищенные входы подобны защищенным процедурам в том, что они гарантируют взаимно исключающий доступ к данным защищенного модуля по чтению и/или записи. Однако внутри тела защищенного модуля защищенные входы предохраняются логическим выражением, которое называют барьером, а результат вычисления этого логического выражения должен иметь предопределенный логический тип "Standard.**Boolean**".

Если при вызове защищенного входа значение барьера есть "**False**", то выполнение вызывающей задачи приостанавливается до тех пор, пока значение барьера не станет равным "**True**" и внутри защищенного модуля будут отсутствовать активные задачи (задачи, которые выполняют тело какого-либо защищенного входа или какой-либо защищенной подпрограммы). Следовательно, вызов защищенного входа может быть использован для реализации условной синхронизации.

Можно заметить, что существует строгая параллель между инструкцией принятия с охранным условием из тела задачи и телом защищенного входа с барьерным условием для защищенного модуля. При этом затраты времени на проверку барьерного условия для защищенного входа значительно меньше, чем затраты времени на проверку охранного условия инструкции принятия.

Хорошим примером решения упоминавшейся ранее проблемы "поставщик-потребитель"служит реализации циклического буфера с помощью защищенного типа:

```
-- спецификация защищенного типа
protected type Bounded_Buffer is

    entry Put (X: in     Item);
    entry Get (X:    out Item);

private
    A: Item_Array (1 .. Max);
    I, J: Integer range 1 .. Max := 1;
    Count: Integer range 0 .. Max := 0;

end Bounded_Buffer;

-- тело защищенного типа
protected body Bounded_Buffer is

    entry Put (X: in Item) when Count < Max is
```

```
        begin
            A (I) := X;
            I := I mod Max + 1; Count := Count + 1;
        end Put;

        entry Get (X: out Item) when Count > 0 is
        begin
            X := A (J);
            J := J mod Max + 1; Count := Count - 1;
        end Get;

    end Bounded_Buffer;
```

Здесь предусмотрен циклический буфер, который позволяет сохранить до "Max" значений типа "Item". Доступ обеспечивается с помощью входов "**Put**" и "**Get**". Описание объекта (переменной) защищенного типа осуществляется традиционным образом, а для обращения к защищенным входам, как и для обращения к защищенным подпрограммам, используется точечная нотация:

```
declare
    . . .
    My_Buffer: Bounded_Buffer;
    . . .
begin
    . . .
    My_Buffer.Put (X);
    . . .
end;
```

Заметим, что так же как и в случае вызова входа задачи, вызывающая задача может использовать инструкции отбора для временного или условного вызова защищенного входа. Например:

```
. . .
select
    My_Buffer.Get (X);
    . . .                   -- список инструкций
else
    . . .                   -- список инструкций
end select;
```

Поведение защищенного типа контролируется барьерами. При вызове входа защищенного объекта выполняется проверка соответствующего барьера. Если значение барьера "**False**", то вызов помещается в очередь, подобно тому, как это происходит при вызове входа задачи. При описании переменной "My_Buffer" буфер — пуст, и, таким образом, барьер для входа "**Put**" имеет значение "**True**", а для входа "**Get**" — "**False**". Следовательно, будет выполняться только вызов "**Put**", а вызов "**Get**" будет отправлен в очередь.

В конце выполнения тела входа (или тела процедуры) защищенного объекта производится вычисление значений всех барьеров, у которых есть задачи, ожидающие в очереди, разрешая, таким образом, обработку обращений к входам, которые ранее были помещены в очередь в результате того, что значение барьера было вычислено как "**False**". Таким образом, после завершения обработки первого же вызова "**Put**", если вызов "**Get**" уже находится в очереди, то значение барьера для "**Get**" будет вычислено заново, и это позволит обслужить ожидающий в очереди вызов "**Get**".

Важно понять, что здесь нет задачи, которая непосредственно ассоциирована с самим буфером. Вычисление барьеров эффективно выполняется системой времени выполнения. Барьеры вычисляются когда вход вызывается впервые и когда происходит что-либо, что может повлиять на состояние барьера ожидающей задачи.

Таким образом, барьеры вычисляются заново только в конце выполнения тела защищенного входа или тела защищенной процедуры, но не в конце выполнения тела защищенной функции, поскольку вызов функции не может повлиять на внутреннее состояние защищенного объекта и, как следствие, не может изменить значения барьеров. Такие правила гарантируют эффективность реализации защищенного объекта.

Следует обратить особое внимание на то, что барьер может ссылаться на значение глобальной переменной. Значение такой переменной может быть изменено независимо от вызова защищенной процедуры или защищенного входа объекта (например, она может быть изменена какой-либо другой задачей или даже в результате вызова защищенной функции), и такие изменения не могут быть обработаны достаточно точно.

Поскольку подобные случаи требуют дополнительной внимательности, то рекомендуется воздерживаться от использования глобальных переменных в барьерах.

Необходимо понимать, что защитный механизм барьеров налагается на обычное взаимное исключение защищенной конструкции, предоставляя, таким образом, два различных уровня защиты. В конце защищенного вызова уже находящиеся в очереди обращения к входам (чей барьер теперь имеет значение "**True**") более приоритетны по сравнению с другими защищенными вызовами. С другой стороны, пока защищенный объект занят обработкой текущего вызова (а также любых уже готовых к обработке, но находящихся в очереди вызовов), проверка значения барьера, для вновь поступившего вызова входа, не может быть даже произведена.

Это имеет одно важное следствие: если состояние защищенного объекта изменяется, и существует задача, которая ожидает новое состояние защищенного объекта, то такая задача получает доступ к ресурсу. При этом гарантируется, что состояние ресурса, когда такая задача получает к нему доступ, будет таким же самым, как и в момент принятия решения о предоставлении этой задаче доступа к ресурсу. Таким образом, полностью предотвращаются неудовлетворенные опросы и состязание задач за ресурс.

Основополагающая концепция защищенных объектов подобна мониторам. Они являются пассивными конструкциями с синхронизацией, предусматриваемой системой времени выполнения языка. Однако защищенные объекты, по сравнению с мониторами, обладают большим преимуществом: протокол взаимодействия с защищенными объектами описывается барьерными условиями (в правильности которых достаточно легко убедиться), а не низкоуровневыми и неструктурируемыми сигналами, используемыми в мониторах (как в языке Modula).

Другими словами, защищенные объекты обладают существенными преимуществами высокоуровневых охранных условий модели рандеву, но без дополнительных издержек по производительности, обусловленных наличием дополнительной задачи.

Защищенные модули позволяют осуществлять очень эффективную реализацию различных сигнальных объектов, семафоров и подобных им парадигм:

```
-- спецификация защищенного типа
protected type Counting_Semaphore (Start_Count : Integer := 1) is

    entry Secure;
    procedure Release;
    function Count return Integer;

private
    Current_Count : Integer := Start_Count;

end;


-- тело защищенного типа
protected body Counting_Semaphore is

    entry Secure when Current_Count > 0 is
    begin
        Current_Count := Current_Count - 1;
    end;

    procedure Release is
    begin
        Current_Count := Current_Count + 1;
    end;

    function Count return Integer is
    begin
        return Current_Count;
    end;

end Counting_Semaphore;
```

Особенностью этого примера является то, что "Start_Count" является дискриминантом. При вызове входа

"Secure" опрашивается барьер этого входа. Если результат опроса барьера есть "**False**", то задача ставится в очередь, а ожидание обслуживания становится "**True**".

Следует заметить, что этот пример демонстрирует реализацию общего семафора Дейкстры (*Dijkstra*), где вход "Secure" и процедура "Release" соответствуют операциям "P" и "V" (*Dutch Passeren* и *Vrijmaken*), а функция "Count" возвращает текущее значение семафора.

Очевидно, что в очереди обращения к защищенному входу может одновременно находиться несколько задач. Так же как и в случае с очередями к входам задач, очереди к входам защищенных объектов обслуживаются в порядке поступления вызовов (*FIFO — First-In-First-Out*). Однако в случае поддержки требований приложения *D* (*Annex D*) стандарта Ada95, в котором указываются требования для систем реального времени, допускается использование другой дисциплины обслуживания очереди.

Для защищенного типа можно описать семейство защищенных входов; это делается заданием дискретного типа в спецификации входа, подобно описанию семейств входов для задач. Однако, в отличие от задач, нет необходимости предусматривать самостоятельно выделенное тело для входа, принадлежащего такому семейству. Индекс семейства может использовать барьер, ассоциируемый с таким входом (обычно такой индекс используется в качестве индекса массива значений логического типа "Standard.**Boolean**").

16.3.4 Особенности программирования защищенных входов и подпрограмм

При программировании действий, выполняемых в телах защищенных входов и подпрограмм, следует учитывать, что время выполнения кода внутри защищенного объекта должно быть настолько кратким, насколько это возможно. Это вызвано тем, что пока выполняется этот код, выполнение других задач, которые пытаются получить доступ к данным защищенного объекта, будет задержано. Ада не позволяет принудительно ограничивать максимальную продолжительность выполнения кода во время защищенных действий, хотя и пытается убедиться в том, что задача не будет заблокирована в состоянии бесконечного ожидания доступа к защищенной процедуре или функции. Потенциальными причинами возникновения подобных ситуаций могут быть попытки выполнения (внутри защищенных действий):

- инструкции отбора ("**select**")
- инструкции принятия ("**accept**")
- инструкции вызова входа
- инструкции задержки выполнения
- создание или активация задачи

Перечисленные действия называют потенциально блокирующими. Кроме того, обращение к любому потенциально блокирующему действию является также потенциально блокирующим.

Напомним, что при выполнении вызова защищенного входа в процессе обработки вызова защищенной процедуры или защищенного входа осуществляется проверка барьера. Если барьер закрыт (условие барьера имеет значение "**False**"), то вызов ставится в очередь. После завершения выполнения тела защищенной процедуры или защищенного входа значения всех барьеров вычисляются заново и, возможно, происходит выполнение тела входа. Вычисление значения барьера для входа и постановка вызова входа в очередь являются защищенными операциями ассоциированного с ними защищенного объекта, и они могут быть названы защищенными действиями.

Любое исключение, возбужденное в процессе вычисления значения барьера для входа, приводит к возбуждению исключения *Program_Error* во всех задачах, которые в текущий момент находятся в очереди ожидания обслуживания вызова входа защищенного объекта.

16.3.5 Атрибуты входов защищенных объектов

Входы защищенных объектов имеют атрибуты, назначение которых подобно назначению атрибутов для входов задач:

E'Caller	Возвращает значение типа "Task_ID", которое идентифицирует обрабатываемую в текущий момент задачу, обратившуюся на вход защищенного объекта "E". Использование этого атрибута допустимо только внутри тела входа (для защищенного объекта).
E'Count	Возвращает значение типа "Universal_Integer", показывающее число обращений на входе "E", которые находятся в очереди.

Здесь, подразумевается, что "E" — это имя любого входа защищенного объекта. Так же как и в случае задач, при использовании этих средств следует учитывать, что по истечении некоторого времени ничто не гарантирует активность задачи или ее присутствие в области видимости.

16.4 Перенаправление `requeue`

16.4.1 Проблема предпочтительного управления

В процессе создания некоторого сложного сервера или планировщика для определения правил очередности предоставления сервиса, предусмотренного телом защищенного входа или телом инструкции принятия, достаточно часто бывает необходимо полагаться на текущие значения различных управляющих элементов. Такие управляющие элементы могут быть локальными для сервера и зависимыми только от внутреннего состояния сервера, они также могут являться атрибутами клиента или управляющей задачи, либо они могут быть глобальными для всей системы в целом. Кроме того, состояние управляющих элементов может изменяться с момента вызова входа (защищенного модуля или задачи) до момента начала обработки обращения к входу.

В самых простых случаях, когда управляющие элементы известны вызывающему клиенту, не изменяются с момента вызова и имеют сравнительно небольшой дискретный диапазон значений, использование возможностей предоставляемых семействами точек входа Ada83, может оказаться вполне достаточным. Однако в тех ситуациях, когда для построения логики обслуживания какого-либо сервиса, необходимо использование внутри сервера предпочтительного управления (*preference control*), такие ограничения, как правило, не соблюдаются.

Примером сервера, который нуждается в использовании предпочтительного управления, может служить сервер, управляющий распределением каких-либо ресурсов. После вызова со стороны клиента подобный сервер должен начать обработку запроса только в том случае, если этот запрос может быть удовлетворен немедленно. В противном случае запрос от клиента должен быть отправлен в очередь для более позднего обслуживания.

Идея альтернативного решения заключается в том, что клиент должен сначала опросить состояние сервера, а затем инициировать вызов с параметрами, которые соответствуют текущему состоянию сервера (предположительно, используя определенный индекс в семействе входов). Однако недостатком такого решения является потенциальная возможность возникновения состязания задач за ресурс, поскольку не гарантируется атомарность операций. Иными словами, между двумя обращениями клиента может произойти обслуживание запроса от другого клиента, что приведет к изменению внутреннего состояния сервера. Таким образом, при последующем обращении клиента может быть утеряна достоверность параметров, значения которых основаны на предыдущем запросе.

В других случаях логика работы сервера нуждается в том, чтобы обслуживание какого-либо сервиса было разделено на два (или более) этапа, и чтобы задача-клиент, запрашивающая сервис, была приостановлена после выполнения первого этапа до тех пор, пока какие-либо условия не позволят выполнить последующий этап обработки. В таком случае необходимы последовательные вызовы двух (или более) входов, но попытки запрограммировать это в Ada83, как правило, затруднительны, поскольку в этом случае также теряется атомарность операций и требуется обеспечить видимость внутреннего протокола работы сервера, что нежелательно.

16.4.2 Инструкция перенаправления очереди `requeue`

С целью поддержки построения необходимых управляющих алгоритмов, позволяющих разделить обработку какого-либо сервиса на несколько (два и более) этапов и для организации предпочтительного управления, стандарт Ada95 ввел единственную и простую инструкцию перенаправления очереди "**requeue**".

Инструкция перенаправления очереди используется для завершения инструкции принятия рандеву ("**accept**") или тела защищенного входа при необходимости перенаправления соответствующего вызова клиента от одного входа задачи или защищенного модуля к другому входу (или даже к тому же самому входу). Общий вид этой инструкции следующий:

requeue *имя_входа* [**with abort**] ;

Для того чтобы сервер получил доступ к параметрам вызова, нет необходимости возобновлять обработку вызова и требовать от клиента инициирование вызова на другм входе, основываясь на результатах первого вызова. С помощью инструкции перенаправления "**requeue**", можно просто переслать вызов клиента в очередь другого входа.

В случае вызова защищенного входа исключительный доступ обеспечивается на протяжении периода проверки параметров и выполнения перенаправления. В случае тела инструкции принятия рандеву ("**accept**"), задача-сервер управляет своим собственным состоянием, и в данном случае атомарность также обеспечивается, так как она может отвергнуть принятие любого промежуточного вызова.

Инструкция перенаправления "**requeue**" предназначена для обработки двух основных ситуаций:

- После начала выполнения инструкции принятия "**accept**" или тела защищенного входа, может быть определено, что запрос не может быть удовлетворен немедленно. Взамен, существует необходимость перенаправлять вызывающего клиента до тех пор, пока его запрос/вызов не сможет быть обработан.
- В качестве альтернативы, часть запроса клиента может быть обработана немедленно, но могут существовать дополнительные шаги, которые должны быть выполнены несколько позже.

В обоих случаях, инструкция принятия "**accept**" или тело защищенного входа, нуждается в передаче управления. Таким образом, могут быть обработаны запросы от других клиентов или выполнена какая-либо другая обработка. Инструкция перенаправления "**requeue**" позволяет разделить обработку оригинального запроса/вызова на два (и более) этапа.

Чтобы продемонстрировать логику работы и использования этой инструкции, рассмотрим пример широковещательного сигнала (или события). Задачи приостанавливаются для ожидания некоторого события, а после того как оно происходит, все приостановленные задачи продолжают свое выполнение, а событие очищается. Сложность состоит в том, как предотвратить от попадания в ожидание задачи которые вызвали операцию ожидания после того как событие произошло, но до того как событие очищено. Другими словами, мы должны очистить событие только после того как новые задачи продолжат свое выполнение. Нам позволяет запрограммировать такое предпочтительное управление инструкция перенаправления "**requeue**":

```
protected Event is

    entry Wait;
    entry Signal;

private
    entry Reset;
    Occurred: Boolean := False;

end Event;

protected body Event is

    entry Wait when Occurred is
    begin
        null;                       -- пустое тело!
    end Wait;

    entry Signal when True is       -- барьер всегда True
    begin
        if Wait'Count > 0 then
            Occurred := True;
            requeue Reset;
        end if;
    end Signal;
```

```
        entry Reset when Wait'Count = 0 is
        begin
            Occurred := False;
        end Reset;

    end Event;
```

Задачи указывают на то, что они желают ждать событие, выполняя вызов:

```
    Event.Wait;
```

а возникновение события индицируется тем, что какая-либо задача выполняет вызов:

```
    Event.Signal;
```

после чего, все приостановленные в ожидании события задачи продолжат свое выполнение и событие очистится. Таким образом, последующие вызовы входа "Wait" работают корректно.

Логическая переменная "Occurred" обычно имеет значение "**False**". Она имеет значение "**True**" только тогда, когда задачи возобновляют свое выполнение после приостановки в ожидании события. Вход "Wait" существует, но фактически не имеет тела. Таким образом, вызывающие задачи приостанавливают свое выполнение в его очереди, ожидая того, что переменная "Occurred" получит значение "**True**".

Особый интерес представляет вход "Signal". Значение его барьера постоянно и всегда равно "**True**", таким образом, он всегда открыт для обработки. Если нет задач ожидающих событие (нет задач вызвавших вход "Wait"), то его вызов просто ничего не делает. С другой стороны, при наличии задач ожидающих событие, он должен позволить им продолжить выполнение, причем так, чтобы ни одна новая задача не попала в очередь ожидания события, после чего, он должен сбросить флаг, чтобы восстановить управление. Он выполняет это перенаправляя себя на вход "**Reset**" (с помощью инструкции перенаправления "**requeue**") после установки флага "Occurred" в значение "**True**", для индикации появления события.

Семантика инструкции перенаправления "**requeue**" подобна тому, что описано при рассмотрении алгоритма работы "Signal". Однако, помните, что в конце обработки тела входа или защищенной процедуры осуществляется повторное вычисление состояний тех барьерных условий в очереди которых находятся задачи, приостановленные в ожидании обслуживания. В этом случае, действительно существуют задачи находящиеся в очереди входа "Wait", и существует задача в очереди входа "**Reset**" (та задача которая перед эти вызвала вход "Signal"). Барьер для "Wait" теперь имеет значение "**True**", а барьер для "**Reset**", естественно, "**False**", поскольку очередь задач на входе "Wait" не пуста. Таким образом, ожидающая задача способна выполнить тело входа "Wait" (которое фактически ничего не делает), после чего опять осуществляется повторное вычисление значений барьеров. Этот процесс повторяется до тех пор, пока все приостановленные в ожидании задачи не возобновят свое выполнение и значение барьера для "**Reset**" не получит значение "**True**". Оригинальная задача, которая вызвала сигнал, теперь выполняет тело входа "**Reset**", сбрасывая флаг "Occurred" в значение "**False**", возвращая всю систему в исходное состояние еще раз. Теперь, защищенный объект (как единое целое) полностью освобожден, поскольку нет ни одной ожидающей задачи, ни на одном барьере.

Следует обратить внимание на то, что если какие-либо задачи пытаются вызвать "Wait" или "Signal", когда происходит обработка всего выше описанного процесса, то эти задачи будут заблокированы, поскольку защищенный объект, как единое целое, будет находиться в занятом состоянии. Это иллюстрирует два уровня защиты и является смысловой основой отсутствия возможности появления состязания задач за ресурс.

Другим следствием двухуровневой защиты является то, что все будет работать надежно даже в случае таких сложностей как временные и условные вызовы и принудительное завершение (*abort*). Можно заметить, что это несколько противоположно использованию атрибута "'Count" для входов задач, который не может быть использован при временных вызовах.

Следует сделать небольшое замечание о том, что вход "**Reset**" описан в приватной части защищенного объекта, и, следовательно, не может быть вызван извне. Ada95 позволяет задачам также иметь приватную часть, содержащую приватные входы.

Следует заметить, что показанный выше пример был приведен только в качестве демонстрации. Внимательный читатель должен заметить, что проверка условия внутри "Signal" не является строго необходимой. Без этой проверки, вызвавшая задача будет просто всегда осуществлять перенаправление "**requeue**" и немедленно продолжать вторую часть обработки, при отсутствии ожидающих задач. Однако это условие делает

описание более ясным и чистым. Еще более внимательный читатель может заметить, что мы можем запрограммировать этот пример на Ada95 вовсе без использования инструкции перенаправления "**requeue**".

Как видно из общего вида инструкции перенаправления "**requeue**", она может быть указана с принудительным завершением "**with abort**". В Ada83, после начала рандеву, вызывающий клиент не обладал возможностью отменить свой запрос (таймаут мог быть получен только если запрос не был принят в обработку задачей-сервером). Такое поведение весьма обосновано. После того как сервер начал обработку запроса, он находится в неустойчивом состоянии, и асинхронное удаление вызывающего клиента может нарушить внутренние структуры данных сервера. В дополнение, при наличии параметров переданных по ссылке, сервер, принимающий вызов, должен быть способен получить доступ к данным вызывающего клиента (таким как данные в стеке). При исчезновении вызывающего клиента, это может привести к "висячим" ссылкам и последующему краху работы программы.

Однако, в некоторых случаях, откладывание аннулирования вызова невозможно. В частности, когда значение таймаута необходимо использовать для контроля за общим временем обращения, которое также включает время непосредственной обработки вызова, а не только время ожидания принятия вызова. В дополнение к асинхронной передаче управления Ada95, подобная ситуация может возникнуть когда вызывающий клиент "прерван" и должен изменить последовательность своего выполнения как можно быстрее.

Поскольку не существует единственно возможного наилучшего решения сразу для всех приложений, и поскольку отсутствует легкий обходной путь, то использование инструкции перенаправления ("**requeue**") с принудительным завершением ("**with abort**") предоставляет программисту возможность выбора для его приложения наиболее подходящего механизма. В основном, когда допускается аннулирование вызова в процессе перенаправления, сервер будет сохранять состояние своих внутренних структур данных перед началом выполнения инструкции перенаправления с принудительным завершением, после чего, если вызывающий клиент будет удален из следующей очереди, то сервер сможет нормально продолжить свою работу. Когда это не возможно, или когда не требуется аннулирование вызова в процессе перенаправления, будет достаточно использования простого перенаправления, и вызывающий клиент будет удерживаться до полного завершения обработки запроса.

Инструкция перенаправления "**requeue**" позволяет перенаправить вызывающего клиента в очередь того же самого или какого-либо другого входа. При этом, вызывающий клиент не обязан заботиться о таком перенаправлении и даже о фактическом количестве необходимых для удовлетворения его запроса этапов обработки, которые вообще могут быть не видимыми извне защищенного типа или типа задачи. Использование такого эффекта, предоставляет гибкие возможности построения сервера со сложной внутренней архитектурой и простым, однозначным для клиента, интерфейсом.

В процессе выполнения перенаправления, не происходит никаких переопределений параметров. Вместо этого, значения параметров прямо переносятся в новый вызов. Если был предусмотрен новый список параметров, то он может включать ссылки на данные, которые локальны для инструкции принятия "**accept**" или тела защищенного входа. Это может вызвать некоторые трудности, поскольку выполнение инструкции принятия "**accept**" или тела защищенного входа будет завершено в результате выполнения инструкции перенаправления "**requeue**" и локальные переменные будут, таким образом, деаллоцированы (*deallocated*). Необходимо соответствие используемых подтипов между вновь вызываемым целевым входом (если он имеет какие-либо параметры) и текущим входом. Это позволяет использовать то же самое представление для нового множества параметров, когда они передаются по значению (*by-copy*) или по ссылке (*by-reference*), а также исключить необходимость размещения (*allocate*) нового пространства для хранения параметров. Необходимо отметить, что при выполнении перенаправления, кроме передачи тех же самых параметров, существует еще только одна возможность - не передавать никаких параметров вовсе.

В заключение, как общий итог обсуждения инструкции перенаправления "**requeue**" и логики ее использования, сделаем следующие выводы:

- Инструкция перенаправления "**requeue**" допустима внутри тела защищенного входа или внутри инструкции принятия рандеву "**accept**". Целевой вход (допустим тот же самый вход) может быть в той же самой задаче или том же защищенном объекте, или другой задаче или другом защищенном объекте. Возможно использование любых перечисленных комбинаций.

- Любые фактические параметры оригинального вызова передаются к новому входу. Следовательно, новый вход должен иметь такой же самый профиль параметров или не иметь никаких параметров.

16.5 Цикл жизни задачи

До сих пор, во всех примерах, мы подразумевали только то, что создание задач осуществляется на этапе элаборации, и никак не рассматривали, что происходит с задачами в процессе их существования. Однако, при построении сложных многозадачных программ, необходимо понимание из чего состоит цикл жизни задачи, то есть, как происходит создание, активация и завершение одной отдельно взятой задачи.

16.5.1 Создание задачи

Напомним, что тип задачи может рассматриваться как шаблон для создания реальных объектов-задач. Типы и объекты задач могут быть описаны в любой описательной части, включая тело задачи. Для любого типа задачи, спецификация и тело должны быть описаны вместе, в одном и том же модуле (тело задачи, как правило, помещается в конце описательной части).

Объект задачи может быть создан в процессе элаборации описания какого-либо объекта, расположенного где-либо в описательной части, или в процессе обработки аллокатора (выражение в виде "**new** ... "). Все задачи, которые созданы в результате элаборации описаний объектов, расположенных в одной описательной части (включая внутренние компоненты описываемых объектов), активируются вместе. Подобным образом, все задачи созданные в процессе обработки одиночного аллокатора, также активируются вместе.

Выполнение объекта-задачи имеет три основные фазы:

Активация — элаборация описательной части тела задачи, если она есть (локальные переменные для тела задачи создаются и инициализируются в процессе активации задачи). *Активатор* идентифицирует задачу, которая создала и активировала задачу.

Нормальное выполнение — выполнение инструкций, видимых внутри тела задачи.

Завершение — выполнение любого кода завершения (*finalization*), ассоциированного с любым объектом в описательной части задачи.

Вновь созданная задача находится в ***неактивированом*** (*unactivated*) состоянии. Затем, система времени выполнения осуществляет ассоциирование с этой задачей потока управления (*thread of control*). Если элаборация задачи терпит неудачу, то задача сразу переходит в ***прекращенное*** (*terminated*) состояние. В противном случае, задача переходит в ***работоспособное*** (*runnable*) состояние и начинает выполнять код инструкций тела задачи. Если этот код выполняет некоторые операции, которые блокируют выполнение задачи (рандеву, защищенные операции, задержки выполнения. . .), то задача переходит в ***приостановленное*** (*sleep*) состояние, а затем возвращается обратно в ***работоспособное*** состояние. Когда задача выполняет альтернативу завершения (*terminate alternative*) или нормальным образом завершает свое выполнение, она переходит в ***прекращенное*** (*terminated*) состояние.

Задача индицирует свою готовность к началу завершения выполнением инструкции "**end**". Кроме того, задача может начать процесс своего завершения в результате необработанного исключения, в результате выполнения инструкции отбора "**select**" с альтернативой завершения или в случае выполнения инструкции принудительного завершения "**abort**". Задача, которая окончила свою работу называется ***завершенной*** (*completed*) или ***прекращенной*** (*terminated*), в зависимости от наличия активных задач которые от нее зависят.

Каждая задача имеет ***владельца*** (*master*), которым может являться задача, подпрограмма, инструкция блока или инструкция принятия рандеву "**accept**", содержащая описание объекта-задачи (или, в некоторых случаях, ссылочного типа который ссылается на тип-задачи). Говорят, что задача ***зависит*** от своего владельца.

Задачу, которая осуществляет выполнение владельца, называют ***ведущей*** или ***родительской*** задачей (*parent task*). Таким образом, для каждой задачи существует родительская задача от которой она также зависит.

Приняты следующие правила:

- Если задача была описана как объект, то ее родительская задача — это та задача, которая содержит описание объекта задачи.
- Если задача была описана как часть аллокатора "**new**", то ее родительская задача — это та задача, которая содержит соответствующее описание ссылочного типа.

Когда родительская задача создает новую, ***дочернюю задачу*** (*child task*), ее выполнение приостанавливается на время активации ее дочерней задачи (немедленно, если дочерняя задача создана аллокатором, или после завершения процесса элаборации соответствующей описательной части). Как только все дочерние задачи завершают свою активацию, родительская задача и все ее дочерние задачи продолжают свое выполнение независимо. Если задача, в процессе своей активации, создает другую задачу, то, перед тем как продолжить свое выполнение, она также должна ждать пока активируется ее дочерняя задача.

Существует концептуальная задача, называемая ***задачей окружения*** (*environment task*), которая ответственна за элаборацию всей программы. В общем случае, задача окружения — это поток управления операционной системы, который осуществляет инициализацию системы времени выполнения и выполняет головную подпрограмму Ады. Перед вызовом головной подпрограммы Ада-программы, задача окружения осуществляет элаборацию всех библиотечных модулей, указанных в спецификаторе "**with**" головной подпрограммы. В результате процесса элаборации происходит создание и активация задач уровня библиотеки до вызова головной подпрограммы Ада-программы.

16.5.2 Активация задачи

Для процесса активации задач приняты следующие правла:

1. Для статических задач, активация начинается немедленно после окончания процесса элаборации описательной части в которой эти задачи описаны.

2. Первая инструкция, следующая за описательной частью, не будет выполняться до тех пор, пока все созданные задачи не окончат свою активацию.

3. Задаче, перед выполнением своего тела, не требуется ожидать активацию других задач, которые созданны одновременно с ней.

4. Задача может попытаться осуществить взаимодействие с другой задачей, которая уже создана, но еще не активирована. При этом, выполнение вызывающей задачи будет задержано до тех пор, пока не произойдет взаимодействие.

5. Если объект-задачи описан в спецификации пакета, то он начнет свое выполнение после окончания процесса элаборации описательной части тела пакета.

6. Динамические задачи активируются немедленно после завершения обработки создавшего их аллокатора.

7. Задача, которая выполнила инструкцию, ответственную за создание задач, будет заблокирована до тех пор, пока созданные задачи не окончат свою активацию.

8. Если в процессе элаборации описательной части возбуждено исключение, то любая задача, созданная в процессе элаборации становится ***прекращенной*** и никогда не активируется. В виду того, что на этом этапе задача сама по себе не может обрабатывать исключения, модель языка требует, чтобы такую ситуацию обрабатывала родительская задача или владелец задачи, поскольку будет возбуждено предопределенное исключение *Tasking_Error*.

   - В случае динамического создания задач, возбуждение исключения осуществляется после инструкции которая содержит аллокатор. Однако, если вызов аллокатора находится в описательной части (как часть инициализации объекта), то описательная часть, в результате неудачи, не выполняется, а возбуждение исключения происходит в объемлющем блоке (или вызвавшей подпрограмме).
   - В случае статического создания задачи, возбуждение исключения осуществляется перед выполнением первой исполняемой инструкции, которая следует сразу после описательной части. Это исключение возбуждается после того как осуществлена активация всех созданных задач, вне зависимости от успешности их активации, и оно, в большинстве случаев, вызывается однократно.

9. Атрибут задачи "`'Callable`" возвращает значение "**True**" только в случае, когда указанная задача не находится в состоянии ***завершенная*** или ***прекращенная***. Любая задача находящаяся в ***ненормальном*** (*abnormal*) состоянии является принудительно завершенной задачей. Атрибут задачи "`'Terminated`" возвращает значение "**True**", когда указанная задача находится в ***прекращенном*** состоянии.

16.5.3 Завершение задачи

Перед выходом, ***владелец*** (*master*) осуществляет выполнение конструкций, которые включают очистку (*finalization*) локальных объектов после их использования (и после ожидания всех локальных задач). Каждая задача зависит от одного владельца, или более:

- Если задача была создана в результате обработки аллокатора ссылочного типа, то она зависит от каждого владельца, который включает элаборацию описаний самого отдаленного предка указанного ссылочного типа.
- Если задача была создана в результате элаборации описания объекта, то она зависит от каждого владельца, который включает эту элаборацию.

Кроме того, если задача зависит от указанного владельца, то считается, что она также зависит от задачи, которая выполняет этого владельца, и (рекурсивно) от любого владельца такой задачи.

В первую очередь, зависимая задача дожидается очистки (*finalization*) владельца. Затем, осуществляется очистка каждого объекта с таким же уровнем вложенности как и у очищаемого владельца, если этот объект был успешно проинициализирован и до сих пор существует. Примечательно, что любой объект, уровень вложенности которого глубже чем у владельца, уже не должен существовать, поскольку он должен был очиститься внутренним владельцем. Таким образом, после выхода из владельца, не очищенными остаются только те объекты, уровень вложенности которых меньше чем у владельца.

16.6 Прерывания

Ада позволяет программисту ассоциировать определяемые пользователем обработчики прерываний с некоторыми прерываниями системы. Хотя обработчик прерываний может быть защищенной процедурой или входом задачи, ассоциация с входом задачи, в настоящее время, считается устаревшим средством языка. Таким образом, внимание будет сфокусировано на определяемых пользователем обработчиках прерываний на основе защищенных процедур.

Следует заметить, что в зависимости от решаемой задачи, прерывания Ады могут ассоциироваться с внешними событиями, которые могут являться как непосредственными аппаратными прерываниями, так и определенными событиями используемой операционной системы. В последнем случае, такие события, вне Ада-системы, могут носить название сигналов.

16.6.1 Модель прерываний Ады

Стандарт описывает следующую модель прерывания:

- Прерывание представляет класс событий, которые детектируются оборудованием или системным программным обеспечением.
- ***Появление*** (*occurence*) прерывания состоит из ***генерации*** (*generation*) и ***доставки*** (*delivery*).
- Генерация прерывания — это событие в оборудовании или системе, которое делает прерывание доступным для программы.
- Доставка прерывания — это действия, которые вызывают часть программы (называемую ***обработчиком прерывания***) в ответ на появление прерывания. Прерывание называют ***ожидающим обслуживание*** (*pending*), когда оно находится между генерацией и доставкой прерывания. Вызов обработчика происходит только после доставки каждого прерывания.
- Пока происходит обработка прерывания, последующие прерывания от того же источника ***заблокированы***, то есть, предотвращается генерация всех последующих прерываний. Будут ли утеряны заблокированные прерывания, обычно, зависит от устройства.
- Некоторые прерывания ***зарезервированы***. Программист не может определить обработчик для зарезервированного прерывания. Обычно, зарезервированные прерывания непосредственно обрабатываются библиотекой времени выполнения Ады (например, прерывание от часов, которое используется для реализации инструкций задержки).
- Каждое не зарезервированное прерывание имеет обработчик по умолчанию, который устанавливается библиотекой времени выполнения Ады.

16.6.2 Защищенные процедуры обработки прерываний

Ада предусматривает два стиля установки обработчиков прерываний: ***вложенный*** (*nested*) и ***не-вложенный*** (*non-nested*). При ***вложенном*** стиле, обработчик прерываний в защищенном объекте устанавливается неявно в момент появления данного защищенного объекта, и после того как защищенный объект прекращает существовать, происходит неявное восстановление старого обработчика прерывания. При ***не-вложенном*** стиле, обработчик прерываний устанавливается явным вызовом процедуры, и старый обработчик прерывания может быть восстановлен только явно.

Существует два способа определения устанавливаемого обработчика прерывания вложенного стиля, в зависимости от используемой в защищенном описании директивы компилятора.

В первом случае, устанавливаемый с использованием вложенного стиля обработчик прерывания, идентифицируется последующей директивой компилятора "Attach_Handler", которая появляется в защищенном описании и имеет следующий вид:

```
pragma Attach_Handler (Handler, Interrupt);
```

Здесь, "Handler" указывает имя защищенной процедуры без параметров, которая описана в защищенном описании, а "Interrupt" является выражением типа "Interrupt_ID". Защищенное описание должно выполняться на уровне библиотеки, то есть, оно не может быть вложено в тело подпрограммы, тело задачи или инструкцию блока. Однако, если защищенное описание описывает защищенный тип, то индивидуальный защищенный объект этого типа может быть размещен в этих местах. Динамическое размещение с помощью аллокатора "**new**" обладает большей гибкостью: аллокация защищенного объекта с обработчиком прерывания устанавливает обработчик прерывания, ассоциированный с этим защищенным объектом, а деаллокация защищенного объекта восстанавливает ранее установленный обработчик. Выражение типа "Interrupt_ID" не обязано быть статическим. В частности, его значение может зависить от дискриминанта защищенного типа. Например:

```
package Nested_Handler_Example  is

    protected type  Device_Interface
            (Int_ID : Ada.Interrupts.Interrupt_ID)  is

        procedure Handler;
        pragma Attach_Handler (Handler, Int_ID);

end Nested_Handler_Example;
```

Во втором случае, устанавливаемый с использованием вложенного стиля обработчик прерывания, идентифицируется последующей директивой компилятора "Interrupt_Handler", которая появляется в защищенном описании и имеет следующий вид:

```
pragma Interrupt_Handler (Handler, Interrupt);
```

Здесь, также как и в первом случае, "Handler" указывает имя защищенной процедуры без параметров, которая описана в защищенном описании, а "Interrupt" является выражением типа "Interrupt_ID". Как и в случае использования директивы "Attach_Handler", защищенное описание не может быть вложено в тело подпрограммы, тело задачи или инструкцию блока. Кроме того, эта директива компилятора имеет одно дополнительное ограничение: если защищенная процедура описана для защищенного типа, то объекты этого типа также не могут быть расположены в этих местах. Следовательно, они должны создаваться динамически, с помощью "**new**".

16.6.3 Пакет *Ada.Interrupts*

Не-вложенная установка и удаление обработчиков прерываний полагается на дополнительные средства стандартного пакета *Ada.Interrupts* спецификация которого имеет следующий вид:

```
package Ada.Interrupts is

    type Interrupt_ID is Определяется_Реализацией;
    type Parameterless_Handler is access protected procedure;

    function Is_Reserved (Interrupt : Interrupt_ID)
```

```
        return Boolean;

    function Is_Attached (Interrupt : Interrupt_ID)
        return Boolean;

    function Current_Handler (Interrupt : Interrupt_ID)
        return Parameterless_Handler;

    procedure Attach_Handler
       (New_Handler : in     Parameterless_Handler;
        Interrupt   : in     Interrupt_ID);

    procedure Exchange_Handler
       (Old_Handler : in out Parameterless_Handler;
        New_Handler : in     Parameterless_Handler;
        Interrupt   : in     Interrupt_ID);

    procedure Detach_Handler
       (Interrupt : in     Interrupt_ID);

    function Reference (Interrupt : Interrupt_ID)
        return System.Address;

private

    . . .   -- стандартом не определено

end Ada.Interrupts;
```

Процедура "Attach_Handler" используется для установки соответствующего обработчика прерывания, переопределяя любой существующий обработчик (включая обработчик пользователя). Если параметр "New_Handler" — "**null**", то осуществляется восстановление обработчика по умолчанию. Если параметр "New_Handler" указывает защищенную процедуру для которой не была применена директива компилятора "Interrupt_Handler", то возбуждается исключение *Programm_Error*.

16.6.4 Приоритеты

Для установки приоритета защищенного объекта может быть использована директива компилятора " Interrupt_Priority", которая имеет следующий вид:

```
pragma Interrupt_Priority ( expression );
```

Отсутствие выражения *expression* воспринимается как установка максимального системного приоритета (" Interrupt_Priority 'Last"). Пока обрабатываются операции этого защищенного объекта, прерывания с равным или низшим приоритетом будут заблокированы. Во избежание возникновения ситуации инверсии приоритетов, любая задача, вызывающая операции этого защищенного объекта, должна устанавливать свой приоритет в приоритет этого защищенного объекта на время выполнения операции, отражая срочность завершения операции. Благодаря этому прерывания становятся не блокируемыми. Любой обработчик прерывания обрабатывается с приоритетом своего защищенного объекта, который может быть выше чем приоритет прерывания, если этот же обработчик защищенного объекта обрабатывает более одного вида прерывания. В дополнение к этому, для динамического изменения приоритера, может быть использована процедура "Set_Priority", расположенная в пакете *Ada.Dynamic_Priorities*.

Глава 17

Интерфейс с другими языками

Обычно, не зависимо от того насколько хорош язык программирования, необходимо учитывать, что он должен сосуществовать с программным обеспечением которое написано с помощью других языков программирования. Разработчики Ады предусмотрели это и предлагают стандартные механизмы для осуществления связи с программами которые написаны на других языках.

При этом следует заметить, что различные компиляторы Ады могут расширять стандартные средства взаимодействия с другими языками программирования добавляя какие-либо дополнительные возможности, специфичные для конкретного компилятора. Поэтому, для получения более точных сведений, необходимо обратиться к справочному руководству используемого компилятора Ады.

17.1 Связь с другими языками в Ada83

Стандартным средством взаимодействия с другими языками в Ada83 является директива компилятора "Interface", которая позволяет вызывать подпрограммы написанные на других языках программирования.

Предположим, что при работе в системе Unix, необходимо использовать команду "kill". Для осуществления этого, необходимо выполнить следующее:

```
function kill (pid : in Integer;
               sig : in Integer) return Integer;

pragma Interface (C, kill);
```

В данном случае, первый параметр директивы компилятора "Interface" — это язык вызываемой подпрограммы, а второй — имя подпрограммы под которым она (подпрограмма) известна в программе на Аде.

Пример пакета который импортирует функции написанные на Фортране может иметь следующий вид.

```
package MATHS is
        function sqrt (X : Float) return Float;
        function exp  (X : Float) return Float;
private
        pragma Interface (Fortran, sqrt);
        pragma Interface (Fortran, exp);
end MATHS;
```

Необходимо заметить, что директива компилятора "Interface" не может быть использована с настраиваемыми подпрограммами.

17.2 Связь с другими языками в Ada95

Стандарт Ada95 внес в средства взаимодействия Ады с другими языками программирования некоторые изменения, которые облегчают использование Ады совместно с программным обеспечением написанным на других языках. Согласно стандарта Ada95, для организации взаимодействия с программным обеспечением, написанным на других языках программирования, можно использовать стандартные директивы компилятора "Import", "Export", "Convention" и "Linker_Options". Кроме того, для описания трансляции типов данных

между Адой и другими языками программирования, можно использовать предопределенную библиотеку — стандартный пакет *Interfaces* и его дочерние модули.

17.2.1 Директивы компилятора

Стандарт Ada95 содержит следующее описание стандартных директив компилятора "Import", "Export", "Convention" и "Linker_Options":

```
pragma Import(
    [Convention =>] идентификатор_соглашения, [Entity =>] локальное_имя
  [, [External_Name =>] строковое_выражение]
  [, [Link_Name =>] строковое_выражение]);

pragma Export(
    [Convention =>] идентификатор_соглашения, [Entity =>] локальное_имя
  [, [External_Name =>] строковое_выражение]
  [, [Link_Name =>] строковое_выражение]);

pragma Convention ([Convention =>] идентификатор_соглашения,
                   [Entity =>] локальное_имя);

pragma Linker_Options (строковое_выражение);
```

При описании синтаксиса этих директив компилятора подразумевается, что:

- "Convention" — обозначает язык или, точнее, соглашения (например, для вызова подпрограмм) используемые в конкретном трансляторе; в качестве *идентификатор_соглашения* могут использоваться:
  - "Ada"
  - "Instrinsic"
  - "C"
  - "Fortran"
  - "Cobol"

  при этом, следует заметить, что обязательно поддерживаемыми являются только "Ada" и "Instrinsic", кроме того, реализация конкретного компилятора может добавить любое другое значение
- "Entity" — обозначает имя идентификатора (например, имя вызываемой подпрограммы) в Ада-программе
- "External_Name" — обозначает имя идентификатора в чужом модуле (модуле написанном на другом языке программирования)
- "Link_Name" — обозначает имя идентификатора с точки зрения линкера (редактора связей)

Директива компилятора "Import" предназначена для импортирования объектов (подпрограмм или переменных), описанных на других языках программирования, в Ада-программу. С ее помощью можно вызывать подпрограммы или использовать переменные модулей которые написанны на других языках программирования.

Директива компилятора "Export" предназначена для экспортирования объектов (подпрограмм или переменных), написанных на Аде, для их использования в модулях, написанных на других языках программирования. Следует заметить, что в модулях, написанных на других языках программирования, может потребоваться выполнение вызовов "adainit" и "adafinal" для осуществления правильной инициализации и деструктуризации импортированного Ада-кода.

Директива компилятора "Convention" позволяет указать компилятору на необходимость использования определенных языковых соглашений для какого-либо объекта (подпрограммы или переменной), который впоследствии будет либо импортирован, либо экспортирован.

Директива компилятора "Linker_Options" предназначена для передачи дополнительных опций линкеру (редактору связей). Содержимое строкового параметра этой директивы будет зависеть от используемого линкера.

17.2.2 Интерфейсные пакеты

Стандарт Ada95 определяет интерфейс взаимодействия с языками программирования *C*, *COBOL* и *Fortran*. Для облегчения осуществления связи программ Ады с этими языками программирования существует стандартно определенная иерархия пакетов, состоящия из пакета *Interfaces* и его дочерних модулей:

```
package Interfaces
package Interfaces.C
package Interfaces.C.Pointers
package Interfaces.C.Strings
package Interfaces.COBOL
package Interfaces.Fortran
```

Эти интерфейсные пакеты обеспечивают достаточно мощные средства для взаимодействия с другими языками программирования.

17.3 Взаимодействие с программами написанными на языке *C*

Обсуждение взаимодействия с программами написанными на *C* полагается на описания которые содержатся в стандартном пакете *Interfaces.C*. В частности, этот пакет предусматривает средства преобразования объектов с типами языка *C* в объектыс типами Ады, и обратно.

17.3.1 Численные и символьные типы

С помощью использования следующих типов, переменные или результаты выражений Ады могут быть конвертированы в форму, совместимую с языком *C*, и обратно:

целочисленные типы	**символьные типы**	**вещественные типы**
Int	**Char**	C_Float
Short	Wchar_T	Double
Long	—	Long_Double
Size_T	—	—

Например, для передачи целочисленного значения переменной "Item" как параметра для функции языка *C* которая ожидает получить параметр типа "**long double**" можно использовать следующее выражение Ады "Long_Double (Item)".

17.3.2 Строки языка *C*

Описание массива символов языка *C* имеет следующий вид:

```
type Char_Array is array (Size_T range <>) of aliased Char;
```

Для представления строк, в языке *C* используются массивы символов заканчивающиеся нулевым символом. Нулевой символ используется как индикатор конца строки. Таким образом, описание строки "Name", содержащей текст "Vasya", которая может быть передана как параметр для функции языка *C* может иметь следующий вид:

```
Name : constant Char_Array := "Vasya" & nul;
```

Примечательно, что "nul" используется для представления нулевого символа (индикатор конца строки) и не является зарезервированным словом Ады "**null**".

Для выполнения символьной конверсии можно использовать функции:

```
function To_C (Item: in Character) return Char;
function To_Ada (Item: in Char) return Character;
```

Для выполнения строковой конверсии можно использовать функции:

```
function To_C (Item       : in String;
               Append_Nul : in Boolean := True) return Char_Array;
function To_Ada (Item     : in Char_Array;
                 Trim_Nul : in Boolean := True) return String;
```

Не сложно догадаться, что функции с именами "To_C" должны использоваться для преобразования переменных в форму совместимую с языком *C*, а функии с именами "To_Ada" — обратно.

17.3.3 Примеры организации взаимодействия с *C*

Простым примером импорта *C*-функции может служить пример использования функции "read" системы UNIX в процедуре, написанной на Аде:

```
procedure Read (File_descriptor : in     Integer;
                Buffer          : in out String;
                No_Bytes        : in     Integer;
                No_Read         :    out Integer) is

    function Read (File_descriptor : Integer;
                   Buffer          : System.Address;
                   No_Bytes        : Integer) return Integer;

    pragma Import (C, read, "read");

begin
    No_Read := Read (File_descriptor,
                     Buffer (Buffer'First)'Address,
                     No_Bytes);
end Read;
```

Проблема связи с *C* возникает в ситуации, когда необходимо взаимодействовать с функциями которые имеют список параметров с переменным числом параметров неоднородного типа. Примером такой функции может служить *C*-функция "printf". Необходимо отметить, что подобные функции, по своей природе, не надежны, и не существует удовлетворительного способа решения этой проблемы.

Однако, Ада позволяет взаимодействовать с функциями, имеющими переменное число параметров, но при этом тип параметров однороден. Для взаимодействие с такими функциями можно использовать типы неограниченных (*unconstrained*) массивов.

Предположим, что существует функция, написанная на *C*, описание которой имеет следующий вид:

```
void something(*int[]);
```

Мы можем использовать эту функцию в Аде следующим образом:

```
type Vector is array (Natural range <>) of Integer;

procedure Something (Item : Vector) is

    function C_Something (Address : System.Address);
    pragma Import (C, C_Something, "something");

begin
    if Item'Length = 0 then
        C_Something (System.Null_Address);
    else
        C_Something (Item (Item'First)'Address);
    end if;
end Something;
```

Рассмотрим более сложный пример, который демонстрирует использование *C*-функции "execv" системы UNIX, описанную в *C* следующим образом:

```
int execv(const char *path, char *const argv[]);
```

В данном случае, дополнительную сложность вызывает необходимость трансляции Ада-строк в массивы символов *C*-стиля. Перед тем как описывать непосредственную реализацию, необходимо сделать некоторые описания:

```
type String_Ptr is access all String;
type String_Array is array (Natural range <>) of String_Ptr;

function execv (Path     : String;
                Arg_List : String_Array) return Interfaces.C.Int;
```

```
-- execv заменяет текущий процесс на новый.
-- Список аргументов передается как параметры командной
-- строки для вызываемой программы.
--
-- Для вызова этой подпрограммы можно:
--
-- Option2 : aliased String := "-b";
-- Option3 : aliased String := "-c";
-- Option4 : String := "Cxy";
-- Result  : Interfaces.C.Int;
-- ...
-- Result := execv (Path => "some_program",
--                  -- построение массива указателей на строки...
--                  argv => String_Array'(new String'("some_program"),
--                                        new String'("-a"),
--                                        Option2'Unchecked_Access,
--                                        Option3'Unchecked_Access,
--                                        new String'('-' & Option4));
--
-- Допустимо использовать любую комбинацию
-- динамически размещаемых строк и 'Unchecked_Access
-- к aliased переменным.
-- Однако, необходимо отметить, что нельзя выполнить
-- "some_String"'Access, поскольку Ада требует имя,
-- а не значение, для отрибута 'Access
```

Теперь, реализация может быть выполнена следующим образом:

```
function execv (Path : String;
                argv : String_Array) return Interfaces.C.Int is

   Package C renames Interfaces.C;
   Package Strings renames Interfaces.C.Strings;

   C_Path  : constant Strings.Chars_Ptr (1..Path'Length + 1)
                                  := Strings.New_String(Path);

   type Char_Star_Array is array (1..argv'Length + 1) of
                                     Strings.Char_Array_Ptr;

   C_Argv : Char_Star_Array;
   Index  : Integer;
   Result : C.int;

   function C_Execv (Path       : Strings.Char_Ptr;
                     C_Arg_List : Strings.Char_Ptr) return C.int;
   pragma Import (C, C_Execv, "execv");

begin
```

```
        -- установка массива указателей на строки

        Index := 0;
        for I in argv'Range loop
                Index := Index + 1;
                C_Argv (Index) := Strings.New_String (argv (I).all);
        end loop;


        -- добавление C-значения null в конец массива адресов

        C_Argv (C_Argv'Last) := Strings.Null_Ptr;


        -- передача адресов первых элементов каждого параметра,
        -- как это ожидает C

        Result := C_Execv (C_Path (1)'Address, C_Argv (1)'Address));


        -- освобождение памяти, поскольку часто это не выполняется

        for I in argv'Range loop
                Strings.Free (argv (I));
        end loop;

        Strings.Free (C_Path);

        return Result;
    end execv;
```

Примечательно, что передается адрес первого элемента массива, а не адрес самого массива. Массивы, описываемые как неограниченные, зачастую содержат вектор с дополнительной информацией, которая включает верхнюю и нижнюю границу массива.

Глава 18

Низкоуровневые средства для системного программирования

Кроме богатого набора традиционных средств абстракции данных, Ада, в отличие от многих современных языков программирования, предоставляет ряд низкоуровневых средств, которые могут быть удобны при организации взаимодействия с используемым оборудованием или внешним программным обеспечением. Например, с помощью таких средств можно управлять требуемым двоичным представлением данных или размещением объектов в фиксированных адресах физической памяти.

Описанию средств системного программирования Ады посвящено приложение *C* (*Annex C*) стандарта Ada95. Кроме того, различные реализации компиляторов могут предусматривать дополнительные атрибуты типов и/или директивы компилятора которые управляют внутренним представлением объектов и поведением окружения времени выполнения.

Следует заметить, что многие зависящие от реализации системы константы располагаются в стандартном пакете *System*. Например, число битов в элементе памяти, число доступных элементов памяти, наибольшее и наименьшее доступное целое число, имя операционной системы и т.д.

18.1 Спецификация внутреннего представления данных

С помощью спецификации представления, для идентификаторов любого перечислимого типа можно указать специфические значения для внутреннего представления значений перечислимого типа. Предположим, что у нас есть перечислимый тип, который описывает страны:

```
type Country is (USA, Russia, France, UK, Australia);
```

Мы можем, используя спецификацию представления, указать для каждого идентификатора этого перечислимого типа значение телефонного кода соответствующей страны следующим образом:

```
type Country is (USA, Russia, France, UK, Australia);
for Country use (USA => 1, Russia => 7, France => 33, UK => 44, Australia => 61);
```

Таким образом, внутреннее значение используемое, например, для представления идентификатора "France" будет 33.

При спецификации внутреннего представления перечислимого типа, значения для внутреннего представления должны указываться в порядке увеличения значений и быть уникальны. Следует также обратить внимание на то, что атрибуты перечислимого типа будут возвращать значения которые не учитывают спецификацию внутреннего представления, например:

Country'Succ (USA)	возвратит: "Russia"
Country'Pred (Australia)	возвратит: "UK"
Country'Pos (Russia)	возвратит: "1"
Country'Val (2)	возвратит: "France"

Для того, чтобы получить доступ к значениям внутреннего представления перечислимого типа необходимо воспользоваться стандартным настраиваемым модулем *Ada.Unchecked_Convertion*, который позволяет получить внутреннее представление объекта как значение другого типа, не выполняя при этом никаких промежуточных преобразований.

Следует заметить, что использование модуля настраиваемой функции *Ada.Unchecked_Convertion* обладает одним ограничением: объект-источник и объект-приемник должны иметь одинаковый битовый размер. Поэтому, чтобы гарантировать соответствие размеров объектов источника и приемника, необходимо указать компилятору размер внутреннего представления перечислимого типа "Country". Например, можно указать, что размер внутреннего представления типа "Country" должен быть равен размеру типа "**Integer**". Это можно выполнить следующим образом:

```
type  Country is (USA, Russia, France, UK, Australia);
for Country'Size use Integer'Size;
for Country use (USA => 1, Russia => 7, France => 33, UK => 44, Australia => 61);
```

Напомним, что атрибут "T'Size" возвращает размер экземпляра объекта типа "T" в битах.

Следующий пример простой программы, которая выводит телефонный код Франции, демонстрирует использование рассмотренных средств спецификации внутреннего представления:

```
with  Ada.Unchecked_Conversion;
with  Ada.Text_IO;                use   Ada.Text_IO;

procedure Main  is

    type  Country is (USA, Russia, France, UK, Australia);
    for Country'Size use Integer'Size;
    for Country use (USA       => 1,
                     Russia    => 7,
                     France    => 33,
                     UK        => 44,
                     Australia => 61);

    function International_Dialing_Code is
            new Ada.Unchecked_Conversion (Country, Integer);

begin
    Put ("International Dialing Code for France is ");
    Put (Integer'Image (International_Dialing_Code (France)) );
    New_Line;
end Main;
```

18.2 Привязка объекта к фиксированному адресу памяти

В некоторых случаях может потребоваться выполнение чтения или записи по фиксированному абсолютному адресу памяти. Простым примером подобной ситуации может быть то, что операционная система MS-DOS хранит значение времени в фиксированных адресах памяти `46E` и `46C` (шестнадцатеричные значения). Более точная спецификация этих значений следующая:

`046E - 046F`	время дня в часах
`046C - 046D`	число отсчетов таймера с начала текущего часа (один отсчет таймера равен 5/91 секунды)

Таким образом, для получения текущего времени необходимо осуществить привязку объекта к фиксированному адресу памяти. Для осуществления этого, можно привязать переменную "Time_Hight" типа "**Integer**" к фиксированному адресу "16#046E#" следующим образом:

```
Time_Hight_Address  : constant  Address := To_Address (16#046E#);

type  Time  is range 0 .. 65365;
for Time'Size use 16;
```

```
Time_Hight   : Time;
for Time_Hight'Address use Time_Hight_Address;
```

Следует заметить, что здесь, тип "Time" является беззнаковым 16-битным целым. Величина адреса "16#046E#" должна иметь тип "Address", который описывается в пакете *System*. Стандартная функция "To_Address", которая выполняет преобразование целочисленного значения в значение адреса, описывается в пакете *System.Storage_Elements*.

18.3 Организация доступа к индивидуальным битам

Организацию доступа к индивидуальным битам можно рассмотреть на примере операционной системы MS-DOS, в которой фиксированный адрес памяти "16#0417#" содержит состояние установок клавиатуры. Вид физического представления этого байта следующий:

7	6	5	4	3	2	1	0
Insert	Caps Lock	Num Lock	Scroll Lock	-	-	-	-

Пример следующей простой программы демонстрирует организацию доступа к индивидуальным битам, характеризующим состояние клавиатуры:

```
with  Ada.Text_IO;                 use  Ada.Text_IO;
with  System.Storage_Elements;     use  System.Storage_Elements;

procedure Keyboard_Status_Demo is

    Keyboard_Address  : constant Address := To_Address (16#0417#);

    type  Status  is  (Not_Active, Active);
    for   Status  use (Not_Active => 0, Active => 1);
    for   Status'Size  use 1;

    type  Keyboard_Status is
        record
            Scroll_Lock : Status;   -- состояние Scroll Lock
            Num_Lock    : Status;   -- состояние Num Lock
            Caps_Lock   : Status;   -- состояние Caps Lock
            Insert      : Status;   -- состояние Insert
        end record;
    for Keyboard_Status use
        record
            Scroll_Lock at 0 range 4..4;  -- бит 4
            Num_Lock    at 0 range 5..5;  -- бит 5
            Caps_Lock   at 0 range 6..6;  -- бит 6
            Insert      at 0 range 7..7;  -- бит 7
        end record;

    Keyboard_Status_Byte  : Keyboard_Status;
    for Keyboard_Status_Byte'Address use Keyboard_Address;

begin

    if  Keyboard_Status_Byte.Insert = Active  then
        Put_Line ("Insert mode ON");
    else
        Put_Line ("Insert mode OFF");
    end if;

    if  Keyboard_Status_Byte.Caps_Lock = Active  then
        Put_Line ("Caps Lock mode ON");
    else
```

```
        Put_Line ("Caps Lock mode OFF");
    end if;

    if  Keyboard_Status_Byte.Num_Lock = Active  then
        Put_Line ("Num Lock mode ON");
    else
        Put_Line ("Num Lock mode OFF");
    end if;

    if  Keyboard_Status_Byte.Scroll_Lock = Active  then
        Put_Line ("Scroll Lock mode ON");
    else
        Put_Line ("Scroll Lock mode OFF");
    end if;

end Keyboard_Status_Demo;
```

В данном примере, тип "Status" описан так, чтобы значения этого типа занимали ровно один бит. Далее, с используя тип "Status", описывается тип записи "Keyboard_Status", внутреннее представление которой соответствует физической структуре байта состояния клавиатуры.

Следует заметить, что спецификатор "Scroll_Lock **at** 0 **range** 4 .. 4" указывает, что объект "Scroll_Lock" должен быть размещен по нулевому смещению в четвертой битовой позиции записи "Keyboard_Status" (отсчет ведется в битах от начала записи).

Часть 2.

Идеология языка Ада и некоторые рекомендации

Глава 19

Язык Ада - взгляд "сверху вниз"

Знакомство с языком программирования Ада можно осуществлять несколькими различными путями. Первая часть рассматривает большинство синтаксических конструкций языка программирования Ада и представляет материал по принципу "снизу вверх", демонстрируя конструкции языка Ада с постепенно возрастающей структурой и/или семантической сложностью.

Теперь попытаемся рассмотреть процесс программирования, как процесс проектирования системы, то есть, воспользуемся принципом "сверху вниз". С инженерной точки зрения под *системой* обычно подразумевается множество взаимосвязанных компонент (возможно, функционирующих параллельно), с каждой из которых может быть связана некоторая *информация о состоянии*.

Таким образом, законченную программу на языке Ада можно рассматривать как совокупность взаимосвязанных компонент, называемых модулями или *программными единицами*. Следует заметить, что важной отличительной чертой языка Ада, по сравнению с остальными широко распространенными языками программирования, является акцент, сделанный на различие спецификации и реализации программных единиц. Следовательно, такой подход обеспечивает более удобные возможности разработки программ в виде совокупностей взаимодействующих и поддерживающих свое состояние программных единиц. Кроме того, это позволяет группам программистов создавать и управлять большими системами и программами с минимальными затратами.

Любая программа на языке Ада состоит из набора пакетов и одной главной подпрограммы (то есть начальной или "стартовой"задачи), которая активизирует эти пакеты. Структурное представление программы в виде набора программных единиц имеет преимущество в том, что по мере возрастания размеров и числа программных единиц программист в большей степени сохраняет для себя общее представление о работе программы в целом (как последовательности действий), поскольку число возможных взаимоотношений между шагами задания и данными ограничено благодаря использованию пакетов. Пакеты ограничивают операции, которые могут быть выполнены над индивидуальными объектами типа "данные", благодаря закону (своего рода алгебре), установленному для каждого из пакетов.

Можно сказать, что пакет может быть единым менеджером над созданием и использованием объектов некоторого заданного типа. Учитывая строгую типизацию языка программирования Ада, программист имеет гарантию того, что все экземпляры объектов данного типа будут обработаны соответствующим образом, то есть целостность "содержимого"каждого объекта будет поддерживаться надлежащим образом, о чем программисту не следует заботиться, так как ответственность за это целиком возлагается на реализацию соответствующего пакета. Программирование по такому принципу позволяет пользователю, выполняющему над пакетом некоторую операцию, сфокусировать свое внимание только на объекте типа "данные"или же только на отдельных его частях. Язык Ада позволяет определять неограниченное число типов объектов типа "данные"и назначать создание и использование отдельных объектов типа "данные"за соответствующими четко выраженными менеджерами типа.

В действительности, здесь только слегка затронуты "дисциплинарные преимущества"пакетов, используемых в языке Ада. В более общем случае пакеты могут включать в свою спецификацию не только набор значимых операций над несколькими (одним или более) объектами типа "данные", но и содержать описания, определяющие тип таких объектов (определения типа).

Использование языка Ада для подсистем и прикладных задач заманчиво не только из-за возможности работы с пакетами. В частности, привлекательной является также возможность декомпозиции программы на группы взаимосвязанных задач. Задачи в языке Ада могут создаваться (и прерываться) как статически, так

и динамически. Они позволяют организовывать выполнение задач на конкурентной основе (абстрактной или фактической), что достигается параллельной или конвейерной организацией задач. Представление системы в виде набора задач обеспечивает более ясное понимание ее работы, а также более быстрое функционирование (хотя выполнение задач на конкурентной основе возможно только при наличии в системе нескольких доступных процессоров).

Программа на языке Ада начинает свое выполнение с единственной *нити управления*, связанной с главной подпрограммой (стартовой задачей). При вызове подпрограммы внутри пакета нить управления "перемещается"по кодам, составляющим этот пакет. В итоге нить управления возвращается к стартовой задаче подобно тому, как нить управления перемещается от самого верхнего блока программы на Паскале к различным вызываемым подпрограммам или от них.

В некоторых точках программы (например, во время обработки спецификации задачи) могут быть порождены новые нити управления. Процесс порождения нитей управления может последовательно развиваться, образуя дерево задачи. Окончание такой задачи может произойти только после окончания работы всех порожденных ею задач.

Любая задача может вызвать подпрограмму из другого пакета, однако в результате такой операции новые нити управления не создаются. Под "вызовом пакета"подразумевается обращение к подпрограмме, то есть *операция*, принадлежащая к общедоступной части пакета. В случае вызова пакета нить управления может быть представлена как линия связи от кода вызывающей программы к коду вызываемого пакета и в обратном направлении при возврате из вызываемого пакета.

В противоположность этому задача "вызывает другую задачу", то есть осуществляет обращение ко входу, посылая ей сообщение. В результате вызова одной задачи из другой нить управления вызывающей задачи задерживается до тех пор, пока вызываемая задача не отреагирует на посланное ей сообщение (завершит прием этого сообщения). Если все завершается благополучно, то вызывающую задачу информируют о том, что сообщение было получено. После этого вызывающая программа возобновляет свою работу. Это означает, что нить управления вызывающей задачи возобновляет свою работу, а вызываемая задача продолжает выполнять свой независимый участок программы.

Как мы уже знаем, такой протокол, включающий в себя временную остановку вызывающей задачи, называется *рандеву*. При завершении рандеву вызывающая задача может снова начать выполнение, возможно, параллельно с вызванной ею задачей. Кроме механизма рандеву, для организации взаимодействия задач могут быть использованы защищенные объекты, которые, в отличие от задач, не являются активными сущностями. Использование рандеву и защищенных объектов гарантирует поддержание "*структурности*"программы в терминах "*структурированного программирования*". В общем случае дерево задач, состоящее из "m" порожденных и активных в текущий момент задач, может обрабатываться с уровнем параллельности, равным "m", если в системе имеется "m" доступных процессоров.

Таким образом, если рассматривать всю систему как совокупность компонент (возможно, работающих параллельно), причем, каждая компонента может включать в себя информацию о своем состоянии, то способность языка Ада реализовывать разнообразные богатые структуры пакетов и задач является крупным достижением в разработке языков программирования.

Рассмотрим противоположную ситуацию на примере языка Паскаль. В этом случае большинство информации о состоянии (например, о скалярных переменных, массивах, записях, и т.д.) связано со всей программой, а не с отдельными подпрограммами. Таким образом, за исключением данных, которые объявлены во внешнем блоке программы, время жизни всей объявленной информации ограничено временем нахождения в активном состоянии блока или подпрограммы.

Исходя из этого, программист, работающий на языке Паскаль, может испытывать затруднения при моделировании реальной системы или при объяснении принципов работы созданной им программы другим лицам, которые знакомы с реальной моделью. Это объясняется в первую очередь тем, что отсутствие разнообразных пространств состояний в Паскале запрещает сохранение соответствия между системой и моделирующей ее программой. Более того, наличие в моделируемой системе параллельно функционирующих компонент еще более уменьшает соответствие между программой на Паскале и моделируемой ею системой.

Чем в меньшей степени поведение системы соответствует поведению моделирующей ее программы, тем труднее осуществлять проверку этого соответствия. Такие программы также труднее поддерживать (модифицировать) по мере того, как вносятся изменения в моделируемую систему (из-за изменений в постановке проблемы или в требованиях к результатам). Поскольку расходы на модификацию особенно при больших размерах программ могут быть значительными, то в этом случае одним из существенных преимуществ,

склоняющих к программированию на языке Ада, оказывается возможность поддержания ясного структурного соответствия между программой и моделируемой системой.

В результате, при использовании языков программирования ориентированных на управление (подобных языку программирования Паскаль), большая часть усилий концентрируется на сохранении логического и функционального соответствия между программами и моделируемыми ими системами.

Глава 20

Абстракция данных

Абстракция данных является мощным инструментом современного программирования. Этот концептуальный подход позволяет объединить тип данных с множеством операций, которые допустимо выполнять над этим типом данных. Более того, эта философия позволяет использовать такой тип данных не задумываясь о деталях внутренней организации данных, которые могут быть продиктованы средствами используемого компьютерного оборудования.

Абстракция данных позволяет рассматривать необходимые объекты данных и операции, которые должны выполняться над такими объектами, без необходимости вникать в несущественные детали. Кроме того, абстракция данных является важнейшей составной частью объектно-ориентированного программирования.

20.1 Объектно-ориентированное программирование

Объектно-ориентированное программирование в отличие от метода, основанного на управлении, характеризуется тем, что программа большей частью описывает создание, манипуляцию и взаимодействие внутри набора независимых и хорошо определенных структур данных, называемых *объектами*. Метод, основанный на управлении, рассматривает программу как управляющую последовательность событий (действий) над множеством структур данных этой программы. В обоих случаях задачей программы является преобразование некоторого заданного набора значений данных из некоторой входной или начальной формы к некоторой выходной или конечной форме. Хотя такое отличие на первый взгляд может показаться несколько неестественным или надуманным, имеющиеся отличия тем не менее весьма существенны.

В программировании, основанном на методе, который базируется на управлении, можно обнаружить, что по мере возрастания сложности и размера программы становится все труднее и труднее сохранять ясное представление о всей последовательности (или последовательностях) действий, которые выполняются этой программой. В этом случае, разбивка длинных последовательностей действий на короткие группы может оказать существенную помощь. Фактически хороший программист никогда не создает непрерывные последовательности недопустимо большой длины, разбивая их на серию подпрограмм.

Однако, данные, которые обрабатываются подпрограммами, не могут быть разбиты аналогичным образом на ряд независимых единиц. Как правило, большинство данных находятся в глобальных структурах, к которым имеют доступ все подпрограммы. Следовательно, большинство удобств, ожидаемых от подпрограмм, никогда не реализуются. Суть проблемы заключается в числе подпрограмм и их зависимости от формы и целостности разделяемых структур данных. По мере того как это число растет, становится все более трудным, если не невозможным, организовывать подпрограммы и данные в подходящие подструктуры, например в строго иерархическом порядке, даже при использовании алголоподобных языков, аналогичных Паскалю.

В противоположность этому, в объектно-ориентированном программировании работа начинается со связывания одной-единственной структуры данных с фиксированным набором подпрограмм. Единственными операциями, определяемыми над данным объектом, являются соответствующие подпрограммы. Такую структуру данных называют объектом, а связанный с ней набор операций — "пакетом". Обычно один набор таких операций "общего пользования"определяется и делается доступным для всех компонентов программы, имеющей доступ к этому объекту данных. Эти общедоступные операции имеют строго определенные спецификации. Тела их подпрограмм и любых подпрограмм, от которых эти тела могут в дальнейшем зависеть, можно

обособить и сделать полностью скрытыми. Пакет также может быть использован для скрытия представления внутренней структуры объекта с тем, чтобы подпрограммы из других пакетов не могли обойти эти подпрограммы общего пользования и непосредственно манипулировать с объектом.

Такое обособление называется *абстрактным типом данных* и оно может привести написание программы к значительному упрощению, то есть сделать программу более понятной, корректной и надежной. В дополнение к этому, обеспечивается гибкость (и переносимость), поскольку части объектов, их представления и операции общего пользования могут быть изменены или заменены другим набором частей.

20.2 Сущность абстрактного типа данных

Суть абстракции данных заключается в определении типа данных как множества значений и множества операций для манипулирования этими значениями. Таким образом, какой-либо абстрактный тип данных является простым формальным именем типа для которого определено множество допустимых значений и множество допустимых операций.

Программу, которая использует абстрактный тип данных, называют клиентской программой. Такая программа может описывать объекты абстрактного типа данных и использовать различные операции над этим типом данных. Причем, клиентская программа не нуждается в детальной информации как о внутреннем представления типа данных, так и о фактической реализации операций над этим типом данных. Все эти детали могут быть скрыты от клиентской программы. Таким образом достигается разделение использования типа данных и операций над ним от деталей внутреннего представления данных и реализации операций. Другими словами, такой подход позволяет отделить то ***что*** предоставляет определенный тип данных в качестве сервиса от того ***как*** этот сервис реализован.

Такой подход предоставляет определенные преимущества, позволяя осуществлять независимую разработку как клиентской программы, так и абстрактного типа данных. Например, изменение реализации какой-либо операции над абстрактным типом данных не требует внесения изменений в клиентскую программу. Кроме того, мы можем изменять представление внутренней структуры данных абстрактного типа данных без необходимости изменения клиентской программы.

Абстрактный тип данных является одним из видов повторно используемого программного компонента, который может быть использован большим количеством клиентских программ. Абстрактный тип данных не нуждается в информации о клиентской программе, которая его использует, а клиентская программа не нуждается в информации о внутреннем устройстве абстрактного типа данных. Таким образом абстрактный тип данных можно рассматривать как своеобразный "черный ящик".

Использование абстрактных типов данных облегчает построение крупных программных проектов, поскольку они, как правило, располагаются в библиотеках программных ресурсов, что позволяет избавиться от необходимости бесконечно "изобретать колесо".

20.2.1 Структура абстрактного типа данных

В настоящее время, абстрактный тип данных является одной из общих концепций программирования, вне зависимости от используемого языка программирования. Как правило, абстрактный тип данных состоит из спецификации одного или более типов данных и множества операций над типом или типами. В общем случае, абстрактный тип данных — это составной тип данных, как правило, запись. Операции, которые выполняются над абстрактным типом данных, могут быть логически разделены на несколько групп:

- Конструкторы, которые выполняют создание (или построение) объекта абстрактного типа данных путем объединения отдельных компонентов в единое целое
- Селекторы, которые осуществляют выборку какого-либо отдельного компонента абстрактного типа данных
- Операции опроса состояния абстрактного типа данных
- Операции ввода/вывода

20.2.2 Средства Ады для работы с абстрактными типами данных

Ада предоставляет набор средств, которые обеспечивают поддержку разработки и использования абстрактных типов данных. Такими средствами являются:

Подтипы — позволяют описывать классы численных и перичислимых значений, и назначать им ограничения диапазонов значений, что позволяет компилятору осуществлять строгий контроль над корректностью используемых значений.

Инициализация полей записи — позволяет описывать тип записи так, что каждое индивидуальное поле записи, в каждой переменной этого типа, будет предварительно инициализировано предопределенным значением.

Пакеты — являются идеальным средством группирования совместно используемых ресурсов (типы, процедуры, функции, важные константы...) и предоставления этих ресурсов клиентским программам. При этом осуществляется строгий контроль над контрактной моделью пакета: все что обещано в спецификации пакета должно быть обеспечено в теле пакета, а клиентская программа должна корректно использовать ресурсы предоставляемые пакетом (например, вызывать процедуры только с корректными параметрами).

Приватные типы — позволяют разрабатывать пакеты, которые предоставляют клиентским программам новый тип данных таким образом, чтобы клиентская программа не имела возможности выполнить непреднамеренное изменение значений путем использования приватной информации о внутреннем представлении данных приватного типа, ограничивая тем самым возможность использования внутреннего представления приватного типа данных только внутри тела пакета.

Совмещение знаков операций — позволяет определять новые арифметические знаки операций и знаки операций сравнения для вновь определяемых типов данных, и использовать их также как и предопределенные знаки операций.

Определяемые пользователем исключения — позволяют разработчику пакета предусматривать исключения для клиентской программы, которые будут сигнализировать клиентской программе о том, что внутри пакета выполнены какие-либо неуместные действия. В свою очередь, разработчик клиентской программы может предусмотреть обработку исключений, которые могут возникнуть внутри используемого клиентской программой пакета.

Атрибуты — позволяют создавать подпрограммы для манипуляции структурами данных без наличия всех деталей организации данных (например, атрибуты "First" и "Last" для массивов).

Настраиваемые модули — позволяют создавать подпрограммы и пакеты, которые имеют более общий характер. Это подразумевает, что в процессе написания настраиваемого модуля отсутствует информация о фактически обрабатываемом типе данных. Тип данных может быть указан как параметр настройки позже, в процессе конкретизации настраиваемого модуля.

20.3 Пакеты в языке Ада

20.3.1 Пакеты как средство абстракции данных

Пакет в языке Ада представляет собой конструкцию, которая может использоваться для определения абстрактных данных. Большинство экспертов рассматривают эту возможность как основной вклад языка Ада в решение задачи разработки программного обеспечения с наименьшими затратами, большей надежностью, гибкостью и верифицируемостью. Пакет предоставляет возможность связать отчетливо выраженный набор спецификаций для работы со структурой данных (или с классом структур данных) с некоторыми, возможно скрытыми, подробностями реализации.

При использовании пакетов для абстракции данных их можно разделить на два вида: *пакеты-преобразователи* и *пакеты-владельцы*. Пакеты-преобразователи могут использоваться для реализации чисто абстрактных типов данных, а пакеты-владельцы — для реализации более обобщенных видов менеджеров типа. Более точно это можно охарактеризовать следующим образом:

Пакет-преобразователь (невладелец) — может обновлять только те данные, которые были переданы ему через формальные параметры. Такой пакет не содержит внутри себя информацию о состоянии, относящуюся к этим данным. Это означает, что пакет не может содержать внутри себя обновляемых данных. Поскольку пакет-преобразователь "не владеет"никакой информацией о состоянии, каждая выполняемая им операция должна возвращать результат вызвавшей его программе. Пакет-преобразователь не может "помнить"результаты предыдущих обращений к нему. Простым примером пакета-преобразователя может служить пакет, который создает комплексные числа, производит операции над комплексными числами или осуществляет обе эти операции.

Пакет-владелец — "чувствителен к истории", поэтому он может "помнить"результаты предыдущих обращений к нему. Пакет-владелец "владеет"некоторой информацией о состоянии. Это означает, что пакет содержит внутри себя данные, которые могут быть обновлены в течение жизни этого пакета. Таким образом, для выполнения операции пакету могут быть переданы данные через его формальные параметры. Пакет-владелец после выполнения операции не должен обязательно возвращать результат вызвавшей его программе. Примером пакета-владельца может служить пакет, который поддерживает для компилятора базовую таблицу активных файлов.

Следует заметить, что пакеты-преобразователи, как правило, используются для реализации абстрактных типов данных, которые предоставляют клиентской программе тип данных с помощью которого в ней могут быть описаны объекты (переменные) этого типа.

В свою очередь, пакеты-владельцы, как правило, используются для реализации абстрактных объектов данных, что подразумевает инкапсуляцию одиночного объекта внутри тела пакета. Причем, такой объект непосредственно не доступен для клиента, а манипуляция этим объектом осуществляется через подпрограммы, которые указаны в спецификации пакета.

Каждый пакет имеет *приватную* и *открытую* части (также называемые *видимой* и *невидимой* частями). Открытая часть непосредственно доступна клиентской программе через:

- явное указание перед программой имен пакетов, к которым должен осуществляться доступ (такое указание носит название списка "**with**")
- вложение пакета внутрь клиентской программы, делая этим видимую часть пакета локально доступной (что осуществляется согласно правилам, напоминающим правила видимости в языке Алгол)

Первый из этих механизмов представляет собой важное новшество, внесенное языками, подобными языку Ада. Использование его настоятельно рекомендуется. Второй из этих механизмов использовать нежелательно. Его применение ведет к появлению трудночитаемых и трудноотлаживаемых программ и является "вредным наследием"блочно-структурированных языков, подобных Алголу-60.

В любом случае приватная часть пакета непосредственно недоступна для клиентской программы. Эта информация становится доступной только косвенным образом, посредством выполнения операций, указанных в информации об интерфейсе в видимой части.

Поскольку структура пакета обеспечивает четкое разделение на приватную и видимую части (спецификации), то процесс разработки приватной части может осуществляться отдельно от видимой. Это означает, что реализация приватной части может быть заменена на другую версию (когда появятся лучшие или более правильные идеи реализации) без изменения оставшейся части программы.

20.3.2 Сравнение пакетов и классов

Необходимо обратить внимание на то, что использование пакетов Ады, как инструмента абстракции данных, подобно классам в других объектно-ориентированных языках программирования, где в качестве средства абстракции данных используется класс, определяющий тип данных, множество операций и действия, необходимые для инициализации.

При сравнении подобного использования пакетов Ады с классами можно обнаружить следующее:

- При описании объекта (переменной) с помощью типа класс она будет автоматически инициализирована с помощью действий инициализации класса. Определенные для такого объекта операции обозначаются именем объекта, за которым указывается имя операции.

- При использовании для реализации абстрактного типа данных пакета, инициализация выполняется явным вызовом процедуры инициализации, а имя каждого абстрактного объекта передается как параметр подпрограммы (операции) пакета.

Таким образом, класс имеет вид более чистой реализации абстрактного типа, однако он требует некоторых дополнительных затрат производительности. Класс обеспечивает неявное дублирование объектов, предоставляя (по крайней мере, в принципе) новое множество операций для каждого объекта, генерируемого классом. Пакет нельзя дублировать, и поэтому все операции пакета одни и те же для всех объектов. При сравнении пакетов и классов доводы в пользу классов, как правило, основаны на эстетических соображениях. Практически, пакеты Ады полностью соответствуют требованиям реализации абстрактных типов данных, и, кроме того, имеют по сравнению с классами множество других применений.

Глава 21

Общие приемы программирования

21.1 Абстракция стека

В качестве простого примера, который демонстрирует использование пакета для реализации абстрактного типа данных, рассмотрим стек. Следует заметить, что клинтам этого пакета абсолютно безразлично каким образом выполнена внутренняя реализация стека, то есть клиенты не нуждаются в информации о том, что стек реализован с помошью массива или связанного списка. Шаблоном подобной абстракции может служить следующее:

```
package Stacks is

    Max : constant  := 10;

    subtype Count is Integer range 0..Max;
    subtype Index is range 1..Max;

    type List is array (Index) of Integer;

    type Stack is
        record
            Values : List;
            Top    : Count;
        end record;

    Overflow  : exception;
    Underflow : exception;

    procedure Push (Item : in out Stack; Value : in Integer);
    procedure Pop (Item : in out Stack; Value : out Integer);

    function Full (Item : Stack) return Boolean;
    function Empty (Item : Stack) return Boolean;

    -- возвращает пустой проинициализированный стек
    function Init return Stack;

end Stacks;
```

Однако, в данном случае детали реализации достаточно очевидны, то есть кроме того, что клиенту пакета доступно то, ***что*** пакет предоставляет в качестве сервиса, клиенту пакета также доступно и то, ***как*** этот сервис реализован. Такую ситуацию можно достаточно просто изменить переместив все, в чем клиент не будет нуждаться, в приватную секцию спецификации пакета:

```
package Stacks is

    type Stack is private;
    -- неполное описание типа Stack.
    -- это делает информацию о деталях реализации недоступной.
```

```
    Overflow  : exception;
    Underflow : exception;

    procedure Push (Item : in out Stack; Value : in Integer);
    procedure Pop (Item : in out Stack; Value : out Integer);

    function Full (Item : Stack) return Boolean;
    function Empty (Item : Stack) return Boolean;

    -- возвращает пустой проинициализированный стек
    function Init return Stack;

private
    -- полное описание типа, вместе с другими приватными деталями.
    Max : constant  := 10;
    subtype Count is Integer range 0..Max;
    subtype Index is range 1..Max;

    type List is array (Index) of Integer;

    type Stack is
        record
            Values : List;
            Top    : Count;
        end record;

end Stacks;
```

Следует заметить, что хотя программист использующий такую абстракцию, имеет возможность увидеть все детали реализации (при наличии исходных текстов кода), он не может написать свою собственную программу, которая будет использовать эту информацию. Также очевидно, что такое разделение вынуждает лучше понять, что для абстракции важно, а что нет.

В качестве альтернативы, можно изменить приватную секцию в соответствии с реализацией стека на связанном списке:

```
package Stacks is

    type Stack is private;
    -- неполное описание типа Stack.
    -- это делает информацию о деталях реализации недоступной.

    Overflow  : exception;
    Underflow : exception;

    procedure Push (Item : in out Stack; Value : in Integer);
    procedure Pop (Item : in out Stack; Value : out Integer);

    function Full (Item : Stack) return Boolean;
    function Empty (Item : Stack) return Boolean;

    -- возвращает пустой проинициализированный стек
    function Init return Stack;

private
    type Stack;

    type Ptr is access Stack;

    type Stack is
        record
            Value  : Integer;
            Next   : Ptr;
        end record;

end Stacks;
```

Программа-клиент может использовать оба варианта пакета следующим образом:

```
with Stacks;          use Stacks;

procedure Demo is

    A    : Stack := Init;
    B    : Stack := Init;
    Temp : Integer;

begin
    for I in 1..10 loop
        Push (A, I);
    end loop;

    while not Empty(A) loop
        Pop (A, Temp);
        Push (B, Temp);
    end loop;
end Demo;
```

Примечательно, что оба варианта реализации стека достаточно отличаются один от другого. Таким образом, показанные выше примеры наглядно демонстрируют использование идеи возможности подстановки различной реализации, или, более точно, использование программой-клиентом только того, что предоставляет публичный интерфейс. Напомним, что этот принцип очень важен при построении надежных систем.

Рассмотрим еще один пример использования рассмотренной абстракции стека:

```
with Stacks;          use Stacks;

procedure Not_Quite_Right is

    A, B : Stack

begin
        Push (A, 1);
        Push (A, 2);

        B := A;
end Not_Quite_Right;
```

Не трудно заметить, что при копировании реализация стека с использованием массива будет работать корректно, а реализация стека с использованием связанного списка скопирует только указатель на "голову"списка, после чего "A" и "B" будут указывать на один и тот же связанный список. Действительно, поскольку внимание текущего обсуждения больше посвящено теме абстракции данных, показанная выше реализация стека на связанном списке максимально упрощена. Для того чтобы избавиться от подобных "сюрпризов"в практической реализации стека на связанном списке необходимо использовать средства, которые предоставляют контролируемые типы Ады.

21.2 Приватное наследование

21.2.1 Абстракция очереди

Очень важно определить различие между тем какие сервисы предусматривает тип, и как этот тип их реализует. Например, абстракция списка, которая имеет соответствующие подпрограммы и сама может быть реализована как связанный список или как массив, может быть использована для реализации абстракции очереди. Абстракция очереди будет иметь только несколько операций:

```
Add_To_Tail
Remove_From_Head
Full
Empty
Init
```

Необходимо убедиться, что в программе существует четкое различие между используемой абстракцией и реализацией. Кроме того, желательно возложить на компилятор проверку правильности осуществления вызова подпрограмм реализации, чтобы автоматически предотвратить непреднамеренные обращения к подпрограммам используемой абстракции.

Ниже приведен пример пакета использование которого может привести к проблемам:

```
package Lists is

    type List is private;

    procedure Add_To_Head (Item : in out List; Value : in Integer);
    procedure Remove_From_Head (Item : in out List; Value : out Integer);

    procedure Add_To_Tail (Item : in out List; Value : in Integer);
    procedure Remove_From_Tail (Item : in out List; Value : out Integer);

    function Full (Item : List) return Boolean;
    function Empty (Item : List) return Boolean;

    function Init return List;

private
    type List is ... -- полное описание типа

end Lists;



with Lists;         use Lists;         -- абстракция списка Lists

package Queues is

    type Queue is new List;
    -- наследует операции типа List
    -- то есть, следующее описано "автоматически" (неявно)
    --
    -- procedure Add_To_Head (Item : in out Queue; Value : in Integer);
    -- procedure Remove_From_Head (
    --         Item   : in out Queue;
    --         Value :    out Integer);
    -- и т.д.

end Queues;
```

Здесь, тип очереди ("Queue") наследует все операции типа список ("List"), даже те, которые для него не предназначены (например "Remove_From_Tail"). При такой реализации типа "Queue", клиенты абстракции могут легко нарушить очередь, которая должна разрешать только вставку в конец очереди и удаление из начала очереди.

Например, клиент пакета *Queues* может легко сделать следующее:

```
with Queues;          use Queues;

procedure Break_Abstraction is

    My_Q : Queue;

begin
    Add_To_Head (My_Q, 5);
    Add_To_Tail (My_Q, 5);

    -- очередь должна разрешать вставку в конец очереди
    -- и удаление из начала очереди,
    -- или наоборот
```

```
end Break_Abstraction;
```

Для решения подобной проблемы при создании абстракции очереди необходимо использовать абстракцию списка приватно:

```
with Lists;          -- это используется только приватной секцией

package Queues is

    type Queue is private;

    procedure Remove_From_Head (Item : in out Queue; Value : out Integer);
    procedure Add_To_Tail (Item : in out Queue; Value : Integer);

    function Full (Item : Queue) return Boolean;
    function Empty (Item : Queue) return Boolean;

    function Init return Queue;

private
    type Queue is new Lists.List;

end Queues;



package body Queues is

    -- выполняем конверсию типов (из Queue в List), а затем,
    -- выполняем вызов соответствующей подпрограммы List

    use Lists;

    ------------------------------------------------------------------------

    procedure Remove_From_Head (Item : in out Queue; Value : out Integer) is
    begin
        Lists.Remove_From_Head (List (Item), Value);
    end Remove_From_Head;

    ------------------------------------------------------------------------

    function Full (Item : Queue) return Boolean is
    begin
        return Lists.Full (List (Item));
    end Full;

        . . .

    function Init return Queue is
    begin
        return Queue (Lists.Init);
    end Init;

end Queues;
```

21.2.2 Еще один пример стека

Предположим, что мы хотим создать пакет стека, но при этом, мы не хотим переписывать все подпрограммы. Мы уже имеем реализацию связанного списка, которая может служить основой для организации стека. Заметим, что пакет *Lists* сохраняет значения типа "**Integer**".

Допустим, что существует пакет *Lists*, который описан следующим образом:

```
package Lists is
```

```
    type List is private;

    Underflow : exception;

    procedure Insert_At_Head (Item : in out List; Value : in Integer);
    procedure Remove_From_Head (Item : in out List; Value : out Integer);

    procedure Insert_At_Tail (Item : in out List; Value : in Integer);
    procedure Remove_From_Tail (Item : in out List; Value : out Integer);

    function Full (Item : List) return Boolean;
    function Empty (Item : List) return Boolean;

    -- возвращает пустой проинициализированный список
    function Init return List;

private

    . . .

end Lists;
```

Позже можно будет использовать возможность Ады, которая позволяет скрыть полное описание типа в приватной секции. В данном случае, описывается абсолютно новый тип (поскольку в этом заинтересован клиент), который реально реализован как производный тип. В результате, клиент не может это "увидеть", и детали реализации остаются надежно спрятанными "под капотом".

Таким образом, реализация стека осуществляется следующим образом:

```
with Lists;

package Stacks is

    type Stack is private;

    Underflow : exception;

    procedure Push (Item : in out Stack; Value : in Integer);
    procedure Pop (Item : in out Stack; Value : out Integer);

    function Full (Item : Stack) return Boolean;
    function Empty (Item : Stack) return Boolean;

    -- возвращает пустой проинициализированный стек
    function Init return Stack;

private
    -- создаем новый тип, производя его от уже существующего
    -- это остается спрятанным от клиента, который будет
    -- использовать наш новый тип

    type Stack is new Lists.List;

    -- тип Stack наследует все операции типа List
    -- это неявно описывается как
    --
    -- procedure Insert_At_Head (Item : in out Stack; Value : in Integer);
    -- procedure Remove_From_Head (Item : in out Stack;
    --                             Value :    out Integer);
    --     . . .
    -- function Full (Item : Stack) return Boolean;
    -- function Empty (Item : Stack) return Boolean;
    -- function Init return Stack;

end Stacks;
```

Теперь нам необходимо установить связь между неявно описанными подпрограммами (унаследованными от типа "List"), и подпрограммами, которые должны быть публично доступны. Это достаточно просто выполняется с помощью механизма "транзитного вызова".

```
package body Stacks is

    procedure Push (Item : in out Stack; Value : in Integer) is
    begin
        Insert_At_Head (Item, Value);
    end Push;
    -- эту процедуру можно сделать более эффективной, используя:
    pragma Inline (Push);


    procedure Pop (Item : in out Stack; Value : out Integer) is
    begin
        Remove_From_Head (Item, Value);
    exception =>
        when Lists.Underflow =>
            raise Stacks.Underflow;
    end Pop;

        . . .

end Stacks;
```

Следует заметить, что все это справедливо для публично описанных подпрограмм, имена которых отличаются от имен неявно описанных подпрограмм (тех, которые унаследованы от типа "List"). Однако, в спецификации пакета, присутствуют две функции "Full", которые имеют профиль:

```
function Full (Item : Stack) return Boolean;
```

В данном случае, одна функция описана в публично доступной секции, а вторая, неявно описана в приватной секции. Что же происходит на самом деле? Спецификация первой функции в публично доступной секции пакета - это только обещание предоставить эту функцию. Таким образом, это еще не реализованая функция, ее реализация будет выполнена где-то в теле пакета. Кроме этой функции существует другая, неявно описанная функция, которая, по-случаю, имеет такие же имя и профиль. Компилятор Ады может обнаружить, что обе функции имеют одинаковые имя и профиль, но он ничего не предполагает о том, что публично описанная функция должна быть реализована приватной функцией.

Например, предположим, что "Lists.Full" всегда возвращает "*ложь*"("**False**"), для индикации того, что связанный список никогда не заполнен полностью и всегда может увеличиваться. Разработчику стекового пакета может понадобиться принудительное ограничение размера стека. Например, так чтобы стек мог содержать максимум 100 элементов. После чего разработчик стекового пакета соответствующим образом пишет функцию "Stacks.Full". Это будет не корректно для реализации функции "Full" по умолчанию, которая наследуется от реализации связанного списка.

Согласно правил видимости Ады, явное публичное описание полностью скрывает неявное приватное описание, и, таким образом, приватное описание становится вообще не доступным. Единственным способом вызвать функцию "Full" пакета *Lists*, будет явный вызов этой функции.

```
package body Stacks is

    function Full (Item : Stack) return Boolean is
    begin
        -- вызов оригинальной функции Full из другого пакета
        return Lists.Full (Lists.List (Item));
                          -- конверсия типа параметра
    end Full;

        . . .

end Stacks;
```

Все это выглядит ужасающе, но в чем заключается смысл таких требований Ады? Оказывается, обнаружение подобного рода проблем заключается в сущности конструирования программ с независимыми пространствами имен, или, выражаясь точнее, в разделении различных областей действия имен, где одинаковые имена не конфликтуют между собой. Таким образом, Ада просто предусматривает решение проблемы, которая является неизбежным последствием наличия различных пространств имен. Вариант языка в котором отсутствуют пространства имен - еще хуже чем такое решение. Поэтому, "не надо выключать пейджер, только потому, что не нравятся сообщения". Это очень существенное замечание, и не следует двигаться дальше, пока суть этого замечания не понятна.

Вид всего тела пакета *Stacks* может быть следующим:

```
use Lists;

package body Stacks is

    procedure Push (Item : in out Stack; Value : in Integer) is
    begin
        Insert_At_Head (Item, Value);
    end Push;


    procedure Pop (Item : in out Stack; Value : out Integer) is
    begin
        Remove_From_Head (Item, Value);

    exception =>
        when Lists.Underflow =>
            raise Stacks.Underflow;
    end Pop;
    -- примечательно, что исключение объявленное в другом пакете
    -- "транслируется" в исключение описанное внутри этого пакета


    function Full (Item : Stack) return Boolean is
    begin
        return Lists.Full (List (Item));
    end Full;


    function Empty (Item : Stack) return Boolean is
    begin
        return Lists.Empty (List (Item));
    end Empty;


    -- возвращает пустой проинициализированный стек
    function Init return Stack is
    begin
        return Stack (Lists.Init);
    end Init;

end Stacks;
```

21.3 Использование настраиваемых модулей

21.3.1 Создание абстракций из настраиваемых абстракций

Достаточно часто встречаются случаи, когда настраиваемый пакет, который предоставляет подходящую структуру данных, может быть использован для построения реализации другой абстракции.

Предположим, что существует пакет *Generic_Lists* спецификация которого имеет следующий вид:

```
generic
    type Element is private;

package Generic_Lists is

    type List is private;

    Underflow : exception;

    procedure Insert_At_Head (Item : in out List; Value : in Element);
    procedure Remove_From_Head (Item : in out List; Value : out Element);

    procedure Insert_At_Tail (Item : in out List; Value : in Element);
    procedure Remove_From_Tail (Item : in out List; Value : out Element);

    function Full (Item : List) return Boolean;
    function Empty (Item : List) return Boolean;

    -- возвращает пустой инициализированный список
    function Init return List;

private

    . . .

end Generic_Lists;
```

Мы можем конкретизировать этот настраиваемый модуль для создания экземпляра нового пакета, который будет сохранять для нас значения типа "**Integer**". Необходимо обратить внимание на то, что конкретизацию такого настраиваемого модуля следует осуществлять в приватной секции пакета. Это позволяет избежать нарушения абстракции и предотвращает возможность появления каких-либо дополнительных трудностей разработки.

После этого, настраиваемый модуль может быть использован следующим образом:

```
with Generic_Lists;

package Stacks is

    type Stack is private;

    Underflow : exception;

    procedure Push (Item : in out Stack; Value : in Integer);
    procedure Pop (Item : in out Stack; Value : out Integer);

    function Full (Item : Stack) return Boolean;
    function Empty (Item : Stack) return Boolean;

    -- возвращает пустой инициализированный стек
    function Init return Stack;


private

    -- конкретизация настраиваемого модуля для получения нового пакета
    package Lists is new Generic_Lists (Integer);


    type Stack is new Lists.List;
    -- как и раньше, тип Stack наследует все операции типа List

end Stacks;
```

Тело пакета *Stack* будет таким же как и описывалось ранее. Следует заметить, что в данном случае мы можем навязать приватность без изменения того, что тип выполняет для клиента.

21.3.2 Настраиваемый модуль как параметр настройки

Настраиваемый модуль может быть конкретизирован с помощью использования настраиваемого модуля в качестве формального параметра настройки. В таком случае спецификация формального параметра может быть выполнена следующим образом:

```
generic
  with package P is new P_G (<>);  -- фактический параметр для P_G определяется
package Module_G is ...            -- в P_G как обычно, а также (new) в Module_G
```

Такой вид формального параметра настройки значительно упрощает импортирование целой абстракции. При этом нет необходимости указывать тип и делать полное перечисление необходимых подпрограмм, с соответственным указанием их профилей (параметр/результат). Следовательно, вся абстракция типа данных, которая описывается настраиваемым пакетом, может быть импортирована очень просто.

Такой механизм подразумевает построение одной настраиваемой абстракции из другой настраиваемой абстракции или расширение уже существующей абстракции. Например, так может быть построен пакет двунаправленного связанного списка (экспортируя "Insert", "Delete" и "Swap") при описании операции "Sort", выполняющей сортировку при вставке.

В качестве иллюстрации данного подхода рассмотрим спецификации двух взаимосвязанных абстрактных типов данных: комплексные числа и матрицы комплексных чисел. Абстрактный тип данных комплексных чисел — это настраиваемый модуль с формальным параметром вещественного типа:

```
generic
  type Float_Type is digits <>;
package Complex_Numbers_G is
  type Complex_Type is private;
  function Value (Re, Im : Float_Type) return Complex_Type;
  function "+" (Left, Right : Complex_Type) return Complex_Type;
  ... -- другие операции

private
  type Complex_Type is record
    Re, Im : Float_Type;
  end record;
end Complex_Numbers_G;
```

Абстрактный тип данных матрицы комплексных чисел — это настраиваемый модуль с формальным параметром абстрактного типа данных комплексных чисел (то есть с любой конкретизацией настраиваемого пакета показанного выше):

```
with Complex_Numbers_G;
generic
  with package Complex_Numbers is
    new Complex_Numbers_G (<>);
  use Complex_Numbers;
package Complex_Matrices_G is
  type Matrix_Type is
    array (Positive range <>, Positive range <>) of
      Complex_Type;
  function "*" (Left : Complex_Type; Right : Matrix_Type)
    return Matrix_Type;
  ... -- другие операции
end Complex_Matrices_G;
```

Следующий пример показывает две типичные конкретизации абстрактных типов данных для выполнения вычислений с комплексными числами и матрицами:

```
package Complex_Numbers is
  new Complex_Numbers_G (Float);
package Complex_Matrices is
  new Complex_Matrices_G (Complex_Numbers);
```

```
package Long_Complex_Numbers is
  new Complex_Numbers_G (Long_Float);
package Long_Complex_Matrices is
  new Complex_Matrices_G (Long_Complex_Numbers);
```

21.3.3 Тэговый тип как параметр настройки

Поскольку стандарт Ada95 различает два вида типов: тэговые и не тэговые, - то бывают случаи когда при работе с настраиваемыми модулями их необходимо различать. Подобная необходимость более точного указания свойств ожидаемого формального параметра настройки (формальный тип тэговый или нет) может возникнуть при желании более строгого определения контрактной модели настраиваемых модулей. Указание тэгового типа как формального параметра настройки может иметь следующий вид:

```
generic
  type T is tagged private;
package Module_G is ...
```

В случае использования тэговых типов как формальных параметров настройки особый интерес представляют конструкции для расширения типа и для надклассового программирования. Конструкция для расширения типа позволяет добавлять новые компоненты, и является основой для множественного наследования. Конструкции для надклассового программирования позволяют предоставить подпрограммы, которые применимы к "T'Class", или описать типы надклассовых ссылок (подобные описания будут точно адаптированы для любого типа в указанном классе наследования при конкретизации).

```
generic
  type T is tagged private;
package Module_G is
  type NT is new T  -- расширение типа
    with record
      B : Boolean;
    end record;

  function Equals (Left, Right : T'Class)  -- надклассовая подпрограмма
    return Boolean;

  type T_Poly_Ref is  -- тип надклассовой ссылки
    access T'Class;
end Module_G;
```

21.3.4 Производный тип как параметр настройки

Использование производного типа как формального параметра настройки подобно использованию в качестве формального параметра настройки настраиваемого модуля. В этом случае, формальный параметр описывается как тип, производный от указанного предка, подразумевая, что он совпадает с любым типом в классе наследования, корнем которого будет указанный предок. Следует учитывать, что синтаксис для тэговых и не тэговых типов различен.

Указание производного не тэгового типа в качестве формального параметра настройки может иметь следующий вид:

```
generic                  -- внутри Module_G, известными для T операциями являются
  type NT is new T;      -- операции для объектов типа NT с неявным
package Module_G is ...  -- преобразованием представления T в NT
```

Указание производного тэгового типа в качестве формального параметра настройки может иметь следующий вид:

```
generic             -- внутри Module_G, известные операции для T индицируют
  type NT is new T  -- диспетчеризуемые операции доступные для объектов типа NT
    with private;   -- причем, все вызовы используют NT-реализацию операций
package Module_G is ...
```

Формальный параметр производного типа осуществляет параметризацию настраиваемого модуля типом и его примитивными операциями не смотря на то, что указанный тип предка не известен. Подобным образом, в случае передачи настраиваемого абстрактного типа данных как формального параметра настройки (при использовании в качестве формального параметра настройки настраиваемого модуля), множество известных операций связывается вместе с фактическим типом.

В качестве примера рассмотрим тэговый абстрактный тип данных для рациональных чисел. Кто-либо может предусмотреть ввод/вывод таких чисел также как для любого другого типа, производного от этого абстрактного типа данных (например, кому-то может потребоваться оптимизация действий, выполняемых какими-либо знаками операций). Вариантом решения может служить настраиваемый пакет ввода/вывода, чьим формальным параметром настройки будет параметр производного типа, а абстрактный тип данных будет указан как тип предок. Тогда требуемые операции не будут нуждаться в перечислении, как дополнительные формальные параметры настройки, поскольку они определяются типом предка.

Спецификация абстрактного типа данных для рациональных чисел может иметь следующий вид:

```
package Rational_Numbers is
  type Rational_Type is tagged private;
  function To_Ratio (Numerator, Denominator : Integer)
    return Rational_Type;
  -- возбуждает Constraint_Error когда Denominator = 0

  function Numerator (Rational : Rational_Type) return Integer;
  function Denominator (Rational : Rational_Type) return Positive;
  -- результат не нормализован к множителю общих делителей

  ... -- другие операции: "+", "-", "*", "/", ...
private
  ...
end Rational_Numbers;
```

Спецификация настраиваемого пакета ввода/вывода для типов, производных от "Rational_Type":

```
with Rational_Numbers, Text_IO;
generic
  type Num_Type is
    new Rational_Numbers.Rational_Type with private;
package Rational_IO is
  procedure Get (File : in Text_IO.File_Type; Item : out Num_Type);
  procedure Put (File : in Text_IO.File_Type; Item : in Num_Type);
end Rational_IO;
```

Такой вид параметров настраиваемого модуля будет полезен при комбинировании абстракций и построении настраиваемых частей кода, которые будут адаптироваться к любому типу входящему в класс наследования, опираясь на базис свойств, которые применимы ко всем типам класса наследования (то есть известны для корневого типа класса наследования).

21.4 Построение абстракции путем композиции

Абстракция данных может быть построена с помощью композиции, когда новый абстрактный тип данных составляется из нескольких уже существующих абстрактных типов данных. В таком случае, новый тип данных описывается как запись (или тэговая запись) полями которой являются уже существующие типы данных. Функциональность нового типа данных обеспечивается путем перенаправления вызовов подпрограмм к соответствующим компонентам этого типа. Рассмотрим следующий пример:

```
with Lists;          use Lists;

package Queues is

    type Queue is private;

    procedure Remove_From_Head (Item : in out Queue; Value : out Integer);
    procedure Add_To_Tail (Item : in out Queue; Value : Integer);
```

```
    function Full (Item : Queue) return Boolean;
    function Empty (Item : Queue) return Boolean;

    function Init return Queue;

private
    -- очередь состоит из следующих элементов:

    type Queue is record
        Values         : List;
        No_Of_Elements : Integer;
    end record;
end Queues;



package body Queues is

    procedure Remove_From_Head (Item : in out Queue; Value : out Integer) is
    begin
        Remove_From_Head (Item.Values, Value);
        Item.No_Of_Elements := Item.No_Of_Elements - 1;
    end Remove_From_Head;


    procedure Add_To_Tail (Item : in out Queue; Value : Integer) is
    begin
        Add_To_Tail (Item.Values, Value);
        Item.No_Of_Elements := Item.No_Of_Elements + 1;
    end Add_To_Tail;


    procedure Reset (Item : in out Queue) is ...


    function Full (Item : Queue) return Boolean is
    begin
        return Full (Item.Values);
    end Full;

    function Empty (Item : Queue) return Boolean is
    begin
        return Item.No_Of_Elements = 0;
    end Empty;

end Queues;
```

В этом примере вызовы подпрограмм, обращенные к типу очереди "Queue", будут "перенаправлены"к компоненту список ("List"), на который возлагается ответственность за реализацию некоторых аспектов абстракции очереди (в данном случае — это основная часть).

21.5 Абстрагирование общей функциональности

Показанные ранее реализации стека, предусматривали общее множество подпрограмм, таких как "Push", "Pop", и т.д. В Аде, мы можем установить общность используя общий тип. Общий тип описывает какие должны быть предусмотрены подпрограммы и какие профили должны иметь эти подпрограммы. При этом, он не указывает как реализовать эти подпрограммы. Тогда, тип, производный от такого общего типа, вынужден представить свою собственную реализацию каждой подпрограммы. Такой общий тип называют абстрактным типом, и он является абстракцией абстракции. Смысл этого заключается в том, чтобы указать клиентам абстракции, что выбор конкретной реализации абстракции не имеет значения. Клиенты всегда получат, как минимум, ту функциональность, которая описана абстрактным типом. Рассмотрим следующий пример:

```
package Stacks is

    type Stack is abstract tagged null record;
    -- абстрактный тип, который не имеет полей, и не имеет "реальных" операций.
    -- Ада требует "tagged"-описание для абстрактных типов.

    Underflow : exception;
    Overflow  : exception;

    procedure Push (Item : in out Stack; Value : in Integer) is abstract;
    procedure Pop (Item : in out Stack; Value : out Integer) is abstract;

    function Full (Item : Stack) return Boolean is abstract;
    function Empty (Item : Stack) return Boolean is abstract;

    function Init return Stack is abstract;

end Stacks;
```

В данном случае, в пакете *Stacks* описан абстрактный тип стека "Stack". Причем, в этом пакете приводится только перечень подпрограмм, которые будут реализованы каким-либо типом, производным от абстрактного типа "Stack". Примечательно, что тип "Stack" может быть приватным:

```
package Stacks is

    type Stack is abstract tagged private;

    -- подпрограммы
        . . .

private

    type Stack is abstract tagged null record;

end Stacks;
```

В этом же пакете или, что более типично, в другом пакете, можно описать расширение типа "Stack" и создать производный тип, который реализует указанную функциональность абстрактного типа:

```
with Lists;
with Stacks;

package Unbounded_Stacks is

    type Unbounded_Stack is new Stacks.Stack with private;
    -- тип Unbounded_Stack, производный от пустой записи стек,
    -- с небольшим расширением, добавленным как описано в приватной
    -- секции

    procedure Push (Item : in out Unbounded_Stack; Value : in Integer);
    procedure Pop (Item : in out Unbounded_Stack; Value : out Integer);

    function Full (Item : Unbounded_Stack) return Boolean;
    function Empty (Item : Unbounded_Stack) return Boolean;

    -- возвращает пустой инициализированный Unbounded_Stack
    function Init return Unbounded_Stack;

private

    -- Расширяет пустую запись Stacks.Stack на одно поле.
    -- Все вызовы будут перенаправлены к внутреннему
    -- связанному списку.

    type Unbounded_Stack is new Stacks.Stack with
```

```
            record
                Values : Lists.List;
            end record;

    end Unbounded_Stacks;
```

В данном случае, для реализации функциональности, которая задана абстрактным типом "Stack", необходимо чтобы вызовы подпрограмм, обращенные к типу "Unbounded_Stack", были соответствующим образом перенаправлены к связанному списку. Таким образом, тело пакета *Unbounded_Stacks* будет иметь следующий вид:

```
package body Unbounded_Stacks is

    procedure Push (Item : in out Unbounded_Stack; Value : in Integer) is
    begin
        Lists.Insert_At_Head (Item.Values, Value);
    end Push;

    procedure Pop (Item : in out Unbounded_Stack; Value : out Integer) is
    begin
        Lists.Remove_From_Head (Item.Values, Value);
    exception
        when Lists.Underflow =>
            raise Stacks.Underflow;
    end Pop;

        . . .

end Unbounded_Stacks;
```

21.6 Многоуровневые абстракции

При построении новой абстракции из другой абстракции, новая абстракция может нуждаться в доступе к приватным описаниям уже существующей абстракции. Например, повсеместные "*widgets*", использующиеся для программирования в *X Window System*, имеют спецификации (такие как "*labels*"), которые зависят от их реализации в приватном представлении "*widgets*".

Описание абстрактного типа данных "Widget_Type" для *X Window System* может иметь следующий вид:

```
with X_Defs; use X_Defs;
package Xtk is
  type Widget_Type is tagged private;
  procedure Show (W : in Widget_Type);
private
  type Widget_Ref is access all Widget_Type'Class;
  type Widget_Type is
    record
      Parent         : Widget_Ref;
      Class_Name     : X_String;
      X, Y           : X_Position;
      Width, Height  : X_Dimension;
      Content        : X_Bitmap;
    end record;
end Xtk;
```

В данном случае, построение абстракции "Label_Type" поверх "Widget_Type" удобно осуществить с помощью создания дочернего модуля *Xtk.Labels*, который может выглядеть следующим образом:

```
with X_Defs; use X_Defs;
package Xtk.Labels is
  type Label_Type is new Widget_Type with private;
  procedure Show (L : in Label_Type);
  -- необходим доступ к приватным описаниям Xtk (например, позиция label)
private
```

```
   type Label_Type is new Widget_Type
     with record
       Label : X_String;
       Color : X_Color_Type;
     end record;
end Xtk.Labels;
```

Следует заметить, что в подобных случаях, иерархия модулей, как правило, параллельна иерархии абстракций представленных в этих модулях.

21.7 Комбинирование абстракций, множественное наследование

Множественное наследование подразумевает построение новой абстракции путем наследования свойств двух и более абстракций. Хотя в основе замысла множественного наследования лежит концепция классификаци, его реализация и использование значительно различается в разных языках программирования. В следствие этого множественное наследование чаще используется не с целью создания сложных иерархий типов данных, а как кустарное приспособление, которое скрадывает слабости языка программирования. В действительности, подлинное практическое использование множественного наследования встречается крайне редко.

Следует также заметить, что поддержка множественного наследования значительно усложняет язык программирования. Она требует введения различных дополнительных правил для "разборок"с патологическими случаями. Примером такого случая может служить повторяющееся наследование, то есть, ситуация, когда наследование выполняется от различных абстракций, которые имеют одного общего предка.

Основываясь на этом, Ада не предусматривает встроенных механизмов поддержки множественного наследования. Следует заметить, что это не является недостатком, поскольку задачи, которые требуют использование множественного наследования в других языках программирования, как правило, могут быть решены другими средствами Ады. Кроме того, Ада предусматривает средства с помощью которых можно построить множественное наследование "вручную".

В отличие от языков программирования для которых единственным используемым понятием абстракции данных является класс, а наследование является главенствующим механизмом комбинирования абстракций и управления пространством имен, при программировании на языке Ада нет необходимости использовать множественное наследование вне задач классификации. Например, сборку новых объектов из других, уже существующих объектов, легче осуществить путем композиции, а не путем использования множественного наследования. Руководство по хорошо известному объектно-ориентированному языку программирования предлагает описать тип "Apple_Pie" (яблочный пирог) путем наследования от типов "Apple" (яблоко) и "Cinnamon" (корица). Это типичный случай ошибочного использования множественного наследования: производный тип наследует все свойства своих предков, хотя не все эти свойства применимы к производному типу. В данном примере, бессмысленно производить "Apple_Pie" (яблочный пирог) от "Apple" (например, "Apple_Pie" — не растет на дереве и не имеет кожуры). Ада не требует использования множественного наследования для управления пространством имен. Это выполняется не зависимо от классов и достигается путем использования соответствующих спецификаторов контекста ("**with**" и/или "**use**").

Зависимость реализации, когда спецификация наследуется от одного типа, а реализация — от другого, может быть выражена без использования множественного наследования. Например, тип "Bounded_Stack" может быть построен путем наследования от абстрактного типа "Stack", который описывает интерфейс (операции "Push", "Pop", ...) и конкретного типа "**Array**", который будет предусматривать реализацию. Однако, такая форма множественного наследования будет правильной только в том случае, если все операции унаследованные от реализующего типа будут иметь смысл для производного типа. В этом случае, операции, которые предоставляют доступ к элементам массива, должны быть скрыты от пользователей типа "Bounded_Stack", поскольку только тот элемент, который находится на вершине стека, должен быть доступен клиентам. Вместо всего этого, в Аде, новая абстракция может использовать простое наследование для описания спецификации и обычную композицию для реализации.

21.7.1 Смешанное наследование

Смешанное наследование является формой множественного наследования при котором одна или две комбинированных абстракции выступают в роли шаблонов (*subpattern*). Любой подобный шаблон предоставляет

множество свойств (компонентов и операций) способных присутствовать в различных, иначе не связанных, типах, самостоятельное использование которых не достаточно для описания более чем одного абстрактного типа.

Построение шаблонов может быть выполнено без явного множественного наследования, путем использования одиночного наследования и средств предоставляемых настраиваемыми модулями:

- Шаблон абстрагируется в производную абстракцию посредством сборки множества свойств, то есть добавлением значений в расширение типа. Такой смешанный тип (*mixin type*) может быть использован для специализации общего поведения типа.

- Настраиваемый пакет используется как смешанный пакет (*mixin package*), который инкапсулирует смешанный тип. Этот пакет имеет формальный параметр настройки тэгового типа, который, в свою очередь, играет роль типа-предка к которому добавляются новые свойства.

Смешивание достигается при конкретизации смешанного пакета путем расширения фактического параметра тэгового типа производным абстрактным типом. При множественном наследовании, производная абстракция установится в обратном направлении иерархии наследования, как тип предок (абстрактный) от которого производится наследование. Этот способ называют смешанным наследованием (*mixin inheritance*).

Например, не смотря на множество свойств, которые описывают степень образованности, хорошо известно, что "образованность"сама по себе — не существует. Подобный шаблон может быть применен по отношению к людям, которые полностью закончили обучение в каком-либо учебном заведении, что и определяет степень (*degree*) образованности (*graduation*). Очевидно, что по отношению к людям, понятие "степень образованности"являться свойством, но самостоятельное использование понятия "степень образованности— бессмысленно. Свойства, которые описывают степень образованности, могут быть упакованы в смешанный пакет *Graduate_G*:

```
with Degrees; use Degrees;  -- экспортирует тип Degree

generic
    type Person is tagged private;

package Graduate_G is

    -- тип Graduate содержит Degree
    type Graduate is new Person with private;

    procedure Give (G: in out Graduate;
                    D: in     Degree);
    function Degree_of (G: Graduate) return Degree;

private

    type Graduate is new Person with
        record
            Given_Degree: Degree;
        end record;

end Graduate_G;
```

При конкретизации этого пакета с формальным параметром настройки тэгового типа "Woman", будет создан новый тип, производный от типа указанного при конкретизации. Этот новый тип будет обладать желаемыми свойствами:

```
package Graduate_Women is new Graduate_G (Woman);

Anne  : Graduate_Women.Graduate;  -- тип полученный в результате
                                  -- конкретизации Graduate_G
D     : Degrees.Degree :=...

Christen  (Somebody, "Anne");   -- операция типа Woman
Give      (Somebody, Degree);   -- операция типа Graduate
```

Свойства обоих типов, типа-предка "Woman" и смешанного типа "Graduate", будут доступны для нового типа "Graduate_Woman.Graduate".

Тип-предок для смешанного типа (тип формального параметра настройки) может быть ограничен принадлежностью к иерархии наследования какого-либо класса. Такое ограничение осуществляется для обеспечения гарантии того, что тип-предок обладает требуемым множеством свойств, которые необходимы для реализации смешанного типа, или для гарантирования логичности смешанного типа. Например, будет абсолютно бессмысленно создавать стек степени образованности.

Ограничение для типа предка может быть выражено указанием класса наследования к которому обязан принадлежать производный тип формального параметра настройки (вместо тэгового типа формального параметра настройки). Следующий пример демонстрирует, что только тип "Human" и его потомки могут быть использованы для конкретизации смешанного настраиваемого модуля:

```
with Degrees; use Degrees;

generic
    type Person is new Human with private;

package Graduate_G is

    type Graduate is new Person with private;

    ... -- описания Give и Degree_of те же, что и в предыдущем примере

    function Title_of (G: Graduate) return String;
            -- переопределение функции Title_of
            -- для получения требуемого заголовка

private
    ...
end Graduate_G;
```

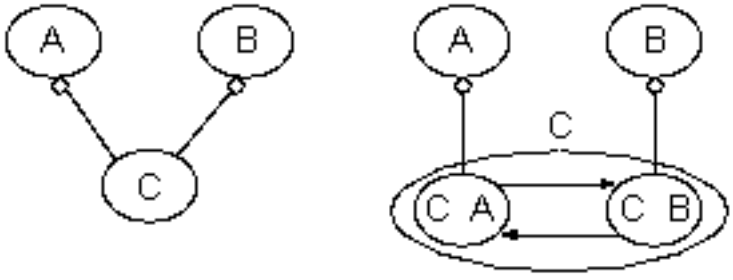
Поскольку свойства типа предка известны, то позже, в случае необходимости, ограниченный смешанный тип может быть переопределен. Так, например, функция "Title_of" переопределяется для предоставления более подходящей реализации.

21.7.2 Родственное наследование

В дополнение к показанному выше способу, можно использовать несколько специализированный способ родственного наследования (*sibling inheritance*), с применением ссылочных дискриминантов. Такой способ используется в случаях когда тип действительно является производным от более чем одного типа предка, или когда клиенты типа требуют представление этого типа как их предка или какого-либо потомка их типа предка.

Основная идея родственного наследования заключается в том, чтобы выразить эффект множественного наследования путем ретрансляции одиночного наследования. Обычно, это заключается в идее представления абстракции, которая осуществляет наследование, можно сказать, от двух суперабстракций, путем установки взаимосвязанных объектов.

В терминах типов, такая идея рассматривает концептуальный тип "C", производный от двух типов "A" и "B", как множество типов "C_A" и "C_B", соответственно производных от "A" и "B".



Объекты концептуального типа "C" создаются путем совместной генерации множества объектов типа "C_A" и типа "C_B". Такие объекты называют родственными объектами (*sibling objects*) или сдвоенными объектами (*twin objects*). Они взаимосвязаны между собой таким образом, что множество объектов может управляться так будто они являются одним объектом, и так, что один объект обладает доступом к свойствам другого сдвоенного объекта. Они также доступны индивидуально. Таким образом, легко доступно частичное представление множества, которое соответствует отдельному объекту.

В качестве примера, рассмотрим создание концептуального типа "Controlled_Humans", производного от стандартного типа "Controlled" и типа "Human". Компонент "First_Name" — это ссылочный тип, а не строка фиксированной длины. Таким образом, имя может иметь динамически изменяемую длину. В данном случае используются средства контролируемых объектов для освобождения памяти ассоциируемой с "First_Name" при разрушении объекта типа "Human". Спецификация типа "Human" может иметь подобный вид:

```
package Dynamic_Humanity is

    type String_Ptr is access String;
    type Human is tagged limited
        record
            First_Name: String_Ptr;
        end record;

    procedure Christen (H: in out Human; N: in String_Ptr);
    -- устанавливает имя для Human

    ...
end Dynamic_Humanity;
```

Для построения комбинации из этих двух типов, необходимо создать два родственных типа "Human_Sibling" и "Controlled_Sibling", путем соответственного производства от "Human" и "Limited_Controlled". Два этих типа, вместе формируют концептуальный тип "Controlled_Humans":

```
with Dynamic_Humanity, Ada.Finalization;

package Controlled_Human is

    type Human_Sibling;
      -- ввиду взаимозависимости типов, необходимо использование
      -- неполного описания типа


    type Controlled_Sibling (To_Human_Sibling: access Human_Sibling) is
        new Ada.Finalization.Limited_Controlled with null record;
      -- тип To_Human_Sibling является связкой с Human_Sibling

    procedure Finalize (C: in out Controlled_Sibling);


    type Human_Sibling is new Dynamic_Humanity.Human with
        record
            To_Controlled_Sibling: Controlled_Sibling (Human_Sibling'Access);
              -- To_Controlled_Sibling является связкой с Controlled_Sibling.
              -- Этот компонент автоматически инициализируется ссылкой
              -- на текущее значение экземпляра Human_Sibling
        end record;

    -- примитивные операции типа Human (могут быть переопределены)

end Controlled_Human;
```

Таким образом, эти типы — взаимосвязаны, и для обращения от одного сдвоенного объекта к другому:

- "Human_Sibling" обладает компонентом связки с "To_Controlled_Sibling", который подходит к типу "Controlled_Sibling".
- "Controlled_Sibling" обладает компонентом связки с "To_Human_Sibling", который подходит к типу "Human_Sibling".

Следует заметить, что эти связки имеют различную природу. Связка "To_Controlled_Sibling" — это связка членства: любой объект типа "Controlled_Sibling" заключен в каждый объект типа "Human_Sibling". Связка, "To_Human_Sibling" — это ссылочная связка: "To_Human_Sibling" обозначает "Human_Sibling".

В результате, существует возможность в любой момент получить представление любого из сдвоенных объектов посредством связки.

Хотя требуются некоторые дополнительные затраты (такие как, получение значения объекта, к которому отсылает ссылочное значение), такой подход обеспечивает всю функциональность, которая требуется от множественного наследования:

Последовательная генерация объектов Поскольку "To_Controlled_Sibling" — это компонент "Human_Sibling", любой объект типа "Controlled_Sibling" каждый раз создается автоматически при создании объекта типа "Human_Sibling". Связка "To_Controlled_Sibling" автоматически инициализируется ссылкой на заключенный объект, поскольку атрибут " 'Access применяется к имени типа записи, внутри описания, автоматически обозначая текущий экземпляр типа. Как результат, описание:

```
CH: Human_Sibling;
```

автоматически объявляет объект концептуального типа "Controlled_Human".

Переопределение операций Примитивные операции могут быть переопределены тем сдвоенным типом который их реализует (например, "Finalize"). Например, для предотвращения утечки памяти, мы переопределяем "Finalize" для автоматической очистки памяти, используемой "First_Name", когда "Human" становится не доступным:

```
package body Controlled_Human is

    procedure Free is new Unchecked_Deallocation (String_Ptr);

    procedure Finalize (C: in out Controlled_Sibling) is
    -- переопределяет Finalize унаследованное от Controlled
    begin
        Free (C.To_Human_Sibling.all.First_Name);
    end Finalize;

end Controlled_Human;
```

Доступ ко всем компонентам и операциям Компоненты каждого сдвоенного объекта могут быть выбраны и использованы любым из сдвоенных объектов, используя связки. Например, операция "Finalize" (описанная для "Controlled_Sibling") может использовать компонент "First_Name" (описанный для "Human_Sibling").

Примитивные операции могут быть вызваны тем же самым способом.

Добавление свойств К концептуальному типу могут быть легко добавлены новые свойства, компоненты и операции, путем расширения любого из сдвоенных объектов. Для сохранения инкапсуляции, сдвоенные типы могут быть также описаны в приватной части расширения.

Расширение типа Концептуальный тип может быть использован как предок для других типов. Это важно при производстве от сдвоенного типа "Human_Sibling", который содержит компоненты другого сдвоенного типа. Свойства другого сдвоенного типа также доступны для производства через связку "To_Human_Sibling".

Проверка принадлежности Проверка принадлежности объекта к любому из типов предков выполняется путем использования согласованного (надклассового) представления любого из сдвоенных объектов:

```
declare

    CH: Human_Sibling;   -- совместная генерация объектов

begin
```

```
        ... CH in Human'Class                ... -- TRUE
        ... CH.To_Controlled_Sibling.all
              in Limited_Controlled'Class    ... -- TRUE
    end;
```

Присваивание Любой объект концептуального типа может быть присвоен объекту обоих типов предков обычным образом, то есть, путем выбора в концептуальном объекте сдвоенного объекта, который совпадает по типу с требуемым предком (используя преобразование представления), и присваивая этот сдвоенный объект назначению операции присваивания.

Такая модель может быть легко расширена для управления множественным наследованием от более чем двух типов.

21.8 Пример программирования посредством расширения

Программирование путем расширения подразумевает, что при необходимости добавления в программную систему каких-либо новых свойств она может быть выполнена простым добавлением нового кода, без необходимости внесения изменений в уже существующие модули программного обеспечения. При этом, поддержка обеспечиваемая со стороны языка программирования является очень полезной.

Представим себе функцию, которая читает до 20 сообщений с какого-либо интерфейса, обрабатывает статистику и генерирует какой-либо итоговый результат. Для остальной части системы непосредственно необходим только итоговый результат, и ее не заботят входные сообщения или работа интерфейса. В таком случае, простым решением будет помещение в самостоятельный пакет функции "Summarizer", вычисляющей итоговый результат, вместе с перечислением переменных, которые указывают этой функции на то, какой интерфейс необходимо использовать:

```
package Simple_Approach is
    type Interfaces is (Disk_File, Serial_Interface, ...);

    type Summary_Type is
        record
            . . .
        end record;

    function Summarizer (Interface_To_Use: Interfaces)
        return Summary_Type;

end Simple_Approach;
```

Естественно, что в результате такого простого решения, при необходимоти добавить какой-либо новый интерфейс потребуется переписать заново тип "Interfaces" и функцию "Summarizer", после чего, перекомпилировать все модули которые указывают пакет "Simple_Approach" в спецификаторе "**with**".

Необходимо заметить, что Ада очень часто используется в жизненно-критических системах (например, в авиационном оборудовании), где каждый программный модуль должен пройти длительный и дорогостоящий цикл тестирования скомпилированного объектного кода. Следовательно, после перекомпиляции модуля этот цикл тестирования должен быть пройден заново даже в случае отсутствия изменений в исходном тексте.

Программирование путем расширения позволяет избавиться от подобных переписываний и перекомпиляций. Первым вопросом, в этом случае, является: "что необходимо расширять?". В этом случае, необходима возможность расширения операций по получению сообщений от интерфейса. Таким образом, следует создать процедуру "**Get**", которая получает сообщение от интерфейса основываясь на тэговом типе, что обеспечит ее полиморфность:

```
with Message;
package Generic_Interface is

    type Flag is tagged null record;

    procedure Get (Which_Interface: Flag;
                   Data: Message.Data_Type);

end Generic_Interface;
```

Теперь можно написать функцию "Summarizer" как надклассовую, которая, для получения сообщений от различных интерфейсов, использует соответствующую диспетчеризацию:

```
with Generic_Interface;
package Extension_Approach is

   function Summarizer
      (Interface_To_Use: Generic_Interface.Flag'Class)
      return Summary_Type;

end Extension_Approach;
```

```
with Messages;
package body Extension_Approach is

   function Summarizer
      (Interface_To_Use: Generic_Interface.Flag'Class)
      return Summary_Type
   is
      Data: array (1..20) of Message.Data_Type;
   begin
      for I in 1 .. 20 loop
         Get (Interface_To_Use, Data (I));
         exit when Data (I).Last_Message;
      end loop;
      ...
```

Тело процедуры "**Get**", в пакете *Generic_Interface*, может возвращать исключение, или получать сообщение от интерфейса по умолчанию (в этом случае пакет, наверное, должен быть назван *Default_Interface*).

Для расширения "Summarizer" с целью получения сообщений от какого-либо нового интерфейса, необходимо просто расширить тип "Generic_Interface.Flag" и переопределить его процедуру "**Get**":

```
with Message;
with Generic_Interface;
package Disk_Interface is

    type Flag is new Generic_Interface.Flag with null record;

    procedure Get (Which_Interface: Flag;
                   Data: Message.Data_Type);

end Disk_Interface;
```

В результате этого, тип "Flag" способен хранить различные фактические данные для любых случаев.

Теперь мы можем обеспечить, чтобы "Summarizer" получал сообщения от диска и возвращал нам итоговый результат:

```
Summary := Extension_Approach.Summarizer
              (Disk_Interface.Flag'(null record));
```

Благодаря этому, даже в случае написания "Disk_Interface" после "Summarizer", не возникает нужды переписывать "Summarizer".

Затратив несколько больше усилий, можно запрограммировать почти все пользовательские системы так, что они не будут нуждаться в перепрограммировании или даже перекомпиляции для работы с любым новым интерфейсом. Сначала необходимо добавить описания к "Generic_Interface", что позволяет пользователю сохранить переменную флага "Flag":

```
    type Interface_Selector is access constant Flag'Class;
end Generic_Interface;
```

Константа "Selected" помещается в каждый пакет интерфейса, избавляя пользователя от необходимости использования синтаксиса, имеющего вид "TYPENAME'(**null record**)". Кроме того, в каждый пакет интерфейса

помещается ссылочная константа для обозначения "Selected", которая может быть сохранена в переменной типа "Interface_Selector".

```
        Selected: constant Flag := Flag'(null record);

        Selection: constant Generic_Interface.Interface_Selector
            := Selected'Access;

    end Disk_Interface;
```

Следует заметить, что эти добавления к пакетам "_Interface" являются простыми соглашениями. Пользовательский код может осуществлять такие описания самостоятельно.

Теперь, для точного указания используемого интерфейса, код может содержать:

```
    with Disk_Interface;
    with Extension_Approach;      use Extension_Approach;

    procedure User is
        Sum: Summary_Type;
    begin
        Sum := Summarizer (Disk_Interface.Selected);
```

Для инкапсуляции сведений об интерфейсах в одиночном пакете, код может иметь следующий вид:

```
    package Interfaces is
       Current: Generic_Interface.Interface_Selector;
    end Interfaces;
```

```
    with Interfaces;
    with Extension_Approach;      use Extension_Approach;
    procedure User is
        Sum: Summary_Type;
    begin
        Sum := Summarizer (Interfaces.Current);
    . . .
```

Теперь, для расширения процедуры "User", необходимо добавить еще один класс и немного кода (возможно, помещенного в интерфейс пользователя), который потребуется изменить так, чтобы он мог установить "Interfaces.Current" в "New_Interface.Selection". Все остальные части системы не будут нуждаться в изменении и даже перекомпиляции.

Еще одна особенность программирования посредством расширения в Ada95 заключается в том, что при производстве нового типа, производного от тэгового, можно повторно использовать подпрограммы типа-предка, если они подходят для использования с производным типом. Например, предположим, что некоторые интерфейсы должны явно запускаться и останавливаться. Тогда код может иметь следующий вид:

```
    with Message;
    package Generic_Interface is
        type Flag is tagged null record;

        procedure Get (Which_Interface: Flag;
                       Data: Message.Data_Type);

        procedure Start (Which_Interface: Flag);  -- ничего не выполняет
        procedure Stop  (Which_Interface: Flag);  -- ничего не выполняет
        . . .
```

```
    with Message;
    package Disk_Interface is
        type Flag is new Generic_Interface.Flag with null record;

        procedure Get (Which_Interface: Flag;
```

```
                    Data: Message.Data_Type);

    -- нам нет нужды запускать или останавливать диск, таким образом, Start
    -- ничего не наследует от Generic_Interface
    . . .
```

```
with Message;
package Serial_Interface is
    type Flag is new Generic_Interface.Flag with null record;

    procedure Get (Which_Interface: Flag;
                   Data: Message.Data_Type);

    -- запуск и остановка последовательного интерфейса
    procedure Start (Which_Interface: Flag);
    procedure Stop  (Which_Interface: Flag);
    . . .
```

При этом код пользователя может иметь вид подобный следующему:

```
with Interfaces;
with Extension_Approach;      use Extension_Approach;
procedure User is
   Sum: Summary_Type;
begin
   Start (Interfaces.Current);
   Sum := Summarizer (Interfaces.Current);
   Stop (Interfaces.Current);
. . .
```

Рассмотренный пример демонстрирует преимущества средств Ada95 для программирования посредством расширения. Он представляет своеобразный вид "бесконечного варианта"для получения которого были использованы тэговые типы. Очевидно, что этот пример несколько искуственный. Следовательно этот пример может быть не полностью корректен для порождения реального кода.

Глава 22

Контекст, видимость и подсистемы

22.1 Контекст и видимость

Ошибочное понимание разницы между контекстом (*scope*) и видимостью (*visibility*) может вызвать трудности у многих программистов, которые знакомятся с языком программирования Ада. Следует учесть, что понимание этого различия имеет важное значение при использовании Ады для разработки программного обеспечения. Спецификатор совместности контекста "**with**" помещает соответствующий библиотечный модуль в контекст, однако, ни один ресурс этого модуля не становится непосредственно видимым для клиента. Это является основным различием от директивы "**#include**" семейства языков *C* или директивы "uses" в реализации Паскаль-системы фирмы *Borland*. Ада предоставляет множество способов обеспечения непосредственной видимости различных ресурсов, после того как они помещены в контекст. Разделение контекста и видимости является важным концептом разработки программного обеспечения. Однако, следует заметить, что этот концепт редко реализовывается в различных языках программирования. Примечательно также, что идея пространства имен *namespace*, принятая в *C++*, подобна концепции контекста и видимости Ады.

Каждая инструкция и/или каждая конструкция языка имеет объемлющий контекст. Обычно, контекст легко увидеть в исходном тексте, поскольку контекст имеет какую-либо точку входа (описательная часть "**declare**", идентификатор подпрограммы, идентификатор составного типа, идентификатор пакета...) и какую-либо явную точку завершения. Как правило, явная точка завершения контекста обозначается с помощью соответствующей инструкции "**end**". Таким образом, как только мы обнаруживаем в исходном тексте какую-либо инструкцию "**end**", мы понимаем, что она является точкой завершения какого-нибудь контекста. Контекст может быть вложенным. Например, одна процедура может быть описана внутри другой процедуры.

Несколько иначе выглядит ситуация, когда спецификатор совместности контекста "**with**" помещает в контекст какой-либо библиотечный модуль. В этом случае в контекст помещаются все ресурсы указанного библиотечного модуля, но при этом, спецификатор "**with**" не делает эти ресурсы непосредственно видимыми.

В Аде, какой-либо ресурс может находиться в контексте (*scope*) и при этом он не будет непосредственно видимым. Эта концепция больше характерна для Ады чем для других хорошо известных языков программирования. Для того чтобы соответствующие ресурсы были непосредственно видимыми, необходимо использовать соответствующие средства:

спецификатор использования "**use**" — делает непосредственно видимыми все публично доступные ресурсы пакета.

спецификатор использования "**use type**" — делает непосредственно видимыми все публично доступные знаки операций для указанного типа.

полная точечная нотация — ресурс указанный с помощью полной точечной нотации становится непосредственно видимым

локальное переименование операций и знаков операций — обычно, является лучшим средством обеспечения непосредственной видимости для операций и знаков операций

В процессе разработки, сообщения об ошибках, получаемые от Ада-компилятора, могут указывать, что некоторые ресурсы не видимы в том месте где они используются. Следует заметить, что подобные проблемы

видимости наиболее часто возникают со знаками операций. Для того, чтобы сделать ресурс непосредственно видимым, можно использовать какой-либо из перечисленных выше способов.

22.2 Управление видимостью

При построении большой модульной системы, существует необходимость в определении того, что должно быть видимым и того, что не должно быть видимым за пределами какого-либо модуля (пакет, тип или подсистема). При этом могут существовать различные промежуточные уровни видимости.

Предположим, что модуль является компонентом какой-либо большой системы. В этом случае, ресурсы данного модуля могут быть не доступны вне системы. Внутри системы, ресурсы модуля могут быть видимы для остальных частей общей системы, но допуская при этом частичную видимость, которая предоставляет некоторые компоненты или операции модуля только для выбранной группы модулей системы.

Необходимо обратить внимание на то, что данные проблемы подобны, вне зависимости от того - рассматриваем-ли мы типы, пакеты или классы наследования. Таким образом, некоторые свойства могут быть:

публичные (*public*), то есть видимые всем

приватные (*private*), то есть видимые, но не для всех, а только для определенного круга "друзей", или наследников и дочерних модулей

внутренние (*internal*), то есть не видимы ни для кого, включая дочерние модули

В качестве примера, можно привести следующую аналогию: то что на фасаде дома — то видно всем (*public*), то что внутри дома — то видно только членам семьи (*private*), то что в чьем-либо теле — не видно никому (*internal*).

Благодаря существованию трех уровням видимости, существуют три уровня управления:

1. публичные (*public*) свойства описываются в интерфейсе (спецификация пакета)
2. приватные (*private*) свойства описываются в спрятанной части интерфейса (приватная часть спецификации пакета)
3. внутренние (*internal*) свойства описываются в части реализации (тело пакета)

Следует заметить, что различие между приватными и внутренними описаниями не требовалось в Ada83. Необходимость в таком различии появилась в Ada95, поскольку появилась новая концепция — концепция дочерних модулей, и сопутствующая ей необходимость в управлении уровнями видимости.

22.3 Подсистемы

Большие системы обычно состоят из сотен и даже тысяч самостоятельных компонентов. Обычно не требуется, чтобы каждый отдельный компонент активно взаимодействовал со всеми остальными частями системы. Таким образом, отдельные компоненты собираются в группы или кластеры модулей, которые интенсивно взаимодействуют между собой. При этом, взаимодействие отдельного компонента с остальной системой не так активно или вообще отсутствует. Подобные группы компонентов называют подсистемами (*subsystem*), и они могут собираться, управляться и обслуживаться изолировано от остальной системы.

Для поддержки построения и облегчения сопровождаемости больших систем, Ada95 предоставляет концепцию дочерних библиотечных модулей. Любой вид библиотечных модулей (подпрограммы, пакеты...) может быть описан как дочерний модуль библиотечного пакета (или настраиваемого модуля). Это позволяет строить иерархии библиотечных модулей. Каждый дочерний модуль (потомок) имеет свою приватную часть и тело, и способен "видеть"приватные описания своего предка, что аналогично текстовой вставке спецификации дочерних модулей в спецификацию пакетов родителей, предоставляя возможность раздельной компиляции их спецификаций и обеспечивая большую гибкость в управлении видимостью.

Подсистема (*subsystem*) — это иерархия библиотечных модулей. Она может состоять из подпрограмм, но чаще всего она состоит из иерархии пакетов. Подсистема может быть настраиваемой, в таком случае все потомки (дочерние модули) также должны быть настраиваемыми. Подсистемой можно манипулировать как

тесно интегрированной сборкой отдельных компонентов, так и как единым целым. Спецификация всей подсистемы — это иерархия спецификаций ее публичных модулей.

Дочерний модуль может быть приватным модулем. Таким образом, его использование будет ограничено телами модулей той же самой иерархии, и будет доступно для спецификаций и тел других приватных модулей той же самой иерархии. Это является дополнительным уровнем управления для обеспечения возможности изоляции внутренних свойств подсистемы в спрятанных модулях, которые известны и совместно используются только внутри иерархически связанной группы библиотечных модулей.

Приватный потомок может быть добавлен к другому потомку, или даже удален из подсистемы, никак не затрагивая общую спецификацию подсистемы. В результате этого, при модификации приватного дочернего модуля, клиенты подсистемы не будут нуждаться в перекомпиляции. Хотя приватный дочерний модуль является внутренним (он не может быть сделан доступным через приватную часть других дочерних модулей), он предусматривает дополнительный уровень управления видимостью, поскольку он используется в телах модулей подсистемы.

Разделение компонентов подсистемы на модули предков и их потомков позволяет осуществлять изменения дочерних модулей без необходимости перекомпиляции модулей предков или независимых родственных дочерних модулей. Таким образом, есть возможность управлять тем, как много компонентов системы зависит от отдельных аспектов самой системы. Как результат, перекомпиляция может быть ограничена подмножеством перекомпилируемых модулей подсистемы, хотя внешне подсистема будет рассматриваться и выглядеть как одиночный библиотечный модуль.

Использование иерархических модулей предусматривает улучшенное управление пространством имен. Предположим, что кто-то приобрел два семейства программных компонентов. Одно, у *Mr. Usable*, который написал пакет *Strings*, для строковой обработки. Другое, у *Mr. Reusable*, которое также включает пакет с именем *Strings*. В такой ситуации, при одновременном использовании обеих семейств программных компонентов, возникает коллизия имен в пространстве имен библиотеки. С другой стороны, если для семейств компонентов предусмотрены префиксы *Usable.Strings* и *Reusable.Strings*, то коллизии имен не происходит. Это дает нам возможность сконцентрировать свои усилия на управлении пространством имен для подсистем или семейств компонентов вместо управления плоским пространством имен для всех компилируемых модулей.

Такое структурирование пространства имен можно обнаружить в предопределенном окружении Ады: теперь, все предопределенные пакеты являются дочерними модулями пакетов *Ada*, *System* и *Interfaces* (с соответствующими библиотечными переименованиями для поддержки обратной совместимости с Ada83).

Предположим, что существует сложная абстракция, которая инкапсулирована в единую подсистему. Кто-либо может предусмотреть различное специализированное представление такой абстракции посредством различных интерфейсных пакетов подсистемы. Такой подход похож на представление в базе данных.

Например, простой банковский счет может иметь *balance* (остаток на счете) и *interest_rate* (процентная ставка). Представление заказчика заключается в том, что он может выполнить *deposit* (положить на счет) или *withdraw* (снять со счета), и посмотреть состояние *interest_rate*. Представление клерка в банке заключается в том, что он может изменить *interest_rate* для любого банковский счета. Очевидно, что два таких представления не должны быть объединены в единый интерфейс (банк не позволяет заказчикам изменять *interest_rate*). Таким образом, легко определить когда код написанный для представления заказчика незаконно использует код предназначенный для представления клерка.

Рассмотрим пример операционной системы (например *POSIX*). Очевидно, что никто не желает иметь один большой пакет с сотнями типов и операций. Должно существовать логическое группирование типов и операций. В Ada95 фактически, некоторые приватные типы, которые должны быть совместно используемы, больше не являются помехой для подобного разделения операций, благодаря видимости приватных описаний, которая обеспечена предками для дочерних модулей. Иерархические модули позволяют логическое структурировани и группирование, вне зависимости от используемых приватных описаний типов.

Подсистема может иметь маленький и простой интерфейс, и при этом быть очень сложной и иметь большую реализацию. Как только спецификация подсистемы (спецификации одного или более пакетов) описана, одна группа разработчиков может использовать подсистему пока другая (контрагент) — будет работать над ее реализацией.

Предположим, что существуют две подсистемы. Например, *Radar* и *Motor*. Между разными группами разработчиков, при разработке своих подсистем, не будет возникать коллизия имен "вспомогательных пакетов". Например, для *Radar.Control* и *Motor.Control*. Таким образом, для разных групп разработчиков упрощаются

описания подсистем и облегчается разработка, а также значительно облегчаются (или полностью исчезают) проблемы общей интеграции.

Глава 23

Элаборация

Процесс элаборации является процессом предварительной подготовки, который осуществляется непосредственно перед началом выполнения головной программы, тела инструкции блока или подпрограммы. Процесс элаборации также называют процессом выполнения описаний.

Следует заметить, что в других языках программирования подобные подготовительные действия, осуществляемые непосредственно перед запуском программы на выполнение, явно не определяются, а последовательность их выполнения неявно продиктована последовательностью компиляции и сборки приложения.

23.1 Код элаборации

Стандарт Ada95 предусматривает общие правила выполнения кода в процессе элаборации (иначе, кода элаборации). Обработка кода элаборации осуществляется в трех контекстах:

Инициализация переменных — Переменные, описанные на уровне библиотеки в спецификациях или телах пакетов, могут требовать инициализацию, которая выполняется в процессе элаборации:

```
Sqrt_Half : Float := Sqrt (0.5);
```

Инициализация пакетов — Код в секции "**begin** ... **end**", на внешнем уровне тела пакета, является кодом инициализации пакета, и выполняется как часть кода элаборации тела пакета.

Аллокаторы задач уровня библиотеки — Задачи, которые описаны с помощью аллокаторов на уровне библиотеки, начинают выполняться немедленно и, следовательно, могут выполняться в процессе элаборации.

Любой из этих контекстов допускает вызов подпрограмм. Это подразумевает, что любая часть программы может быть выполнена как часть кода элаборации. Более того, существует возможность написания программы, вся работа которой осуществляется в процессе элаборации, при этом, головная прорамма — пуста. Следует однако заметить, что подобный способ структурирования программы считается стилистически неверным.

В отношении кода элаборации существует один важный аспект: необходима уверенность в том, что выполнение кода элаборации осуществляется в соответствующем порядке. Программа обладает множеством секций кода элаборации (потенциально, каждый модуль программы содержит одну секцию кода элаборации). Следовательно, важна правильность последовательности, в которой осуществляется выполнение этих секций. Для примера описания переменной "Sqrt_Half", который показан выше, это подразумевает, что если другая секция кода элаборации использует "Sqrt_Half", то она должна выполняться после секции кода элаборации, который содержит описание переменной "Sqrt_Half".

Порядок элаборации не вызывает проблем, когда соблюдается следующее правило: элаборация спецификации и тела, каждого указанного в спецификаторе "**with**" модуля, осуществляется перед элаборацией модуля, который содержит такой спецификатор "**with**".
Например:

```
with Unit_1;
package Unit_2 is ...
```

В данном случае необходимо, чтобы элаборация спецификации и тела модуля *Unit_1* осуществлялась перед спецификацией модуля *Unit_2*. Однако, такое правило накладывает достаточно жесткие ограничения. В частности, это делает невозможным наличие взаимно рекурсивных подпрограмм, которые располагаются в разных пакетах.

Можно предположить, что достаточно "умный"компилятор способен проанализировать код элаборации и определить правильный порядок элаборации, но в общем случае это не возможно. Рассмотрим следующий пример.

Тело модуля *Unit_1* содержит подпрограмму "Func_1", которая использует переменную "Sqrt_1" описанную в коде элаборации тела *Unit_1*:

```
Sqrt_1 : Float := Sqrt (0.1);
```

Код элаборации тела модуля *Unit_1* также содержит:

```
if expression_1 = 1 then
   Q := Unit_2.Func_2;
end if;
```

Существует также модуль *Unit_2*, структура которого аналогична: он содержит подпрограмму "Func_2", которая использует переменную "Sqrt_2" описанную в коде элаборации тела *Unit_2*:

```
Sqrt_2 : Float := Sqrt (0.1);
```

Код элаборации тела модуля *Unit_2* также содержит:

```
if expression_2 = 2 then
   Q := Unit_1.Func_1;
end if;
```

Для показанных фрагментов кода суть вопроса заключена в выборе правильного порядка элаборации, то есть выбрать последовательность вида:

```
Spec of Unit_1
Spec of Unit_2
Body of Unit_1
Body of Unit_2
```

или последовательность вида:

```
Spec of Unit_2
Spec of Unit_1
Body of Unit_2
Body of Unit_1
```

При внимательном рассмотрении можно обнаружить, что ответить на этот вопрос во время компиляции не возможно. Если выражение "expression_1" не равно 1, а выражение "expression_2" не равно 2, то можно использовать любую из показанных выше последовательностей элаборации, поскольку отсутствуют вызовы функций. В случае, когда результаты обеих проверок истинны ("**true**"), не может быть использована ни одна из показанных последовательностей элаборации.

Если результат проверки одного условия истина ("**true**"), а другого — ложь ("**false**"), то одна из показанных последовательностей элаборации будет справедлива, а другая — нет. Например, если "expression_1 = 1" и "expression_2 /= 2", то присутствует вызов "Func_2", но нет вызова "Func_1". Это подразумевает, справедливость выполнения элаборации тела модуля *Unit_1* перед элаборацией тела *Unit_2*, то есть первая последовательность элаборации правильна, а вторая — нет.

Если выражения "expression_1" и "expression_2" зависят от вводимых данных, то компилятор и/или редактор связей будут не в состоянии определить какое из условий будет истинным. Таким образом, для данного случая не возможно гарантировать правильность порядка элаборации во время выполнения программы.

23.2 Проверка порядка элаборации

В некоторых языках программирования, которые подразумевают наличие аналогичных проблем элаборации (например Java и C++), программист вынужден заботиться о порядке элаборации самостоятельно. В следствие этого, часто встречаются случаи написания программ в которых выбор неправильного порядка элаборации приводит к неожиданным результатам, поскольку при этом используются переменные, значения которых не инициализированы. Язык Ада был разработан как надежный язык и заботы программиста о порядке элаборации не так значительны. В результате, язык предусматривает три уровня защиты:

Стандартные правила элаборации накладывают некоторые ограничения на выбор порядка элаборации. В частности, если какой-либо модуль указан в спецификаторе "**with**", то элаборация его спецификации всегда осуществляется перед элаборацией модуля, который содержит этот спецификатор "**with**". Подобным образом, элаборация спецификации модуля-предка всегда осуществляется перед элаборацией спецификации модуля-потомка, и, в заключение, элаборация спецификации всегда осуществляется перед элаборацией соответствующего тела.

Динамические проверки элаборации осуществляются во время выполнения программы. Таким образом, если доступ к какой-либо сущности осуществляется до ее элаборации (обычно это происходит при вызовах подпрограмм), то возбуждается исключение *Program_Error*.

Средства управления элаборацией позволяют программисту явно указывать необходимый порядок элаборации.

Рассмотрим перечисленные возможности более детально. Во-первых, правила динамической проверки. Одно из правил заключается в том, что при попытке использования переменной, элаборация которой не была выполнена, возбуждается исключение *Program_Error*. Недостатком такого подхода является то, что затраты производительности на проверку каждой переменной могут оказаться достаточно дорогостоящими. Вместо этого, Ada95 поддерживает два несколько более строгих правила, соблюдение которых легко проверить:

Ограничения для вызовов подпрограмм — Какая-либо подпрограмма может быть вызвана в процессе элаборации только в случае, когда осуществлена элаборация тела этой подпрограммы. Такое правило гарантирует, что элаборация спецификации подпрограммы будет выполнена перед вызовом подпрограммы, а не перед элаборацией тела. При нарушении этого правила возбуждается исключение *Program_Error*.

Ограничения для конкретизации настраиваемых модулей — Настраиваемый модуль может быть конкретизирован только после после осуществления элаборации тела настраиваемого модуля. Такое правило гарантирует, что элаборация спецификации настраиваемого модуля будет выполнена перед конкретизацией настраиваемого модуля, а не перед элаборацией тела. При нарушении этого правила возбуждается исключение *Program_Error*.

Смысл идеи заключается в том, что при осуществлении элаборации тела элаборация любых переменных, которые используются в этом теле, должна быть уже выполнена. Таким образом, при проверке элаборации тела гарантируется, что ни один из ресурсов, которые использует это тело, не окажется источником каких-либо неожиданностей. Как было замечено ранее, такие правила несколько более строгие, поскольку необходима гарантия корректности вызова подпрограммы, которая использует в своем теле не локальные для этой подпрограммы ресурсы. Однако, полностью полагаться на эти правила будет не безопасно, поскольку это подразумевает, что вызывающая подпрограмма должна заботится о деталях реализации, которые размещенны в теле, что противоречит основным принципам Ады.

Вероятная реализация такой идеи может выглядеть следующим образом. С каждой подпрограммой и с каждым настраиваемым модулем может быть ассоциирована логическая переменная. Первоначально такая переменная имеет значение "ложь"("**False**"), а после осуществления элаборации тела эта переменная устанавливается в значение "истина"("**True**"). Проверка значения этой переменной выполняется при осуществлении вызова или конкретизации, и когда значение переменной — "ложь"("**False**"), возбуждается исключение *Program_Error*.

23.3 Управление порядком элаборации

Выше мы обсудили правила согласно которых возбуждается исключение *Program_Error*, когда выбран неправильный порядок элаборации. Это предотвращает ошибочное выполнение программы, но Ада также предоставляет механизмы, которые позволяют явно указавать необходимый порядок элаборации и избежать возбуждение исключения. Следует заметить, что при разработке небольших программных проектов, как правило, нет необходимости вмешиваться в последовательность элаборации множества программных модулей. Однако, при разработке больших систем, такое вмешательство бывает необходимым.

Во-первых, существует несколько способов указать компилятору, что данный модуль потенциально не содержит никаких проблем связанных с элаборацией:

Пакеты, которые не требуют наличия тела.

Ада не допускает наличие тела для библиотечного пакета, который не требует наличия тела. Это подразумевает, что можно иметь пакет подобный следующему:

```
package Definitions is
   generic
      type m is new integer;
   package Subp is
      type a is array (1 .. 10) of m;
      type b is array (1 .. 20) of m;
   end Subp;
end Definitions;
```

В данном случае, пакет, который указывает в спецификаторе "**with**" пакет *Definitions*, может безопасно конкретизировать пакет *Definitions.Subp* поскольку компилятор способен определить очевидное отсутствие тела

Директива компилятора "**pragma** Pure"

Данная директива накладывает строгие ограничения на модуль, гарантируя, что обращение к любой подпрограмме модуля не повлечет за собой никаких проблем элаборации. Это подразумевает, что компилятору нет нужды заботиться о порядке элаборации подобного модуля, в частности, нет необходимости осуществлять проверку обращений к подпрограммам в этом модуле.

Директива компилятора "**pragma** Preelaborate"

Данная директива накладывает несколько менее строгие ограничения на модуль чем директива "Pure". Однако эти ограничения остаются достаточно значительными, чтобы гарантировать отсутствие проблем при вызове подпрограмм модуля.

Директива компилятора "**pragma** Elaborate_Body"

Данная директива требует, чтобы элаборация тела модуля была осуществлена сразу после элаборации спецификации. Предположим, что модуль *A* содержит данную директиву, а модуль *B* указывает в спецификаторе "**with**" модуль *A*. Напомним, что стандартные правила требуют, чтобы элаборация спецификации модуля *A* была выполнена перед элаборацией модуля, который указывает модуль *A* в спецификаторе "**with**". Указание этой директивы компилятора в модуле *A* говорит о том, что элаборация тела модуля *A* будет выполнена перед элаборацией *B*. Таким образом, обращения к модулю *A* будут безопасны и не нуждаются в дополнительных проверках.

Примечательно, что в отличие от директив "Pure" и "Preelaborate", использование директивы "Elaborate_Body" не гарантирует, что программа свободна от проблем связанных с элаборацией, поскольку возможно наличие ситуации, которая не удовлетворяет требования порядка элаборации. Вернемся к примеру с модулями *Unit_1* и *Unit_2*. Если поместить директиву "Elaborate_Body" в *Unit_1*, а модуль *Unit_2* оставить без изменений, то порядок элаборации будет следующий:

```
Spec of Unit_2
Spec of Unit_1
Body of Unit_1
Body of Unit_2
```

В этом случае подразумевается, что нет необходимости проверять вызов "Func_1" из модуля *Unit_2* поскольку он должен быть безопасным. Вызов "Func_2" из модуля *Unit_1*, когда результат вычисления выражения "Expression_1" равен 1, может быть ошибочным. Следовательно, ответственность за отсутствие подобной ситуации возлагается на программиста.

Когда все модули содержат директиву компилятора "Elaborate_Body", то проблемы порядка элаборации отсутствуют, кроме случаев когда вызовы подпрограмм осуществляются из тела, забота о котором в любом случае возлагается на программиста. Следует заметить, что повсеместное использование этой директивы не всегда возможно. В частности, для показанного ранее примера с модулями *Unit_1* и *Unit_2*, когда оба модуля содержат директиву "Elaborate_Body", отсутствие возможности определить правильный порядок элаборации очевидно.

Показанные выше директивы компилятора позволяют гарантировать клиентам безопасное использование серверов, и такой подход является предпочтительным. Следовательно, маркирование модулей как "Pure" или "Preelaborate", если это возможно, является хорошим правилом. В противном случае следует маркировать модули как "Elaborate_Body". Однако, как мы уже видели, существуют случаи, когда эти три директивы компилятора не могут быть использованы. Таким образом, для клиентов предусмотрены дополнительные методы управления порядком элаборации серверов от которых эти клиенты зависят:

Директива компилятора "**pragma** Elaborate (‘\textit{unit}‘)"

Эта директива помещается после указания спецификатора "**with**", и она требует, чтобы элаборация тела указанного модуля *unit* осуществлялась перед элаборацией модуля в котором указывается эта директива. Эта директива используется когда в процессе элаборации текущий модуль вызывает, прямо или косвенно, какую-либо подпрограмму модуля *unit*.

Директива компилятора "**pragma** Elaborate_All (‘\textit{unit}‘)"

Это более строгая версия директивы "Elaborate". Рассмотрим следующий пример:

```
Модуль A указывает в with модуль B и вызывает B.Func в процессе элаборации
Модуль B указывает в with модуль C, и B.Func вызывает C.Func
```

Если поместить директиву "Elaborate (B)" в модуль *A*, то это гарантирует, что элаборация тела *B* будет выполнена перед вызовом, однако элаборация тела *C* перед вызовом не выполняется. Таким образом, вызов "C.Func" может стать причиной возбуждения исключения *Program_Error*. Результат действия директивы "Elaborate_All" более строгий. Она требует, чтобы была осуществлена предварительная элаборация не только тела того модуля, который указан в директиве (и в спецификаторе "**with**"), но и тела всех модулей, которые используются указанным модулем (для поиска используемых модулей используется транзитивная цепочка спецификаторов "**with**"). Например, если поместить директиву "Elaborate_All (B)" в модуль *A*, то она потребует, чтобы перед элаборацией модуля *A* была осуществлена элаборация не только тела модуля *B*, но и тела модуля *C*, поскольку модуль *B* указывает модуль *C* в спецификаторе "**with**".

Теперь можно продемонстрировать использование этих правил для предупреждения проблем связанных с элаборацией, в случаях, когда не используются динамическая диспетчеризация и ссылочные значения для подпрограмм. Такие случаи будут рассмотренны несколько позже.

Суть правила проста. Если модуль содержит код элаборации, который может прямо или косвенно осуществить вызов подпрограммы модуля указанного в спецификаторе "**with**", или конкретизировать настраиваемый модуль расположенный внутри модуля указанного в спецификаторе "**with**", то в случае, когда модуль, который указан в спецификаторе "**with**", не содержит директиву "Pure", "Preelaborate" или "Elaborate_Body", клиент должен указывать директиву "Elaborate_All" для модуля, который указан в спецификаторе "**with**". Соблюдение этого правила гарантирует клиенту, что вызовы и конкретизация настраиваемых модулей могут быть выполнены без риска возбуждения исключения. Если это правило не соблюдается, то программа может попасть в одно из четырех состояний:

Правильный порядок элаборации отсутствует :

Отсутствует порядок элаборации, который соблюдает правила учитывающие использование любой из директив "Pure", "Preelaborate" или "Elaborate_Body". В этом случае, компилятор Ada95 обязан распознать эту ситуацию на этапе связывания, и отвергнуть построение исполняемой программы.

Порядок элаборации, один или более, существует, но все варианты не правильны :

В этом случае, редактор связей может построить исполняемую программу, но при выполнении программы будет возбуждаться исключение *Program_Error*.

Существует несколько вариантов порядка элаборации, некоторые правильны, некоторые — нет :

Подразумевается, что программист не управляет порядком элаборации. Таким образом, редактор связей может выбрать или не выбрать один из правильных вариантов порядка элаборации, и программа может не возбуждать или возбуждать исключение *Program_Error* во время выполнения. Этот случай является наихудшим, поскольку программа может перестать выполняться после переноса в среду другого компилятора, или даже в среду другой версии того же самого компилятора.

Порядок элаборации, один или более, существует, все варианты правильны :

В этом случае программа выполняется успешно. Такое состояние может быть гарантировано при соблюдении указанных ранее правил, но оно также может быть получено без соблюдения этих правил.

Следует обратить внимание на одно дополнительное преимущество соблюдения правила "Elaborate_All", которое заключается в том, что программа продолжает оставаться в идеальном состоянии даже в случае изменения тел некоторых подпрограмм. В противном случае, если программа, которая не соблюдает указанное правило, в некоторый момент времени оказывается безопасной, то это состояние программы может также незаметно исчезнуть, в результате изменений вызванных нуждами сопровождения программы.

Введение ссылочных типов для подпрограмм усложняется обработкой элаборации. Трудность заключается в том, что не возможно определить вызываемую подпрограмму на этапе компиляции. Это подразумевает, что в таком случае редактор связей не может проанализаровать требования элаборации.

Если в точке, в которой создается ссылочное значение, известно тело подпрограммы, для которого необходимо выполнить элаборацию, то ссылочное значение будет безопасным и его использование не требует проверки. Это может быть достигнуто соответствующей организацией последовательности описаний, если подпрограмма расположена в текущем модуле, или использованием директивы "Pure", "Preelaborate" или "Elaborate_Body", когда подпрограмма расположена в другом модуле, который указан в спецификаторе "**with**".

Если тело используемой подпрограммы, для которого необходимо выполнить элаборацию, не известно в точке создания ссылочного значения, то любое используемое ссылочное значение должно подвергаться динамической проверке, и такая динамическая проверка должна возбуждать исключение *Program_Error* в случае, когда элаборация тела не осуществлена.

При использовании динамической диспетчеризации для тэговых типов необходимые динамические проверки генерируются аналогичным образом. Это означает, что в случае вызова любой примитивной операции тэгового типа, который выполняется до элаборации тела примитивной операции, будет возбуждаться исключение *Program_Error*.

Глава 24

Трудности и рекомендации

24.1 Сюрпризы численных типов

Достаточно часто распространены случаи, когда программист, который только начинает знакомиться с языком программирования Ада, попадается в ловушку численных типов. Рассмотрим следующий пример программы:

```
with  Ada.Text_IO;

procedure Numeric is

    type  My_Integer_Type is new Integer range 1..5;

    My_Integer_Variable       : My_Integer_Type := 2;
    Standard_Integer_Variable : Integer := 2;
    Result_Variable           : Integer;

begin
    Result_Variable := Standard_Integer_Variable * My_Integer_Variable;

    Ada.Text_IO.Put_Line
       ("Result_Variable = " &
        Result_Variable'Img);
end Numeric;
```

Хотя на первый взгляд все выглядит достаточно логично, данная программа компилироваться не будет. Причиной ошибки будет тип "My_Integer_Type", или точнее — отсутствие описания знака операции умножения "*", который одновременно воспринимает левый операнд, стандартного целочисленного типа "**Integer**", и правый операнд, целочисленного типа "My_Integer_Type". Такое поведение компилятора Ады продиктовано требованиями строгой именной эквивалентности типов.

Чтобы решить возникшую проблему, необходимо описать тип "My_Integer_Type" в самостоятельном пакете и описать соответствующие знаки операций, которые будут допустимы для использования над значениями данного целочисленного типа. Вполне естественно, что подобный вариант решения проблемы малопривлекателен, а подобная строгость типизации может показаться чрезмерной даже приверженцам языка Паскаль.

Следует заметить, что существует альтернативный вариант, который гораздо проще и удобнее. Следует описать "My_Integer_Type" не как тип, производный от типа "**Integer**", а как подтип типа "**Integer**" с ограниченным диапазоном значений. После таких изменений наш пример программы будет иметь слудующий вид:

```
with  Ada.Text_IO;

procedure Numeric is

    subtype  My_Integer_Type is Integer range 1..5;

    My_Integer_Variable       : My_Integer_Type := 2;
    Standard_Integer_Variable : Integer := 2;
```

```
    Result_Variable            : Integer;

begin
    Result_Variable := Standard_Integer_Variable * My_Integer_Variable;

    Ada.Text_IO.Put_Line
       ("Result_Variable = " &
        Result_Variable'Img);
end Numeric;
```

В данном случае процедура "Numeric" будет успешно откомпилирована компилятором.

24.2 Принудительная инициализация

В объектно-ориентированных языках программирования, которые используют классы (например язык C++), можно принудительно навязать автоматическую инициализацию для всех объектов класса, снабдив класс конструктором, который гарантированно вызывается при создании объекта. Чтобы добиться подобного эффекта в Аде, необходимо при описании объекта указать присваивание начального значения, которое осуществляется в результате вызова функции. Следует заметить, что в данном случае мы не рассматриваем средства, которые предоставляют контролируемые типы Ады. Рассмотрим следующий пример:

```
package Stacks is

    type Stack is tagged private;

    function Init return Stack;

    -- другие операции
        . . .

private
    type List is array (1..10) of Integer;
    type Stack is record
        Values : List;
        Top    : Integer range 0..10;
    end record;
end Stacks;



with Stacks; use Stacks;
procedure Main is

    A : Stack := Init;
    B : Stack;

begin
    null;
end Main;
```

В этом примере объект "A" будет надежно инициализирован для представления пустого стека. Причем, для этого можно использовать только функции предусмотренные пакетом *Stacks*. Выполнить инициализацию такого объекта каким-либо другим способом — нельзя, поскольку его тип — приватный. Однако, не трудно заметить, что объект "B" — не инициализирован. Таким образом видно, что Ада не предъявляет никаких требований по принудительной инициализации объектов.

Чтобы в подобных случаях возложить контроль за инициализацией объектов на компилятор необходимо использовать дискриминанты. Существует возможность указать наличие дискриминанта в неполном описании типа при фактическом отсутствии дискриминанта в полном описании типа. Поскольку Ада предпочитает ограниченные (*constrained*) объекты (то есть те объекты, размер которых известен), то применение комбинации из дискриминанта и приватного типа послужит причиной того, что компилятор будет требовать от программиста явной инициализации всех объектов такого типа.

```
package Stacks is
    type Stack (<>) is private;
    -- указание того, что Stack может иметь дискриминант...

    function Init return Stack;

    -- другие операции
        . . .

private
    type List is array (1..10) of Integer;

    -- ...даже если вы не стремитесь иметь здесь дискриминант
    type Stack is record
        Values : List;
        Top    : Integer range 0..10;
    end record;
end Stacks;


with Stacks; use Stacks;
procedure Main is

    A : Stack := Init;
    B : Stack;
    -- теперь это недопустимо!
    -- вы должны вызвать функцию для инициализации B

begin
    null;
end Main;
```

Подобный подход может показаться несколько интуитивным, однако за счет явности описаний инициализации он повышает общую читабельность исходного текста.

24.3 Взаимно рекурсивные типы

Бывают случаи, когда необходимо иметь два (или более) типа, которые являются взаимно рекурсивными. Такие типы содержат указатели, ссылаются друг на друга. В качестве примера рассмотрим типы: "Doctor" — врач и "Patient" — пациент. Эти типы нуждаются в указателях, которые ссылаются друг на друга. Ада потребует поместить оба этих типа в один пакет. Чтобы сохранить возможность раздельной компиляции, можно описать два базовых типа в одном пакете:

```
package Doctors_And_Patients is

    type Doctor is abstract tagged null record;
    type Doctor_Ptr is access all Doctor'Class;


    type Patient is abstract tagged null record;
    type Patient_Ptr is access all Patient'Class;

end Doctors_And_Patients;
```

Позже, для непосредственного фактического использования, можно создать в самостоятельных пакетах производные типы, которые используют описания этих базовых типов. Подобные описания могут иметь следующий вид:

```
with Doctors_And_Patients; use Doctors_And_Patients;

package Doctors is
    type Doc is new Doctor with private;
```

```
    -- подпрограммы
        . . .

private
    type Doc is new Doctor with
        record
            Pat : Patient_Ptr;
                . . .

        end record;
end Doctors;



with Doctors_And_Patients; use Doctors_And_Patients;

package Patients is

-- далее подобно Doctors...
    . . .
```

При описании базовых типов, в пакете *Doctors_And_Patients*, можно также описать операции, которые будут определять элементарную функциональность базовых типов. Дополнительные операции могут быть добавлены позднее, в описаниях производных типов (например в типе "Doc").

Примечательно, что тело одного пакета может видеть спецификацию другого пакета. Таким образом, тела пакетов могут иметь следующий вид:

```
with Doctors; use Doctors;
package body Patients is

    -- тело пакета Patients имеет доступ к сервисам Doctors
    -- описанным в спецификации пакета Doctors

        . . .

end Patients;



with Patients; use Patients;
package body Doctors is

    -- тело пакета Doctors имеет доступ к сервисам Patients
    -- описанным в спецификации пакета Patients

        . . .

end Doctors;
```

24.4 Рекомендации по построению абстракций

Как уже говорилось, пакет Ады является инструментом абстракции данных. В этом случае пакет содержит какой-либо главный тип, и такая конструкция очень подобна классам, используемым в других объектно-ориентированных языках программирования. Чтобы избежать неоднозначностей (напомним, что понятие класса в Аде отличается от понятия класса в других объектно-ориентированных языках), мы называем такую конструкцию абстракцией. В большинстве подобных случаев, главный тип описан как приватный тип или тип, имеющий приватное расширение. Спецификация пакета также может содержать описания других типов, мы будем называть их обычными типами. Эти типы, как правило, используются для построения интерфейса (когда они описаны публично), или для представления внутренних структур данных (когда они описаны приватно).

Простой демонстрацией подобного подхода построения абстракции может служить следующий схематический пример:

```
with Angle;
package Coordinates is

    type Object is ...      -- главный тип абстракции
                            -- (приватный или с приватным расширением)

    type Geographic is      -- обычный тип (описан публично)
        record
            Latitude  : Angle.Radian;
            Longitude : Angle.Radian;
        end record;

private
    . . .

end Coordinates;
```

Подобный подход является базисом. При этом, программисту предоставляется широкий выбор: должен-ли он использовать тэговые типы? контролируемые типы? . . .

24.4.1 Тэговые типы — не для всех абстракций!

Первое правило — очень простое: не следует использовать тэговые типы для очень простых абстракций, таких как расстояние, угол, температура, и так далее:

- во-первых, такие абстракции нет смысла расширять.
- во-вторых, такие абстракции достаточно интенсивно используются, а тэговые типы требуют дополнительные затраты производительности
- в третьих, если тип описан как тэговый и вы порождаете от него другой тип, то вам потребуется переопределить все примитивные операции, такие как "+" и "−".

24.4.2 Контролируемые или не контролируемые?

Когда главный тип абстракции является производным от "Ada.Finalization.Controlled", то вызовы " Initialize", "Adjust" и "Finalize" автоматически выполняются компилятором (так указывается в руководстве по языку). Это полезно для предотвращения утечек памяти (компилятор никогда не забудет удалить объект, который больше не используется). Это полезно при дублировании объекта: две копии могут быть сделаны независимыми.

Однако эти свойства должны использоваться **только** в случаях, когда традиционное поведение компилятора, принятое по умолчанию, не удовлетворяет потребности разрабатываемой абстракции. Действительно, как только главный тип абстракции описан как контролируемый, то программист обязан заботиться о контролируемой обработке для каждого потомка. Таким образом, если потоммок добавляет какое-либо расширение, то унаследованные операции " Initialize", "Adjust" и "Finalize" не имеют никакого представления о таком расширении. Следовательно, вы **обязаны** переопределять эти операции для потомков вашей абстракции.

24.4.3 Никогда не используйте неинициализированные объекты

Для избежения непредсказуемого поведения при использовании неинициализированных объектов, описание главного типа абстракции должно устанавливать значения по умолчанию для каждого атрибута. Следовательно, каждая абстракция должна описывать публичный объект "**null**", который может быть использован как значение атрибута по умолчанию, в контексте описания абстракции более высокого уровня.

Для простых типов, таких как "Integer_32" или "Float_6", значение по умолчанию соответствует ограниченному выбору значений, которые могут быть использованы вместо "Integer_32'Last" или "Float_6'Last".

Для перечислимых типов значение по умолчанию должны соответствовать выбору по умолчанию.

24.4.4 Создание и удаление объектов

Если абстракция описывает функцию "Create", которая возвращает экземпляр объекта главного типа абстракции, то все потомки этой абстракции будут наследовать такую операцию. Это может быть опасно в случаях, когда абстракция-потомок имеет больше атрибутов чем абстракция-родитель. Действительно, унаследованная от предка функция "Create" может не инициализировать дополнительные атрибуты абстракции-потомка. В концепции Ada 95 предлагаются два варианта решения для предотвращения наследования функции "Create":

- вместо возвращения экземпляра объекта главного типа абстракции, функция "Create" возвращает ссылку на объект этого типа
- функция "Create" описывается во внутреннем или дочернем пакете

К сожалению, ни одно из этих решений нельзя назвать полностью удовлетворительным.

Первое решение заставляет выполнять явную деаллокацию каждого экземпляра объекта, который больше не используется. В некоторых случаях деаллокация просто не возможна, то есть

```
A := Create (...).all;
```

Второе решение затрудняет автоматическую генерацию кода. Кроме того, оно вызывает принудительное копирование объектов после их создания (минимум однократно), что является дополнительным расходом производительности.

К счастью, в Аде реально не требуется явная подпрограмма "Create". Любой экземпляр объекта абстракции создается автоматически, или при его описании (статический экземпляр), или с использованием динамического размещения:

```
Instance := new Class.Object
```

Следовательно, должна быть описана только подпрограмма инициализации. Если инициализация принятая по умолчанию не удовлетворяет потребностям, то могут быть описаны новые подпрограммы инициализации (с дополнительными параметрами или без). Чтобы такие подпрограммы были не наследуемыми, они должны принимать в качестве главного параметра объект надклассового типа ("'Class"). Для простых абстракций, вместо " Initialize", может быть использован метод "Set_« Attribute_Name»".

Для удаления объекта абстракции, когда главный тип абстракции — контролируемый, подпрограмма "Delete" может быть безболезненно заменена на "Finalize". В случае когда главный тип абстракции — не контролируемый, необходимость в подпрограмме "Delete" — отсутствует.

24.4.5 Именование тэговых типов

При построении абстракций с применением тэговых типов, необходимо использовать согласованные наименования как для тэговых типов, так и для ассоциируемых с ними пакетов. Следует заметить, что соглашения по наименованию часто являются источником "религиозных войн". В следствие этого существует два различных подхода.

Первый подход, за исключением двух специальных случаев, позволяет установить единые соглашения по наименованию описаний, вне зависимости от использования объектно-ориентированных свойств, и при этом он интегрирует в себе использование объектно-ориентированных свойств:

- тэговые типы именуются традиционным образом
- для имен пакетов экспортирующих какие-либо абстракции, для которых предполагается множество реализаций, используется префикс "Abstract_"
- для имен пакетов предусматривающих функциональные модули, которые могут быть подмешаны (*mixed in*) в основную абстракцию используется суффикс "_Mixin"

Следующий пример, который состоит из двух частей, демонстрирует использование этого варианта соглашений по наименованию. Для первой части этого примера предположим, что тип "Set_Element" описывается где-то в другом месте:

```
package Abstract_Sets is
   type Set is abstract tagged private;
   -- пустое множество
   function Empty return Set is abstract;
   -- построение множества из одного элемента
   function Unit (Element: Set_Element) return Set is abstract;
   -- объединение двух множеств
   function Union (Left, Right: Set) return Set is abstract;
   -- пересечение двух множеств
   function Intersection (Left, Right: Set) return Set is abstract;
   -- удаление элемента из множества
   procedure Take (From    : in out Set;
                   Element :    out Set_Element) is abstract;
   Element_Too_Large : exception;

private
   type Set is abstract tagged null record;
end Abstract_Sets;


with Abstract_Sets;
package Bit_Vector_Sets is   -- одна реализация абстракции множества
   type Bit_Set is new Abstract_Sets.Set with private;
   . . .
private
   Bit_Set_Size : constant := 64;
   type Bit_Vector is ...
   type Bit_Set is new Abstract_Sets.Set with
      record
         Data : Bit_Vector;
      end record;
end Bit_Vector_Sets;


with Abstract_Sets;
package Sparse_Sets is  -- альтернативная реализация абстракции множества
   type Sparse_Set is new Abstract_Sets.Set with private;
   . . .
private
   . . .
end Sparse_Sets;
```

Вторая часть этого примера демонстрирует использование соглашений по наименованию для смешанных пакетов (*mixin packages*), которые обеспечивают поддержку оконной системы.

Предположим, что у нас есть тэговый лимитированный приватный тип "Basic_Window", описание которого имеет следующий вид:

```
type Basic_Window is tagged limited private;
```

Тогда:

```
generic
   type Some_Window is abstract new Basic_Window with private;
package Label_Mixin is
   type Window_With_Label is abstract new Some_Window with private;
   . . .
private
   . . .
end Label_Mixin;


generic
   type Some_Window is abstract new Basic_Window with private;
package Border_Mixin is
   type Window_With_Label is abstract new Some_Window with private;
```

```
   . . .
private
   . . .
end Border_Mixin;
```

Второй подход отображает использование объектно-ориентированных свойств с помощью специальных имен и суффиксов:

- пакет абстракции именуется без каких-либо суффиксов, в соответствии с объектом, который этот пакет представляет
- смешанные пакеты (*mixin packages*) именуются в соответствии с аспектами (*facet*) которые они представляют, используя суффикс "_Facet"
- для главного тэгового типа абстракции используется имя "Instance" ("Object"...)
- для ссылочного типа (если такой определен), значения которого ссылаются на значения главного тэгового типа, используется имя "Reference" ("Handle", "View"...)
- для последующего описания типа, который является надклассовым типом соответствущим тэговому типу, используется имя "Class"

Следующий пример демонстрирует использование этого варианта соглашений по наименованию:

```
package Shape is
   subtype Side_Count is range 0 .. 100;
   type Instance (Sides: Side_Count) is tagged private;
   subtype Class is Instance'Class;
   . . .
   -- операции Shape.Instance
private
   . . .
end Shape;


with Shape; use Shape;
package Line is
   type Instance is new Shape.Instance with private;
   subtype Class is Instance'Class;
   . . .
   -- переопределенные или новые операции
private
   . . .
end Line;


with Shape; use Shape;
generic
   type Origin is new Shape.Instance;
package With_Color_Facet is
   type Instance is new Origin with private;
   subtype Class is Instance'Class;
   -- операции для colored shapes
private
   . . .
end With_Color_Facet;


with Line; use Line;
with With_Color_Facet;
package Colored_Line is new With_Color_Facet (Line.Instance);
```

Простые описания, в этом случае, могут иметь следующий вид:

```
Red_Line : Colored_Line.Instance;
procedure Draw (What : Shape.Instance);
```

Показанная выше схема соглашений по наименованию корректно работает как в случае использования полных имен, так и при использовании спецификатора "**use**". При использовании одного и того же имени для всех специфических типов (например "**type** Instance") и надклассовых тпов, одни не полностью квалифицированные имена всегда будут "прятать"другие не полностью квалифицированные имена. Таким образом, компилятор будет требовать применение полной точечной нотации, чтобы избежать возникновение двусмысленности имен при использовании спецификатора "**use**".

Необходимо использовать такую схему наименований, которая будет согласованной, читабельной и выразительной для представления абстракции. В идеале, схема наименований для различных способов использования тэговых типов, которые используются для построения различных абстракций, должна быть единой. Однако если используемая схема соглашений по наименованиям слишком жесткая, то использующие ее фрагменты кода будут выглядеть неестественно с точки зрения читабельности. При использовании подобных соглашений по наименованию для расширений типа, посредством наследования или подмешивания настраиваемых модулей (*generic mixin*), достигается читабельность описаний объектов.

24.4.6 Именование методов

Для именования методов абстракций предлагаются следующие правила:

1. Имена методов в комбинации с их аргументами должны читаться подобно фразам английского языка.
2. функции, возвращающие логические результаты, должны использовать предикативные предложения.
3. Именная ассоциация должна использоваться для методов, которые имеют более одного параметра.
4. Аргумент, который обозначает обрабатываемый методом экземпляр объекта, не должен называться также как абстракция к которой он принадлежит. Вместо этого, он должен использовать более общее имя, такое как: *Affecting*, *Self*, *This*... (примечание: это правило обязательно используется только при именном ассоцировании параметров).

Следующий пример демонстрирует использование этих правил:

```
Insert (The_Item      => An_Apple,
        Before_Index => 3,
        Into          => The_Fruit_List);

Merge (Left  => My_Fruit_List,
       Right => The_Citrus_Fruits);

Is_Empty (The_Queue); -- используется позиционная ассоциация
```

Следует учитывать, что при несоблюдении правила 4 может быть нарушена читабельность наследуемых операций. Рассмотрим следующий случай, где правило 4 нарушено:

```
package List is
    . . .
    procedure Pop (The_Item  : out    List.Item;
                   From_List : in out List.Object); -- используется From_List
                                                    -- вместо From
end List;

package Queue is
    . . .
    type Object is new List.Object; -- Queue.Object наследует
                                    -- операции List.Object
end Queue;

with List;
with Queue;
procedure Main is
begin
    . . .
    Queue.Pop
      (The_Item  => My_Item
```

```
        From_List => The_Queue); -- не очень "чисто":
                                 -- имя ассоциации указывает список на (List),
                                 -- а аргументом является очередь (Queue) ?!
end Main;
```

24.4.7 Опасность наследования

Когда главный тип абстракции — производный, он наследует все примитивные методы своих предков. Иногда некоторые из унаследованных методов не будут работать или будут не правильно работать с производной абстракцией (как правило из-за того, что они не имеют представления о новых атрибутах). В подобных случаях возникает необходимость переопределения унаследованных методов.

Упущение необходимости переопределения метода является одной из широко распространенных ошибок. Такую ошибку не всегда можно обнаружить во время тестирования самостоятельного модуля, без проверки всех унаследованных методов в контексте производной абстракции.

Контролируемые объекты Ada 95 — это первые кандидаты для повышенного внимания в подобных ситуациях. Любой контролируемый атрибут, добавленный в производной абстракции, требует переопределения унаследованных подпрограмм "Adjust" и "Finalize" (как минимум). Они должны вызываться, соответственно, для подгонки и очистки новых атрибутов.

Рассмотрим следующий пример:

```
package Coordinates is

    type Object is tagged private;
    . . .

    function Distance_Between
       (Left : in Coordinates.Object;
        Right: in Coordinates.Object)
        return Distance.Object;
    . . .
end Coordinates;


package Position is

    type Object is new Coordinates.Object with private;
    . . .

    function Distance_Between        -- переопределяет
       (Left  : in Position.Object; -- Coordinates.Distance_Between
        Right : in Position.Object)
        return Distance.Object;
    . . .
end Position;
```

Здесь, абстракция "Position" является производной от абстракции "Coordinates", но метод "Distance_Between" должен быть переопределен для учета в векторе позиции компонента высоты.

24.5 Советы Паскаль-программистам

Ада является языком программирования, который во многих аспектах подобен языку Паскаль. Однако, Ада не является "супермножеством"средств языка Паскаль. Таким образом, синтаксис инструкций Ады несколько отличается от синтаксиса инструкций языка Паскаль и множество средств языка Паскаль реализовано в Аде несколько иначе. Здесь приводится некоторое обобщение различий языков программирования Паскаль и Ада, которое может быть полезно для программистов которые хорошо знают язык Паскаль и желают изучить язык Ада.

Самое важное отличие Ады от Паскаля заключается в том, что Ада строжайшим образом стандартизированный язык программирования, дополнительно сопряженный с процессом сертификации компилятора. Таким

образом, Ада свободна от наличия различных синтактических расширений, которые можно встретить, практически, в каждой Паскаль-системе. С другой строны, стандартно описанный язык программирования Ада содержит почти все средства и особенности современных расширений языка Пасскаль.

24.5.1 Описания и их последовательность

Стандарт языка Паскаль требует использование правильного порядка следования описаний (константы, типы, переменные, подпрограммы), что ослабляется некоторыми реализациями Паскаль-систем. Ада обладает более гибкими требованиями к порядку следования описаний. Так, стандарт Ады подразумевает "базовые описания"и "поздние описания". Таким образом, к "базовым"описаниям можно отнести описания констант, типов и переменных, а к "поздним"описаниям — описания подпрограмм (процедур и функций). Следует заметить, что мы не рассматриваем остальные описания в целях упрощения. В описательной части, программы или подпрограммы, базовые описания могут быть свободно перемешаны (с естественным пониманием того, что перед тем как что-либо используется оно должно быть предварительно описано). Все базовые описания должны предшествовать всем поздним описаниям.

В Паскале, зарезервированные слова "**type**", "**const**" и "**var**" должны появляться в описательной части только один раз. В Аде, описание каждого типа или подтипа должно соответственно начинаться с "**type**" или "**subtype**". Примером описания константы может служить следующее:

```
FirstLetter: constant Character := 'A';
```

Зарезервированное слово "**var**" не используется вовсе, поэтому переменные описываются подобным образом:

```
Sum : Integer;
```

Кроме того, описание типа записи, в Аде, всегда должно завершаться "**end record**".

24.5.2 Структуры управления

Все структуры управления последовательностью выполнения (иначе, управляющие структуры) Ады имеют соответствующие закрывающие инструкции, такие как "**if** ... **end if**", "**loop** ... **end loop**", "**case** ... **end case**". Далее, в Аде символ двоеточия ";" используется для завершения инструкции, а не для разделения инструкций, как в Паскале. Это обеспечивает синтаксис, который, по сравнению с синтаксисом Паскаля, легче использовать корректно. Например, инструкция Паскаля:

```
if X < Y then
    A := B;
```

будет написана в Аде следующим образом:

```
if X < Y then
    A := B;
end if;
```

А инструкция Паскаля:

```
if X < Y then
    begin
        A := B;
        Z := X
    end
else
    begin
        A := X;
        Z := B
    end;
```

будет написана в Аде следующим образом:

```
if X < Y then
    A := B;
    Z := X;
else
    A := X;
    Z := B;
end if;
```

Использование подобного синтаксиса в Аде гарантирует остутствие "висячих**else**".

Переменные управляющие циклами "**for**" всегда описываются неявно, что является единственным исключением из правил, требующих чтобы все было описано явно. Счетчик цикла "**for**" локален для тела цикла. Описание счетчика цикла "**for**" как переменной, так как это принято в Паскале, не наносит реального ущерба, но описывает самостоятельную переменную, которая спрятана за фактическим счетчиком цикла и, следовательно, не видима в теле цикла.

Диапазоны, для цикла "**for**", часто назначаются как имена типов или подтипов, подобно следующему:

```
for Count in Index'Range loop
```

Ада не имеет структуры цикла "**repeat**". Вместо этого используется "**loop** ... **end loop**" с инструкцией выхода из цикла "**exit when**" в конце тела цикла.

Переменная выбора в инструкции "**case**" должна быть дискретного (целочисленного или перечислимого) типа. Различные альтернативы выбора "**case**" должны указывать все возможные значения переменной выбора в неперекрывающейся манере.

24.5.3 Типы и структуры данных

Двумерные массивы **не** являются массивами массивов. Следовательно, "A (J)(K)" **не** является тем же самым, что и "A (J, K)". Разработчики, в действительности, относят последний из двумерных массивов к массивам массивов. Смысл этого заключается в том, что существуют различные структуры Ады для которых стандарт **не** определяет отображение на хранение многомерного массива в памяти (ориентирование на строку или на столбец). Например, это позволяет предусмотрительным реализаторам Ада-систем использовать нелинейное отображение. На практике, большая часть существующих Ада-компиляторов использует отображение ориентирование на строку, соответствующее правилам Паскаля и Си.

В качестве типов полей записи всегда должны использоваться имена типов или подтипов, что означает, что поле записи не может иметь анонимный тип, подобно "**array**" или "**record**". Для построения иерархических типов записей, сначала необходимо построить типы записей нижнего уровня, а затем использовать имена этих типов для описания полей записей более высого уровня.

Ада не обеспечивает какого-либо средства, которое соответствует "**with**" в Паскале. Это значит, что все обращения к элементам массивов и записей должны использовать полную точечную нотацию.

Ада более жестко контролирует использование записей с вариантами чем Паскаль. Таким образом, нет возможности описать "свободное объединение", или вариантную запись без указания дискриминанта. В Паскале и Си, свободные объединения часто используются для обхода проверки типа, но такой подход не может быть использован в Аде (Ада имеет настраиваемую функцию "Unchecked_Conversion", которая может быть использована для действительного обхода проверки типа).

В отличие от Паскаля, Ада не имеет предопределенного средства построения типа множества "**set**". Однако, средства Ады обеспечивают программисту возможность построить подобный тип самостоятельно.

24.5.4 Совместимость типов и подтипов

Важно помнить и понимать, что Ада использует именную, а не структурную эквивалентность типов. Например, пусть даны следующие описания:

```
A, B: array (1..10) of Float;
C   : array (1..10) of Float;
```

Тогда, следующие инструкции присваивания:

```
A := B;
C := B;
```

будут недопустимыми, поскольку все три массива имеют различные анонимные типы, назначенные компилятором (некоторые компиляторы Паскаля будут позволять осуществление первого присваивания). Для обеспечения возможности присваивания массивов, необходимо явное указание имени для типа массива:

```
type List is array (1..10) of Float;

A, B: List;
C: List;
```

Теперь оба показанных выше присваивания будут допустимы. При программировании на Аде, настоятельно рекомендуется избегать использования анонимных типов, подобно тому как это делается в Паскале.

24.5.5 Параметры подпрограмм

Режимы передачи параметров "**in**", "**out**" и "**in out**". для подпрограмм Ады, являются грубыми эквивалентами режимов передачи параметров по значению и по ссылке ("**var**") для подпрограмм Паскаля.

В теле подпрограммы, "**in**"-параметры могут быть только прочитаны. Основное отличие между "**out**" и "**in out**" параметрами заключается в том, что текущее значение фактического "**in out**"-параметра передается в процедуру, а текущее значение фактического "**out**"-параметра в процедуру не передается и, следовательно, может считаться неопределенным до указания значения внутри подпрограммы. Функции Ады не могут иметь "**out**" и "**in out**" параметры.

В случае передачи в подпрограмму массива, использование режима передачи параметров "**in out**" не обеспечивает никакого выигрыша в эффективности по сравнению с режимом передачи параметров "**in**". Подобный прием является широко распространенным в языке Паскаль, когда большие массивы передаются как "**var**"-параметры. Правила языка Паскаль требуют, чтобы "**var**"-параметры передавались по ссылке, а параметры передаваемые по значению — копировались. Ада использует другие правила: параметры скалярных типов всегда передаются по значению/результату не зависимо от указанного режима передачи параметров. Ада допускает, чтобы параметры составных типов (массивы и/или записи) также передавались по значению/результату, но реальные Ада-компиляторы никогда не так не делают, в особенности, когда составной тип имеет большие размеры. Таким образом, реальные Ада-компиляторы передают в подпрограммы массивы и записи больших размеров по ссылке, даже если для них указан режим передачи параметров "**in**". Поскольку "**in**"-параметры гарантировано не могут быть записаны, то не существует опасности непреднамеренного изменения содержимого параметра внутри вызванной подпрограммы.

В случае использования режимов "**in**" и "**in out**" для передачи параметров скалярных типов, значения копируются обратно в вызвавшую подпрограмму после **нормального** завершения вызвынной процедуры. Таким образом, если вызов процедуры заканчивается распространением исключения в вызвавшую подпрограмму, то значения параметров не копируются обратно, в вызвавшую подпрограмму, и, следовательно, вызвавшая подпрограмма сохраняет оригинальные значения (те которые были до вызова процедуры).

Инструкции ввода/вывода Ады, являются обычными вызовами процедур, которые подразумевают, что только одиночные целочисленные, вещественные, символьные, строковые или перечислимые значения могут быть прочитаны или отображены с помощью каждого вызова "**Get**" или "**Put**". При написании инструкций ввода/вывода, для этих процедур нельзя предоставлять произвольное число параметров, как это допускается в Паскале. Попытка выполнения этого, приведет к естественной ошибке компиляции, говорящей о не совпадении числа параметров (*unmatched procedure call*), когда компилятор пытается найти версию процедуры "**Get**" или "**Put**", с соответствующим числом параметров.

24.5.6 Пакеты Ады и их соответствие модулям Паскаля

Реально, модули Паскаля ("unit") не являются частью стандарта *ISO Pascal*. Однако, их использование предусматривается многими современными расширениями языка Паскаль, например, версиями Паскаль-систем от *Borland*, *Symantec* или *Free Pascal*. Модули Паскаля являются грубым эквивалентом пакетов Ады с двумя важными отличиями:

- В то время, когда интерфейс модуля Паскаля является, в основном, частью того же самого файла с исходным текстом который содержит детали внутренней реализации этого модуля, пакет Ады может быть нормально разделен на самостоятельные файлы спецификации и тела пакета. Некоторые компиляторы явно требуют такое разделение. В любом случае, такое разделение настоятельно рекомендуется, поскольку Ада, при перекомпиляции спецификации пакета, осуществляет принудительную перекомпиляцию всех модулей-клиентов, которые используют этот пакет, а при перекомпиляции тела пакета — нет.
- Паскаль-системы не предусматривают непосредственного эквивалента для приватных ("**private**") типов Ады.

24.5.7 Использование "`is`" и символа точки с запятой "`;`"

Бесконечное горе ждет тех пользователей Ады, которые путают назначение символа точки с запятой ";" с назначением "**is**". При использовании некоторых компиляторов, это приводит к длинной последовательности сообщений об ошибках. Наиболее вероятен случай использования точки с запятой вместо "**is**" — при описании подпрограмм, также как это делается в языке Паскаль:

```
procedure Do_Something (X : Integer); --   <---- это подразумевает проблему!!!

  -- описания

begin

  -- инструкции

end Do_Something;
```

Проблема заключается в том, что такое использование символа точки с запятой допустимо, но смысл его отличается от того, что вы ожидаете. Строка:

```
procedure Do_Something (X : Integer);
```

является, в реальности, описанием спецификации процедуры, что в языке Паскаль больше похоже на опережающее описание процедуры с помощью директивы "forward". Следовательно, перепутывание символа точки с запятой ";" с "**is**" почти гарантировано приводит к порождению большого количества сообщений об ошибках компилятора, поскольку синтаксический анализатор компилятора Ады будет трактовать такую инструкцию как спецификацию подпрограммы и будет сбит с толку появлением последующего блока "**begin** ... **end**", который ошибочно располагается вне контекста. Появление "**is**" точно указывает Ада-компилятору на то, что далее следует тело подпрограммы.

Спецификации подпрограмм, являющиеся частью спецификации пакета, могут рассматриваться в контексте опережающих описаний, для чего в Паскале обычно используется директива "forward". В этом случае, первая строка тела подпрограммы (в теле пакета) должна быть идентична спецификации, за исключением того, что символ точки с запятой ";" заменяется на "**is**". Это отличается от Паскаля, где список параметров может не повторяться.

Часть 3.

Средства разработки

Глава 25

Средства разработки

25.1 Доступность средств разработки

Практическое использование любого языка программирования требует наличия соответствующих средств разработки, и язык программирования Ада в этом случае не является исключением.

В настоящее время существует целое множество фирм, которые предлагают различные средства для ведения разработки программного обеспечения с использованием языка программирования Ада. Примерами таких фирм могут служить:

- *Ada Core Technologies*
- *Aonix*
- *AverCom*
- *DDC-I*
- *Green Hills Software*
- *Irvine Compiler Corp.*
- *OC Systems*
- *Rational Software Corp.*
- *RR Software*

Такие фирмы специализируются на разработке компиляторов, различных библиотек и инструментальных средств, охватывая, как правило, достаточно широкий спектр используемых аппаратных платформ и операционных систем. Естественно, стоимость коммерческих средств разработки достаточно высока, однако некоторые фирмы предоставляют значительные скидки для учебных заведений.

Кроме коммерчески доступных средств разработки, в частности, компиляторов, существуют свободно доступные версии средств разработки:

- Компилятор *ObjectAda* от *Aonix* (`http://www.aonix.com/`), который позволяет вести разработку под операционными системами Windows и Solaris.
- Компилятор *GNAT* от *Ada Core Technologies* (`http://www.gnat.com/`), который поставляется со всеми исходными текстами под лицензией *GPL* и поддерживает большое число платформ.

Следует заметить, что на свободно распространяемую версию компилятора *ObjectAda* от *Aonix* накложены достаточно жесткие ограничения: ограничено общее число модулей из которых может состоять программа и ограничено общее число одновременно выполняемых задач в пределах одной программы. Кроме того, компилятор *ObjectAda* поставляется в виде файлов исполняемых программ без исходных текстов.

Исходя из этого, значительно больший интерес представляет свободно распространяемая версия компилятора *GNAT* от *Ada Core Technologies*, для которой подобные ограничения отсутствуют. Справедливо будет заметить, что *Ada Core Technologies* строго предупреждает об отсутствии каких либо гарантий на использование данной версии компилятора, и не рекомендует использовать эту версию компилятора при разработке программного обеспечения от которого требуется высокая надежность.

25.2 Система Ада-компилятора GNAT

Система Ада-компилятора GNAT (сокращение от ***G****NU* ***N****ew York University* ***A****da* ***T****ranslator*) одновременно является компилятором и системой времени выполнения для Ada95, которая использует многоплатформенный кодогенератор GCC, благодаря которому обеспечивается поддержка большого количества аппаратных платформ. Таким образом, GNAT является частью программного обеспечения GNU.

Система Ада-компилятора GNAT была разработана в процессе тесного сотрудничества двух коллективов разработчиков:

GNAT Development Team — из Нью-Йоркского университета, под руководством профессоров *Edmond Schonberg* и *Robert B. K. Dewar*. Эта группа разрабатывала препроцессор и компилятор.

*Project PART (**P**OSIX **A**da **R**eal-**T**ime) Team* — из университета штата Флорида, под руководством профессора *Theodore P. Baker*. Эта группа разрабатывала библиотеку времени выполнения Ады.

Изначально этот проект финансировался правительством USA (с 1991 по 1994). В августе 1994 года основные авторы проекта создали компанию *Ada Core Technologies Inc*. (*ACT*), которая предоставляет техническую поддержку по использованию системы GNAT в промышленных и коммерческих разработках. В настоящее время, *Ada Core Technologies* продолжает расширять количество платформ на которых можно использовать Ада-компилятор GNAT, а также предусматривает различные средства для разработки и отладки Ада-программ. Компания предусматривает свободное обновления компилятора для сообщества пользователей языка программирования Ада.

Глава 26

Установка GNAT

Как уже говорилось, GNAT является частью проекта GCC. Команда "`gcc`", сама по себе, не является компилятором. Это программа которая определяет тип исходного текста и затем осуществляет запуск соответствующего компилятора. Например, компилятором языка Ада является программа называемая "`gnat1`", а компилятором языка C является программа называемая "`cc1`". Когда программа "`gcc`"определяет, что предоставленный ей исходный текст написан на языке Ада она просто запускает компилятор "`gnat1`"для компиляции этого исходного текста.

Поскольку GNAT и GCC должны работать совместно, то определенная версия GNAT создается в рассчете на то, что она будет взаимодействовать с определенной версией GCC. Так, версии GNAT: 3.11p, 3.12p, 3.13p, 3.14p и 3.15p собраны с GCC версии 2.8.1 (чтобы узнать какая версия GCC используется в системе нужно запустить "`gcc`"с опцией командной строки "`-v`").

Стандартный дистрибутив GNAT от *ACT* поставляется со своей собственной копией GCC, имеющей корректную версию "`gcc`", и позволяет установить GNAT и GCC в различные каталоги. Дистрибутив GNAT от *ACT* содержит двоичные исполняемые файлы у которых отключена поддержка языка C++. Таким образом, если необходима поддержка языка C++ одновременно с языком Ада, то необходимо пересобрать GNAT и GCC из исходных текстов.

Доступ к свободно распространяемой версии компилятора GNAT от *ACT* предоставляется через сеть Internet с сервера Нью-Йоркского университета по адресу "`ftp://ftp.cs.nyu.edu/pub/gnat`". В момент написания этого текста на этом сервере доступны две версии компилятора: 3.14p и 3.15p (последняя версия 3.15p), которые соответственно расположены в подкаталогах "`ftp://ftp.cs.nyu.edu/pub/gnat/3.14p`" и "`ftp://ftp.cs.nyu.edu/pub/gnat/3.15p`". Следует заметить, что на различных зеркалах этого сервера (которых в Internet немало) можно обнаружить и более ранние версии компилятора.

Каждый подкаталог в котором храниться соответствующая версия компилятора содержит архивы бинарных файлов, файлов документации и файлов с исходными текстами. Архивы бинарных файлов упакованы с учетом использования компилятора на различных целевых платформах. Тип целевой платформы, как правило, можно определить по имени архивного файла. Например, архив "`gnat-3.15p-i686-pc-redhat71-gnu-bin.tar.gz`" содержат компилятор GNAT версии 3.15p, который может использоваться на процессоре i686 и операционной системе Linux (желательно дистрибутив Red Hat 7.1), а архив "`gnat-3.15p-sparc-sun-solaris2.5.1-bin.tar.gz`" содержат компилятор GNAT версии 3.15p, который может использоваться на процессоре SPARK и операционной системе Solaris-2.5.1.

Вне зависимости от того какую версию компилятора вы выберете, перед началом установки настоятельно рекомендуется ознакомиться с содержимым соответствующих файлов "`README`".

26.1 Установка GNAT на Windows

Компилятор GNAT, который собран для работы на аппаратной платформе Intel x86 под управлением операционной системой Windows находится в подкаталоге "`winnt`". Как говорится в документации, версия 3.14p может быть использована в среде Windows 95/98 или Windows NT 4.0/2000, а версия 3.15p — в среде Windows 95/98 или Windows NT 4.0/2000/XP.

Компилятор GNAT для Windows разбит на две части. Файл "`gnat-3.14p-nt.exe`" (или "`gnat-3.15p-nt.exe`") содержит базовый комплект инструментов Ада-компилятора, а файл "`gnatwin-3.14p.exe`" (или "`gnatwin-3.15p.exe`") содержит компоненты Ада-интерфейса Win32.

Для установки компилятора необходимо просто запустить на выполнение файл "gnat-3.14p-nt.exe" (или "gnat-3.15p-nt.exe") и следовать инструкциям интерактивной установки. После успешного завершения установки базового комплекта инструментов, аналогичным образом, устанавливаются компоненты Ада-интерфейса Win32.

Кроме этого, специально для пользователей Windows, существует свободно распространяема интегрированная среда разработки AdaGide, которая ориентированна на компилятор GNAT. Дистрибутив AdaGide свободно доступен с WEB-странички проекта: "http://www.usafa.af.mil/dfcs/bios/mcc_html/adagide.html".

26.2 Установка GNAT на Linux

При установке GNAT на Linux следует учитывать, что в системе возможна совместная одновременная установка нескольких различных версий "gcc". Для выбора определенной версии "gcc", необходимой для компиляции исходных текстов, можно использовать опцию командной строки "-V". Однако, для выполнения этого, необходимо пересобрать GCC из исходных текстов так, чтобы можно было использовать различные версии "gcc".

26.2.1 Установка бинарных файлов от *ACT*

При установке бинарных файлов от *ACT* следует учитывать, что они построены с учетом определенной версии библиотеки языка C. Для того, чтобы определить какая версия библиотеки "libc" используется в вашем дистрибутиве Linux, следует проверить на какой файл указывает ссылка "/lib/libc.so". Например, если ссылка "/lib/libc.so" указывает на файл "libc5", то вам необходима версия GNAT которая использует "libc5".

По умолчанию, GNAT от *ACT* устанавливается в каталог "/usr/gnat". Если у вас отсутствует GCC версии 2.8.1, то вы можете указать другой каталог для установки GNAT и его собственной копии "gcc"версии 2.8.1. Используя такой метод, вам необходимо выполнить дополнительный шаг. Программа установки автоматически создает скрипт командного интерпретатора "shell", в котором содержатся установки некоторых переменных окружения необходимых GCC для поиска файлов GNAT. В результате, вы можете скопировать эти установки в ваши скрипты настройки среды (для "bash", как правило, это файл ".profile" в вашем домашнем каталоге).

Вам также необходимо указать каталог установки файлов GNAT в начале переменной окружения "PATH", для того чтобы GNAT не использовал версию "gcc"которая поставляется вместе с вашим дистрибутивом Linux. Например, это может быть выполнено с помощь команды:

```
export PATH="/usr/gnat/bin:$PATH"
```

Эту команду следует применять только тогда когда вы используете GNAT, поскольку она эффективно "прячет"копию "gcc"которая поставляется вместе с вашим дистрибутивом Linux.

В случаях когда окружение среды, соответствующее требованиям использования GNAT, не желательно устанавливать как окружение по умолчанию можно написать короткий скрипт командного интерпретатора "shell", который будет выполнять необходимые установки и запускать новый экземпляр командного интерпретатора.

26.2.2 Установка RPM-пакетов *ALT*/*ALR*

Кроме стандартного дистрибутива GNAT от *ACT*, существует альтернативный дистрибутив GNAT от *ALT* (*Ada for Linux Team*, WEB-страничка проекта доступна по адресу: "http://www.gnuada.org/alt.html") и альтернативный дистрибутив *ALR* (*Ada for Linux RU*, WEB-страничка проекта доступна по адресу: "http://www.prz.rzeszow.pl/ada/").

Версия дистрибутива GNAT от *ALT* построена с учетом использования GNAT в среде дисрибутивов системы Linux от *Red Hat*, *S.u.S.E.* или *Debian*. GNAT от *ALT* может быть также использован с дистрибутивами *Mandrake* и *Caldera*. В настоящий момент дистрибутив от *ALT* содержит компилятор GNAT версии 3.13p.

Дистрибутив *ALT* содержит поддержку для *ASIS*, *GLADE* и использует "родные нити"Linux (*native Linux threads*). Пакеты дистрибутива содержат "gnatgcc"(версия "gcc"с поддержкой GNAT и C++), отладчик

"gnatgdb"(версия "gdb", которая поддерживает исходные тексты на языке Ада), препроцессор "gnatprep"и другие утилиты разработки. Кроме того, дистрибутив от *ALT* включает множество дополнительных пакетов в которых находятся различные библиотеки и программы, которые могут быть полезны при разработке программного обеспечения на языке программирования Ада в среде операционной системы Linux.

Дистрибутив *ALR* построен в среде системы *Red Hat* Linux. Он наследует основные особенности дистрибутива *ALT*, но построен на основе более новой версии компилятора GNAT - 3.15p, и включает более новые версии пакетов дополнительного программного обеспечения. Кроме того, в этой сборке компилятор позволяет использовать в идентификаторах кириллицу в кодировках KOI8-R/KOI8-U. Необходимо заметить, что при дальнейшем упоминании в тексте дистрибутива *ALT* равноправно подразумевается дистрибутив *ALR*.

Следует учесть, что все пакеты *rpm* были построены с учетом сред дистрибутивов *Red Hat* и *S.u.S.E.*. Поэтому, для того чтобы проигнорировать предупреждающие сообщения о зависимости пакетов, которые могут возникнуть при попытке установить *rpm*-пакеты *ALT* в среде других дистрибутивов Linux, может потребоваться использование опции командной строки "-nodep" в команде "rpm".

Для установки *rpm*-пакетов дистрибутива от *ALT* необходимо выполнить следующее:

1. Загрузить и прочитать файл "readme".
2. Загрузить файл *rpm*-пакета "gnat-3.*xx*p-runtime*" (здесь, "*xx*" обозначает текущую версию GNAT, а "*" обозначает остаток имени файла). Для старых версий *rpm*-пакетов, содержимое этого пакета находится в пакете "gnat-3.*xx*p*".
3. Загрузить файл *rpm*-пакета "gnat-3.xxp". Для старых версий *rpm*-пакетов, содержимое этого пакета находится в пакете "gnat-3.*xx*p-devel*".
4. Выполнить команду: "rpm -i gnat-3.*xx*p-runtime*"
5. Выполнить команду: "rpm -i gnat-3.*xx*p*"
6. При необходимости, загрузить и установить любые дополнительные пакеты дистрибутива GNAT от *ALT*.

Файлы всех *rpm*-пакетов дистрибутива GNAT от *ALT* сконфигурированы для совместной работы с версией GNAT от *ALT*. Для их установки необходимо просто загрузить их с интернет-сайта *ALT* и выполнить команду установки "rpm -i ..." для соответствующего пакета.

С помощью CVS, система построения пакетов дистрибутива GNAT от *ALT* доступна для тех кто желает ознакомиться с деталями построения *rpm*-пакетов от *ALT*:

```
export CVSROOT=":pserver:anoncvs@hornet.rus.uni-stuttgart.de:/var/cvs"
cd $HOME
cvs login  # (use empty password)
cvs -z9 co -d ALT gnuada/alt-build
```

Глава 27

От исходного текста к загружаемому файлу программы

27.1 Соглашения GNAT по наименованиям файлов

27.1.1 Общие правила наименования файлов

Не зависимо от используемой операционной системы (Microsoft Windows или Linux) компилятор GNAT достаточно активно использует суффиксы имен файлов (иначе, расширения имен файлов). Ниже перечисляются общие соглашения GNAT по использованию суффиксов имен файлов:

Суффикс	**Пояснения**
`.ads`	файл с исходным текстом спецификации пакета Ады
`.adb`	файл с исходным текстом тела пакета Ады или Ада-программы
`.adc`	файл конфигурации GNAT
`.adt`	файл дерева зависимостей
`.ali`	файл содержащий информацию для связывания и отладки, который генерируется GNAT в процессе компиляции
`.xrb`	файл перекрестных ссылок генерируемый утилитой `gnatf` (GNAT версии 3.10p)

По умолчанию, имя файла определяется именем модуля, который содержится в этом файле. Имя файла формируется путем взятия полного расширенного имени модуля и замене разделительных точек символами дефиса ("-"). Следует заметить, что при формировании имени файла используются буквы только нижнего регистра.

Исключение из этого правила касается файлов чьи имена начинаются символами `a`, `g`, `i` или `s`, а следующим символом является символ дефиса. В этом случае вместо символа дефиса ("-") используется символ тильды ("~"). Смысл таких специальных правил заключается в том, что это позволяет избежать конфликта имен с файлами стандартной библиотеки, которые содержат дочерние модули пакетов *System*, *Ada*, *Interfaces* и *GNAT*, использующих для имен файлов префиксы "`s-`", "`a-`", "`i-`" или "`g-`", соответственно.

Следующий список демонстрирует некоторые примеры использования этих правил именования файлов:

Имя файла	**Пояснения**
`main.ads`	Спецификация главной программы
`main.adb`	Тело главной программы
`arith_functions.ads`	Спецификация пакета *Arith_Functions*
`arith_functions.adb`	Тело пакета *Arith_Functions*
`func-spec.ads`	Спецификация дочернего пакета *Func.Spec*
`func-spec.adb`	Тело дочернего пакета *Func.Spec*
`main-sub.adb`	Тело *Sub* субмодуля *Main*
`a~bad.adb`	Тело дочернего пакета *A.Bad*

Соблюдение этих правил при наличии длинных имен модулей (например, в случае глубокой вложенности модулей) может привести к чрезмерно длинным именам файлов. Для усечения длин имен файлов можно использовать "уплотнение"имен файлов. В частности, это может быть полезно когда в используемой операционной системе действуют ограничения на длину имени файла.

Естественно, алгоритм "уплотнения"имен файлов не может полностью гарантировать уникальность для всех возможных имен файлов. Это означает, что при использовании "уплотнения"имен файлов окончательная ответственность за уникальность имен файлов возлагается на разработчика приложения. В качестве альтернативы может быть использовано явное указание имен файлов (рассматривается далее).

При переносе Ада-программы из среды компилятора который поддерживает другие соглашения по наименованию файлов для создания файлов с исходными текстами, имена которых будут соответствовать соглашениям GNAT, может быть использована утилита "`gnatchop`".

27.1.2 Использование других имен файлов

В большинстве случаев следование принимаемым по умолчанию правилам именования файлов позволяет компилятору без каких-либо дополнительных указаний, самостоятельно определять имя файла в котором расположен соответствующий модуль.

Однако в некоторых случаях, в частности, при импортировании программы из окружения другого Ада-компилятора, для программиста может оказаться более удобной возможность непосредственного указания имен файлов в которых находятся определенные модули. *GNAT* позволяет указывать "случайные"имена файлов с помощью директивы компилятора "Source_File_Name", которая имеет следующий вид:

```
pragma Source_File_Name (My_Utilities.Stacks,
  Spec_File_Name => "myutilst_a.ada");
pragma Source_File_name (My_Utilities.Stacks,
  Body_File_Name => "myutilst.ada");
```

Первый аргумент директивы указывает имя модуля (в случае показанного примера — имя дочернего модуля). Второй аргумент использует форму именной ассоциации, которая указывает, что имя файла используется для спецификации или тела модуля. Непосредственное имя файла указывается строковым литералом.

Следует заметить, что директива указания имени файла с исходным текстом является конфигурационной директивой *GNAT*. Это подразумевает, что она обычно располагается в файле "`gnat.adc`", который используется для сохранения директив конфигурации применяемых для среды компиляции.

Таким образом, использование директивы указания имени файла с исходным текстом позволяет указывать абсолютно "случайные"имена файлов. Однако если указанное имя файла имеет расширение имени файла, которое отличается от "`.ads`" или "`.adb`", то при выполнении компиляции файла необходимо применять специальную опцию командной строки "`gcc`", которая указывает язык программирования используемый в файле. Такой опцией командной строки "`gcc`" является опция "**`-x`**". Для указания языка программирования она должна сопровождаться пробелом и именем используемого языка программирования, в данном случае — "**`ada`**":

```
gnatgcc -c -x ada peculiar_file_name.sim
```

Утилита "`gnatmake`" обрабатывает не стандартные имена файлов обычным образом, то есть не стандартное имя файла для главной программы используется как простой аргумент "`gnatmake`". Примечательно, что когда используется не стандартное расширение имени файла главной программы то оно должно быть обязательно указано в команде запуска утилиты "`gnatmake`".

27.1.3 Альтернативные схемы именования

Ранее мы рассмотрели использование директивы компилятора "Source_File_Name", которая позволяет использовать произвольные имена для отдельных файлов с исходными текстами. Этот подход требует наличия директивы для каждого файла именуемого произвольным образом. Таким образом, в случае разработки большой системы, это ведет к значительному увеличению размеров файла "`gnat.adc`" и, как следствие, может осложнить сопровождение проекта.

Начиная с версии 3.15, GNAT предусматривает возможность указания общей схемы именования файлов с исходными текстами, которая отличается от используемой по умолчанию стандартной схемы именования файлов. Для этого используются показанные ниже формы директивы компилятора "Source_File_Name":

```
pragma Source_File_Name (
   Spec_File_Name  => FILE_NAME_PATTERN
 [,Casing          => CASING_SPEC]
 [,Dot_Replacement => STRING_LITERAL]);

pragma Source_File_Name (
   Body_File_Name  => FILE_NAME_PATTERN
 [,Casing          => CASING_SPEC]
 [,Dot_Replacement => STRING_LITERAL]);

pragma Source_File_Name (
   Subunit_File_Name  => FILE_NAME_PATTERN
 [,Casing             => CASING_SPEC]
 [,Dot_Replacement    => STRING_LITERAL]);

FILE_NAME_PATTERN ::= STRING_LITERAL
CASING_SPEC ::= Lowercase | Uppercase | Mixedcase
```

Строка "`FILE_NAME_PATTERN`" является шаблоном имен файлов. Она содержит один символ звездочки, вместо которого подставляется имя модуля. Необязательный параметр "`Casing`" указывает используемый в имени файла регистр символов: "`Lowercase`" - нижний регистр (маленькие буквы), "`Uppercase`" - верхний регистр (большие буквы), "`Mixedcase`" - смешанное использование регистра символов. При отсутствии параметра "`Casing`", по умолчанию, используется "`Lowercase`".

Необязательный параметр, строка "`Dot_Replacement`", используется для подстановки вместо точек, которые присутствуют в именах субмодулей. Когда строка "`Dot_Replacement`" не указывается, разделяющие точки неизменно присутствуют в имени файла. Хотя показанный выше синтаксис демонстрирует, что параметр "`Casing`" указывается перед параметром "`Dot_Replacement`", допускается запись этих параметров в противоположной последовательности.

Из показанной выше формы директивы "Source_File_Name" видно, что возможно указание различных схем именования для тел, спецификаций и субмодулей. Достаточно часто для именования субмодулей желательно использовать такое же правило как и для тел модулей. В таком случае можно не указывать правило "`Subunit_File_Name`", и для именования субмодулей будет использоваться правило "`Body_File_name`".

Указание отдельного правила для именования субмодулей может быть использовано в случае реализации необычного окружения компиляции (например, при компиляции в одном каталоге), которое содержит субмодули и дочерние модули с одинаковыми именами. Хотя оба модуля не могут одновременно присутствовать в одном разделе программы, стандарт допускает (но не требует) возможность сосуществования двух таких модулей в одном окружении.

Трансляция имен файлов включает следующие шаги:

- Если для данного модуля существует соответствующая директива "Source_File_Name", то всегда используется информация указанная в этой директиве, а любые другие шаблонные правила игнорируются.
- Если указана директива "Source_File_Name" задающая тип шаблона, который применим для модуля, то полученное имя файла будет использовано в случае существования соответствующего файла. Когда обнаружено совпадение более одного шаблона, то первым проверяется последний шаблон, и, в итоге, будет использован результат первой успешной проверки существования файла.
- Если директива "Source_File_Name" задающая тип шаблона применимого для модуля, для которого существует соответствующий файл, отсутствует, то используется правило именования, которое применяется по умолчанию и является стандартным для GNAT.

Как пример использования этого механизма рассмотрим распространенную схему именования, когда для имен файлов используется нижний регистр, разделительная точка неизменно копируется в результирующее имя файла, имена файлов спецификаций заканчиваются как "`.1.ada`", а имена файлов тел заканчиваются как "`.2.ada`". GNAT будет следовать этой схеме при наличии двух директив:

```
pragma Source_File_Name
  (Spec_File_Name => "*.1.ada");
pragma Source_File_Name
  (Body_File_Name => "*.2.ada");
```

Стандартная, используемая по умолчанию схема именования фактически реализуется следующими директивами, которые устанавливаются по умолчанию внутренне:

```
pragma Source_File_Name
  (Spec_File_Name => "*.ads", Dot_Replacement => "-");
pragma Source_File_Name
  (Body_File_Name => "*.adb", Dot_Replacement => "-");
```

В качестве заключительного примера, рассмотрим реализацию схемы именования, которую использует один из компиляторов Ada83: как символ-разделитель для субмодулей используется "__" (два символа подчеркивания); файлы спецификаций идентифицируются добавлением "_.ADA"; файлы тел идентифицируются добавлением ".ADA"; файлы субмодулей идентифицируются добавлением ".SEP". Для всех имен файлов используется верхний регистр символов. Естественно, что дочерние модули - не представлены, поскольку это компилятор Ada83, однако, будет логично расширить эту схему указанием использовать двойное подчеркивание в качестве разделителя для дочерних модулей.

```
pragma Source_File_Name
  (Spec_File_Name => "*_.ADA",
   Dot_Replacement => "__",
   Casing = Uppercase);
pragma Source_File_Name
  (Body_File_Name => "*.ADA",
   Dot_Replacement => "__",
   Casing = Uppercase);
pragma Source_File_Name
  (Subunit_File_Name => "*.SEP",
   Dot_Replacement => "__",
   Casing = Uppercase);
```

27.2 Сборка первой программы

Используя любой текстовый редактор вашей системы (примечание для пользователей Windows: этим текстовым редактором не может быть MS Word!!!), создайте файл "hello.adb", который содержит следующую программу на языке Ада:

```
with Ada.Text_IO;
use  Ada.Text_IO;
procedure Hello is
begin
    Put_Line( "Hello World!" );
end Hello;
```

Эта программа будет печатать на экране простое сообщение приветствия. После сохранения файла, для построения выполняемого файла программы необходимо выполнить команду "gnatmake":

```
gnatmake hello.adb
```

Если при написании файла с исходным текстом были допущены ошибки, "gnatmake" покажет вам номер строки, номер символа в строке и сообщение об ошибке, описывающее суть возникшей проблемы. Если текст программы был написан без ошибок, то вы получите исполняемый файл "hello.exe", если вы используете Windows, или "hello", если вы используете Linux. Запустить полученную программу на выполнение можно командой:

```
hello
```

или, в некоторых дистрибутивах Linux:

```
./hello
```

В результате, на экране дисплея появится сообщение:

```
Hello World!
```

После построения этого проекта, каталог будет содержать следующие файлы:

Файл	**Пояснения**
`hello.adb`	файл с исходным текстом программы на языке Ада
`hello.o`	двоичный файл созданный компилятором, который содержит объектный код
`hello.ali`	дополнительный информационный файл созданный компилятором (***A**da **L**ibrary **I**nformation file*)
`hello.exe` или `hello`	исполнимый файл программы

Мы можем спокойно удалить файлы "`hello.o`" и "`hello.ali`", которые в данном случае являются дополнительными информационными файлами. Однако, при разработке больших проектов, содержащих большое количество файлов, сохранение подобных файлов в последствии может ускорить процесс построения проекта.

27.3 Три этапа сборки проекта

Весь процесс построения проекта осуществляется в три этапа, которые утилита "`gnatmake`" выполняет автоматически:

1. **Компиляция** (*compiling*): утилита "`gnatmake`"осуществляет проверку вашего файла на наличие в нем каких-либо ошибок. Если ошибки не обнаружены, то выполняется создание объектного файла, содержащего двоичную версию программы. При обнаружении ошибок, работа утилиты "`gnatmake`" завершается.

2. **Связывание** (*binding*): утилита "`gnatmake`"проверяет согласованность обновления версий всех файлов проекта. Если при этом обнаруживаются файлы которые нуждаются в перекомпиляции, "`gnatmake`" осуществит компиляцию таких файлов.

3. **Компоновка** (*linking*): утилита "`gnatmake`" осуществляет комбинирование всех объектных файлов проекта для создания результирующего исполняемого файла программы.

Любой из этих этапов построения проекта может быть выполнен "вручную". Например, для полностью "ручной"сборки программы "`hello.adb`" можно выполнить следующее:

1. **Компиляция** программы с помощью команды:

   ```
   gcc -c hello.adb
   ```

   Примечание: при использовании в операционной системе Linux дистрибутива GNAT от *ALT* вместо команды "`gcc`" применяется команда "`gnatgcc`".

2. **Связывание** с помощью команды:

   ```
   gnatbind hello.ali
   ```

3. **Компоновка** программы с помощью команды:

   ```
   gnatlink hello.ali
   ```

В случаях простых проектов проще и удобнее использовать автоматическое управление сборкой проекта. Необходимость сборки "вручную"возникает в случаях когда какой-либо этап сборки нуждается в настройках, которые должны соответствовать определенным требованиям или условиям сборки проекта.

27.3.1 Опции компилятора

Программа "`gcc`" (или "`gnatgcc`", для дистрибутива *ALT*) принимает следующие опции командной строки, которые позволяют непосредственно управлять процессом компиляции:

-b *target* — Компиляция программы которая предназначена для запуска на платформе *target*. Здесь *target* подразумевает имя конфигурации системы. Если *target* не является той же самой системой на которой выполняется компиляция, то система компилятора *GNAT* должна быть построена и сконфигурирована так, чтобы обеспечивать кросс-компиляцию для платформы *target*.

-B*dir* — Загрузить компилятор (например, Ада-компилятор "gnat1") из каталога *dir* вместо того, который определен соглашениями по умолчанию. Использование этой опции имеет смысл только в случае доступности различных версий компилятора GNAT. За более подробными сведениями следует обратиться к руководству по "gcc". Вместо этой опции обычно используются опции **-b** или **-V**.

-c — Выполнить компиляцию. Эта опция всегда используется при выполнении компиляции Ада-программы. Примечательно, что можно использовать "gcc" (или "gnatgcc") без опции **-c** для выполнения компиляции и компоновки за один шаг. Это вызвано необходимостью вызова редактора связей, но в текущий момент "gnatgcc"не может быть использован для запуска редактора связей GNAT.

-g — Выполнить генерацию отладочной информации. Эта информация сохраняется в объектном файле и, затем, копируется из него в результирующий выполняемый файл компоновщиком, после чего, отладочная информация может быть прочитана и использована отладчиком. Опцию **-g** необходимо использовать если планируется последующее использование отладчика (как правило "gdb").

-I*dir* — Указывает компилятору каталог *dir* для поиска файлов с исходными текстами необходимыми для текущей компиляции.

-I- — Указывает компилятору, что не нужно искать файлы с исходными текстами в каталогах указанных в командной строке.

-o *file* — Эта опция используется для перенаправления вывода генерируемого компилятором объектного файла и ассоциируемого с ним ALI-файла. Данная опция нуждается в аккуратном использовании поскольку это может привести к тому, что объектный файл и ALI-файл будут иметь различные имена, что в результате может "запутать"редактор связей "gnatbind" и компоновщик "gnatlink".

-O[*n*] — Здесь, значение *n* определяет уровень оптимизации:

- n = 0 — Оптимизация отсутствует, установлено по умолчанию если не указана опция **-O**
- n = 1 — Нормальная оптимизация, установлено по умолчанию если указана опция **-O** без операнда
- n = 2 — Экстенсивная оптимизация
- n = 3 — Экстенсивная оптимизация с автоматической встроенной подстановкой (*inline*). Это применяется только для встроенной подстановки внутри модулей.

-S — Используется совместно с опцией **-c** для генерации файла с исходным текстом на языке ассемблера (расширение имени файла .s) вместо файла с объектным кодом. Это может быть полезно когда необходима проверка генерируемого кода для ассемблера.

-v — Установить "многословный"режим, который показывает все команды, генерируемые "gcc" ("gnatgcc"). Обычно используется только в целях отладки или когда необходимо убедиться в правильности используемой/запускаемой версии компилятора.

-V *ver* — Запустить на выполнение версию компилятора *ver*. Подразумевается версия "gcc" ("gnatgcc"), а не версия GNAT.

-funwind-tables — Эта опция производит генерацию объектного файла с *unwind table information*. Это требуется для обработки исключений с нулевыми затратами производительности, которая использует трассировочные способности библиотеки GNAT.

-gnata — Активирует использование контрольных инструкций (*assertion*) устанавливаемых директивами компилятора "**pragma** Assert" и "**pragma** Debug".

-gnatb — Генерировать краткие сообщения на *stderr* даже при установленном опцией **-v** "многословном"режиме.

-gnatc — Выполнить проверку синтаксиса и семантики (без генерации какого-либо выходного кода).

-gnatD — Выводить расширенные исходные тексты для отладки на уровне исходных текстов. Эта опция подавляет генерацию информации о перекрестных ссылках (см. опцию **-gnatx**).

-gnate — Принудительная генерация сообщений об ошибках (используется в случае разрушения компилятора при компиляции).

-gnatE — Выполнить полную проверку динамической элаборации.

-gnatf — Полные ошибки. Множественные ошибки в одной строке, все неописанные ссылки.

-gnatF — Все внешние имена приводить к верхнему регистру.

-gnatg — Активация проверки стиля написания исходного текста.

-gnatG — Отобразить генерируемый расширенный код в виде исходного текста.

-gnati*c* — Установить кодировку символов для идентификаторов в исходном тексте (*c*=1/2/3/4/8/p/f/n/w)

-gnath — Выдать информацию об использовании. Вывод осуществляется на *stdout*.

-gnatk*n* — Ограничить длину имен файлов длиной *n* (1-999) символов (k = krunch).

-gnatl — Выводить полный листинг исходных текстов, включая встроенные сообщения об ошибках.

-gnatm*n* — Ограничить число детектируемых ошибок величиной *n* (1-999).

-gnatn — Активировать встроенную подстановку (*inline*) в пределах модуля для подпрограмм, которые указаны в директивах компилятора `Inline`.

-fno-inline — Подавить любые встроенные подстановки (*inline*), даже если активированы другие опции оптимизации и встроенная подстановка.

-fstack-check — Активировать проверку состояния стека.

-gnato — Активация прочих проверок, которые обычно не используются по умолчанию, включая проверку численного переполнения и проверку доступа до проверки элаборации.

-gnatp — Подавить все проверки.

-gnatq — При обнаружении ошибки синтаксического разбора, попытаться осуществить семантическую проверку.

-gnatP — Включение опроса. Это требуется на некоторых системах (в частности, для Windows NT) для осуществления асинхронного принудительного прекращения и способности асинхронной передачи управления.

-gnatR — Вывести информацию о представлении описанных типов массивов и записей.

-gnats — Выполнить синтаксическую проверку.

-gnatt — Сгенерировать выходной файл дерева.

-gnatT *nnn* — Установить квант времени (*time slice*) в указанное число миллисекунд.

-gnatu — Вывести список модулей текущей компиляции.

-gnatU — Обозначить все сообщения об ошибках уникальной строкой "error:".

-gnatv — Установить "многословный"режим. Выполняется полный вывод сообщений об ошибках с выводом строк исходного текста на `stdout`.

-gnatw*m* — Установить режим выдачи предупредительных сообщений. Где *m*=`s,e,l` соответственно означает: подавление сообщений (*suppress*), трактовать как ошибку (*treat as error*), предупреждения элаборации (*elaboration warnings*).

-gnatW*e* — Установить кодировку для *wide character* (*e*=n/h/u/s/e/8).

-gnatx — Подавление генерации информации о перекрестных ссылках.

-gnatw*m* — Режим предупреждающих сообщений.

-gnaty — Включение встроенной проверки стиля.

-gnatz*m* — Генерация и компиляция распределенных "заглушек". Где, *m*=r/c "заглушка"приемника (*receiver*)/вызова (*caller*).

-gnat83 — Установить принудительные ограничения Ada 83.

-gnat95 — Установить стандартный режим Ada 95.

27.3.1.1 Проверка ошибок во время выполнения программы

Ада предоставляет обширные возможности для проверки ошибок во время выполнения программы. По умолчанию, GNAT отключает некоторые проверки для повышения скорости выполнения программы. Для включения всех проверок, необходимо использовать опции "**-gnato -gnatE**". Для того чтобы отключить все проверки, необходимо использовать опцию "**-gnatp**".

27.3.1.2 Проверка ошибок в исходном тексте без компиляции

Если вы используете версию GNAT 3.10p, то для проверки ошибок в исходном тексте программы, без выполнения фактической компиляции исходных текстов, можно использовать утилиту "`gnatf`". Для более новых версий GNAT, начиная с версии 3.11p, для проверки ошибок в исходном тексте программы без компиляции, можно использовать команду "`gcc`" с опцией "**-gnatc**".

27.3.1.3 Обнаружение большого количества ошибок при компиляции

В случае обнаружения очень большого количества ошибок компиляции в исходном тексте, может оказаться полезным перенаправление вывода сообщений об ошибках в отдельный файл (например, путем использования "`gcc . . . 2> temp.out`"). После этого, содержимое такого файла с сообщениями об ошибках может быть исследовано с помощью соответствующих средств системы.

27.3.2 Связывание Ада-программы

При использовании таких языков программирования как C и C++, как только выполнена компиляция файлов с исходными текстами остается только один шаг до построения результирующей исполняемой программы, которым является компоновка в единое целое полученных в результате компиляции объектных файлов. Это подразумевает, что существует возможность компоновки (иначе - сборки) результирующей исполняемой программы из несогласованных версий объектных файлов, что означает попадание в результирующую программу модулей которые включают различные версии одного и тот же заголовочного файла.

Правила Ады не позволяют скомпоновать подобную программу. Например, если два каких-либо клиентских модуля используют различные версии одного и того же пакета, то выполнить построение программы, которая одновременно включает в себя оба этих модуля, не возможно. Такие правила принудительно навязываются редактором связей GNAT ("`gnatbind`"), который также осуществляет определение последовательности элаборации в соответствии с требованиями языка Ада.

Запуск редактора связей GNAT выполняется после того как созданы все объектные файлы, которые необходимы для компоновки результирующей программы. При запуске, редактор связей "`gnatbind`" получает имя головного модуля программы и путем чтения соответствующих `ALI`-файлов определяет перечень требуемых для компоновки этой программы модулей. При обнаружении попытки скомпоновать программу из несогласованных версий объектных файлов или при отсутствии допустимой последовательности элаборации редактор связей "`gnatbind`" выдает соответствующие сообщения об ошибках.

В случае отсутствия ошибок, редактор связей "`gnatbind`" осуществляет генерацию главного файла программы (по умолчанию, генерируется исходный текст на языке Ада), который содержит вызовы процедур элаборации для компилируемых модулей, которые нуждаются в элаборации, с последующим вызовом головной программы. В результате компиляции этого файла с исходным текстом генерируется объектный файл главной программы. Имя главного файла программы, который генерируется редактором связей "`gnatbind`", имеет следующий вид: "`b~`*`xxx`*`.adb`", а соответствующий ему файл спецификации - "`b~`*`xxx`*`.ads`", — где "*`xxx`*" является именем головного модуля программы.

В заключение, для компоновки результирующей исполняемой программы, используется компоновщик "`gnatlink`", которому предоставляется объектный файл главной программы, полученный на этапе связывания, а также все объектные файлы модулей Ады, которые необходимы для компоновки результирующей исполняемой программы.

27.3.2.1 Опции редактора связей `gnatbind`

Программа "`gnatbind`" является редактором связей (*binder*) системы компилятора GNAT. Она принимает следующие опции командной строки, которые позволяют непосредственно управлять процессом связывания:

-aO — Определяет каталог в котором будет производиться поиск `*.ALI`-файлов.

-aI — Определяет каталог в котором будет производиться поиск файлов с исходными текстами.

-A — Генерировать программу редактора связей на Аде (установлено по умолчанию).

-b — Генерировать краткие сообщения в *stderr* даже когда установлен режим многословных (*verbose*) сообщений.

-c — Только проверка, генерация выходного файла редактора связей отсутствует.

-C — Генерировать программу редактора связей на *C*.

-e — Выводить полный список зависимостей последовательности элаборации.

-E — Сохранять обратную трассировку в точках возникновения исключений для целевых платформ которые ее поддерживают. Действует по умолчанию, с нулевыми затратами производительности для механизма исключений. В настоящее время, опция поддерживается только для *Solaris*, *Linux* и *Windows* на платформе *ix86*.

Для *Solaris* и *Linux* необходимо явное использование флага **-funwind-tables** для "`gcc`" ("`gnatgcc`"), при компиляции каждого файла приложения. Дополнительная информация находится в пакетах *GNAT.Traceback* и *GNAT.Traceback.Symbolic*.

Чтобы активировать действие этой опции для *Windows* нет необходимости в использовании дополнительных опций, однако нельзя использовать флаг **-fomit-frame-pointer** для "`gnatgcc`".

-f — Полная семантика элаборации, в соответствии с требованиями стандарта.

-h — Вывод справочного сообщения (*help*) об использовании.

-I — Определяет каталог для поиска файлов с исходными текстами и `*.ALI`-файлов.

-I- — Не производить поиск файлов с исходными текстами в текущем каталоге, откуда "`gnatbind`" был запущен, и не производить поиск `*.ALI`-файлов в каталогах указанных в командной строке.

-l — Отобразить выбранный порядок элаборации.

-Mxyz — Переименовать сгенерированную главную программу из `main` в `xyz`.

-m*n* — Ограничить число обнаруженных ошибок до *n* (1-999).

-n — Главная программа отсутствует.

-nostdinc — Не производить поиск файлов с исходными текстами в системных каталогах по умолчанию.

-nostdlib — Не производить поиск библиотечных файлов в системных каталогах по умолчанию.

-o *file* — Указывает имя *file* для выходного файла (по умолчанию имя выходного файла задается как *b~xxx.adb*). Примечательно, что при использовании этой опции компоновка приложения должна быть выполнена вручную, то есть, компоновщик "`gnatlink`" не может быть использован автоматически.

-O — Вывод списка объектов.

-p — Пессимистический (худший случай) порядок элаборации.

-s — Требуется присутствие всех файлов с исходными текстами.

-static — Выполнить компоновку приложения со статическими библиотеками времени выполнения GNAT.

-shared — Выполнить компоновку приложения с динамически связываемыми библиотеками времени выполнения GNAT, если они доступны.

-t — Допускать ошибку метки времени создания и другие ошибки целостности/согласованности.

-T*n* — Установить значение кванта времени (*time slice*) в *n* миллисекунд. Нулевое значение подразумевает отсутствие квантования по времени, а также указывает могозадачному окружению времени выполнения на необходимость максимального соответствия требованиям приложения *D* (*Annex D*) *RM*.

-v — Режим многословных (*verbose*) сообщений. Осуществляет вывод сообщений об ошибках, заголовков и общий отчет на *stdout*.

-w*x* — Установка режима предупредительных (*warning*) сообщений. (***x***=s/e для подавления / для трактования как ошибки).

-x — Исключить проверку целостности/согласованности для файлов с исходными текстами (проверка выполняется только для объектных файлов).

-z — Главная подпрограмма отсутствует.

Следует заметить, что данный список опций может быть получен при запуске программы редактора связей "gnatbind" без аргументов.

27.3.2.2 Правила поиска файлов для gnatbind

Программа редактора связей "gnatbind" принимает имя ALI-файла как аргумент и нуждается в указании файлов с исходными текстами и других ALI-файлов для проверки целостности и согласованности всех объектов проекта.

При поиске файлов с исходными текстами используются те же правила, которые используются для "gcc" ("gnatgcc"). Для ALI-файлов используются следующие правила поиска:

1. Каталоги с ALI-файлами, указанные в командной строке, если не указана опция "**-I-**".
2. Все каталоги с ALI-файлами, указанные в командной строке с помощью опции "**-I**", в соответствии с порядком указания каталогов.
3. В каждом каталоге, который перечислен в значении переменной окружения "ADA_OBJECTS_PATH". Значение этой переменной окружения строится также как значение переменной окружения "PATH": список каталогов разделенных двоеточием.
4. Содержимое файла *ada_object_path*, который является составной частью инсталляции GNAT и используется для хранения информации о стандартных библиотеках, таких как *GNAT Run Time Library (RTL)* пока в командной строке не указана опция "**-nostdlib**".

Опция "**-I**" редактора связей "gnatbind" используется для указания пути поиска как для файлов с исходными текстами, так и для библиотечных файлов. Чтобы указать путь поиска только для файлов с исходными текстами необходимо использовать опцию "**-aI**", а при необходимости указания пути поиска только для библиотечных файлов необходимо использовать опцию "**-aO**". Это подразумевает, что для редактора связей "gnatbind", указание опции "**-I***dir*" будет эквивалентно одновременному указанию опций "**-aI***dir*" и "**-aO***dir*". Редактор связей "gnatbind" генерирует файл связывания (файл с исходным текстом на языке *C*) в текущем рабочем каталоге.

Пакеты *Ada*, *System* и *Interfaces* совместно со своими дочерними модулями, а также пакет GNAT совместно со своими дочерними модулями, которые содержат множество дополнительных библиотечных функций, формируют окружение времени выполнения реализации системы компилятора GNAT (*GNAT Run-Time Library*). Исходные тексты для этих модулей необходимы компилятору и храняться в одном каталоге. ALI-файлы и объектные файлы, которые сгенерированы при компиляции библиотеки времени выполнения (*RTL*), необходимы редактору связей "gnatbind" и компоновщику "gnatlink". Эти файлы сохраняются с одном каталоге, как правило отличном от каталога в котором храняться файлы с исходными текстами. При обычной инсталляции, нет необходимости в спецификации этих каталогов при выполнении компиляции или связывания (*binding*). Обнаружение этих файлов возможно как при использовании переменных окружения, так и при использовании встроенных соглашений по умолчанию.

В дополнение к упрощению доступа к библиотеке времени выполнения (*RTL*), основным предназначением использования путей поиска является компиляция исходных текстов из различных каталогов. Это позволяет сделать окружение разработки более гибким.

27.3.3 Компоновка проекта

27.3.3.1 Компоновщик gnatlink

Компоновщик "gnatlink" используется для компоновки Ада-программ и построения исполняемых файлов. Эта простая программа вызывает компоновщик системы (через команду "gcc"/"gnatgcc") и предоставляет ему корректный список ссылок на объектные и библиотечные файлы. Программа "gnatlink" выполняет

автоматическое определение списка необходимых файлов для сборки Ада-программы. Для определения этого списка файлов "gnatlink" использует файл сгенерированный редактором связей "gnatbind".

Команда запуска компоновщика "gnatlink" имеет следующий вид:

```
gnatlink [switches] mainprog[.ali] [non-Ada objects] [linker options]
```

В данном случае, файл "mainprog.ali" является ALI-файлом главной программы. Расширение имени файла ".ali" может не указываться, поскольку такое расширение имени файла подразумевается по умолчанию. Из этой команды компоновщик "gnatlink" определяет соответствующий файл "b~*mainprog*.adb", который генерируется редактором связей "gnatbind", и, используя информацию из этого файла, вместе со списком не Ада-объектов и опций компоновщика, конструирует команду для запуска компоновщика системы, в результате выполнения которой выполняется построение результирующего исполняемого файла.

Аргументы команды, которые следуют за "mainprog.ali" передаются компоновщику без какой-либо дополнительной интерпретации. Обычно такие аргументы содержат имена объектных файлов модули которых написаны на других языках программирования, а также ссылки на библиотечные файлы которые необходимы для модулей написанных на других языках программирования или используются директивами компилятора "**pragma** Import" в модулях Ада-программы.

Список "*linker options*" (опции компоновщика) - это не обязательный список опций, которые специфичны для компоновщика, По умолчанию, в качестве компоновщика используется "gcc" ("gnatgcc"), который вызывает соответствующий системный компоновщик, обычно называемый "ld". При этом, как обычно, могут быть представлены стандартные опции компоновщика, такие как "-lmy_lib" или "-Ldir". Для опций, которые не распознаются "gcc" ("gnatgcc") как опции компоновщика должны использоваться "-Xlinker" или "-Wl,". Для получения более детальной информации необходимо обратиться к документации на GCC. Примером того как выполнить генерацию компоновщиком файла-карты (*linker map file*), подразумевая, что используется системный компоновщик "ld", может служить следующее:

```
gnatlink my_prog -Wl,-Map,MAPFILE
```

Компоновщик "gnatlink" определяет список объектов, необходимых для Ада-программы, и предваряет их списком объектов передаваемых компоновщику. Кроме того, "gnatlink" собирает любые аргументы установленные директивой компилятора "**pragma** Linker_Options" и добавляет их к списку аргументов передаваемых компоновщику системы.

27.3.3.2 Опции компоновщика gnatlink

Следующий список перечисляет опции которые допускается использовать с компоновщиком "gnatlink":

- **-A** — Указывает "gnatlink", что сгенерированный редактором связей "gnatbind" код является Ада-кодом. Это принимается по умолчанию.
- **-C** — Указывает "gnatlink", что сгенерированный редактором связей "gnatbind" код является C-кодом.
- **-g** — Опция включения отладочной информации, использование которой приводит к тому, что Ада-файл редактора связей "gnatbind" ("b~*mainprog*.adb") будет скомпилирован с опцией "-g".

  Дополнительно, при указании этой опции, редактор связей "gnatbind" не будет удалять файлы "b~*mainprog*.adb", "b~*mainprog*.o" и "b~*mainprog*.ali", которые удаляются в случае отсутствия опции "-g". Та же самая процедура выполняется при генерации редактором связей "gnatbind" C-файла, в результате указания для редактора связей опции "-C". В этом случае файлы, которые генерирует редактор связей "gnatbind", будут иметь следующие имена: "b_*mainprog*.c" и "b_*mainprog*.o".
- **-n** — Не выполнять компиляцию файла сгенерированного редактором связей "gnatbind". Это может быть использовано при перезапуске компоновщика с различными опциями, но при этом нет необходимости выполнять перекомпиляцию файла, который сгенерирован редактором связей "gnatbind".
- **-v** — Использование этой опции приводит к выводу дополнительной информации, включая полный список подключаемых объектных файлов (такой режим также называют "многословным"). Эта опция полезна когда необходимо видеть множество объектных файлов используемых на этапе компоновки проекта.

-v -v — "Очень многословный режим". Такой режим указывает, что компилятор, при компиляции файла который сгенерирован редактором связей "gnatbind", и системный компоновщик должны работать в "многословном"режиме.

-o *exec-name* — "*exec-name*" указывает альтернативное имя для генерируемого исполняемого файла программы. Если эта опция не указана, то имя исполняемого файла программы будет таким же как и имя головного модуля. Например, команда

```
gnatlink try.ali
```

будет создавать исполняемый файл с именем "try".

-b *target* — Компиляция программы для запуска на платформе "*target*", при этом "*target*" определяет имя системной конфигурации. Для выполнения этого, необходимо наличие построенного кросс-компилятора GNAT, если платформа "*target*" не является хост-системой.

-B*dir* — Загрузить исполняемый файл компилятора (например, "gnat1" - Ада-компилятор) из каталога "*dir*", вместо каталога по умолчанию. Эта опция может быть использована только при наличии множества версий компилятора GNAT. Вместо этой опции могут быть использованы опции "-b" или "-V".

-GCC=*compiler_name* — Эта опция указывает программу "*compiler_name*", которая будет использована для компиляции файла сгенерированного редактором связей "gnatbind". По умолчанию - используется программа "gcc" ("gnatgcc"). Если указание "*compiler_name*" содержит пробелы или другие разделительные символы, то "*compiler_name*" необходимо заключать в кавычки. В качестве примера, "-GCC="foo -x -y"" указывает "gnatlink", что в качестве компилятора необходимо использовать "foo -x -y". Примечательно, что опция "-c" всегда вставляется после имени команды. Таким образом, показанный выше пример команды компилятора, которая будет использована "gnatlink", в результате, будет иметь вид "foo -c -x -y".

-LINK=*name* — Данная опция указывает имя "*name*" используемого компоновщика системы. Это удобно при работе с программами части которых написаны на разных языках программирования, например, *c++* требует использования своего собственного компоновщика. Когда эта опция не указана, то используется имя компоновщика по умолчанию - "gcc" ("gnatgcc").

27.3.4 Утилита gnatmake

Как уже было показано ранее, утилита "gnatmake" используется для компиляции Ада-программ. Например:

```
gnatmake main.adb
```

Она осуществляет проверку всех пакетов от которых зависит программа "main.adb" и автоматически компилирует любой пакет, который нуждается в компиляции или перекомпиляции. После завершения процесса компиляции утилита "gnatmake" автоматически выполняет процесс связывания и компоновки, для получения, в результате, исполняемого файла программы.

Бывают случаи, когда необходимо выполнить только компиляцию проекта и не выполнять связывание и компоновку. Это можно сделать используя опцию "**-n**":

```
gnatmake -n main.adb
```

27.3.4.1 Опции gnatmake

В команде запуска утилиты "gnatmake" могут быть использованы различные опции, которые управляют процессом сборки проекта. Следующий список перечисляет опции, которые допускается использовать с утилитой "gnatmake":

-GCC=*compiler_name* — Указывает программу используемую для компиляции. По умолчанию используется "gcc" ("gnatgcc").

Если "*compiler_name*" содержит пробелы или другие разделительные символы, то "*compiler_name*" должно заключаться в кавычки.

Как пример, "-GCC="foo -x -y"" указывает утилите "gnatmake", что в качестве команды запуска компилятора необходимо использовать команду "foo -x -y".

Примечательно, что опция "-c" всегда вставляется после имени команды. Таким образом, показанный выше пример команды компилятора, которая будет использована утилитой "gnatmake", в результате, будет иметь вид "foo -c -x -y".

-GNATBIND=*binder_name* — Указывает программу используемую в качестве редактора связей. По умолчанию используется редактор связей "gnatbind".

Если "*binder_name*" содержит пробелы или другие разделительные символы, то "*binder_name*" должно заключаться в кавычки.

Как пример, "-GNATBIND="bar -x -y"" указывает утилите "gnatmake", что в качестве команды запуска редактора связей необходимо использовать команду "bar -x -y".

Опции редактора связей, которые обычно принимаются утилитой "gnatmake"для передачи их редактору связей, будут добавлены после "bar -x -y".

-GNATLINK=*linker_name* — Указывает программу используемую в качестве компоновщика. По умолчанию используется компоновщик "gnatlink".

Если "*linker_name*" содержит пробелы или другие разделительные символы, то "*linker_name*" должно заключаться в кавычки.

Как пример, "-GNATLINK="lan -x -y"" указывает утилите "gnatmake", что в качестве команды запуска компоновщика необходимо использовать команду "lan -x -y".

Опции компоновщика, которые обычно принимаются утилитой "gnatmake"для передачи их компоновщику, будут добавлены после "lan -x -y".

-a — Эта опция указывает утилите "gnatmake", что в процессе построения приложения, необходимо выполнять анализ всех файлов, включая внутренние системные файлы GNAT (например, предопределенные файлы Ада-библиотеки), а также все заблокированные файлы. Заблокированными файлами являются файлы у которых запись в соответствующие ALI-файлы запрещена.

По умолчанию, утилита "gnatmake" не выполняет анализ таких файлов, поскольку подразумевается, что внутренние системные файлы GNAT обновлены, а также то, что все заблокированные файлы правильно установлены. Примечательно, что если для таких файлов существует какая-либо ошибка инсталляции, то она может быть обнаружена с помощью редактора связей. Реально, необходимость использования этой опции может возникнуть только в случае если вы непосредственно работаете с исходными текстами GNAT.

При необходимости полной перекомпиляции исходных текстов приложения, включая исходные тексты файлов библиотеки времени выполнения, может также оказаться полезным использование опции "-f" совместно со специальными директивами управления конфигурацией компилятора, подобными не стандартной директиве компилятора "Float_Representation".

По умолчанию, "gnatmake -a" выполняет компиляцию всех внутренних файлов GNAT путем использования "gnatgcc -c -gnatg", а не "gnatgcc -c".

-c — Выполнить только компиляцию и не выполнять связывание и компоновку.

Выполняется по умолчанию, если корневой модуль, который специфицирован как *file_name* не является головным модулем.

В другом случае, утилита "gnatmake" будет пытаться выполнить связывание и компоновку, пока все объекты не будут обновлены и результирующий выполняемый файл не будет новее объектов.

-f — Принудительная перекомпиляция.

Выполнить перекомпиляцию всех файлов с исходными текстами, даже в случае если некоторые объектные файлы обновлены, но при этом не выполнять перекомпиляцию предопределенных внутренних файлов GNAT или заблокированных файлов (файлы для которых запрещена запись в соответствующие ALI-файлы), пока не указана опция "-a".

-i — В обычном режиме, утилита "gnatmake" выполняет компиляцию всех объектных файлов и ALI-файлов только в текущем каталоге. При использовании опции "**-i**", существующие в различных каталогах объектные и ALI-файлы будут соответственно переписаны. Это подразумевает, что утилита "gnatmake" способна правильным образом выполнять автоматическую обработку и обновление файлов большого проекта, у которого различные файлы размещаются в различных каталогах. При этом, если объектные и ALI-файлы не будут обнаружены в путях поиска объектных файлов Ады, то будут созданы новые объектные и ALI-файлы в каталоге, который содержит файлы с компилируемыми исходными текстами. Если желательна другая организация, когда объектные файлы и файлы с исходными

текстами храняться в отдельных каталогах, может быть использован удобный прием при котором в требуемых каталогах создаются ALI-файлы "заглушки". При обнаружении таких файлов "заглушек", утилита "gnatmake" будет вынуждена выполнить перекомпиляцию соответствующих файлов с исходными текстами, и она поместит результирующие объектные и ALI-файлы в те каталоги, в которых были обнаружены файлы "заглушки".

-j*n* — Использовать "*n*" процессов для выполнения перекомпиляции.

В этом случае, на многопроцессорной системе, компиляция будет выполняться параллельно. В случае ошибок компиляции, сообщения от различных процессов компиляции могут быть получены в разбросанном порядке (однако "gnatmake" в завершении представит полностью упорядоченный список ошибочных компиляций). Если это вызывает проблемы, то следует перезапустить процесс сборки установив "*n*" в 1, для получения более чистого списка сообщений.

-k — Продолжать выполнение сборки проекта после обнаружения ошибки компиляции настолько долго, насколько это возможно.

При использовании этой опции, в случае ошибок компиляции, после завершения своей работы, утилита "gnatmake" выдает список файлов с исходными текстами при компиляции которых были обнаружены ошибки.

-m — Эта опция указывает, что необходимо выполнить минимально необходимое количество перекомпиляций.

В этом режиме "gnatmake" игнорирует различие меток времени вызванное только модификацией исходных текстов при котором производилось только добавление/удаление комментариев, пустых строк, символов пробела или табуляции.

Это подразумевает, что если в исходном тексте изменения затронули только комментарии или исходный текст был только переформатирован, то использование этой опции указывает утилите "gnatmake", что нет необходимости выполнять перекомпиляцию файлов, зависимых от этого файла.

-M — Проверка согласованности обновлений всех объектов.

В случае согласованности обновлений вывод взаимозависимости объектов производится на устройство стандартного вывода "stdout" в форме, которая позволяет непосредственное использование этой информации в файле "Makefile", для сборки проекта под управлением программы GNU "make".

По умолчанию, префиксом для каждого файла с исходным текстом является имя каталога (относительное или абсолютное) в котором этот файл с исходным текстом находится. Такое префиксное имя будет соответствовать тому, что было указано с помощью различных опций "**-aI**" и "**-I**".

При использовании "gnatmake **-M -q**", на "stdout" будут выводиться только имена файлов с исходными текстами, без относительных путей поиска файлов в каталогах.

Если указана только опция "**-M**", то вывод зависимостей от внутренних системных файлов GNAT не производится. Как правило, это отображает именно ту информацию которая необходима.

Если дополнительно указать опцию "**-a**", то тогда будет отображаться информация о зависимости от внутренних системных файлах GNAT.

Примечательно, что зависимость от объектов которые расположены во внешних Ада-библиотеках (см. опции "**-aL***dir*") никогда не отображается.

-n — Выполнить только проверку согласованности обновлений всех объектов, и не выполнять компиляцию, связывание и/или компоновку приложения.

Если при выполнении такой проверки обнаружена не согласованность обновлений, то будет показано полное имя первого файла который нуждается в перекомпиляции.

Многократное повторное использование этой опции, с последующей компиляцией показанных файлов с исходными текстами, в результате приведет к перекомпиляции всех требуемых модулей.

-o *exec_name* — Определение имени для генерируемого исполняемого файла приложения. При этом имя исполняемого файла будет "*exec_name*".

Если имя для генерируемого исполняемого файла приложения не указано явно, то тогда, по умолчанию, для получения имени исполняемого файла будет использовано имя указанного

входного файла. При этом, имя входного файла будет приведено, согласно правил хост системы, в форму, соответствующую имени исполняемого файла для хост системы.

-q — Устанавливает "молчаливый"режим работы утилиты "gnatmake".

При обычном режиме (без использования этой опции) утилита "gnatmake"отображает все выполняемые ею команды.

-s — Выполнить перекомпиляцию, если опции компилятора были изменены при последней компиляции.

Все опции компиляции, кроме "**-I**" и "**-o**", рассматриваются в следующем порядке: порядок следования между различными опциями (которые заданы первыми буквами) игнорируется, но выполняется анализ порядка следования между одинаковыми опциями.

Например, указание опций "**-O -O2**" будет отличаться от "**-O2 -O**", но указание опций "**-g -O**" эквивалентно "**-O -g**".

-v — Устанавливает "многословный"режим при котором утилита "gnatmake" отображает смысл причины выполнения всех необходимых перекомпиляций.

-z — Указывает на отсутствие главной подпрограммы.

В результате, выполняется связывание и компоновка программы даже в случае когда имя модуля, указанное в команде запуска утилиты "gnatmake", является именем пакета.

Результирующий исполняемый файл будет запускать на выполнение подпрограммы элаборации пакета, а его завершение будет запускать на выполнение подпрограммы очистки.

Опции "gcc" ("gnatgcc") — Опция "**-g**" или любая другая опция заданная символом верхнего регистра (кроме "**-A**" или "**-L**") или любая опция, которая описывается более чем одним символом будет передана "gnatgcc" (например, "**-O**", "**-gnato**" и т.д.).

27.3.4.2 Указание путей поиска файлов для gnatmake

Кроме показанных ранее опций, утилита "gnatmake" поддерживает опции указания путей поиска для файлов с исходными текстами и библиотечных файлов:

-aI*dir* — При поиске файлов с исходными текстами, поиск файлов должен также осуществляться в каталоге "*dir*".

-aL*dir* — Предположим, что в каталоге "*dir*" расположена дополнительная внешняя Ада-библиотека. Эта опция указывает утилите "gnatmake" на то, что нет нужды выполнять перекомпиляцию модулей чьи ".ali" файлы расположены в каталоге "*dir*". Это позволяет не иметь тела (реализацию) для модулей в каталоге "*dir*". Однако, при этом остается необходимость в указании места расположения файлов со спецификациями этих модулей с помощью опций "**-aI***dir*" или "**-I***dir*".

Примечание: эта опция предусмотрена для совместимости с предыдущими версиями утилиты "gnatmake". Наиболее простым способом, который приводит к исключению анализа стандартной библиотеки, является блокировка ALI-файлов этой библиотеки (запрещение записи в ALI-файлы).

-aO*dir* — При поиске библиотечных и объектных файлов, поиск должен также осуществляться в каталоге "*dir*".

-A*dir* — Эквивалентно "**-aL***dir* **-aI***dir*".

-I*dir* — Эквивалентно "**-aO***dir* **-aI***dir*".

-I- — Не выполнять поиск файлов с исходными текстами в каталогах с исходными текстами, которые указаны в командной строке.

Не выполнять поиск объектных и ALI-файлов в каталоге из которого утилита "gnatmake"была запущена.

-L*dir* — Добавить каталог "*dir*" к списку каталогов в которых компоновщик будет осуществлять поиск библиотечных файлов.

Это эквивалентно использованию "**-largs -L***dir*".

-nostdinc — Не осуществлять поиск файлов с исходными текстами в системном каталоге по умолчанию.

-nostdlib — Не осуществлять поиск библиотечных файлов в системном каталоге по умолчанию.

27.3.4.3 Управление режимами gnatmake

Опции управления режимами "gnatmake" позволяют использовать опции командной строки, которые будет непосредственно переданы компилятору, редактору связей или компоновщику. Эффект использования опций управления режимом приводит к тому, что последующий список опций, вплоть до завершения всего списка или до следующей опции управления режимом, будет интерпретироваться как опции, которые необходимо передать определенному компоненту системы GNAT (компилятору, редактору связей или компоновщику).

-cargs *switches* — Опции компилятора. Здесь, "*switches*" представляет список опций которые допустимо использовать с "gcc" ("gnatgcc"). Этот список опций будет передаваться компилятору каждый раз когда утилита "gnatmake"проходит этап компиляции.

-bargs *switches* — Опции редактора связей. Здесь, "*switches*" представляет список опций которые допустимо использовать с "gcc" ("gnatgcc"). Этот список опций будет передаваться редактору связей каждый раз когда утилита "gnatmake"проходит этап связывания.

-largs *switches* — Опции компоновщика. Здесь, "*switches*" представляет список опций которые допустимо использовать с "gcc" ("gnatgcc"). Этот список опций будет передаваться компоновщику каждый раз когда утилита "gnatmake"проходит этап компоновки.

27.3.4.4 Примечания для командной строки gnatmake

Здесь предоставляются некоторые дополнтельные сведения, которые могут быть полезны при работе с командной строкой для утилиты "gnatmake".

- Если утилита "gnatmake" не находит ALI-файлов, то она выполняет перекомпиляцию главного файла программы и всех модулей, которые необходимы для главной программы. Это подразумевает, что утилита "gnatmake" может быть использована как для выполнения начальной компиляции, так и на последующих этапах цикла разработки.

- Если будет введена команда:

  ```
  gnatmake file.adb
  ```

  где "*file*.adb" является субмодулем или телом настраиваемого модуля, то утилита "gnatmake" выполнит перекомпиляцию "*file*.adb" (поскольку она не обнаружит ALI-файла) и остановиться, выдав предупреждающее сообщение.

- При использовании утилиты "gnatmake" с опцией "**-I**" определяются пути поиска как для файлов с исходными текстами, так и для библиотечных файлов. Если необходимо указать пути поиска только для файлов с исходными текстами, то вместо опции "**-I**" необходимо использовать опцию "**-aI**", а если необходимо указать пути поиска только для библиотечных файлов, то необходимо использовать опцию "**-aO**".

- Утилита "gnatmake" выполняет проверку согласованности ALI-файлов и соответствующих объектных файлов. Если обнаруживается, что ALI-файл новее соответствующего объектного файла, или объектный файл отсутствует, то выполняется компиляция соответствующего файла с исходным текстом. Примечательно, что утилита "gnatmake", при выполнении проверки ALI-файлов и соответствующих объектных файлов, ожидает их наличие в одном каталоге.

- Утилита "gnatmake" игнорирует любые файлы для которых соответствующие ALI-файлы заблокированы (запрещены по записи). Это может быть использовано для того, чтобы исключить из проверки на согласованность файлы стандартной библиотеки, и это также подразумевает, что использование опции "**-f**" не приведет к перекомпиляции таких файлов пока одновременно с опцией "**-f**" не будет указана опция "**-a**".

- Утилита "gnatmake" была разработана с учетом согласованного использования самостоятельных Ада-библиотек. Предположим, что имеется Ада-библиотека, которая организована следующим образом: каталог "*obj-dir*" содержит объектные файлы и ALI-файлы компилируемых модулей Ады, в то время как каталог "*include-dir*" содержит файлы спецификаций этих модулей, но не содержит тела (реализацию) этих модулей. Тогда, для компиляции модуля который находится в файле "main.adb" и использует эту Ада-библиотеку необходимо будет использовать следующую команду:

```
gnatmake -aIinclude-dir  -aLobj-dir  main
```

- Использование утилиты "gnatmake" с опцией "**-m (minimal recompilation)**" является очень удобным средством, которое позволяет свободно обновлять комментарии и форматирование исходных текстов без необходимости выполнения полной перекомпиляции. Примечательно однако, что добавление строк к файлам с исходными текстами может повлиять на отладку на уровне исходного текста, ввиду устаревания отладочной информации. Если такая модификация касается файла со спецификацией, то влияние будет менее значительным, поскольку затронутая отладочная информация будет полезна только во время фазы элаборации программы. Для файлов тел, такое влияние будет более значительным. Во всех случаях, отладчик выдаст предупреждение о том, что файл с исходными текстами более новый по сравнению с файлом который содержит объектный код, и, таким образом, устаревшая отладочная информация пройдет не замеченной.

27.3.5 Связывание и компоновка, утилита gnatbl

Бывают случаи, когда необходимо выполнить сборку проекта части которого написаны на других языках программирования. В таких ситуациях, компиляция выполняется с помощью различных компиляторов, а этапы связывания и компоновки могут быть выполнены автоматически за один шаг. Для этого можно использовать утилиту "gnatbl", которая автоматически вызывает редактор связей "gnatbind" и компоновщик "gnatlink", соответственно выполняя связывание и компоновку проекта.

27.4 Сравнение моделей компиляции

27.4.1 Модели компиляции GNAT и C/C++

Модель компиляции GNAT очень похожа на модель компиляции, которая традиционна для языков программирования C/C++. Файлы спецификации могут рассматриваться как соответствующие файлы заголовков ("*.h") C/C++. Также как и в случае с файлами заголовков C/C++, нет необходимости выполнять компиляцию файлов спецификаций. Файлы спецификаций компилируются только в случае их непосредственного использования. Спецификатор "**with**" обладает эффектом, который подобен использованию директивы "**#include**" в C/C++.

При этом существует одно примечательное отличие: в Аде можно выполнять раздельную компиляцию спецификации с целью осуществления ее семантической и синтаксической проверки. Это не всегда возможно для файлов заголовков C/C++ поскольку они являются фрагментами программы, которая использует иные семантические и синтаксические правила.

Еще одним значительным отличием является необходимость запуска редактора связей "gnatbind", на который возложены две важных функции. Во-первых, редактор связей осуществляет проверку согласованности отдельных компонентов программы. В C/C++ такая проверка согласованности различных частей программы, как правило, возлагается на файл управляющий сборкой проекта ("Makefile"). Таким образом, редактор связей "gnatbind"не позволяет скомпоновать результирующую программу из несогласованных компонентов при обычном режиме использования компилятора, что обеспечивает соответствие стандартным требованиям Ады.

Второй, весьма важной задачей, которая возложена на редактор связей, является обработка и анализ очередности элаборации отдельных компонентов программы. Следует заметить, что обработка очередности элаборации присутствует также в C++. Считается, что в C++ она осуществляется автоматически и поэтому о ней, как правило, не упоминается. Однако фактическая очередность элаборации самостоятельных объектных модулей C++ определяется порядком компоновки объектных модулей компоновщиком. Преимуществом подхода C++ является простота, но при этом программист не имеет реального контроля над процессом элаборации. В результате, в случаях когда редактор связей "gnatbind"может указать на отсутствие корректного порядка элаборации и, следовательно, не возможность компоновки результирующей программы, компилятор C++ выполнит построение программы не работоспособность которой будет обнаружена только во время выполнения.

27.4.2 Модель компиляции GNAT и общая согласованная Ада-библиотека

Эта информация может оказаться полезной для тех программистов, которые ранее использовали какой-либо другой Ада-компилятор использующий традиционную модель общей согласованной Ада-библиотеки, как описано в "Руководстве по языку программирования Ада 95".

Система компилятора GNAT не использует традиционную модель общей согласованной Ада-библиотеки. Вместо этого, GNAT использует множество файлов с исходными текстами совокупность которых играет роль общей библиотеки. Компиляция Ада-программы не генерирует никакой централизованной информации. Вместо этого в результате компиляции генерируются объектные файлы и `ALI`-файлы, которые предназначены для обработки редактором связей "`gnatbind`"и компоновщиком "`gnatlink`". В традиционной Ада-системе компилятор читает информацию не только из файла с исходным текстом, который компилируется, но и из центральной библиотеки, что подразумевает зависимость текущей компиляции от того, что было скомпилировано ранее. В частности:

- При указании какого-либо модуля в спецификаторе "**with**" компилятору будет представлена самая свежая версия этого модуля скомпилированного в общую библиотеку.
- Встроенная подстановка (*inline*) будет эффективна только в случае когда необходимое тело уже было скомпилировано в общую библиотеку.
- Компиляция какого-либо нового модуля может сделать устаревшими версии других модулей в общей библиотеке.

В системе компилятора GNAT, компиляция одного модуля никогда не влияет на компиляцию каких-либо других модулей поскольку компилятор всегда читает только файлы с исходным текстом. Только изменение файлов с исходными текстами может повлиять на результаты компиляции. В частности:

- При указании какого-либо модуля в спецификаторе "**with**" компилятору будет представлена версия этого модуля соответствующий исходный текст которой будет доступен компилятору в процессе текущей компиляции.
- Встроенная подстановка (*inline*) требует наличия и доступности для компилятора файлов с соответствующими исходными текстами тел пакетов и/или подпрограмм. Таким образом, встроенная подстановка всегда эффективна, вне зависимости от порядка компиляции различных модулей.
- Компиляция какого-либо модуля никогда не влияет на результаты компиляции других модулей. Редактирование файлов с исходными текстами может сделать устаревшими результаты предыдущих компиляций в случае когда они зависят от модифицированных файлов с исходными текстами.

Наиболее существенным результатом таких различий является то, что для GNAT последовательность выполнения компиляции не имеет значения. Не существует ситуации при которой для компиляции чего-либо необходимо выполнить какую-либо другую, предварительную компиляцию. То что в традиционной Ада-системе с общей библиотекой проявляется как требование строго определенной последовательности компиляции для GNAT проявляется как простая зависимость файлов с исходными текстами. Другими словами, существует только набор правил, который указывает какие файлы с исходными текстами должны быть представлены при компиляции файла.

27.5 Директивы конфигурации

Согласно "Руководства по языку программирования Ада 95"существует набор директив компилятора, которые предназначены для управления конфигурацией среды компиляции Ада-системы. Такие директивы называют директивами конфигурации. Кроме того, некоторые директивы определяемые реализацией Ада-системы также являются директивами конфигурации. Примером директивы конфигурации может служить определяемая в реализации GNAT директива "Source_File_Name", которая позволяет указывать имена файлов с исходными текстами когда эти имена не соответствуют принятым по умолчанию соглашениям именования для файлов с исходными текстами. Ниже представлен полный перечень директив конфигурации распознаваемых компилятором GNAT:

```
Ada_83                          Normalize_Scalars
Ada_95                          Polling
C_Pass_By_Copy                  Propagate_Exceptions
Component_Alignment             Queuing_Policy
Discard_Names                   Ravenscar
Elaboration_Checks              Restricted_Run_Time
Eliminate                       Restrictions
Extend_System                   Reviewable
Extensions_Allowed              Source_File_Name
External_Name_Casing            Style_Checks
Float_Representation            Suppress
Initialize_Scalars              Task_Dispatching_Policy
License                         Unsuppress
Locking_Policy                  Use_VADS_Size
Long_Float                      Warnings
No_Run_Time                     Validity_Checks
```

27.5.1 Обработка директив конфигурации

Директивы конфигурации могут присутствовать в начале исходного текста компилируемого модуля, оказывая, таким образом, воздействие только на соответствующий компилируемый модуль. Кроме того, директивы конфигурации могут использоваться для воздействия на всю среду компиляции в целом.

Система компилятора GNAT предусматривает утилиту "`gnatchop`", которая обеспечивает способ автоматической обработки директив конфигурации в соответствии с семантикой компиляции (а именно, для файлов с множеством модулей) описанной в "Руководстве по языку программирования Ада 95". Однако, в большинстве случаев, для общей настройки среды компиляции, предпочтительнее модифицировать содержимое файла "`gnat.adc`", который предназначен для непосредственного хранения директив конфигурации.

27.5.2 Файлы директив конфигурации

В системе компилятора GNAT, среда компиляции определяется текущим каталогом на момент запуска команды компиляции. Этот текущий каталог используется для поиска файла с именем "`gnat.adc`". При обнаружении этого файла, предполагается, что он содержит одну или более директив конфигурации, которые должны быть использованы в текущей компиляции. Однако, если в командной строке команды компиляции присутствует опция "`-gnatA`", то содержимое файла "`gnat.adc`" не рассматривается..

Директивы конфигурации могут быть введены в файл "`gnat.adc`", как путем запуска утилиты "`gnatchop`" для обработки файла с исходным текстом, который состоит только из директив конфигурации, так и с помощью непосредственного редактирования "`gnat.adc`", который является обычным текстовым файлом (последний способ считается более предпочтительным).

В дополнение к "`gnat.adc`" для воздействия на среду текущей компиляции может быть дополнительно использован еще один файл содержащий директивы конфигурации. Для указания этого, в командной строке необходимо использовать опцию "`-gnatec`*`path`*". В этом случае, параметр опции "*`path`*" должен указывать существующий файл, который состоит только из директив конфигурации. Директивы конфигурации указанного файла будут служить как дополнение к директивам конфигурации, которые расположены в файле "`gnat.adc`" (предполагается, что файл "`gnat.adc`" присутствует, и не используется опция "`-gnatA`").

Следует заметить, что в командной строке может присутствовать несколько опций "`-gnatec`". Однако, фактическое значение будет иметь только последняя указанная опция.

Глава 28

Вспомогательные утилиты

28.1 Уменьшение затрат времени с помощью утилиты `gnatstub`

Начиная с версии 3.11p, система компилятора GNAT предусматривает утилиту "`gnatstub`". Утилита "`gnatstub`" может быть использована для построения файла прототипа тела пакета в соответствии с предоставленным файлом спецификации пакета. В результате обработки файла спецификации пакета, в котором указаны спецификации подпрограмм, утилита "`gnatstub`" создает файл тела пакета, в котором располагаются "пустые"тела подпрограмм, соответствующие их спецификациям. Такие "пустые" подпрограммы часто называют "заглушками" (*stubs*).

Использование утилиты "`gnatstub`" может оказаться полезным при разработке больших программных проектов, когда программист разрабатывает целый набор файлов спецификаций пакетов, а затем проверяет правильность общего дизайна проекта. В таких случаях, как только закончена разработка файлов спецификаций, утилита "`gnatstub`" может быть использована для создания базовых шаблонов тел, сохраняя тем самым программисту время, затрачиваемое на копирование и модификацию спецификаций вручную.

Предположим, что у нас есть файл с исходным текстом спецификации пакета "`tiny.ads`", содержимое которого имеет следующий вид:

```
package Tiny is

   procedure Simple_Procedure;
   function Simple_Function return Boolean;

end Tiny;
```

Шаблон тела пакета может быть создан с помощью команды:

```
gnatstub tiny.ads
```

В результате, утилита "`gnatstub`" генерирует файл "`tiny.adb`", который имеет следующий вид:

```
package body Tiny is

   ----------------------
   -- Simple_Function --
   ----------------------

   function Simple_Function return Boolean is
   begin
      return Simple_Function;
   end Simple_Function;

   -----------------------
   -- Simple_Procedure --
   -----------------------

   procedure Simple_Procedure is
   begin
```

```
      null;
   end Simple_Procedure;

   end Tiny;
```

Такое тело пакета имеет соответствующий формат и готово к компиляции. Естественно, что такое тело пакета, реально, ничего полезного не выполняет. Ответственность за описание соответствующих деталей реализации полностью возлагается на программиста.

28.2 Утилита перекрестных ссылок `gnatxref`

Утилита "`gnatxref`" (или "`gnatf`" для GNAT 3.10) — это утилита которая генерирует индексы для каждого появления идентификатора в программе, включая все идентификаторы использованные в пакетах от которых данная программа зависит.

Опция "**`-v`**" позволяет генерировать листинг в формате "`tag`"-файла редактора "`vi`".

Для программы "`hello.adb`", показанной ранее, утилита "`gnatxref`" генерирует следующее:

```
Text_IO U a-textio.ads:51:13  hello.adb:1:10 4:7
Put_Line U a-textio.ads:260:14  hello.adb:4:15
Ada U ada.ads:18:9  hello.adb:1:6 4:3
hello U hello.adb:2:11
```

Каждая строка начинается с указания имени индексируемого идентификатора. Далее следует имя файла в котором данный идентификатор объявлен с указанием расположения этого объявления в файле. В заключение, следует секция в которой перечислены все появления идентификатора в программе.

В данном примере, идентификатор "Text_IO" появляется в первой строке (с учетом спецификатора "**with**") и в четвертой строке ("Put_Line").

28.3 Оценка "мертвого" кода с помощью утилиты `gnatelim`

Утилита "`gnatelim`", основанная на ASIS (***A**da **S**emantic **I**nterface **S**pecification*), может быть использована для поиска неиспользуемых программой частей в объектных файлах и удаления их из финального исполняемого файла. В результате ее работы, создается файл перечисляющий подпрограммы которые компилятор не должен компилировать. При сохранении этого списка не используемых подпрограмм в файле "`gnat.adc`", утилита "`gnatmake`" будет автоматически читать этот файл и отбрасывать указанные подпрограммы при компиляции.

Согласно рекомендаций руководства пользователя по компилятору GNAT, для использования утилиты "`gnatelim`", необходимо сгенерировать дерево файлов, используя опцию "**`-gnatt`**". Предположим, что главной программой является файл "`main.adb`", тогда мы можем выполнить следующее:

```
gnatmake -c main
gnatbind main
gnatmake -f -c -gnatc -gnatt main
gnatelim main > gnat.adc
gnatmake -f main
```

Эти команды сгенерируют полный набор дерева файлов для проекта, отбросят все не используемые в проекте подпрограммы, и, затем, перекомпилируют весь проект как финальный исполняемый файл.

28.4 Отслеживание состояния стека и обнаружение утечек памяти во время выполнения программы

Начиная с версии 3.12 GNAT предусматривает средства обратной трассировки, которые позволяют получить информацию об источнике исключения и состоянии стека времени выполнения в случае возникновения

исключения. Эти средства предоставляются пакетами *Gnat.Traceback* и *Gnat.Traceback.Symbolic* (за более подробной информацией следует обратиться к спецификациям этих пакетов), которые позволяют точно идентифицировать место возникновения исключения, вплоть до определения файла с исходным текстом и строки в результате выполнения которой было возбуждено исключение. Для того, чтобы активировать использование этих средств необходимо при компиляции программы указать опцию "**-funwind-tables**", а при связывании программы — опцию "**-E**".

28.4.1 Утилита gnatmem

Для мониторинга программы, которая выполняется под управлением отладчика "gdb" (поставляемого вместе с GNAT), может быть использована утилита "gnatmem" (начиная с версии 3.12 GNAT). После того как выполнение программы завершено, утилита "gnatmem" отображает общую информацию о динамическом распределении памяти, в процессе работы программы. Эта информация может быть использована для поиска "утечек" памяти, то есть, мест где программа осуществляет динамическое распределение памяти и не возвращает распределенное пространство памяти системе. Поскольку утилита "gnatmem" использует отладчик "gdb", программа должна быть скомпилирована с подключением поддержки отладки под управлением "gdb", что выполняется указанием опции "**-g**" при компиляции программы.

Для запуска программы "program" под управлением утилиты "gnatmem"можно сделать следующее:

```
gnatmem program
```

Утилита "gnatmem" может принимать следующие опции командной строки:

- **-q** — Активирует "молчаливый" режим — выводится не вся статистика, а только информация о потенциально возможных утечках памяти.
- ***n*** — Число в интервале от 1 до 10 указывающее глубину вложенности информации обратной трассировки.
- **-o** *file* — Сохранить вывод отладчика "gdb" в указанный файл "*file*". Скрипт отладчика "gdb" сохраняется как файл "gnatmem.tmp".
- **-i** *file* — Осуществить мониторинг используя первоначально сохраненный с помощью опции "**-o**" файл "*file*". Удобно использовать для проверки программы выполнение которой под управлением утилиты "gnatmem" было неожиданно прервано в результате возникновения каких-либо ошибок.

28.4.2 Средства пакета *GNAT.Debug_Pools*

Использование "Unchecked_Deallocation" и/или "Unchecked_Conversion" может легко привести к некорректным ссылкам памяти. От проблем, порождаемых подобными ссылками, достаточно сложно избавиться поскольку проявляемые симптомы, как правило, далеки от реального источника проблемы. В подобных случаях, очень важно обнаружить наличие подобной проблемы как можно раньше. Именно такая задача возлагается на "Storage_Pool", который предусмотрен в *GNAT.Debug_Pools*.

Чтобы воспользоваться средствами отладочного пула динамической памяти GNAT, пользователь должен ассоциировать объект отладочного пула с каждым ссылочным типом, который потенциально способен быть источником проблем.

```
type Ptr is access Some_Type;
Pool : GNAT.Debug_Pools.Debug_Pool;
for Ptr'Storage_Pool use Pool;
```

Тип "GNAT.Debug_Pool" является производным от "Checked_Pool", который является характерным для GNAT пулом динамической памяти. Такие пулы динамической памяти, подобно стандартным пулам динамической памяти Ады, позволяют пользователю переопределять стратегию распределения (аллокацию) и освобождения (деаллокацию) динамической памяти. Они также предусматривают контрольные точки, для каждого разыменования (расшифровки) ссылок посредством операции "Dereference", которая неявно вызывается в процессе разыменования каждого значения ссылочного типа.

После того как ссылочный тип ассоциирован с отладочным пулом, операции над значениями этого типа способны возбудить четыре различных исключения, которые соответствуют четырем потенциальным видам разрушения памяти:

- GNAT.Debug_Pools.Accessing_Not_Allocated_Storage
- GNAT.Debug_Pools.Accessing_Deallocated_Storage
- GNAT.Debug_Pools.Freeing_Not_Allocated_Storage
- GNAT.Debug_Pools.Freeing_Deallocated_Storage

Для типов, ассоциированных с отладочным пулом "Debug_Pool" динамическое распределение выполняется с помощью использования стандартных подпрограмм размещения динамической памяти GNAT. Ссылки ко всем фрагментам распределенной памяти сохраняются во внутреннем каталоге. Стратегия освобождения состоит не в освобождении памяти используемой системы, а в заполнении памяти шаблоном, который может быть легко распознан в процессе отладочной сессии. Шаблон соответствует старому двоично десятичному соглашению IBM и имеет значение "16#DEADBEEF#". При каждом разыменовании, осуществляется проверка того, что ссылочное значение указывает на правильно распределенное место памяти. Ниже показан полный пример использования отладочного пула "Debug_Pool", который содержит типичный образец разрушения памяти:

```
with Gnat.Io; use Gnat.Io;
with Unchecked_Deallocation;
with Unchecked_Conversion;
with GNAT.Debug_Pools;
with System.Storage_Elements;
with Ada.Exceptions; use Ada.Exceptions;
procedure Debug_Pool_Test is

   type T is access Integer;
   type U is access all T;

   P : GNAT.Debug_Pools.Debug_Pool;
   for T'Storage_Pool use P;

   procedure Free is new Unchecked_Deallocation (Integer, T);
   function UC is new Unchecked_Conversion (U, T);
   A, B : aliased T;

   procedure Info is new GNAT.Debug_Pools.Print_Info (Put_Line);

begin
   Info (P);
   A := new Integer;
   B := new Integer;
   B := A;
   Info (P);
   Free (A);
   begin
      Put_Line (Integer'Image(B.all));
   exception
      when E : others => Put_Line ("raised: " & Exception_Name (E));
   end;
   begin
      Free (B);
   exception
      when E : others => Put_Line ("raised: " & Exception_Name (E));
   end;
   B := UC (A'Access);
   begin
      Put_Line (Integer'Image(B.all));
   exception
      when E : others => Put_Line ("raised: " & Exception_Name (E));
   end;
   begin
      Free (B);
   exception
      when E : others => Put_Line ("raised: " & Exception_Name (E));
```

```
   end;
   Info (P);
end Debug_Pool_Test;
```

Механизм отладочного пула предусматривает отображение следующей диагностики, при выполнении показанной выше программы с ошибками:

```
Debug Pool info:
  Total allocated bytes :  0
  Total deallocated bytes :  0
  Current Water Mark:  0
  High Water Mark:  0

Debug Pool info:
  Total allocated bytes :  8
  Total deallocated bytes :  0
  Current Water Mark:  8
  High Water Mark:  8

raised: GNAT.DEBUG_POOLS.ACCESSING_DEALLOCATED_STORAGE
raised: GNAT.DEBUG_POOLS.FREEING_DEALLOCATED_STORAGE
raised: GNAT.DEBUG_POOLS.ACCESSING_NOT_ALLOCATED_STORAGE
raised: GNAT.DEBUG_POOLS.FREEING_NOT_ALLOCATED_STORAGE
Debug Pool info:
  Total allocated bytes :  8
  Total deallocated bytes :  4
  Current Water Mark:  4
  High Water Mark:  8
```

28.5 Условная компиляция с помощью препроцессора `gnatprep`

Хотя разработчики Ады решили не включать в язык директивы поддержки препроцессора, подобные используемым в языке программирования C, GNAT предусматривает препроцессор "`gnatprep`", который позволяет осуществлять условную компиляцию Ада-программ.

Препроцессор "`gnatprep`" (*GNAT PREProcessor*) принимает файл содержащий директивы условной компиляции и генерирует, в результате своей работы, файл с исходным текстом в котором удалены все инструкции, которые не соответствуют условиям указанным в директивах условной компиляции.

Следует обратить внимание на то, что директивы условной компиляции не допускают использования выражений (в отличие от препроцессора языка C). Таким образом, в директивах условной компиляции могут использоваться только переменные, которые могут принимать значения "истина" (*True*) или "ложь" (*False*).

Предположим, что у нас есть файл "`prepvalues`" со следующими определениями для препроцессора:

```
ALPHAVERSION    := True
BETAVERSION     := False
RELEASEVERSION  := False
TRANSLATION     := English
```

Предположим также, что у нас есть небольшая программа, текст которой содержит директивы препроцессора "`gnatprep`":

```
with Text_IO;
use  Text_IO;

procedure Preptest is

   -- включить только ту часть кода, которая уместна для этой версии

   #if ALPHAVERSION
      S_Version : String := "Alpha";
   #elsif BETAVERSION
      S_Version : String := "Beta";
```

```
    #elsif RELEASEVERSION
        S_Version : String := "Release";
    #else
        S_Version : String := "Unknown";
    #end if;


    -- строковой переменной S_Translation будет присваиваться
    -- значение переменной препроцессора $TRANSLATION

    S_Translation : String := "$TRANSLATION";

begin

    Put_Line( "This is the " & S_Version & " edition" );
    Put_Line( "This is the " & S_Translation & " translation" );

end Preptest;
```

В результате обработки показанной выше программы препроцессором "**gnatprep**", с учетом значений указанных в файле "**prepvalues**", будет получен следующий исходный текст программы:

```
with Text_IO;
use  Text_IO;

procedure Preptest is

    -- включить только ту часть кода, которая уместна для этой версии

        S_Version : String := "Beta";

    -- строковой переменной S_Translation будет присваиваться
    -- значение переменной препроцессора $TRANSLATION

    S_Translation : String := "English";


begin

    Put_Line( "This is the " & S_Version & " edition" );
    Put_Line( "This is the " & S_Translation & " translation" );

end Preptest;
```

Препроцессор "**gnatprep**" может принимать следующие опции командной строки:

- **-D***symbol*=*value* — Позволяет объявить значение "*value*" для символа "*symbol*" в командной строке запуска препроцессора "**gnatprep**", а не в файле (аналогично опции "**-D**" для препроцессора языка C). Например, "**-DMacintosh=FALSE**".
- **-b** — Заменить в выходном файле с исходным текстом команды препроцессора "**gnatprep**" пустыми строками (вместо "**-c**").
- **-c** — Закомментировать в выходном файле с исходным текстом команды препроцессора "**gnatprep**" вместо "**-b**").
- **-r** — Сгенерировать в выходном файле с исходным текстом директиву компилятора "Source_Reference".
- **-s** — Распечатать отсортированный список символов и их значений.
- **-u** — Трактовать необъявленные символы как символы имеющие значение "**FALSE**".

Следует заметить, что препроцессор "**gnatprep**" не имеет конструкций, эквивалентных "**__FILE__ (name of current s**
или "**__LINE__ (number of current line)**", которые поддерживаются препроцессором языка C.

28.6 Утилиты gnatpsys и "gnatpsta

Множество описаний стандартных пакетов *System* и *Standard* зависят от конкретной реализации системы и используемой платформы. Исходный текст пакета *Standard* не доступен, а доступный исходный текст пакета *System* использует множество специальных атрибутов для параметризации различных критических значений и в некоторых случаях может оказаться не достаточно информативным.

Для подобных случаев предусматриваются специальные утилиты "gnatpsys" и "gnatpsta" (начиная с версии 3.15, утилита "gnatpsys" — не поставляется). Эти утилиты осуществляют динамическое определение значений параметров, которые зависят от конкретной реализации системы и используются пакетами *System* и *Standard*, после чего, отображают их в виде листингов с исходными текстами этих пакетов, показывая все фактические значения, которые могут быть интересны программисту.

Обе утилиты осуществляют вывод на стандартное устройство вывода используемой системы и не требуют указания каких-либо параметров в командной строке. Утилита "gnatpsys" предназначена для отображения листинга пакета *System*, а утилита "gnatpsta" отображает исходный текст пакета *Standard*.

28.7 Произвольное именование файлов, утилита gnatname

Как уже указывалось, компилятор должен обладать возможностью определить имя файла с исходным текстом компилируемого модуля. При использовании принимаемых по умолчанию, стандартных для GNAT соглашений именования (".ads" — для спецификаций, ".adb" — для тел), компилятор GNAT не нуждается в какой-либо дополнительной информации.

Когда именование файлов с исходными текстами не соответствует принимаемым по умолчанию, стандартным для GNAT соглашениям именования, компилятор GNAT должен получать дополнительно необходимую информацию с помощью рассмотренных ранее файлов с директивами конфигурации или с помощью файлов проектов, которые будут рассмотрены позже.

Ранее, при рассмотрении альтернативных схем именования, указывалось, что в случае хорошо организованной нестандартной схемы именования, для указания правил именования файлов с исходными текстами необходимо незначительное количество директив "Source_File_Name". Однако, когда используемая схема именования не регулярна или произвольна, для указания имен файлов с исходными текстами может потребоваться значительное число директив "Source_File_Name". Для облегчения поддержки взаимосвязи между именами компилируемых модулей и именами файлов с исходными текстами, система компилятора GNAT (версия 3.15 и более новые версии) предусматривает утилиту "gnatname", которая предназначена для генерации необходимых директив для набора файлов.

Обычно запуск утилиты "gnatname" осуществляется с помощью команды, которая имеет следующий вид:

```
gnatname [опции] [шаблоны_имен]
```

Следует заметить, что все аргументы командной строки запуска "gnatname"не являются обязательными.

При использовании без аргументов, "gnatname" создаст в текущем каталоге файл "gnat.adc", который будет содержать директивы конфигурации для всех компилируемых модулей расположенных в текущем каталоге. Для обнаружения всех компилируемых модулей, "gnatname", для всех обычных файлов расположенных в текущем каталоге, использует запуск компилятора GNAT в режиме проверки синтаксиса. В результате, для файлов, которые содержат компилируемые модули Ады, будет осуществляться генерация директивы "Source_File_Name".

В качестве аргументов "gnatname" могут быть указаны один или более шаблонов имен. Каждый шаблон имен должен быть заключен в двойные кавычки. Шаблон имен является регулярным выражением, которое используется для указания шаблона имен командными интерпретаторами UNIX ("shell") или DOS. Ниже показаны примеры указания шаблонов имен:

```
"*.[12].ada"
"*.ad[sb]*"
"body_*"    "spec_*"
```

Более полное описание синтаксиса, который используется для указания шаблонов имен, приводится при описании второго вида регулярных выражений, описанных в файле исходного текста GNAT "`g-regexp.ads`" (регулярные выражения "Glob").

Отсутствие в командной строке "`gnatname`" аргументов, которые указывают шаблоны имен, является эквивалентным указанию единственного шаблона имен "**`"*"`**".

Все опции, которые указываются в командной строке запуска "`gnatname`", должны предшествовать любым аргументам, которые указывают шаблоны имен.

В командной строке запуска "`gnatname`" могут быть указаны следующие опции:

`-c`*`file`* — Создать файл директив конфигурации "***`file`***" (вместо создаваемого по умолчанию файла "`gnat.adc`"). Между "**`-c`**" и "***`file`***" допускается как отсутствие, так и присутствие пробелов (один и более). Указание файла "***`file`***" может содержать информацию о каталоге. Пользователь должен обладать правами записи в указываемый файл "***`file`***". В командной строке может присутствовать только одна опция "**`-c`**". При указании опции "**`-c`**" не допускается указание опции "**`-P`**".

`-d`*`dir`* — Осуществлять поиск файлов с исходными текстами в каталоге "***`dir`***". Между "**`-d`**" и "***`dir`***" допускается как отсутствие, так и присутствие пробелов (один и более). При указании опции "**`-d`**", поиск исходных файлов в текущем рабочем каталоге не осуществляется за исключением случаев когда текущий каталог явно указан в опции "**`-d`**" или "**`-D`**". В командной строке допускается указание нескольких опций "**`-d`**".

Когда каталог "***`dir`***" указывается как относительный путь, то его расположение выбирается относительно каталога в котором располагается файл с директивами конфигурации указываемый с помощью опции "**`-c`**", или относительно каталога содержащего файл проекта указываемый с помощью опции "**`-P`**", или, при отсутствии опций "**`-c`**" и "**`-P`**", относительно текущего рабочего каталога.

Каталог, указываемый опцией "**`-d`**", должен существовать и должен быть доступен для чтения.

`-D`*`file`* — Осуществлять поиск файлов с исходными текстами в каталогах, которые перечислены в текстовом файле "***`file`***". Между "**`-D`**" и "***`file`***" допускается как отсутствие, так и присутствие пробелов (один и более). Текстовый файл "***`file`***" должен существовать и должен быть доступен для чтения. В файле "***`file`***", каждая непустая строка должна указывать каталог. Указание опции "**`-D`**" является эквивалентом указания в командной строке такого же количества опций "**`-d`**", сколько непустых строк содержится в файле "***`file`***".

`-h` — Выводит информацию подсказки (*help*) об использовании. Вывод направляется на стандартное устройство вывода "`stdout`".

`-P`*`proj`* — Создать или обновить файл проекта "***`proj`***". Между "**`-P`**" и "***`proj`***" допускается как отсутствие, так и присутствие пробелов (один и более). Указание файла проекта "***`proj`***" может содержать информацию о каталоге. Файл "***`proj`***" должен быть доступен по записи. В командной строке может присутствовать только одна опция "**`-P`**". При указании опции "**`-P`**" не допускается указание опции "**`-c`**".

`-v` — "Многословный" (*verbose*) режим. Вывод детального объяснения поведения на устройство стандартного вывода "`stdout`". Это включает: имя записонного файла; имена каталогов поиска и имена каждого файла, в этих каталогах поиска, которые соответствуют хотябы одному из указанных шаблонов имен; индикацию — является ли файл компилируемым модулем, и, если да, то вывод имени модуля.

`-v -v` — Очень "многословный" (*verbose*) режим. В дополнение к выводу информации генерируемой в многословном режиме, для каждого файла, который обнаружен в указанных каталогах поиска, имя которого не совпадает ни с одним из заданных шаблонов имен, выводится индикация о несовпадении имени файла.

В заключение рассмотрим несколько простых примеров команд запуска "`gnatname`".

Команда запуска "`gnatname`" без параметров

```
gnatname
```

будет эквивалентна команде

```
gnatname -d. "*"
```

В следующем примере

```
gnatname -c /home/me/names.adc -d sources "[a-z]*.ada*"
```

каталог "`/home/me`" должен существовать и быть доступным для записи. Кроме того, каталог "`/home/me/sources`" (указанный как "`-d sources`") должен существовать и быть доступным для чтения. Следует заметить наличие необязательных пробелов после опций "**`-c`**" и "**`-d`**".

Еще один пример:

```
gnatname -P/home/me/proj -dsources -dsources/plus -Dcommon_dirs.txt "body_*" "spec_*"
```

Здесь следует обратить внимание на возможность одновременного использования нескольких опций "**`-d`**" совместно с одной (или более) опцией "**`-D`**". Кроме того, в этом примере используются несколько шаблонов имен.

Глава 29

Оптимизация проекта

Оптимизация является настройкой программы, которая позволяет уменьшить размер результирующего исполняемого файла или увеличить скорость выполнения программы на компьютере определенного типа.

Если программа выполняется не так быстро как это ожидалось или если программа использует пространство памяти или диска размер которого превышает ожидаемый, то для улучшения характеристик результирующей программы, в первую очередь, необходимо пересмотреть дизайн исходных текстов программы. При этом, в первую очередь, необходимо убедиться в правильности выбора используемых структур данных и алгоритмов. Например, алгоритм пузырьковой сортировки является легким способом сортировки данных относительно маленького объема, однако алгоритм квиксорт (*quick sort*) обладает производительностью, которая во много раз выше.

В большой программе, не так легко определить подпрограмму скорость выполнения которой оказывает критическое влияние на общую производительность системы. Как правило, подобные подпрограммы называют узкими местами. Эксперименты с различными наборами входных тестовых данных и анализ времени выполнения различных участков программы позволяет отыскать такие узкие места. Для облегчения процесса поиска, можно прибегнуть к помощи специальных средств профилирования работы программы (например, использовать программу GNU "`gprof`"), которые позволяют автоматизировать процесс сбора статистики выполнения различных участков программы.

Некоторая оптимизация может быть выполнена компилятором GNAT автоматически. Для оптимизации программы можно использовать как опции командной строки, так и директивы компилятора.

29.1 Опции оптимизации компилятора

Существует несколько опций командной строки компилятора которые могут быть использованы для общей оптимизации программы:

-O0 — Отсутствие оптимизации. Выполняет самую быструю компиляцию, а при наличии директив управляющих оптимизацией в исходном тексте программы GNAT будет выдавать предупреждающие сообщения. Рекомендуется для случаев когда важна быстрота компиляции.

-O или **-O1** — Обычная оптимизация, которая устанавливается по умолчанию. Это ведет к более медленной компиляции и отсутствию предупреждающих сообщений о наличии директив управляющих оптимизацией в исходном тексте программы. Как правило, вам необходимо использовать этот режим оптимизации.

-O2 — Экстенсивная оптимизация, позволяющая получить исполнимый файл меньшего размера.

-O3 — Полная оптимизация, с выполнением автоматической встроенной подстановки (*inline*) для подпрограмм и циклов маленького размера. Позволяет получить наиболее быстро исполняемый код.

При использовании вещественных чисел с плавающей точкой, можно сознательно округлять ошибки если не используется опция командной строки компилятора "**`-ffloat-store`**". При этом следует учитывать, что согласно замечаний в "GCC FAQ"округление вещественных чисел с плавающей точкой может вызвать

проблемы при использовании опций командной строки компилятора "**-O2**" и "**-O3**" без одновременного использования опции "**-ffloat-store**" (сохранение вещественных чисел с плавающей точкой вне регистров процессора, что замедляет выполнение программы).

На производительность результирующей программы оказывают влияние опции командной строки компилятора которые управляют встроенной подстановкой (*inline*):

-gnatn — позволяет осуществлять встроенную подстановку между пакетами, когда директива компилятора "Inline" используется в спецификациях пакетов.

-gnatN — позволяет осуществлять автоматическую встроенную подстановку между пакетами (ведет к большему расходу памяти)

отсутствие: **-gnatn** / **-gnatN** — встроенная подстановка между пакетами не осуществляется даже в случаях, когда директива компилятора "Inline" используется в спецификациях пакетов.

Следует заметить, что использование этих опций требует одновременного использования опции "**-O**", иначе эти опции не приведут к ожидаемому результату.

Кроме того, использование опции "**-gnatp**", которая отключает некоторые несущественные проверки (проверка ограничений и проверка диапазона), может также несколько повысить общую производительность программы. Следует заметить, что действие этой опции аналогично использованию в исходном тексте директивы компилятора:

```
pragma Suppress (All_Checks);
```

Для общей оптимизации программы могут быть также использованы следующие опции "`gcc`" ("`gnatgcc`"):

-ffast-math — GCC будет игнорировать некоторые требования математических правил ANSI и IEEE. Например, в результате применения этой опции, перед вызовом функции "sqrt" не будет выполняться проверка того, что число имеет не отрицательное значение.

Следует однако учитывать, что хотя применения этой опции может повысить производительность выполнения математических действий, в результате могут быть обнаружены побочные эффекты при использовании библиотек ожидающих соответствие требованиям математических правил ANSI/IEEE.

-fomit-frame-pointer — GCC будет освобождать регистры, которые обычно предназначены для сохранения указателя на кадр стека.

Это повышает производительность, но осложняет процесс отладки, поскольку многие утилиты отладки требуют наличие указателя на кадр стека.

29.2 Средства оптимизации GNAT, используемые в исходном тексте

Существует несколько директив компилятора, которые позволяют изменять размер и скорость выполнения программы:

Директива	**Описание**
pragma Pack (Aggregate);	Использовать минимальный размер пространства для агрегата.
pragma Optimize (Space / Time / Off);	Выбор типа оптимизации инструкций.
pragma Inline (Subprogram); **pragma** Inline_Always (Subprogram);	Указывают на необходимость выполнения встроенной подстановки (*inline*) подпрограммы "Subprogram".
pragma Discard_Names (**type**);	Не помещать ASCII-идентификаторы в результирующий исполняемый файл.

Директива "**Pack**" позволяет упаковывать массивы, записи и тэговые записи, что позволяет им, в результате, занимать меньшее пространство. Например, упакованный массив переменных логического типа "**Boolean**" приводит к тому, что каждая переменная занимает всего один бит. Директива "**Pack**" позволяет упаковывать только структуры данных. Следует также учесть, что не каждый самостоятельный элемент структуры данных может быть упакован. Например, если имеется массив записей, то для того чтобы при распределении пространства использовался минимально возможный размер, понадобиться выполнить как упаковку

массива, так и упаковку записи. Также следует учитывать, что упаковка структур данных, как правило, ухудшают скорость выполнения программы. Примером использования этой директивы компилятора для упаковки записи может служить следующее:

```
type CustomerProfile is
    record
        Preferred          : Boolean;
        Preorders_Allowed  : Boolean;
        Sales_To_Date      : Float;
    end record;
pragma Pack (CustomerProfile);
```

Следует заметить, что GNAT способен выполнять достаточно плотную упаковку, упаковывая отдельные компоненты структур вплоть до индивидуальных битов.

Директива компилятора "Optimize" позволяет указать компилятору требуемый тип оптимизации инструкций: для максимально возможной скорости выполнения инструкций ("Time"), для использования инструкциями минимально возможного размера ("Space") или без выполнения какой-либо оптимизации вообще ("Off"). Эта директива никак не воздействует на структуры данных.

```
pragma Optimize (Space);
package body AccountsPayable is
```

Директива компилятора "Inline" указывает на то, что необходимо осуществлять встроенную вставку (*inline*) кода подпрограммы в случаях когда это возможно. Это значит, что в местах вызова указанной подпрограммы вместо генерации вызова подпрограммы выполняется непосредственная вставка машинного кода подпрограммы, что позволяет несколько повысить скорость выполнения подпрограммы. При этом следует учитывать, что использование этой директивы может привести к увеличению общего размера результирующего исполняемого файла, поэтому ее следует использовать для тех подпрограмм, которые имеют маленький размер.

```
procedure Increment (X : in out Integer) is
begin
    X := X + 1;
end Increment;
pragma Inline (Increment);
```

Следует учесть, что директива встроенной вставки "'Inline" будет проигнорирована, если при компиляции программы не используется опция командной строки компилятора "**-O3**". Также следует учесть, что опция "**-O3**" будет автоматически осуществлять встроенную вставку коротких подпрограмм.

Директива компилятора "Inline_Always" вызывает принудительную встроенную вставку подпрограмм, описанных в разных пакетах (подобно опции "**-gnatn**") не зависимо от указания в командной строке компилятора опций "**-gnatn**" или "**-gnatN**".

Директива компилятора "Discard_Names" позволяет освободить пространство занимаемое ASCII-строками имен идентификаторов. Например, при наличии большого перечислимого типа, Ада обычно сохраняет строки имен для каждого идентификатора значения перечислимого типа. Это выполняется для поддержки использования атрибута "'Img". Если использование атрибута "'Img" не планируется, то можно указать компилятору на необходимость очистки этих имен.

```
type Dog_Breed is (Unknown, Boxer, Shepherd, Mixed_Breed);
pragma Discard_Names (Dog_Breed);
```

Примечательно, что при выполнении очистки имен, атрибут "'Img" остается доступным. В этом случае, вместо возвращения строкового представления имени идентификатора, атрибут "'Img" будет возвращать позицию значения в списке перечисления значений перечислимого типа (например, 0, 1, 2 и так далее).

29.3 Оптимизация для специфического типа процессора

Для GCC версий 2.x существует две основные опции оптимизации, которые основаны на специфическом типе процессора. Это указано в руководстве по GCC:

-mno-486 — оптимизация для 80386.

-m486 — оптимизация для 80486. Однако, такая программа сможет выполняться на процессоре 80386.

Следует заметить, что в настоящий момент для GCC версий 2.x, нет опций поддержки новых типов процессоров фирмы Intel (Pentium и более новые). Однако предполагается, что будущие версии GNAT, которые будут собраны с использованием GCC версий 3.x и более новыми версиями GCC, возможно, будут полноценно поддерживать следующие опции:

-mpentium — оптимизация для Pentium / Intel 586.

-mcpu=i686 — оптимизация для Pentium II / Intel 686.

-mcpu=k6 — оптимизация для AMD K6.

Для GCC 2.8.1, который используется совместно с GNAT, рекомендуется следующая комбинация опций компилятора для получения разумного выигрыша в производительности при использовании процессора Pentium:

```
-m486 -malign-loops=2 -malign-jumps=2 -malign-functions=2 -fno-strength-reduce
```

Существуют также другие опции, которые могут быть полезны или бесполезны, в зависимости от конкретной программы. Для получения более полной информации следует обратиться к "GCC FAQ".

Рассмотрим совместное использование этих опций. Предположим, что необходимо разработать программу которая будет выполняться на процессоре Intel Pentium, и скорость выполнения программы имеет существенное значение. В процессе разработки программы, можно использовать утилиту "`gnatmake`" с опцией "**`-O1`**". Такая установка будет подавлять предупреждающие сообщения об использовании директив оптимизации в исходном тексте. После завершения разработки, для выполнения заключительной сборки проекта, можно использовать следующую комбинацию опций для утилиты "`gnatmake`":

```
-m486 -O3 -malign-loops=2 -malign-jumps=2 -malign-functions=2 -fno-strength-reduce -gnatp
```

В результате, это позволяет обеспечить максимальную производительность при выполнении программы на процессоре Intel Pentium.

Глава 30

GNAT и библиотеки

Эта глава предоставляет информацию, которая может оказаться полезной при построении и использовании библиотек с системой компилятора GNAT. Кроме того, здесь описывается как можно перекомпилировать библиотеку времени выполнения GNAT.

30.1 Создание Ада-библиотеки

В среде системы компилятора GNAT любая библиотека состоиз из двух компонентов:

- Файлы с исходными текстами.
- Скомпилированный код и `ALI`-файлы (***A**da **L**ibrary **I**nformation files*)

Согласно требований системы компилятора GNAT, при использовании других пакетов необходимо, чтобы некоторые исходные тексты были доступны компилятору. Таким образом, минимально необходимым множеством файлов с исходными текстами являются файлы с исходными текстами спецификаций всех пакетов, которые формируют видимую часть библиотеки, а также файлы с исходными текстами от которых зависят файлы спецификаций пакетов библиотеки. Кроме того, должны быть доступны исходные тексты тел всех видимых настраиваемых модулей. Хотя это не является строго необходимым требованием, рекомендуется, чтобы пользователю были доступны все исходные тексты, которые необходимы для перекомпиляции библиотеки. В результате этого пользователю предоставляются исчерпывающие возможности в использовании межмодульных встроенных вставок (*inline*) и в осуществлении полноценной отладки на уровне исходных текстов. Это также может облегчить ситуацию для тех пользователей, которые нуждаются в обновлении инструментальных средств при котором может возникнуть необходимость перекомпиляции библиотек из исходных текстов.

Обеспечение скомпилированного кода библиотеки может предусматриваться различными способами. Самый простой способ предусматривает непосредственное множество объектных файлов сгенерированных в процессе компиляции библиотеки. Кроме того, возможно объединение всех объектных файлов в едином библиотечном архиве с помощью какой-либо команды, которая предусматривается в используемой операционной системе. И, наконец, возможно создание динамически загружаемой библиотеки (см. использование опции "**`-shared`**" в справочном руководстве по GCC).

Существует множество способов компиляции модулей, совокупность которых формирует библиотеку. Например, это можно выполнить с помощью написания файла управления сборкой проекта "`Makefile`" и последующего использования утилиты GNU "`make`", или путем создания скрипта для командного интерпретатора. В случае построения простой библиотеки, можно создать фиктивную головную программу, которая зависит от всех интерфейсных пакетов библиотеки. В последствии, такая фиктивная головная программа может быть представлена утилите "`gnatmake`", в результате чего "`gnatmake`"выполнит построение всех объектных файлов библиотеки. Ниже показан пример подобной фиктивной головной программы и общий вид команд, которые могут быть использованы для построения единого библиотечного архива объектных файлов или динамически загружаемой библиотеки.

```
with My_Lib.Service1;
with My_Lib.Service2;
with My_Lib.Service3;
```

```
procedure My_Lib_Dummy is
begin
    null;
end;
```

```
# компиляция библиотеки
$ gnatmake -c my_lib_dummy.adb

# нам не нужен объектный код фиктивной головной программы
$ rm my_lib_dummy.o my_lib_dummy.ali

# создание единого библиотечного архива
$ ar rc libmy_lib.a *.o
# some systems may require "ranlib" to be run as well

# или создание динамически загружаемой библиотеки
$ gnatgcc -shared -o libmy_lib.so *.o
# некоторые системы могут требовать, чтобы компиляция выполнялась с опцией -fPIC
```

При группировке объектных файлов библиотеки в единый библиотечный архив или динамически загружаемую библиотеку пользователь должен указывать требуемую библиотеку на этапе компоновки проекта или использовать директиву компилятора "**pragma** Linker_Options" в одном из файлов с исходными текстами:

```
pragma Linker_Options ("-lmy_lib");
```

30.2 Установка Ада-библиотеки

Установка какой-либо библиотеки в системе компилятора GNAT осуществляется копированием файлов, которые составляют библиотеку, в какое-либо определенное место на диске. Существует возможность установки файлов с исходными текстами библиотеки в самостоятельный каталог, отдельно от остальных файлов библиотеки (`ALI`-файлов, объектных файлов, библиотечных архивов). Это возможно благодаря тому, что пути поиска для файлов с исходными текстами и для объектных файлов определяются отдельно.

Системный администратор может установить библиотеки общего назначения в место, которое будет указано для компилятора в пути поиска по умолчанию. Для осуществления этого он должен указать расположение таких библиотек в файлах конфигурации "`ada_source_path`"и "`ada_object_path`", которые должны располагаться там же где и файл "`specs`", который определяет конфигурацию "`gcc`", в дереве установки системы компилятора GNAT. Расположение файла "`specs`", определяющего конфигурацию "`gcc`", можно узнать следующим образом:

```
$ gnatgcc -v
```

Упомянутые файлы конфигурации имеют простой формат: каждая строка должна содержать одно уникальное имя каталога. Такие имена добавляются к соответствующему пути поиска в порядке их появления внутри файла. Имена каталогов могут быть абсолютными или относительными, в последнем случае они указываются относительно расположения этих файлов.

Файлы "`ada_source_path`"и "`ada_object_path`"могут отсутствовать после установки системы компилятора GNAT. В этом случае GNAT будет осуществлять поиск своей библиотеки времени выполнения в каталогах: "`adainclude`— файлы с исходными текстами, и "`adalib`— объектные и `ALI`-файлы. При наличии файлов "`ada_source_path`"и "`ada_object_path`"компилятор не осуществляет поиска в каталогах "`adainclude`"и "`adalib`". Таким образом, файл "`ada_source_path`"должен указывать расположение файлов с исходными текстами библиотеки времени выполнения GNAT (которые могут быть расположены в "`adainclude`"). Подобным образом, файл "`ada_object_path`"должен указывать расположение объектных файлов библиотеки времени выполнения GNAT (которые могут быть расположены в "`adalib`").

Библиотека может быть установлена до или после стандартной библиотеки GNAT путем изменения порядка следования строк в конфигурационных файлах. В общем случае, биибиблиотека должна быть установлена перед стандартной библиотеки GNAT в случае, когда она переопределяет какую-либо часть стандартной библиотеки.

30.3 Использование Ада-библиотеки

При использовании Ада-библиотеки необходимо указывать ее как в пути поиска файлов с исходными текстами, так и в пути поиска объектных файлов. Например, вы можете использовать библиотеку "`mylib`", которая установлена в каталогах "`/dir/my_lib_src`"и "`/dir/my_lib_obj`"следующим образом:

```
$ gnatmake -aI/dir/my_lib_src -aO/dir/my_lib_obj my_appl    -largs -lmy_lib
```

Такая команда может быть упрощена до команды вида:

```
$ gnatmake my_appl
```

когда соблюдаются следующие условия:

- каталог "`/dir/my_lib_src`"добавлен пользователем в переменную окружения "`ADA_INCLUDE_PATH`", или указан администратором системы в файле "`ada_source_path`"
- каталог "`/dir/my_lib_obj`"добавлен пользователем в переменную окружения "`ADA_OBJECTS_PATH`", или указан администратором системы в файле "`ada_object_path`"
- в исходных текстах была использована директива компилятора "**pragma** Linker_Options"

30.4 Перекомпиляция библиотеки времени выполнения GNAT

Бывают случаи когда возникает необходимость в самостоятельной перекомпиляции библиотеки времени выполнения GNAT. Для таких случаев предусматривается специальный файл "`Makefile.adalib`" для управления сборкой проекта с помощью утилиты GNU "`make`". Этот файл находится в каталоге, который содержит библиотеку времени выполнения GNAT. Расположение этого каталога в системе зависит от способа установки окружения ссистемы компилятора GNAT, что можно узнать с помощью команды:

```
$ gnatls -v
```

Последняя строка, которая показывает путь поиска объектных файлов (*Object Search Path*), как правило, содержит путь к библиотеке времени выполнения GNAT. Файл "`Makefile.adalib`" содержит в себе документацию с инструкциями, которые необходимы для самостоятельной сборки и использования новой библиотеки.

Глава 31

Средства управления проектами в системе GNAT

Эта глава описывает средства управления проектами, которые присутствуют в системе компилятора GNAT. Основой этих средств является *Менеджер Проектов GNAT* (*GNAT Project Manager*). Следует заметить, что менеджер проектов появился в системе GNAT версии 3.15 и отсутствует в более ранних версиях. Менеджер проектов позволяет конфигурировать различные свойства для групп файлов с исходными текстами. В частности, он позволяет определять:

- Каталог или набор каталогов, где располагаются файлы с исходными текстами, и/или имена определенных самостоятельных файлов с исходными текстами.
- Каталог в котором будут сохраняться файлы являющиеся результатом работы компилятора (файлы `ALI`, объектные файлы, файлы деревьев).
- Каталог в котором будут сохраняться исполняемые файлы программ (иначе — исполняемые программные модули).
- Установки опций для любых средств обработки проектов ("`gnatmake`", компилятор, редактор связей, компоновщик, "`gnatls`", "`gnatxref`", "`gnatfind`"); такие установки могут быть заданы глобально и/или указаны для индивидуальных программных модулей.
- Исходные файл (или файлы), который содержит главную подпрограмму
- Используемый исходным текстом язык программирования (в настоящий момент Ада и/или C/C++)
- Соглашения по именованию файлов с исходными текстами; допускается глобальная установка и/или установка для индивидуальных модулей.

31.1 Файлы проектов GNAT

Проект является определенным набором значений для перечисленных выше свойств. Установки этих значений могут быть определены в *файле проекта*, который является обычным текстовым файлом с Ада-подобным синтаксисом. В общем случае, значением какого-либо свойства может служить строка или список строк. Свойствам, значения которых не определены явно, назначаются значения по умолчанию. Файл проекта может запрашивать значения *внешних переменных* (определяемых пользователем опций командной строки или переменных окружения), кроме того, для свойств допускается условная установка значений, которая основывается на значениях внешних переменных.

В простейших случаях, исходные файлы проектов зависят только от других исходных файлов одного и того же проекта, или от предопределенных библиотек. Следует заметить, что "зависимость", в данном случае, рассматривается в техническом смысле, например, один Ада-модуль указывает в спецификаторе "**with**" какой-либо другой Ада-модуль. Однако, менеджер проектов позволяет также осуществлять более естественное упорядочивание, когда исходные тексты одного проекта зависят от исходных текстов другого проекта (или проектов):

- Один проект может *импортировать* другие проекты, которые содержат необходимые файлы с исходными текстами.
- Существует возможность упорядочивания проектов GNAT в иерархию: *проект-потомок* проект способен расширить *проект-предок*, наследуя файлы с исходными текстами предка и, при необходимости, заменяя любые из них своими альтернативными версиями.

Таким образом, в общем случае, менеджер проектов GNAT позволяет структурировать большой разрабатываемый проект в иерархию подсистем сборка которой планируется на уровне подсистем и, таким образом, позволяет использовать различную среду компиляции (различные настройки опций) для сборки различных подсистем.

Активация менеджера проектов GNAT осуществляется с помощью опции "`textbf-P`*`projectfile`*" утилиты "`gnatmake`" или управляющей программы "`gnat`" (или "`gnatcmd`"). При необходимости определения в командной строке значения внешней переменной, установка которой опрашивается в файле проекта, дополнительно можно использовать опцию командной строки "`textbf-X`*`vbl=value`*". В результате, менеджер проектов GNAT осуществляет анализ и интерпретацию файла проекта, и управляет запуском инструментальных средств системы GNAT основываясь на установках/настройках свойств проекта.

Менеджер проектов поддерживает обширный диапазон стратегий разработки для систем различного размера. Некоторыми легко обрабатываемыми типичными ситуациями являются:

- Использование общего множества файлов с исходными текстами с генерацией объектных файлов в различные каталоги, в зависимости от различий в установках опций.
- Использование большей части общих файлов, но при этом, использование различных версий для какого-либо модуля (или модулей).

Расположение получаемого в результате сборки проекта исполняемого модуля может быть указано внутри файла проекта или в командной строке путем использования опции "`textbf-o`". При отсутствии этой опции в файле проекта или командной строке, любой исполняемый файл, генерируемый командой "`gnatmake`", будет помещен в каталог "Exec_Dir", который указывается в файле проекта. Если в файле проекта каталог "Exec_Dir" не указан, то исполняемый модуль будет помещен в каталог в котором сохраняются объектные файлы проекта.

Файлы проектов могут использоваться с целью получения некоторых эффектов, которые характерны для систем управления версиями файлов с исходными текстами (например, определяя отдельные проекты для различных наборов файлов с исходными текстами, которые составляют различные реализации разрабатываемой системы). При этом менеджер проектов не зависит от каких-либо средств управления конфигурациями, которые могут быть использованы разработчиками.

Далее, с помощью демонстрируемых примеров, будут рассмотрены основные свойства средств управления проектами GNAT, а также более детально будут рассмотрены синтаксис и семантика файлов проектов.

31.2 Примеры файлов проектов

Рассмотрим несколько простых примеров, которые демонстрируют основные свойства средств управления проектами GNAT и базовую структуру файлов проектов.

31.2.1 Различные опции сборки и каталоги выходных результатов для общих исходных файлов

Предположим, что файлами с исходными текстами на Аде являются следующие файлы: "`pack.ads`", "`pack.adb`" и "`proc.adb`". Предположим также, что все они располагаются в каталоге "`/common`", и файл "`proc.adb`" содержит главную подпрограмму "Proc", которая указывает в спецификаторе "**with**" пакет *Pack*. Необходимо осуществлять компиляцию этих исходных файлов используя два набора опций:

- При отладке, необходимо утилите "`gnatmake`" передать опцию "`textbf-g`", а компилятору опции "`textbf-gnata`", "`textbf-gnato`" и "`textbf-gnatE`"; при этом вывод результатов компиляции должен осуществляться в каталог "`/common/debug`".

- При подготовке версии реализации, компилятору необходимо передать опцию "`textbf-O2`", а вывод результата компиляции должен быть осуществлен в каталог "`/common/release`".

Показанные ниже файлы проектов GNAT, которые позволяют решить поставленную задачу, имеют соответствующие имена "`debug.gpr`" и "`release.gpr`", и располагаются в каталоге "`/common`".

Для наглядности, представим рассматриваемый пример схематически:

```
/common
  debug.gpr
  release.gpr
  pack.ads
  pack.adb
  proc.adb
/common/debug -g, -gnata, -gnato, -gnatE
  proc.ali, proc.o
  pack.ali, pack.o
/common/release -O2
  proc.ali, proc.o
  pack.ali, pack.o
```

Файла проекта "`debug.gpr`" имеет следующий вид:

```
project Debug is
  for Object_Dir use "debug";
  for Main use ("proc");

  package Builder is
    for Default_Switches ("Ada") use ("-g");
  end Builder;

  package Compiler is
    for Default_Switches ("Ada")
       use ("-fstack-check", "-gnata", "-gnato", "-gnatE");
  end Compiler;
end Debug;
```

Файла проекта "`release.gpr`" имеет следующий вид:

```
project Release is
  for Object_Dir use "release";
  for Exec_Dir use ".";
  for Main use ("proc");

  package Compiler is
    for Default_Switches ("Ada") use ("-O2");
  end Compiler;
end Release;
```

Именем проекта, который описан в файле "`debug.gpr`", является "Debug" (регистр символов значения не имеет). Аналогично, проект, который описан в файле "`release.gpr`", имеет имя "Release". Для согласованности, файл проекта должен иметь такое же имя как и проект, а расширением имени файла проекта должно быть расширение "`.gpr`". Такое соглашение не является жестким требованием, однако, при его несоблюдении будет выдаваться предупреждающее сообщение.

Предположим, что текущим каталогом является каталог "`/temp`". Тогда, согласно установок в файле проекта "`debug.gpr`", команда

```
gnatmake -P/common/debug.gpr
```

будет генерировать вывод объектных файлов и файлов `ALI` в каталог "`/common/debug`", и исполняемый файл "`proc`" (в системе Windows "`proc.exe`") также будет помещен в каталог "`/common/debug`".

Подобным образом, согласно установок в файле проекта "`release.gpr`", команда

```
gnatmake -P/common/release.gpr
```

будет генерировать вывод объектных файлов и файлов `ALI` в каталог "`/common/release`", а исполняемый файл "`proc`" (в системе Windows "`proc.exe`") будет помещен в каталог "`/common`".

В случаях, когда файл проекта явно не указывает набор каталогов, в которых храняться файлы с исходными текстами, или непосредственный набор исходных файлов, по умолчанию предполагается, что исходными файлами проекта являются файлы с исходными текстами Ады, которые расположены в том же каталоге, в котором находится файл проекта. Таким образом, файлы "`pack.ads`", "`pack.adb`" и "`proc.adb`" являются исходными файлами для обоих проектов.

Различные свойства проекта выражаются в виде *атрибутов* в стиле языка Ада. Подобным свойством проекта является каталог для сохранения объектных файлов (и файлов `ALI`), которому соответствует атрибут "Object_Dir". Значением атрибута "Object_Dir" может быть строка или строковое выражение. Каталог для сохранения объектных файлов может быть указан как абсолютный или как относительный путь к каталогу. В последнем случае, указывается относительный путь к каталогу в котором содержится файл проекта. Таким образом, в показанных выше примерах, вывод компилятора направляется в каталог "`/common/debug`" (для проекта "Debug"), и в каталог "`/common/release`" (для проекта "Release"). В случае, когда значение "Object_Dir" не указано, значением по умолчанию является каталог в котором содержится файл проекта.

Другим свойством проекта является каталог для сохранения исполняемых файлов (файлов с исполняемыми модулями), которому соответствует атрибут "Exec_Dir". Значением атрибута "Exec_Dir" также может быть строка или строковое выражение, которые указывают абсолютный или относительный путь к каталогу. Когда значение "Exec_Dir" не указано, значением по умолчанию является каталог указанный для "Object_Dir" (который, в свою очередь, может быть каталогом в котором содержится файл проекта, при отсутствии указания значения для "Object_Dir"). Таким образом, в показанных выше примерах, исполняемый файл будет помещен в каталог "`/common/debug`" для проекта "Debug" (атрибут "Exec_Dir" не указан), и в каталог "`/common`" для проекта "Release".

Инструментальные средства системы компилятора GNAT, которые интегрированы с менеджером проектов GNAT, моделируются внутри файла проекта как соответствующие пакеты. В показанных ранее примерах, проект "Debug" описывает пакеты *Builder* (соответствует команде "`gnatmake`") и *Compiler* (соответствует компилятору, команда "`gcc`" или "`gnatgcc`"), а проект "Release" описывает только пакет *Compiler*.

Используемый в файлах проектов синтаксис Ада-пакетов не следует рассматривать буквально. Хотя пакеты файлов проектов обладают внешним сходством с пакетами исходного текста Ады, их нотация является только способом передачи группы свойств именованной сущности. Кроме того, следует заметить, что допустимые к использованию в файлах проектов имена пакетов ограничены предопределенным перечнем имен, находящихся в строгом соответствии с существующими инструментальными средствами системы компилятора GNAT, которые интегрированы с менеджером проектов GNAT, а содержимое таких пакетов ограничено небольшим набором конструкций (в показанных выше примерах пакеты содержат описания атрибутов).

Указание опций для инструментальных средств системы компилятора GNAT, которые интегрированы с менеджером проектов, может быть осуществлено с помощью установки значений соответствующих атрибутов в пакетах файлов проектов, которые соответствуют инструментальным средствам. В показанные выше примеры демонстрируется использование атрибута "Default_Switches", который описан в пакетах обоих файлов проектов и является одним из таких атрибутов. В отличие от простых атрибутов, таких как "Source_Dirs", атрибут "Default_Switches" является *ассоциативным массивом*. При описании такого атрибута необходимо обеспечить какой-либо "индекс" (литеральная строка), а результатом описания атрибута является установка значения "массива" для указанного "индекса". Для атрибута "Default_Switches", индексом является язык программирования (в данном случае — Ada), а указанное (после "**use**") значение должно быть списком строковых выражений.

В файлах проектов допускается использование предопределенного набора атрибутов. При этом, одни атрибуты могут быть указаны на уровне проекта, а другие — на уровне пакета проекта. Для любого атрибута, который является ассоциативным массивом, индекс должен быть представлен как литеральная строка, причем, налагаемые на эту строку ограничения (например, имя файла или имя языка программирования) зависят от индивидуального атрибута. Кроме того, в зависимости от атрибута, указываемое значение атрибута должно быть строкой или списком строк.

В показанном ранее проекте "Debug", осуществлялась установка опций для двух инструментальных средств: команды "`gnatmake`" и компилятора. Таким образом, в файл проекта были включены соответствующие пакеты, и каждый пакет описывал значение атрибута "Default_Switches" для индекса "`Ada`". Следует заметить,

что пакет соответствующий команде "gnatmake" именуется *Builder*. Проект "Release" подобен проекту "Debug", но он содержит только пакет *Compiler*.

Отметим, что в проекте "Debug", все опции, которые начинаются с "textbf-gnat" и указаны в пакете *Compiler*, могут быть перемещены в пакет *Builder*, поскольку команда "gnatmake" передает все подобные опции компилятору.

Одним из свойств проекта является список головных подпрограмм (фактически, список имен файлов с исходными текстами содержащими головные подпрограммы, причем, указание расширений имен файлов — не обязательно). Это свойство указывается атрибутом "Main", значением которого является список строк. Когда проект описывает атрибут "Main", при запуске команды "gnatmake", отсутствует необходимость в указании головной подпрограммы (или головных подпрограмм).

Когда файл проекта не определяет использование какой-либо схемы именования файлов, используется стандартная для GNAT схема именования файлов принимаемая по умолчанию. Механизм, который используется в файлах проектов для определения соглашений именования файлов с исходными текстами, обеспечивается пакетом *Naming* и будет подробно рассмотрен несколько позже.

Когда файл проекта не определяет значение атрибута "Languages" (указывает используемые в проекте языки программирования), инструментальные средства GNAT по умолчанию подразумевают, что языком программирования является Ada. В общем случае предполагается, что проект может состоять из файлов с исходными текстами, которые написаны на языках программирования: Ada, C и/или других языках программирования.

31.2.2 Использование внешних переменных

Вместо написания различных самостоятельных файлов проектов, для получения отладочной версии и версии реализации, можно написать единственный файл проекта, который опрашивает состояние внешних переменных (могут быть установлены как переменные окружения или переданы в командной строке), и осуществляет условную подстановку соответствующих значений. Предположим, что исходные тексты "pack.ads", "pack.adb" и "proc.adb" расположены в каталоге "/common". Показанный ниже файл проекта "build.gpr", осуществляет опрос состояния внешней переменной с именем "STYLE", и определяет, таким образом, расположение каталога для сохранения объектных модулей и используемые опции, в зависимости от значения этой переменной. При этом, когда значением переменной "STYLE" является "deb" (*debug*) — осуществляется сборка отладочной версии, а когда значением переменной является "rel" (*release*) — версия реализации. По умолчанию, значением переменной "STYLE" является "deb".

```
project Build is
  for Main use ("proc");

  type Style_Type is ("deb", "rel");
  Style : Style_Type := external ("STYLE", "deb");

  case Style is
    when "deb" =>
      for Object_Dir use "debug";

    when "rel" =>
      for Object_Dir use "release";
      for Exec_Dir use ".";
  end case;


  package Builder is
    case Style is
      when "deb" =>
        for Default_Switches ("Ada") use ("-g");
    end case;
  end Builder;


  package Compiler is
    case Style is
```

```
         when "deb" =>
           for Default_Switches ("Ada") use ("-gnata", "-gnato", "-gnatE");

         when "rel" =>
           for Default_Switches ("Ada") use ("-O2");
       end case;
     end Compiler;

  end Build;
```

Тип "Style_Type" является примером *строкового типа* (*string type*), который в файлах проектов является своеобразным аналогом перечислимого типа Ады, и вместо идентификаторов содержит строковые литералы. Переменная "Style" описана как переменная этого типа.

Форма "**external** ("STYLE", "deb")" является *внешним обращением* (или *внешней ссылкой — external reference*). Первый аргумент такого внешнего обращения является именем *внешней переменной* (*external variable*), а второй аргумент — определяет значение, которое будет использоваться как значение по умолчанию в случае отсутствия указанной внешней переменной. Внешняя переменная может быть определена с помощью опции командной строки "**-X**", или, в качестве внешней переменной, может быть использована переменная окружения.

Каждая конструкция "**case**" расширяется менеджером проектов согласно значения переменной "Style". Таким образом, команда

```
gnatmake -P/common/build.gpr -XSTYLE=deb
```

эквивалентна запуску команды "`gnatmake`", которая использует файл проекта "`debug.gpr`" из ранее рассмотренного примера. Кроме того, для данного примера, аналогичным образом будет обработана команда

```
gnatmake -P/common/build.gpr
```

Поскольку значение "`deb`" является значением по умолчанию для переменной "STYLE".

Аналогичным образом, команда

```
gnatmake -P/common/build.gpr -XSTYLE=rel
```

является эквивалентом запуска команды "`gnatmake`", которая использует файл проекта "`release.gpr`" из ранее рассмотренного примера.

31.2.3 Импорт других проектов

Какой-либо компилируемый модуль расположенный в файле с исходным текстом, который принадлежит одному проекту, может зависить от компилируемых модулей расположенных в файлах с исходными текстами, которые принадлежат другим проектам. Для получения такого поведения, зависимый проект должен *импортировать* проекты, которые содержат необходимые файлы с исходными текстами. Достижение такого эффекта заключено в синтаксисе, который подобен использованию спецификатора "**with**" в языке Ада. При этом указываемыми в "**with**" сущностями являются строки, которые обозначают файлы проектов.

В качестве простого примера предположим, что два проекта "GUI_Proj" и "Comm_Proj" описаны в файлах проектов "`gui_proj.gpr`" и "`comm_proj.gpr`", которые расположены в каталогах "`/gui`" и "`/comm`" соответственно. Предположим также, что исходными файлами проекта "GUI_Proj" являются файлы "`gui.ads`" и "`gui.adb`", а исходными файлами проекта "Comm_Proj" являются файлы "`comm.ads`" и "`comm.adb`", и файлы каждого проекта размещаются в каталоге соответствующего проекта. Для наглядности, представим схематическую диаграмму:

```
/gui
  gui_proj.gpr
  gui.ads
  gui.adb

/comm
  comm_proj.gpr
  comm.ads
  comm.adb
```

Предположим, что в каталоге "/app" необходимо разработать приложение, которое будет указывать в спецификаторе "**with**" пакеты *GUI* и *Comm*, используя свойства соответствующих файлов проекта (например, установки опций и указание каталога сохранения объектных файлов). Скелет кода для головной процедуры приложения может иметь следующий вид:

```
with GUI, Comm;
procedure App_Main is
   ...
begin
   ...
end App_Main;
```

Файл проекта приложения "app_proj.gpr", который позволяет добиться желаемого эффекта, может выглядеть следующим образом:

```
with "/gui/gui_proj", "/comm/comm_proj";
project App_Proj is
   for Main use ("app_main");
end App_Proj;
```

Сборка исполняемого файла приложения может быть получена с помощью команды:

```
gnatmake -P/app/app_proj
```

При этом, генерация исполняемого модуля "app_main" осуществляется в том же каталоге где располагается файл проекта "app_proj.gpr".

Когда импортируемый файл проекта использует стандартное расширение имени файла (".gpr") указание расширение имени файла в предложении "**with**" файла проекта может быть опущено (как показано в примере выше).

Показанный выше пример использует указание абсолютных путей для каждого импортируемого файла проекта. Альтернативно, каталог может быть не указан, когда:

- Импортируемый файл проекта располагается в том же каталоге, в котором расположен импортирующий файл проекта.
- Определена переменная окружения "ADA_PROJECT_PATH", которая включает каталог содержащий необходимый файл проекта.

Следовательно, при наличии переменной окружения "ADA_PROJECT_PATH", которая включает каталоги "/gui" и "/comm", файл проекта приложения "app_proj.gpr", может быть написан следующим образом:

```
with "gui_proj", "comm_proj";
project App_Proj is
   for Main use ("app_main");
end App_Proj;
```

Следует учитывать, что способность импорта других проектов обладает потенциальной двусмысленностью. Например, один и тот же модуль может присутствовать в различных импортируемых проектах, или такой модуль может присутствовать как в импортируемом проекте, так и в импортирующем проекте. Оба случая являются условием возникновения ошибки. Следует заметить, что для текущей версии менеджера проектов GNAT (версия 3.15) наличие двусмысленного модуля является недопустимым даже тогда, когда импортирующий проект не содержит обращений к этому модулю. Это строгое правило может быть ослаблено в последующих реализациях менеджера проектов.

31.2.4 Расширение существующего проекта

Общим случаем для больших программных систем является наличие множества реализаций общего интерфейса. В терминах Ады, это представляется как множество версий тела пакета для одной и той же спецификации этого пакета. Например, одна реализация может быть безопасна при использовании в многозадачных/многопоточных программах, а другая может быть более эффективно использована в последовательных/однопоточных приложениях В среде GNAT это может быть смоделировано с помощью использования концепции *расширения проекта* (*project extension*). Если один проект ("проект-потомок") *расширяет* другой проект ("проект-предок"), то по умолчанию, все исходные файлы проекта-предка наследуются

проектом-потомком, но проект-потомок может заменить версию любого исходного файла проекта-предка новой версией, а также способен добавить новые файлы. Для проектов, такая способность аналогична использованию расширения в объектно-ориентированном программировании. В среде GNAT допускаются иерархии проектов: проект-потомок может служить проектом-предком для другого проекта, и проект, который наследует один проект, может также импортировать другие проекты.

В качестве примера предположим, что каталог "`/seq`" содержит файл проекта "`seq_proj.gpr`" и исходные файлы "`pack.ads`", "`pack.adb`" и "`proc.adb`":

```
/seq
  pack.ads
  pack.adb
  proc.adb
  seq_proj.gpr
```

Следует заметить, что файл проекта может быть просто пуст (то есть, он не содержит каких-либо описаний атрибутов и/или пакетов):

```
project Seq_Proj is
end Seq_Proj;
```

допуская, что все его исходные файлы являются исходными файлами Ады и все они расположены в каталоге проекта.

Предположим, что необходимо предоставить какую-либо альтернативную версию файла "`pack.adb`" в каталоге "`/tasking`", но, при этом, сохранить использование существующих версий файлов "`pack.ads`" и "`proc.adb`". С этой целью можно описать проект "Tasking_Proj", который наследует проект "Seq_Proj". Схематически это будет иметь следующий вид:

```
/tasking
  pack.adb
  tasking_proj.gpr
```

При этом, файл проекта "`tasking_proj.gpr`" может быть следующим:

```
project Tasking_Proj extends "/seq/seq_proj" is
end Tasking_Proj;
```

Таким образом, используемая в процессе сборки версия файла "`pack.adb`", будет зависеть от указываемого файла проекта.

Следует заметить, что для решения рассматриваемой задачи можно вместо наследования проекта использовать импорт проекта. Базовый проект "`base`" будет содержать исходные файлы "`pack.ads`" и "`proc.adb`", последовательный/однопоточный проект будет импортировать проект "`base`" и добавлять "`pack.adb`", а многозадачный/многопоточный проект будет, подобным образом, импортировать проект "`base`" и добавлять свою версию "`pack.adb`", Фактический выбор решения зависит от необходимости замены других исходных файлов оригинального проекта. При отсутствии такой необходимости достаточно использовать импорт проекта. В противном случае, необходимо использовать расширение проекта.

31.3 Синтаксис файлов проектов

Рассмотрим структуру и синтаксис файлов проектов. Проект может быть *независимым проектом*, который полностью описывается в одном единственным файле проекта. В независимом проекте, любой исходный файл Ады может зависить только от предопределенной библиотеки и/или от других исходных файлов Ады этого же проекта.

Проект может также *зависеть* от других проектов в одном или обоих следующих случаях:

- проект может импортировать любое количество других проектов
- проект может расширять не более одного другого проекта

Отношения зависимости проектов могут быть представлены как ориентированный граф без петель (подграф отображает в дереве отношение "расширения").

Непосредственными исходными файлами проекта являются исходные файлы, которые прямо определяются проектом. При этом, исходные файлы проекта могут определяться неявно, когда они располагаются в том же каталоге в котором расположен файл проекта, или исходные файлы проекта могут определяться явно, с помощью любого рассматриваемого ниже, относящегося к исходным файлам атрибута. В общем смысле, исходные файлы проекта "`proj`" являются непосредственными исходными файлами проекта "`proj`" одновременно со всеми непосредственными исходными файлами проектов от которых проект "`proj`" прямо или косвенно зависит (кроме тех исходных файлов, которые заменяются при расширении, в случае наследования проекта).

31.3.1 Базовый синтаксис

Как видно в показанных ранее примерах, файлы проектов обладают Ада-подобным синтаксисом. Минимальный файл проекта имеет следующий вид:

```
project Empty is

end Empty;
```

Здесь, идентификатор "Empty" является именем проекта. Это имя проекта должно присутствовать после зарезервированного слова "**end**", в конце файла проекта, и должно сопровождаться символом точки с запятой ("`;`").

Любое имя в файле проекта, такое как имя проекта или имя переменной, обладает таким же синтаксисом как идентификатор Ады.

Файлы проектов предполагают использование таких же зарезервированных слов, которые используются в языке Ада, а также дополнительные зарезервированные слова: "**extends**", "**external**" и "**project**". Следует заметить, что в настоящее время синтаксис файлов проектов реально использует только следующие зарезервированные слова Ады:

case	**others**	**use**
end	**package**	**when**
for	**renames**	**with**
is	**type**	

Файлы проектов используют такой же синтаксис комментариев, который используется в языке программирования Ада: комментарий начинается с двух следующих подряд символов дефиса ("`--`") и распространяется вплоть до завершения строки.

31.3.2 Пакеты

Файл проекта может содержать *пакеты*. Именем пакета должен быть один из предопределенных идентификаторов (не зависит от регистра символов), кроме того, пакет с указанным именем может упоминаться в файле проекта только однократно. Список предопределенных идентификаторов имен для пакетов файлов проектов следующий:

Naming	Binder	Cross_Reference
Builder	Linker	gnatls
Compiler	Finder	

Следует заметить, что полный список имен пакетов и их атрибуты указываются в файле "`prj-attr.adb`" (из комплекта файлов с исходными текстами компилятора GNAT).

В простейшем случае, пакет файла проекта может быть пустым:

```
project Simple is
  package Builder is
  end Builder;
end Simple;
```

Как будет показано далее, пакет файла проекта может содержать *описания атрибутов*, *описания переменных* и *конструкции* "**case**".

При наличии двусмысленности между именем проекта и именем пакета (в файле проекта), имя всегда обозначает проект. Для предотвращения таких коллизий, рекомендуется избегать именования проектов с помощью имен, которые предназначены для именования пакетов файлов проектов, или использовать имена, которые начинаются с "`gnat`".

31.3.3 Выражения

Какое-либо *выражение* является или *строковым выражением*, или *выражением списка строк*.

Какое-либо *строковое выражение* является или *простым строковым выражением*, или *составным строковым выражением*.

Какое-либо *простое строковое выражение* является:

- Строковым литералом (например, "`comm/my_proj.gpr`")
- Обращеие к переменной обладающей строковым значением (см. "Переменные")
- Обращение к атрибуту обладающему строковым значением (см. "Атрибуты")
- Внешняя ссылка (см. "Внешние ссылки в Файлах Проектов")

Какое-либо *составное строковое выражение* является конкатенацией строковых выражений с помощью символа "`&`". Например:

```
Path & "/" & File_Name & ".ads"
```

Какое-либо *выражение списка строк* является *простым выражением списка строк* или *составным выражением списка строк*.

Каким-либо *простым выражением списка строк* является:

- Заключенный в скобки список, состоящий из нуль или более строковых выражений, разделенных запятыми:

```
File_Names := (File_Name, "gnat.adc", File_Name & ".orig");
Empty_List := ();
```

- Обращение к переменной обладающей значением списка строк
- Обращение к атрибуту обладающему значением списка строк

Каким-либо *составным выражением списка строк* является конкатенация простого выражения списка строк и какого-либо выражения с помощью символа "`&`". Примечательно, что каждая лексема составного выражения списка строк, за исключением первой, может быть как строковым выражением, так и выражением списка строк. Например:

```
File_Name_List := () & File_Name;
-- в этом списке одна строка

Extended_File_Name_List := File_Name_List & (File_Name & ".orig");
-- две строки

Big_List := File_Name_List & Extended_File_Name_List;
-- Конкатенация двух списков строк: три строки

Illegal_List := "gnat.adc" & Extended_File_Name_List;
-- не допустимо: конкатенация должна начинаться со списка строк
```

31.3.4 Строковые типы

Значение строковой переменной может быть ограничено списком строковых литералов. Ограниченный список строковых литералов представляется как *описание строкового типа*.

Примером описания строкового типа может служить следующее:

```
type OS is ("NT, "nt", "Unix", "Linux", "other OS");
```

Переменные строкового типа называют *типированными переменными*, а все остальные переменные называют *нетипированными переменными*. Типированные переменные удобно использовать в конструкциях "**case**".

Любое описание строкового типа начинается с зарезервированного слова "**type**", за которым следует имя строкового типа (не зависит от регистра символов), затем следует зарезервированное слово "**is**", далее — помещенный в скобки список из одного или более строковых литералов, которые отделяются друг от друга запятыми, и, в завершение, следует символ точки с запятой.

Строковые литералы, которые помещаются в список, являются зависимыми от регистра символов и должны отличаться друг от друга. Они могут содержать в себе любые допустимые для языка Ада символы, включая пробелы.

Строковый тип может быть описан только на уровне файла проекта, и не может быть описан внутри пакета файла проекта.

Ссылку (обращение) к строковому типу можно осуществить с помощью имени строкового типа, когда он описан в том же самом файле проекта, или путем использования точечной нотации: имя проекта, символ точки, имя строкового типа.

31.3.5 Переменные

Переменная может быть описана на уровне файла проекта или внутри пакета файла проекта. Примерами описания переменных может служить следующее:

```
This_OS : OS := external ("OS"); -- описание типированной переменной
That_OS := "Linux";              -- описание нетипированной переменной
```

Любое *описание типированной переменной* начинается с имени переменной за которым следует символ двоеточия, затем следует имя строкового типа, сопровождаемое ":=" и, далее, простое строковое выражение.

Любое *описание нетипированной переменной* начинается с имени переменной за которым следует ":=", сопровождаемое выражением. Следует заметить, что несмотря на терминологию, такая форма "описания" больше похоже на присваивание чем на описание в языке Ада. Такая форма является описанием в нескольких смыслах:

- Имя переменной не нуждается в предварительном описании.
- Описание основывает *разновидность* (строка или список строк) переменной, и последующие описания той же переменной должны быть согласованы

Описание строковой переменной (типированной или нетипированной) описывает переменную значением которой является строка. Эта переменная может быть использована как строковое выражение. Например:

```
File_Name       := "readme.txt";
Saved_File_Name := File_Name & ".saved";
```

Описание переменной списка строк описывает переменную значением которой является список строк. Такой список может содержать любое число (нуль и более) строк.

```
Empty_List := ();
List_With_One_Element := ("-gnaty");
List_With_Two_Elements := List_With_One_Element & "-gnatg";
Long_List := ("main.ada", "pack1_.ada", "pack1.ada", "pack2_.ada"
              "pack2.ada", "util_.ada", "util.ada");
```

Одна и та же типированная переменная не может быть описана более одного раза на уровне проекта и она не может быть описана более одного раза в любом пакете файла проекта. Типированная переменная подобна константе или переменной, которая доступна только для чтения.

Одна и та же нетипированная переменная может быть описана более одного раза. В таком случае, новое значение переменной будет заменять ее старое значение, и последующие ссылки (обращения) к этой переменной будут использовать новое значение. Однако, как отмечалось ранее, если переменная была описана как строковая, то все последующие описания должны предоставлять строковое значение. Подобным образом, если переменная была описана как список строк, все последующие описания переменной должны предоставлять значение в виде списка строк.

Любая *ссылка к переменной* (или *обращение к переменной*) может иметь несколько форм:

- Имя переменной, для переменных расположенных в текущем пакете (если есть) или в текущем проекте
- Имя контекста, сопровождаемое символом точки и, далее, имя переменной.

В качестве *контекста* переменной может служить:

- Имя существующего пакета в текущем проекте.
- Имя импортированного проекта в текущем проекте.
- Имя проекта-предка (например, какой-либо проект, прямо или косвенно, расширяется текущим проектом).
- Имя проекта-предка/импортируемого проекта, сопровождаемое символом точки и именем имени пакета.

Ссылка (обращение) к переменной может быть использована в выражении.

31.3.6 Атрибуты

Проект (и его пакеты) может иметь *атрибуты*, которые описывают свойства проекта. Одни атрибуты имеют значения, которыми являются строки, другие атрибуты имеют значения, которыми являются списки строк. Существуют две категории атрибутов: *простые атрибуты* и *ассоциативные массивы*.

Используемые имена атрибутов строго ограничены — все имена атрибутов предопределены. Существуют атрибуты проектов и атрибуты пакетов (для каждого пакета). Имена атрибутов не зависят от регистра символов.

Ниже перечислены атрибуты проектов (все они являются простыми атрибутами):

Имя атрибута	**Значение**	**Имя атрибута**	**Значение**
Source_Files	список строк	Languages	список строк
Source_Dirs	список строк	Library_Dir	строка
Source_List_File	строка	Library_Name	строка
Object_Dir	строка	Library_Kind	строка
Exec_Dir	строка	Library_Elaboration	строка
Main	список строк	Library_Version	строка

Ниже перечислены атрибуты пакета *Naming*:

Имя атрибута	Категория	Индекс	Значение
Specification_Suffix	ассоциативный массив	имя языка	строка
Implementation_Suffix	ассоциативный массив	имя языка	строка
Separate_Suffix	простой атрибут	-	строка
Casing	простой атрибут	-	строка
Dot_Replacement	простой атрибут	-	строка
Specification	ассоциативный массив	имя модуля Ады	строка
Implementation	ассоциативный массив	имя модуля Ады	строка
Specification_Exceptions	ассоциативный массив	имя языка	список строк
Implementation_Exceptions	ассоциативный массив	имя языка	список строк

Ниже перечислены атрибуты пакетов *Builder*, *Compiler*, *Binder*, *Linker*, *Cross_Reference* и *Finder* (см. также "Опции и Файлы проектов")

Имя атрибута	Категория	Индекс	Значение
Default_Switches	ассоциативный массив	имя языка	список строк
Switches	ассоциативный массив	имя файла	список строк

Дополнительно, пакет *Builder* обладает однострочными атрибутами "Local_Configuration_Pragmas" и "Global_Configuration_Pragmas"; атрибуты пакета *Glide* не документированы и предназначены для внутреннего использования.

Каждый простой атрибут обладает значением по умолчанию: пустая строка для атрибутов обладающих строковым значением, и пустой список для атрибутов обладающих значением списка строк.

Подобно описаниям переменных, какое-либо описание атрибута определяет новое значение атрибута.

Ниже показаны примеры простых описаний атрибутов:

```
for Object_Dir use "objects";
for Source_Dirs use ("units", "test/drivers");
```

Любое *простое описание атрибута* начинается с зарезервированного слова "**for**", после которого следует имя атрибута, сопровождаемое зарезервированным словом "**use**", за которым следует выражение (разновидность выражения зависит от атрибута), и, в завершение, следует символ точки с запятой.

Ссылки (обращения) к атрибутам могут быть использованы в выражениях. Общая форма такого обращения имеет вид:

```
entity'attribute
```

Где "`entity`" является сущностью для которой определен атрибут "`attribute`". Для атрибутов, которые принадлежат к категории ассоциативных массивов, после имени атрибута необходимо в скобках указать строковый литерал, который используется в качестве индекса. Для наглядности, продемонстрируем несколько примеров:

```
project'Object_Dir
Naming'Dot_Replacement
Imported_Project'Source_Dirs
Imported_Project.Naming'Casing
Builder'Default_Switches ("Ada")
```

Сущностью "`entity`" может являться:

- "**project**", для какого-либо атрибута текущего проекта

- имя существующего в текущем проекте пакета
- имя какого-либо импортируемого проекта
- имя какого-либо проекта-предка (расширяемого текущим проектом)
- имя какого-либо импортируемого/проекта-предка, сопровождаемое точкой и, затем, именем пакета

Например:

```
project Prj is
   for Source_Dirs use project'Source_Dirs & "units";
   for Source_Dirs use project'Source_Dirs & "test/drivers"
end Prj;
```

В показанном выше примере, при первом описании атрибута "Source_Dirs", его начальным значением является значение по умолчанию, то есть, пустой список строк. После первого описания, атрибут "Source_Dirs" является списком строк, который содержит один элемент ("`units`"), а после второго описания — два элемента ("`units`" и "`test/drivers`").

Следует заметить, что показанный выше пример приведен только в качестве демонстрации. На приктике, файл проекта, как правило, будет содержать только одно описание атрибута:

```
   for Source_Dirs use ("units", "test/drivers");
```

31.3.7 Атрибуты как ассоциативные массивы

Некоторые атрибуты описываются как *ассоциативные массивы*. Ассоциативный массив можно рассматривать как функцию, которая принимает в качестве параметра строку, и возвращает строку или список строк как значение результата.

Ниже демонстрируется нескольно примеров описания атрибутов как ассоциативных массивов:

```
   for Implementation ("main") use "Main.ada";
   for Switches ("main.ada") use ("-v", "-gnatv");
   for Switches ("main.ada") use Builder'Switches ("main.ada") & "-g";
```

Подобно нетипированным переменным и простым атрибутам, атрибуты ассоциативных массивов могут быть описаны несколько раз. В результате каждого описания атрибут получает новое значение, замещаяя предыдущую установку.

31.3.8 Конструкция `case`

Конструкция "**case**" используется внутри файла проекта с целью обеспечения условного поведения. Типичным примером может служить следующее:

```
project MyProj is
   type OS_Type is ("Linux", "Unix", "NT", "VMS");

   OS : OS_Type := external ("OS", "Linux");

   package Compiler is
     case OS is
       when "Linux" | "Unix" =>
         for Default_Switches ("Ada") use ("-gnath");
       when "NT" =>
         for Default_Switches ("Ada") use ("-gnatP");
       when others =>
     end case;
   end Compiler;
end MyProj;
```

Синтаксис конструкции "**case**" основан на синтаксисе инструкции выбора "**case**" языка Ада (хотя, в данном случае, не существует конструкции "**null**" для пустых альтернатив).

Следом за зарезервированным словом "**case**" указывается переменная выбора (типированная строковая переменная), далее — зарезервированное слово "**is**", а затем последовательность из одной и/или более альтернатив выбора. Каждая альтернатива выбора состоит из зарезервированного слова "**when**" в сопровождении списка строковых литералов (разделяемых символом "|"), или зарезервированного слова "**others**"; далее следует лексема "=>". Каждый строковый литерал должен принадлежать к строковому типу переменной выбора. При наличии альтернативы "**others**", она должна быть последней в перечне альтернатив. Завершителем конструкции "**case**" служит последовательность "**end case**;".

После каждой лексемы "=>" присутствует нуль или более конструкций. Внутри конструкции "**case**" допускается использовать только другие конструкции "**case**" и описания атрибутов. Описания строковых типов, описания переменных и пакетов являются недопустимыми.

Достаточно часто значение переменной выбора указывается как значение внешней ссылки (внешней переменной).

31.4 Исходные, объектные и исполняемые файлы проекта

Каждый проект обладает единственным каталогом для сохранения объектных файлов, и одним или более каталогом с исходными файлами. Каталоги с исходными файлами должны содержать как минимум по одному исходному файлу, если файл проекта явно не указывает на отсутствие исходных файлов.

31.4.1 Каталог объектных файлов

Каталог для сохранения объектных файлов проекта является каталогом в котором будут сохраняться результаты работы компилятора (файлы "`ALI`" и файлы объектных модулей), которые получены в результате обработки непосредственных исходных файлов проекта. Следует заметить, что для унаследованных исходных файлов (при расширении проекта-предка) будет использоваться каталог для сохранения объектных файлов проекта-предка.

Каталог для сохранения объектных файлов указывается внутри файла проекта как значение атрибута "Object_Dir".

```
for Object_Dir use "objects";
```

Атрибут "Object_Dir" содержит строковое значение — путь к каталогу для сохранения объектных файлов проекта. Путь к каталогу может быть указан как абсолютный или как относительный (задан относительно каталога в котором расположен файл проекта). Указываемый каталог должен существовать и должен быть доступен для чтения и записи.

По умолчанию, когда отсутствует явное описание значения атрибута "Object_Dir" или значением атрибута "Object_Dir" является пустая строка, как каталог для сохранения объектных файлов проекта будет использован каталог в котором расположен файл проекта.

31.4.2 Каталог исполняемых файлов

Каталогом для сохранения исполняемых файлов проекта является каталог, который содержит исполняемые файлы головных подпрограмм проекта.

Каталог для сохранения исполняемых файлов проекта указывается внутри файла проекта как значение атрибута "Exec_Dir".

```
for Exec_Dir use "executables";
```

Атрибут "Exec_Dir" содержит строковое значение — путь к каталогу для сохранения исполняемых файлов проекта. Путь к каталогу может быть указан как абсолютный или как относительный (задан относительно каталога в котором расположен файл проекта). Указываемый каталог должен существовать и должен быть доступен для записи.

По умолчанию, когда отсутствует явное описание значения атрибута "Exec_Dir" или значением атрибута "Exec_Dir" является пустая строка, как каталог для сохранения исполняемых файлов проекта будет использован каталог в котором расположен файл проекта.

31.4.3 Каталоги исходных файлов

Каталоги с исходными файлами проекта могут быть указаны с помощью атрибута файла проекта "Source_Dirs".

Значением этого атрибута является список строк. При отсутствии явного описания значения атрибута "Source_Dirs", по умолчанию предполагается, что существует единственный каталог с исходными файлами проекта, и этим каталогом является каталог в котором расположен файл проекта.

В случае, когда явно указывается, что значением атрибута "Source_Dirs" является пустой список:

```
for Source_Dirs use ();
```

предполагается, что проект не содержит исходных файлов.

В противном случае, каждая строка, в списке строк, обозначает один или более каталогов с исходными файлами:

```
for Source_Dirs use ("sources", "test/drivers");
```

Если строка в списке заканчивается на "`/**`", то каталог, имя которого предшествует двум звездочкам, а также все его подкаталоги (рекурсивно) являются каталогами с исходными файлами проекта:

```
for Source_Dirs use ("/system/sources/**");
```

В показанном выше примере, каталог "`/system/sources`" и все его подкаталоги (рекурсивно) являются каталогами с исходными файлами проекта.

Чтобы указать, что каталогами с исходными файлами проекта являются каталог содержащий файл проекта и все его подкаталоги, можно описать атрибут "Source_Dirs" следующим образом:

```
for Source_Dirs use ("./**");
```

Каждый каталог с исходными файлами должен существовать и должен быть доступен по чтению.

31.4.4 Имена исходных файлов

В проекте, который содержит исходные файлы, имена исходных файлов могут быть указаны с помощью атрибутов "Source_Files" (список строк) и/или "Source_List_File" (строка). Имена исходных файлов никогда не содержат в себе информацию о каталоге.

Если для атрибута "Source_Files" дано явное значение, то каждый элемент списка является именем исходного файла проекта.

```
for Source_Files use ("main.adb");
for Source_Files use ("main.adb", "pack1.ads", "pack2.adb");
```

Если для атрибута "Source_Files" явное значение не дано, но для атрибута "Source_List_File" указано строчное значение, то имена исходных файлов содержатся в текстовом файле, полное имя которого является значением атрибута "Source_List_File" (при этом, путь к каталогу, который содержится в полном имени файла, может быть указан как абсолютный или как заданный относительно каталога в котором располагается файл проекта).

В указанном файле, именем исходного файла является каждая непустая и не являющаяся комментарием строка. Строки комментариев начинаются с двух дефисов.

```
for Source_List_File use "source_list.txt";
```

По умолчанию, когда значения атрибутов "Source_Files" и "Source_List_File" не заданы, каждый файл, обнаруженный в каком-либо каталоге для исходных файлов проектов, имя которого соответствует используемой в проекте схеме именования, является непосредственным исходным файлом проекта.

В случае одновременной установки явных значений для атрибутов "Source_Files" и "Source_List_File" осуществляется генерация предупреждающего сообщения. При этом, более приоритетным является значение атрибута "Source_Files".

Каждое имя исходного файла должно быть именем единственного исходного файла, который существует в одном из каталогов с исходными файлами проекта.

В случае, когда при описании атрибута "Source_Files" его значением является пустой список, предполагается, что проект не содержит исходных файлов.

Любой проект должен содержать по крайней мере один непосредственный исходный файл, за исключением проектов, для которых точно указано отсутствие исходных файлов Ады (атрибут "Source_Dirs" или атрибут "Source_Files" описан как пустой список, или атрибут "Languages" описан без указания в списке языка "Ada"):

```
for Source_Dirs use ();
for Source_Files use ();
for Languages use ("C", "C++");
```

Проекты без исходных файлов полезно использовать в качестве шаблонных пакетов для других проектов. В частности, для описания пакета *Naming*.

31.5 Импорт проектов

Какой-либо непосредственный исходный файл проекта "P" может зависеть от исходных файлов, которые не являются непосредственными исходными файлами проекта "P" и не содержатся в предопределенной библиотеке. Для получения такого эффекта проект "P" должен *импортировать* проекты, которые содержат необходимые исходные файлы:

```
with "project1", "utilities.gpr";
with "/namings/apex.gpr";
project Main is
   ...
```

В показанном выше примере можно увидеть, что синтаксис, который используется для импортирования проектов, подобен синтаксису Ады, который используется для импортирования компилируемых моделей. Однако, в файлах проектов вместо имен используются строковые литералы, и в спецификаторе "**with**" указываются файлы проектов а не пакеты.

Каждый строковый литерал является именем файла проекта или полным именем файла проекта (абсолютным или относительным). Если строка является простым именем файла, без указания пути к каталогу, то расположение файла определяется с помощью *пути к каталогу проекта* (*project path*):

- Если переменная окружения "ADA_PROJECT_PATH" существует, то путь к каталогу проекта содержит все каталоги, которые указаны в этой переменной окружения, плюс каталог в котором расположен текущий файл проекта.
- Если переменная окружения "ADA_PROJECT_PATH" не существует, то путь к каталогу проекта содержит только один каталог: каталог в котором расположен текущий файл проекта.

Если используется относительный путь к каталогу проекта

```
with "tests/proj";
```

то путь к каталогу проекта определяется относительно каталога в котором расположен импортирующий (текущий) файл проекта. Полная расшифровка любых символьных ссылок осуществляется в каталоге импортирующего файла проекта перед обнаружением и выборкой импортируемого файла проекта.

Когда имя файла проекта, которое указано в спецификаторе "**with**", не содержит расширения имени файла, по умолчанию, подразумевается расширение имени файла ".gpr". Если указанное имя файла с таким расширением не обнаружено, то будет использовано имя файла без расширения (как непосредственно указано в спецификаторе "**with**"). Таким образом, согласно показанного выше примера: если обнаружен файл "project1.gpr", то он будет использован; в противном случае, если обнаружен файл "project1", то он будет использован; если не найден ни один из этих файлов, то это будет ошибкой.

При несовпадении имени проекта с именем файла проекта, генерируется предупреждающее сообщение (эта проверка не зависит от регистра символов).

Любой исходный файл, который является непосредственным исходным файлом импортируемого проекта, может быть использован непосредственным исходным файлом импортирующего проекта, и рекурсивно. Таким образом, если "A" импортирует "B", а "B" импортирует "C", то непосредственные исходные файлы "A" могут зависеть от непосредственных исходных файлов "C", даже если "A" не импортирует "C" явным

образом. Однако, использование такого подхода не рекомендуется, поскольку возможна ситуация, когда "B" прекратит импортировать "C"; после чего некоторые исходные файлы в "A" перестанут компилироваться.

Побочным эффектом такой способности является то, что циклические зависимости — не допустимы: если "A" импортирует "B" (прямо или косвенно), то для "B" не разрешается импортировать "A".

31.6 Расширение проекта

Иногда, при разработке большой программной системы, необходимо использование модифицированных версий некоторых исходных файлов без изменения оригинальных версий исходных файлов. Такое поведение может быть достигнуто с помощью возможности *расширения проекта*:

```
project Modified_Utilities extends "/baseline/utilities.gpr" is ...
```

Файл проекта для расширяемого проекта (*проект-предок*) указывается строковым литералом, указываемым вслед за зарезервированным словом "**extends**", которое следует за именем расширяющего проекта (*проект-потомок*).

По умолчанию, проект-потомок наследует все исходные файлы проекта-предка. Однако, унаследованные исходные файлы могут быть переопределены: какой-либо модуль с таким же именем как модуль проекта-предка будет "скрывать" оригинальный модуль проекта-предка. Наследуемые исходные файлы рассматриваются как исходные файлы (но не как непосредственные исходные файлы) проекта-потомка. Примечательно, что какой-либо унаследованный исходный файл сохраняет любые опции, которые указаны в проекте-предке.

Например, если проект " Utilities" содержит спецификацию и тело Ада-пакета *Util_IO*, то проект " Modified_Utilities" может содержать новое тело для пакета *Util_IO*. Оригинальное тело *Util_IO* не будет рассматриваться при сборке программы. Однако, при этом будет использоваться спецификация этого пакета, которая расположена в проекте " Utilities".

Следует заметить, что проект-потомок может иметь только одного предка, но он может импортировать любое число других проектов. Кроме того, для какого-либо проекта не допускается, чтобы он одновременно импортировал (прямо или косвенно) проект-потомок и любой из его предков.

31.7 Обращение к внешним переменным в файлах проектов

Файл проекта может содержать обращения к внешним переменным. Такие обращения называют *внешними ссылками*.

Внешняя переменная может быть определена как часть среды окружения (например, какая-либо переменная окружения UNIX), или указана в командной строке с помощью опции "`textbf-X`*`vbl=value`*". При одновременном наличии переменной окружения и значения заданного в командной строке, будет использовано значение из командной строки.

Внешняя ссылка указывается с помощью встроенной функции "**external**", которая возвращает строковое значение. Эта функция имеет две формы:

```
external (имя_внешней_переменной)
external (имя_внешней_переменной, значение_по_умолчанию)
```

Каждый параметр должен быть строковым литералом. Например:

```
external ("USER")
external ("OS", "Linux")
```

В форме с одним параметром, функция возвращает значение внешней переменной, которая указана как параметр. Если в среде окружения такое имя отсутствует, возвращается пустая строка.

В форме с двумя аргументами, второй параметр является значением, которое возвращается когда переменная указанная как первый параметр отсутствует в среде окружения. В показанном выше примере, если "OS" не является именем переменной окружения и не указано в командной строке, то возвращаемым значением будет "Linux".

Внешняя ссылка может быть частью строкового выражения или выражения списка строк, используемого для описания переменных и/или атрибутов.

```
type Mode_Type is ("Debug", "Release");
Mode : Mode_Type := external ("MODE");
case Mode is
  when "Debug" =>
     ...
```

31.8 Пакеты файлов проектов

Пакет является свойством файла проекта с помощью которого обеспечивается возможность описания установок для индивидуальных инструментов, которые интегрированы со средствами управления проектами системы компилятора GNAT. Внутри файла проекта, для каждого такого инструмента можно описать соответствующий пакет (напомним, что имена пакетов строго предопределены). Пакет может содержать в себе описания переменных, атрибутов и конструкции выбора "**case**".

```
project Proj is
   package Builder is  -- используется утилитой gnatmake
      for Default_Switches ("Ada") use ("-v", "-g");
   end Builder;
end Proj;
```

Описание пакета начинается с зарезервированного слова "**package**", далее следует имя пакета (не зависит от регистра символов) сопровождаемое зарезервированным словом "**is**". Описание пакета завершается зарезервированным словом "**end**", которое сопровождается именем пакета и символом точки с запятой.

Большинство описываемых пакетов содержит атрибут "Default_Switches". Этот атрибут является ассоциативным массивом, а его значением является список строк. Индексом ассоциативного массива является имя языка программирования (не зависит от регистра символов). Этот атрибут указывает опцию или набор опций, которые будут использоваться соответствующим инструментальным средством.

Некоторые пакеты содержат также другой атрибут "Switches" - ассоциативный массив, значением которого является список строк. В этом случае индексом является имя исходного файла. Этот атрибут указывает опцию или набор опций, которые будут использоваться соответствующим инструментальным средством при обработке конкретного файла.

Пакет может быть описан путем *переименования* другого пакета, например, пакета из импортируемого файла проекта:

```
with "/global/apex.gpr";
project Example is
  package Naming renames Apex.Naming;
  ...
end Example;
```

Пакеты, которые переименовываются в других файлах проектов, часто описываются в файлах проектов, которые не обладают исходными файлами и используются в качестве шаблонов. Любая модификация такого шаблона будет автоматически отражена во всех файлах проектов, которые переименовывают пакеты шаблона.

В дополнение к пакетам, которые непосредственно ориентированы на соответствующие инструментальные средства, существует возможность описания пакета *Naming*, который позволяет установить необходимые соглашения по именованию исходных файлов.

31.9 Переменные импортируемых проектов

Какой-либо атрибут или переменная, которые были описаны в импортируемом проекте или проекте-предке, могут быть использованы в выражениях, которые используются в импортирующем или расширяющем проекте. В этом случае, для обращения к атрибуту или переменной используется префикс состоящий из имени проекта и, при необходимости, имени пакета, где атрибут или переменная были описаны.

```
with "imported";
project Main extends "base" is
   Var1 := Imported.Var;
   Var2 := Base.Var & ".new";

   package Builder is
      for Default_Switches ("Ada") use Imported.Builder.Ada_Switches &
                        "-gnatg" & "-v";
   end Builder;

   package Compiler is
      for Default_Switches ("Ada") use Base.Compiler.Ada_Switches;
   end Compiler;
end Main;
```

В показанном выше примере:

- Переменная "Var1" является копией переменной "**Var**", которая описана в файле проекта "`imported.gpr`".
- Значение переменной "Var2" является конкатенацией копии значения переменной "**Var**", которая описана в файле проекта "`base.gpr`", со строкой "`.new`".
- Атрибут "`Default_Switches ("Ada")`", в пакете *Builder*, является списком строк, который включает в свое значение копию переменной "Ada_Switches", описанную в пакете *Builder* в файле проекта "`imported.gpr`", плюс два новых элемента: "`textbf-gnatg`" и "`textbf-v`".
- Атрибут "`Default_Switches ("Ada")`", в пакете *Compiler*, является копией переменной "Ada_Switches", которая описана в пакете *Compiler* в файле расширяемого проекта "`base.gpr`".

31.10 Схемы именования файлов

Бывают случаи когда необходимо осуществить перенос программной системы, которая была разработана в среде какого-либо Ада-компилятора, в среду компилятора GNAT. При этом, имена файлов, которые использовались в среде другого Ада-компилятора, могут не соответствовать соглашениям по именованию файлов, которые стандартны и используются по умолчанию в среде GNAT. В такой ситуации, вместо переименования всех файлов, - что может быть практически неосуществимо по целому ряду причин, - в файле проекта, внутри пакета *Naming*, можно описать необходимую схему именования файлов. Например, показанное ниже описание пакета *Naming* моделирует соглашения по наименованию, которые традиционны для системы компилятора Apex:

```
package Naming is
  for Casing                         use "lowercase";
  for Dot_Replacement                use ".";
  for Specification_Suffix ("Ada")   use ".1.ada";
  for Implementation_Suffix ("Ada") use ".2.ada";
end Naming;
```

Внутри пакета *Naming* могут быть описаны следующие атрибуты:

Имя атрибута	**Описание**
Casing	Этот атрибут может принимать одно из трех значений: "`lowercase`", "`uppercase`" или "`mixedcase`" (все значения не зависят от используемого регистра символов). Когда значение этого атрибута не указано, по умолчанию, предполагается значение "`lowercase`".

Dot_Replacement

Значением этого атрибута может быть строка, которая удовлетворяет следующие условия:

- строка не должна быть пустой
- строка не может начинаться или заканчиваться буквой или цифрой
- строка не может состоять из одиночного символа подчеркивания
- строка не может начинаться одиночным символом подчеркивания сопровождаемым одиночной буквой или цифрой
- строка не может содержать символ точки (`"."`), за исключением случая когда вся эта строка состоит из одного символа точки (`"."`)

Когда значение этого атрибута не указано, по умолчанию, предполагается строка "–".

Specification_Suffix

Атрибут является ассоциативным массивом (индексируется именем языка программирования; не зависит от регистра символов), значением которого может быть строка, удовлетворяющая следующие условия:

- строка не должна быть пустой
- строка не может начинаться с символа буквы или цифры
- строка не может начинаться одиночным символом подчеркивания сопровождаемым одиночной буквой или цифрой

Когда значение атрибута "Specification_Suffix ("Ada")" не указано, по умолчанию, предполагается строка "`.ads`".

Implementation_Suffix

Атрибут является ассоциативным массивом (индексируется именем языка программирования; не зависит от регистра символов), значением которого может быть строка, удовлетворяющая следующие условия:

- строка не должна быть пустой
- строка не может начинаться с символа буквы или цифры
- строка не может начинаться одиночным символом подчеркивания сопровождаемым одиночной буквой или цифрой
- строка не может иметь такое же значение как "Specification_Suffix"

Когда значение атрибута "Implementation_Suffix ("Ada")" не указано, по умолчанию, предполагается строка "`.adb`".

Separate_Suffix

Значение этого атрибута должно удовлетворять те же условия, что и значение для атрибута "Implementation_Suffix". Когда значение атрибута "Separate_Suffix ("Ada")" не указано, по умолчанию, предполагается значение атрибута "Implementation_Suffix ("Ada")".

Specification

Этот атрибут является ассоциативным массивом и может быть использован для описания имен исходных файлов содержащих индивидуальные спецификации компилируемых модулей Ады. Индекс массива должен быть строковым литералом, который указывает модуль Ады (не зависит от регистра символов). Значением атрибута должна быть строка, которая указывает файл содержащий спецификацию компилируемого модуля (зависимость от регистра символов определяется используемой операционной системой).

```
for Specification ("MyPack.MyChild")
   use "mypack.mychild.spec";
```

Implementation

Этот атрибут является ассоциативным массивом и может быть использован для описания имен исходных файлов содержащих индивидуальные тела (возможно субмодули) компилируемых модулей Ады. Индекс массива должен быть строковым литералом, который указывает модуль Ады (не зависит от регистра символов). Значением атрибута должна быть строка, которая указывает файл содержащий тело компилируемого модуля (зависимость от регистра символов определяется используемой операционной системой).

```
for Implementation ("MyPack.MyChild")
   use "mypack.mychild.body";
```

31.11 Проекты библиотек

Проект библиотеки являются проектом объектный код которого помещается в какую-либо библиотеку (следует отметить, что в настоящий момент такая возможность не поддерживается для всех платформ).

Для создания проекта библиотеки, в файле проекта необходимо описать двухуровневые атрибуты: "Library_Name" и "Library_Dir". Дополнительно, можно описать атрибуты, которые относятся к библиотекам: "Library_Kind", "Library_Version" и "Library_Elaboration".

Атрибут "Library_Name" имеет строчное значение, которое должно начинаться с буквы и должно включать в себе только буквы и цифры.

Атрибут "Library_Dir" имеет строчное значение, которое обозначает путь к каталогу (абсолютный или относительный) где будет располагаться библиотека. Он должен обозначать уже существующий каталог, и этот каталог должен отличаться от каталога в котором будут сохраняться объектные файлы проекта. Кроме того, этот каталог должен быть доступен для записи.

В случае, когда оба атрибута, "Library_Name" и "Library_Dir", указаны и являются допустимыми, предполагается, что файл проекта описывает проект библиотеки. Только для таких файлов проектов будут осуществляться проверка дополнительных, необязательных атрибутов, которые относятся к библиотекам.

Атрибут "Library_Kind" является строкой, которая должна иметь одно из следующих значений (указанные значения не зависят от регистра символов): "`static`", "`dynamic`" или "`relocatable`". При отсутствии описания этого атрибута, предполагается, что библиотека является статической (то есть, значение по умолчанию — "`static`"). В противном случае, библиотека может быть динамической ("`dynamic`") или перемещаемой ("`relocatable`"). Следует заметить, что в зависимости от используемой операционной системы, различие между динамической и перемещаемой библиотекой может отсутствовать. Например, в UNIX такое различие отсутствует.

Атрибут "Library_Version" является строкой, интерпретация смысла которой зависит от платформы. В UNIX, это значение используется только для динамических/перемещаемых библиотек, как внутреннее имя библиотеки (известное как "`soname`"). Если имя файла библиотеки (построенное из "Library_Name") отличается от "Library_Version", то файл библиотеки будет символической ссылкой на фактический файл библиотеки, именем которого будет "Library_Version".

Пример файла проекта библиотеки (для UNIX):

```
project Plib is

   Version := "1";

   for Library_Dir       use "lib_dir";
   for Library_Name      use "dummy";
   for Library_Kind      use "relocatable";
   for Library_Version   use "libdummy.so." & Version;

end Plib;
```

Каталог "`lib_dir`" будет содержать внутренний файл библиотеки именем которого будет "`libdummy.so.1`", а "`libdummy.so`" будет символической ссылкой на "`libdummy.so.1`".

Когда утилита "gnatmake" обнаруживает, что файл проекта (не главный файл проекта) является файлом проекта библиотеки, она проверяет все непосредственные исходные файлы проекта и осуществляет пересборку библиотеки если любой из исходных файлов был перекомпилирован. Кроме того, все файлы "ALI" будут скопированы из каталога объектных файлов в каталог библиотеки. При этом, для сборки исполняемых файлов утилита "gnatmake"будет использовать библиотеку, а не индивидуальные объектные файлы.

31.12 Опции командной строки, относящиеся к файлам проектов

В системе компилятора GNAT, инструментальные средства, которые поддерживают файлы проектов, могут использовать в командной строке следующие опции:

Опция	Описание
-P*project*	Указывает имя файла проекта ("project"). Этот файл проекта будет анализироваться с учетом степени "многословности", которая может быть указана опцией "textbf-vP*x*", и используя внешние ссылки, которые могут быть указаны опциями "textbf-X". В командной строке может присутствовать только одна опция "textbf-P". Поскольку менеджер проектов GNAT начинает анализировать файла проекта только после проверки всех опций командной строки, порядок указания опций "textbf-P", "textbf-vP*x*" и/или "textbf-X" значения не имеет.

-X*name=value*"	Эта опция позволяет указать в командной строке, что переменная "*name*" имеет значение "*value*". Менеджер проектов GNAT будет использовать это значение в процессе анализа файла проекта, при обнаружении внешней ссылки "**external** (name)". Если "*name*" или "*value*" содержит пробелы, то в командной строке "*name=value*" должно быть помещено между символами кавычек:
	`-XOS=NT`
	`-X"user=John Doe"`
	Одновременно, в командной строке может быть указано несколько опций "textbf-X". Если в командной строке одновременно несколько опций "textbf-X" указывают значения для одной и той же переменной "*name*", то будет использовано последнее указанное значение. Значение внешней переменной, которое указано с помощью опции "textbf-X", более приоритетно по сравнению со значением переменной окружения с таким же именем "*name*".

-vP*x*"	Устанавливает уровень/степень "многословности" при анализе файла проекта. Указание "textbf-vP0" предполагается по умолчанию и подразумевает низкий уровень "многословности"(отсутствует вывод для синтаксически корректных файлов проектов); "textbf-vP1" — подразумевает средний уровень "многословности"; "textbf-vP2" — подразумевает высокий уровень "многословности". Если в командной строке одновременно указаны несколько опций "textbf-vP*x*", то будет использовано последнее указанное значение.

31.13 Инструментальные средства поддерживающие файлы проектов

31.13.1 Утилита gnatmake и файлы проектов

Рассмотрим два вопроса, которые касаются утилиты "gnatmake" и файлов проектов: описание опций для утилиты "gnatmake" и всех инструментальных средств, запуск которых она осуществляет; использование атрибута "Main".

Для каждого пакета: *Builder*, *Compiler*, *Binder* и *Linker*, - можно указать атрибуты "Default_Switches" и "Switches". Имена этих атрибутов подразумевают, что эти атрибуты определяют какие опции (*switches* — переключатели) и для каких файлов необходимо использовать как аргументы командной строки запуска

утилиты "gnatmake". Ниже будет показано, что эти ориентированные на пакеты опции предшествуют опциям, которые передаются в командную строку запуска утилиты "gnatmake".

Атрибут "Default_Switches" является ассоциативным массивом, который индексируется именем языка программирования (не зависимо от регистра символов) и возвращает значение в виде списка строк. Например:

```
package Compiler is
  for Default_Switches ("Ada") use ("-gnaty", "-v");
end Compiler;
```

Атрибут "Switches" также вляется ассоциативным массивом. Он индексируется именем файла (которое зависит или не зависит от регистра символов, в зависимости от используемой операционной системы) и возвращает значение в виде списка строк. Например:

```
package Builder is
   for Switches ("main1.adb") use ("-O2");
   for Switches ("main2.adb") use ("-g");
end Builder;
```

Для пакета *Builder*, имена файлов должны обозначать исходные файлы головных подпрограмм. Для пакетов *Binder* и *Linker*, имена файлов должны обозначать файлы "ALI" или исходные файлы головных подпрограмм. В каждом случае применимо только самостоятельное имя файла (без явного указания расширения имени файла).

Для каждого используемого в процессе сборки программы инструментального средства (утилита "gnatmake", компилятор, редактор связей и компоновщик), соответствующий ему пакет указывает набор опций, которые необходимо использовать при обработке этим инструментальным средством каждого файла. Такой набор используемых опций определяется с помощью описания внутри пакета соответствующих, ориентированных на указание опций атрибутов. В частности, опции, которые каждый такой пакет определяет для указанного файла F, включают:

- значение атрибута "Switches (F)", когда он указан в пакете для данного файла
- в противном случае, значение атрибута " Default_Switches ("Ada") ", когда он указан в пакете

Когда внутри пакета ни один их этих атрибутов не описан. пакет не указывает никаких опций для данного файла.

При запуске утилиты "gnatmake" для обработки заданного файла, весь набор опций командной строки запуска утилиты "gnatmake" состоит из двух множеств, располагаемых в следующем порядке:

1. набор опций, которые определены для этого файла в пакете *Builder*
2. набор опций, которые были переданы в командной строке

Когда, для обработки файла, утилита "gnatmake" осуществляет запуск какого-либо инструментального средства (компилятор, редактор связей, компоновщик) набор опций, которые передаютсяя инструментальному средству, состоит из трех множеств, располагаемых в следующем порядке:

1. набор опций, которые определены для этого файла в пакете *Builder* файла проекта, заданного в командной строке
2. набор опций, которые определены для этого файла внутри пакета соответствующего инструментального средства (в соответствующем файле проекта)
3. набор опций, которые были переданы в командной строке

Следует заметить, что опции, заданные в командной строке запуска утилиты "gnatmake", могут передаваться или не передаваться индивидуальному инструментальному средству, в зависимости от конкретной опции.

Утилита "gnatmake" может осуществлять запуск компилятора для обработки исходных файлов различных проектов. Менеджер проектов GNAT будет использовать соответствующий файл проекта, чтобы определить пакет *Compiler* для каждого исходного файла, который нуждается в компиляции. То же самое справедливо для пакетов *Binder* (редактор связей) и *Linker* (компоновщик).

В качестве примера, рассмотрим пакет *Compiler* в показанном ниже файле проекта:

```
project Proj1 is
   package Compiler is
      for Default_Switches ("Ada") use ("-g");
      for Switches ("a.adb") use ("-O1");
      for Switches ("b.adb") use ("-O2", "-gnaty");
   end Compiler;
end Proj1;
```

В этом случае, при запуске утилиты "`gnatmake`" с указанием этого файла проекта, когда необходимо осуществить компиляцию файлов "`a.adb`", "`b.adb`" и "`c.adb`", файл "`a.adb`" будет компилироваться с опцией "`textbf-O1`", файл "`b.adb`" будет компилироваться с опциями "`textbf-O2`" и "`textbf-gnaty`", а файл "`c.adb`" будет компилироваться с опцией "`textbf-g`".

Следующий пример демонстрирует упорядочивание опций, которые предоставлены различными пакетами:

```
project Proj2 is
   package Builder is
      for Switches ("main.adb") use ("-g", "-O1", "-f");
   end Builder;

   packager Compiler is
      for Switches ("main.adb") use ("-O2");
   end Compiler;
end Proj2;
```

При использовании команды:

```
gnatmake -PProj2 -O0 main
```

Компилятор будет запущен для компиляции "`main.adb`" с указанием следующей последовательности опций командной строки:

```
-g -O1 -O2 -O0
```

В этом случае, последняя опция "`textbf-O`" будет иметь приоритет перед всеми предыдущими опциями "`textbf-O`". Некоторые другие опции (например, такие как "`textbf-c`") будут добавлены неявно.

В рассматриваемом примере, опции "`textbf-g`" и "`textbf-O1`" обеспечиваются пакетом *Builder*, опция "`textbf-O2`" обеспечена пакетом *Compiler*, а опция "`textbf-O0`" была указана в командной строке.

Следует заметить, что опция "`textbf-g`" будет также передаваться при запуске "`gnatlink`".

Заключительный пример демонстрирует предоставление опций из пакетов, которые описаны в различных файлах проектов:

```
project Proj3 is
   for Source_Files use ("pack.ads", "pack.adb");
   package Compiler is
      for Default_Switches ("Ada") use ("-gnata");
   end Compiler;
end Proj3;

with "Proj3";
project Proj4 is
   for Source_Files use ("foo_main.adb", "bar_main.adb");
   package Builder is
      for Switches ("foo_main.adb") use ("-s", "-g");
   end Builder;
end Proj4;

-- Ada source file:
with Pack;
procedure Foo_Main is
   ...
end Foo_Main;
```

При использовании команды:

```
gnatmake -PProj4 foo_main.adb -cargs -gnato
```

Передаваемыми компилятору опциями, для компиляции "foo_main.adb", являются опция "textbf-g" (обеспечивается пакетом *Proj4.Builder*) и опция "textbf-gnato" (передается из командной строки). При компиляции импортируемого Ада-пакета *Pack*, будут использоваться опция "textbf-g" (из пакета *Proj4.Builder*), опция "textbf-gnata" (из пакета *Proj3.Compiler*) и опция "textbf-gnato" (передается из командной строки).

При использовании файла проекта, в командной строке запуска утилиты "gnatmake" можно указать несколько головных подпрограмм. Исходный файл каждой указываемой таким образом головной подпрограммы должен являться непосредственным исходным файлом проекта.

```
gnatmake -Pprj main1 main2 main3
```

Кроме того, при использовании файла проекта, запуск утилиты "gnatmake" может быть осуществлен без явного указания какой-либо головной подпрограммы. В этом случае, полученный результат будет зависесть от наличия описания атрибута "Main". Значением этого атрибута является список строк, каждый элемент которого является именем исходного файла (указание расширения имени файла является не обязательным), который содержит головную подпрограмму.

Когда атрибут "Main" описан в файле проекта как не пустой список строк и в командной строке запуска утилиты "gnatmake" не указана опция "textbf-u", запуск "gnatmake", - с указанием такого файла проекта, но без указания в командной строке какой-либо головной подпрограммы, - является эквивалентным запуску "gnatmake" с указанием в командной строке всех имен файлов, которые указываются атрибутом "Main". Например:

```
project Prj is
   for Main use ("main1", "main2", "main3");
end Prj;
```

Для показанного выше примера, команда "gnatmake -Pprj" будет эквивалентна команде "gnatmake -Pprj main1 main2 main3".

Когда файл проекта не содержит описание атрибута "Main", или этот атрибут описан как пустой список строк, или в командной строке указана опция "textbf-u", и запуск утилиты "gnatmake" осуществляется без указания в командной строке какой-либо головной подпрограммы, будет осуществлена проверка всех непосредственных исходных файлов проекта и, потенциально, они будут перекомпилированы. В зависимости от присутствия в командной строке опции "textbf-u", исходные файлы других проектов, от которых зависят непосредственные исходные файлы главного проекта, будут также проверены и, потенциально, перекомпилированы. Другими словами, опция "textbf-u" применяется ко всем непосредственным исходным файлам главного файла проекта.

31.13.2 Управляющая программа gnat (gnatcmd) и файлы проектов

Кроме утилиты "gnatmake", существуют другие инструментальные средства системы GNAT, которые ориентированы на обработку проектов. К таким инструментальным средствам относятся: "gnatbind", "gnatfind", "gnatlink", "gnatls" и "gnatxref". Следует однако заметить, что ни один из этих инструментов не может непосредственно использовать опцию указания файла проекта ("textbf-P"). Таким образом, необходимо осуществлять запуск этих инструментов с помощью управляющей программы "gnat" (или "gnatcmd").

Управляющая программа "gnat" обеспечивает внешний интерфейс инструментальных средств. Она принимает целый перечень команд и осуществляет вызов соответствующих инструментальных средств. Изначально, управляющая программа "gnat" была разработана для VMS, с целью преобразования квалификаторов стиля VMS в опции стиля UNIX. Однако, в настоящий момент эта программа доступна для всех платформ, которые поддерживает система компилятора GNAT.

На платформах отличных от VMS, управляющая программа "gnat" принимает следующие команды (вне зависимости от регистра символов):

- BIND для запуска "gnatbind"

- CHOP для запуска "gnatchop"
- COMP или COMPILE для запуска компилятора
- ELIM для запуска "gnatelim"
- FIND для запуска "gnatfind"
- KR или KRUNCH для запуска "gnatkr"
- LINK для запуска "gnatlink"
- LS или LIST для запуска "gnatls"
- MAKE для запуска "gnatmake"
- NAME для запуска "gnatname"
- PREP или PREPROCESS для запуска "gnatprep"
- PSTA или STANDARD для запуска "gnatpsta"
- STUB для запуска "gnatstub"
- XREF для запуска "gnatxref"

Следует заметить, что запуск компилятора осуществляется с помощью команды

```
gnatmake -f -u
```

Вслед за командой можно указать опции и аргументы, которые будут переданы соответствующему инструментальному средству:

```
gnat bind -C main.ali
gnat ls -a main
gnat chop foo.txt
```

Для команд: BIND, FIND, LS или LIST, LINK и XREF, — в дополнение к опциям, которые применимы для непосредственно вызываемого инструментального средства, могут быть использованы опции, которые характерны для файлов проектов ("**-P**", "**-X**" and "**-vP*x***").

Для каждой из этих команд, в главном проекте возможно существование пакета, который соответствует запускаемому инструментальному средству:

- "**package** Binder" — для команды BIND (запускает "gnatbind")
- "**package** Finder" — для команды FIND (запускает "gnatfind")
- "**package** Gnatls" — для команды LS или LIST (запускает "gnatls")
- "**package** Linker" — для команды LINK (запускает "gnatlink")
- "**package** Cross_Reference" — для команды XREF (запускает "gnatlink")

Пакет *Gnatls* обладает уникальным атрибутом "Switches", простая переменная со значением в виде списка строк. Атрибут содержит опции используемые при запуске "gnatls".

```
project Proj1 is
   package gnatls is
      for Switches use ("-a", "-v");
   end gnatls;
end Proj1;
```

Все остальные пакеты обладают атрибутом "Default_Switches", ассоциативный массив, который индексируется именем языка программирования (не зависит от регистра символов) и имеет значение в виде списка строк. Атрибут "Default_Switches ("Ada")" содержит опции, которые используются при запуске инструментального средства соответствующего пакету.

```
project Proj is

   for Source_Dirs use ("./**");

   package gnatls is
      for Switches use ("-a", "-v");
   end gnatls;

   package Binder is
      for Default_Switches ("Ada") use ("-C", "-e");
   end Binder;

   package Linker is
      for Default_Switches ("Ada") use ("-C");
   end Linker;

   package Finder is
      for Default_Switches ("Ada") use ("-a", "-f");
   end Finder;

   package Cross_Reference is
      for Default_Switches ("Ada") use ("-a", "-f", "-d", "-u");
   end Cross_Reference;
end Proj;
```

Для показанного выше файла проекта, команды:

```
gnat ls -Pproj main
gnat xref -Pproj main
gnat bind -Pproj main.ali
```

будут правильно устанавливать рабочее окружение и осуществлять запуск инструментальных средств используя опции, которые указаны в пакетах соответствующих инструментальным средствам.

31.14 Расширенный пример

Предположим, что у нас есть две программы "`prog1`" и "`prog2`", исходные файлы которых расположены в соответствующих каталогах. Предположим также, что нам необходимо осуществлять сборку этих программ с помощью одной команды запуска утилиты "`gnatmake`", и мы хотим сохранять объектные файлы этих программ в подкаталогах "`.build`" соответствующих каталогов с исходными файлами. Кроме того, нам необходимо иметь внутри каждого подкаталога "`.build`" два отдельных подкаталога "`release`" и "`debug`", которые будут содержать объектные файлы скомпилированные с различными наборами опций компиляции.

Более наглядно, мы имеем следующую структуру каталогов:

```
main
  |- prog1
  |    |- .build
  |         | debug
  |         | release
  |- prog2
       |- .build
            | debug
            | release
```

Рассмотрим файлы проектов, которые нам необходимо создать в каталоге "`main`", для сопровождения этой структуры:

1. Создадим проект "Common" с пакетом *Compiler*, который указывает опции компиляции:

Файл "common.gpr":

```
project Common is

   for Source_Dirs use (); -- нет исходных файлов

   type Build_Type is ("release", "debug");
   Build : Build_Type := External ("BUILD", "debug");
   package Compiler is
      case Build is
         when "release" =>
           for Default_Switches ("Ada") use ("-O2");
         when "debug"   =>
           for Default_Switches ("Ada") use ("-g");
      end case;
   end Compiler;

end Common;
```

2. Создадим отдельные проекты для двух программ:

Файл "prog1.gpr":

```
with "common";
project Prog1 is

    for Source_Dirs use ("prog1");
    for Object_Dir  use "prog1/.build/" & Common.Build;

    package Compiler renames Common.Compiler;

end Prog1;
```

Файл "prog2.gpr":

```
with "common";
project Prog2 is

    for Source_Dirs use ("prog2");
    for Object_Dir  use "prog2/.build/" & Common.Build;

    package Compiler renames Common.Compiler;

end Prog2;
```

3. Создадим проект-обертку "Main":

Файл "main.gpr":

```
with "common";
with "prog1";
with "prog2";
project Main is

   package Compiler renames Common.Compiler;

end Main;
```

4. В заключение, необходимо создать процедуру-заглушку, которая указывает в своем спецификаторе "**with**" (прямо или косвенно) все файлы с исходными текстами двух программ.

Теперь, с помощью команды

```
gnatmake -Pmain dummy
```

можно осуществить сборку программ для режима отладки ("Debug"), а с помощью команды

```
gnatmake -Pmain -XBUILD=release
```

можно осуществить сборку программ для режима реализации ("Release").

31.15 Диаграмма полного синтаксиса файлов проектов

В заключение обсуждения средств управления проектами системы компилятора GNAT, приведем диаграмму полного синтаксиса файлов проектов, которая непосредственно позаимствована из сопроводительной документации GNAT:

```
project ::=
  context_clause project_declaration

context_clause ::=
  {with_clause}

with_clause ::=
  with literal_string { , literal_string } ;

project_declaration ::=
  project <project_>simple_name [ extends literal_string ] is
    {declarative_item}
  end <project_>simple_name;

declarative_item ::=
  package_declaration |
  typed_string_declaration |
  other_declarative_item

package_declaration ::=
  package <package_>simple_name package_completion

package_completion ::=
  package_body | package_renaming

package body ::=
  is
    {other_declarative_item}
  end <package_>simple_name ;

package_renaming ::==
  renames <project_>simple_name.<package_>simple_name ;

typed_string_declaration ::=
  type <typed_string_>_simple_name is
   ( literal_string {, literal_string} );

other_declarative_item ::=
  attribute_declaration |
  typed_variable_declaration |
  variable_declaration |
  case_construction

attribute_declaration ::=
  for attribute use expression ;

attribute ::=
  <simple_attribute_>simple_name |
  <associative_array_attribute_>simple_name ( literal_string )

typed_variable_declaration ::=
  <typed_variable_>simple_name : <typed_string_>name :=  string_expression ;
```

```
variable_declaration ::=
  <variable_>simple_name := expression;

expression ::=
  term {& term}

term ::=
  literal_string |
  string_list |
  <variable_>name |
  external_value |
  attribute_reference

literal_string ::=
  (same as Ada)

string_list ::=
  ( <string_>expression { , <string_>expression } )

external_value ::=
  external ( literal_string [, literal_string] )

attribute_reference ::=
  attribute_parent ' <simple_attribute_>simple_name [ ( literal_string ) ]

attribute_parent ::=
  project |
  <project_or_package>simple_name |
  <project_>simple_name . <package_>simple_name

case_construction ::=
  case <typed_variable_>name is
    {case_item}
  end case ;

case_item ::=
  when discrete_choice_list => {case_construction | attribute_declaration}

discrete_choice_list ::=
  literal_string {| literal_string}

name ::=
  simple_name {. simple_name}

simple_name ::=
  identifier (same as Ada)
```

Глава 32

Построение больших проектов

Утилита "gnatmake", которая входит в состав системы компилятора GNAT, является прекрасным инструментом для построения небольших проектов. Однако при построении больших проектов или в случае каких-либо особенностей разрабатываемого проекта (например, при использовании большого количества внешних функций языка C) может возникнуть желание использовать для управления построением проекта каких-либо дополнительных средств.

Данная глава предоставляет некоторую информацию по совместному использованию с GNAT достаточно широко распространенной утилиты управления сборкой проектов GNU "make", а также пакетов GNU *Autoconf* и GNU *Automake*. Следует заметить, что здесь не преследуется цель описать все возможности и варианты использования этих программных инструментов GNU. Поэтому для получения более подробных сведений необходимо обратиться к соответствующей документации GNU. Кроме того, следует учитывать, что утилита GNU "make" не предназначена для замены утилиты "gnatmake", которая входит в систему компилятора GNAT.

32.1 Использование утилиты GNU make

Прежде чем рассматривать какие-либо примеры, следует заметить, что все приведенные далее примеры относятся непосредственно к утилите GNU "make". Хотя "make" является стандартной утилитой и базовый язык оформления файлов управления сборкой проекта одинаков, демонстрируемые примеры используют некоторые развитые свойства, которые характерны именно для версии GNU "make".

32.1.1 Общие сведения о GNU make

В процессе своей работы утилита GNU "make" осуществляет интерпретацию набора правил, которые сохранены в файле управления сборкой проекта с именем "Makefile". Правила описывают зависимость одних файлов от других файлов. Каждое правило сопровождается командой (или последовательностью команд), которую необходимо выполнить для обновления файлов (например, команда может выполнять компиляцию файлов). Кроме того, файлы управления сборкой проекта ("Makefile") могут содержать в себе комментарии, переменные и правила, которые ссылаются на параметры команды "make".

Предположим, что Ада-программа называется "dbase" и в ее состав входят следующие файлы с исходными текстами: "common.adb", "scanner.adb" и "parser.adb". В этом случае файл "Makefile" может содержать в себе следующее правило:

```
dbase: common.o scanner.o parser.o
        gnatlink -o dbase
```

Это правило говорит о том, что исполняемый файл "dbase" зависит от трех файлов с исходными текстами на языке Ада, и чтобы выполненить обновление файла "dbase" программа "make" должна заново скомпоновать его из указанных объектных файлов с помощью команды "gnatlink".

Следует заметить, что при написании Ада-программ, которые использует файлы с исходными текстами на языке C основополагающая стратегия использования утилиты GNU "make" с компилятором GNAT заключается в создании правил, которые гарантируют правильную компиляцию файлов с исходными текстами на языке C, а затем завершают построение всего проекта с помощью выполнения команды "gnatmake".

Рассмотрим пример простого файла управления сборкой проекта ("Makefile"), который будет осуществлять компиляцию Ада-программы "main.adb", а также любых Ада-пакетов, которые используются в программе "main.adb". Этот пример будет работать со всеми небольшими проектами. Для этого понадобиться отредактировать содержимое переменной "OBJS", которая содержит перечень всех объектных файлов соответствующих Ада-пакетов используемых в реальной программе.

Для использования этого файла управления сборкой проекта необходимо просто выполнить команду "make", которая осуществит автоматическую сборку проекта, или выполнить команду "make clean" для удаления всех промежуточных файлов, которые были сгенерированы компилятором.

```
# Пример простого файла управления сборкой проекта Makefile
#
# Предполагается, что головная программа имеет имя "main.adb"
#

OBJS = main.o somepackage.o

# Правило компиляции файлов с исходными тектами на языке Ада
.adb.o:
        gcc -c $<

.SUFFIXES: .adb .o

# Правило компоновки головной программы
main: $(OBJS)
        gnatbind -xf main.ali; gnatlink main.ali

clean:
        rm *.o *.ali core
```

32.1.2 Использование утилиты gnatmake в файлах Makefile

Управление построением сложного проекта удобно осуществлять с помощью комбинированного использования утилит GNU "make" и "gnatmake". Например, возможно существование файла управления сборкой проекта "Makefile", который позволяет построить каждую отдельную подсистему большого проекта как динамически загружаемую библиотеку. Подобный файл управления сборкой проекта позволяет значительно уменьшить время компоновки большого приложения при сопровождении всех взаимосвязей на каждом этапе процесса сборки проекта.

В таком случае, первичные списки зависимостей автоматически обрабатываются с помощью утилиты "gnatmake", а файл управления сборкой проекта "Makefile" используется для звпуска "gnatmake" в каждом подкаталоге проекта.

```
## Этот Makefile предназначен для использования при следующей структуре
## каталогов:
##  - Исходные тексты разделены на серию компонентов программного обеспечения
##    csc (computer software components)
##    Каждый из этих csc помещается в своем собственном каталоге.
##    Их имена соответствуют названиям каталогов.
##    Они будут компилироваться в динамически загружаемые библиотеки
##    (хотя они могут работать и при статической компоновке)
##  - Головная программа (и возможно другие пакеты, которые не являются
##    частями каких-либо csc, располагаются в каталоге верхнего уровня
##    (там где располагается Makefile).
##       toplevel_dir __ first_csc  (sources) __ lib (will contain the library)
##                    _ second_csc (sources) __ lib (will contain the library)
##                    _ ...
##
## Хотя этот Makefile предназначен для построения динамически загружаемых
## библиотек, его достаточно легко модифицировать для построения
## частично скомпонованых объектов (необходимо модифицировать показанные ниже
## строки с -shared и gnatlink)
##
## При использовании этого Makefile, можно изменить любой существующий файл
```

```
## системы или добавить какой-либо новый файл, и все будет корректно
## перекомпилировано (фактически, будут перекомпилированы только объекты,
## которые нуждаются в перекомпиляции и будет осуществлена перекомпоновка
## головной программы).
##

# Список компонентов программного обеспечения csc проекта.
# Он может быть сгенерирован автоматически.
CSC_LIST=aa bb cc

# Имя головной программы (без расширения)
MAIN=main

# Для построения объектов с опцией -fPIC
# необходимо раскомментировать следующую строку
#NEED_FPIC=-fPIC

# Следующая переменная должна указывать каталог
# в котором расположена библиотека libgnat.so
# Узнать расположение libgnat.so можно с помощью команды
# 'gnatls -v'. Обычно, это последний каталог в Object_Path.
GLIB=...

# Каталоги для библиотек. Этот макрос расширяется в список CSC,
# который перечисляет динамически загружаемые библиотеки.
# Возможно использование простого перечисления:
#   LIB_DIR=aa/lib/libaa.so bb/lib/libbb.so cc/lib/libcc.so
LIB_DIR=$foreach dir,$CSC_LIST,$dir/lib/lib$dir.so

$MAIN: objects $LIB_DIR
        gnatbind $MAIN $CSC_LIST:%=-aO%/lib -shared
        gnatlink $MAIN $CSC_LIST:%=-l%

objects::
        # перекомпиляция исходных текстов
        gnatmake -c -i $MAIN.adb $NEED_FPIC $CSC_LIST:%=-I%

# Примечание: в будущих версиях GNAT следующие команды будут упрощены
#             с помощью использования нового инструментального средства
#             gnatmlib
$LIB_DIR:
        mkdir -p $dir $
        cd $dir $ ; gnatgcc -shared -o $notdir $  ../*.o -L$GLIB -lgnat
        cd $dir $ ; cp -f ../*.ali .

# Зависимости для модулей
aa/lib/libaa.so: aa/*.o
bb/lib/libbb.so: bb/*.o
bb/lib/libcc.so: cc/*.o

# Перед запуском программы необходимо убедиться в том,
# что динамически загружаемые библиотеки указаны в пути поиска
run::
        LD_LIBRARY_PATH=pwd/aa/lib:pwd/bb/lib:pwd/cc/lib ./$MAIN

clean::
        $RM -rf $CSC_LIST:%=%/lib
        $RM $CSC_LIST:%=%/*.ali
        $RM $CSC_LIST:%=%/*.o
        $RM *.o *.ali $MAIN
```

32.1.3 Автоматическое создание списка каталогов

Для большинства файлов "Makefile", возникает необходимость перечисления списка каталогов и сохранения этого списка в какой-либо переменной. Для небольших проектов чаще всего такое перечисление проще выполнять вручную, поскольку после этого сохраняется полное управление над правильностью порядка перечисления этих каталогов. Однако, для больших проектов, которые используют сотни подкаталогов, автоматическая генерация списка каталогов может оказаться предпочтительнее.

Показанный ниже пример файла "Makefile" демонстрирует использование двух различных способов автоматического создания списка каталогов. Первый способ, который является менее общим, обеспечивает более полное управление над списком каталогов. Он использует шаблонные символы, которые будут автоматически расширены утилитой GNU "make". Недостатком этого способа является то, что необходимо более явное указание организации структуры проекта, например, указание глубины вложенности иерархии каталогов проекта, или указание того, что проект расположен в различных иерархиях каталогов.

Второй способ — более общий. Он требует использования внешней программы "find", которая стандартна для большинства систем UNIX. Все подкаталоги, которые расположены внутри корневого каталога проекта, будут добавлены в список каталогов проекта.

```
# Показанный ниже пример основывается на следующей иерархии каталогов,
# причем, все каталоги могут содержать произвольное количество файлов:
# ROOT_DIRECTORY ->  a  ->  aa  ->  aaa
#                       ->  ab
#                       ->  ac
#                ->  b  ->  ba  ->  baa
#                       ->  bb
#                       ->  bc
# Этот файл Makefile создает переменную с именем DIRS, которая может
# быть использована в любой момент, когда необходим список каталогов
# (см. другие примеры)

# Корневой каталог иерархии каталогов проекта
ROOT_DIRECTORY=.

####
# Первый способ: явно определяет список каталогов.
# Он позволяет определять любое подмножество необходимых каталогов.
####

DIRS := a/aa/ a/ab/ b/ba/

####
# Второй способ: использует шаблоны
# Примечательно, что аргументы показанных ниже шаблонов
# должны звканчиваться символом '/'.
# Поскольку шаблоны также возвращают имена файлов, их необходимо
# отфильтровывать, чтобы избежать дублирование имен каталогов.
# Для этого используются встроенные функции make dir и sort.
# Это устанавливает переменную DIRS в следующее значение (примечательно, что
# каталоги aaa и baa не будут указаны до тех пор, пока не будут
# изменены аргументы шаблона).
# DIRS= ./a/a/ ./b/ ./a/aa/ ./a/ab/ ./a/ac/ ./b/ba/ ./b/bb/ ./b/bc/
####

DIRS := $sort $dir $wildcard $ROOT_DIRECTORY/*/ $ROOT_DIRECTORY/*/*/

####
# Третий способ: использует внешнюю программу
# Эта команда будет более быстро выполняться на локальных дисках.
# В результате выполнения этой команды переменная DIIRS будет
# установлена в следующее значение:
# DIRS= ./a ./a/aa ./a/aa/aaa ./a/ab ./a/ac ./b ./b/ba ./b/ba/baa ./b/bb ./b/bc
####

DIRS := $shell find $ROOT_DIRECTORY -type d -print
```

32.1.4 Генерация опций командной строки для gnatmake

После создания списка каталогов, как было описано выше, можно легко сгенерировать аргументы командной строки, которые будут переданы утилите "gnatmake".

С целью полноты этот пример подразумевает, что путь к исходным текстам не соответствует пути к объектным файлам. Таким образом, имеется два различных списка каталогов.

```
# см. "Автоматическое создание списка каталогов"
# для создания этих переменных
SOURCE_DIRS=
OBJECT_DIRS=

GNATMAKE_SWITCHES := $patsubst %,-aI%,$SOURCE_DIRS
GNATMAKE_SWITCHES += $patsubst %,-aO%,$OBJECT_DIRS

all:
        gnatmake $GNATMAKE_SWITCHES main_unit
```

32.1.5 Преодоление ограничения на длину командной строки

Многие операционные системы ограничивают длину командной строки. Это может вызвать некоторые трудности при работе над большим проектом, поскольку затрудняет передачу списков каталогов, которые содержат файлы с исходными текстами и объектные файлы, в командной строке утилиты "gnatmake".

Показанный ниже пример демонстрирует то, как можно установить переменные окружения, которые позволят утилите "gnatmake" выполнять обработку также как и в случае указания списков каталогов в командной строке запуска "gnatmake". При этом ограничения операционной системы позволяют использовать более длинные списки каталогов или, как в большинстве систем, длина списка будет не ограничена.

Подразумевается, что для создания списка каталогов в файле "Makefile" используются ранее рассмотренные способы. С целью полноты примера предполагается, что путь к объектным файлам (где также располагаются ALI-файлы) не соответствует пути к файлам с исходными текстами.

Следует обратить внимание на небольшую хитрость, которая использована в файле "Makefile": для повышения эффективности создаются две временные переменные ("SOURCE_LIST" и "OBJECT_LIST"), которые немедленно расширяются утилитой GNU "make". Примечательно, что этот способ перекрывает стандартное поведение утилиты "make", когда переменные расширяются только в момент их фактического использования.

```
# В этом примере создаются две переменные окружения
# ADA_INCLUDE_PATH и ADA_OBJECT_PATH.
# Они обладают таким же эффектом как и опции -I в командной строке.
# Эквивалентом использования -aI в командной строке будет
# только указание переменной окружения ADA_INCLUDE_PATH,
# а эквивалентом использования -aO является ADA_OBJECT_PATH.
# Естественно, что для этих переменных необходимо использование
# двух различных значений.
#
# Примечательно также, что необходимо сохранение предыдущих значений
# этих переменных, поскольку они могут быть определены перед запуском
# 'make' с целью указания установленных библиотек для GNAT.


# см. "Автоматическое создание списка каталогов"
# для создания этих переменных
SOURCE_DIRS=
OBJECT_DIRS=

empty:=
space:=$empty $empty
SOURCE_LIST := $subst $space,:,$SOURCE_DIRS
OBJECT_LIST := $subst $space,:,$OBJECT_DIRS
ADA_INCLUDE_PATH += $SOURCE_LIST
```

```
ADA_OBJECT_PATH += $OBJECT_LIST
export ADA_INCLUDE_PATH
export ADA_OBJECT_PATH

all:
        gnatmake main_unit
```

32.2 Переносимость в UNIX, пакеты GNU *Automake* и GNU *Autoconf*

Если вы используете систему Linux и желаете создать проект, который будет выполняться на различных платформах UNIX, а не только в Linux, то вам необходимо использовать пакеты GNU *Autoconf* и GNU *Automake*. Следует заметить, что свободно распространяемое программное обеспечение GNU использует средства этих пакетов очень интенсивно.

Поставляемый в системе Linux пакет GNU *Autoconf* позволяет создать скрипт командного интерпретатора с именем "`configure`", который настроен под конкретный проект. Когда этот скрипт выполняется, он осуществляет сканирование средств той системы UNIX на которой он выполняется. В результате, этот скрипт осуществляет необходимую подстройку файлов "`Makefile`", которые управляют сборкой проекта. В качестве необязательного дополнения он может генерировать C-файл "`config.h`", который содержит информацию об обнаруженных средствах.

Пакет GNU *Automake* является дополнением к пакету GNU *Autoconf*. С помощью утилиты "`automake`" он позволяет создавать файлы "`Makefile.in`" из шаблонных файлов "`Makefile.am`". Как только работа утилиты "`automake`" завершена, для получения окончательных версий файлов "`Makefile`" необходимо запустить скрипт "`configure`". После этого остается только выполнить команду "`make`", которая построит проект для любой версии UNIX.

Существует возможность использования средств пакетов GNU *Autoconf* и GNU *Automake* с файлами "`Makefile`", которые используются для управления сборкой проектов программ на языке Ада. Для получения подробных сведений об использовании этих пакетов следует обратиться к соответствующей документации или использовать команды "`info autoconf`" и "`info automake`".

Глава 33

Использование встроенного ассемблера

Использование встроенного ассемблера вместе с компилятором GNAT в реальности не является сложностью и позволяет генерировать очень эффективный код. Однако, для того кто использовал средства встроенного ассемблера с другими языками программирования и компиляторами, использование встроенного ассемблера вместе с компилятором GNAT может сначала показаться несколько мистическим.

Прежде чем продолжить обсуждение этой темы, необходимо сделать несколько замечаний:

- Предполагается, что вы уже знакомы с программирование на языке Ада и имеете представление о том как писать программы на ассемблере.
- Основное внимание уделяется использованию встроенного ассемблера для процессоров семейства Intel x86, однако рассмотренные принципы могут пригодиться для использования встроенного ассемблера с другими типами процессоров.

33.1 Общие сведения

33.1.1 Пакет *System.Machine_Code*

Ада-программист не часто нуждается в непосредственном использовании ассемблера. Однако, когда возникает ситуация при которой необходимо использовать ассемблер, тогда действительно необходимо использовать ассемблер. Для таких случаев Ада предоставляет пакет *System.Machine_Code* который позволяет использовать ассемблер внутри Ада-программы. Поскольку разработчики стандарта Ады не имели возможности предусмотреть каких-либо разумных средств, которые обязан предоставлять этот пакет, то ответственность за реальную реализацию этих средств полностью возложена на разработчиков конкретного компилятора, которые способны адаптировать свой компилятор под то окружение в котором компилятор будет использоваться. Следовательно, здесь основное внимание сосредоточено на реализации этого пакета для компилятора GNAT.

33.1.2 Различия в использовании внешнего и встроенного ассемблера

Справедливо заметить, что всегда существует возможность скомпоновать программу на языке Ада с внешней подпрограммой на ассемблере. Однако, для небольших фрагментов ассемблерного кода, использование встроенного ассемблера обладает несколькими отличительными преимуществами:

- нет необходимости в использовании дополнительных не Ада-средств
- автоматическое использование соответствующих соглашений по вызову подпрограмм
- более простой доступ к описанным в Аде константам и переменным
- возможность написания внутренних (*intrinsic*) подпрограмм
- возможность использования кода встроенного ассемблера для встроенной подстановки (*inline*)
- оптимизатор кода может учитывать использование кода встроенного ассемблера

Таким образом, в случаях когда нет необходимости писать большое количество ассемблерного кода, удобнее использовать средства встроенного ассемблера.

33.1.3 Особенности реализации компилятора GNAT

Поскольку компилятор GNAT является частью семейства компиляторов GCC, то он использует средства встроенного ассемблера, которые характерны для этого семейства компиляторов. В следствие этого, реализация GNAT обладает следующими основными преимуществами:

- базовый интерфейс хорошо согласован с множеством различных целевых платформ
- обеспечено хорошее взаимодействие с оптимизатором компилятора, что позволяет генерировать очень эффективный код

В результате, возможность отображения естественных средств GCC на Ada95 выглядит достаточно впечатляющей.

33.2 Особенности используемого ассемблера

Для тех кто мигрирует с платформы PC/MS, вид ассемблера, используемого в GNAT, может сначала показаться обескураживающим. Причина в том, что ассемблер, который используется семейством компиляторов GCC, не использует привычный язык ассемблера Intel, а использует язык ассемблера, который происходит от ассемблера "`as`" системы AT&T Unix (его часто называют ассемблером с синтаксисом AT&T). Таким образом, даже человек, который знаком с языком ассемблера Intel, будет вынужден потратить некоторое время на изучение нового языка ассемблера, перед тем как использовать ассемблер с GNAT.

33.2.1 Именование регистров процессора

При использовании соглашений Intel, именование регистров процессора осуществляется следующим образом:

```
mov eax, 4
```

В случае синтаксиса AT&T, перед именем регистра процессора должен располагаться символ процента "`%`":

```
mov %eax, 4
```

33.2.2 Порядок следования операндов источника и приемника

В действительности, приведенный выше пример не будет успешно ассемблироваться с GNAT. В то время как соглашения Intel устанавливают порядок следования операндов источника и приемника как:

```
mov приемник, источник
```

синтаксис AT&T использует обратный порядок:

```
mov источник, приемник
```

Таким образом пример должен быть переписан:

```
mov 4, %eax
```

33.2.3 Значения констант

Это может показаться странным, но попытка ассемблирования показанного ранее кода также окажется безуспешной. Причина в том, что синтаксис AT&T подразумевает, что перед непосредственными статическими константными значениями необходимо помещать символ доллара "`$`":

```
mov $4, %eax
```

В результате, этот код выполнит загрузку регистра "`eax`" значением "`4`". Такая же нотация используется когда необходимо загрузить в регистр какое-либо адресное значение. Например, можно написать следующее:

```
mov $my_var, %eax
```

для загрузки адреса переменной "`my_var`" в регистр "`eax`".

33.2.4 Шестнадцатеричные значения

При использовании соглашений Intel, для загрузки в регистр шестнадцатеричного значения можно было написать:

```
mov eax, 1EAh
```

Увы, в этом случае синтаксис AT&T также отличается! Для загрузки в регистр шестнадцатеричного значения необходимо использовать соглашения языка C, то есть, шестнадцатеричное значение должно иметь префикс "**0x**", причем, даже при использовании встроенного ассемблера Ады. Таким образом, показанный выше пример должен быть переписан следующим образом:

```
mov $0x1EA, %eax
```

где сама константа не зависит от используемого регистра символов.

33.2.5 Суффиксы размера

Ассемблер Intel пытается определить размер инструкции пересылки анализируя размеры операндов. Таким образом, код:

```
mov ax, 10
```

будет пересылать 16-битное слово в регистр "**ax**". Ассемблер "**as**", как правило, отказывается "играть в догадки". Вместо этого, при написании имени инструкции необходимо использовать суффикс явной символьной индикации размеров операндов:

```
movw $10, %ax
```

где допустимыми символами являются **b**, **w** и **l**:

b - byte (8-бит) [`movb $10, %al`]
w - word (16-бит) [`movw $10, %ax`]
l - long (32-бит) [`movl $10, %eax`]

Можно встретить и другие случаи использования модификаторов размера, подобно тому как "**pushfd**" становиться "**pushfl**". Чаще всего использование модификаторов размера очевидно, а в случае ошибки - ассемблер обязательно подскажет.

33.2.6 Загрузка содержимого памяти

Ранее мы рассмотрели инструкцию вида:

```
movl $my_var, %eax
```

которая позволяет загрузить регистр "**eax**" значением адреса переменной "**my_var**". А как можно загрузить в регистр "**eax**" значение содержимого переменной "**my_var**"? Используя привычный синтаксис Intel можно написать:

```
mov eax, [my_var]
```

Синтаксис AT&T потребует изменить это на:

```
movl my_var, %eax
```

33.2.7 Косвенная адресация

Теперь нам известно как загрузить значение адреса и содержимое памяти. Но как загрузить содержимое памяти на которое указывает адрес расположенный в памяти или регистре? Синтаксис AT&T не использует средства подобные "`BYTE PTR`", которые традиционны для ассемблера Intel. Для загрузки регистра "`ebx`" 32-битным содержимым памяти адрес которого храниться в регистре "`eax`" необходимо написать:

```
movl (%eax), %ebx
```

Для добавления смещения к адресу в регистре "`eax`", необходимо использовать это смещение как префикс. Например, для получения содержимого 4-мя байтами ниже адреса, указанного в регистре "`eax`", следует написать:

```
movl -4(%eax), %ebx
```

или использовать содержимое памяти подобным образом:

```
movl my_var(%eax), %ebx
```

Существует большое количество возможных схем адресации (при необходимости, лучше обратиться к руководству по "`as`") однако того, что перечислено, как правило, достаточно для использования встроенного ассемблера.

33.2.8 Инструкции повторения

При использовании синтаксиса Intel, инструкции повторения записывались в одной строке кода. Так, для пересылки строки, можно было написать:

```
rep stos
```

В случае синтаксиса AT&T, эти инструкции должны быть записаны в отдельных строчках ассемблерного кода:

```
rep
stosl
```

33.3 Использование пакета *System.Machine_Code*

После рассмотрения общих правил написания инструкций ассемблера, можно сконцентрировать внимание на интеграции кода ассемблера с Ада-программой GNAT.

33.3.1 Пример элементарной программы

Пожалуй, более элементарную программу придумать не возможно:

```
with System.Machine_Code; use System.Machine_Code;

procedure Nothing is
begin
    Asm ("nop");
end Nothing;
```

Не составит труда догадаться, что эта программа состоит из единственной инструкции ассемблера "**nop**", которая действительно ничего не делает.

33.3.2 Проверка примера элементарной программы

Для исследования показанного примера элементарной программы, который демонстрирует использование встроенного ассемблера, можно воспользоваться способностью GNAT генерировать ассемблерный листинг программы. Это позволяет убедиться в том, что мы реально получаем то, что и ожидаем получить. При последующем рассмотрении более сложных примеров использования встроенного ассемблера и Ады, а также при самостоятельном использовании встроенного ассемблера такая способность GNAT будет оценена по достоинству. Следует также заметить, что анализ генерируемого ассемблерного листинга может оказаться более ефективным способом отладки программы чем компиляция и запуск отладчика. Для того, чтобы получить ассемблерный листинг программы необходимо выполнить следующую команду:

```
gcc -c -S -fomit-frame-pointer -gnatp 'nothing.adb'
```

Поясним значение используемых в этой команде опций:

-c — выполнить только компиляцию исходного текста

-S — сгенерировать листинг ассемблера

-fomit-frame-pointer — не устанавливать отдельные кадры стека

-gnatp — не добавлять код проверки времени выполнения

Выполнение команды предоставляет хорошо читаемую версию ассемблерного кода. Результирующий файл будет иметь такое же базовое имя, а расширение имени файла будет "`.s`". В случае файла с исходным текстом "`nothing.adb`", результирующий файл с ассемблерным листингом будет "`nothing.s`". Содержимое этого файла будет иметь следующий вид:

```
.file "nothing.adb"
gcc2_compiled.:
___gnu_compiled_ada:
.text
   .align 4
.globl __ada_nothing
__ada_nothing:
#APP
   nop
#NO_APP
   jmp L1
   .align 2,0x90
L1:
   ret
```

Примечательно, что ассемблерный код, который непосредственно был введен в исходный текст примера программы, помещен между метками "`#APP`" и "`#NO_APP`".

33.3.3 Поиск ошибок в коде ассемблера

Часто, при совершении ошибки в ассемблерном коде (подобной использованию не правильного модификатора размера или оператора для какой-либо инструкции) GNAT сообщает о такой ошибке во временном файле, который уничтожается после завершения процесса компиляции. В подобных ситуациях, генерация ассемблерного файла, показанная в примере выше, может оказаться не заменимым отладочным средством, поскольку полученный ассемблерный файл может быть самостоятельно ассемблирован с помощью ассемблера "`as`" (используемый в системе или поставляемый вместе с GNAT). Так, для ассемблирования файла "`nothing.s`", полученного ранее с помощью GNAT, можно использовать следующую команду:

```
as nothing.s
```

Сообщения об ошибках ассемблирования будут указывать строки в соответствующем ассемблерном файле. В результате, такая информация предоставляет возможность достаточно легко обнаружить и исправить ошибки, которые были допущены в первоначальном исходном тексте.

33.3.4 Более реальный пример

Более реальным и интересным примером может служить пример, который позволет увидеть содержимое регистра флагов процессора:

```
with Interfaces;                  use Interfaces;
with Ada.Text_IO;                 use Ada.Text_IO;
with System.Machine_Code;         use System.Machine_Code;
with Ada.Characters.Latin_1;      use Ada.Characters.Latin_1;

procedure Getflags is
   Eax : Unsigned_32;
begin
   Asm ("pushfl"          & LF & HT & -- сохранить регистр флагов в стеке
        "popl %%eax"      & LF & HT & -- загрузить флаги из стека в регистр eax
        "movl %%eax, %0",             -- сохранить значение флагов в переменной
        Outputs => Unsigned_32'Asm_Output ("=g", Eax));
   Put_Line ("Flags register:" & Eax'Img);
end Getflags;
```

Текст этого примера может неожиданно показаться несколько сложным. Однако это не должно вызывать большие опасения, поскольку далее будут рассмотрены все тонкости этого примера. Следует заметить, что понимание текста этого примера позволит использовать всю мощь встроенного ассемблера GNAT.

Первое, на что необходимо обратить внимание - это способ записи множества инструкций ассемблера. Реально, следующий пример записи множества инструкций ассемблера будет полностью корректным:

```
Asm ("pushfl popl %%eax movl %%eax, %0")
```

В генерируемом ассемблерном файле этот фрагмент будет отображен следующим образом:

```
#APP
   pushfl popl %eax movl %eax, -40(%ebp)
#NO_APP
```

Не трудно заметить, что такой результат не очень легко и удобно читать.

Таким образом, удобнее записывать различные инструкции ассемблера в отдельных строчках исходного текста, заключая их в двойные кавычки, а с помощью стандартного пакета *Ada.Characters.Latin_1* выполнять "ручную"вставку символов перевода строки (*LineFeed* / "LF") и горизонтальной табуляции (*Horizontal Tab* / "HT"). Это позволяет сделать более читабельным как исходный текст Ады:

```
Asm ("pushfl"          & LF & HT & -- сохранить регистр флагов в стеке
     "popl %%eax"      & LF & HT & -- загрузить флаги из стека в регистр eax
     "movl %%eax, %0")             -- сохранить значение флагов в переменной
```

так и генерируемый файл ассемблера:

```
#APP
   pushfl
   popl %eax
   movl %eax, -40(%ebp)
#NO_APP
```

Однако, стоит обратить внимание на тот факт, что Ада полностью игнорирует комментарии в конце строки при записи реальных инструкций ассемблера в генерируемый ассемблерный файл.

Настоятельно рекомендуется применять данный способ для оформления исходных текстов с использованием встроенного ассемблера, поскольку далеко не все (включая и автора, спустя несколько месяцев) будут способны легко понять смысл фрагмента исходного текста на ассемблере!

При более внимательном рассмотрении исходного текста примера можно обнаружить примечательный факт наличия двух символов процента перед каждым именем регистра, хотя в генерируемом ассемблерном файле перед именами регистров присутствует только один символ процента. Причина этого заключается в том, что при написании инструкций встроенного ассемблера, всем именам переменных и именам регистров должен предшествовать символ процента "%". Следовательно, поскольку ассемблер сам требует использование

символа процента перед именем регистра, то мы получаем следующую инструкцию встроенного ассемблера для пересылки содержимого из регистра "ax" в регистр "bx":

```
Asm ("movw %%ax, %%bx");
```

Это будет отображено в сгенерированном ассемблерном файле как:

```
#APP
   movw %ax, %bx
#NO_APP
```

Фактическое использование символа процента в инструкциях ассемблера необходимо для введения операнда более общего вида:

```
Asm ("pushfl"          & LF & HT & -- сохранить регистр флагов в стеке
     "popl %%eax"      & LF & HT & -- загрузить флаги из стека в регистр eax
     "movl %%eax, %0",            -- сохранить значение флагов в переменной
     Outputs => Unsigned_32'Asm_Output ("=g", Eax));
```

Здесь, "%0", "%1", "%2"... индицируют операнды которые позже описывают использование параметров ввода (*Input*) и вывода (*Output*).

33.3.5 Параметры вывода

При желании сохранить содержиме регистра "eax" в какой-либо переменной, первым желанием будет написать что-нибудь подобное следующему:

```
procedure Getflags is
   Eax : Unsigned_32;
begin
   Asm ("pushfl"          & LF & HT & -- сохранить регистр флагов в стеке
        "popl %%eax"      & LF & HT & -- загрузить флаги из стека в регистр eax
        "movl %%eax, Eax")           -- сохранить значение флагов в переменной
   Put_Line ("Flags register:" & Eax'Img);
end Getflags;
```

с попыткой сразу сохранить содержимое регистра процессора "eax" в переменной "Eax". Увы, также просто как это может выглядеть, это не будет работать, поскольку нельзя точно сказать чем будет являться переменная "Eax". Несколько подумав, можно сказать: "Eax" - это локальная переменная, которая, в обычной ситуации, будет размещена в стеке. Однако, в результате оптимизации, компилятор, для хранения этой переменной во время выполнения подпрограммы, может использовать не пространство в стеке, а обычный регистр процессора. Таким образом, возникает законный вопрос: как необходимо специфицировать переменную, чтобы обеспечить корректную работу для обоих случаев? Ответ на этот вопрос заключается в том, чтобы не сохранять результат в переменной "Eax" самостоятельно, а предоставить компилятору возможность самостоятельно выбирать правильный операнд для использования. Для этой цели применяется понятие параметра вывода.

Программа, использующая инструкцию вывода, будет выглядеть так как она была написана ранее:

```
Asm ("pushfl"          & LF & HT & -- сохранить регистр флагов в стеке
     "popl %%eax"      & LF & HT & -- загрузить флаги из стека в регистр eax
     "movl %%eax, %0")            -- сохранить значение флагов в переменной
```

Следует обратить внимание на то, что мы заменили обращение к переменной "Eax" на обращение к операнду "%0". Однако, компилятору необходимо указать, чем является "%0":

```
Outputs => Unsigned_32'Asm_Output ("=g", Eax));
```

Здесь, часть "Outputs =>" укажет, что это именованный параметр инструкции ассемблера (полный синтаксис инструкций ассемблера будет рассмотрен позже в "Синтаксис GNAT"). Напомним также, что ранее мы описали переменную "Eax", как переменную типа "Interfaces.Unsigned_32". Мы описали ее таким образом, поскольку 32-битный беззнаковый целый тип удачнее всего подходит для представления регистра процессора. Не трудно заметить, что вывод назначен как атрибут типа, который мы реально хотим использовать для нашего вывода.

```
Unsigned_32'Asm_Output ("=g", Eax);
```

Общая форма такого описания имеет следующий вид:

```
Type'Asm_Output (строка_ограничений, имя_переменной)
```

Смысл имени переменной и атрибута "'Asm_Output" достаточно понятны. Остается понять, что означает и для чего используется "*строка_ограничений*".

33.3.6 Ограничения

Строка ограничений, состоящая из символов, указывает компилятору на то, как он должен управлять переменной, которую мы ему предоставляем. Это значит, что мы указываем компилятору то, как мы интерпретируем смысл этой переменной. Например:

```
Unsigned_32'Asm_Output ("=m", Eax);
```

Здесь, использование ограничения "`m`" (*memory*) указывает компилятору, что переменная "Eax" должна быть переменной которая размещается в памяти. В результате указания этого ограничения, компилятор никогда не будет использовать регистр процессора для хранения этой переменной.

Рассмотрим еще один пример указания ограничения:

```
Unsigned_32'Asm_Output ("=r", Eax);
```

Здесь, использование ограничения "`r`" (*register*) указывает компилятору на использование регистровой переменной. Следовательно, для хранения этой переменной компилятор будет использовать регистр процессора. Если ограничению предшествует символ равенства ("="), то это указывает компилятору, что переменная будет использоваться для сохранения данных.

В показанном ранее примере, при указании ограничения, использовалось ограничение "`g`" (*global*), что позволяет оптимизатору использовать то, что он сочтет более эффективным.

Следует заметить, что существующее число различных ограничений достаточно обширно. При этом, для процессоров архитектуры Intel x86 наиболее часто используемыми ограничениями являются:

- **`=`** — ограничение вывода
- **`g`** — глобальная переменная (т.е. может быть чем угодно)
- **`m`** — переменная в памяти
- **`I`** — константа
- **`a`** — использовать регистр "`eax`"
- **`b`** — использовать регистр "`ebx`"
- **`c`** — использовать регистр "`ecx`"
- **`d`** — использовать регистр "`edx`"
- **`S`** — использовать регистр "`esi`"
- **`D`** — использовать регистр "`edi`"
- **`r`** — использовать один из регистров "`eax`", "`ebx`", "`ecx`" или "`edx`"
- **`q`** — использовать один из регистров "`eax`", "`ebx`", "`ecx`", "`edx`", "`esi`" или "`edi`"

При использовании другого типа процессора или при необходимости получения более исчерпывающей информации об ограничениях, следует обратиться к руководствам по "`gcc`" и "`as`". При необходимости использования более специфических ограничений, которые описаны в руководстве по "`gcc`", следует помнить, что допускается использование комбинаций из нескольких самостоятельных ограничений.

В настоящий момент должно быть понятно, что ограничения используются для указания компилятору как необходимо управлять конкретной переменной (или как ее интерпретировать). Теперь необходимо рассмотреть то, как указать компилятору где эта переменная находится. Для связывания параметра вывода с ассемблерным операндом используется "`%x`"-нотация, в которой "`x`" является числом.

```
Asm ("pushfl"          & LF & HT & -- сохранить регистр флагов в стеке
     "popl %%eax"      & LF & HT & -- загрузить флаги из стека в регистр eax
     "movl %%eax, %0",            -- сохранить значение флагов в переменной
     Outputs => Unsigned_32'Asm_Output ("=g", Eax));
```

Таким образом, в показанном выше фрагменте кода, "%0" будет заменяться фактическим кодом, в соответствии с решением компилятора о фактическом месторасположении переменной "Eax". Означает-ли это, что мы можем иметь только одну переменную вывода? Нет, мы можем иметь их столько сколько нам необходимо. Это работает достаточно просто:

- можно описать множество параметров вывода разделяя их запятыми и завершая список символом точки с запятой
- отсчет операндов ведется в последовательности "%0", "%1", "%2" и т.д., начиная с первого параметра вывода

Для демонстрации сказанного, приведем простой пример:

```
Asm ("movl %%eax, %0" &
     "movl %%ebx, %1" &
     "movl %%ecx, %2",
     Outputs => (Unsigned_32'Asm_Output ("=g", Eax"), --  %0 = Eax
                 (Unsigned_32'Asm_Output ("=g", Ebx"), --  %1 = Ebx
                 (Unsigned_32'Asm_Output ("=g", Ecx)); --  %2 = Ecx
```

Следует нпомнить с чего начиналось написание нашего примера:

```
Asm ("pushfl"          & LF & HT & -- сохранить регистр флагов в стеке
     "popl %%eax"      & LF & HT & -- загрузить флаги из стека в регистр eax
     "movl %%eax, %0",            -- сохранить значение флагов в переменной
Outputs => Unsigned_32'Asm_Output ("=g", Eax));
```

Фактически, мы можем использовать регистровое ограничение для указания компилятору на необходимость сохранять содержимое регистра "**eax**" в переменной "Eax" путем написания следующих инструкций встроенного ассемблера:

```
with Interfaces;               use Interfaces;
with Ada.Text_IO;              use Ada.Text_IO;
with System.Machine_Code;      use System.Machine_Code;
with Ada.Characters.Latin_1; use Ada.Characters.Latin_1;

procedure Flags is
   Eax : Unsigned_32;
begin
   Asm ("pushfl"       & LF & HT & -- сохранить регистр флагов в стеке
        "popl %%eax",              -- загрузить флаги из стека в регистр eax
        Outputs => Unsigned_32'Asm_Output ("=a", Eax));
   Put_Line ("Flags register:" & Eax'Img);
end Flags;
```

Примечательно, что ограничение "**a**", указывает компилятору, что переменная "Eax" будет располагаться в регистре "**eax**". Генерируемый компилятором, код ассемблера будет иметь следующий вид:

```
#APP
   pushfl
   popl %eax
#NO_APP
   movl %eax,-40(%ebp)
```

Очевидно, что значение "**eax**" сохранено в "Eax" компилятором после выполнения кода ассемблера.

Следя за последовательностью обсуждения, внимательный читатель может быть удивлен необходимостью сохранения содержимого регистра "**eax**" в переменной "Eax", когда значение регистра флагов можно сразу вытолкнуть (*pop*) в переменную вывода. Следует согласиться с тем, что такое удивление справедливо. Это было необходимо как стартовая точка дискуссии посвященной ограничениям, но, в действительности, в этом нет необходимости. Таким образом, мы можем переписать исходный текст следующим образом:

```
with Interfaces;                 use Interfaces;
with Ada.Text_IO;                use Ada.Text_IO;
with System.Machine_Code;        use System.Machine_Code;
with Ada.Characters.Latin_1;     use Ada.Characters.Latin_1;

procedure Okflags is
   Eax : Unsigned_32;
begin
   Asm ("pushfl"  & LF & HT & -- сохранить регистр флагов в стеке
        "pop %0",             -- загрузить флаги из стека в переменную Eax
        Outputs => Unsigned_32'Asm_Output ("=g", Eax));
   Put_Line ("Flags register:" & Eax'Img);
end Okflags;
```

В результате, мы получим результат, который был показан ранее.

33.3.7 Использование самостоятельно описываемых типов

Параметры вывода позволяют использовать не только предопределенные типы. Они также позволяют использовать типы которые самостоятельно описаны программистом. В качестве примера, рассмотрим ситуацию в которой необходимо прочитать содержимое регистра флагов процессора и проверить состояние флага переноса (*carry flag*). Известно, что флаг переноса находится в бите 0 регистра флагов процессора. Следовательно, можно использовать соответствующую битовую маску для проверки его состояния:

```
if (Eax and 16#00000001#) = 1 then
   Put_Line ("Carry flag set");
end if;
```

Однако, может возникнуть необходимость в установке или чтении других битов. В таком случае, для облегчения доступа к индивидуальным битам, может оказаться более предпочтительным применение соответствующего типа данных. Примером, который демонстрирует такой подход, может служить следующая программа:

```
with Ada.Text_IO;                use Ada.Text_IO;
with System.Machine_Code;        use System.Machine_Code;
with Ada.Characters.Latin_1;     use Ada.Characters.Latin_1;

procedure Bits is

   subtype Num_Bits is Natural range 0 .. 31;

   type Flags_Register is array (Num_Bits) of Boolean;
   pragma Pack (Flags_Register);
   for Flags_Register'Size use 32;

   Carry_Flag : constant Num_Bits := 0;

   Register : Flags_Register;

begin

   Asm ("pushfl" & LF & HT & --  сохранить регистр флагов в стеке
        "pop %0",            --  загрузить флаги из стека в переменную Register
        Outputs => Flags_Register'Asm_Output ("=g", Register));

   if Register (Carry_Flag) = True then
      Put_Line ("Carry flag set");
   end if;

end Bits;
```

Следует заметить, что в подобных случаях выбор варианта решения больше зависит от требований, которые предъявляются к уровню абстракции в программе. При использовании полной оптимизации ("-O2"), генерируемый компилятором код ассемблера будет идентичен для обоих вариантов:

```
#APP
   pushfl
   pop %eax
#NO_APP
   testb $1,%al
```

Очевидно, что компилятор использует инструкции ассемблера для непосредственной проверки битов которые нас интересуют.

33.3.8 Параметры ввода

До сих пор мы рассматривали программы на встроенном ассемблере, которые способны осуществлять только вывод информации. Однако также часто возникает необходимость в осуществлении ввода информации в программу которая написана на встроенном ассемблере. Рассмотрим пример программы, которая использует функцию написанную на встроенном ассемблере, причем, эта функция принимает входное значение своего параметра, увеличивает это значение на единицу, а затем возвращает его в качестве результата.

```
with Interfaces;              use Interfaces;
with Ada.Text_IO;             use Ada.Text_IO;
with System.Machine_Code;     use System.Machine_Code;

procedure Inc_It is

   function Increment (Value : Unsigned_32) return Unsigned_32 is
      Result : Unsigned_32;
   begin
      Asm ("incl %0",
           Inputs  => Unsigned_32'Asm_Input ("a", Value),
           Outputs => Unsigned_32'Asm_Output ("=a", Result));
      return Result;
   end Increment;

   Value : Unsigned_32;

begin
   Value := 5;
   Put_Line ("Value before is" & Value'Img);
   Value := Increment (Value);
   Put_Line ("Value after is" & Value'Img);
end Inc_It;
```

Следует заметить, что на данном этапе принцип кодирования инструкций встроенного ассемблера уже должен быть понятен:

```
Asm ("incl %0",
     Inputs  => Unsigned_32'Asm_Input ("a", Value),
     Outputs => Unsigned_32'Asm_Output ("=a", Result));
```

Как видно из этого примера, параметр вывода описан так же как и ранее, определяя, что результат, полученный в регистре "`eax`", будет сохранен в переменной "Result". Описание параметра ввода во многом похоже на описание параметра вывода, но использует атрибут "'Asm_Input" вместо атрибута "'Asm_Output". Кроме того, при описании параметра ввода, отсутствует указание ограничения "`=`", указывающего на вывод значения. Также как и в случае параметров вывода, допускается наличие множества параметров ввода. Отсчет параметров ("`%0`", "`%1`", ...) начинается с первого параметра ввода и продолжается при перечислении инструкций вывода. Следует заметить, что в специальных случаях, когда оба параметра используют один и тот же регистр процессора, компилятор будет рассматривать их как один и тот же "`%x`" операнд (например, как в этом случае). Не трудно догадаться, что если параметр вывода сохраняет значение регистра в переменной назначения после выполнения инструкций ассемблера, то параметр ввода используется для загрузки значения переменной в регистр процессора до начала выполнения инструкций ассемблера. Таким образом, следующий фрагмент кода:

```
Asm ("incl %0",
     Inputs  => Unsigned_32'Asm_Input ("a", Value),
     Outputs => Unsigned_32'Asm_Output ("=a", Result));
```

указывает компилятору:

1. загрузить 32-битное значение переменной "Value" в регистр "`eax`"
2. выполнить инструкцию `incl %eax`
3. сохранить содержимое регистра "`eax`" в переменной "Result"

Если посмотреть на сгенерированный компилятором файл ассемблера (используя полную оптимизацию), то можно обнаружить следующее:

```
_inc_it__increment.1:
   subl $4,%esp
   movl 8(%esp),%eax
#APP
   incl %eax
#NO_APP
   movl %eax,%edx
   movl %ecx,(%esp)
   addl $4,%esp
   ret
```

При внимательном рассмотрении всего ассемблерного файла, который был сгенерирован компилятором, можно заметить, что функция, которая описана внутри процедуры, содержит много дополнительного кода, необходимого для того, чтобу установить для этой функции отдельный кадр стека. Следовательно, чтобы улучшить производительность, необходимо описывать маленькие функции на уровне библиотеки или оформлять их как функции для встроенной подстановки (*inline*).

33.3.9 Встроенная подстановка (*inline*) для кода на встроенном ассемблере

При использовании маленьких функций, затраты на выполнение вызова могут оказаться больше чем затраты на выполнение непосредственного кода функции, которая написана на встроенном ассемблере. Это хорошо демонстрирует пример функции "Increment". Решением такой проблемы может быть использование встроенной подстановки (*inline*), которая позволяет в месте вызова подпрограммы (процедуры или функции), написанной на встроенном ассемблере, осуществить непосредственную вставку машинного кода, который соответствует этой подпрограмме. Таким образом, для выполнения встроенной подстановки показанной ранее функции "Increment", необходимо просто добавить соответствующую директиву компилятора:

```
with Interfaces;              use Interfaces;
with Ada.Text_IO;             use Ada.Text_IO;
with System.Machine_Code;     use System.Machine_Code;

procedure Inc_It2 is

   function Increment (Value : Unsigned_32) return Unsigned_32 is
      Result : Unsigned_32;
   begin
      Asm ("incl %0",
           Inputs  => Unsigned_32'Asm_Input ("a", Value),
           Outputs => Unsigned_32'Asm_Output ("=a", Result));
      return Result;
   end Increment;
   pragma Inline (Increment);

   Value : Unsigned_32;

begin
   Value := 5;
   Put_Line ("Value before is" & Value'Img);
   Value := Increment (Value);
   Put_Line ("Value after is" & Value'Img);
end Inc_It2;
```

После компиляции этой программы с указанием выполнения полной оптимизации (опция "-O2") и разрешением осуществления встроенной подстановки (опция "-gnatpn") функция "Increment" будет откомпилирована как обычно, но в том месте, где раньше указывался вызов функции:

```
pushl %edi
call _inc_it__increment.1
```

теперь будет расположен непосредственный код этой функции:

```
movl %esi,%eax
#APP
   incl %eax
#NO_APP
   movl %eax,%edx
```

В конечном итоге, это повысит общую эффективность выполнения кода программы.

33.3.10 "Затирание" содержимого регистров

До настоящего момента времени, мы явно указывали компилятору (посредством параметров ввода и вывода) какие регистры процессора мы используем, и генератор кода компилятора вынужден был это учитывать. Однако, что произойдет в том случае когда мы используем какой-либо регистр процессора, а компилятор об этом не знает? Можно догадаться, что в таком случае компилятор тоже будет использовать этот регистр процессора, а результатом этого будет какой-нибудь "сюрприз"при работе программы. Следовательно, использование в коде ассемблера какого-либо регистра процессора, о котором компилятору ничего не известно, может привести к каким-нибудь побочным эффектам в инструкциях (например, "mull", сохраняющая результат своего действия в двух регистрах "eax" и "edx") или в действиях, описанных в коде ассемблера, подобно следующему:

```
Asm ("movl %0, %%ebx" &
     "movl %%ebx, %1",
     Inputs  => Unsigned_32'Asm_Input  ("=g", Input),
     Outputs => Unsigned_32'Asm_Output ("g", Output));
```

Здесь, компилятор не подозревает о том, что регистр "ebx" уже используется. В подобных случаях, необходимо использовать параметр "Clobber", который дополнительно укажет компилятору регистры, используемые в коде ассемблера:

```
Asm ("movl %0, %%ebx" &
     "movl %%ebx, %1",
     Inputs  => Unsigned_32'Asm_Input  ("=g", Input),
     Outputs => Unsigned_32'Asm_Output ("g", Output),
     Clobber => "ebx");
```

Как видно из этого примера, параметр "Clobber" является простой строкой, которая содержит список используемых регистров процессора. Для случаев использования параметра "Clobber" необходимо сделать два дополнительных замечания:

1. именам регистров не должен предшествовать символ процента
2. имена регистров в списке разделяются символом запятой: "eax, ebx"

Кроме перечисления используемых регистров в параметре "Clobber", существует еще два именованых параметра которые удобно использовать в некоторых случаях:

1. использование регистра "cc" указывает, что могут быть изменены флаги
2. использование регистра "memory" указывает, что может быть изменено расположение в памяти

33.3.11 Изменяемые инструкции

Иногда, оптимизатор компилятора может перехитрить самого себя. Например, когда инструкция ассемблера с параметром ввода расположена внутри цикла, он может переместить загрузку параметра ввода за пределы цикла, подразумевая, что это одноразовая инициализация. Если ваш код не допускает такой трактовки (и вы обнаруживаете этот "сюрприз"достаточно быстро, после анализа генерируемого компилятором ассемблерного файла), то вы можете указать компилятору, чтобы он не пытался перехитрить вас путем установки в "`True`" параметра "`Volatile`". Рассмотрим следующий пример:

```
Asm ("movl %0, %%ebx" &
     "movl %%ebx, %1",
     Inputs   => Unsigned_32'Asm_Input  ("=g", Input),
     Outputs  => Unsigned_32'Asm_Output ("g", Output),
     Clobber  => "ebx",
     Volatile => True");
```

Примечательно, что по умолчанию параметр "`Volatile`" установлен в "`False`" пока не существует ни одного параметра вывода. В результате демонстрации такого примера, может показаться, что установку параметра "`Volatile`" в "`True`" стоит использовать повсеместно. Однако, необходимо заметить, что такая установка будет блокировать оптимизацию кода. Следовательно, ее использование реально необходимо только в тех случаях, когда без этой установки возникают какие-либо проблемы.

33.4 Синтаксис GNAT

В заключение обсуждения использования средств встроенного ассемблера представим полный синтаксис инструкции встроенного ассемблера GNAT, который непосредственно позаимствован из документации GNAT:

```
ASM_CALL ::= Asm (
                  [Template =>] static_string_EXPRESSION
                [,[Outputs  =>] OUTPUT_OPERAND_LIST      ]
                [,[Inputs   =>] INPUT_OPERAND_LIST       ]
                [,[Clobber  =>] static_string_EXPRESSION ]
                [,[Volatile =>] static_boolean_EXPRESSION] )

OUTPUT_OPERAND_LIST ::=
  No_Output_Operands
| OUTPUT_OPERAND_ATTRIBUTE
| (OUTPUT_OPERAND_ATTRIBUTE {,OUTPUT_OPERAND_ATTRIBUTE})

OUTPUT_OPERAND_ATTRIBUTE ::=
  SUBTYPE_MARK'Asm_Output (static_string_EXPRESSION, NAME)

INPUT_OPERAND_LIST ::=
  No_Input_Operands
| INPUT_OPERAND_ATTRIBUTE
| (INPUT_OPERAND_ATTRIBUTE {,INPUT_OPERAND_ATTRIBUTE})

INPUT_OPERAND_ATTRIBUTE ::=
  SUBTYPE_MARK'Asm_Input (static_string_EXPRESSION, EXPRESSION)
```

Глава 34

Отладка проекта

34.1 Директивы компилятора для отладки

GNAT предусматривает две директивы компилятора, которые могут быть полезны при отладке программ:

Директива	Описание
pragma Assert (*условие*);	Вставить условие контрольной проверки
pragma Debug (*имя_процедуры*);	Указание необходимости выполнения вызова отладочной процедуры

Следует заметить, что для активации использования этих директив, необходимо указать опцию командной строки компилятора "**-gnata**" при компиляции исходных текстов. В тех случаях, когда при компиляции исходных текстов опция "**-gnata** не указана, GNAT будет просто игнорировать присутствие этих директив. Таким образом, это позволяет легко компилировать окончательно отлаженную, правильно работающую версию программы, которая предназначена для публичного использования, без необходимости удаления инструкций отладки из исходных текстов программы.

Директива компилятора "Assert" позволяет осуществлять проверку истинности некоторых условий в процессе выполнения программы. Если указанные условия ложны, то осуществляется возбуждение исключения. Эту директиву удобно использовать при отладке программы когда над разработкой проекта работает несколько программистов, поскольку в таком случае некоторые условия выполнения программы могут изменяться достаточно часто. Примером использования директивы "Assert" может служить следующее:

```
pragma Assert (Screen_Height = 24);
```

Здесь, если значение переменной "Screen_Height" не будет равно 24, то будет возбуждено исключение "ASSERT_ERROR".

Директива компилятора "Debug" позволяет осуществлять вызов процедуры, которая предназначена для отладки. Например, она может быть использована на этапе отладки для отображения какой-либо внутренней информации, которая может быть полезна для анализа процесса выполнения программы. Примечательно, что эту директиву компилятора можно поместить в любое место программы, и даже внутрь описания переменной. Например:

```
X := 5;
pragma Debug (Print_To_Log_File ("X is now" & X'Img));
```

В данном случае, если предположить, что процедура "Print_To_Log_File" используется для сохранения сообщения в каком-либо файле протокола (*log*-файле), то этот пример сохранит в этом файле протокола сообщение "`X is now 5`". Следует также заметить, что действие директивы компилятора "Debug" может быть подавлено с помощью директивы "Suppress_Debug_Info".

При выполнении отладки программ могут быть полезными еще две директивы компилятора. Директива компилятора "No_Return" может быть использована для указания подпрограмм, которые никогда не завершаются (иначе, не возвращают управление). Это позволяет подавить соответствующие предупреждающие сообщения компилятора.

Директива компилятора "Normalize_Scalars" инициализирует любую переменную, которая не является массивом, записью или тэговой записью, недопустимым значением (если это возможно). Применение этой директивы позволяет осуществлять проверку правильности инициализации переменных перед тем как они будут использованы (при этом, ее следует размещать в начале подпрограммы или пакета).

В качестве примера, предположим, что существует целочисленная переменная, диапазон значений которой ограничен величинами от 1 до 100. Обычно Ада не осуществляет установку начальных значений, пока это не указано явно. При указании директивы "Normalize_Scalars", такая переменная будет инициализирована каким-либо начальным значением, величина значения которого будет заведомо находиться вне указанного диапазона значений (например, -1). Таким образом, попытка использования значения этой переменной без предварительной инициализации приведет к возбуждению исключения "CONSTRAINT_ERROR".

Следует заметить, что директива "Normalize_Scalars" будет работать лучше если при компиляции исходных текстов использована опция компилятора "**-gnatVf**".

34.2 Получение расширенной информации компилятора

Опция командной строки компилятора "**-gnatG**" позволяет получить информацию о том как GNAT интерпретирует исходный текст после его первоначального анализа. Если при этом одновременно использована опция "**-gnatD**", то GNAT сохранит эту информацию в файле с расширением имени "`.dg`" (для отладки "*debug*").

Для демонстрации использования опции "**-gnatG**" рассмотрим пример простой программы "Pointers":

```
with System.Address_To_Access_Conversions;
with Ada.Text_IO;                              use  Ada.Text_IO;

procedure Pointers is

  package Int_Ptrs is new System.Address_To_Access_Conversions( Integer );
    -- Конкретизация настраиваемого пакета для обеспечения возможности
    -- выполнения преобразования между ссылочными типами и адресами.
    -- Это также дополнительно создает ссылочный тип Object_Pointer,
    -- позволяющий ссылаться на значения целочисленного типа Integer.

  Five        : aliased Integer := 5;
    -- Переменная Five описана как косвенно адресуемая,
    -- поскольку мы будем обращаться к ней с помощью
    -- значений ссылочного типа

  Int_Pointer : Int_Ptrs.Object_Pointer;
    -- Обобщенный ссылочный тип Ады

  Int_Address : System.Address;
    -- Адрес в памяти, подобный указателю языка C

begin

  Int_Pointer := Five'Unchecked_Access;
    -- Использование атрибута 'Unchecked_Access необходимо,
    -- поскольку переменная Five локальна для головной программы.
    -- Если бы она была глобальной, то мы могли бы использовать
    -- атрибут 'Access вместо атрибута  'Unchecked_Access.

  Int_Address := Five'Address;
    -- Адреса моут быть определены с помощью атрибута 'Address.
    -- Это эквивалентно указателям языка C.

  Int_Pointer := Int_Ptrs.To_Pointer( Int_Address );
  Int_Address := Int_Ptrs.To_Address( Int_Pointer );
    -- Преобразование типов указателей Ады и C.

end Pointers;
```

Используя опцию командной строки компилятора "**-gnatG**", для показанной выше программы "Pointers", можно получить от GNAT информацию о результате анализа этой программы. В данном случае, это отображает результаты конкретизации настраиваемого пакета:

Source recreated from tree for Pointers (body)

```
with ada;
with system;
with system.system__address_to_access_conversions;
with ada.ada__text_io;
use ada.ada__text_io;
with system;
with system;
with unchecked_conversion;

procedure pointers is

   package int_ptrs is
      subtype int_ptrs__object is integer;
      package int_ptrs__address_to_access_conversions renames int_ptrs;
      type int_ptrs__object_pointer is access all int_ptrs__object;
      for int_ptrs__object_pointer'size use 32;
      function pointers__int_ptrs__to_pointer (value : system__address)
        return int_ptrs__object_pointer;
      function pointers__int_ptrs__to_address (value :
        int_ptrs__object_pointer) return system__address;
      pragma convention (intrinsic, pointers__int_ptrs__to_pointer);
      pragma convention (intrinsic, pointers__int_ptrs__to_address);
   end int_ptrs;

   package body int_ptrs is

      function pointers__int_ptrs__to_address (value :
        int_ptrs__object_pointer) return system__address is
      begin
         if value = null then
            return system__null_address;
         else
            return value.all'address;
         end if;
      end pointers__int_ptrs__to_address;

      function pointers__int_ptrs__to_pointer (value : system__address)
        return int_ptrs__object_pointer is

         package a_to_pGP3183 is
            subtype a_to_pGP3183__source is system__address;
            subtype a_to_pGP3183__target is int_ptrs__object_pointer;
            function
              pointers__int_ptrs__to_pointer__a_to_pGP3183__a_to_pR (s :
              a_to_pGP3183__source) return a_to_pGP3183__target;
         end a_to_pGP3183;
         function a_to_p is new unchecked_conversion (system__address,
           int_ptrs__object_pointer);
      begin
         return a_to_pGP3183__target!(a_to_pGP3183__source(value));
      end pointers__int_ptrs__to_pointer;
   end int_ptrs;

   package int_ptrs is new system.system__address_to_access_conversions
     (integer);
   five : aliased integer := 5;
   int_pointer : int_ptrs.int_ptrs__object_pointer := null;
   int_address : system.system__address;
```

```
begin
   int_pointer := five'unchecked_access;
   int_address := five'address;
   int_pointer := int_ptrs.pointers__int_ptrs__to_pointer (int_address);
   int_address := int_ptrs.pointers__int_ptrs__to_address (int_pointer);
   return;
end pointers;
```

34.3 Использование отладчика GNU GDB

Начиная с версии 3.11, компилятор GNAT поставляется вместе с работающим в режиме командной строки отладчиком GNU GDB. Следует заметить, что при программировании на языке Ада частое использование отладчика, как правило, не требуется. Это объясняется тем, что Ада обеспечивает хорошую раннюю диагностику, которая осуществляется на этапе компиляции, а при необходимости более точной проверки правильности работы программы удобно использовать соответствующие отладочные директивы компилятора и хорошо известную процедуру "Put_Line", которая выведет нуждающиеся в проверке значения в какой-либо файл протокола. Тем не менее, бывают случаи, когда в результате какой-либо смутной ошибки выполнение программы выглядит непредсказуемо или, по непонятной причине, программа вовсе не выполняется. В подобных ситуациях, в качестве инструмента для поиска источника проблемы можно использовать отладчик GNU GDB.

34.3.1 Общие сведения об отладчике GNU GDB

Отладчик GDB является платформенно независимым отладчиком общего назначения. Он может быть использован для отладки программ, которые написаны на разных языках программирования и скомпилированы с помощью GCC. В частности, с его помощью можно отлаживать Ада-программы, которые скомпилированы с помощью компилятора GNAT. Последние версии отладчика GDB обеспечивают поддержку Ада-программ и позволяют работать со сложными структурами данных Ады. Следует заметить, что здесь приводится только краткий обзор, а исчерпывающая информация по использованию отладчика GDB содержится в руководстве "Отладка с помощью GDB"(*Debugging with GDB*).

В процессе компиляции программы, компилятор, опционально, способен записывать в генерируемый объектный файл отладочную информацию, которая может включать номера строк, описываемые типы данных и переменные Эта информация хранится в результирующем файле отдельно от сгенерированного объектного кода. Следует заметить, что наличие отладочной информации может значительно увеличить результирующий файл, но отладочная информация никак не влияет на исполняемый код, который непосредственно загружается в память, и на его производительность. Генерация отладочной информации инициируется указанием опциии "`-g`" в команде запуска компилятора "`gnatgcc`"или утилиты "`gnatmake`". Необходимо особо подчеркнуть, что использование этой опции никак не влияет на генерируемый машинный код.

Отладочная информация записывается в стандартном системном формате, который используется множеством инструментальных средств, включая отладчики и профиляторы. Обычно, используемый формат разработан для описания семантики и типов языка C, но GNAT реализует схему трансляции использование которой позволяет кодировать информацию о типах и переменных Ады в стандартном формате языка C. В большинстве случаев подробности реализации этой схемы трансляции не представляют интереса для пользователя, поскольку GDB выполняет необходимые преобразования автоматически. Однако, в случае необходимости, информация об этой схеме трансляции может быть получена из файла "`exp_dbug.ads`" дистрибутива исходных текстов GNAT.

При компоновке исполняемой программы, отладочная информация из отдельных объектных файлов собирается воедино и сохраняется в образе исполняемого программного файла программы. Хотя в результате такого процесса получается значительное увеличение размера генерируемого исполняемого файла программы, при этом не происходит непосредственное увеличение размера исполняемой программы. Более того, если программа выполняется обычным образом (не под управлением отладчика), то она выполняется также как и в случае отсутствия отладочной информации, и, в реальности, не требует дополнительной памяти.

В случае запуска программы под управлением GDB осуществляется активация отладчика. Образ программы загружается вплоть до точки запуска программы на выполнение. При указании команды "`run`" выполнение программы происходит также как и без GDB. Это является ключевым моментом в философии дизайна GDB. GDB не вмешивается в работу программы до тех пор пока не будет встречена точка прерывания работы

программы. Если процесс выполнения программы не встречает точек прерывания, то программа выполняется также как и при отсутствии отладчика. При обнаружении точки прерывания, GDB выполняет чтение отладочной информации и, затем, может реагировать на команды пользователя, позволяющие анализировать состояние исполняемой программы.

Здесь рассматривается базовое использование GDB в текстовом режиме. Команда запуска GDB имеет следующий вид:

```
$ gdb program
```

Здесь, `program` — это имя исполняемого файла программы.

Заметим, что в случае использования дистрибутива GNAT от *ALT*, вместо команды "`gdb`"следует использовать команду "`gnatgdb`":

```
$ gnatgdb program
```

В результате выполнения показанной команды осуществляется активация отладчика и отображение приглашения командной строки отладчика "`(gdb)`". В этом случае, самая простая команда "`run`", которая запускает программу на выполнение таким же образом, как и без отладчика.

34.3.2 Знакомство с командами GDB

Отладчик GDB поддерживает обширный набор команд. Руководство "Отладка с помощтью GDB"(*Debugging with GDB*) содержит полное описание всех команд и предоставляет большое количество примеров их использования. Более того, GDB имеет встроенную команду "`help`", которая выдает краткую справку о доступных командах и их опциях. Следует заметить, что здесь, для демонстрации принципов использования отладчика GDB, рассматриваются только несколько наиболее часто употребляемых команд. При чтении данного материала следует создать какую-нибудь простую программу с отладочной информацией и поэкспериментировать с указанными командами.

`set args` ***`arguments`*** С помощью данной команды можно указать список аргументов "***`arguments`***", которые будут переданы программе при выполнении последующей команды "`run`". В использовании команды "`set args`" нет необходимости, когда программа не нуждается в аргументах.

`run` Команда "`run`" осуществляет запуск программы на выполнение со стартовой точки. Если программа была запущена ранее, а в текущий момент ее выполнение приостановлено в какой-либо точке прерывания, то отладчик запросит подтверждение на прекращение текущего выполнения программы и перезапуск выполнения программы со стартовой точки.

`breakpoint` ***`location`*** С помощью данной команды можно осуществлять установку точек прерывания, в которых GDB приостановит выполнение программы и будет ожидать последующие команды. В качестве "***`location`***" можно использовать номер строки в файле, который указывается в виде "`файл:номер_строки`", или имя подерограммы. При указании совмещенного имени, которое используют несколько подпрограмм, отладчик потребует уточнение на какой из этих подпрограмм необходима установка точки прерывания. В этом случае можно также указать, что точки прерывания должны быть установлены на всех подпрограммах. Когда выполняемая программа достигает точки прерывания, ее выполнение приостанавливается, а GDB сигнализирует об этом печатью строки кода, перед которой произошла остановка выполнения программы.

`breakpoint exception` ***`name`*** Это специальная форма команды установки точки прерывания, которая позволяет прервать выполнение программы при возбуждении исключения с именем "***`name`***". Если имя "***`name`***" не указано, то приостановка выполнения программы осуществляется при возбуждении любого исключения.

`print` ***`expression`*** Эта команда печатает значение выражения "***`expression`***". GDB нормально обрабатывает основные простые выражения Ады, благодаря чему, выражение "***`expression`***" может содержать вызовы функций, переменные, знаки операций и обращения к атрибутам.

`continue` Эта команда позволяет продолжить выполнение программы, после приостановки в какой-либо точке прерывания, вплоть до завершения работы программы или до достижения какой-либо другой точки прерывания.

step Выполнить одну строку кода после точки прерывания. Если следующая инструкция является вызовом подпрограммы, то последующее выполнение будет продолжено внутри (с первой инструкции) вызываемой подпрограммы

next Выполнить одну строку кода. Если следующая инструкция является вызовом подпрограммы, то выполнение вызова подпрограммы и возврат из нее осуществляется без остановок.

list Отобразить несколько строк вокруг текущего положения в исходном тексте. Практически, более удобно иметь отдельное окно редактирования, в котором открыт соответствующий файл с исходным текстом. Последующее исполнение этой команды отобразит несколько последующих строк исходного текста. Эта команда может принимать номер строки в качестве аргумента. В этом случае она отобразит несколько строк исходного текста вокруг указанной строки.

backtrace Отобразить обратную трассировку цепочки вызовов. Обычно эта команда используется для отображения последовательности вызовов, которые привели в текущую точку прерывания. Отображение содержит по одной строке для каждой записи активации (кадр стека) соответствующей активной подпрограмме.

up При остановке выполнения программы в точке прерывания, GDB способен отобразить значения переменных, которые локальны для текущего кадра стека (иначе — уровень вложения вызовов). Команда "`up`" может быть использована для анализа содержимого других активных кадров стека, перемещаясь на один кадр стека вверх (от кадра стека вызванной подпрограммы к кадру стека вызвавшей подпрограммы).

down Выполняет перемещение на один кадр стека вниз. Действие этой команды противоположно действию команды "`up`".

frame *n* Переместиться в кадр стека с номером "*n*". Значение 0 указывает на кадр стека, который соответствует положению текущей точки прерывания, то есть, вершине стека вызовов.

Показанный выше список команд GDB является ознакомительным, и он очень краток. О дополнительных возможностях, которыми являются условные точки прерывания, способность исполнения последовательности команд в точке прерывания, способность выполнять отладку программы на уровне команд процессора, и многих других полезных свойствах отладчика GDB можно узнать из руководства "Отладка с помощью GDB"(*Debugging with GDB*). Примечательно, что многие основные команды имеют сокращенные аббривиатуры. Например, "`c`" — для "`continue`", "`bt`" — для "`backtrace`".

34.3.3 Использование выражений Ады

Отладчик GDB обеспечивает поддержку для достаточно обширного подмножества синтаксиса выражений Ады. Основой философии дизайна этого подмножества является следующее:

- GDB должен предусматривать элементарные средства позволяющие использовать литералы, доступ к арифметическим операциям и операциям со ссылочными значениями, выборку полей записей, работу с индексами и вызовами подпрограмм, а все более сложные вычисления возлагать на подпрограммы написанные внутри программы (которые, таким образом, могут быть вызваны из GDB).

- Надежность и сохранность типов, а также строгая приверженность к ограничениям языка программирования Ада не так важна для пользователя GDB.

- Для пользователя GDB важна краткость.

Таким образом, для краткости, отладчик действует так, как будто спецификаторы "**with**" и "**use**" явно указаны для всех написанных пользователем пакетов, что позволяет избавиться от необходимости использования полной точечной нотации для большинства имен. В случае возникновения двусмысленности GDB запрашивает у пользователя соответствующее уточнение. Более подпробно поддержка синтаксиса Ады рассматривается в руководстве "Отладка с помощью GDB"(*Debugging with GDB*).

34.3.4 Вызов подпрограмм определяемых пользователем

Одной из важных особенностей отладчика GDB является его способность вызывать в процессе отладки подпрограммы, определяемые пользователем. Это можно выполнить путем простого ввода команды вызова подпрограммы, которая имеет следующий вид:

```
call subprogram-name (parameters)
```

Ключевое слово "`call`" может быть опущено в случае, когда имя подпрограммы "`subprogram-name`" не совпадает с какой-либо предопределенной командой GDB.

В результате выполнения этой команды осуществляется вызов указанной подпрограммы и передача ей списка параметров, которые заданы в команде. В качестве параметров могут быть указаны выражения и они могут содержать переменные отлаживаемой программы. Подпрограмма должна быть определена в программе на уровне библиотеки, поскольку GDB вызывает подпрограмму в окружении, которое устанавливается для выполнения отлаживаемой программы (это также подразумевает, что подпрограмма обладает возможностью доступа и даже может модифицировать значения переменных внутри программы).

Важность данного средства заключается в том, что оно позволяет использовать в программе различные отладочные подпрограммы, которые приспособлены к каким-либо специфическим структурам данных отлаживаемой программы. Подобные подпрограммы отладки могут быть написаны с целью получения высокоуровневого отображения внутреннего состояния какого-либо абстрактного типа данных, что намного удобнее чем использование распечатки содержимого памяти (иначе — дамп памяти), которая заната структурой данных, на физическом уровне. Следует учитывать, что стандартная команда отладчика GDB "`print`" имеет представление только о физическом расположении типов в памяти, а не об их смысловом значении и абстрактном представлении. Отладочные подпрограммы могут предусматривать отображение информации с учетом требуемого семантического уровня и, таким образом, их использование при отладке программы может быть очень полезно.

Например, при непосредственной отладке GNAT, ключевым моментом является наличие доступа к содержимому узлов семантического дерева, которое содержит внутреннее представление программы. Однако, узлы семантического дерева представляются в виде целочисленных значений, которые являются индексами в таблицу узлов. Использование встроенной команды "`print`", для узлов семантического дерева, обеспечит простое отображение целочисленных значений, что не обладает желаемой информативностью. Таким образом, намного удобнее использовать подпрограмму "`PN`" (описана в файле "`treepr.adb`" дистрибутива исходных текстов GNAT), которая принимает на входе узел семантического дерева и отображает его высокоуровневое представление, включающее синтаксическую категорию данного узла, его место в исходном тексте, целочисленные значения, соответствующие родительскому узлу и узлам потомкам, а также множество дополнительной семантической информации. Для более детального изучения этого примера можно обратиться к телу этой процедуры в указанном файле.

34.3.5 Исключения и точки прерывания

С помощью отладчика GDB можно устанавливать точки прерывания, в которых будет осуществляться приостановка выполнения программы, когда ее выполнение будет возбуждать указанное исключение.

`break exception` Установить точку прерывания, в которой будет осуществляться приостановка выполнения программы, когда программа (или любая ее задача) возбуждает какое-либо (любое) исключение.

`break exception` ***`name`*** Установить точку прерывания, в которой будет осуществляться приостановка выполнения программы, когда программа (или любая ее задача) возбуждает исключение с именем "***`name`***".

`break exception unhandled` Установить точку прерывания, в которой будет осуществляться приостановка выполнения программы, когда программа (или любая ее задача) возбуждает какое-либо исключение для которого не предусмотрен обработчик.

`info exceptions`

`info exceptions` ***`regexp`*** Команда "`info exceptions`" позволяет пользователю анализировать все исключения, которые определены внутри Ада-программы. При использовании в качестве аргумента регулярного выражения "***`regexp`***", будет отображена информация только о тех исключениях, чьи имена совпадают с указанным регулярным выражением.

34.3.6 Задачи Ады

При работе с задачами Ады, GDB позволяет использовать следующие команды:

info tasks Эта команда отображает список текущих задач Ады, который имеет вид подобный следующему:

```
(gdb) info tasks
  ID       TID P-ID   Thread Pri State                  Name
   1   8088000   0   807e000  15 Child Activation Wait main_task
   2   80a4000   1   80ae000  15 Accept/Select Wait    b
   3   809a800   1   80a4800  15 Child Activation Wait a
*  4   80ae800   3   80b8000  15 Running               c
```

При отображении списка, символ звездочки указывает на выполнение задачи в текущий момент времени. Первый столбец отображает идентификатор задачи (*task ID*), который используется для обращения к задачам в следующих командах.

break *linespec* task *taskid*

break *linespec* task *taskid* if ... Эти команды подобны командам "`break ... thread ...`", а "*`linespec`*" указывает строки исходного текста. Чтобы указать GDB на необходимость приостановки выполнения программы, когда определенная задача Ады достигнет точки прерывания, следует использовать квалификатор "`task` *`taskid`*". При этом, "*`taskid`*" является численным идентификатором задачи, который назначается отладчиком GDB (он отображается в первом столбце, при отображении списка задач с помощью команды "`info tasks`"). Если при установке точки прерывания квалификатор "`task` *`taskid`*" не указан, то точки прерывания будут установлены для всех задач программы. Квалификатор "`task`" может быть также использован при установке условных точек прерывания. В этом случае квалификатор "`task` *`taskid`*" указывается перед указанием условия точки прерывания (перед "`if`").

task *taskno* Эта команда позволяет переключиться на задачу, которая указывается с помощью параметра "*`taskno`*". В частности, это позволяет просматривать и анализировать обратную трассировку стека указанной задачи. Необходимо заметить, что перед продолжением выполнения программы следует выполнить переключение на оригинальную задачу, в которой произошла приостановка выполнения программы, иначе возможен сбой работы планировщика задач программы.

Более подробные сведения о поддержке задач содержатся в руководстве "Отладка с помощью GDB"(*Debugging with GDB*).

34.3.7 Отладка настраиваемых модулей

При конкретизации настраиваемого модуля GNAT всегда использует расширение кода. Это подразумевает, что при каждой конкретизации осуществляется копирование оригинального кода с соответствующими подстановками фактических параметров настройки вместо формальных параметров.

GDB не обеспечивает возможность обращения к оригинальным сущностям настраиваемого модуля, но он позволяет выполнять отладку конкретного экземпляра настроенного модуля, который получен в результате конкретизации, путем исползования соответствующих расширенных имен. Рассмотрим следующий пример:

```
procedure g is

   generic package k is
      procedure kp (v1 : in out integer);
   end k;

   package body k is
      procedure kp (v1 : in out integer) is
      begin
         v1 := v1 + 1;
      end kp;
   end k;
```

```
   package k1 is new k;
   package k2 is new k;

   var : integer := 1;

begin
   k1.kp (var);
   k2.kp (var);
   k1.kp (var);
   k2.kp (var);
end;
```

В этом случае, для того, чтобы прервать выполнение программы при вызове процедуры "kp" в экземпляре настроенного модуля "k2", необходимо использовать следующую команду:

```
(gdb) break g.k2.kp
```

Теперь, при попадании в точку прерывания, можно обычным образом осуществить пошаговое выполнение кода экземпляра настроенного модуля, и проанализировать значения локальных переменных, также как и в случае обычного программного модуля.

34.4 Ограничение возможностей языка

Существует несколько директив компилятора, которые позволяют накладывать ограничения на использование в программе некоторых возможностей языка. Такие директивы компилятора могут быть использованы для принудительной установки каких-либо специфических правил и выдачи программисту предупреждающих сообщений, когда эти правила нарушаются. Например, в случае написания программы реального времени, может возникнуть необходимость в отключении средств Ады которые обладают неопределенным временем реакции (или отклика). В результате этого, программа не будет иметь случайных задержек во время своего выполнения.

Директивами компилятора, которые управляют ограничением средств языка являются:

Ada_83	Запретить использование средств стандарта Ada 95.
Ada_95	Разрешить использование средств стандарта Ada 95 (установлено по умолчанию).
Controlled	Отключить "сборку мусора"(*garbage collection*) для указанного типа данных. Реально, это не воздействует на GNAT, поскольку он не обеспечивает средств "сборки мусора".
Ravanscar	Принудительная установка ограничений реального времени, которые соответствуют требованиям профиля *Ravanscar*.
Restricted_Run_Time	Подобна директиве "Ravanscar".
Restrictions	Отключает некоторые средства языка.

Следует заметить, что директива компилятора "No_Run_Time" также является директивой установки ограничений, поскольку осуществляет принудительное отключение использования библиотеки времени выполнения Ады.

Более подробная информация об использовании этих директив компилятора находится в документации компилятора GNAT.

Глава 35

Дополнительные сведения о компиляторе GNAT

35.1 Некорректное завершение работы компилятора

В редких случаях, при компиляции программ, которые содержат серьезные ошибки в синтаксисе и/или в семантике, работа GNAT может быть завершена как результат ошибки сегментации или результат недопустимого обращения к памяти, возбуждая внутреннее исключение, или каким-либо другим не корректным образом. В таких случаях, возможна активация различных свойств GNAT, которые способны помочь в поиске источника проблемы внутри программы.

Ниже, представлен перечень стратегий, которые могут быть использованы в подобных случаях. Стратегии перечисляются в порядке усложнения, и их использование зависит от опыта и знакомства с внутренним строением компиляора.

1. Запуск "`gnatgcc`" с опциями "**`-gnatf`**" и "**`-gnate`**". Опция "**`-gnatf`**" указывает на необходимость вывода сообщений о всех ошибках, которые обнаружены в строке исходного текста. При ее отсутствии, выдается сообщение только о первой обнаруженной ошибке. Опция "**`-gnate`**" указывает на необходимость вывода сообщения об ошибке сразу после обнаружения ошибки, а не после завершения процесса компиляции. Когда GNAT завершает свою работу преждевременно, наиболее вероятным источником возникшей проблемы будет сообщение об ошибке, которое отображается последним.

2. Запуск "`gnatgcc`" с указанием опции "многословности"вывода "**`-v (verbose)`**". В этом режиме, "`gnatgcc`" генерирует непрерывную информацию о процессе компиляции и предусматривает имя каждой процедуры код которой сгенерирован. Эта опция позволяет осуществлять поиск Ада-процедуры при генерации кода которой возникают проблемы.

3. Запуск "`gnatgcc`" с указанием опции "**`-gnatdc`**". Эта опция специфична для GNAT и ее воздействие на препроцессор (*front-end*) аналогично воздействию опции "**`-v`**" на кодогенератор (*back-end*). Система распечатывает имя каждого модуля (как компилируемого модуля, так и вложенного модуля) в процессе его анализа.

4. В заключение, можно запустить отладчик "`gnatgdb`" непосредственно для исполняемого файла "`gnat1`". Исполняемый файл "`gnat1`" является препроцессором (*front-end*) GNAT, который может быть самостоятельно запущен на выполнение (в обычной ситуации его запуск осуществляется из "`gnatgcc`"/"`gcc`"). Использование отладчика "`gnatgdb`" для "`gnat1`" аналогично использованию "`gnatgdb`" для программ на языке C. Команда "`where`" является первой командой, которая используется при поиске проблемы; переменная "`lineno`" (которую можно увидеть с помощью "`print lineno`"), используется во второй фазе "`gnat1`" и в кодогенераторе "`gnatgcc`", индицирует строку исходного текста, в которой произошла остановка выполнения, а "`input_file name`" индицирует имя файла с исходным текстом.

35.1.1 Получение внутренней отладочной информации

Большинство компиляторов обладают внутренними опциями и режимами отладки. В этом смысле GNAT не является исключением, более того, все внутренние опции и режимы отладки GNAT не являются секретом.

Общее и полное описание всех опций отладки компилятора и редактора связей можно найти в файле "`debug.adb`", который присутствует в комплекте исходных файлов GNAT.

Опции позволяющие получить и распечатать исходный текст программы, который построен из дерева внутреннего представления, представляют большой интерес для пользователей программ. Не менее интересны опции, которые позволяют распечатывать полное дерево внутреннего представления и таблицу сущностей (иначе, таблица с информацией о символах). Восстановленный из внутреннего представления исходный текст предоставляет читаемую версию программы, которая получена из внутреннего представления после того как препроцессор (*front-end*) завершил анализ и расширение. Такой исходный текст может быть полезен при изучении эффективности характерных конструкций. Например, в получаемом таким образом исходном тексте: индицированы проверки ограничений, сложные агрегаты заменены циклами и присваиваниями, примитивы задач заменены обращениями к библиотеке времени выполнения.

35.1.2 Соглашения по наименованию исходных файлов GNAT

При анализе внутреннего устройства системы GNAT, может оказаться полезным следующее краткое описание соглашений по именованию исходных файлов системы:

- Файлы, которые начинаются префиксом "`sc`", содержат лексический сканер.
- Все файлы, которые начинаются префиксом "`par`", являются компонентами синтаксического анализатора (*parser*). Число в имени соответствует главе в "Руководстве по языку программирования Ада 95"("*Ada 95 Reference Manual*"). Например, синтаксический разбор инструкции "**select**"может быть обнаружен в файле "`par-ch9.adb`".
- Все файлы, которые начинаются префиксом "`sem`", осуществляют семантический анализ. Число в имени соответствует главе в "Руководстве по языку программирования Ада 95"("*Ada 95 Reference Manual*"). Например, все случаи использования спецификаторов контекста могут быть обнаружены в файле "`sem_ch10.adb`". Дополнительно, некоторые свойства языка требуют значительной специальной обработки. Для поддержки этого служат свои собственные семантические файлы: "`sem_aggr`" — для агрегатов, "`sem_disp`" — для динамической диспетчеризации, и т.д.
- Все файлы, которые начинаются префиксом "`exp`", осуществляют нормализацию и расширение внутреннего представления (абстрактного синтаксического дерева — *abstract syntax tree* или сокращенно — *AST*). Эти файлы используют такую же схему нумерации, которую используют файлы синтаксического и семантического анализа. Например, конструкция процедур инициализации записи выполнена в файле "`exp_ch3.adb`".
- Все файлы, которые начинаются префиксом "`bind`", реализуют редактор связей (*binder*), который осуществляет проверку согласованности компиляции, определяет порядок элаборации и генерирует файл редактора связей (*bind file*).
- Файлы "`atree.ads`" и "`atree.adb`" описывают низкоуровневые структуры данных, которые используются препроцессором (*front-end*).
- Файлы "`sinfo.ads`" и "`sinfo.adb`" описывают подробности структуры абстрактного синтаксического дерева, которое генерируется синтаксическим анализатором (*parser*).
- Файлы "`einfo.ads`" и "`einfo.adb`" описывают подробности атрибутов всех сущностей, которые вычисляются в процессе семантического анализа.
- Управление библиотекой осуществляется в файлах, которые начинаются префиксом "`lib`".
- Ада-файлы, которые начинаются префиксом "`a-`", являются дочерними модулями пакета *Ada*, как это определено в Дополнении A (*Annex A*).
- Все файлы, которые начинаются префиксом "`i-`", являются дочерними модулями пакета *Interfaces*, как это определено в Дополнении B (*Annex B*).
- Все файлы, которые начинаются префиксом "`s-`", являются дочерними модулями пакета *System*. Они включают дочерние модули, которые определены в стандарте языка, и подпрограммы библиотеки времени выполнения GNAT.

- Все файлы, которые начинаются префиксом "`g-`", являются дочерними модулями пакета *GNAT*. Здесь присутствует множество пакетов общего назначения, которые полностью документированы в своих файлах спецификаций.
- Все остальные файлы "`.c`" являются модификацией общих файлов "`gnatgcc`".

Часть 4.

Приложения

Приложение A

Директивы компилятора (*pragma*)

В данном приложении приводится краткое описание директив компилятора. Более подробное описание использования этих директив содержится в справочных руководствах по языку программирования Ада и по реализации компилятора GNAT.

A.1 Стандартные директивы Ады

Список стандартных директив компилятора Ады приводится в приложении L (*Annex L*) руководства по языку программирования Ада (RM-95).

All_Calls_Remote [(library_unit_name)];	Для выполнения вызова подпрограмм пакета всегда использовать механизм RPC (*Remote Procedure Call* — вызов удаленной подпрограммы)
Asynchronous (local_name);	Вызов подпрограммы производится асинхронно. При этом, активируется выполнение удаленной подпрограммы и, не дожидаясь её завершения, происходит продолжение выполнения вызвавшей подпрограммы.
Atomic (local_name);	Чтение/запись указанного объекта должно выполняться без прерываний
Atomic_Components (array_local_name);	Чтение/запись указанного массива компонентов должно выполняться без прерываний
Attach_Handler (handler_name, expression);	Установить процедуру обработки прерывания
Controlled (first_subtype , local_name);	Отключает "сборку мусора"для указанного типа данных. (не имеет эффекта в реализации компилятора GNAT)
Convention ([Convention =>] convention_identifier, [Entity=>] local_name);	Использовать соответствующие языковые соглашения для указанного объекта
Discard_Names [([On =>] local_name)];	Отказаться от *ASCII*-представления объекта, которое используется атрибутом 'Img
Elaborate (library_unit_name{, library_unit_name});	Предварительно выполнить элаборацию указанного пакета
Elaborate_All (library_unit_name{, library_unit_name});	Предварительно выполнить элаборацию всех спецификаций и тел пакетов от которых зависит указанный модуль

Elaborate_Body [(library_unit_name)];	Выполнить элаборацию тела указанного модуля сразу после элаборации его спецификации
Export ([Convention =>] convention_identifier, [Entity =>] local_name[, [External_name =>] string_expression] [, [Link_Name =>] string_expression]);	Экспортировать объект из Ада-программы для использования в другом языке программирования
Import ([Convention =>] convention_identifier, [Entity =>] local_name[, [External_name =>] string_expression] [, [Link_Name =>] string_expression]);	Импортирует объект, описанный средствами другого языка программирования, для использования его в Ада-программе
Inline (name{, name});	Указывает подпрограмму при обращении к которой должна осуществляться встроенная подстановка машинного кода этой подпрограммы вместо выполнения стандартного вызова этой подпрограммы (так называемая *inline*-подстановка).
Inspection_Point [(object_name{, object_name})];	В данной точке программы должна быть обеспечена возможность чтения значения указанного объекта (необходимо для сертификации правильности кода).
Interrupt_Handler (handler_name);	Указывает процедуру-обработчик прерывания
Interrupt_Priority ([expression]);	Определяет приоритет задачи и/или защищенного объекта для случаев возникновения блокировки
Linker_Options (string_expression);	Передает строку опций для компоновщика (*linker*).
List (identifier);	Вывести листинг исходного текста после компиляции.
Locking_Policy (policy_identifier);	Определяет как защищенный объект будет заблокирован при возникновении блокировки.
Normalize_Scalars ;	Устанавливать, когда это возможно, значения скалярных переменных в недопустимое значение
Optimize (identifier);	Указать как должны быть оптимизированы инструкции
Pack (first_subtype_local_name);	Указывает, что тип данных должен быть упакован
Page;	Указывает на начало новой страницы в листинге программы
Preelaborate (library_unit_name);	Указывает на необходимость предварительной элаборации указаанного пакета
Priority (expression);	Определяет приоритет задачи
Pure [(library_unit_name)];	Указывает, что пакет "чистый"(*pure*).
Queuing_Policy (policy_identifier);	Определяет правило сортировки задач и/или защищенных объектов при постановке в очередь.

Remote_Call_Interface [(library_unit_name)];	Подпрограммы пакета могут вызываться с использованием механизма RPC (*Remote Procedure Call* — вызов удаленной подпрограммы)
Remote_Types [(library_unit_name)];	Пакет определяет типы, предназначенные для использования совместно с механизмом RPC (*Remote Procedure Call* — вызов удаленной подпрограммы)
Restrictions (restriction {, restriction });	Отключает некоторые языковые средства.
Reviewable;	Предусматривает профилирование во время выполнения (подобно `gprof`).
Shared_Passive [(library_unit_name)];	Используется для совместного использования глобальных данных между разными *RPC*-разделами распределенной программы.
Storage_Size (expression);	Указывает общий размер пространства стека для задачи.
Suppress (identifier [, [On =>] name]);	Отключает специфические проверки для общих исключений.
Task_Dispatching (policy_identifier);	Определяет, для задачи, правила сортировки вызовов при диспетчеризации (например, *FIFO_Within_Priorities*).
Volatile (local_name);	Указывает, что значение переменной может изменяться в непредсказуемые моменты, вне зависимости от работы программы.
Volatile_Components (array_local_name);	Указывает, что значение массива компонентов может изменяться в непредсказуемые моменты, вне зависимости от работы программы.

A.2 Директивы определенные в реализации компилятора GNAT

Abort_Defer;	Откладывает принудительное завершение до конца блока инструкций (должна быть помещена в начале блока инструкций)
Ada_83;	Использовать при компиляции исходного текста соглашения стандарта Ada83, даже если переключатели опций компилятора указывают обратное
Ada_95;	Использовать при компиляции исходного текста соглашения стандарта Ada95, даже если переключатели опций компилятора указывают обратное
`Annotate ( identifier{, name|expression});`	Добавить дополнительную информацию для внешних инструментальных средств
`Assert ( boolean_expression [, static_string_expression ]);`	Вставить условие контрольной проверки
Ast_Entry (entry_identifier);	Реализована только для OpenVMS

```
C_Pass_By_Copy (
   [Max_Size =>] static_integer_expression );
```

При вызове функции Си, использовать передачу параметра по значению (не по ссылке), если это возможно

```
Comment (static_string_expression);
```

То же самое, что и Ident

```
Common_Object (
      [Internal =>] local_name,
   [, [External =>] external_symbol]
   [, [Size     =>] external_symbol] )
```

Для Фортрана. Создать переменную, которая использует общее пространство

```
Complex_Representation (
   [Entity =>] local_name);
```

Использовать формат комплексных чисел GCC (для повышения скорости вычислений)

```
Component_Alignment (
      [Form =>] alignment_choice
   [, [Name =>] type_local_name]);
```

Указывает как должно осуществляться выравнивание компонентов записи

```
CPP_Class ([Entity =>] local_name);
```

Трактовать запись или тэговую запись как класс Си++

```
CPP_Constructor (
   [Entity =>] local_name);
```

Трактовать импортируемую функцию как конструктор класса Си++

```
CPP_Destructor (
   [Entity =>] local_name);
```

Трактовать импортируемую функцию как деструктор класса Си++

```
CPP_Virtual (
      [Entity     =>] entity,
   [, [Vtable_Ptr =>] vtable_entity,]
   [, [Position   =>] static_integer_expression ]);
```

Импортировать виртуальную функцию Си++

```
CPP_Vtable (
      [Entity     =>] entity,
   [, [Vtable_Ptr =>] vtable_entity,]
   [, [Entry_Count =>] static_integer_expression ]);
```

Определить таблицу виртуальных функций

```
Debug (procedure_call_statement);
```

Определить вызов отладочной процедуры

```
Eliminate ([Unit_Name =>] identifier);
```

Указать объект который не используется программой. Создается утлитой `gnatelim`

Реализована только для OpenVMS

```
Export_Exception (
      [Internal =>] local_name,
   [, [External =>] external_symbol,]
   [, [Form     =>] Ada | VMS]
   [, [Code     =>] static_integer_expression ]);
```

```
Export_Function (
      [Internal          =>] local_name,
   [, [External          =>] external_symbol]
   [, [Parameter_Types   =>] parameter_types]
   [, [Result_Type       =>] result_subtype_mark]
   [, [Mechanism         =>] mechanism]
   [, [Result_Mechanism =>] mechanism_name]);
```

Экспортировать Ада-функцию с указанием дополнительной информации по отношению к информации представленной директивой Export

```
Export_Object (
      [Internal =>] local_name,
   [, [External =>] external_symbol]
   [, [Size     =>] external_symbol]);
```

Экспортировать тэговую запись Ады с указанием дополнительной информации по отношению к информации представленной директивой Export

```
Export_Procedure (
      [Internal          =>] local_name,
   [, [External          =>] external_symbol]
   [, [Parameter_Types   =>] parameter_types]
   [, [Result_Type       =>] result_subtype_mark]
   [, [Mechanism         =>] mechanism]);
```

Экспортировать Ада-процедуру с указанием дополнительной информации по отношению к информации представленной директивой Export

```
Export_Valued_Procedure (
      [Internal          =>] local_name,
   [, [External          =>] external_symbol]
   [, [Parameter_Types   =>] parameter_types]
   [, [Result_Type       =>] result_subtype_mark]
   [, [Mechanism         =>] mechanism]);
```

Экспортировать Ада-функцию обладающую побочными эффектами с указанием дополнительной информации по отношению к информации представленной директивой Export

```
Extend_System ([Name =>] identifier);
```

устарела

```
External_Name_Casing (
   Uppercase | Lowercase
   [, Uppercase | Lowercase | As_Is]);
```

Установить режим перевода в символы верхнего регистра для импорта в регистрозависимый язык. Может быть установлено в *Uppercase*, *Lowercase* или *As_Is* (по умолчанию для GNAT).

```
Finalize_Storage_Only (
   first_subtype_local_name);
```

Не выполнять очистку (*Finalize*) для объектов описанных на уровне библиотеки. Изначально предназначено для внутреннего использования в реализации компилятора GNAT.

```
Float_Representation ( float_rep );
```

Реализована только для OpenVMS

```
Ident ( static_string_expression );
```

Строка идентификации объектного файла (в *Linux*, значения не имеет)

```
Import_Exception (
      [Internal =>] local_name,
   [, [External =>] external_symbol,]
   [, [Form     =>] Ada | VMS]
   [, [Code     =>] static_integer_expression]);
```

Реализована только для OpenVMS

```
Import_Function (
      [Internal         =>] local_name,
   [, [External         =>] external_symbol]
   [, [Parameter_Types  =>] parameter_types]
   [, [Result_Type      =>] result_subtype_mark]
   [, [Mechanism        =>] mechanism]
   [, [Result_Mechanism =>] mechanism_name]
   [, [First_Optional_Parameter =>] identifier]);
```

Импортирует функцию, описанную средствами другого языка программирования, для использования ее в Ада-программе. При этом указывает дополнительную информацию по отношению к информации представленной директивой Import.

```
Import_Object (
      [Internal =>] local_name,
   [, [External =>] external_symbol]
   [, [Size     =>] external_symbol]);
```

Импортирует объект, описанный средствами другого языка программирования, для использования его в Ада-программе. При этом указывает дополнительную информацию по отношению к информации представленной директивой Import.

```
Import_Procedure (
      [Internal         =>] local_name,
   [, [External         =>] external_symbol]
   [, [Parameter_Types  =>] parameter_types]
   [, [Mechanism        =>] mechanism]
   [, [First_Optional_Parameter =>] identifier]);
```

Импортирует процедуру, описанную средствами другого языка программирования, для использования ее в Ада-программе. При этом указывает дополнительную информацию по отношению к информации представленной директивой Import.

```
Import_Valued_Procedure (
      [Internal         =>] local_name,
   [, [External         =>] external_symbol]
   [, [Parameter_Types  =>] parameter_types]
   [, [Mechanism        =>] mechanism]
   [, [First_Optional_Parameter =>] identifier]);
```

Импортирует функцию обладающую побочными эффектами, которая описана средствами другого языка программирования, для использования ее в Ада-программе. При этом указывает дополнительную информацию по отношению к информации представленной директивой Import.

```
Inline_Always (name[, name]);
```

Указывает, на необходимость выполнения межмодульной *inline*-подстановки не зависимо от состояния переключателей опций компилятора.

```
Inline_Generic (generic_package_name);
```

Введена для совместимости с другими Ада-компиляторами

```
Interface (
      [Convention    =>] convention_identifier,
      [Entity =>] local_name
   [, [External_Name =>] static_string_expression],
   [, [Link_Name     =>] static_string_expression]);
```

Введена для совместимости с другими Ада-компиляторами

```
Interface_Name (
      [Entity        =>] local_name
   [, [External_Name =>] static_string_expression]
   [, [Link_Name     =>] static_string_expression]);
```

Введена для совместимости с другими Ада-компиляторами

```
Linker_Alias (
   [Entity =>] local_name
   [Alias  =>] static_string_expression);
```

Определяет альтернативный компоновщик (*linker*) для указанного пакета (или другого компонуемого модуля).

`Linker_Section (`
`  [Entity  =>] local_name`
`  [Section =>] static_string_expression);`
	Указание компоновщику GCC использовать соответствующую секцию для указанного имени

Long_Float (float_format);	Реализована только для OpenVMS

`Machine_Attribute (`
`  [Attribute_Name =>] string_expression,`
`  [Entity  =>] local_name);`
	Определяет атрибуты машины для GCC.

`Main_Storage (` `  main_storage_option [, main_storage_option]);`	Реализована только для OpenVMS
No_Run_Time;	Указать компилятору GNAT не использовать библиотеку времени выполнения (например, в случае написания драйвера устройства).
No_Return (procedure_local_name);	Указывает процедуру, которая преднамеренно не возвращает управление. Предназначена для подавления предупреждающих сообщений компилятора.
Passive ([Semaphore | No]);	Введена для совместимости с другими Ада-компиляторами
Polling (On | Off);	Если включена, то разрешает опрос исключений
Propagate_Exceptions (subprogram_local_name);	Определяет импортированную подпрограмму, которая будет обрабатывать исключения Ады (без потерь производительности).

`Psect_Object (`
`  [Internal =>] local_name,`
`  [, [External =>] external_symbol]`
`  [, [Size  =>] external_symbol]);`
	То же самое, что и Common_Object.

`Pure_Function (` `  [Entity =>] function_local_name);`	Указывает, что функция не имеет побочных эффектов.
Ravenscar;	Устанавливает политику реального времени RAVENSCAR
Restricted_Run_Time;	Подобно директиве Ravenscar, устанавливает некоторые ограничения на программирование задач реального времени
Share_Generic (name {, name});	Введена для совместимости с другими Ада-компиляторами

`Source_File_Name (`
`  [Unit_Name  =>] unit_name,`
`  [FNAME_DESIG =>] string_literal);`
	Переопределяет традиционные для реализации компилятора GNAT правила именования файлов.

```
Source_Reference (
   integer_literal, string_literal);
```

Применяется для совместного использования с утилитой `gnatchop`.

```
Stream_Convert (
   [Entity =>] type_local_name,
   [Read   =>] function_name,
   [Write  =>] function name);
```

Упрощает создание подпрограмм потокового ввода/вывода для указанного типа данных

```
Style_Checks (
   string_literal | ALL_CHECKS |
   On | Off [, local_name]);
```

Указывает реализации компилятора GNAT выполнять проверку стиля оформления исходных текстов (если включено в процессе компиляции).

```
Subtitle (
   [Subtitle =>] string_literal);
```

Введена для совместимости с другими Ада-компиляторами

```
Suppress_All;
```

Введена для совместимости с другими Ада-компиляторами

```
Suppress_Initialization (
   [Entity =>] type_name);
```

Запретить инициализацию переменных указанного типа данных

```
Task_Info (expression);
```

Определяет информацию о задаче.

```
Task_Name (string_expression);
```

Определяет имя задачи.
Примечание: отсутствует в реализации GNAT V 3.13p.

```
Task_Storage (
   [Task_Type =>] local_name,
   [Top_Guard =>] static_integer_expression );
```

Определяет "охранное"пространство для задачи.

```
Time_Slice (static_duration_expression );
```

Определяет квант времени задачи (*tasking time slice*) для главной программы

```
Title ( titling_option  [,  titling  option ]);
```

Введена для совместимости с другими Ада-компиляторами

```
Unchecked_Union (first_subtype_local_name);
```

Трактовать запись как тип объединения (*union*) языка Си.

```
Unimplemented_Unit;
```

Предназначена для использования в не законченных модулях. Порождает генерацию ошибки при компиляции такого модуля.

```
Unreserve_All_Interrupts;
```

Позволяет переназначение прерываний, которые обычно обрабатываются реализацией компилятора GNAT (например `SIGINT`).

```
Unsuppress ( identifier  [, [ On =>] name]);
```

Обладает эффектом, противоположным директиве Suppress.

```
Use_VADS_Size;
```

Для старого Ада-кода атрибут 'Size эквивалентен атрибуту 'VADS_Size.

Warnings (On | Off [, local_name]);	Включает или выключает генерацию предупредительных сообщений компилятором.
Weak_External ([Entity =>] local_name);	Определяет объект, который не должен распознаваться компоновщиком.

Приложение B

Атрибуты типов

В данном приложении перечислены атрибуты типов Ады, и дано их краткое описание. Следует заметить, что при описании атрибутов типов, терминология несколько упрощена. В частности, вместо термина "подтип"используется термин "тип", что не всегда правильно (например, для вещественных типов с плавающей точкой).

B.1 Стандартно определенные атрибуты типов

Список стандартно определенных атрибутов типов Ады приводится в приложении K (*Annex K*) руководства по языку программирования Ада (RM-95).

P'**Access**	где P — любая подпрограмма. Возвращает значение ссылочного типа, которое указывает на подпрограмму P.
X'**Access**	где X: любой объект с косвенным доступом. Возвращает значение ссылочного типа, которое указывает на X.
X'Address	где X: любой объект или программный модуль. Возвращает адрес первого распределенного для хранения X элемента памяти как значение типа System.Address.
S'Adjacent	где S: любой вещественный тип с плавающей точкой. Функция которая возвращает смежное по отношению к первому параметру машинное число, в указанном вторым параметром направлении.
S'Aft	где S: любой вещественный тип с фиксированной точкой. Возвращает значение типа Universal_Integer, которое показывает требуемое число десятичных цифр после десятичной точки для обеспечения величины **delta** при представлении S.
X'Alignment	где X: любой тип или объект. Возвращает значение типа Universal_Integer, указывающее выравнивание X в памяти.
S'Base	где S: любой скалярный тип. Возвращает неограниченный базовый тип для S.
S'Bit_Order	где S: любой тип записи. Возвращает битовый порядок для S как значение типа System.Bit_Order.
P'Body_Version	где P: любой программный модуль. Возвращает строку которая идентифицирует версию тела компилируемого модуля P.
T'Callable	где T: любая задача. Возвращает значение **True**, если T может быть вызвана.

E'Caller	где E: имя любой точки входа. Возвращает значение типа Task_ID, которое идентифицирует обрабатываемую в текущий момент задачу, обратившуюся к точке входа задачи E. Использование этого атрибута допустимо только внутри инструкции принятия (для задачи-сервера) или внутри тела входа (для защищенного модуля).
S'Ceiling	где S: любой вещественный тип с плавающей точкой. Функция возвращающая наименьшее целочисленное значение, которое больше или равно ее параметру.
S'Class	где S: любой тэговый тип. Возвращает надклассовый тип для класса корнем иерархии которого будет тип S.
X'Component_Size	где X: любой объект или тип массива. Возвращает значение типа Universal_Integer которое представляет битовый размер компонентов X.
S'Compose	где S: любой вещественный тип с плавающей точкой. Функция которая возвращает значение вещественного типа с плавающей точкой, комбинируя дробную часть, которая представлена первым параметром, и порядок, который представлен вторым параметром.
A'Constrained	где A: любой тип с дискриминантом. Возвращает **True** когда A является ограниченным.
S'Copy_Sign	где S: любой вещественный тип с плавающей точкой. Функция которая возвращает значение величина которого определяется первым параметром, а знак — вторым параметром.
E'Count	где E: имя любой точки входа. Возвращает значение типа Universal_Integer, показывающее число обращений к точке входа E которые находятся в очереди.
S'Definite	где S: любой формально неопределенный тип. Возвращает **True** если фактический тип S определен.
S'**Delta**	где S: любой вещественный тип с плавающей точкой. Возвращает значение типа Universal_Real, показывающее величину значения **delta** для S
S'Denorm	где S: любой вещественный тип с плавающей точкой. Возвращает **True** если ненормализованные значения S являются машинными числами.
S'**Digits**	где S: любой тип decimal или вещественный тип с плавающей точкой. Возвращает значение типа Universal_Integer, показывающее число цифр для S.
S'Exponent	где S: любой вещественный тип с плавающей точкой. Функция возвращающая значение типа Universal_Integer, которое представляет нормализованный порядок ее параметра.
S'External_Tag	где S: любой тэговый тип. Возвращает представление S'Tag как значение типа String.
A'First (N)	где A: любой массив. Возвращает нижнюю границу диапазона N-го индекса массива A.
A'First	где A: любой массив. Возвращает нижнюю границу диапазона первого индекса массива A.
S'First	где S: любой скалярный тип. Возвращает нижнюю границу диапазона значений типа S.
R.C'First_Bit	где R.C: любой компонент C записи типа R. Возвращает значение типа Universal_Integer, которое представляет число битов до первого бита C внутри записи типа R.

S'Floor	где S: любой вещественный тип с плавающей точкой. Функция возвращающая наибольшее целочисленное значение, которое больше или равно ее параметру.
S'Fore	где S: любой вещественный тип с плавающей точкой. Возвращает значение типа Universal_Integer которое показывает минимально необходимое для представления значения S число символов до десятичной точки.
S'Fraction	где S: любой вещественный тип с плавающей точкой. Функция которая возвращает дробную часть ее параметра.
E'Identity	где E: любое исключение. Возвращает значение типа Exception_ID которое уникально идентифицирует E.
T'Identity	где T: любая задача. Возвращает значение типа Task_ID которое уникально идентифицирует T.
S'Image	где S: любой скалярный тип. Функция которая возвращает представление ее параметра как значение типа String.
S'Input	где S: любой тип. Функция которая читает и возвращает значение типа S из потока, который указывается ее параметром.
S'Class'Input	где S'Class: любой надклассовый тип. Функция которая читает из указанного как параметр потока тэг, затем, выполняет диспетчеризацию (перенаправление) вызова к подпрограмме указываемой атрибутом 'Input для типа, который идентифицируется значением прочитанного тэга, после чего, возвращает значение этого типа.
A'Last (N)	где A: любой массив. Возвращает верхнюю границу диапазона N-го индекса массива A.
A'Last	где A: любой массив. Возвращает верхнюю границу диапазона первого индекса массива A.
S'Last	где S: любой скалярный тип. Возвращает верхнюю границу диапазона значений типа S.
R.C'Last_Bit	где R.C: любой компонент C записи типа R. Возвращает значение типа Universal_Integer, которое представляет число битов до последнего бита C внутри записи типа R.
S'Leading_Part	где S: любой вещественный тип с плавающей точкой. Функция которая возвращает главную часть ее первого параметра как число системы исчисления указанной как второй параметр.
A'Length (N)	где A: любой массив. Возвращает длину N-го измерения массива A как значение типа Universal_Integer.
A'Length	где A: любой массив. Возвращает длину первого измерения массива A как значение типа Universal_Integer.
S'Machine	где S: любой вещественный тип с плавающей точкой. Функция которая возвращает ближайшее к ее параметру машинно-представляемое число.
S'Machine_Emax	где S: любой вещественный тип с плавающей точкой. Возвращает наибольший порядок для S как значение типа Universal_Integer.

S'Machine_Emin	где S: любой вещественный тип с плавающей точкой. Возвращает наименьший порядок для S как значение типа Universal_Integer.
S'Machine_Mantissa	где S: любой вещественный тип с плавающей точкой. Возвращает для машинного представления S число цифр в мантиссе как значение типа Universal_Integer.
S'Machine_Overflows	где S: любой вещественный тип. Возвращает **True** если было обнаружено переполнение или деление на ноль и возбуждено исключение Constraint_Error для каждой предопределенной операции возвращающей результат типа S.
S'Machine_Radix	где S: любой вещественный тип. Возвращает основание системы счисления для машинного представления S как значение типа Universal_Integer.
S'Machine_Rounds	где S: любой вещественный тип. Возвращает **True** если было выполнено округление неточного результата для каждой предопределенной операции возвращающей результат типа S.
S'Max	где S: любой скалярный тип. Функция которая возвращает большее значение из двух переданных ей параметров.
S'Max_Size_In_Storage_Elements	где S: любой тип. Возвращает значение типа Universal_Integer показывающее максимальное число элементов памяти занимаемых значением типа S, которое требуется при обращении к подпрограмме System.Storage_Pools.Allocate как параметр Size_In_Storage_Elements для получения значения ссылающегося на S типа.
S'Min	где S: любой скалярный тип. Функция которая возвращает меньшее значение из двух переданных ей параметров.
S'Model	где S: любой вещественный тип с плавающей точкой. Функция возвращающая образец числа значение которого будет смежным значением ее параметра.
S'Model_Emin	где S: любой вещественный тип с плавающей точкой. Возвращает образец числа соответствующий S'Model_Emin.
S'Model_Epsilon	где S: любой вещественный тип с плавающей точкой. Возвращает абсолютную разницу между 1.0 и следующим наибольшим образцом числа S как значение типа Universal_Real.
S'Model_Mantissa	где S: любой вещественный тип с плавающей точкой. Возвращает образец числа соответствующий S'Machine_Mantissa.
S'Model_Small	где S: любой вещественный тип с плавающей точкой. Возвращает наименьший положительный образец числа S как значение типа Universal_Real.
S'Modulus	где S: любой модульный тип. Возвращает основание системы счисления S как значение типа Universal_Integer.
S'Output	где S: любой тип. Процедура записывающая ее второй параметр в поток, который представлен ее первым параметром, включая любые границы или дискриминанты.

S'Class'Output	где S'Class: любой надклассовый тип. Процедура записывающая тэг ее второго параметра в поток, который указан как первый параметр, затем, выполняет диспетчеризацию (перенаправление) вызова к подпрограмме указываемой атрибутом 'Output для типа, который идентифицируется значением тэга ее второго параметра.
D'Partition_ID	где D: любое описание уровня библиотеки. Возвращает значение типа Universal_Integer которое идентифицирует раздел (*partition*) распределенной программы в которой выполнена элаборация D.
S'Pos	где S: любой дискретный тип. Функция которая возвращает номер позиции ее параметра как значение типа Universal_Integer.
R.C'Position	где R.C: любой компонент C записи типа R. То же самое, что и R.C'Address – R'Address.
S'Pred	где S: любой дискретный тип. Функция возвращающая значение у которого величина номера позиции на единицу меньше чем номер позиции ее параметра.
A'**Range** (N)	где A: любой тип массива. Эквивалентно A'First (N) .. A'Last (N), кроме случая когда вычисление A было выполнено один раз.
A'**Range**	где A: любой тип массива. Эквивалентно A'First .. A'Last, кроме случая когда вычисление A было выполнено один раз.
S'**Range**	где S: любой скалярный тип. Эквивалентно S'First .. S'Last.
S'**Read**	где S: любой тип. Процедура читающая ее второй параметр из потока, который указан ее первым параметром.
S'Class'**Read**	где S'Class: любой надклассовый тип. Процедура выполняющая диспетчеризацию (перенаправление) вызова к подпрограмме указываемой атрибутом **Read** для типа, который идентифицируется значением тэга ее второго параметра.
S'Remainder	где S: любой вещественный тип с плавающей точкой. Функция которая возвращает остаток от деления ее первого параметра на ее второй параметр.
S'Round	где S: любой вещественный тип с плавающей точкой. Функция которая возвращает округленное значение ее параметра.
S'Rounding	где S: любой вещественный тип с плавающей точкой. Функция которая возвращает ближайшее к ее параметру целое значение.
S'Safe_First	где S: любой вещественный тип с плавающей точкой. Возвращает нижнюю границу сохранного диапазона S как значение типа Universal_Real.
S'Safe_Last	где S: любой вещественный тип с плавающей точкой. Возвращает верхнюю границу сохранного диапазона S как значение типа Universal_Real.
S'Scale	где S: любой вещественный тип с плавающей точкой. Возвращает для значения S позицию точки относительно крайней правой значащей цифры как значение типа Universal_Integer.

S'Scaling	где S: любой вещественный тип с плавающей точкой. Функция которая масштабирует ее первый параметр согласно основания машинной системы исчисления возведенной в степень ее второго параметра.
S'Signed_Zeros	где S: любой вещественный тип с плавающей точкой. Возвращает **True** если аппаратное представление для S обладает возможностью представления и положительного, и отрицательного нуля.
X'Size	где X: любой тип или объект. Возвращает размер X в битах как значение типа Universal_Integer.
S'Small	где S: любой вещественный тип с плавающей точкой. Возвращает наименьшее используемое внутри типа значение для представления S как значение типа Universal_Real.
S'Storage_Pool	где S: любой ссылочный тип. Возвращает пул пространства памяти который используется для S как значение типа Root_Storage_Pool'Class.
S'Storage_Size	где S: любой ссылочный тип. Возвращает результат вызова Storage_Size (S'Storage_Pool).
T'Storage_Size	где T: любая задача. Возвращает число элементов памяти зарезервированных для T как значение типа Universal_Integer.
S'Succ	где S: любой скалярный тип. Функция возвращающая значение у которого величина номера позиции на единицу больше чем номер позиции ее параметра.
X'Tag	где X: любой тэговый тип или объект надклассового типа. Возвращает тэг X как значение типа Ada.Tags.Tag.
T'Terminated	где T: любая задача. Возвращает **True** если выполнение T завершено.
S'Truncation	где S: любой вещественный тип с плавающей точкой. Функция которая округляет ее параметр в направлении нуля.
S'Unbiased_Rounding	где S: любой вещественный тип с плавающей точкой. Функция которая возвращает ближайшее к ее параметру целочисленное значение, выполняя округление в направлении четного целого если значение параметра располагается точно между двумя целыми.
X'Unchecked_Access	где X: любой объект с косвенным доступом. Возвращает то же самое значение, что и X'**Access**, однако при этом не осуществляется проверка правильности обращения.
S'Val	где S: любой дискретный тип. Функция возвращающая значение типа S номера позиции которого равен величине ее параметра.
X'Valid	где X: любой скалярный объект. Возвращает **True** если X — это нормальный стандартный объект имеющий допустимое представление.
S'Value	где S: любой скалярный тип. Функция возвращающая значение типа S представление которого задано в виде значения типа String, игнорируя ведущие и завершающие пробелы.
P'Version	где P: любой программный модуль. Возвращает строку (значение типа String) идентифицирующую версию компилируемого модуля который содержит описание P.
S'Wide_Image	где S: любой скалярный тип. Функция которая возвращает представление ее параметра как значение типа Wide_String.

S'Wide_Value	где S: любой скалярный тип. Функция возвращающая значение типа S представление которого задано в виде значения типа Wide_String, игнорируя ведущие и завершающие пробелы.
S'Wide_Width	где S: любой скалярный тип. Возвращает максимальную длину значения типа Wide_String, возвращенного при обращении к S'Wide_Image, как значение типа Universal_Integer.
S'Width	где S: любой скалярный тип. Возвращает максимальную длину значения типа String, возвращенного при обращении к S'Image, как значение типа Universal_Integer.
S'**Write**	где S: любой тип. Процедура записывающая значение ее второго параметра в поток, который указан ее первым параметром.
S'Class'**Write**	где S'Class: любой надклассовый тип. Процедура выполняющая диспетчеризацию (перенаправление) вызова к подпрограмме указываемой атрибутом 'Write для типа, который идентифицируется значением тэга ее второго параметра.

B.2 Атрибуты типов определенные в реализации компилятора GNAT

Список атрибутов типов определенных .

Standard'Abort_Signal	где Standard — единственно возможный префикс. Предусматривает сущность для специального исключения, которая используется при принудительном завершении задачи или асинхронной передаче управления. Обычно, этот атрибут должен использоваться только в окружении задачи (атрибут достаточно специфичен и выходит за пределы нормальной семантики Ады; он предназначен для пользовательских программ перехватывающих исключение принудительного завершения).
Standard'Address_Size	где Standard — единственно возможный префикс. Статическая константа представляющая количество битов в представлении адреса ('Address). Ее первичное предназначение — построение описания Memory_Size в пакете *Standard*, однако она может быть свободно использована в программах пользователя.
'Asm_Input	Определяет функцию которая принимает два параметра. Первый параметр — это строка, а второй параметр — это выражение типа определяемое префиксом. Первый параметр (строка) должен быть статическим выражением и является ограничением для параметра (например, указывает требуемый регистр процессора). Второй аргумент это значение которое будет использоваться как аргумент ввода. Допустимые значения для констант будут такими же как и используемые в RTL, и они зависят от файла конфигурации который был использован для построения генератора кода GCC (*GCC back end*).

'Asm_Output

Определяет функцию которая принимает два параметра. Первый параметр — это строка, а второй параметр — это имя переменной определяемое префиксом. Первый параметр (строка) должен быть статическим выражением и является ограничением для параметра (например, указывает требуемый тип регистра процессора). Второй аргумент это переменная в которой будет сохранен результат. Допустимые значения для констант будут такими же как и используемые в RTL, и они зависят от файла конфигурации который был использован для построения генератора кода GCC (*GCC back end*). Если операнды вывода отсутствуют, то аргумент может быть опущен или явно указан как No_Output_Operands.

'AST_Entry

Этот атрибут реализован только для работы с OpenVMS версией GNAT.

obj'Bit

где obj: любой объект.
Возвращает битовое смещение в элементе памяти (как правило байте) в котором находится первый бит пространства размещенного для объекта obj. Возвращаемое этим атрибутом значение имеет тип Universal_Integer, величина которого всегда не отрицательна и не достигает величины System.Storage_Unit. Для объектов, которые являются переменными или константами размещаемыми в регистре возвращаемое значение — нуль (использование этого атрибута не обязывает размещать переменные в памяти). Для объектов, которые являются формальными параметрами, этот атрибут применяется как для совпадающего фактического параметра, так и для копии совпадающего фактического параметра. Для ссылочных объектов, возвращаемое значение — нуль. Примечательно, что obj.**all**'Bit является объектом проверки Access_Check для обозначенного объекта. Подобным образом, этот атрибут для компонента записи X.C'Bit является объектом проверки дискриминанта, а в случае массива X (I)'Bit или X (I1 .. I2)'Bit — объектом проверки индекса. Этот атрибут был разработан для совместимости с описанием и реализацией атрибута 'Bit в *DEC Ada 83*.

'Bit_Position

R.C'Bit где R — это запись, а C — это одно из полей типа записи, возвращает битовое смещение внутри записи, которое содержит первый бит пространства размещенного для объекта.
Возвращаемое этим атрибутом значение имеет тип Universal_Integer. Величина значения зависит только от указанного поля типа записи C и не зависит от выравнивания.

'Code_Address

Атрибут 'Address может быть применен для подпрограмм Ады, но предполагаемый эффект, согласно руководства по языку (RM-95), должен предусматривать адресное значение, которое может быть использовано для вызова подпрограммы подразумевая использование адресного выражения как в данном примере:

```
procedure K is ...

procedure L;
for L'Address use K'Address;
pragma Import (Ada, L);
```

После чего вызов L ожидается как результат вызова K. В Ada 83, где не предусматривалось ссылочных типов для подпрограмм, это был широко используемый прием для получения эффекта косвенного вызова. GNAT реализует показанное в примере выше использование атрибута 'Address, и подобные приемы работают правильно. Однако, в некоторых случаях, полезно иметь возможность получить значение адреса начала сгенерированного для подпрограммы кода. На некоторых архитектурах в этом нет необходимости, также как и в использовании приема показанного в примере выше. Например, значение 'Address может быть ссылкой на дескриптор подпрограммы, а не на саму подпрограмму. Атрибут 'Code_Address, который может быть использован только для подпрограмм, всегда возвращает адрес начала сгенерированного для указанной подпрограммы кода, который может соответствовать или не соответствовать значению которое возвращает атрибут 'Address.

Standard'Default_Bit_Order

где Standard — единственно возможный префикс.
Предусматривает значение System.Default_Bit_Order как значение 'Pos (0 для High_Order_First, и 1 для Low_Order_First). Это используется для построения описания Default_Bit_Order в пакете *System*.

U'Elaborated

где U: имя модуля.
Возвращаемое значение имеет тип **Boolean** и индицирует была ли выполнена элаборация указанного модуля. Этот атрибут первоначально был предназначен для внутреннего использования генерируемым кодом для проверки динамической элаборации, но он также может быть использован в программах пользователя. Возвращаемое значение всегда будет **True**, как только выполнена элаборация всех модулей.

P'Elab_Body

где P: имя программного модуля.
Возвращает объекты для соответствующей процедуры элаборации для выполнения элаборации тела указанного модуля. Атрибут используется в главной процедуре элаборации редактора связей (*binder*) и, обычно, не должен использоваться в любом другом контексте. Однако, могут возникнуть какие-либо особенные ситуации в которых использование этого атрибута может оказаться полезным для получения возможности вызвать процедуру элаборации из кода Ады (например, в случае необходимости выполнения выборочной переэлаборации для исправления какой-либо ошибки).

P'Elab_Spec

где P: имя программного модуля.
Возвращает объекты для соответствующей процедуры элаборации для выполнения элаборации спецификации указанного модуля. Атрибут используется в главной процедуре элаборации редактора связей (*binder*) и, обычно, не должен использоваться в любом другом контексте. Однако, могут возникнуть какие-либо особенные ситуации в которых использование этого атрибута может оказаться полезным для получения возможности вызвать процедуру элаборации из кода Ады (например, в случае необходимости выполнения выборочной переэлаборации для исправления какой-либо ошибки).

'Emax

Этот атрибут предусмотрен для совместимости с Ada 83.

S'Enum_Rep

где S: перечислимый тип, объект перечислимого типа или несовмещенный перечислимый литерал.
Для перечислимого типа обозначает функцию, которая имеет следующую спецификацию:

```
function S'Enum_Rep (Arg : S'Base)
  return Universal_Integer;
```

Для объекта перечислимого типа или несовмещенного перечислимого литерала S'Enum_Rep эквивалентен T'Enum_Rep (S) где T — это тип перечислимого литерала или объекта S.

Функция возвращает значение внутреннего представления для указанного перечислимого значения, которое будет равно значению возвращаемому атрибутом 'Pos если отсутствует спецификация представления. Этот атрибут статический (то есть, результат статический, если аргумент статический). Атрибут S'Enum_Rep также может быть использован для целочисленных типов и объектов. В таких случаях, он просто возвращает целочисленное значение. Смысл этого в том, чтобы разрешить его использование для дискретных формальных аргументов (<>) в настраиваемых модулях, которые могут быть конкретизированы как с перечислимыми типами, так и с целочисленными типами.

'Epsilon

Предусмотрен для обеспечения совместимости с Ada 83.

S'Fixed_Value

где S: вещественный тип с фиксированной точкой.
Обозначает функцию, которая имеет следующую спецификацию:

```
function S'Fixed_Value
    (Arg : Universal_Integer)
                    return S;
```

Возвращаемое значение V — это вещественное значение с фиксированной точкой, подобное:

```
V = Arg * S'Small
```

Таким образом, это эквивалентно: сначала, преобразованию аргумента в значение целочисленного типа, используемого для представления S, а затем, выполнению непроверяемого преобразования в вещественный тип с фиксированной точкой. Первоначально, этот атрибут был предназначен для реализации функций ввода/вывода для значений вещественного типа с фиксированной точкой.

T'Has_Discriminants

где T: тип.
Возвращенное значение типа **Boolean** будет **True** в случае когда тип T имеет дискриминант, и **False** — в обратном случае. Предполагается использование этого атрибута совместно с описаниями настраиваемых модулей. Если атрибут используется с приватным настраиваемым типом, то он индицирует имеет ли фактический тип дискриминант.

'Img

Этот атрибут отличается от стандартного атрибута 'Image тем, что он может быть использован как с объектами, так и с типами. В обоих случаях он возвращает 'Image для подтипа объекта. Это удобно для отладки:

```
Put_Line ("X = " & X'Img);
```

будет иметь такой же смысл, что и более "многословная"запись:

```
Put_Line ("X = " & type'Image (X));
```

где *type* — это подтип объекта X.

S'Integer_Value

где S: целочисленный тип.
Обозначает функцию, которая имеет следующую спецификацию:

```
function S'Integer_Value
    (Arg : Universal_Fixed)
                return S;
```

Возвращаемое значение V подобно:

```
Arg = V * type'Small
```

Таким образом, это эквивалентно: сначала, выполнению непроверяемого преобразования из вещественного типа с фиксированной точкой в соответствующий ему тип реализации, а затем, преобразование результата в целочисленный тип назначения. Первоначально этот атрибут был предназначен для использования в функциях стандартного ввода/вывода для вещественных типов с фиксированной точкой.

'Large

Предусмотрен для обеспечения совместимости с Ada 83.

'Machine_Size

Этот атрибут идентичен атрибуту 'Object_Size и предусмотрен для совместимости с одноименным атрибутом компилятора *DEC Ada 83*.

'Mantissa

Предусмотрен для обеспечения совместимости с Ada 83.

Standard'Max_Interrupt_Priority

где Standard — единственно возможный префикс.
Предоставляет значение System.Max_Interrupt_Priority и первоначально предназначен для построения этого описания в пакете *System*.

Standard'Max_Priority

где Standard — единственно возможный префикс.
Предоставляет значение System.Max_Priority и первоначально предназначен для построения этого описания в пакете *System*.

Standard'Maximum_Alignment

где Standard — единственно возможный префикс.
Предоставляет максимальное пригодное значение выравнивания для целевой платформы. Это статическое значение которое может быть использовано для указания требуемого выравнивания объектов. При этом во всех случаях будет гарантироваться правильность выравнивания объектов Это может быть полезно при импортировании внешних объектов, когда требования для их выравнивания не известны.

'Mechanism_Code

function'Mechanism_Code возвращает целочисленный код, который определяет используемый механизм для передачи результата вызова функции **function**, а subprogram'Mechanism_Code (n) возвращает целочисленный код, который определяет используемый механизм для передачи формального параметра с порядковым номером n (статическое целочисленное значение, 1 — подразумевает первый параметр) подпрограммы subprogram.
Смысл возвращаемых кодов следующий:

- **1** — по значению (*by copy / value*)
- **2** — по ссылке (*by reference*)
- **3** — по дескриптору (*by descriptor / default descriptor class*)
- **4** — по дескриптору (*by descriptor / UBS: unaligned bit string*)
- **5** — по дескриптору (*by descriptor / UBSB: aligned bit string with arbitrary bounds*)
- **6** — по дескриптору (*by descriptor / UBA: unaligned bit array*)
- **7** — по дескриптору (*by descriptor / S: string, also scalar access type parameter*)
- **8** — по дескриптору (*by descriptor / SB: string with arbitrary bounds*)
- **9** — по дескриптору (*by descriptor / A: contiguous array*)
- **10** — по дескриптору (*by descriptor / NCA: non-contiguous array*)

Значения 3-10 справедливы только для *Digital OpenVMS* реализации GNAT.

'Null_Parameter

Ссылка T'Null_Parameter обозначает воображаемый объект типа T который размещается по нулевому машинному адресу. Этот атрибут допустим только для выражений в формальных параметрах по умолчанию или как фактическое выражение вызова подпрограммы. В обоих случаях подпрограмма должна быть импортированной. Идентичность объекта представляется нулевым адресом в списке аргументов, вне зависимости от механизма передачи (явного или используемого по умолчанию). Такая возможность необходима для спецификации того, что нулевой адрес должен быть передан для записи или другого составного объекта передаваемого по ссылке. Другого способа такой индикации, без использования атрибута 'Null_Parameter, не существует.

'Object_Size

Размер какого-либо объекта не обязан быть таким же самым как размер типа объекта, поскольку размер объекта по умолчанию увеличен в соответствии с выравниванием объекта. Например, Natural'Object_Size равен 31 бит, а размер объекта типа Natural по умолчанию будет иметь размер 32 бита. Подобным образом, запись которая содержит **Integer** и Character:

```
type Rec is record
   I : Integer;
   C : Character;
end record;
```

будет иметь размер 40 бит (Rec'Size будет 40 бит; выравнивание будет 4, из-за поля типа **Integer**; таким образом размер записи этого типа по умолчанию будет 64 бита или 8 байт).
Атрибут T'Object_Size был добавлен к реализации GNAT, чтобы обеспечить легкое определение размера объекта типа T по умолчанию. Например Natural'Object_Size равен 32 бита, а Rec'Object_Size (для записи из примера) будет 64. Примечательно также то, что в отличие от ситуации с атрибутом 'Size который описан в руководстве по языку, атрибут 'Object_Size может быть определен индивидуально для различных подтипов. Например:

```
type R is new Integer;
subtype R1 is R range 1 .. 10;
subtype R2 is R range 1 .. 10;
for R2'Object_Size use 8;
```

В этом примере R'Object_Size и R1'Object_Size, оба равны 32, поскольку размер объекта по умолчанию для подтипа такой же как и размер для подтипа-предка. Это подразумевает, что размер по умолчанию для объектов с типами R и R1 будет 32 бита (4 байта). Но объекты типа R2 будут иметь только 8 бит (1 байт), поскольку R2'Object_Size был установлен в 8.

T'Passed_By_Reference

где T: тип.
Возвращает значение типа **Boolean** которое равно **True** если тип T может быть нормально передан по ссылке и **False** если тип T может быть нормально передан по значению при вызове. Для скалярных типов результат всегда будет **False** и является статическим. Для не скалярных типов, результат — не статический.

T'Range_Length

где T: любой дискретный тип.
Возвращает число значений представляемых типом (нуль, в случае пустого диапазона значений). Результат статический для статических типов. Этот атрибут применяется для типов индекса одномерных массивов и дает те же результаты, что и атрибут 'Range для того же массива.

'Safe_Emax

Предусмотрен для обеспечения совместимости с Ada 83.

'Safe_Large

Предусмотрен для обеспечения совместимости с Ada 83.

'Small

Этот атрибут предусмотрен в Ada-95 только для вещественных типов с фиксированной точкой. GNAT, для обеспечения совместимости с Ada 83, допускает использование этого атрибута для вещественных типов с плавающей точкой.

Standard'Storage_Unit

где Standard — единственно возможный префикс.
Предоставляет значение System.Storage_Unit и первоначально предназначен для построения этого описания в пакете *System*.

Standard'Tick

где Standard — единственно возможный префикс.
Предоставляет значение System.Tick и первоначально предназначен для построения этого описания в пакете *System*.

System'To_Address

где System — единственно возможный префикс.
Обозначает функцию которая идентична System.Storage_Elements.To_Address с тем исключением, что это статический атрибут. Это подразумевает, что подобное выражение может быть использовано в контексте (например, при преэлаборации пакета) который требует использования статических выражений и где не может быть использован вызов функции (поскольку вызов функции всегда является не статическим, даже при статических аргументах).

T'Type_Class

где T: любой тип.
Возвращает значение класса типа для типа T. Если тип T является настраиваемым типом, то возвращаемое значение будет значением соответствующего фактического типа. Значение этого атрибута имеет тип System.Aux_DEC.Type_Class, который имеет следующее описание:

```
type Type_Class is
  (Type_Class_Enumeration,
   Type_Class_Integer,
   Type_Class_Fixed_Point,
   Type_Class_Floating_Point,
   Type_Class_Array,
   Type_Class_Record,
   Type_Class_Access,
   Type_Class_Task,
   Type_Class_Address);
```

Защищенные типы возвращают значение Type_Class_Task, которое, таким образом, подходит для всех многозадачных типов. Этот атрибут был разработан для обеспечения совместимости с одноименным атрибутом компилятора *DEC Ada 83*.

P'UET_Address

где P: библиотечный пакет.
Возвращает адрес таблицы исключений модуля при использовании обработки исключений без потерь производительности. Этот атрибут предназначен для использования только с реализацией GNAT. См. модуль *Ada.Exceptions* в файлах "`a-except.ads`"и "`a-except.adb`" для получения более полной информации о том как эти атрибуты используются в реализации.

N'Universal_Literal_String

где N: именованное число.
Возвращает статический результат, которым является строка символов представляющая число так, как оно было описано в оригинальном исходном тексте. Это позволяет пользователю программы получить доступ к фактическому тексту описания именованных чисел без промежуточного преобразования и без необходимости заключения строк в кавычки (которые препятствуют использование строк как чисел). Это имеет внутреннее использование для построения значений вещественных атрибутов с плавающей точкой из файла "`ttypef.ads`", но может быть также использовано в программах пользователя.

'Unrestricted_Access	Этот атрибут подобен атрибуту 'Access, за исключением того, что отсутствуют проверка на доступность и проверка представления косвенного доступа. Ответственность за правильность использования полностью возлагается на пользователя. Это подобно атрибуту 'Addess, для которого необходима замена в тех случаях, когда требуемое значение должно иметь ссылочный тип. Другими словами, эффект действия этого атрибута идентичен первоначальному получению атрибута 'Addess с последующим непроверяемым преобразованием в требуемое значение ссылочного типа. В GNAT, но не обязательно в других реализациях, использование статической цепочки для подпрограмм внутреннего уровня подразумевает, что 'Unrestricted_Access, примененный для подпрограммы, возвращает значение, которое может быть вызвано пока подпрограмма находится в области действия (видимости). Нормальные правила доступа/видимости в Ada 95 ограничивают такое использование.
'VADS_Size	Для обеспечения совместимости с компилятором *VADS Ada 83*.
T'Value_Size	где T: тип. Возвращает число бит, которое требуется для представления значений указанного типа T. Это то же самое, что и T'Size, но в отличие от 'Size, может быть установлено для не первого подтипа.
Standard'Wchar_T_Size	где Standard — единственно возможный префикс. Предоставляет битовый размер типа wchar_t языка *C* и первоначально предназначен для построения этого типа в пакете *Interfaces.C*. **Примечание**: отсутствует в реализации GNAT V 3.13p.
Standard'Word_Size	где Standard — единственно возможный префикс. Предоставляет значение System.Word_Size и первоначально предназначен для построения этого описания в пакете *System*.

Приложение C

Спецификация пакета *System*

```
package System is
pragma Pure (System);

   type Name is Определяемый_Реализацией_Перечислимый_Тип;
   System_Name : constant Name := Определяется_Реализацией;


   -- Системно-зависимые именованные числа

   Min_Int               : constant := Root_Integer'First;
   Max_Int               : constant := Root_Integer'Last;

   Max_Binary_Modulus    : constant := Определяется_Реализацией;
   Max_Nonbinary_Modulus : constant := Определяется_Реализацией;

   Max_Base_Digits       : constant := Root_Real'Digits;
   Max_Digits            : constant := Определяется_Реализацией;

   Max_Mantissa          : constant := Определяется_Реализацией;
   Fine_Delta            : constant := Определяется_Реализацией;

   Tick                  : constant := Определяется_Реализацией;


   -- Описания относящиеся к хранению информации в памяти

   type Address is Определяется_Реализацией;   -- обычно, приватный тип
   Null_Address : constant Address;

   Storage_Unit          : constant := Определяется_Реализацией;
   Word_Size             : constant := Определяется_Реализацией * Storage_Unit;
   Memory_Size           : constant := Определяется_Реализацией;


   -- Сравнение адресов

   function "<"  (Left, Right : Address) return Boolean;
   function "<=" (Left, Right : Address) return Boolean;
   function ">"  (Left, Right : Address) return Boolean;
   function ">=" (Left, Right : Address) return Boolean;
   function "="  (Left, Right : Address) return Boolean;


   -- Другие системно-зависимые описания
```

```
    type Bit_Order is (High_Order_First, Low_Order_First);
    Default_Bit_Order : constant Bit_Order;



    -- Описания относящиеся к приоритетам (RM D.1)

    subtype Any_Priority is Integer range Определяется_Реализацией;

    subtype Priority is Any_Priority
        range Any_Priority'First..Определяется_Реализацией;

    subtype Interrupt_Priority is Any_Priority
        range Priority'Last + 1 .. Any_Priority'Last;

    Default_Priority : constant Priority :=
        (Priority'First + Priority'Last) / 2;

private

    -- Стандартом языка не определено

end System;
```

Приложение D

Спецификация пакета *Standard*

Пакет *Standard* всегда находится в области видимости, поэтому все описания этого пакета всегда непосредственно доступны.

```
package Standard is
pragma Pure(Standard);

   type Boolean is (False, True);

   -- предопределенные знаки операций сравнения для этого типа следующие:
   --    function "="  (Left, Right : Boolean) return Boolean;
   --    function "/=" (Left, Right : Boolean) return Boolean;
   --    function "<"  (Left, Right : Boolean) return Boolean;
   --    function "<=" (Left, Right : Boolean) return Boolean;
   --    function ">"  (Left, Right : Boolean) return Boolean;
   --    function ">=" (Left, Right : Boolean) return Boolean;
   --
   -- предопределенные знаки логических операций для этого типа следующие:
   --    function "and"  (Left, Right : Boolean) return Boolean;
   --    function "or"   (Left, Right : Boolean) return Boolean;
   --    function "xor"  (Left, Right : Boolean) return Boolean;
   --    function "not"  (Right : Boolean) return Boolean;



   type Integer is range Определяется_Реализацией;

   subtype Natural  is Integer range 0 .. Integer'Last;
   subtype Positive is Integer range 1 .. Integer'Last;

   -- предопределенные знаки операций для этого типа следующие:
   --    function "="  (Left, Right : Integer'Base) return Boolean;
   --    function "/=" (Left, Right : Integer'Base) return Boolean;
   --    function "<"  (Left, Right : Integer'Base) return Boolean;
   --    function "<=" (Left, Right : Integer'Base) return Boolean;
   --    function ">=" (Left, Right : Integer'Base) return Boolean;
   --    function ">"  (Left, Right : Integer'Base) return Boolean;
   --
   --    function "+"   (Right : Integer'Base) return Integer'Base;
   --    function "-"   (Right : Integer'Base) return Integer'Base;
   --    function "abs" (Right : Integer'Base) return Integer'Base;
   --
   --    function "+"   (Left, Right : Integer'Base) return Integer'Base;
   --    function "-"   (Left, Right : Integer'Base) return Integer'Base;
   --    function "*"   (Left, Right : Integer'Base) return Integer'Base;
   --    function "/"   (Left, Right : Integer'Base) return Integer'Base;
   --    function "rem" (Left, Right : Integer'Base) return Integer'Base;
   --    function "mod" (Left, Right : Integer'Base) return Integer'Base;
   --
```

```
--    function "**" (Left: Integer'Base; Right: Natural) return Integer'Base;


type Float              is digits Определяется_Реализацией;

-- предопределенные знаки операций для этого типа следующие:
--    function "="  (Left, Right : Float) return Boolean;
--    function "/=" (Left, Right : Float) return Boolean;
--    function "<"  (Left, Right : Float) return Boolean;
--    function "<=" (Left, Right : Float) return Boolean;
--    function ">=" (Left, Right : Float) return Boolean;
--    function ">"  (Left, Right : Float) return Boolean;
--
--    function "+"   (Right : Float) return Float;
--    function "-"   (Right : Float) return Float;
--    function "abs" (Right : Float) return Float;
--
--    function "+"   (Left, Right : Float) return Float;
--    function "-"   (Left, Right : Float) return Float;
--    function "*"   (Left, Right : Float) return Float;
--    function "/"   (Left, Right : Float) return Float;
--
--    function "**" (Left: Float; Right: Integer'Base) return Float;
--
--
-- дополнительные знаки операций предопределены для корневых численных типов
-- Root_Integer и Root_Real
--
--    function "*" (Left : Root_Integer; Right : Root_Real) return Root_Real;
--    function "*" (Left : Root_Real; Right : Root_Integer) return Root_Real;
--    function "/" (Left : Root_Real; Right : Root_Integer) return Root_Real;
--
--
-- тип Universal_Fixed - является предопределенным
-- для него предопределены только следующие знаки операций:
--
--    function "*" (Left : Universal_Fixed; Right : Universal_Fixed)
--                                  return Universal_Fixed;
--    function "/" (Left : Universal_Fixed; Right : Universal_Fixed)
--                                  return Universal_Fixed;




-- Описание символьного типа Character основано
-- на множестве символов описанных стандартом  ISO 8859-1.
--
-- Символьные литералы которые соответствуют
-- позициям управляющих символов отсутствуют.
-- Они описываются идентификаторами
-- которые записаны в нижнем регистре.

type Character is

(nul,  soh,  stx,  etx,   eot,  enq,  ack,  bel,   --   0 (16#00#) .. (16#07#)
 bs,   ht,   lf,   vt,    ff,   cr,   so,   si,    --   8 (16#08#) .. (16#0F#)

 dle,  dc1,  dc2,  dc3,   dc4,  nak,  syn,  etb,   -- 16 (16#10#) .. (16#17#)
 can,  em,   sub,  esc,   fs,   gs,   rs,   us,    -- 24 (16#18#) .. (16#1F#)

 ' ',  '!',  '"',  '#',   '$',  '%',  '&',  ''',   -- 32 (16#20#) .. (16#27#)
 '(',  ')',  '*',  '+',   ',',  '-',  '.',  '/',   -- 40 (16#28#) .. (16#2F#)
```

```
'0',  '1',  '2',  '3',   '4',  '5',  '6',  '7',   -- 48 (16#30#) .. (16#37#)
'8',  '9',  ':',  ';',   '<',  '=',  '>',  '?',   -- 56 (16#38#) .. (16#3F#)

'@',  'A',  'B',  'C',   'D',  'E',  'F',  'G',   -- 64 (16#40#) .. (16#47#)
'H',  'I',  'J',  'K',   'L',  'M',  'N',  'O',   -- 72 (16#48#) .. (16#4F#)

'P',  'Q',  'R',  'S',   'T',  'U',  'V',  'W',   -- 80 (16#50#) .. (16#57#)
'X',  'Y',  'Z',  '[',   '\',  ']',  '^',  '_',   -- 88 (16#58#) .. (16#5F#)

'`',  'a',  'b',  'c',   'd',  'e',  'f',  'g',   -- 96 (16#60#) .. (16#67#)
'h',  'i',  'j',  'k',   'l',  'm',  'n',  'o',   -- 104 (16#68#) .. (16#6F#)

'p',  'q',  'r',  's',   't',  'u',  'v',  'w',   -- 112 (16#70#) .. (16#77#)
'x',  'y',  'z',  '{',   '|',  '}',  '~',  del,   -- 120 (16#78#) .. (16#7F#)

 reserved_128,  reserved_129,  bph,  nbh,         -- 128 (16#80#) .. (16#83#)
 reserved_132,  nel,     ssa,  esa,               -- 132 (16#84#) .. (16#87#)

 hts,  htj,  vts,  pld,   plu,  ri,   ss2,  ss3,  -- 136 (16#88#) .. (16#8F#)

 dcs,  pu1,  pu2,  sts,   cch,  mw,   spa,  epa,  -- 144 (16#90#) .. (16#97#)

 sos, reserved_153, sci, csi,                     -- 152 (16#98#) .. (16#9B#)
 st,   osc,  pm,   apc,                           -- 156 (16#9C#) .. (16#9F#)

... );


-- Описание символьного типа Wide_Character основано
-- на множестве символов описанных стандартом  ISO 10646 BMP.
--
-- Первые 256 символьных позиций соответствуют содержимому
-- символьного типа Character

type Wide_Character is (nul, soh ... FFFE, FFFF);


-- Пакет ASCII считается устаревшим

package ASCII is

   -- Управляющие символы:

   NUL  : constant Character := Character'Val (16#00#);
   SOH  : constant Character := Character'Val (16#01#);
   STX  : constant Character := Character'Val (16#02#);
   ETX  : constant Character := Character'Val (16#03#);
   EOT  : constant Character := Character'Val (16#04#);
   ENQ  : constant Character := Character'Val (16#05#);
   ACK  : constant Character := Character'Val (16#06#);
   BEL  : constant Character := Character'Val (16#07#);
   BS   : constant Character := Character'Val (16#08#);
   HT   : constant Character := Character'Val (16#09#);
   LF   : constant Character := Character'Val (16#0A#);
   VT   : constant Character := Character'Val (16#0B#);
   FF   : constant Character := Character'Val (16#0C#);
   CR   : constant Character := Character'Val (16#0D#);
   SO   : constant Character := Character'Val (16#0E#);
   SI   : constant Character := Character'Val (16#0F#);
   DLE  : constant Character := Character'Val (16#10#);
   DC1  : constant Character := Character'Val (16#11#);
   DC2  : constant Character := Character'Val (16#12#);
```

```
DC3    : constant Character := Character'Val (16#13#);
DC4    : constant Character := Character'Val (16#14#);
NAK    : constant Character := Character'Val (16#15#);
SYN    : constant Character := Character'Val (16#16#);
ETB    : constant Character := Character'Val (16#17#);
CAN    : constant Character := Character'Val (16#18#);
EM     : constant Character := Character'Val (16#19#);
SUB    : constant Character := Character'Val (16#1A#);
ESC    : constant Character := Character'Val (16#1B#);
FS     : constant Character := Character'Val (16#1C#);
GS     : constant Character := Character'Val (16#1D#);
RS     : constant Character := Character'Val (16#1E#);
US     : constant Character := Character'Val (16#1F#);
DEL    : constant Character := Character'Val (16#7F#);

-- Остальные символы:

Exclam      : constant Character := '!';
Quotation   : constant Character := '"';
Sharp       : constant Character := '#';
Dollar      : constant Character := '$';
Percent     : constant Character := '%';
Ampersand   : constant Character := '&';
Colon       : constant Character := ':';
Semicolon   : constant Character := ';';
Query       : constant Character := '?';
At_Sign     : constant Character := '@';
L_Bracket   : constant Character := '[';
Back_Slash  : constant Character := '\';
R_Bracket   : constant Character := ']';
Circumflex  : constant Character := '^';
Underline   : constant Character := '_';
Grave       : constant Character := '`';
L_Brace     : constant Character := '{';
Bar         : constant Character := '|';
R_Brace     : constant Character := '}';
Tilde       : constant Character := '~';

-- Буквы нижнего регистра:

LC_A : constant Character := 'a';
LC_B : constant Character := 'b';
LC_C : constant Character := 'c';
LC_D : constant Character := 'd';
LC_E : constant Character := 'e';
LC_F : constant Character := 'f';
LC_G : constant Character := 'g';
LC_H : constant Character := 'h';
LC_I : constant Character := 'i';
LC_J : constant Character := 'j';
LC_K : constant Character := 'k';
LC_L : constant Character := 'l';
LC_M : constant Character := 'm';
LC_N : constant Character := 'n';
LC_O : constant Character := 'o';
LC_P : constant Character := 'p';
LC_Q : constant Character := 'q';
LC_R : constant Character := 'r';
LC_S : constant Character := 's';
LC_T : constant Character := 't';
LC_U : constant Character := 'u';
LC_V : constant Character := 'v';
LC_W : constant Character := 'w';
LC_X : constant Character := 'x';
```

```
        LC_Y : constant Character := 'y';
        LC_Z : constant Character := 'z';

    end ASCII;



    -- Предопределенные строковые типы:

    type String is array (Positive range <>) of Character;
    pragma Pack (String);

    -- предопределенные знаки операций для этого типа следующие:
    --    function "="  (Left, Right : String) return Boolean;
    --    function "/=" (Left, Right : String) return Boolean;
    --    function "<"  (Left, Right : String) return Boolean;
    --    function "<=" (Left, Right : String) return Boolean;
    --    function ">"  (Left, Right : String) return Boolean;
    --    function ">=" (Left, Right : String) return Boolean;
    --
    --    function "&" (Left : String,    Right : String)    return String;
    --    function "&" (Left : Character, Right : String)    return String;
    --    function "&" (Left : String,    Right : Character) return String;
    --    function "&" (Left : Character, Right : Character) return String;

    type Wide_String is array (Positive range <>) of Wide_Character;
    pragma Pack (Wide_String);

    -- для типа Wide_String предопределены такие же знаки операций
    -- что и для типа String



    type Duration is delta Определяется_Реализацией
        range Определяется_Реализацией;

    -- для типа Duration предопределены такие же знаки операций
    -- что и для любого вещественного типа с фиксированной точкой.



    -- Предопределенные исключения:

    Constraint_Error : exception;
    Program_Error    : exception;
    Storage_Error    : exception;
    Tasking_Error    : exception;

end Standard;
```

Приложение E

Спецификации пакетов ввода/вывода

E.1 Пакеты текстового ввода/вывода

E.1.1 Пакет *Ada.Text_IO*

```
with Ada.IO_Exceptions;     -- описано в приложении A (Аппех A)
                            -- руководства по языку (RM-95)

package Ada.Text_IO is
pragma Elaborate_Body (Text_IO);

   type File_Type is limited private;                    -- внутреннее представление
                                                         --    файла для программы

   type File_Mode is (In_File, Out_File, Append_File);   -- управляет направлением
                                                         --    передачи данных
                                                         --    (в/из файла)



   type Count is range 0 .. Определяется_Реализацией;
   subtype Positive_Count is Count range 1 .. Count'Last;

   Unbounded : constant Count := 0;                      -- длина строки и страницы


   subtype Field is Integer range 0 .. Определяется_Реализацией;

   subtype Number_Base is Integer range 2 .. 16;         -- используются только:
                                                         --     2, 8, 10 и 16

   type Type_Set is (Lower_Case, Upper_Case);            -- для перечислимых типов



   -- Подпрограммы управления файлами

   procedure Create
     (File : in out File_Type;          -- ассоциируемая с внешним файлом
                                        --     переменная в программе
      Mode : in File_Mode := Out_File;  -- по умолчанию - файл для вывода данных
      Name : in String := "";           -- имя внешнего файла
      Form : in String := "");          -- использование, не определено стандартом

   procedure Open
     (File : in out File_Type;
      Mode : in File_Mode;
      Name : in String;
```

```
   Form : in String := "");

procedure Close  (File : in out File_Type);
procedure Delete (File : in out File_Type);
procedure Reset  (File : in out File_Type; Mode : in File_Mode); -- меняет режим
                                                                 --   доступа
                                                                 --   к файлу
procedure Reset  (File : in out File_Type);

function Mode (File : in File_Type) return File_Mode;  -- режим доступа к файлу
function Name (File : in File_Type) return String;     -- имя внешнего файла
function Form (File : in File_Type) return String;     -- зависит от реализации

function Is_Open (File : in File_Type) return Boolean;



-- Подпрограммы управления файлами ввода, вывода и вывода ошибок
--    устанавливаемыми по умолчанию

procedure Set_Input  (File : in File_Type);  -- установить как файл ввода
procedure Set_Output (File : in File_Type);  -- установить как файл вывода
procedure Set_Error  (File : in File_Type);  -- установить как файл вывода ошибок
                      -- ПРИМЕЧАНИЕ: перед вызовом подпрограмм Set_Input,
                      --             Set_Output и/или Set_Error,
                      --             файл File должен быть открыт

function Standard_Input  return File_Type;   -- обычно, клавиатура
function Standard_Output return File_Type;   -- обычно, видеодисплей
function Standard_Error  return File_Type;

function Current_Input  return File_Type;    -- обычно, то же что и Standard_Input
function Current_Output return File_Type;
function Current_Error  return File_Type;


type File_Access is access constant File_Type; -- ссылочный тип (указатель)
                                               --    позволяющий ссылаться
                                               --    на переменные внутреннего
                                               --    представления файла, что
                                               --    во многих случаях удобно
                                               -- (введен стандартом Ada-95)

function Standard_Input  return File_Access;
function Standard_Output return File_Access;
function Standard_Error  return File_Access;

function Current_Input  return File_Access;
function Current_Output return File_Access;
function Current_Error  return File_Access;



-- Подпрограммы управления буферами

procedure Flush (File : in out File_Type);
procedure Flush;



-- Подпрограммы спецификации длин строк и страниц для ввода/вывода

procedure Set_Line_Length (File : in File_Type; To : in Count);
procedure Set_Line_Length (To : in Count);
```

```
procedure Set_Page_Length (File : in File_Type; To : in Count);
procedure Set_Page_Length (To : in Count);

function Line_Length (File : in File_Type) return Count;
function Line_Length return Count;

function Page_Length (File : in File_Type) return Count;
function Page_Length return Count;



-- Подпрограммы спецификации Column, Line и Page (колонка, строка и страница)

procedure New_Line (File : in File_Type;                -- вывод терминатора строки
                    Spacing : in Positive_Count := 1); -- по умолчанию, один раз
procedure New_Line (Spacing : in Positive_Count := 1);

procedure Skip_Line (File : in File_Type;               -- отбросить ввод символов
                     Spacing : in Positive_Count := 1); -- по умолчанию, для одной
procedure Skip_Line (Spacing : in Positive_Count := 1);-- строки

function End_Of_Line (File : in File_Type) return Boolean;
function End_Of_Line return Boolean;

procedure New_Page (File : in File_Type);   -- завершить текущую страницу
procedure New_Page;                         -- выводом терминатора страницы

procedure Skip_Page (File : in File_Type);  -- отбросить ввод символов
procedure Skip_Page;                        -- вплоть до терминатора страницы


function End_Of_Page (File : in File_Type) return Boolean;
function End_Of_Page return Boolean;

function End_Of_File (File : in File_Type) return Boolean;
function End_Of_File return Boolean;


procedure Set_Col (File : in File_Type;  To : in Positive_Count);
procedure Set_Col (To : in Positive_Count);

procedure Set_Line (File : in File_Type; To : in Positive_Count);
procedure Set_Line (To : in Positive_Count);

function Col (File : in File_Type) return Positive_Count;
function Col return Positive_Count;

function Line (File : in File_Type) return Positive_Count;
function Line return Positive_Count;

function Page (File : in File_Type) return Positive_Count;
function Page return Positive_Count;



-- Посимвольный ввод/вывод

procedure Get (File : in File_Type; Item : out Character);
procedure Get (Item : out Character);
procedure Put (File : in File_Type; Item : in Character);
procedure Put (Item : in Character);

procedure Look_Ahead
```

```
   (File         : in File_Type;
    Item         : out Character;
    End_Of_Line  : out Boolean);

procedure Look_Ahead
   (Item         : out Character;
    End_Of_Line  : out Boolean);

procedure Get_Immediate
   (File : in File_Type;
    Item : out Character);

procedure Get_Immediate
   (Item : out Character);

procedure Get_Immediate
   (File       : in File_Type;
    Item       : out Character;
    Available  : out Boolean);

procedure Get_Immediate
   (Item       : out Character;
    Available  : out Boolean);



-- Построчный ввод/вывод

procedure Get (File : in File_Type; Item : out String);
procedure Get (Item : out String);
procedure Put (File : in File_Type; Item : in String);
procedure Put (Item : in String);

procedure Get_Line
   (File : in File_Type;
    Item : out String;
    Last : out Natural);

procedure Get_Line
   (Item : out String;
    Last : out Natural);

procedure Put_Line
   (File : in File_Type;
    Item : in String);

procedure Put_Line
   (Item : in String);



-- Исключения

Status_Error : exception renames IO_Exceptions.Status_Error;
Mode_Error   : exception renames IO_Exceptions.Mode_Error;
Name_Error   : exception renames IO_Exceptions.Name_Error;
Use_Error    : exception renames IO_Exceptions.Use_Error;
Device_Error : exception renames IO_Exceptions.Device_Error;
End_Error    : exception renames IO_Exceptions.End_Error;
Data_Error   : exception renames IO_Exceptions.Data_Error;
Layout_Error : exception renames IO_Exceptions.Layout_Error;
```

```
private

   . . .    -- стандартом языка не определено

end Ada.Text_IO;
```

E.1.2 Пакет *Ada.Text_IO.Integer_IO*

```
generic
   type Num is range <>;  -- параметр настройки модуля

package Ada.Text_IO.Integer_IO is

   Default_Width : Field := Num'Width;  -- ширина числа в символах
   Default_Base  : Number_Base := 10;   -- устанавливает основание системы счисления
                                        --    принимаемое по умолчанию для всех
                                        --    операций ввода/вывода

   procedure Get                        -- ввод числа из указанного файла
     (File  : in File_Type;
      Item  : out Num;                  -- тип соответствует формальному параметру настройки
      Width : in Field := 0);           -- может быть использовано для точного указания
                                        --   количества вводимых символов

   procedure Get                        -- ввод числа из файла стандартного ввода
     (Item  : out Num;
      Width : in Field := 0);

   procedure Put                        -- вывод числа в указанный файл
     (File  : in File_Type;
      Item  : in Num;
      Width : in Field := Default_Width;
      Base  : in Number_Base := Default_Base);

   procedure Put                        -- вывод числа в файл стандартного вывода
     (Item  : in Num;
      Width : in Field := Default_Width;
      Base  : in Number_Base := Default_Base);

   procedure Get                        -- ввод числа из строки
                                        --   (преобразование из строкового представления)
     (From : in String;
      Item : out Num;
      Last : out Positive);             -- индекс последнего преобразуемого символа в From

   procedure Put                        -- вывод числа в строку
                                        --   (преобразование в строковое представление)
     (To   : out String;
      Item : in Num;
      Base : in Number_Base := Default_Base);

end Ada.Text_IO.Integer_IO;
```

E.1.3 Пакет *Ada.Text_IO.Modular_IO*

```
generic
   type Num is mod <>;  -- параметр настройки модуля

package Ada.Text_IO.Modular_IO is

   Default_Width : Field := Num'Width;  -- ширина числа в символах
   Default_Base  : Number_Base := 10;   -- устанавливает основание системы счисления
                                        --    принимаемое по умолчанию для всех
                                        --    операций ввода/вывода
```

```
   procedure Get                  -- ввод числа из указанного файла
     (File  : in File_Type;
      Item  : out Num;
      Width : in Field := 0);

   procedure Get                  -- ввод числа из файла стандартного ввода
     (Item  : out Num;
      Width : in Field := 0);

   procedure Put                  -- вывод числа в указанный файл
     (File  : in File_Type;
      Item  : in Num;
      Width : in Field := Default_Width;
      Base  : in Number_Base := Default_Base);

   procedure Put                  -- вывод числа в файл стандартного вывода
     (Item  : in Num;
      Width : in Field := Default_Width;
      Base  : in Number_Base := Default_Base);

   procedure Get                  -- ввод числа из строки
                                  --   (преобразование из строкового представления)
     (From : in String;
      Item : out Num;
      Last : out Positive);

   procedure Put                  -- вывод числа в строку
                                  --   (преобразование в строковое представление)
     (To   : out String;
      Item : in Num;
      Base : in Number_Base := Default_Base);

end Ada.Text_IO.Modular_IO;
```

E.1.4 Пакет *Ada.Text_IO.Float_IO*

```
generic
   type Num is digits <>;  -- параметр настройки модуля

package Ada.Text_IO.Float_IO is

   Default_Fore : Field := 2;              -- количество символов слева
                                          --   от десятичной точки
   Default_Aft  : Field := Num'Digits - 1; -- количество символов справа
                                          --   от десятичной точки
   Default_Exp  : Field := 3;             -- для нотации используемой
                                          --   в научных рассчетах

   procedure Get                  -- ввод числа из указанного файла
     (File  : in File_Type;
      Item  : out Num;
      Width : in Field := 0);

   procedure Get                  -- ввод числа из файла стандартного ввода
     (Item  : out Num;
      Width : in Field := 0);

   procedure Put                  -- вывод числа в указанный файл
     (File : in File_Type;
      Item : in Num;
      Fore : in Field := Default_Fore;
      Aft  : in Field := Default_Aft;
      Exp  : in Field := Default_Exp);
```

```
   procedure Put                    -- вывод числа в файл стандартного вывода
     (Item : in Num;
      Fore : in Field := Default_Fore;
      Aft  : in Field := Default_Aft;
      Exp  : in Field := Default_Exp);

   procedure Get                    -- ввод числа из строки
                                    --   (преобразование из строкового представления)
     (From : in String;
      Item : out Num;
      Last : out Positive);

   procedure Put                    -- вывод числа в строку
                                    --   (преобразование в строковое представление)
     (To   : out String;
      Item : in Num;
      Aft  : in Field := Default_Aft;
      Exp  : in Field := Default_Exp);

end Ada.Text_IO.Float_IO;
```

E.1.5 Пакет *Ada.Text_IO.Fixed_IO*

```
generic
   type Num is delta <>;  -- параметр настройки модуля

package Ada.Text_IO.Fixed_IO is

   Default_Fore : Field := Num'Fore; -- количество символов слева
                                    --   от десятичной точки
   Default_Aft  : Field := Num'Aft;  -- количество символов справа
                                    --   от десятичной точки
   Default_Exp  : Field := 0;       -- для нотации используемой
                                    --   в научных рассчетах

   procedure Get                    -- ввод числа из указанного файла
     (File  : in File_Type;
      Item  : out Num;
      Width : in Field := 0);

   procedure Get                    -- ввод числа из файла стандартного ввода
     (Item  : out Num;
      Width : in Field := 0);

   procedure Put                    -- вывод числа в указанный файл
     (File : in File_Type;
      Item : in Num;
      Fore : in Field := Default_Fore;
      Aft  : in Field := Default_Aft;
      Exp  : in Field := Default_Exp);

   procedure Put                    -- вывод числа в файл стандартного вывода
     (Item : in Num;
      Fore : in Field := Default_Fore;
      Aft  : in Field := Default_Aft;
      Exp  : in Field := Default_Exp);

   procedure Get                    -- ввод числа из строки
                                    --   (преобразование из строкового представления)
     (From : in String;
      Item : out Num;
      Last : out Positive);

   procedure Put                    -- вывод числа в строку
                                    --   (преобразование в строковое представление)
```

```
      (To   : out String;
       Item : in Num;
       Aft  : in Field := Default_Aft;
       Exp  : in Field := Default_Exp);

end Ada.Text_IO.Fixed_IO;
```

E.1.6 Пакет *Ada.Text_IO.Decimal_IO*

```
generic
   type Num is delta <> digits <>;  -- параметр настройки модуля

package Ada.Text_IO.Decimal_IO is   -- десятичные типы, как правило, используются
                                    --     в финансовых приложениях

   Default_Fore : Field := Num'Fore; -- количество символов слева
                                    --   от десятичной точки
   Default_Aft  : Field := Num'Aft;  -- количество символов справа
                                    --   от десятичной точки
   Default_Exp  : Field := 0;       -- для нотации используемой
                                    --   в научных рассчетах

   procedure Get                    -- ввод числа из указанного файла
     (File  : in File_Type;
      Item  : out Num;
      Width : in Field := 0);

   procedure Get                    -- ввод числа из файла стандартного ввода
     (Item  : out Num;
      Width : in Field := 0);

   procedure Put                    -- вывод числа в указанный файл
     (File : in File_Type;
      Item : in Num;
      Fore : in Field := Default_Fore;
      Aft  : in Field := Default_Aft;
      Exp  : in Field := Default_Exp);

   procedure Put                    -- вывод числа в файл стандартного вывода
     (Item : in Num;
      Fore : in Field := Default_Fore;
      Aft  : in Field := Default_Aft;
      Exp  : in Field := Default_Exp);

   procedure Get                    -- ввод числа из строки
                                    --   (преобразование из строкового представления)
     (From : in String;
      Item : out Num;
      Last : out Positive);

   procedure Put                    -- вывод числа в строку
                                    --   (преобразование в строковое представление)
     (To   : out String;
      Item : in Num;
      Aft  : in Field := Default_Aft;
      Exp  : in Field := Default_Exp);

end Ada.Text_IO.Decimal_IO;
```

E.1.7 Пакет *Ada.Text_IO.Enumeration_IO*

```
generic
   type Enum is (<>);  -- параметр настройки модуля

package Ada.Text_IO.Enumeration_IO is
```

```
   Default_Width : Field := 0;
   Default_Setting : Type_Set := Upper_Case;

   procedure Get      -- ввод значения перечислимого типа из указанного файла
     (File  : in  File_Type;
      Item  : out Enum);

   procedure Get      -- ввод значения перечислимого типа из файла стандартного ввода
     (Item  : out Enum);

   procedure Put      -- вывод значения перечислимого типа в указанный файл
     (File  : in File_Type;
      Item  : in Enum;
      Width : in Field := Default_Width;
      Set   : in Type_Set := Default_Setting);

   procedure Put      -- вывод значения перечислимого типа в файл стандартного вывода
     (Item  : in Enum;
      Width : in Field := Default_Width;
      Set   : in Type_Set := Default_Setting);

   procedure Get      -- ввод значения перечислимого типа из строки
                      --   (преобразование из строкового представления)
     (From : in String;
      Item : out Enum;
      Last : out Positive);

   procedure Put      -- вывод значения перечислимого типа в строку
                      --   (преобразование в строковое представление)
     (To   : out String;
      Item : in Enum;
      Set  : in Type_Set := Default_Setting);

end Ada.Text_IO.Enumeration_IO;
```

E.2 Пакет *Ada.Sequential_IO*

```
with Ada.IO_Exceptions;

generic
   type Element_Type (<>) is private;  -- параметр настройки модуля

package Ada.Sequential_IO is

   type File_Type is limited private;

   type File_Mode is (In_File, Out_File, Append_File);



   -- Подпрограммы управления файлами

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

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

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

   function Mode    (File : in File_Type) return File_Mode;
   function Name    (File : in File_Type) return String;
   function Form    (File : in File_Type) return String;

   function Is_Open (File : in File_Type) return Boolean;


   -- Подпрограммы ввода/вывода

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

   function End_Of_File (File : in File_Type) return Boolean;


   -- Исключения

   Status_Error : exception renames IO_Exceptions.Status_Error;
   Mode_Error   : exception renames IO_Exceptions.Mode_Error;
   Name_Error   : exception renames IO_Exceptions.Name_Error;
   Use_Error    : exception renames IO_Exceptions.Use_Error;
   Device_Error : exception renames IO_Exceptions.Device_Error;
   End_Error    : exception renames IO_Exceptions.End_Error;
   Data_Error   : exception renames IO_Exceptions.Data_Error;

private

  . . .   -- стандартом языка не определено

end Ada.Sequential_IO;
```

E.3 Пакет *Ada.Direct_IO*

```
with Ada.IO_Exceptions;

generic
   type Element_Type is private;  -- параметр настройки модуля

package Ada.Direct_IO is

   type File_Type is limited private;

   type File_Mode is (In_File, Inout_File, Out_File);


   type Count is range 0 .. System.Direct_IO.Count'Last;

   subtype Positive_Count is Count range 1 .. Count'Last;


   -- Подпрограммы управления файлами

   procedure Create
     (File : in out File_Type;
      Mode : in File_Mode := Inout_File;
```

```
      Name : in String := "";
      Form : in String := "");

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

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

   function Mode (File : in File_Type) return File_Mode;
   function Name (File : in File_Type) return String;
   function Form (File : in File_Type) return String;

   function Is_Open (File : in File_Type) return Boolean;



   -- Подпрограммы ввода/вывода

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

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

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

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

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

   function Index (File : in File_Type) return Positive_Count;
   function Size  (File : in File_Type) return Count;

   function End_Of_File (File : in File_Type) return Boolean;



   -- Исключения

   Status_Error : exception renames IO_Exceptions.Status_Error;
   Mode_Error   : exception renames IO_Exceptions.Mode_Error;
   Name_Error   : exception renames IO_Exceptions.Name_Error;
   Use_Error    : exception renames IO_Exceptions.Use_Error;
   Device_Error : exception renames IO_Exceptions.Device_Error;
   End_Error    : exception renames IO_Exceptions.End_Error;
   Data_Error   : exception renames IO_Exceptions.Data_Error;


private
```

```
   . . .   -- стандартом языка не определено

end Ada.Direct_IO;
```

E.4 Пакет *Ada.Streams.Stream_IO*

```
with Ada.IO_Exceptions;

package Ada.Streams.Stream_IO is

   type Stream_Access is access all Root_Stream_Type'Class;
   type File_Type is limited private;
   type File_Mode is (In_File, Out_File, Append_File);


   type Count is range 0 .. Определяется_Реализацией;

   subtype Positive_Count is Count range 1 .. Count'Last;  -- индекс в файле,
                                                           --   в потоковых
                                                           --   элементах

   -- Подпрограммы управления файлами

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

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

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

   function Mode (File : in File_Type) return File_Mode;
   function Name (File : in File_Type) return String;
   function Form (File : in File_Type) return String;

   function Is_Open     (File : in File_Type) return Boolean;
   function End_Of_File (File : in File_Type) return Boolean;

   function Stream (File : in File_Type) return Stream_Access;



   -- Подпрограммы ввода/вывода

   procedure Read
     (File : in  File_Type;
      Item : out Stream_Element_Array;
      Last : out Stream_Element_Offset;
      From : in  Positive_Count);

   procedure Read
     (File : in  File_Type;
      Item : out Stream_Element_Array;
      Last : out Stream_Element_Offset);
```

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

   procedure Write
     (File : in File_Type;
      Item : in Stream_Element_Array);



   -- Подпрограммы управления позиционированием внутри файла

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

   function Index (File : in File_Type) return Positive_Count;
   function Size  (File : in File_Type) return Count;

   procedure Set_Mode (File : in out File_Type; Mode : in File_Mode);

   procedure Flush (File : in out File_Type);



   -- Исключения

   Status_Error : exception renames IO_Exceptions.Status_Error;
   Mode_Error   : exception renames IO_Exceptions.Mode_Error;
   Name_Error   : exception renames IO_Exceptions.Name_Error;
   Use_Error    : exception renames IO_Exceptions.Use_Error;
   Device_Error : exception renames IO_Exceptions.Device_Error;
   End_Error    : exception renames IO_Exceptions.End_Error;
   Data_Error   : exception renames IO_Exceptions.Data_Error;

private

   . . .    -- стандартом языка не определено

end Ada.Streams.Stream_IO;
```

Приложение F

Глоссарий

Это дополнение содержит пояснение терминов и понятий используемых в стандарте Ada-95, и соответствует информации представленной в дополнении N (*Annex N*) стандарта Ada-95.

N п/п	Английский термин	Русский эквивалент	Описание
1	***Access type***	***Ссылочный тип***	Ссылочный тип определяет значения которые указывают на объекты с косвенным доступом. Ссылочные типы соответствуют "указателям"или "ссылкам"в некоторых других языках программирования.
2	***Aliased***	***Объект с косвенным доступом***	Косвенный доступ к представлению объекта может быть получен с помощью значения ссылочного типа. Динамические объекты, которые размещаются с помощью аллокаторов, являются объектами с косвенным доступом. Статический объект может быть также явно описан как объект с косвенным доступом путем использования зарезервированного слова "**aliased**" в описании объекта. В этом случае, для получения значения ссылочного типа, указывающего на объект с косвенным доступом, может быть использован атрибут "'Access".
3	***Array type***	***Тип массива***	Составной тип, который состоит из компонентов одного и того же типа. Выбор индивидуальных компонентов осуществляется путем индексирования.
4	***Character type***	***Символьный тип***	Символьный тип - это перечислимый тип значениями которого являются символы.
5	***Class***	***Класс***	Класс - это набор типов которые объединены одной иерархией наследования. Это подразумевает, что если указанный тип принадлежит классу, то все типы, производные от этого типа, также принадлежат этому классу. Множество типов одного класса обладает одинаковыми свойствами, такими как одинаковые примитивные операции.
6	***Compilation unit***	***Компилируемый модуль***	Текст программы может быть представлен компилятору для одной или более компиляций. Каждая компиляция - это последовательность компилируемых модулей. Компилируемый модуль содержит или описания, или тело, или переименование программного модуля.
7	***Composite type***	***Составной тип***	Составной тип имеет компоненты.

8	***Construct***	***Конструкция***	Конструкция - это фрагмент текста (явный или неявный), являющийся экземпляром синтаксической категории, которая определена в разделе синтаксиса.
9	***Controlled type***	***Контролируемый тип***	Контролируемый тип поддерживает определяемые пользователем подпрограммы присваивания (инициализации) и очистки. Такие объекты всегда перед разрушением выполняют очистку.
10	***Declaration***	***Описание***	Описание - это языковая конструкция, которая ассоциирует имя с сущностью (с представлением сущности). Описание в тексте программы может присутствовать явно (явное описание), или может подразумеваться как присутствующее в данном месте текста программы как семантический результат других конструкций (неявное описание).
11	***Definition***	***Определение***	Все определения содержат определения для представления какой-либо сущности. Представление состоит из какой-либо идентификации сущности (сущность представления), плюс специфических для представления характеристик которые влияют на использование сущности посредством такого представления (например, режим доступа к объекту, имена формальных параметров и их значения по умолчанию для подпрограммы, или видимость/доступность компонентов типа). В большинстве случаев, определение также содержит определение для самой сущности (определение_переименования является примером определения, которое не определяет новую сущность, но определяет представление существующей сущности, см. RM-95 8.5).
12	***Derived type***	***Производный тип***	Производный тип - это тип объявленный в терминах другого типа, который является типом-предком для производного типа. Каждый класс содержащий тип-предок также содержит производный тип. Производный тип наследует свойства, такие как компоненты и примиривные операции типа-предка. Любой тип вместе с производными от него типами (прямо или косвенно) формируют образование класса.
13	***Discrete type***	***Дискретный тип***	Любой дискретный тип это или любой целочисленный тип, или любой перечислимый тип. Например, дискретные типы могут быть использованы в инструкциях выбора ("**case**") или как индексы массивов.
14	***Discriminant***	***Дискриминант***	Дискриминант - это параметр составного типа. Например, он может управлять границами компонента типа если такой тип - это тип массива. Дискриминант задачного типа может быть использован для передачи данных во время создания фактического объекта задачного типа (иначе, задаче).
15	***Elementary type***	***Элементарный тип***	Любой элементарный тип не содержит компоненты.

16	***Enumeration type***	***Перечислимый тип***	Любой перечислимый тип - это тип который определен перечислением его значений, которые могут быть именованы идентификаторами или символьными литералами.
17	***Exception***	***Исключение***	Любое исключение представляет какую-либо исключительную ситуацию, а возникновение такой ситуации (во время выполнения программы) называют возникновением исключения. Возбуждение исключения используется для прекращения нормальной последовательности исполнения программы для предупреждения о факте наличия соответствующей ситуации. Выполнение некоторых действий, в ответ на возникшее исключение, называют обработкой исключения.
18	***Execution***	***Исполнение***	Процесс при котором проявляется действие какой-либо конструкции языка, во время выполнения программы, называют исполнением. Исполнение описаний называют элаборацией. Исполнение выражений называют вычислением.
19	***Generic unit***	***Настраиваемый модуль***	Настраиваемый модуль - это шаблон для построения (ненастраиваемого) программного модуля. Такой шаблон может быть параметризован объектами, типами, подпрограммами и пакетами. Экземпляр настроенного модуля может быть получен путем конкретизации настраиваемого модуля. Правила языка, при компиляции настраиваемого модуля, требуют использования контрактной модели настройки. При конкретизации настраиваемого модуля выполняются дополнительные проверки соблюдения контрактной модели. Таким образом, описание настраиваемого модуля представляет контракт между телом настраиваемого модуля и экземпляром настроенного модуля. Настраиваемые модули могут быть использованы в качестве макросов, используемых другими языками программирования.
20	***Integer type***	***Целочисленный тип***	Целочисленные типы включают целочисленные типы со знаком и модульные типы. Целочисленный тип со знаком имеет базовый диапазон, который содержит положительные и отрицательные числа, и имеет операции, которые могут возбуждать исключения когда результат операции выходит за пределы базового диапазона значений. Модульные типы имеют базовый диапазон значений нижняя граница которого - нуль, и используют операции с "кольцевой"семантикой. Модульные типы можно отнести к целочисленным типам без знака, используемым другими языками программирования.
21	***Library unit***	***Библиотечный модуль***	Библиотечный модуль - это отдельно компилируемый программный модуль, которым может быть пакет, подпрограмма или настраиваемый модуль. Библиотечные модули могут иметь другие (логически находящиеся внутри) библиотечные модули как дочерние модули, а также могут иметь другие программные модули, которые размещаются внутри них физически. Любой корневой библиотечный модуль, вместе со своими дочерними модулями, формирует подсистему.

22	***Limited type***	***Лимитированный тип***	Лимитированный тип - это тип (представление типа) для которого не допускается использование операции присваивания. Нелимитированный тип - это тип (представление типа) для которого использование операции присваивания допускается.
23	***Object***	***Объект***	Объектом является константа или переменная. Любой объект содержит значение. Любой объект может быть создан при помощи описания_объекта или при помощи аллокатора. Любой формальный параметр - это объект (представление объекта). Компонент объекта - это также объект.
24	***Package***	***Пакет***	Пакеты - это программные модули, которые позволяют объединять группы логически связанных между собой сущностей. Обычно пакет содержит описание типа (часто приватного типа или приватного расширения типа) в совокупности с описаниями примитивных подпрограмм для этого типа, которые могут быть вызваны извне этого пакета. При этом внутренняя работа таких подпрограмм будет скрыта от внешних пользователей.
25	***Partition***	***Раздел***	Раздел - это часть программы. Каждый раздел состоит из набора библиотечных модулей. Каждый раздел может выполняться в собственном адресном пространстве, возможно на отдельном компьютере. Программа может содержать один раздел. Распределенная программа обычно содержит множество разделов, которые могут выполняться одновременно.
26	***Pragma***	***Директива компилятора***	Директивы компилятора предоставляют дополнительную информацию для оптимизации, управления листингом трансляции и т.д. Существуют директивы компилятора определенные стандартом языка. Конкретная реализация может поддерживать дополнительные директивы компилятора.
27	***Primitive operations***	***Примитивные операции***	Примитивные операции типа - это операции (такие как подпрограммы) описанные вместе с описанием типа. Они наследуются другими типами того же класса типов. Для тэговых типов, примитивные подпрограммы являются диспетчеризуемыми (перенаправляемыми) подпрограммами, предусматривающими полиморфизм во время выполнения. Диспетчеризуемая подпрограмма может быть вызвана со статическими тэговыми операндами, в таком случае тело вызываемой подпрограммы определяется во время компиляции. Дополнительно, диспетчеризуемая подпрограмма может быть вызвана используя способ диспетчеризованного (перенаправленного) вызова, в таком случае, тело вызываемой подпрограммы определяется во время выполнения.
28	***Private extension***	***Приватное расширение***	Приватное расширение подобно расширению записи. Отличительной особенностью является то, что компоненты такого расширения не видимы для клиентов.

29	***Private type***	***Приватный тип***	Приватный тип - это частичное представление типа, полное представление которого скрыто от клиентов.
30	***Program unit***	***Программный модуль***	Программный модуль это пакет, задачный модуль, защищенный модуль, защищенная точка входа, настраиваемый модуль или явно описанная подпрограмма отличная от перечислимого литерала. Определенные программные модули могут быть откомпилированы отдельно. Дополнительно, они могут быть физически размещены внутри других программных модулей.
31	***Program***	***Программа***	Программа - это множество разделов, каждый из которых может исполняться в отдельном адресном пространстве, возможно на разных компьютерах. Раздел состоит из набора библиотечных модулей.
32	***Protected type***	***Защищенный тип***	Защищенный тип - это составной тип, чьи компоненты защащены от одновременного доступа множества задач.
33	***Real type***	***Вещественный тип***	Вещественный тип имеет значения, которые являются приближенными значениями вещественных чисел. Типы с плавающей точкои и типы с фиксированной точкой являются вещественными типами.
34	***Record extension***	***Расширение записи***	Расширение записи - это тип, который расширяет другой тип путем добавления дополнительных компонентов.
35	***Record type***	***Тип записи***	Тип записи - это составной тип, состоящий из ни одного или более именованных компонентов, тип которых может быть разным.
36	***Scalar type***	***Скалярный тип***	Скалярный тип - это или дискретный тип, или вещественный тип.
37	***Subtype***	***Подтип***	Подтип - это тип с ограничением, которое ограничивает значения подтипа с целью удовлетворения некоторых условий. Значения подтипа являются подмножеством значений его типа.
38	***Tagged type***	***Тэговый тип***	Объекты тэгового типа характеризуются наличием тэга типа времени выполнения. Такой тэг однозначно определяет тип созданного объекта. Операнд надклассового тэгового типа может быть использован для формирования диспетчеризуемого (перенаправляемого) вызова. В таком случае, тэг индицирует тело подпрограммы которую необходимо вызвать. Допустимы также недиспетчеризуемые вызовы, в которых тело подпрограммы, которую необходимо вызвать, определяется во время компиляции. Тэговые типы могут быть расширены дополнительными компонентами.
39	***Task type***	***Тип задачи*** ***Задачный тип***	Тип задачи - это составной тип значениями которого являются задачи. Задачи являются активными сущностями, которые могут выполняться одновременно с другими задачами. Задачу раздела программы верхнего уровня называют задачей окружения.

40	***Type***	***Тип***	Каждый объект имеет тип. Тип обладает ассоциированным с ним множеством значений и множеством примитивных операций, которые реализуют фундаментальный аспект его семантики. Типы группируются в классы. Типы определенного класса используют общее множество примитивных операций. Классы объединяются в иерархию наследования. Таким образом, если тип принадлежит определенному классу, то все производные от него типы будут принадлежать этому же классу.
41	***View***	***Представление***	см. ***Definition***

Литература

[1] Ada 95 Language Reference Manual ANSI/ISO/IEC 8652:1995
http://www.adapower.com/rm95/index.html

[2] Ada 95 Rationale, The Language and Standard Libraries
http://www.adapower.com/rationale/index.html

[3] Ada 95 Quality and Style: Guidelines for Professional Programmers
http://www.informatik.uni-stuttgart.de/ifi/ps/ada-doc/style_guide/cover.html

[4] GNAT Reference Manual

[5] GNAT User Guide

[6] Smith, Michael A., Object-Oriented Software in Ada 95
http://burks.bton.ac.uk/burks/language/ada/ada95.pdf

[7] English, John, Ada 95 The Craft of Object-Oriented Programming
http://www.it.bton.ac.uk/staff/je/adacraft/

[8] Feldman, M.B. and Koffman E.B., Ada 95 Problem Solving and Program Design

[9] Richard Riehle, Ada Distilled
http://www.adapower.com/learn/adadistilled.pdf

[10] Ken O. Burtch, The Big Online Book of Linux Ada programming
http://www.vaxxine.com/pegasoft/homes/book.html

[11] Dale Stanbrough, Quick Ada
http://goanna.cs.rmit.edu.au/ dale/ada/aln.html

Оглавление

Часть 3. Средства разработки

Часть 4. Приложения