Введение в язык Ада

Introduction to Ada

Raphaël Amiard
Gustavo A. Hoffmann

LEARN.
ADACORE.COM

LEARN.
ADACORE.COM

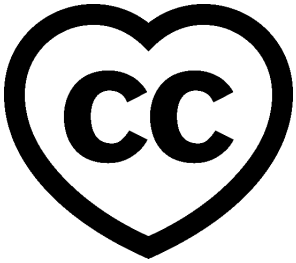
Введение в язык программирования Ada

Release 2022-08

Raphaël Amiard
and Gustavo A. Hoffmann

Aug 22, 2022

CONTENTS:



Этот курс научит вас основам языка программирования Ada и предназначен для тех, кто уже имеет базовое представление о методах программирования. Вы узнаете, как применить эти методы к программированию в Ada.

Этот документ был написан Рафаэлем Амьяром и Густаво А. Хоффманом с рецензией Ричарда Кеннера.

[1] http://creativecommons.org/licenses/by-sa/4.0

CHAPTER

ONE

ВВЕДЕНИЕ

1.1 История

В 1970-х годах Министерство обороны Соединенных Штатов (МО) столкнулось с серьезной проблемой резкого увеличения числа языков программирования, заметив, что различные проекты использовали разные и нестандартные диалекты, языковые подмножества и расширения языков. Что бы решить эту проблему, Министерство обороны запустило конкурс на разработку нового современного языка программирования общего назначения. Победителем вышло предложение, представленное Жаном Ичбией из CII Honeywell-Bull.

Первый стандарт языка Ада был выпущен в 1983 году; впоследствии он был пересмотрен и усовершенствован в 1995, 2005 и 2012 годах, причем каждый пересмотр приносил новые полезные функции.

Этот учебник посвящен Ада 2012 в целом и не освещает различия прошлых версий языка.

1.2 Ада сегодня

Сегодня Ада широко используется во встраиваемых системах реального времени, во многих из которых критически важна надежность. Хотя Ада может использоваться в качестве языка общего назначения, он особенно подходит для низкоуровневых приложений:

- Встроенные системы с ограниченным объёмом памяти (сборщик мусора не допускается).
- Прямое взаимодействие с оборудованием.
- Мягкие или жесткие системы реального времени.
- Низкоуровневое системное программирование.

Конкретные области, в которых используется Ада, включают аэрокосмическую и оборонную промышленность, гражданскую авиацию, железную дорогу и многие другие. Эти приложения требуют высокой степени безопасности: дефект программного обеспечения не просто раздражает, но может иметь серьезные последствия. Язык Ада обладает свойствами благодаря которые возможно обнаружить дефекты на ранней стадии разработки — обычно во время компиляции или с помощью инструментов статического анализа. Язык Ада также можно использовать для создания приложений в различных других областях, таких как:

- Программирование видеоигр[2]
- Аудио в реальном времени[3]
- Модули ядра[4]

[2] https://github.com/АдаDoom3/АдаDoom3
[3] http://www.electronicdesign.com/embedded-revolution/assessing-ada-language-audio-applications
[4] http://www.nihamkin.com/tag/kernel.html

Это неполный список, который, надеюсь, прольет свет на то, в каких сферах программирования хорош этот язык.

С точки зрения современных языков, наиболее близкими с точки зрения целей и уровня абстракции, вероятно, являются C++[5] и Rust[6].

1.3 Философия

Философия языка Ада отличается от большинства других языков. В основе дизайна Ада лежат принципы, среди которых можно выделить следующее:

- Удобочитаемость важнее краткости. Синтаксически это проявляется в том, что ключевые слова предпочтительнее символов, что ни одно ключевое слово не является аббревиатурой и т.д.
- Очень строгая типизация. В Аде очень легко вводить новые типы, что позволяет предотвратить ошибки использования данных.
  - В этом отношении он похож на многие функциональные языки, за исключением того, что программист должен гораздо более четко описывать набор типов в Аде, потому что здесь почти не применяется вывод типов.
- Явное действие лучше, чем неявное. Хотя это заповедь языка Python[7], Ада идет дальше, чем любой известный нам язык:
  - В большинстве случаев структурная типизация отсутствует, и программист должен явно именовать большинство типов.
  - Как уже говорилось ранее, в основном нет вывода типов.
  - Семантика очень чётко определена, а неопределенное поведение сведено к абсолютному минимуму.
  - Обычно программист может предоставить компилятору (и другим программистам) *много* информации о свойствах его программы. Это позволяет компилятору быть чрезвычайно полезным (читай: строгим) программисту.

В ходе этого курса мы объясним отдельные языковые особенности, которые являются строительными блоками этой философии.

1.4 SPARK

Хотя этот курс посвящен исключительно языку Ада, стоит упомянуть, что существует еще один язык, чрезвычайно близкий к Аде и совместимый с ней: язык SPARK.

SPARK — это подмножество языка Ада, разработанное таким образом, что код, написанный на SPARK, поддается автоматической проверке. Это обеспечивает гораздо более высокий уровень уверенности в правильности вашего кода, чем при использовании обычного языка программирования.

Существует специальный курс для языка SPARK[8]. Но следует иметь ввиду, когда мы говорим о мощи спецификаций языка Ада в этом курсе, можно усилить возможности этих спецификаций используя SPARK и доказать различные свойства программы, начиная с отсутствия ошибок во время выполнения до соответствия формально определенным функциональным требованиям.

[5] https://en.wikipedia.org/wiki/C%2B%2B
[6] https://www.rust-lang.org/en-US/
[7] https://www.python.org
[8] https://learn.adacore.com/courses/intro-to-spark/index.html

CHAPTER

TWO

ИМПЕРАТИВЫ ЯЗЫКА

Язык Ада поддерживает множество парадигм программирования, включая объектно-ориентированное программирование и некоторые элементы функционального программирования, но в его основе лежит простой сбалансированный процедурный/императивный язык, аналогичный C или Pascal.

На других языках

Одно важное различие между Адой и таким языком, как C, заключается в том, что операторы и выражения очень четко различаются. В Ада, если вы попытаетесь использовать выражение, там, где требуется оператор, ваша программа не будет скомпилированна. Это правило реализует полезный стилистический принцип: преднозначение выражений в вычислении значений, а не для в побочных эффектах. Оно также может предотвратить некоторые ошибки программирования, такие как ошибочное использование операции проверки на равенства `=` вместо операции присваивания `:=` в операторе присваивания.

2.1 Hello world

Вот очень простая императивная программа Ада:

Listing 1: greet.adb

```
with Ada.Text_IO;

procedure Greet is
begin
   --  Print "Hello, World!" to the screen
   Ada.Text_IO.Put_Line ("Hello, World!");
end Greet;
```

Runtime output

```
Hello, World!
```

Текс программы, как мы предполагаем, находится в исходном файле `greet.adb`.

В вышеупомянутой программе есть несколько примечательных вещей:

- Подпрограмма в Аде может быть либо процедурой, либо функцией. Процедура, как показано выше, не возвращает значение при вызове.
- **`with`** используется для ссылки на внешние модули, которые необходимы в процедуре. Это похоже на `import` в разных языках или примерно похоже на *`#include`* в C и C++. Позже мы разберемся в деталях их работы. Здесь мы запрашиваем стандартный библиотечный модуль, пакет `Ada.Text_IO`, который содержит процедуру печати текста на экране: `Put_Line`.

- `Greet` - это процедура и основная точка входа в нашу первую программу. В отличие от C или C++, его можно назвать как угодно по вашему усмотрению. Построитель должен знать точку входа. В нашем простом примере **`gprbuild`**, построитель GNAT, будет использовать файл, который вы передали в качестве параметра.
- `Put_Line` - это такая же процедура, как и `Greet`, за исключением того, что она объявлена в модуле `Ada.Text_IO`. Это Ада-эквивалент `printf` в Си.
- Комментарии начинаются с `--` и продолжаются до конца строки. Многострочные комментарии отсутствует, то есть невозможно начать комментарий в одной строке и продолжить его в следующей строке. Единственный способ создать несколько строк комментариев в Аде - это использовать "`--`" в каждой строке. Например:

```
--  We start a comment in this line...
--  and we continue on the second line...
```

На других языках

Процедуры аналогичны функциям в C или C++, которые возвращают значение **`void`**. Позже мы увидим, как объявлять функции в Аде.

Вот вариация примера «Hello, World»:

Listing 2: greet.adb

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

procedure Greet is
begin
   --  Print "Hello, World!" to the screen
   Put_Line ("Hello, World!");
end Greet;
```

Runtime output

```
Hello, World!
```

В этой версии используется конструкция Аді, известное как спецификатор использования (**`use`** clause), которая имеет форму **`use`** *имя-пакета*. Как видно на вызове `Put_Line`, эффект заключается в том, что на объекты из указанного пакета можно ссылаться напрямую – нет необходимости использовать *имя-пакета.* как префикс.

2.2 Условный оператор

В этом разделе описывается условный оператор **`if`** в Аде, и вводятся некоторые другие основные возможности языка, такие как целочисленный ввод-вывод, объявления данных и виды параметров подпрограммы.

Условный оператор **`if`** в Аде довольно неудивителен по форме и функции:

Listing 3: check_positive.adb

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

procedure Check_Positive is
   N : Integer;
begin
```

(continues on next page)

(continued from previous page)

```
   --  Put a String
   Put ("Enter an integer value: ");

   --  Read in an integer value
   Get (N);

   if N > 0 then
      --  Put an Integer
      Put (N);
      Put_Line (" is a positive number");
   end if;
end Check_Positive;
```

Оператор **if** в простейшей форме состоит из зарезервированного слова **if**, условия (которое должно быть логическим значением), зарезервированного слова **then** и непустой последовательности операторов (часть **then**), которая выполняется, если условие вычисляется как True, и оканчивается **end if**.

Этот примере объявляет целочисленную переменную N, запрашивает у пользователя целое число, проверяет, является ли значение положительным, и, если да, отображается значение целого числа, за которым следует строка "является положительным числом" (is a positive number). Если значение не является положительным, то процедура не выводит ничего.

Тип Integer является предопределенным типом со знаком, и его диапазон зависит от архитектуры компьютера. На типичных современных процессорах целое число имеет 32-разрядный знак.

Пример иллюстрирует некоторые основные функциональные возможности для целочисленного ввода-вывода. Соответствующие подпрограммы находятся в предопределенном пакете `Ada.Integer_Text_IO` и включают процедуру `Get` (которая считывает число с клавиатуры) и процедуру `Put` (которая отображает целое значение).

Вот небольшое изменение в примере, которое иллюстрирует оператор **if** с частью **else**:

Listing 4: check_positive.adb

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

procedure Check_Positive is
   N : Integer;
begin
   --  Put a String
   Put ("Enter an integer value: ");

   --  Reads in an integer value
   Get (N);

   --  Put an Integer
   Put (N);

   if N > 0 then
      Put_Line (" is a positive number");
   else
      Put_Line (" is not a positive number");
   end if;
end Check_Positive;
```

В этом примере, если входное значение не является положительным, то программа отображает значение, за которым следует строка "не является положительным числом" (is not a positive number).

Наш последний вариант иллюстрирует оператор **if** с **elsif** частями:

Listing 5: check_direction.adb

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

procedure Check_Direction is
   N : Integer;
begin
   Put ("Enter an integer value: ");
   Get (N);
   Put (N);

   if N = 0 or N = 360 then
      Put_Line (" is due north");
   elsif N in 1 .. 89 then
      Put_Line (" is in the northeast quadrant");
   elsif N = 90 then
      Put_Line (" is due east");
   elsif N in 91 .. 179 then
      Put_Line (" is in the southeast quadrant");
   elsif N = 180 then
      Put_Line (" is due south");
   elsif N in 181 .. 269 then
      Put_Line (" is in the southwest quadrant");
   elsif N = 270 then
      Put_Line (" is due west");
   elsif N in 271 .. 359 then
      Put_Line (" is in the northwest quadrant");
   else
      Put_Line (" is not in the range 0..360");
   end if;
end Check_Direction;
```

В этом примере ожидается, что пользователь введет целое число от 0 до 360 включительно, и отобразит, какому квадранту или оси соответствует значение. Операция **in** в Аде проверяет, находится ли скалярное значение в указанном диапазоне, и возвращает логический результат. Эффект программы должен быть очевиден; позже мы увидим альтернативный и более эффективный стиль для достижения того же эффекта используя оператор выбора **case**.

Ключевое слово **elsif** в Аде отличается от C или C ++, где вместо него будут использоваться блоки **else .. if**. И еще одно отличие - это наличие конца **end if** в Аде, что позволяет избежать проблемы, известной как «висящий else» (dangling else).

2.3 Циклы

У Ада есть три способа записи циклов. Каждый из них отличается от циклов for в C / Java / Javascript тем, что обладает более простыми синтаксисом и семантикой в соответствии с философией Ада.

2.3.1 Циклы for

Первый форма цикла - это цикл **for**, который позволяет выполнять итерацию по дискретному диапазону.

Listing 6: greet_5a.adb

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

procedure Greet_5a is
begin
   for I in 1 .. 5 loop
      --  Put_Line is a procedure call
      Put_Line ("Hello, World!" & Integer'Image (I));
      --        ^ Procedure parameter
   end loop;
end Greet_5a;
```

Runtime output

```
Hello, World! 1
Hello, World! 2
Hello, World! 3
Hello, World! 4
Hello, World! 5
```

Несколько моментов, которые следует отметить:

- `1 .. 5` - это дискретный диапазон, от `1` до `5` включительно.
- Параметр цикла `I` (имя произвольное) в теле цикла имеет значение в этом диапазоне.
- `I` является локальным для цикла, поэтому вы не можете ссылаться на `I` вне цикла.
- Хотя значение `I` увеличивается на каждой итерации, с точки зрения программы оно является константой. Попытка изменить его значение является незаконной; компилятор отклонит такую программу.
- **`Integer`**`'Image` - это функция, которая принимает целое число и преобразует его в строку. Это пример языковой конструкции, известной как *атрибут*, обозначенной синтаксисом `'`, который будет рассмотрен более подробно позже.
- Символ `&` является оператором конкатенации для строковых значений.
- **`end loop`** обозначает конец цикла

"Шаг" цикла ограничен 1 (направление вперед) и -1 (назад). Чтобы выполнить итерацию в обратном направлении по диапазону, используйте ключевое слово **`reverse`**:

Listing 7: greet_5a_reverse.adb

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

procedure Greet_5a_Reverse is
begin
   for I in reverse 1 .. 5 loop
      Put_Line ("Hello, World!"
                & Integer'Image (I));
   end loop;
end Greet_5a_Reverse;
```

Runtime output

```
Hello, World! 5
Hello, World! 4
Hello, World! 3
Hello, World! 2
Hello, World! 1
```

Границы цикла **for** могут быть вычислены во время выполнения; они вычисляются один раз, перед выполнением тела цикла. Если значение верхней границы меньше значения нижней границы, то цикл вообще не выполняется. Это относится и к **reverse** циклам. Таким образом, в следующем примере вывод не производится:

Listing 8: greet_no_op.adb

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

procedure Greet_No_Op is
begin
   for I in reverse 5 .. 1 loop
      Put_Line ("Hello, World!"
                & Integer'Image (I));
   end loop;
end Greet_No_Op;
```

Build output

```
greet_no_op.adb:5:23: warning: loop range is null, loop will not execute [enabled␣
↪by default]
```

Цикл **for** имеет более общую форму, чем та, которую мы проиллюстрировали здесь; подробнее об этом позже.

2.3.2 Простой цикл

Самая простая форма цикла в Аде - это «голый» цикл, который образует основу других циклов Ада.

Listing 9: greet_5b.adb

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

procedure Greet_5b is
   --  Variable declaration:
   I : Integer := 1;
   --  ^ Type
   --              ^ Initial value
begin
   loop
      Put_Line ("Hello, World!"
                & Integer'Image (I));

      --  Exit statement:
      exit when I = 5;
      --        ^ Boolean condition

      --  Assignment:
      I := I + 1;
      --  There is no I++ short form to
      --  increment a variable
   end loop;
end Greet_5b;
```

Runtime output

```
Hello, World! 1
Hello, World! 2
Hello, World! 3
Hello, World! 4
Hello, World! 5
```

Этот пример имеет тот же эффект, что и `Greet_5a`, показанный ранее.

Он иллюстрирует несколько концепций:

- Мы объявили переменную с именем `I` между **`is`** и **`begin`**. Этот участок кода представляет собой *зону описания*. Ада чётко отделяет зону описания от секции операторов подпрограммы. Объявление может находиться в зоне описания, но не допускается в качестве оператора.
- Оператор простого цикла начинается с ключевого слова **`loop`** и, как и любой другой тип оператора цикла, завершается комбинацией ключевых слов **`end loop`**. Сам по себе это бесконечный цикл. Вы можете выйти из этого цикла с помощью оператора выхода **`exit`**.
- Синтаксис для присваивания `:=`, а синтаксис для равенства `=`. Их невозможно перепутать, потому что, как отмечалось ранее, в Аде операторы и выражения различны, а выражения не являются допустимыми операторами.

2.3.3 Циклы while

Последняя форма цикла в Аде - это цикл **`while`**.

Listing 10: greet_5c.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Greet_5c is
4     I : Integer := 1;
5  begin
6     --  Condition must be a Boolean value
7     --  (no Integers).
8     --  Operator "<=" returns a Boolean
9     while I <= 5 loop
10       Put_Line ("Hello, World!"
11                 & Integer'Image (I));
12
13       I := I + 1;
14    end loop;
15 end Greet_5c;
```

Runtime output

```
Hello, World! 1
Hello, World! 2
Hello, World! 3
Hello, World! 4
Hello, World! 5
```

Условие вычисляется перед каждой итерацией. Если результат равен «ложь», то цикл завершается.

Эта программа имеет тот же эффект, что и предыдущие примеры.

На других языках

Обратите внимание, что Ада имеет иную семантику, чем языки на основе C, связанную с условием цикла while. В Ада условие должно быть логическим значением, иначе компилятор отклонит программу; условие не может быть целым числом, которое расценивается как истинное или ложное в зависимости от того, является ли оно ненулевым или нулевым.

2.4 Оператор выбора

Оператор выбора **case** в Аде аналогичен оператору **switch** из C/C++, но с некоторыми важными отличиями.

Вот пример, вариация программы, которая была показана ранее с инструкцией **if**:

Listing 11: check_direction.adb

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

procedure Check_Direction is
   N : Integer;
begin
   loop
      Put ("Enter an integer value: ");
      Get (N);
      Put (N);

      case N is
         when 0 | 360 =>
            Put_Line (" is due north");
         when 1 .. 89 =>
            Put_Line (" is in the northeast quadrant");
         when 90 =>
            Put_Line (" is due east");
         when 91 .. 179 =>
            Put_Line (" is in the southeast quadrant");
         when 180 =>
            Put_Line (" is due south");
         when 181 .. 269 =>
            Put_Line (" is in the southwest quadrant");
         when 270 =>
            Put_Line (" is due west");
         when 271 .. 359 =>
            Put_Line (" is in the northwest quadrant");
         when others =>
            Put_Line (" Au revoir");
            exit;
      end case;
   end loop;
end Check_Direction;
```

Эта программа неоднократно запрашивает целочисленное значение, а затем, если значение находится в диапазоне `0 .. 360`, отображает соответствующий квадрант или ось. Если значение является целым числом за пределами этого диапазона, цикл (и программа) завершаются после вывода прощального сообщения.

Эффект оператора выбора аналогичен условному оператору в предыдущем примере, но оператор выбора может быть более эффективным, нет нужды выполнять несколько тестов диапазона.

Примечательные моменты в операторе выбора в Аде:

- Выражение выбора (здесь переменная N) должно быть дискретного типа, то есть либо целочисленного типа, либо типа перечисления. *Дискретные типы* (page 39) будут рассмотрены более подробно позже.
- Каждое возможное значение для выражения выбора должно быть охвачено уникальной ветвью оператора **case**. Это будет проверено во время компиляции.
- Ветвь может указывать одно значение, например 0; диапазон значений, например 1 .. 89; или любая их комбинация (с разделителем |).
- Отдельно может быть задана конечная ветвь с ключевым словом **others**, которая охватывает все значения, не включенные в предыдущие ветки.
- Выполнение состоит из вычисления выражения выбора, а затем передачи управления последовательности инструкций в уникальной ветви, которая охватывает это значение.
- Когда выполнение операторов в выбранной ветви завершено, управление возобновляется после последней ветви (после **end case**). В отличие от C, выполнение не попадает в следующую ветвь. Таким образом, в Аде не нужен (и не существует) оператор **break**.

2.5 Зоны описания

Как упоминалось ранее, Ада проводит четкое синтаксическое разделение между объявлениями, которые вводят имена для сущностей, которые будут использоваться в программе, и операторами, выполняющими обработку. Области в программе, в которых могут появляться объявления, называются зонами описания.

В любой подпрограмме участок кода между **is** и **begin** является зоной описания. Там могут быть переменные, константы, типы, внутренние подпрограммы и другие сущности.

Мы кратко упоминали объявления переменных в предыдущем подразделе. Давайте посмотрим на простой пример, где мы объявляем целочисленную переменную X в зоне описания и выполняем инициализацию и добавление к ней единицы:

Listing 12: main.adb

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

procedure Main is
   X : Integer;
begin
   X := 0;
   Put_Line ("The initial value of X is "
             & Integer'Image (X));

   Put_Line ("Performing operation on X...");
   X := X + 1;

   Put_Line ("The value of X now is "
             & Integer'Image (X));
end Main;
```

Runtime output

```
The initial value of X is  0
Performing operation on X...
The value of X now is  1
```

Давайте рассмотрим пример вложенной процедуры:

Listing 13: main.adb

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

procedure Main is
   procedure Nested is
   begin
      Put_Line ("Hello World");
   end Nested;
begin
   Nested;
   --  Call to Nested
end Main;
```

Runtime output

```
Hello World
```

Объявление не может использоваться как оператор. Если вам нужно объявить локальную переменную среди операторов, вы можете ввести новую зону описания с помощью блочного оператора:

Listing 14: greet.adb

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

procedure Greet is
begin
   loop
      Put_Line ("Please enter your name: ");

      declare
         Name : String := Get_Line;
         --                ^ Call to the
         --                  Get_Line function
      begin
         exit when Name = "";
         Put_Line ("Hi " & Name & "!");
      end;

      --  Name is undefined here
   end loop;

  Put_Line ("Bye!");
end Greet;
```

Build output

```
greet.adb:9:10: warning: "Name" is not modified, could be declared constant [-
↪gnatwk]
```

Attention: Функция `Get_Line` позволяет получать входные данные от пользователя и выдавать результат в виде строки. Это более или менее эквивалентно функции `scanf` C.

Она возвращает строку (типа **`String`**), которая, как мы увидим позже, является *неограниченным типом массива* (page 69). Пока мы просто отмечаем, что, если вы хотите объявить строковую переменную (с типом **`String`**) и заранее не знаете ее размер, вам необходимо инициализировать переменную во время ее объявления.

2.6 Условные выражения

В Ада 2012 были введены выражения-аналоги условных операторов (**if** и **case**).

2.6.1 Условное выражение

Вот альтернативная версия примера приведённого выше; операторы **if** заменены на **if** выражения:

Listing 15: check_positive.adb

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

procedure Check_Positive is
   N : Integer;
begin
   Put ("Enter an integer value: ");
   Get (N);
   Put (N);

   declare
      S : constant String :=
        (if N > 0 then " is a positive number"
         else " is not a positive number");
   begin
      Put_Line (S);
   end;
end Check_Positive;
```

Выражение **if** вычисляет одну из двух строк в зависимости от N и присваивает это значение локальной переменной S.

Выражения **if** в Аде аналогичны операторам **if**. Однако есть несколько отличий, следующих из того факта, что это выражение:

- Выражения всех ветвей должны быть одного типа
- Оно *должно* быть заключено в круглые скобки, если уже не охвачено ими
- Ветвь **else** обязательна, если только следующее за **then** выражение не имеет логического значения. В этом случае ветвь **else** является необязательной и, если она отсутствует, по умолчанию аналогична **else True**.

Вот еще один пример:

Listing 16: main.adb

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

procedure Main is
begin
   for I in 1 .. 10 loop
      Put_Line (if I mod 2 = 0 then "Even" else "Odd");
   end loop;
end Main;
```

Runtime output

```
Odd
Even
```

(continues on next page)

(continued from previous page)

```
Odd
Even
Odd
Even
Odd
Even
Odd
Even
```

Эта программа выдает 10 строк вывода, чередующихся слов "Нечетный" и "Четный" ("Odd" и "Even").

2.6.2 Выражение выбора

Аналогично выражениям **if**, в Аде также есть выражения **case**. Они работают именно так, как вы и ожидали.

Listing 17: main.adb

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

procedure Main is
begin
   for I in 1 .. 10 loop
      Put_Line (case I is
                when 1 | 3 | 5 | 7 | 9 => "Odd",
                when 2 | 4 | 6 | 8 | 10 => "Even");
   end loop;
end Main;
```

Runtime output

```
Odd
Even
Odd
Even
Odd
Even
Odd
Even
Odd
Even
```

Эта программа имеет тот же эффект, что и в предыдущем примере.

Синтаксис отличается от операторов **case** тем, что ветви разделены запятыми.

CHAPTER

THREE

ПОДПРОГРАММЫ

3.1 Подпрограммы

До сих пор мы использовали процедуры, в основном, чтобы расположить там кода для исполнения. Процедуры являются одним из видов *подпрограмм*.

В Аде есть два вида подпрограмм: *функции* и *процедуры*. Различие между ними заключается в том, что функция возвращает значение, а процедура - нет.

В этом примере показано объявление и определение функции:

Listing 1: increment.ads

```
function Increment (I : Integer) return Integer;
```

Listing 2: increment.adb

```
--  We declare (but don't define) a function with
--  one parameter, returning an integer value

function Increment (I : Integer) return Integer is
   --  We define the Increment function
begin
    return I + 1;
end Increment;
```

Подпрограммы в Аде, конечно, могут иметь параметры. Одно важное замечание касающееся синтаксиса заключается в том, что подпрограмма, у которой нет параметров, вообще не имеет раздела параметров, например:

```
procedure Proc;

function Func return Integer;
```

Вот еще один вариант предыдущего примера:

Listing 3: increment_by.ads

```
function Increment_By
  (I    : Integer := 0;
   Incr : Integer := 1) return Integer;
--                ^ Default value for parameters
```

В этом примере мы видим, что параметры могут иметь значения по умолчанию. При вызове подпрограммы вы можете опустить параметры, если они имеют значение по умолчанию. В отличие от C/C++, вызов подпрограммы без параметров не использует скобки.

Вот реализация функции, описанной выше:

Listing 4: increment_by.adb

```
function Increment_By
  (I    : Integer := 0;
   Incr : Integer := 1) return Integer is
begin
   return I + Incr;
end Increment_By;
```

В наборе инструментов GNAT

Стандарт языка Ада не регламентирует в каких файлах следует расположить спецификацию и тело подпрограммы. Другими словами, стандарт не навязывает какую-либо структуру организации файлов или расширения имен файлов. К примеру, мы могли бы сохранить и спецификацию и тело указаной выше функции `Increment` в файле с названием `increment.txt`. (Мы даже могли бы поместить весь исходный код системы в один файл.) С точки зрения стандарта это вполне допустимо.

С другой стороны, набор инструментов GNAT требует следующую схему наименования файлов:

- файлы с расширением *.ads* содержат спецификацию, тогда, как
- файлы с расширением *.adb* содержат реализацию.

Таким образом, для инструментария GNAT, спецификация функции `Increment` должна находиться в файле `increment.ads`, а ее реализация должна находиться в файле `increment.adb`. Это правило также применяется для пакетов, которые мы обсудим *позже* (page 27). (Отметим, однако, что это правило можно обойти.) Дополнительные детали смотрите в курсе Introduction to GNAT Toolchain[9] или в GPRbuild User's Guide[10].

3.1.1 Вызовы подпрограмм

Далее мы можем вызвать нашу подпрограмму следующим образом:

Listing 5: show_increment.adb

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

procedure Show_Increment is
   A, B, C : Integer;
begin
   C := Increment_By;
   --              ^ Parameterless call,
   --                value of I is 0
   --                and Incr is 1

   Put_Line ("Using defaults for Increment_By is "
             & Integer'Image (C));

   A := 10;
   B := 3;
   C := Increment_By (A, B);
   --                 ^ Regular parameter passing

```

(continues on next page)

[9] https://learn.adacore.com/courses/GNAT_Toolchain_Intro/index.html
[10] https://docs.adacore.com/gprbuild-docs/html/gprbuild_ug.html

(continued from previous page)

```
   Put_Line ("Increment of "
             & Integer'Image (A)
             & " with "
             & Integer'Image (B)
             & " is "
             & Integer'Image (C));

   A := 20;
   B := 5;
   C := Increment_By (I    => A,
                      Incr => B);
    --                ^ Named parameter passing

   Put_Line ("Increment of "
             & Integer'Image (A)
             & " with "
             & Integer'Image (B)
             & " is "
             & Integer'Image (C));
end Show_Increment;
```

Runtime output

```
Using defaults for Increment_By is  1
Increment of  10 with  3 is  13
Increment of  20 with  5 is  25
```

Ада позволяет вам выполнять указывать имена параметров при передачи их во время вызова, независимо от того, есть ли значения по умолчанию или нет. Но есть несколько правил:

- Позиционные параметры должны идти первыми.
- Позиционный параметр не может следовать за именованным параметром.

Как правило, пользователи используют именованные параметры во время вызова, если соответствующий параметр функции имеет значение по умолчанию. Однако также вполне приветствуется использовать вызов с именованием каждого параметра, если это делает код более понятным.

3.1.2 Вложенные подпрограммы

Как кратко упоминалось ранее, Ада позволяет вам объявлять одну подпрограмму внутри другой.

Это полезно по двум причинам:

- Это позволяет вам получить более понятную программу. Если вам нужна подпрограмма только как «помощник» для другой подпрограммы, то принцип локализации указывает, что подпрограмма-помощник должна быть объявлена вложенной.
- Это облегчает вам доступ к данным объемлющей подпрограммы и сохранить при этом контроль, потому что вложенные подпрограммы имеют доступ к параметрам, а также к любым локальным переменным, объявленным во внешней области.

Используя предыдущий пример, можно переместить часть кода (вызов `Put_Line`) в отдельную процедуру и избежать дублирования. Вот укороченная версия с вложенной процедурой `Display_Result`:

Listing 6: show_increment.adb

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

procedure Show_Increment is
   A, B, C : Integer;

   procedure Display_Result is
   begin
      Put_Line ("Increment of "
                & Integer'Image (A)
                & " with "
                & Integer'Image (B)
                & " is "
                & Integer'Image (C));
   end Display_Result;

begin
   A := 10;
   B := 3;
   C := Increment_By (A, B);
   Display_Result;
end Show_Increment;
```

Runtime output

```
Increment of  10 with  3 is  13
```

3.1.3 Вызов функций

Важной особенностью вызовов функций в Аде является то, что возвращаемое значение при вызове нельзя игнорировать; то есть вызов функции не может использоваться как оператор.

Если вы хотите вызвать функцию и вам не нужен ее результат, вам все равно нужно будет явно сохранить его в локальной переменной.

Listing 7: quadruple.adb

```
function Quadruple (I : Integer) return Integer is
    function Double (I : Integer) return Integer is
    begin
       return I * 2;
    end Double;

   Res : Integer := Double (Double (I));
   --               ^ Calling the Double
   --                 function
begin
   Double (I);
   --  ERROR: cannot use call to function
   --         "Double" as a statement

   return Res;
end Quadruple;
```

Build output

```
quadruple.adb:11:04: error: cannot use call to function "Double" as a statement
quadruple.adb:11:04: error: return value of a function call cannot be ignored
gprbuild: *** compilation phase failed
```

В наборе инструментов GNAT

В GNAT, когда все предупреждения активированы, становится еще сложнее игнорировать результат функции, потому что неиспользуемые переменные будут выявлены. Например, этот код будет недействительным:

```
function Read_Int
   (Stream :     Network_Stream;
    Result : out Integer) return Boolean;

procedure Main is
    Stream : Network_Stream := Get_Stream;
    My_Int : Integer;

    -- Warning: in the line below, B is
    --          never read.
    B : Boolean := Read_Int (Stream, My_Int);
begin
   null;
end Main;
```

Затем у вас есть два решения, чтобы отключить это предупреждение:

- Либо аннотировать переменную с помощью pragma Unreferenced, таким образом:

```
B : Boolean := Read_Int (Stream, My_Int);
pragma Unreferenced (B);
```

- Или дайть переменной имя, которое содержит любую из строк: `discard` `dummy` `ignore` `junk` `unused` (без учета регистра)

3.2 Виды параметров

До сих пор мы видели, что Ада - это язык, ориентированный на безопасность. Существует много механизмов реализации этого принципа, но два важных момента заключаются в следующем:

- Ада позволяет пользователю как можно более точно указать ожидаемое поведение программы, чтобы компилятор мог предупреждать или отклонять при обнаружении несоответствия.
- Ада предоставляет множество методов для достижения общности и гибкости указателей и динамического управления памятью, но без недостатков последнего (таких как утечка памяти и висячие ссылки).

Виды параметров - это возможность, которая помогает воплотить эти два момента на практике. Параметр подпрограммы может быть одного из следующих видов:

in	Параметр может быть только считан, но не записан
out	Параметр можно записать, а затем прочитать
in out	Параметр может быть, как считан, так и записан

По умолчанию вид параметра будет **in**; до сих пор большинство примеров использовали параметры вида **in**.

Исторически

Функции и процедуры изначально были более разными по философии. До Ада 2012 функции могли принимать только «входящие» (**in**) параметры.

3.3 Вызов процедуры

3.3.1 Параметры in

Первый вид параметра - это тот, который мы неявно использовали до сих пор. Параметры этого вида нельзя изменить, поэтому следующая программа вызовет ошибку:

Listing 8: swap.adb

```
procedure Swap (A, B : Integer) is
   Tmp : Integer;
begin
   Tmp := A;

   --  Error: assignment to "in" mode
   --         parameter not allowed
   A := B;

   --  Error: assignment to "in" mode
   --         parameter not allowed
   B := Tmp;
end Swap;
```

Build output

```
swap.adb:8:04: error: assignment to "in" mode parameter not allowed
swap.adb:12:04: error: assignment to "in" mode parameter not allowed
gprbuild: *** compilation phase failed
```

Тот факт, что это вид используется по умолчанию, сам по себе очень важен. Это означает, что параметр не будет изменен, если вы явно не укажете ему другой вид, для которого разрешено изменение.

3.3.2 Параметры in out

Для исправления кода, приведенного выше, можно использовать параметр **in out** .

Listing 9: in_out_params.adb

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

procedure In_Out_Params is
   procedure Swap (A, B : in out Integer) is
      Tmp : Integer;
   begin
      Tmp := A;
      A   := B;
```

(continues on next page)

(continued from previous page)

```
      B   := Tmp;
   end Swap;

   A : Integer := 12;
   B : Integer := 44;
begin
    Swap (A, B);

    --  Prints 44
    Put_Line (Integer'Image (A));
end In_Out_Params;
```

Runtime output

```
 44
```

Параметр **in out** обеспечивает доступ для чтения и записи к объекту, переданному в качестве параметра, поэтому в приведенном выше примере видно, что значение A изменяется после вызова функции Swap.

Attention: В то время как параметры **in out** немного похожи на ссылки в C++ или обычные параметры в Java, которые передаются по ссылке, стандарт языка Ада не требует передачи параметров **in out** "по ссылке", за исключением определенных категорий типов, как будет объяснено позже.

В общем, лучше думать о видах параметров как о более высоком уровне, чем о семантике «по значению» или «по ссылке». Для компилятора это означает, что массив, передаваемый в качестве параметра **in**, может передаваться по ссылке, поскольку это более эффективно (что ничего не меняет для пользователя, поскольку параметр не может быть назначен). Однако параметр дискретного типа всегда будет передаваться копией, независимо от его вида (ведь так более эффективно на большинстве архитектур).

3.3.3 Параметры out

Вид «**out**» применяется, когда подпрограмме необходимо выполнить запись в параметр, который может быть не инициализирован в момент вызова. Чтение значения выходного параметра разрешено, но оно должно выполняться только после того, как подпрограмма присвоила значение параметру. Параметры **out** немного похожи на возвращаемые значения функций. Когда подпрограмма возвращается, фактический параметр (переменная) будет иметь значение параметра в точке возврата.

На других языках

Ада не имеет конструкции кортежа и не позволяет возвращать несколько значений из подпрограммы (за исключением объявления полноценного типа записи). Следовательно, способ вернуть несколько значений из подпрограммы состоит в использовании параметров out.

Например, процедура считывания целых чисел из сети может иметь одну из следующих спецификаций:

```
procedure Read_Int
   (Stream  :     Network_Stream;
    Success : out Boolean;
```

(continues on next page)

(continued from previous page)

```
    Result  : out Integer);

function Read_Int
   (Stream :     Network_Stream;
    Result : out Integer) return Boolean;
```

При чтении переменной **out** до записью в нее в идеале должна возникать ошибка, но, если бы было введено такое правило, то это бы привело либо к неэффективным проверкам во время выполнения, либо к очень сложным правилам во время компиляции. Таким образом, с точки зрения пользователя параметр **out** действует как неинициализированная в момент вызова подпрограммы переменная.

В наборе инструментов GNAT

GNAT обнаружит простые случаи неправильного использования параметров **out**. Например, компилятор выдаст предупреждение для следующей программы:

Listing 10: outp.adb

```
procedure Outp is
   procedure Foo (A : out Integer) is
      B : Integer := A;
      --             ^ Warning on reference
      --               to uninitialized A
   begin
      A := B;
   end Foo;
begin
   null;
end Outp;
```

Build output

```
outp.adb:2:14: warning: procedure "Foo" is not referenced [-gnatwu]
outp.adb:3:07: warning: "B" is not modified, could be declared constant [-gnatwk]
outp.adb:3:22: warning: "A" may be referenced before it has a value [enabled by␣
↪default]
```

3.3.4 Предварительное объявление подпрограмм

Как мы видели ранее, подпрограмма может быть объявлена без полного определения. Это возможно в целом и может быть полезно, если вам нужно, чтобы подпрограммы были взаимно рекурсивными, как в примере ниже:

Listing 11: mutually_recursive_subprograms.adb

```
procedure Mutually_Recursive_Subprograms is
    procedure Compute_A (V : Natural);
    --  Forward declaration of Compute_A

    procedure Compute_B (V : Natural) is
    begin
       if V > 5 then
          Compute_A (V - 1);
          --  Call to Compute_A
       end if;
    end Compute_B;
```

(continues on next page)

(continued from previous page)

```
12
13      procedure Compute_A (V : Natural) is
14      begin
15         if V > 2 then
16            Compute_B (V - 1);
17            --  Call to Compute_B
18         end if;
19      end Compute_A;
20   begin
21      Compute_A (15);
22   end Mutually_Recursive_Subprograms;
```

3.4 Переименование

Подпрограммы можно переименовать, используя ключевое слово **renames** и объявив новое имя для подпрограммы:

```
procedure New_Proc renames Original_Proc;
```

Это может быть полезно, например, для улучшения читаемости вашей программы, когда вы используете код из внешних источников, который нельзя изменить в вашей системе. Давайте посмотрим на пример:

Listing 12: a_procedure_with_very_long_name_that_cannot_be_changed.ads

```
1  procedure A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed
2    (A_Message : String);
```

Listing 13: a_procedure_with_very_long_name_that_cannot_be_changed.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed
4    (A_Message : String) is
5  begin
6     Put_Line (A_Message);
7  end A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed;
```

Как следует из названия процедуры, мы не можем изменить его. Однако мы можем переименовать процедуру во что-то вроде Show в нашем тестовом приложении и использовать это более короткое имя. Обратите внимание, что мы также должны объявить все параметры исходной подпрограммы, но мы можем именовать их по другому при объявлении. Например:

Listing 14: show_renaming.adb

```
1  with A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed;
2
3  procedure Show_Renaming is
4
5     procedure Show (S : String) renames
6       A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed;
7
8  begin
9     Show ("Hello World!");
10 end Show_Renaming;
```

Runtime output

```
Hello World!
```

Обратите внимание, что исходное имя (`A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed`) по-прежнему остается доступно после объявления процедуры Show.

Можно также переименовать подпрограммы из стандартной библиотеки. Например, можно переименовать `Integer'Image` в `Img`:

Listing 15: show_image_renaming.adb

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

procedure Show_Image_Renaming is

   function Img (I : Integer) return String
     renames Integer'Image;

begin
   Put_Line (Img (2));
   Put_Line (Img (3));
end Show_Image_Renaming;
```

Runtime output

```
 2
 3
```

Переименование также позволяет вводить выражения по умолчанию, которые не были указаны в исходном объявлении. Например, можно задать `"Hello World!"` в качестве значения по умолчанию для параметра `String` процедуры Show:

```
with A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed;

procedure Show_Renaming_Defaults is

   procedure Show (S : String := "Hello World!")
     renames
       A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed;

begin
   Show;
end Show_Renaming_Defaults;
```

CHAPTER

FOUR

МОДУЛЬНОЕ ПРОГРАММИРОВАНИЕ

До сих пор наши примеры были простыми автономными подпрограммами. Возможность расположить произвольные объявления в зоне описания способствует этому. Благодоря этой возможности мы смогли объявить наши типы и переменные в телах основных процедур.

Однако легко увидеть, что такой подход не вписывается в масштаб реальных приложений. Нам нужен лучший способ структурировать наши программы в модульные и отдельные блоки.

Ада поощряет разделение программ на несколько пакетов и подпакетов, предоставляя программисту множество инструментов в поисках идеальной организации кода.

4.1 Пакеты

Вот пример объявления пакета в Аде:

Listing 1: week.ads

```
package Week is

   Mon : constant String := "Monday";
   Tue : constant String := "Tuesday";
   Wed : constant String := "Wednesday";
   Thu : constant String := "Thursday";
   Fri : constant String := "Friday";
   Sat : constant String := "Saturday";
   Sun : constant String := "Sunday";

end Week;
```

А вот как вы его используете:

Listing 2: main.adb

```
with Ada.Text_IO; use Ada.Text_IO;
with Week;
--  References the Week package, and
--  adds a dependency from Main to Week

procedure Main is
begin
   Put_Line ("First day of the week is "
             & Week.Mon);
end Main;
```

Runtime output

```
First day of the week is Monday
```

Пакеты позволяют вам сделать код модульным, разбивая программы на семантически значимые единицы. Кроме того, отделение спецификации пакета от его тела (которое мы увидим ниже) может сократить время компиляции.

Хотя спецификатор контекста **with** указывает зависимость, в приведенном выше примере видно, что нам всё еще нужно использовать префикс с менем пакета, чтобы сослаться на имя в этом пакете. (Если бы мы также указали спецификатор использования **use** Week, то такой префикс уже не был бы необходим.)

При доступе к объектам из пакета используется точечная нотация `A.B`, аналогичная той, что используется для доступа к полям записей.

Спецификатор контекста **with** может появляться *только* в начале модуля компиляции (т. е. перед зарезервированным словом, таким как **procedure**, которое отмечает начало модуля). В других местах он запрещен. Это правило необходимо только по методологическим соображениям: человек, читающий ваш код, должен сразу видеть, от каких модулей он зависит.

На других языках

Пакеты похожи на файлы заголовков в C / C ++, но семантически сильно отличаются от них.

- Первое и самое важное отличие состоит в том, что пакеты представляют собой механизм уровня языка. В то время, как заголовочный файл из *#include* обрабатывается препроцессором C.
- Непосредственным следствием этого является то, что конструкция **with** работает на семантическом уровне, а не с помощью подстановки текста. Следовательно, когда вы работаете с пакетом указывая **with**, вы говорите компилятору: «Я зависим от этой семантической единицы», а не «включите сюда эту кучу текста».
- Таким образом, действие пакета не зависит от того, откуда на него ссылается **with**. Сравните это с C/C++, где смысл включенного текста может меняться в зависимости от контекста, в котором появляется *#include*.

  Это позволяет повысить эффективность компиляции/перекомпиляции. Это также облегчает инструментам, таким как IDE, получать правильную информацию о семантике программы. Что, в свою очередь, позволяет иметь лучший инструментарий в целом и код, который легче поддается анализу даже людьми.

Важным преимуществом **with** в Аде по сравнению с *#include* является то, что он не имеет состояния. Порядок спецификаторов **with** и **use** не имеет значения и может быть изменен без побочных эффектов.

В наборе инструментов GNAT

Стандарт языка Ада не предусматривает каких-либо особых отношений между исходными файлами и пакетами; например, теоретически вы можете поместить весь свой код в один файл или использовать свои собственные соглашения об именовании файлов. На практике, однако, каждая реализация имеет свои правила. Для GNAT каждый модуль компиляции верхнего уровня должен быть помещен в отдельный файл. В приведенном выше примере пакет `Week` будет находиться в файле `.ads` (для Ада спецификации), а основная процедура `Main` - в файле `.adb` (для Ада тела).

4.2 Использование пакета

Как мы видели выше, спецификатор контекста **with** указывает на зависимость от другого пакета. Тем не менее, каждая ссылка на сущъность, находящуюся в пакете Week, должна иметь префикс в виде полного имени пакета. Можно сделать все сущности пакета непосредственно видимым в текущей области с помощью спецификатора использования **use**.

Фактически, мы использовали спецификатор **use** почти с самого начала этого руководства.

Listing 3: main.adb

```
with Ada.Text_IO; use Ada.Text_IO;
--                ^ Make every entity of the
--                  Ada.Text_IO package
--                  directly visible.
with Week;

procedure Main is
   use Week;
   --  Make every entity of the Week
   --  package directly visible.
begin
   Put_Line ("First day of the week is " & Mon);
end Main;
```

Runtime output

```
First day of the week is Monday
```

Как вы можете видеть в приведенном выше примере:

- Put_Line - это подпрограмма из пакета Ada.Text_IO. Мы можем ссылаться на нее непосредственно, потому что мы указали пакет в **use** в верхней части основного модуля Main.
- В отличие от спецификатора контекста **with**, спецификатор использования **use** может быть помещен либо в заголовок, либо в любую область описания. В последнем случае эффект **use** будет распространяться только на соответствующую область видимости.

4.3 Тело пакета

В приведенном выше простом примере пакет Week содержит только объявления и не содержит реализаций. Это не ошибка: в спецификации пакета, которая проиллюстрирована выше, нельзя объявлять реализации. Реализации должны находиться в теле пакета.

Listing 4: operations.ads

```
package Operations is

   --  Declaration
   function Increment_By
     (I    : Integer;
      Incr : Integer := 0) return Integer;

   function Get_Increment_Value return Integer;

end Operations;
```

Listing 5: operations.adb

```
package body Operations is

   Last_Increment : Integer := 1;

   function Increment_By
     (I    : Integer;
      Incr : Integer := 0) return Integer is
   begin
      if Incr /= 0 then
         Last_Increment := Incr;
      end if;

      return I + Last_Increment;
   end Increment_By;

   function Get_Increment_Value return Integer is
   begin
      return Last_Increment;
   end Get_Increment_Value;

end Operations;
```

Здесь мы видим, что тело функции `Increment_By` должно быть объявлено в теле. Пользуясь появлением тела можно поместить переменную `Last_Increment` в тело, и сделать ее недоступной для пользователя пакета `Operations`, обеспечив первую форму инкапсуляции.

Это работает, поскольку объекты, объявленные в теле, видимы *только* в теле.

В этом примере показано, как непосредственно использовать `Increment_By`:

Listing 6: main.adb

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

procedure Main is
   use Operations;

   I : Integer := 0;
   R : Integer;

   procedure Display_Update_Values is
      Incr : constant Integer := Get_Increment_Value;
   begin
      Put_Line (Integer'Image (I)
                & " incremented by "
                & Integer'Image (Incr)
                & " is "
                & Integer'Image (R));
      I := R;
   end Display_Update_Values;
begin
   R := Increment_By (I);
   Display_Update_Values;
   R := Increment_By (I);
   Display_Update_Values;

   R := Increment_By (I, 5);
   Display_Update_Values;
   R := Increment_By (I);
```

(continues on next page)

(continued from previous page)

```
   Display_Update_Values;

   R := Increment_By (I, 10);
   Display_Update_Values;
   R := Increment_By (I);
   Display_Update_Values;
end Main;
```

Runtime output

```
 0 incremented by  1 is  1
 1 incremented by  1 is  2
 2 incremented by  5 is  7
 7 incremented by  5 is  12
 12 incremented by  10 is  22
 22 incremented by  10 is  32
```

4.4 Дочерние пакеты

Пакеты можно использовать для создания иерархий. Мы достигаем этого с помощью дочерних пакетов, которые расширяют функциональность родительского пакета. Одним из примеров дочернего пакета, который мы использовали до сих пор, является пакет `Ada.Text_IO`. Здесь родительский пакет называется `Ada`, а дочерний пакет называется `Text_IO`. В предыдущих примерах мы использовали процедуру `Put_Line` из дочернего пакета `Text_IO`.

Важное замечание

Ада также поддерживает вложенные пакеты. Однако, поскольку их использование может быть более сложным, рекомендуется использовать дочерние пакеты. Вложенные пакеты будут рассмотрены в расширенном курсе.

Давайте начнем обсуждение дочерних пакетов с нашего предыдущего пакета `Week`:

Listing 7: week.ads

```
package Week is

   Mon : constant String := "Monday";
   Tue : constant String := "Tuesday";
   Wed : constant String := "Wednesday";
   Thu : constant String := "Thursday";
   Fri : constant String := "Friday";
   Sat : constant String := "Saturday";
   Sun : constant String := "Sunday";

end Week;
```

Если мы хотим создать дочерний пакет для `Week`, мы можем написать:

Listing 8: week-child.ads

```
package Week.Child is

   function Get_First_Of_Week return String;

end Week.Child;
```

Здесь `Week` - это родительский пакет, а `Child` - дочерний. Это соответствующее тело пакета `Week.Child`:

Listing 9: week-child.adb

```
package body Week.Child is

   function Get_First_Of_Week return String is
   begin
      return Mon;
   end Get_First_Of_Week;

end Week.Child;
```

В реализации функции `Get_First_Of_Week` мы можем использовать строку `Mon` непосредственно, хотя она объявлена в родительском пакете `Week`. Мы не пишем здесь **`with`** `Week`, потому что все элементы из спецификации пакета `Week`, такие как `Mon`, `Tue` и т. д., видны в дочернем пакете `Week.Child`.

Теперь, когда мы завершили реализацию пакета `Week.Child`, мы можем использовать элементы из этого дочернего пакета в подпрограмме, просто написав **`with`** `Week.Child`. Точно так же, если мы хотим использовать эти элементы непосредственно, мы дополнительно напишем **`use`** `Week.Child`. Например:

Listing 10: main.adb

```
with Ada.Text_IO; use Ada.Text_IO;
with Week.Child;  use Week.Child;

procedure Main is
begin
   Put_Line ("First day of the week is "
             & Get_First_Of_Week);
end Main;
```

Runtime output

```
First day of the week is Monday
```

4.4.1 Дочерний пакет от дочернего пакета

До сих пор мы видели двухуровневую иерархию пакетов. Но иерархия, которую мы потенциально можем создать, этим не ограничивается. Например, мы могли бы расширить иерархию предыдущего примера исходного кода, объявив пакет `Week.Child.Grandchild`. В этом случае `Week.Child` будет родительским для пакета `Grandchild`. Рассмотрим эту реализацию:

Listing 11: week-child-grandchild.ads

```
package Week.Child.Grandchild is

   function Get_Second_Of_Week return String;

end Week.Child.Grandchild;
```

Listing 12: week-child-grandchild.adb

```
package body Week.Child.Grandchild is

```

(continues on next page)

(continued from previous page)

```
   function Get_Second_Of_Week return String is
   begin
      return Tue;
   end Get_Second_Of_Week;

end Week.Child.Grandchild;
```

Мы можем использовать этот новый пакет `Grandchild` в нашем тестовом приложении так же, как и раньше: мы можем повторно использовать предыдущее тестовое приложение адаптировав **`with`**, **`use`** и вызов функции. Вот обновленный код:

Listing 13: main.adb

```
with Ada.Text_IO;           use Ada.Text_IO;
with Week.Child.Grandchild; use Week.Child.Grandchild;

procedure Main is
begin
   Put_Line ("Second day of the week is "
             & Get_Second_Of_Week);
end Main;
```

Runtime output

```
Second day of the week is Tuesday
```

Опять же, это не предел иерархии пакетов. Мы могли бы продолжить расширение иерархии предыдущего примера, реализовав пакет `Week.Child.Grandchild.Grand_grandchild`.

4.4.2 Множественные потомки

До сих пор мы видели лишь один дочерний пакет родительского пакета. Однако родительский пакет также может иметь несколько дочерних. Мы могли бы расширить приведенный выше пример и создать пакет `Week.Child_2`. Например:

Listing 14: week-child_2.ads

```
package Week.Child_2 is

   function Get_Last_Of_Week return String;

end Week.Child_2;
```

Здесь `Week` по-прежнему является родительским пакетом для пакета `Child`, но также родительским пакетом и для пакета `Child_2`. Таким же образом, `Child_2`, очевидно, является одним из дочерних пакетов `Week`.

Это соответствующее тело пакета `Week.Child_2`:

Listing 15: week-child_2.adb

```
package body Week.Child_2 is

   function Get_Last_Of_Week return String is
   begin
      return Sun;
   end Get_Last_Of_Week;

end Week.Child_2;
```

Теперь мы можем сослаться на оба потомка в нашем тестовом приложении:

Listing 16: main.adb

```
with Ada.Text_IO;  use Ada.Text_IO;
with Week.Child;   use Week.Child;
with Week.Child_2; use Week.Child_2;

procedure Main is
begin
   Put_Line ("First day of the week is "
             & Get_First_Of_Week);
   Put_Line ("Last day of the week is "
             & Get_Last_Of_Week);
end Main;
```

Runtime output

```
First day of the week is Monday
Last day of the week is Sunday
```

4.4.3 Видимость

В предыдущем разделе мы видели, что элементы, объявленные в спецификации родительского пакета, видны в дочернем пакете. Однако это не относится к элементам, объявленным в теле родительского пакета.

Рассмотрим пакет `Book` и его дочерний элемент `Additional_Operations`:

Listing 17: book.ads

```
package Book is

   Title : constant String :=
     "Visible for my children";

   function Get_Title return String;

   function Get_Author return String;

end Book;
```

Listing 18: book-additional_operations.ads

```
package Book.Additional_Operations is

   function Get_Extended_Title return String;

   function Get_Extended_Author return String;

end Book.Additional_Operations;
```

Это тела обоих пакетов:

Listing 19: book.adb

```
package body Book is

   Author : constant String :=
     "Author not visible for my children";
```

(continues on next page)

(continued from previous page)

```
5
6     function Get_Title return String is
7     begin
8        return Title;
9     end Get_Title;
10
11    function Get_Author return String is
12    begin
13       return Author;
14    end Get_Author;
15
16 end Book;
```

Listing 20: book-additional_operations.adb

```
1  package body Book.Additional_Operations is
2
3     function Get_Extended_Title return String is
4     begin
5        return "Book Title: " & Title;
6     end Get_Extended_Title;
7
8     function Get_Extended_Author return String is
9     begin
10       --  "Author" string declared in the body
11       --  of the Book package is not visible
12       --  here. Therefore, we cannot write:
13       --
14       --  return "Book Author: " & Author;
15
16       return "Book Author: Unknown";
17    end Get_Extended_Author;
18
19 end Book.Additional_Operations;
```

В реализации `Get_Extended_Title` мы используем константу `Title` из родительского пакета `Book`. Однако, как указано в комментариях к функции `Get_Extended_Author`, строка `Author`, которую мы объявили в теле пакета `Book`, не отображается в пакете `Book.Additional_Operations`. Следовательно, мы не можем использовать его для реализации функции `Get_Extended_Author`.

Однако мы можем использовать функцию `Get_Author` из `Book` в реализации функции `Get_Extended_Author` для получения этой строки. Точно так же мы можем использовать эту стратегию для реализации функции `Get_Extended_Title`. Это адаптированный код:

Listing 21: book-additional_operations.adb

```
package body Book.Additional_Operations is

   function Get_Extended_Title return String is
   begin
      return "Book Title: " & Get_Title;
   end Get_Extended_Title;

   function Get_Extended_Author return String is
   begin
      return "Book Author: " & Get_Author;
   end Get_Extended_Author;

end Book.Additional_Operations;
```

Вот простое тестовое приложение для указанных выше пакетов:

Listing 22: main.adb

```
with Ada.Text_IO;                 use Ada.Text_IO;
with Book.Additional_Operations; use Book.Additional_Operations;

procedure Main is
begin
   Put_Line (Get_Extended_Title);
   Put_Line (Get_Extended_Author);
end Main;
```

Runtime output

```
Book Title: Visible for my children
Book Author: Author not visible for my children
```

Объявляя элементы в теле пакета, мы можем реализовать инкапсуляцию в языке Ада. Эти элементы будут видимы только в теле пакета, но нигде больше. Но это не единственный способ добиться инкапсуляции в Аде: мы обсудим другие подходы в главе *Изоляция* (page 105).

4.5 Переименование

Ранее мы упоминали, что *подпрограммы можно переименовывать* (page 25). Мы также можем переименовывать пакеты. Опять же, для этого мы используем ключевое слово **renames**. В следующем примере пакет `Ada.Text_IO` переименовывается как `TIO`:

Listing 23: main.adb

```
with Ada.Text_IO;

procedure Main is
   package TIO renames Ada.Text_IO;
begin
   TIO.Put_Line ("Hello");
end Main;
```

Runtime output

```
Hello
```

Мы можем использовать переименование, чтобы улучшить читаемость нашего кода, используя более короткие имена пакетов. В приведенном выше примере мы пишем `TIO.Put_Line` вместо более длинного имени (`Ada.Text_IO.Put_Line`). Этот подход особенно полезен, когда мы не используем спецификатор использования **`use`**, но хотим, чтобы код не становился слишком многословным.

Обратите внимание, что мы также можем переименовывать подпрограммы и объекты внутри пакетов. Например, мы могли бы просто переименовать процедуру `Put_Line` в приведенном выше примере исходного кода:

Listing 24: main.adb

```
with Ada.Text_IO;

procedure Main is
   procedure Say (Something : String)
     renames Ada.Text_IO.Put_Line;
begin
   Say ("Hello");
end Main;
```

Runtime output

```
Hello
```

CHAPTER

FIVE

СИЛЬНО ТИПИЗИРОВАННЫЙ ЯЗЫК

Ада - это строго типизированный язык. Удивительно, как она современна в этом: сильная статическая типизация становится все более популярной в дизайне языков программирования, если судить по таким факторам, как развитие функционального программирования со статической типизацией, прилагаемые усилия в области типизации со стороны исследовательского сообщества и появление множества практических языков с сильными системами типов.

5.1 Что такое тип?

В статически типизированных языках тип в основном (но не только) является конструкцией *времени компиляции*. Это конструкция, обеспечивающая инварианты поведения программы. Инварианты - это неизменяемые свойства, которые сохраняются для всех переменных данного типа. Их применение гарантирует, например, что значения переменных данного типа никогда не будут иметь недопустимых значений.

Тип используется для описания *объектов*, которыми управляет программа (объект это переменная или константа). Цель состоит в том, чтобы классифицировать объекты по тому, что можно с ними сделать (т.е. по разрешенным операциям), и, таким образом, судить о правильности значений объектов.

5.2 Целочисленные типы - Integers

Приятной возможностью языка Ада является то, что вы можете определить свои собственные целочисленные типы, основываясь на требованиях вашей программы (т.е. на диапазоне значений, который имеет смысл). Фактически механизм определения типов, который предоставляет Ада, лежит в основе предопределённых целочисленных типов. Таким образом, в языке нет «магических» встроенных типов, как в большинстве других языков, и это, пожалуй, очень элегантно.

Listing 1: integer_type_example.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Integer_Type_Example is
4     --  Declare a signed integer type,
5     --  and give the bounds
6     type My_Int is range -1 .. 20;
7     --                         ^ High bound
8     --                   ^ Low bound
9
10    --  Like variables, type declarations can
11    --  only appear in declarative regions.
```

(continues on next page)

(continued from previous page)

```
begin
   for I in My_Int loop
      Put_Line (My_Int'Image (I));
      --                ^ 'Image attribute
      --                  converts a value
      --                  to a String.
   end loop;
end Integer_Type_Example;
```

Runtime output

```
-1
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
 10
 11
 12
 13
 14
 15
 16
 17
 18
 19
 20
```

Этот пример иллюстрирует объявление целочисленного типа со знаком, и несколько моментов, связанных с его использованием.

Каждое объявление типа в Аде начинается с ключевого слова **type** (кроме *задачных типов* (page 150)). После ключевого слова мы можем видеть определение нижней и верхней границ типа в виде диапазона, который очень похож на диапазоны используемые в циклах **for**. Любое целое число этого диапазона является допустимым значением для данного типа.

Целочисленные типы Ада

В Аде целочисленный тип задается не в терминах его машинного представления, а скорее его диапазоном. Затем компилятор сам выберет наиболее подходящее представление.

Еще один момент, который следует отметить в приведенном выше примере, - это выражение `My_Int'Image (I)`. Обозначение вида `Name'Attribute (необязательные параметры)` используется для того, что в Аде называется атрибутом. Атрибут - это встроенная операция над типом, значением или какой-либо другой программной сущностью. Доступ к нему осуществляется с помощью символа ' (апостроф в ASCII).

Ада имеет несколько "встроенных" типов; **Integer** - один из них. Вот как тип целых чисел **Integer** может быть определен для типичного процессора:

```
type Integer is
  range -(2 ** 31) .. +(2 ** 31 - 1);
```

Знак ** обозначает возведение в степень, в итоге, первое допустимое значение для типа

Integer равно -2^{31}, а последнее допустимое значение равно 2^{31} - 1.

Ада не регламентирует диапазон встроенного типа **Integer**. Реализация для 16-битного целевого процессора, вероятно, выберет диапазон от - 2^{15} до 2^{15} - 1.

5.2.1 Семантика операций

В отличие от некоторых других языков, Ада требует, чтобы операции с целыми числами контролировали переполнение.

Listing 2: main.adb

```
procedure Main is
   A : Integer := Integer'Last;
   B : Integer;
begin
   B := A + 5;
   --  This operation will overflow, eg. it
   --  will raise an exception at run time.
end Main;
```

Build output

```
main.adb:2:04: warning: "A" is not modified, could be declared constant [-gnatwk]
main.adb:3:04: warning: variable "B" is assigned but never read [-gnatwm]
main.adb:5:04: warning: possibly useless assignment to "B", value might not be␣
↪referenced [-gnatwm]
main.adb:5:11: warning: value not in range of type "Standard.Integer" [enabled by␣
↪default]
main.adb:5:11: warning: "Constraint_Error" will be raised at run time [enabled by␣
↪default]
```

Runtime output

```
raised CONSTRAINT_ERROR : main.adb:5 overflow check failed
```

Существует два типа проверок переполнения:

- Переполнение на уровне процессора, когда результат операции превышает максимальное значение (или меньше минимального значения), которое может поместиться в машинном представлении объекта данного типа, и
- Переполнение на уровне типа, если результат операции выходит за пределы диапазона, определенного для типа.

В основном по соображениям эффективности, переполнение уровня типа будет проверяться лишь в определенные моменты, такие как присваивание значения, тогда как, переполнение низкого уровня всегда приводит к возбуждению исключения:

Listing 3: main.adb

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

procedure Main is
   type My_Int is range 1 .. 20;
   A : My_Int := 12;
   B : My_Int := 15;
   M : My_Int := (A + B) / 2;
   --  No overflow here, overflow checks
   --  are done at specific boundaries.
```

(continues on next page)

(continued from previous page)

```
begin
   for I in 1 .. M loop
      Put_Line ("Hello, World!");
   end loop;
   --  Loop body executed 13 times
end Main;
```

Build output

```
main.adb:5:04: warning: "A" is not modified, could be declared constant [-gnatwk]
main.adb:6:04: warning: "B" is not modified, could be declared constant [-gnatwk]
main.adb:7:04: warning: "M" is not modified, could be declared constant [-gnatwk]
```

Runtime output

```
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
```

Переполнение уровня типа будет проверяться только в определенных точках выполнения. Результат, как мы видим выше, состоит в том, что у вас может быть операция в промежуточном вычислении, которая переполняется, но никаких исключений не будет, пока конечный результат не вызовет переполняения.

5.3 Беззнаковые типы

Ада также поддерживает целочисленные типы без знака. На языке Ада они называются *модульными* типами. Причина такого обозначения связана с их поведением в случае переполнения: они просто "заварачиваются", как если бы была применена операция по модулю.

Для модульных типов машинного размера, например модуля 2^{32}, это имитирует наиболее распространенное поведение реализации беззнаковых типов. Однако преимущество Ада в том, что модуль может быть произвольным:

Listing 4: main.adb

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

procedure Main is
   type Mod_Int is mod 2 ** 5;
   --              ^ Range is 0 .. 31

   A : constant Mod_Int := 20;
   B : constant Mod_Int := 15;

   M : constant Mod_Int := A + B;
   --  No overflow here,
```

(continues on next page)

(continued from previous page)

```
   --  M = (20 + 15) mod 32 = 3
begin
   for I in 1 .. M loop
      Put_Line ("Hello, World!");
   end loop;
end Main;
```

Runtime output

```
Hello, World!
Hello, World!
Hello, World!
```

В отличие от C/C++, такое поведение гарантировано спецификацией языка Ада и на него можно положиться при создании переносимого кода. Кроме того, для реализации определенных алгоритмов и структур данных, таких как кольцевые буферы[11], очень удобно иметь возможность использовать эффект "заворачивания" на произвольных границах, ведь модуль не обязательно должен быть степенью 2.

5.4 Перечисления

Перечислимые типы - еще одна особенность системы типов в Аде. В отличие от перечислений C, они *не* являются целыми числами, и каждый новый перечислимый тип несовместим с другими перечислимыми типами. Перечислимые типы являются частью большего семейства дискретных типов, что делает их пригодными для использования в определенных ситуациях, которые мы опишем позже, но один контекст, с которым мы уже встречались, - это оператор case.

Listing 5: enumeration_example.adb

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

procedure Enumeration_Example is
   type Days is (Monday, Tuesday, Wednesday,
                 Thursday, Friday,
                 Saturday, Sunday);
   --  An enumeration type
begin
   for I in Days loop
      case I is
         when Saturday .. Sunday =>
            Put_Line ("Week end!");

         when Monday .. Friday =>
            Put_Line ("Hello on "
                      & Days'Image (I));
            --  'Image attribute, works on
            --  enums too
      end case;
   end loop;
end Enumeration_Example;
```

Runtime output

```
Hello on MONDAY
Hello on TUESDAY
```

(continues on next page)

[11] https://ru.wikipedia.org/wiki/Кольцевой_буфер

(continued from previous page)

```
Hello on WEDNESDAY
Hello on THURSDAY
Hello on FRIDAY
Week end!
Week end!
```

Типы перечисления достаточно мощные, поэтому, в отличие от большинства языков, они используются для определения стандартного логического типа:

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

Как упоминалось ранее, каждый "встроенный" тип в Аде определяется с помощью средств, обычно доступных пользователю.

5.5 Типы с плавающей запятой

5.5.1 Основные свойства

Как и большинство языков, Ада поддерживает типы с плавающей запятой. Наиболее часто используемый тип с плавающей запятой - **Float**:

Listing 6: floating_point_demo.adb

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

procedure Floating_Point_Demo is
   A : constant Float := 2.5;
begin
   Put_Line ("The value of A is "
             & Float'Image (A));
end Floating_Point_Demo;
```

Runtime output

```
The value of A is  2.50000E+00
```

Приложение отобразит `2.5` как значение A.

Язык Ада не регламентирует точность (количество десятичных цифр в мантиссе) для Float; на типичной 32-разрядной машине точность будет равна 6.

Доступны все общепринятые операции, которые можно было бы ожидать для типов с плавающей запятой, включая получение абсолютного значения и возведение в степень. Например:

Listing 7: floating_point_operations.adb

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

procedure Floating_Point_Operations is
   A : Float := 2.5;
begin
   A := abs (A - 4.5);
   Put_Line ("The value of A is "
             & Float'Image (A));

   A := A ** 2 + 1.0;
   Put_Line ("The value of A is "
```

(continues on next page)

(continued from previous page)

```
12             & Float'Image (A));
13 end Floating_Point_Operations;
```

Runtime output

```
The value of A is  2.00000E+00
The value of A is  5.00000E+00
```

Значение A равно `2.0` после первой операции и `5.0` после второй операции.

В дополнение к **Float**, реализация Ада может предлагать типы данных с более высокой точностью, такие как **Long_Float** и **Long_Long_Float**. Как и для Float, стандарт не указывает требуемую точность этих типов: он только гарантирует, что тип **Long_Float**, например, имеет точность не хуже **Float**. Чтобы гарантировать необходимую точность, можно определить свой пользовательский тип с плавающей запятой, как будет показано в следующем разделе.

5.5.2 Точность типов с плавающей запятой

Ада позволяет пользователю определить тип с плавающей запятой с заданной точностью, выраженной в десятичных знаках. Все операции этого типа будут иметь, по крайней мере, заданную точность. Синтаксис простого объявления типа с плавающей запятой:

```
type T is digits <number_of_decimal_digits>;
```

Компилятор выберет представление с плавающей запятой, поддерживающее требуемую точность. Например:

Listing 8: custom_floating_types.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Custom_Floating_Types is
4     type T3  is digits 3;
5     type T15 is digits 15;
6     type T18 is digits 18;
7  begin
8     Put_Line ("T3  requires "
9               & Integer'Image (T3'Size)
10              & " bits");
11    Put_Line ("T15 requires "
12              & Integer'Image (T15'Size)
13              & " bits");
14    Put_Line ("T18 requires "
15              & Integer'Image (T18'Size)
16              & " bits");
17 end Custom_Floating_Types;
```

Runtime output

```
T3  requires  32 bits
T15 requires  64 bits
T18 requires  128 bits
```

В этом примере атрибут «`'Size`» используется для получения количества бит, используемых для указанного типа данных. Как видно из этого примера, компилятор выделяет 32 бита для T3, 64 бита для T15 и 128 битов для T18. Сюда входят как мантисса, так и экспонента.

Количество цифр, указанное в типе данных, также используется в формате при отображении переменных с плавающей точкой. Например:

Listing 9: display_custom_floating_types.adb

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

procedure Display_Custom_Floating_Types is
   type T3  is digits 3;
   type T18 is digits 18;

   C1 : constant := 1.0e-4;

   A : constant T3  := 1.0 + C1;
   B : constant T18 := 1.0 + C1;
begin
   Put_Line ("The value of A is "
             & T3'Image (A));
   Put_Line ("The value of B is "
             & T18'Image (B));
end Display_Custom_Floating_Types;
```

Runtime output

```
The value of A is  1.00E+00
The value of B is  1.00010000000000000E+00
```

Как и ожидалось, приложение будет отображать переменные в соответствии с заданной точностью (1.00E + 00 и 1.00010000000000000E + 00).

5.5.3 Диапазон значений для типов с плавающей запятой

В дополнение к точности для типа с плавающей запятой можно также задать диапазон. Синтаксис аналогичен записи для целочисленных типов данных — с использованием ключевого слова **range**. В этом простом примере создается новый тип с плавающей запятой на основе типа **Float** с диапазоном от `-1.0` до `1.0`:

Listing 10: floating_point_range.adb

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

procedure Floating_Point_Range is
   type T_Norm  is new Float range -1.0 .. 1.0;
   A  : T_Norm;
begin
   A := 1.0;
   Put_Line ("The value of A is "
             & T_Norm'Image (A));
end Floating_Point_Range;
```

Runtime output

```
The value of A is  1.00000E+00
```

Приложение отвечает за обеспечение того, чтобы переменные этого типа находились в пределах этого диапазона; в противном случае возникает исключение. В следующем примере `Constraint_Error` исключения возникает при присваивании значения `2.0` переменной A:

Listing 11: floating_point_range_exception.adb

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

procedure Floating_Point_Range_Exception is
   type T_Norm  is new Float range -1.0 .. 1.0;
   A  : T_Norm;
begin
   A := 2.0;
   Put_Line ("The value of A is "
             & T_Norm'Image (A));
end Floating_Point_Range_Exception;
```

Build output

```
floating_point_range_exception.adb:7:09: warning: value not in range of type "T_
↪Norm" defined at line 4 [enabled by default]
floating_point_range_exception.adb:7:09: warning: "Constraint_Error" will be␣
↪raised at run time [enabled by default]
```

Runtime output

```
raised CONSTRAINT_ERROR : floating_point_range_exception.adb:7 range check failed
```

Диапазоны также могут быть заданы для пользовательских типов с плавающей запятой. Например:

Listing 12: custom_range_types.adb

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

procedure Custom_Range_Types is
   type T6_Inv_Trig  is
     digits 6 range -Pi / 2.0 .. Pi / 2.0;
begin
   null;
end Custom_Range_Types;
```

Build output

```
custom_range_types.adb:1:09: warning: no entities of "Ada.Text_IO" are referenced␣
↪[-gnatwu]
custom_range_types.adb:1:20: warning: use clause for package "Text_IO" has no␣
↪effect [-gnatwu]
custom_range_types.adb:5:09: warning: type "T6_Inv_Trig" is not referenced [-
↪gnatwu]
```

В этом примере мы определяем тип под названием `T6_Inv_Trig`, который имеет диапазон от −□/2 до □/2 с минимальной точностью 6 цифр. (`Pi` определяется в предопределенном пакете `Ada.Numerics`.)

5.6 Строгая типизация

Как отмечалось ранее, язык Ада строго типизирован. В результате разные типы одного семейства несовместимы друг с другом; значение одного типа не может быть присвоено переменной другого типа. Например:

Listing 13: illegal_example.adb

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

procedure Illegal_Example is
   --  Declare two different floating point types
   type Meters is new Float;
   type Miles is new Float;

   Dist_Imperial : Miles;

   --  Declare a constant
   Dist_Metric : constant Meters := 1000.0;
begin
   --  Not correct: types mismatch
   Dist_Imperial := Dist_Metric * 621.371e-6;
   Put_Line (Miles'Image (Dist_Imperial));
end Illegal_Example;
```

Build output

```
illegal_example.adb:14:33: error: expected type "Miles" defined at line 6
illegal_example.adb:14:33: error: found type "Meters" defined at line 5
gprbuild: *** compilation phase failed
```

Следствием этих правил является то, что в общем случае выражение «смешанного режима» типа `2 * 3.0` инициирует ошибку компиляции. В языке, таком как C или Python, такие выражения допустимы благодаря неявным преобразованиям типов. В Ада такие преобразования должны быть явными:

Listing 14: conv.adb

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Conv is
   type Meters is new Float;
   type Miles is new Float;
   Dist_Imperial : Miles;
   Dist_Metric : constant Meters := 1000.0;
begin
   Dist_Imperial := Miles (Dist_Metric) * 621.371e-6;
   --               ^ Type conversion,
   --                 from Meters to Miles
   --  Now the code is correct

   Put_Line (Miles'Image (Dist_Imperial));
end Conv;
```

Runtime output

```
 6.21371E-01
```

Конечно, мы, вероятно, не хотим писать код преобразования каждый раз, когда мы преобразуем метры в мили. Идиоматическим решением в этом случае считается введение функции преобразования вместе с типами.

Listing 15: conv.adb

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

procedure Conv is
   type Meters is new Float;
   type Miles is new Float;

   --  Function declaration, like procedure
   --  but returns a value.
   function To_Miles (M : Meters) return Miles is
   --                             ^ Return type
   begin
      return Miles (M) * 621.371e-6;
   end To_Miles;

   Dist_Imperial : Miles;
   Dist_Metric   : constant Meters := 1000.0;
begin
   Dist_Imperial := To_Miles (Dist_Metric);
   Put_Line (Miles'Image (Dist_Imperial));
end Conv;
```

Runtime output

```
 6.21371E-01
```

Если вы пишете код, где много вычислений, то необходимость явных преобразований может показаться обременительным. Однако такой подход дает определенные преимущества. Ведь вы можете полагаться на отсутствие неявных преобразований, которые, в свою очередь, могут привести к тяжело обнаружимым ошибкам.

На других языках

В C, например, правила для неявных преобразований не всегда могут быть полностью очевидными. Однако в Аде код всегда будет делать именно то, что, явно определено программистом. Например:

```
int a = 3, b = 2;
float f = a / b;
```

Этот код будет компилироваться нормально, но результатом `f` будет 1.0 вместо 1.5, потому что компилятор сгенерирует целочисленное деление (три, разделенное на два), что приведет к единице. Разработчик программного обеспечения должен знать о проблемах преобразования данных и использовать соответствующее приведение типов:

```
int a = 3, b = 2;
float f = (float)a / b;
```

В исправленном примере компилятор преобразует обе переменные в соответствующее представление с плавающей запятой перед выполнением деления. Что даст ожидаемый результат.

Этот пример очень прост, и опытные разработчики C, вероятно, заметят и исправят его, прежде чем это создаст большие проблемы. Однако в более сложных приложениях, где объявление типа не всегда видно - например, при ссылке на элементы структуры **`struct`**, - эта ситуация может не всегда быть очевидной и быстро привести к дефектам программного обеспечения, которые обнаружить может быть сложнее.

Компилятор Ада, напротив, всегда будет отклонять код, который смешивает переменные с плавающей запятой и целочисленные переменные без явного преобразования. Следующий

Ада код, основанный на ошибочном примере в C, не будет компилироваться:

Listing 16: main.adb

```
procedure Main is
   A : Integer := 3;
   B : Integer := 2;
   F : Float;
begin
   F := A / B;
end Main;
```

Build output

```
main.adb:6:04: warning: possibly useless assignment to "F", value might not be␣
 ↪referenced [-gnatwm]
main.adb:6:11: error: expected type "Standard.Float"
main.adb:6:11: error: found type "Standard.Integer"
gprbuild: *** compilation phase failed
```

Строка с ошибкой должна быть изменена на `F := Float (A) / Float (B);`.

- Вы можете использовать строгую типизацию Ада, чтобы обеспечить соблюдение инвариантов в вашем коде, как в приведенном выше примере: поскольку мили и метры - это два разных типа, вы не можете случайно использовать значения одного типа вместо другого.

5.7 Производные типы

В Ада можно создавать новые типы на основе существующих. Это очень полезно: вы получаете тип, который имеет те же свойства, что и некоторый существующий тип, но ведет себя как отдельный тип в соответствии с правилами сильной типизации.

Listing 17: main.adb

```
procedure Main is
   --  ID card number type,
   --  incompatible with Integer.
   type Social_Security_Number is new Integer
     range 0 .. 999_99_9999;
   --        ^ Since a SSN has 9 digits
   --          max., and cannot be
   --          negative, we enforce
   --          a validity constraint.

   SSN : Social_Security_Number :=
     555_55_5555;
   --    ^ You can put underscores as
   --      formatting in any number.

   I   : Integer;

   --  The value -1 below will cause a
   --  runtime error and a compile time
   --  warning with GNAT.
   Invalid : Social_Security_Number := -1;
begin
   --  Illegal, they have different types:
   I := SSN;
```

(continues on next page)

(continued from previous page)

```
25
26    --  Likewise illegal:
27    SSN := I;
28
29    --  OK with explicit conversion:
30    I := Integer (SSN);
31
32    --  Likewise OK:
33    SSN := Social_Security_Number (I);
34 end Main;
```

Build output

```
main.adb:21:40: warning: value not in range of type "Social_Security_Number"␣
 ↪defined at line 4 [enabled by default]
main.adb:21:40: warning: "Constraint_Error" will be raised at run time [enabled by␣
 ↪default]
main.adb:24:09: error: expected type "Standard.Integer"
main.adb:24:09: error: found type "Social_Security_Number" defined at line 4
main.adb:27:11: error: expected type "Social_Security_Number" defined at line 4
main.adb:27:11: error: found type "Standard.Integer"
main.adb:33:04: warning: possibly useless assignment to "SSN", value might not be␣
 ↪referenced [-gnatwm]
gprbuild: *** compilation phase failed
```

Тип `Social_Security`, как говорят, является *производным типом*; его *родительский тип* - Integer.

Как показано в этом примере, вы можете уточнить допустимый диапазон значений при определении производного скалярного типа (такого как целое число, число с плавающей запятой и перечисление).

Синтаксис перечислений использует синтаксис **range** <диапазон>:

Listing 18: greet.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Greet is
4     type Days is (Monday, Tuesday, Wednesday,
5                   Thursday, Friday,
6                   Saturday, Sunday);
7
8     type Weekend_Days is new
9       Days range Saturday .. Sunday;
10    --  New type, where only Saturday and Sunday
11    --  are valid literals.
12 begin
13    null;
14 end Greet;
```

Build output

```
greet.adb:1:09: warning: no entities of "Ada.Text_IO" are referenced [-gnatwu]
greet.adb:1:19: warning: use clause for package "Text_IO" has no effect [-gnatwu]
greet.adb:4:18: warning: literal "Monday" is not referenced [-gnatwu]
greet.adb:4:26: warning: literal "Tuesday" is not referenced [-gnatwu]
greet.adb:4:35: warning: literal "Wednesday" is not referenced [-gnatwu]
greet.adb:5:18: warning: literal "Thursday" is not referenced [-gnatwu]
greet.adb:5:28: warning: literal "Friday" is not referenced [-gnatwu]
greet.adb:8:09: warning: type "Weekend_Days" is not referenced [-gnatwu]
```

5.8 Подтипы

Вышеизложенное может привести нас к идее, что типы в Аде могут быть использованы для наложения ограничений на диапазон допустимый значений. Но иногда бывает нужно ограничить значения оставаясь в пределах одного типа. Здесь приходят на помощь подтипы. Подтип не вводит новый тип.

Listing 19: greet.adb

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

procedure Greet is
   type Days is (Monday, Tuesday, Wednesday,
                 Thursday, Friday,
                 Saturday, Sunday);

   --  Declaration of a subtype
   subtype Weekend_Days is
     Days range Saturday .. Sunday;
   --      ^ Constraint of the subtype

   M : Days := Sunday;

   S : Weekend_Days := M;
   --  No error here, Days and Weekend_Days
   --  are of the same type.
begin
   for I in Days loop
      case I is
         --  Just like a type, a subtype can
         --  be used as a range
         when Weekend_Days =>
            Put_Line ("Week end!");
         when others =>
            Put_Line ("Hello on "
                      & Days'Image (I));
      end case;
   end loop;
end Greet;
```

Build output

```
greet.adb:13:04: warning: "M" is not modified, could be declared constant [-gnatwk]
greet.adb:15:04: warning: variable "S" is not referenced [-gnatwu]
```

Runtime output

```
Hello on MONDAY
Hello on TUESDAY
Hello on WEDNESDAY
Hello on THURSDAY
Hello on FRIDAY
Week end!
Week end!
```

Несколько подтипов предопределены в стандартном пакете Ада и автоматически доступны вам:

```
subtype Natural  is Integer range 0 .. Integer'Last;
subtype Positive is Integer range 1 .. Integer'Last;
```

Хотя подтипы одного типа статически совместимы друг с другом, ограничения проверяются

во время выполнения: если вы нарушите ограничение подтипа, будет возбуждено исключение.

Listing 20: greet.adb

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

procedure Greet is
   type Days is (Monday, Tuesday, Wednesday,
                 Thursday, Friday,
                 Saturday, Sunday);

   subtype Weekend_Days is
     Days range Saturday .. Sunday;

   Day     : Days := Saturday;
   Weekend : Weekend_Days;
begin
   Weekend := Day;
   --         ^ Correct: Same type, subtype
   --           constraints are respected
   Weekend := Monday;
   --         ^ Wrong value for the subtype
   --           Compiles, but exception at runtime
end Greet;
```

Build output

```
greet.adb:1:09: warning: no entities of "Ada.Text_IO" are referenced [-gnatwu]
greet.adb:1:19: warning: use clause for package "Text_IO" has no effect [-gnatwu]
greet.adb:4:26: warning: literal "Tuesday" is not referenced [-gnatwu]
greet.adb:4:35: warning: literal "Wednesday" is not referenced [-gnatwu]
greet.adb:5:18: warning: literal "Thursday" is not referenced [-gnatwu]
greet.adb:5:28: warning: literal "Friday" is not referenced [-gnatwu]
greet.adb:11:04: warning: "Day" is not modified, could be declared constant [-
 ↪gnatwk]
greet.adb:12:04: warning: variable "Weekend" is assigned but never read [-gnatwm]
greet.adb:14:04: warning: useless assignment to "Weekend", value overwritten at␣
 ↪line 17 [-gnatwm]
greet.adb:17:04: warning: possibly useless assignment to "Weekend", value might␣
 ↪not be referenced [-gnatwm]
greet.adb:17:15: warning: value not in range of type "Weekend_Days" defined at␣
 ↪line 8 [enabled by default]
greet.adb:17:15: warning: "Constraint_Error" will be raised at run time [enabled␣
 ↪by default]
```

Runtime output

```
raised CONSTRAINT_ERROR : greet.adb:17 range check failed
```

5.8.1 Подтипы в качестве псевдонимов типов

Ранее мы видели, что мы можем создавать новые типы, объявляя `type Miles is new Float`. Но нам также может потребоваться переименовать тип, просто чтобы ввести альтернативное имя-псевдоним для существующего типа. Следует отметить, что *псевдонимы* типов иногда называются *синонимами* типов.

В Аде это делается с помощью подтипов без новых ограничений. Однако в этом случае мы не получаем всех преимуществ строгой типизации Ады. Перепишем пример, используя псевдонимы типов:

Listing 21: undetected_imperial_metric_error.adb

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

procedure Undetected_Imperial_Metric_Error is
   --  Declare two type aliases
   subtype Meters is Float;
   subtype Miles is Float;

   Dist_Imperial : Miles;

   --  Declare a constant
   Dist_Metric : constant Meters := 100.0;
begin
   --  No conversion to Miles type required:
   Dist_Imperial := (Dist_Metric * 1609.0) / 1000.0;

   --  Not correct, but undetected:
   Dist_Imperial := Dist_Metric;

   Put_Line (Miles'Image (Dist_Imperial));
end Undetected_Imperial_Metric_Error;
```

Build output

```
undetected_imperial_metric_error.adb:14:04: warning: useless assignment to "Dist_
 ↪Imperial", value overwritten at line 17 [-gnatwm]
```

Runtime output

```
 1.00000E+02
```

В приведенном выше примере тот факт, что и метры (`Meters`), и мили (`Miles`) являются подтипами `Float`, позволяет нам смешивать переменные обоих типов без преобразования типов. Это, однако, может привести к всевозможным ошибкам в программировании, которых мы стремимся избежать, как можно видеть в необнаруженной ошибке, выделенной в приведенном выше коде. В этом примере ошибка в присвоении значения в метрах переменной, предназначенной для хранения значений в милях, остается необнаруженной, поскольку и метры (`Meters`), и мили (`Miles`) являются подтипами `Float`. Поэтому, для случаев, подобных приведенному выше, рекомендуется использовать строгую типизацию, опеделив тип X производным от типа Y (`type X is new Y`).

Однако существует много ситуаций, где псевдонимы типов полезны. Например, в приложении, которое использует типы с плавающей запятой в нескольких контекстах, мы могли бы использовать псевдонимы типов, чтобы уточнить преднозначение или избежать длинных имен переменных. Например, вместо того, чтобы писать:

```
Paid_Amount, Due_Amount : Float;
```

Мы можем написать:

```
subtype Amount is Float;

Paid, Due : Amount;
```

На других языках

Например, в C для создания псевдонима типа можно использовать объявление **typedef**. Например:

```
typedef float meters;
```

Это соответствует определению подтипа без ограничений, которое мы видели выше. Другие языки программирования вводят эту концепцию аналогичными способами. Например:

- C++: `using meters = float;`
- Swift: `typealias Meters = Double`
- Kotlin: `typealias Meters = Double`
- Haskell: `type Meters = Float`

Однако следует отметить, что подтипы в Аде соответствуют псевдонимам типов, если и только если они не вводят новых ограничений. Таким образом, если добавить новое ограничение к описанию подтипа, у нас больше не будет псевдонима типа. Например, следующее объявление *не может* считаться синонимом типа **Float**:

```
subtype Meters is Float range 0.0 .. 1_000_000.0;
```

Рассмотрим другой пример:

```
subtype Degree_Celsius is Float;

subtype Liquid_Water_Temperature is
  Degree_Celsius range 0.0 .. 100.0;

subtype Running_Water_Temperature is
  Liquid_Water_Temperature;
```

В этом примере `Liquid_Water_Temperature` не является псевдонимом `Degree_Celsius`, поскольку добавляет новое ограничение, которое не было частью объявления `Degree_Celsius`. Однако здесь есть два псевдонима типа:

- `Degree_Celsius` является псевдонимом **Float**;
- `Running_Water_Temperature` является псевдонимом `Liquid_Water_Temperature`, даже если сам `Liquid_Water_Temperature` имеет ограниченный диапазон.

CHAPTER

SIX

ЗАПИСИ

Пока что все типы, с которыми мы столкнулись, имеют значения, которые нельзя разложить: каждый экземпляр представляет собой неделимый элемент данных. Теперь мы обратим наше внимание на наш первый составной тип: записи.

Записи позволяют составлять значения из значений других типов. Каждому из таких значений будет присвоено имя. Пара, состоящая из имени и значения определенного типа, называется полем или компонентой.

6.1 Объявление типа записи

Вот пример простого объявления записи:

```
type Date is record
   --  The following declarations are
   --  components of the record
   Day   : Integer range 1 .. 31;
   Month : Months;
   --  You can add custom constraints
   --  on fields
   Year  : Integer range 1 .. 3000;
end record;
```

Поля во многом похожи на объявления переменных, за исключением того, что они находятся внутри определения записи. Как и при объявлениях переменных, можно указать дополнительные ограничения при предоставлении подтипа поля.

```
type Date is record
   Day   : Integer range 1 .. 31;
   Month : Months := January;
   --  This component has a default value
   Year  : Integer range 1 .. 3000 := 2012;
   --                                 ^ Default value
end record;
```

Компоненты записи могут иметь значения по умолчанию. Когда объявлена переменная с типом записи, для поля с инициализацией получат заданные значения автоматически. Значение может быть задано любым выражением соответствующего типа и может вычислятся во время исполнения.

6.2 Агрегаты

```
Ada_Birthday    : Date := (10, December, 1815);
Leap_Day_2020   : Date := (Day   => 29,
                           Month => February,
                           Year  => 2020);
--                         ^ By name
```

Записи имеют удобную форму для записи значений, показанную выше. Эта нотация называется агрегированной, а сама конструкция - агрегатом. Ее можно использовать в различных контекстах, которые мы увидим на протяжении всего курса, и одним из применений является инициализация записей.

Агрегат - это список значений, разделенных запятыми и заключенных в круглые скобки. Он разрешен в любом контексте, где ожидается значение записи.

Значения для компонент можно указать позиционно, как в примере `Ada_Birthday`, или по имени, как в `Leap_Day_2020`. Разрешено сочетание позиционных и именованных значений, но вы не можете использовать позиционную форму после появления именованной.

6.3 Извлечение компонент

Для доступа к компонентам экземпляра записи используется операция, называемая извлечением компоненты. Она имеет форму точечной нотации. Например, если мы объявляем переменную `Some_Day` типа записи `Date`, упомянутого выше, мы можем получить доступ к компоненте `Year`, написав `Some_Day.Year`.

Рассмотрим пример:

Listing 1: record_selection.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Record_Selection is
4
5     type Months is
6       (January, February, March, April,
7        May, June, July, August, September,
8        October, November, December);
9
10    type Date is record
11       Day   : Integer range 1 .. 31;
12       Month : Months;
13       Year  : Integer range 1 .. 3000 := 2032;
14    end record;
15
16    procedure Display_Date (D : Date) is
17    begin
18       Put_Line ("Day:" & Integer'Image (D.Day)
19                 & ", Month: "
20                 & Months'Image (D.Month)
21                 & ", Year:"
22                 & Integer'Image (D.Year));
23    end Display_Date;
24
25    Some_Day : Date := (1, January, 2000);
26
27 begin
28    Display_Date (Some_Day);
```

(continues on next page)

(continued from previous page)

```
29
30    Put_Line ("Changing year...");
31    Some_Day.Year := 2001;
32
33    Display_Date (Some_Day);
34 end Record_Selection;
```

Runtime output

```
Day: 1, Month: JANUARY, Year: 2000
Changing year...
Day: 1, Month: JANUARY, Year: 2001
```

Как вы можете видеть в этом примере, мы можем использовать точечную нотацию в выражении `D.Year` или `Some_Day.Year`. как для доступа к информации компонеты, так и для ее изменения в операторе присваивания. Говоря конкретнее, когда мы используем `D.Year` в вызове `Put_Line`, мы читаем информацию, хранящуюся в этом компоненте. Когда мы пишем `Some_Day.Year := 2001`, то перезаписываем информацию, которая была ранее сохранена в компоненте `Year` в переменной `Some_Day`.

6.4 Переименование

В предыдущих главах мы обсуждали переименование *подпрограмм* (page 25) и *пакетов* (page 36). Мы также можем переименовывать компоненты записи. Вместо того, чтобы каждый раз писать извлечение компоненты с использованием точечной записи, мы можем объявить псевдоним, который позволит нам получить доступ к этой компоненте. Это полезно, например, для упрощения реализации подпрограммы.

Мы можем переименовать компоненты записи, используя ключевое слово **renames** в объявлении переменной. Например:

```
Some_Day : Date;
Y        : Integer renames Some_Day.Year;
```

Здесь `Y` является псевдонимом, так что каждый раз, когда мы используем `Y`, мы в действительности используем компоненту `Year` переменной `Some_Day`.

Давайте рассмотрим полный пример:

Listing 2: dates.ads

```
1  package Dates is
2
3     type Months is
4       (January, February, March, April,
5        May, June, July, August, September,
6        October, November, December);
7
8     type Date is record
9        Day   : Integer range 1 .. 31;
10       Month : Months;
11       Year  : Integer range 1 .. 3000 := 2032;
12    end record;
13
14    procedure Increase_Month (Some_Day : in out Date);
15
16    procedure Display_Month (Some_Day : Date);
```

(continues on next page)

(continued from previous page)

```

end Dates;
```

Listing 3: dates.adb

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

package body Dates is

   procedure Increase_Month (Some_Day : in out Date) is
      --  Renaming components from
      --  the Date record
      M : Months  renames Some_Day.Month;
      Y : Integer renames Some_Day.Year;

      --  Renaming function (for Months
      --  enumeration)
      function Next (M : Months) return Months
        renames Months'Succ;
   begin
      if M = December then
         M := January;
         Y := Y + 1;
      else
         M := Next (M);
      end if;
   end Increase_Month;

   procedure Display_Month (Some_Day : Date) is
      --  Renaming components from
      --  the Date record
      M : Months  renames Some_Day.Month;
      Y : Integer renames Some_Day.Year;
   begin
      Put_Line ("Month: "
                & Months'Image (M)
                & ", Year:"
                & Integer'Image (Y));
   end Display_Month;

end Dates;
```

Listing 4: main.adb

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

procedure Main is
   D : Date := (1, January, 2000);
begin
   Display_Month (D);

   Put_Line ("Increasing month...");
   Increase_Month (D);

   Display_Month (D);
end Main;
```

Runtime output

```
Month: JANUARY, Year: 2000
Increasing month...
Month: FEBRUARY, Year: 2000
```

Мы применяем переименование к двум компонентам записи `Date` в реализации процедуры `Increase_Month`. Затем вместо непосредственного использования `Some_Day.Month` и `Some_Day.Year` в последующих операциях мы просто используем переименованные версии `M` и `Y`.

Обратите внимание, что в приведенном выше примере мы также переименовали `Months'Succ` - функцию, которая дает нам следующий месяц - в `Next`.

CHAPTER

SEVEN

МАССИВЫ

Массивы (или индексируемые типы) предоставляют еще одно фундаментальное семейство составных типов в Аде.

7.1 Объявление типа массива

Массивы в Аде используются для определения непрерывных коллекций элементов, к которым можно обращаться по индексу.

Listing 1: greet.adb

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

procedure Greet is
   type My_Int is range 0 .. 1000;
   type Index is range 1 .. 5;

   type My_Int_Array is
     array (Index) of My_Int;
   --                  ^ Type of elements
   --       ^ Bounds of the array
   Arr : My_Int_Array := (2, 3, 5, 7, 11);
   --                    ^ Array literal
   --                      (aggregate)

   V : My_Int;
begin
   for I in Index loop
      V := Arr (I);
      --        ^ Take the Ith element
      Put (My_Int'Image (V));
   end loop;
   New_Line;
end Greet;
```

Build output

```
greet.adb:11:04: warning: "Arr" is not modified, could be declared constant [-
↪gnatwk]
```

Runtime output

```
 2 3 5 7 11
```

Первое, что следует отметить, - это то, что мы указываем не размер массива, а тип его индекса. Здесь мы объявили целочисленный тип с именем `Index` в диапазоне от `1` до `5`, поэтому каждый экземпляр массива будет иметь 5 элементов, с индексами от 1 до 5.

Хотя в этом примере в качестве индекса использовался целочисленный тип, в Аде любой дискретный тип может служить для индексации массива, включая (*перечислимые типы* (page 43)). Скоро мы увидим, что это значит.

Следующий момент, который следует отметить, заключается в том, что доступ к элементу массива по заданному индексу использует тот же синтаксис, что и для вызовов функций: то есть имя массива, за которым следует индекс в скобках.

Таким образом, когда вы видите выражение, такое как `A (B)`, является ли оно вызовом функции или индексом массива, зависит от того, на что ссылается `A`.

Наконец, обратите внимание, как мы инициализируем массив выражением `(2, 3, 5, 7, 11)`. Это еще один вид агрегата в Аде, который в некотором смысле является литералом для массива, точно так же, как `3` является литералом для целого числа. Обозначение очень мощное, с рядом свойств, которые мы представим позже. Подробный обзор приводится в главе о *агрегатах типов* (page 81).

Пример также иллюстрирует две процедуры из:ada:*Ada.Text_IO*, которые не связанны с массивами:

- `Put` - выводит строку без завершающего конца строки.
- `New_Line` - вывод конца строки

Давайте теперь углубимся в то, что значит иметь возможность использовать любой дискретный тип в качестве индекса массива.

На других языках

Семантически объект массива в Аде - это цельная структура данных, а не просто дескриптор или указатель. В отличие от C и C ++, не существует неявной эквивалентности между массивом и указателем на его начальный элемент.

Listing 2: array_bounds_example.adb

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

procedure Array_Bounds_Example is
   type My_Int is range 0 .. 1000;
   type Index is range 11 .. 15;
   --                  ^ Low bound can be any value
   type My_Int_Array is array (Index) of My_Int;
   Tab : constant My_Int_Array := (2, 3, 5, 7, 11);
begin
   for I in Index loop
      Put (My_Int'Image (Tab (I)));
   end loop;
   New_Line;
end Array_Bounds_Example;
```

Runtime output

```
 2 3 5 7 11
```

Как следствие границы массива могут иметь произвольные значения. В первом примере мы создали тип массива, первый индекс которого равен `1`, но в приведенном выше примере мы объявляем тип массива, первый индекс которого равен `11`.

Это прекрасно работает в Аде, а, кроме того, поскольку мы для итерации по массиву указали как диапазон тип индекса, то код использующий массив, не приходится менять.

Это приводит нас к важному принципу написания кода, оперирующего с массивами. Поскольку границы могут меняться, лучше не полагаться на конкретные значения и не

указывать их в коде использующем массив. Это означает, что приведенный выше код хорош, потому что он использует тип индекса, но цикл **for**, приведенный ниже, считается плохой практикой, даже если он работает правильно:

```
for I in 11 .. 15 loop
   Tab (I) := Tab (I) * 2;
end loop;
```

Поскольку для индексации массива можно использовать любой дискретный тип, разрешены и перечислимые типы.

Listing 3: month_example.adb

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

procedure Month_Example is
   type Month_Duration is range 1 .. 31;
   type Month is (Jan, Feb, Mar, Apr,
                  May, Jun, Jul, Aug,
                  Sep, Oct, Nov, Dec);

   type My_Int_Array is
     array (Month) of Month_Duration;
   --       ^ Can use an enumeration type
   --         as the index

   Tab : constant My_Int_Array :=
   --    ^ constant is like a variable but
   --      cannot be modified
     (31, 28, 31, 30, 31, 30,
      31, 31, 30, 31, 30, 31);
   --  Maps months to number of days
   --  (ignoring leap years)

   Feb_Days : Month_Duration := Tab (Feb);
   --  Number of days in February
begin
   for M in Month loop
      Put_Line
        (Month'Image (M) & " has "
         & Month_Duration'Image (Tab (M))
         & " days.");
   --    ^ Concatenation operator
   end loop;
end Month_Example;
```

Build output

```
month_example.adb:22:04: warning: variable "Feb_Days" is not referenced [-gnatwu]
```

Runtime output

```
JAN has  31 days.
FEB has  28 days.
MAR has  31 days.
APR has  30 days.
MAY has  31 days.
JUN has  30 days.
JUL has  31 days.
AUG has  31 days.
SEP has  30 days.
OCT has  31 days.
```

(continues on next page)

(continued from previous page)

```
NOV has  30 days.
DEC has  31 days.
```

В приведенном выше примере мы:

- Создание типа массива для отображения месяца в его продолжительность в днях.
- Создание экземпляра этого массива и инициализация его с помощью агрегата указывающего фактическую продолжительностью каждого месяца в днях.
- Итерация по массиву, печать месяцев и количество дней для каждого.

Возможность использования перечислимых значений в качестве индекса очень полезна при создании сопоставлений, как показано выше, и часто используется в Аде.

7.2 Доступ по индексу

Мы уже видели синтаксис обращения к элементам массива. Однако следует отметить еще несколько моментов.

Во-первых, как и в целом в Аде, операция индексирования строго типизирована. Если для индексации массива используется значение неправильного типа, будет получена ошибка времени компиляции.

Listing 4: greet.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Greet is
4     type My_Int is range 0 .. 1000;
5
6     type My_Index   is range 1 .. 5;
7     type Your_Index is range 1 .. 5;
8
9     type My_Int_Array is array (My_Index) of My_Int;
10    Tab : My_Int_Array := (2, 3, 5, 7, 11);
11 begin
12    for I in Your_Index loop
13       Put (My_Int'Image (Tab (I)));
14    --                          ^ Compile time error
15    end loop;
16    New_Line;
17 end Greet;
```

Build output

```
greet.adb:13:31: error: expected type "My_Index" defined at line 6
greet.adb:13:31: error: found type "Your_Index" defined at line 7
gprbuild: *** compilation phase failed
```

Во-вторых, в Аде контролируется выход за границы массива. Это означает, что если вы попытаетесь обратиться к элементу за пределами массива, вы получите ошибку во время выполнения вместо доступа к случайной памяти, как в небезопасных языках.

Listing 5: greet.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Greet is
```

(continues on next page)

(continued from previous page)

```
   type My_Int is range 0 .. 1000;
   type Index is range 1 .. 5;
   type My_Int_Array is array (Index) of My_Int;
   Tab : My_Int_Array := (2, 3, 5, 7, 11);
begin
   for I in Index range 2 .. 6 loop
      Put (My_Int'Image (Tab (I)));
      --                      ^ Will raise an
      --                        exception when
      --                        I = 6
   end loop;
   New_Line;
end Greet;
```

Build output

```
greet.adb:7:04: warning: "Tab" is not modified, could be declared constant [-
↪gnatwk]
greet.adb:9:30: warning: static value out of range of type "Index" defined at line␣
↪5 [enabled by default]
greet.adb:9:30: warning: "Constraint_Error" will be raised at run time [enabled by␣
↪default]
greet.adb:9:30: warning: suspicious loop bound out of range of loop subtype␣
↪[enabled by default]
greet.adb:9:30: warning: loop executes zero times or raises Constraint_Error␣
↪[enabled by default]
```

Runtime output

```

raised CONSTRAINT_ERROR : greet.adb:9 range check failed
```

7.3 Более простые объявления массива

В предыдущих примерах мы всегда явно создавали тип индекса для массива. Хотя это может быть полезно для типизации и удобства чтения, иногда вам просто нужно указать диапазон значений. Ада позволяет вам делать и так.

Listing 6: simple_array_bounds.adb

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

procedure Simple_Array_Bounds is
   type My_Int is range 0 .. 1000;
   type My_Int_Array is array (1 .. 5) of My_Int;
   --                          ^ Subtype of Integer
   Tab : constant My_Int_Array := (2, 3, 5, 7, 11);
begin
   for I in 1 .. 5 loop
   --       ^ Subtype of Integer
      Put (My_Int'Image (Tab (I)));
   end loop;
   New_Line;
end Simple_Array_Bounds;
```

Runtime output

```
 2 3 5 7 11
```

В этом примере диапазон массива определен с помощью синтаксиса диапазона, который указывает анонимный подтип Integer и использует его в качестве индекса массива.

Это означает, что индекс имеет целочисленный тип **Integer**. Точно так же, когда вы используете анонимный диапазон в цикле **for**, как в приведенном выше примере, тип параметра цикла также является **Integer**, поэтому вы можете использовать I для индексации Tab.

Также вы можете использовать именованный подтип для указания границ массива.

7.4 Атрибут диапазона

Ранее мы отметили, что указывать жесткие границы при итерации по массиву - плохая идея, и показали, как использовать тип/подтип индекса массива для итерации в цикле **for**. Это поднимает вопрос о том, как написать итерацию, когда массив имеет анонимный диапазон для своих границ, поскольку нет имени позволяющего сослаться на диапазон. В Аде эта проблема решается с помощью нескольких атрибутов существующих у массивов:

Listing 7: range_example.adb

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

procedure Range_Example is
   type My_Int is range 0 .. 1000;
   type My_Int_Array is array (1 .. 5) of My_Int;
   Tab : constant My_Int_Array := (2, 3, 5, 7, 11);
begin
   for I in Tab'Range loop
   --          ^ Gets the range of Tab
      Put (My_Int'Image (Tab (I)));
   end loop;
   New_Line;
end Range_Example;
```

Runtime output

```
 2 3 5 7 11
```

Если требуется более точный контроль, можно использовать отдельные атрибуты 'First («Первый») и 'Last («Последний»).

Listing 8: array_attributes_example.adb

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

procedure Array_Attributes_Example is
   type My_Int is range 0 .. 1000;
   type My_Int_Array is array (1 .. 5) of My_Int;
   Tab : My_Int_Array := (2, 3, 5, 7, 11);
begin
   for I in Tab'First .. Tab'Last - 1 loop
   --          ^ Iterate on every index
   --            except the last
      Put (My_Int'Image (Tab (I)));
   end loop;
   New_Line;
end Array_Attributes_Example;
```

Build output

```
array_attributes_example.adb:6:04: warning: "Tab" is not modified, could be␣
 ↪declared constant [-gnatwk]
```

Runtime output

```
 2 3 5 7
```

Атрибуты `'Range`, `'First` и `'Last` показанные в этих примерах также можно применять к имени типа массива, а не только к экземплярам массива.

Хотя это не демонстрируется в вышеприведенных примерах, другим полезным атрибутом для экземпляра массива A является A`'Length`, который возвращает количеством элементов в A.

Законно и иногда полезно иметь «пустой массив», который не содержит элементов. Чтобы получить его, достаточно определить диапазон индексов, верхняя граница которого меньше нижней границы.

7.5 Неограниченные массивы

Давайте теперь рассмотрим одину из самых мощных возможностей массивов в языке Ада.

Все типы массивов, который мы определили до сих пор, имеют фиксированный размер: все экземпляры такого типа будут иметь одинаковые границы и, следовательно, одинаковое количество элементов и одинаковый размер.

Но Ада также позволяет объявлять типы массивов, границы которых не являются фиксированными: в этом случае границы необходимо будет указать при создании экземпляров типа.

Listing 9: unconstrained_array_example.adb

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

procedure Unconstrained_Array_Example is
   type Days is (Monday, Tuesday, Wednesday,
                 Thursday, Friday,
                 Saturday, Sunday);

   type Workload_Type is
     array (Days range <>) of Natural;
   --  Indefinite array type
   --        ^ Bounds are of type Days,
   --          but not known

   Workload : constant
     Workload_Type (Monday .. Friday) :=
   --                ^ Specify the bounds
   --                  when declaring
      (Friday => 7, others => 8);
   --                ^ Default value
   --  ^ Specify element by name of index
begin
   for I in Workload'Range loop
      Put_Line (Integer'Image (Workload (I)));
   end loop;
end Unconstrained_Array_Example;
```

Build output

```
unconstrained_array_example.adb:4:26: warning: literal "Tuesday" is not referenced␣
 ↪[-gnatwu]
unconstrained_array_example.adb:4:35: warning: literal "Wednesday" is not␣
 ↪referenced [-gnatwu]
unconstrained_array_example.adb:5:18: warning: literal "Thursday" is not␣
 ↪referenced [-gnatwu]
unconstrained_array_example.adb:6:18: warning: literal "Saturday" is not␣
 ↪referenced [-gnatwu]
unconstrained_array_example.adb:6:28: warning: literal "Sunday" is not referenced␣
 ↪[-gnatwu]
```

Runtime output

```
8
8
8
8
7
```

Тот факт, что границы массива неизвестны, указывается синтаксисом `Days range <>`. Взяв, для примера дискретный тип `Discrete_Type`, если бы мы указали просто `Discrete_Type` как индекс массива, то для *каждого* значения из `Discrete_Type` существовал бы индекс и соответствующий элемент в *каждом* экземпляре массива.

Но, если мы определим индекс как `Discrete_Type range <>`, то, хотя `Discrete_Type` все еще будет типом индекса, но разные экземпляры массива смогут иметь свои границы.

Тип массива, который определяется с помощью синтаксиса `Discrete_Type range <>`, называется неограниченным индексируемым типом, и, как показано выше, при создании экземпляра необходимо указать границы.

В приведенном выше примере также показаны другие формы записи агрегата. Вы можете использовать именованное сопоставление, задав значение индекса слева от стрелки. Таким образом, `1 => 2` означает «присвоить значение 2 элементу с индексом 1 в моем массиве». Запись `others => 8` означает «присвоить значение 8 каждому элементу, который ранее не был назначен в этом агрегате».

> **Attention:** Так называемый «бокс» (<>) в Аде обычно используется для обозначение места, где отсутствует некий элемент. Как мы увидим еще не раз, такое обозначение можно читать, как «значение, не заданое явно».

На других языках

В то время как неограниченные массивы в Аде могут казаться похожими на массивы переменной длины в C, в действительности они гораздо более мощные, поскольку работают как настоящие значения. Их можно передавать их как параметры при вызове подпрограммам и возвращать как результат функции, и они неявно содержат границы как часть своего значения. Это означает, что нет нужды явно передавать границы или длину массива вместе с массивом, поскольку эти значения доступны через атрибуты `'First`, `'Last`, `'Range` и `'Length`, описанные ранее.

Хотя различные экземпляры одного и того же неограниченного типа массива могут иметь разные границы, конкретный экземпляр имеет постоянные границы в течение всего срока его существования. Это позволяет компилятору языка Ада эффективно реализовывать неограниченные массивы; массивы могут храниться в стеке и не требуют выделения в куче, как в других языках, типа Java.

7.6 Предопределенный тип String

В нашем введении в типы языка Ада было отмечено, что такие важные встроенные типы, как **Boolean** или **Integer**, определяются с помощью тех же средств, что доступны пользователю. Это также верно и для строк: тип **String** в Аде является простым массивом.

Вот как тип строки определяется в Аде:

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

Единственной дополнительной возможностью, которую Ада добавляет, чтобы сделать строки более простыми в использовании, являются строковые литералы, как мы можем видеть в примере ниже.

Hint: Строковые литералы являются синтаксическим сахаром для агрегатов, так что в следующем примере A и B имеют одинаковое значение.

Listing 10: string_literals.ads

```
package String_Literals is
    --  Those two declarations are equivalent
    A : String (1 .. 11) := "Hello World";
    B : String (1 .. 11) :=
        ('H', 'e', 'l', 'l', 'o', ' ',
         'W', 'o', 'r', 'l', 'd');
end String_Literals;
```

Listing 11: greet.adb

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

procedure Greet is
   Message : String (1 .. 11) := "dlroW olleH";
   --        ^ Pre-defined array type.
   --          Component type is Character
begin
   for I in reverse Message'Range loop
      --    ^ Iterate in reverse order
      Put (Message (I));
   end loop;
   New_Line;
end Greet;
```

Однако явное указание границ объекта - это немного хлопотно; вам нужно вручную подсчитать количество символов в литерале. К счастью, Ада предлагает более простой способ.

Вы можете опустить границы при создании экземпляра неограниченного типа массива, если вы предоставляете инициализацию, поскольку границы могут быть вычислены по выражению инициализации.

Listing 12: greet.adb

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

procedure Greet is
   Message : constant String := "dlroW olleH";
   --                ^ Bounds are automatically
   --                  computed from
```

(continues on next page)

(continued from previous page)

```
   --                    initialization value
begin
   for I in reverse Message'Range loop
      Put (Message (I));
   end loop;
   New_Line;
end Greet;
```

Runtime output

```
Hello World
```

Listing 13: main.adb

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

procedure Main is
   type Integer_Array is array (Natural range <>) of Integer;

   My_Array : constant Integer_Array := (1, 2, 3, 4);
   --                  ^ Bounds are automatically
   --                    computed from
   --                    initialization value
begin
    null;
end Main;
```

Attention: Как вы можете видеть выше, стандартный тип *String* в Аде - это массив. Таким образом, он сохраняет преимущества и недостатки массивов: значение **String** выделяется в стеке, доступ к нему осуществляется эффективно, а его границы неизменны.

Если вам нужно что-то вроде :c++:*std::string* в C++, вы можете использовать *Unbounded Strings* (page 233) из стандартной библиотеки Ада. Этот тип больше похож на изменяемый, автоматически управляемый строковый буфер, в который вы можете добавлять содержимое.

7.7 Ограничения

Очень важный момент, касающийся массивов: границы *должны* быть известны при создании экземпляров. Например, незаконно делать следующее.

```
declare
   A : String;
begin
   A := "World";
end;
```

Кроме того, хотя вы, конечно, можете изменять значения элементов в массиве, вы не можете изменять границы массива (и, следовательно, его размер) после его инициализации. Так что это тоже незаконно:

```
declare
   A : String := "Hello";
begin
   A := "World";        --  OK: Same size
```

(continues on next page)

(continued from previous page)

```
   A := "Hello World"; --  Not OK: Different size
end;
```

Кроме того, хотя вы можете ожидать предупреждения об ошибке такого рода в очень простых случаях, подобных этому, но компилятор в общем случае не может знать, присваиваете ли вы значение правильной длины, поэтому это нарушение обычно приведет к ошибке во время выполнения.

Обратите внимание

Хотя мы остановимся на этом позже, важно знать, что массивы - не единственные типы, экземпляры которых могут быть неизвестного размера в момент компиляции.

Говорят, что такие объекты имеют *неопределенный подтип*, что означает, что размер подтипа неизвестен во время компиляции, но динамически вычисляется (во время выполнения).

Listing 14: indefinite_subtypes.adb

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

procedure Indefinite_Subtypes is
    function Get_Number return Integer is
    begin
        return Integer'Value (Get_Line);
    end Get_Number;

   A : String := "Hello";
   --  Indefinite subtype

   B : String (1 .. 5) := "Hello";
   --  Definite subtype

   C : String (1 .. Get_Number);
   --  Indefinite subtype
   --  (Get_Number's value is computed at
   --  run-time)
begin
   null;
end Indefinite_Subtypes;
```

7.8 Возврат неограниченных массивов

Тип возвращаемого значения функции может быть любым; функция может возвращать значение, размер которого неизвестен во время компиляции. Точно так же параметры могут быть любого типа.

Например, это функция, которая возвращает неограниченную строку:

Listing 15: main.adb

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

procedure Main is

   type Days is (Monday, Tuesday, Wednesday,
```

(continues on next page)

(continued from previous page)

```
                  Thursday, Friday,
                  Saturday, Sunday);

   function Get_Day_Name (Day : Days := Monday)
                          return String is
   begin
      return
        (case Day is
         when Monday    => "Monday",
         when Tuesday   => "Tuesday",
         when Wednesday => "Wednesday",
         when Thursday  => "Thursday",
         when Friday    => "Friday",
         when Saturday  => "Saturday",
         when Sunday    => "Sunday");
   end Get_Day_Name;

begin
   Put_Line ("First day is "
             & Get_Day_Name (Days'First));
end Main;
```

Runtime output

```
First day is Monday
```

Примечание. Этот пример приведен только в иллюстративных целях. Существует встроенный механизм, атрибут `'Image` для скалярных типов, который возвращает имя (в виде строки – типа **`String`**) любого элемента типа перечисления. Например, `Days'Image(Monday)` есть `"MONDAY"`.)

На других языках

Возврат объектов переменного размера в языках, в которых отсутствует сборщик мусора, довольно сложен с точки зрения реализации, поэтому C и C++ не допускают такого, предпочитая зависеть от явного динамического распределения/освобождения памяти пользователем.

Проблема в том, что явное управление памятью становится небезопасным, как только вы захотите использовать выделенную память повторно. Способность Ада возвращать объекты переменного размера устраняют необходимость динамического распределения для одгого варианта использования и, следовательно, удаляет один потенциальный источник ошибок из ваших программ.

Rust следует модели C/C++, но использует указатели с безопасной семантикой. При этом динамическое распределение все еще используется. Ада может извлечь выгоду в виде повышения производительности, поскольку оставляет возможность использовать любой из этих механизмов.

7.9 Объявление массивов (2)

Хотя мы можем иметь типы массивов, размер и границы которых определяются во время выполнения, тип компоненты массива всегда должен быть определенного и ограниченного типа.

Таким образом, если необходимо объявить, например, массив строк, подтип **String**, используемый в качестве компоненты, должен иметь фиксированный размер.

Listing 16: show_days.adb

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

procedure Show_Days is
   type Days is (Monday, Tuesday, Wednesday,
                 Thursday, Friday,
                 Saturday, Sunday);

   subtype Day_Name is String (1 .. 2);
   --  Subtype of string with known size

   type Days_Name_Type is
     array (Days) of Day_Name;
   --       ^ Type of the index
   --                 ^ Type of the element.
   --                   Must be definite

   Names : constant Days_Name_Type :=
     ("Mo", "Tu", "We", "Th", "Fr", "Sa", "Su");
   --  Initial value given by aggregate
begin
   for I in Names'Range loop
      Put_Line (Names (I));
   end loop;
end Show_Days;
```

Build output

```
show_days.adb:4:18: warning: literal "Monday" is not referenced [-gnatwu]
show_days.adb:4:26: warning: literal "Tuesday" is not referenced [-gnatwu]
show_days.adb:4:35: warning: literal "Wednesday" is not referenced [-gnatwu]
show_days.adb:5:18: warning: literal "Thursday" is not referenced [-gnatwu]
show_days.adb:5:28: warning: literal "Friday" is not referenced [-gnatwu]
show_days.adb:6:18: warning: literal "Saturday" is not referenced [-gnatwu]
show_days.adb:6:28: warning: literal "Sunday" is not referenced [-gnatwu]
```

Runtime output

```
Mo
Tu
We
Th
Fr
Sa
Su
```

7.10 Отрезки массива

Последняя особенность массивов Аде, которую мы собираемся рассмотреть, - это отрезки массива. Можно взять и использовать отрезок массива (непрерывную последовательность элементов) в качестве имени или значения.

Listing 17: main.adb

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

procedure Main is
    Buf : String := "Hello ...";

    Full_Name : String := "John Smith";
begin
    Buf (7 .. 9) := "Bob";
    --  Careful! This works because the string
    --  on the right side is the same length as
    --  the replaced slice!

    --  Prints "Hello Bob"
    Put_Line (Buf);

     --  Prints "Hi John"
    Put_Line ("Hi " & Full_Name (1 .. 4));
end Main;
```

Build output

```
main.adb:6:05: warning: "Full_Name" is not modified, could be declared constant [-
↪gnatwk]
```

Runtime output

```
Hello Bob
Hi John
```

Как мы видим выше, вы можете использовать отрезок слева от присваивания, чтобы заменить только часть массива.

Отрезок массива имеет тот же тип, что и массив, но имеет другой подтип, ограниченния которого заданы границами отрезка.

Attention: В Ада есть многомерные массивы[12], которые не рассматриваются в этом курсе. Отрезки будут работать только с одномерными массивами.

[12] http://www.adaic.org/resources/add_content/standards/12rm/html/RM-3-6.html

7.11 Переименование

До сих пор мы видели, что следующие элементы могут быть переименованы: *подпрограммы* (page 25), *пакеты* (page 36) и компоненты записи. Мы также можем переименовывать объекты с помощью ключевого слова **renames**. Это позволяет создавать альтернативные имена для данных объектов. Давайте посмотрим на пример:

Listing 18: measurements.ads

```
package Measurements is

   subtype Degree_Celsius is Float;

   Current_Temperature : Degree_Celsius;

end Measurements;
```

Listing 19: main.adb

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

procedure Main is
   subtype Degrees is Measurements.Degree_Celsius;

   T : Degrees
     renames Measurements.Current_Temperature;
begin
    T := 5.0;

    Put_Line (Degrees'Image (T));
    Put_Line (Degrees'Image
      (Measurements.Current_Temperature));

    T := T + 2.5;

    Put_Line (Degrees'Image (T));
    Put_Line (Degrees'Image
      (Measurements.Current_Temperature));
end Main;
```

Runtime output

```
 5.00000E+00
 5.00000E+00
 7.50000E+00
 7.50000E+00
```

В приведенном выше примере мы объявляем переменную T, переименовывая объект `Current_Temperature` из пакета `Measurements`. Как вы можете видеть, запустив этот пример, и `Current_Temperature`, и его альтернативное имя T имеют одинаковые значения:

- сначала они показывают значение 5.0
- после сложения они показывают значение 7.5.

Это потому, что они по существу обозначают однин и тот же объект, но с двумя разными именами.

Обратите внимание, что в приведенном выше примере мы используем `Degrees` как псевдоним `Degree_Celsius`. Мы обсуждали этот метод переименования *ранее в курсе* (page 54).

Переименование может быть полезно для улучшения читаемости кода со сложной индексацией массивов. Вместо того, чтобы явно использовать индексы каждый раз, когда мы обращаемся к определенным позициям массива, мы можем создавать короткие имена для этих позиций, переименовывая их. Давайте посмотрим на следующий пример:

Listing 20: colors.ads

```
package Colors is

   type Color is (Black, Red, Green, Blue, White);

   type Color_Array is
     array (Positive range <>) of Color;

   procedure Reverse_It (X : in out Color_Array);

end Colors;
```

Listing 21: colors.adb

```
package body Colors is

   procedure Reverse_It (X : in out Color_Array) is
   begin
      for I in X'First .. (X'Last + X'First) / 2 loop
         declare
            Tmp     : Color;
            X_Left  : Color
              renames X (I);
            X_Right : Color
              renames X (X'Last + X'First - I);
         begin
            Tmp     := X_Left;
            X_Left  := X_Right;
            X_Right := Tmp;
         end;
      end loop;
   end Reverse_It;

end Colors;
```

Listing 22: test_reverse_colors.adb

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

with Colors; use Colors;

procedure Test_Reverse_Colors is

   My_Colors : Color_Array (1 .. 5) :=
     (Black, Red, Green, Blue, White);

begin
   for C of My_Colors loop
      Put_Line ("My_Color: " & Color'Image (C));
   end loop;

   New_Line;
   Put_Line ("Reversing My_Color...");
   New_Line;
   Reverse_It (My_Colors);
```

(continues on next page)

(continued from previous page)

```
19
20    for C of My_Colors loop
21       Put_Line ("My_Color: " & Color'Image (C));
22    end loop;
23
24 end Test_Reverse_Colors;
```

Runtime output

```
My_Color: BLACK
My_Color: RED
My_Color: GREEN
My_Color: BLUE
My_Color: WHITE

Reversing My_Color...

My_Color: WHITE
My_Color: BLUE
My_Color: GREEN
My_Color: RED
My_Color: BLACK
```

В приведенном выше примере пакет `Colors` содержит процедуру `Reverse_It`, где объявлены новые имена для двух позиций массива. Таким образом реализация становится легко читаемой:

```
begin
   Tmp     := X_Left;
   X_Left  := X_Right;
   X_Right := Tmp;
end;
```

Сравните это с альтернативной версией без переименования:

```
begin
   Tmp                       := X (I);
   X (I)                     := X (X'Last + X'First - I);
   X (X'Last + X'First - I) := Tmp;
end;
```

CHAPTER

EIGHT

ПОДРОБНЕЕ О ТИПАХ

8.1 Агрегаты: краткая информация

До сих пор мы говорили об агрегатах довольно мало и видели лишь ряд примеров. Теперь мы рассмотрим эту конструкцию более подробно.

Агрегат в Аде фактически является литералом составного типа. Это очень мощная форма записи во многих случаях позволяет избежать написания процедурного кода для инициализации структур данных.

Основным правилом при записи агрегатов является то, что *каждый компонент* массива или записи должен быть указан, даже компоненты, которые имеют значение по умолчанию.

Это означает, что следующий код неверен:

Listing 1: incorrect.ads

```
1  package Incorrect is
2     type Point is record
3        X, Y : Integer := 0;
4     end record;
5
6     Origin : Point := (X => 0);
7  end Incorrect;
```

Build output

```
incorrect.ads:6:22: error: no value supplied for component "Y"
gprbuild: *** compilation phase failed
```

Существует несколько сокращений, которые можно использовать, чтобы сделать представление более удобным:

- Чтобы задать значение по умолчанию для компоненты, можно использовать нотацию <>.
- Символ | можно использовать для присвоения нескольким компонентам одинакового значения.
- Вы можете использовать `others` вариант для ссылки на все компоненты, которые еще не были указаны, при условии, что все эти поля имеют одинаковый тип.
- Можно использовать нотацию диапазона `..` чтобы указать непрерывную последовательность индексов в массиве.

Однако следует отметить, что, как только вы использовали именованное сопоставление, все последующие компоненты также должны быть указаны с помощью именованного сопоставления.

Listing 2: points.ads

```
package Points is
   type Point is record
      X, Y : Integer := 0;
   end record;

   type Point_Array is
     array (Positive range <>) of Point;

   --  use the default values
   Origin   : Point := (X | Y => <>);

   --  likewise, use the defaults
   Origin_2 : Point := (others => <>);

   Points_1 : Point_Array := ((1, 2), (3, 4));
   Points_2 : Point_Array := (1       => (1, 2),
                              2       => (3, 4),
                              3 .. 20 => <>);
end Points;
```

8.2 Совмещение и квалифицированные выражения

В Ада есть общая концепция совмещения имен, которую мы видели ранее в разделе о *перечислимых типах* (page 43).

Давайте возьмем простой пример: в Аде возможно иметь функции с одинаковым именем, но разными типами для их параметров.

Listing 3: pkg.ads

```
package Pkg is
   function F (A : Integer) return Integer;
   function F (A : Character) return Integer;
end Pkg;
```

Это распространенное понятие в языках программирования, называемое совмещением, или перегрузкой имен[13].

Одной из особенностей совмещения имен в Аде является возможность разрешить неоднозначности на основе типа возвращаемого функцией.

Listing 4: pkg.ads

```
package Pkg is
   type SSID is new Integer;

   function Convert (Self : SSID) return Integer;
   function Convert (Self : SSID) return String;
end Pkg;
```

Listing 5: main.adb

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

```

(continues on next page)

[13] https://ru.m.wikipedia.org/wiki/Перегрузка_процедур_и_функций

(continued from previous page)

```
procedure Main is
   S : String := Convert (123_145_299);
   --             ^ Valid, will choose the
   --               proper Convert
begin
   Put_Line (S);
end Main;
```

Attention: Заметим, что разрешение совмещения на основе типа в Аде работает как для функций, так и для литералов перечисления - вот почему у вас может быть несколько литералов перечисления с одинаковым именем. Семантически литерал перечисления рассматривается как функция, не имеющая параметров.

Однако иногда возникает двусмысленность из-за которой невозможно определить, к какому объявлению совмещенного имени относится данное употребление имени. Именно здесь становится полезным квалифицированное выражение.

Listing 6: pkg.ads

```
package Pkg is
   type SSID is new Integer;

   function Convert (Self : SSID) return Integer;
   function Convert (Self : SSID) return String;
   function Convert (Self : Integer) return String;
end Pkg;
```

Listing 7: main.adb

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

procedure Main is
   S : String := Convert (123_145_299);
   --             ^ Invalid, which convert
   --               should we call?

   S2 : String := Convert (SSID'(123_145_299));
   --                      ^ We specify that the
   --                        type of the expression
   --                        is SSID.

   --  We could also have declared a temporary

   I : SSID := 123_145_299;

   S3 : String := Convert (I);
begin
   Put_Line (S);
end Main;
```

Синтаксически квалифицированное выражение используется либо с выражением в круглых скобках, либо с агрегатом:

Listing 8: qual_expr.ads

```
package Qual_Expr is
   type Point is record
```

(continues on next page)

(continued from previous page)

```
      A, B : Integer;
   end record;

   P : Point := Point'(12, 15);

   A : Integer := Integer'(12);
end Qual_Expr;
```

Это иллюстрирует, что квалифицированные выражения являются удобным (а иногда и необходимым) способом явно обозначить тип выражения и помочь, как компилятору, так и для другим программистам разобраться в коде.

Attention: Хотя преобразования типов и квалифицированные выражения выглядят и ощущаются похожими это *не* одно и тоже.

Квалифицированное выражение указывает точный тип, в котором следует трактовать выражение, в то время как преобразование типа попытается преобразовать значение и выдаст ошибку во время выполнения, если исходное значение не может быть преобразовано.

Обратите внимание, что вы можете использовать квалифицированное выражение для преобразования из одного подтипа в другой, с исключением, возникающим при нарушении ограничения.

```
X : Integer := Natural'(1);
```

8.3 Символьные типы

Как отмечалось ранее, каждый перечислимый тип отличается и несовместим с любым другим перечислимым типом. Однако ранее мы не упоминали, что символьные литералы разрешены в качестве литералов перечисления. Это означает, что в дополнение к стандартным символьным типам пользователь может определить свои символьные типы:

Listing 9: character_example.adb

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

procedure Character_Example is
   type My_Char is ('a', 'b', 'c');
   --  Our custom character type, an
   --  enumeration type with 3 valid values.

   C : Character;
   --  ^ Built-in character type
   --    (it's an enumeration type)

   M : My_Char;
begin
   C := '?';
   --   ^ Character literal
   --     (enumeration literal)

   M := 'a';

   C := 65;
   --   ^ Invalid: 65 is not a
```

(continues on next page)

(continued from previous page)

```
   --     Character value

   C := Character'Val (65);
   --  Assign the character at
   --  position 65 in the
   --  enumeration (which is 'A')

   M := C;
   --   ^ Invalid: C is of type Character,
   --     and M is a My_Char

   M := 'd';
   --   ^ Invalid: 'd' is not a valid
   --     literal for type My_Char
end Character_Example;
```

Build output

```
character_example.adb:14:04: warning: useless assignment to "C", value overwritten␣
 ↪at line 20 [-gnatwm]
character_example.adb:18:04: warning: useless assignment to "M", value overwritten␣
 ↪at line 29 [-gnatwm]
character_example.adb:20:04: warning: useless assignment to "C", value overwritten␣
 ↪at line 24 [-gnatwm]
character_example.adb:20:09: error: expected type "Standard.Character"
character_example.adb:20:09: error: found type universal integer
character_example.adb:29:04: warning: useless assignment to "M", value overwritten␣
 ↪at line 33 [-gnatwm]
character_example.adb:29:09: error: expected type "My_Char" defined at line 4
character_example.adb:29:09: error: found type "Standard.Character"
character_example.adb:33:04: warning: possibly useless assignment to "M", value␣
 ↪might not be referenced [-gnatwm]
character_example.adb:33:09: error: character not defined for type "My_Char"␣
 ↪defined at line 4
gprbuild: *** compilation phase failed
```

CHAPTER

NINE

ССЫЛОЧНЫЕ ТИПЫ (УКАЗАТЕЛИ)

9.1 Введение

Указатели - потенциально опасная конструкция, которая вступает в противоречие с основополагающей философией Ады.

Есть два способа, которыми Ада помогает оградить программистов от опасности указателей:

1. Один из подходов, который мы уже видели, заключается в обеспечении альтернативных возможностей, чтобы программисту не нужно было использовать указатели. Виды параметров, массивы и типы произвольных размеров - это все конструкции, которые могут заменить типичное использование указателей в C.
2. Во-вторых, Ада сделала указатели максимально безопасными и ограниченными, хотя допускает «аварийные люки», которые программист явно затребовать и, предположительно, будет использовать их с соответствующей осторожностью.

Вот как в Аде объявляется простой тип указателя, ссылочный тип:

Listing 1: dates.ads

```
package Dates is
   type Months is
     (January, February, March, April,
      May, June, July, August, September,
      October, November, December);

   type Date is record
      Day   : Integer range 1 .. 31;
      Month : Months;
      Year  : Integer;
   end record;
end Dates;
```

Listing 2: access_types.ads

```
with Dates; use Dates;

package Access_Types is
    --  Declare an access type
    type Date_Acc is access Date;
    --                      ^ "Designated type"
    --                      ^ Date_Acc values point
    --                        to Date objects

    D : Date_Acc := null;
    --             ^ Literal for
    --               "access to nothing"
```

(continues on next page)

(continued from previous page)

```
13      --  ^ Access to date
14  end Access_Types;
```

На этом примере показано, как:

- Объявить ссылочный тип, значения которого *указывают* на объекты определенного типа
- Объявить переменную (для ссылочных значений) этого ссылочного типа
- Присвоить ему значение **null**

В соответствии с философией строгой типизации Ада, если объявить второй ссылочный тип, указывающий на Date, эти два ссылочных типа будут несовместимы друг с другом:

Listing 3: access_types.ads

```
1   with Dates; use Dates;
2
3   package Access_Types is
4       --  Declare an access type
5       type Date_Acc   is access Date;
6       type Date_Acc_2 is access Date;
7
8       D  : Date_Acc   := null;
9       D2 : Date_Acc_2 := D;
10      --                 ^ Invalid! Different types
11  end Access_Types;
```

Build output

```
access_types.ads:9:24: error: expected type "Date_Acc_2" defined at line 6
access_types.ads:9:24: error: found type "Date_Acc" defined at line 5
gprbuild: *** compilation phase failed
```

На других языках

Большинство других языков используют структурно типизацию для указателей, что означает, что два типа указателей считаются одинаковыми, если они имеют один и тот же целевой тип.

В Аде это не так, и к этому, возможно, придется какое-то время привыкать. Казалось бы, простой вопрос, если вы хотите для вашего типа иметь канонический ссылочный тип, где его объявить? Обычно используют следующий подход, если понадобится ссылочный тип для некоторого вашего типа, то вы объявите его вместе с типом:

```
package Access_Types is
   type Point is record
      X, Y : Natural;
   end record;

   type Point_Access is access Point;
end Access_Types;
```

9.2 Выделение (allocation) памяти

После того, как мы объявили ссылочный тип, нам нужен способ присвоить переменным этого типа осмысленные значения! Вы можете получить значение ссылочного типа с помощью ключевого слова **new**.

Listing 4: access_types.ads

```
with Dates; use Dates;

package Access_Types is
    type Date_Acc is access Date;

    D : Date_Acc := new Date;
    --              ^ Allocate a new Date record
end Access_Types;
```

Если тип, память для значение которого требуется выделить, требует ограничений, их можно указать после подтипа, как в объявлении переменной:

Listing 5: access_types.ads

```
with Dates; use Dates;

package Access_Types is
   type String_Acc is access String;
   --                        ^
   --  Access to unconstrained array type
   Msg : String_Acc;
   --    ^ Default value is null

   Buffer : String_Acc :=
     new String (1 .. 10);
   --            ^ Constraint required
end Access_Types;
```

Build output

```
access_types.ads:1:06: warning: no entities of "Dates" are referenced [-gnatwu]
access_types.ads:1:13: warning: use clause for package "Dates" has no effect [-
 ↪gnatwu]
```

Однако, в некоторых случаях выделение памяти путем указания типа не является идеальным, поэтому Ада позволяет инициализировать объект одновременно с выделением памяти. Чтобы сделать это необходимо использовать квалифицированное выражение:

Listing 6: access_types.ads

```
with Dates; use Dates;

package Access_Types is
   type Date_Acc is access Date;
   type String_Acc is access String;

   D   : Date_Acc   := new Date'(30, November, 2011);
   Msg : String_Acc := new String'("Hello");
end Access_Types;
```

9.3 Извлечение по ссылке

Последняя часть мозайки ссылочных типов языка Ада покажет нам, как получить значение объекту по ссылке, то есть как «разыменовать» указатель. Для этого в Аде используется синтаксис **.all**, но часто в нем вообще нет необходимости - во многих случаях использования ссылочного значения эта операция выполняется неявно:

Listing 7: access_types.ads

```
with Dates; use Dates;

package Access_Types is
   type Date_Acc is access Date;

   D     : Date_Acc := new Date'(30, November, 2011);

   Today : Date := D.all;
   --             ^ Access value dereference
   J     : Integer := D.Day;
   --                 ^ Implicit dereference for
   --                   record and array components
   --                   Equivalent to D.all.day
end Access_Types;
```

9.4 Другие особенности

Как вы, возможно, заметили, если пользовались указатели в C или C++, мы не показали некоторых функций, которые считаются основополагающими при использования указателей, таких как:

- Арифметика над указателями (возможность увеличивать или уменьшать указатель, чтобы переместить его на следующий или предыдущий объект)
- Освобождение памяти вручную - то, что в C делается с помощью `free` или `delete`. Это потенциально опасная операция. Чтобы оставаться в безопасном пространстве Ады, вам лучше не освобождать память вручную.

Эти функции существуют в Аде, но воспользоваться ими можно только с помощью определенных интерфейсов стандартной библиотеки (API).

> **Attention:** Общепринятый принцип языка гласит, что в большенстве случаев вы можете избегать ручного управления памятью, и вам лучше следовать ему.

Существует множество способов избежать распределения памяти вручную, с некоторыми из них мы уже встречались (например, виды параметров). Язык также предоставляет библиотечные абстракции, чтобы избежать указателей:

1. Одним из них является использование контейнеров. Контейнеры помогают пользователям избегать указателей, поскольку сами управляют памятью.
2. Контейнер, который следует отметить в этом контексте, является Indefinite holder[14]. Этот контейнер позволяет хранить значение неопределенного типа, например String.
3. GNATCOLL имеет библиотеку для интеллектуальных указателей, называемую Refcount[15], память этих указателей автоматически управляется, так что, когда у выделенного объекта больше нет ссылок на него, память автоматически освобождается.

9.5 Взаимно рекурсивные типы

Связанный список является широко известной идиомой в программировании; в Аде его наиболее естественная запись включает определение двух типов - тип записи и ссфлочный тип, которые будут взаимно зависимыми. Для объявления взаимно зависимых типов можно использовать неполное объявление типа:

Listing 8: simple_list.ads

```
1  package Simple_List is
2     type Node;
3     --  This is an incomplete type declaration,
4     --  which is completed in the same
5     --  declarative region.
6
7     type Node_Acc is access Node;
8
9     type Node is record
10       Content    : Natural;
11       Prev, Next : Node_Acc;
12    end record;
13 end Simple_List;
```

[14] http://www.ada-auth.org/standards/12rat/html/Rat12-8-5.html
[15] https://github.com/AdaCore/gnatcoll-core/blob/master/src/gnatcoll-refcount.ads

CHAPTER

TEN

ПОДРОБНЕЕ О ЗАПИСЯХ

10.1 Типы записей динамически изменяемого размера

Ранее мы видели *несколько простых примеров типов записей* (page 57). Давайте рассмотрим некоторые из более продвинутых возможностей этой фундаментальной конструкции языка Ада.

Следует отметить, что размер объекта для типа записи не обязательно должен быть известен во время компиляции. Это проиллюстрировано в приведенном ниже примере:

Listing 1: runtime_length.ads

```
package Runtime_Length is
   function Compute_Max_Len return Natural;
end Runtime_Length;
```

Listing 2: var_size_record.ads

```
with Runtime_Length; use Runtime_Length;

package Var_Size_Record is
    Max_Len : constant Natural
      := Compute_Max_Len;
    --    ^ Not known at compile time

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

    type Growable_Stack is record
       Items : Items_Array (1 .. Max_Len);
       Len   : Natural;
    end record;
    --  Growable_Stack is a definite type, but
    --  size is not known at compile time.

    G : Growable_Stack;
end Var_Size_Record;
```

Совершенно нормально определять размер ваших записей во время выполнения, но учтите, что все объекты этого типа будут иметь одинаковый размер.

10.2 Записи с дискриминантом

В приведенном выше примере размер поля Items определяется один раз во время выполнения, но размер всех экземпляров `Growable_Stack` будет совпадать. И, возможно, это не то, что вы хотите получить. Мы видели, что массивы в целом имеют такую гибкость: для неограниченного типа массива разные объекты могут иметь разные размеры.

Получить аналогичную функциональность для записей можно используя специальную разновидность компонент, которые называются дискриминантами:

Listing 3: var_size_record_2.ads

```
package Var_Size_Record_2 is
    type Items_Array is array (Positive range <>)
      of Integer;

    type Growable_Stack (Max_Len : Natural) is
    record
    --                    ^ Discriminant. Cannot be
    --                      modified once initialized.
       Items : Items_Array (1 .. Max_Len);
       Len   : Natural := 0;
    end record;
    --  Growable_Stack is an indefinite type
    --  (like an array)
end Var_Size_Record_2;
```

Дискриминанты, грубо говоря, являются константами: вы не можете изменять их значение после инициализации объекта. Это интуитивно понятно, поскольку они определяют размер объекта.

Кроме того, они делают тип неопределенным. В независимости от того, используется ли дискриминант для указания размера объекта или нет, тип с дискриминантом считается неопределенным, пока дискриминант не имеет выражение для инициализации:

Listing 4: test_discriminants.ads

```
package Test_Discriminants is
   type Point (X, Y : Natural) is record
      null;
   end record;

   P : Point;
   --  ERROR: Point is indefinite, so you
   --  need to specify the discriminants
   --  or give a default value

   P2 : Point (1, 2);
   P3 : Point := (1, 2);
   --  Those two declarations are equivalent.

end Test_Discriminants;
```

Build output

```
test_discriminants.ads:6:08: error: unconstrained subtype not allowed (need␣
 ↪initialization)
test_discriminants.ads:6:08: error: provide initial value or explicit discriminant␣
 ↪values
test_discriminants.ads:6:08: error: or give default discriminant values for type
 ↪"Point"
gprbuild: *** compilation phase failed
```

Это также означает, что в приведенном выше примере вы не можете объявить массив значений Point, потому что размер Point неизвестен.

Как упоминалось выше, мы могли бы предоставить значение по умолчанию для дискриминантов, чтобы легально объявлять переменные типа `Point` без указания значения дискриминантов. В приведенном выше примере это будет выглядеть так:

Listing 5: test_discriminants.ads

```
package Test_Discriminants is
   type Point (X, Y : Natural := 0) is record
      null;
   end record;

   P : Point;
   --  We can now simply declare a "Point"
   --  without further ado. In this case,
   --  we're using the default values (0)
   --  for X and Y.

   P2 : Point (1, 2);
   P3 : Point := (1, 2);
   --  We can still specify discriminants.

end Test_Discriminants;
```

Также обратите внимание, что, хотя тип `Point` теперь имеет дискриминанты по умолчанию, это не мешает нам указывать дискриминанты, как мы это делаем в объявлениях P2 и P3.

Во многих других отношениях дискриминанты ведут себя как обычные поля: вы должны указать их значения в агрегатах, как показано выше, и вы можете извлекать их значения с помощью точечной нотации.

Listing 6: main.adb

```
with Var_Size_Record_2; use Var_Size_Record_2;
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
   procedure Print_Stack (G : Growable_Stack) is
   begin
      Put ("<Stack, items: [");
      for I in G.Items'Range loop
         exit when I > G.Len;
         Put (" " & Integer'Image (G.Items (I)));
      end loop;
      Put_Line ("]>");
   end Print_Stack;

   S : Growable_Stack :=
     (Max_Len => 128,
      Items   => (1, 2, 3, 4, others => <>),
      Len     => 4);
begin
   Print_Stack (S);
end Main;
```

Build output

```
main.adb:15:04: warning: "S" is not modified, could be declared constant [-gnatwk]
```

Runtime output

```
<Stack, items: [  1  2  3  4]>
```

Note: В примере выше, мы использовали дискриминант чтобы указать размер массива, но возможны и другие применения, например, определение дискриминанта вложенной записи.

10.3 Записи с вариантами

Ранее мы привели примеры использования дискриминантов для объявление записей разлного размера, содержащих компоненты, размер которых зависит от дискриминанта.

Но с помощью дискриминантов также можно построить конструкцию часто именуемую «запись с вариантами»: это записи, которые могут содержать разные наборы полей.

Listing 7: variant_record.ads

```
package Variant_Record is
   --  Forward declaration of Expr
   type Expr;

   --  Access to a Expr
   type Expr_Access is access Expr;

   type Expr_Kind_Type is (Bin_Op_Plus,
                           Bin_Op_Minus,
                           Num);
   --  A regular enumeration type

   type Expr (Kind : Expr_Kind_Type) is record
      --       ^ The discriminant is an
      --         enumeration value
      case Kind is
         when Bin_Op_Plus | Bin_Op_Minus =>
            Left, Right : Expr_Access;
         when Num =>
            Val : Integer;
      end case;
      --  Variant part. Only one, at the end of
      --  the record definition, but can be
      --  nested
   end record;
end Variant_Record;
```

Поля, которые находятся в варианте **when**, будут доступны только тогда, когда значение дискриминанта совпадает с указанным. В приведенном выше примере вы сможете обращаться к полям `Left` и `Right`, только если `Kind` равен `Bin_Op_Plus` или `Bin_Op_Minus`.

Если вы попытаетесь получить доступ к полю, когда значение дискриминанта не совпадает, будет возбуждено исключение `Constraint_Error`.

Listing 8: main.adb

```
with Variant_Record; use Variant_Record;

procedure Main is
   E : Expr := (Num, 12);
begin
```

(continues on next page)

(continued from previous page)

```
   E.Left := new Expr'(Num, 15);
   --  Will compile but fail at runtime
end Main;
```

Build output

```
main.adb:4:04: warning: variable "E" is not referenced [-gnatwu]
main.adb:6:05: warning: component not present in subtype of "Expr" defined at line␣
↪4 [enabled by default]
main.adb:6:05: warning: "Constraint_Error" will be raised at run time [enabled by␣
↪default]
```

Runtime output

```
raised CONSTRAINT_ERROR : main.adb:6 discriminant check failed
```

А вот как можно написать вычислитель выражений:

Listing 9: main.adb

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

procedure Main is
   function Eval_Expr (E : Expr) return Integer is
     (case E.Kind is
      when Bin_Op_Plus  => Eval_Expr (E.Left.all)
                           + Eval_Expr (E.Right.all),
      when Bin_Op_Minus => Eval_Expr (E.Left.all)
                           - Eval_Expr (E.Right.all),
      when Num => E.Val);

   E : Expr := (Bin_Op_Plus,
                new Expr'(Bin_Op_Minus,
                          new Expr'(Num, 12),
                          new Expr'(Num, 15)),
                new Expr'(Num, 3));
begin
   Put_Line (Integer'Image (Eval_Expr (E)));
end Main;
```

Build output

```
main.adb:13:04: warning: "E" is not modified, could be declared constant [-gnatwk]
```

Runtime output

```
 0
```

На других языках

Записи вариантов Аде очень похожи на Sum типы в функциональных языках, таких как OCaml или Haskell. Основное отличие состоит в том, что дискриминант является отдельным полем в Аде, тогда как «тег» Sum типа является встроенным и доступен только при сопоставлении шаблонов.

Есть и другие различия (записи с вариантами в Аде могут иметь несколько дискриминантов). Тем не менее, они допускают тот же подход к моделированию, что и типы Sum функциональных языков.

По сравнению с объединениями C/C++ записи с вариантами Аде более мощны, а также благодаря проверкам во время выполнения, более безопасны.

CHAPTER

ELEVEN

ТИПЫ С ФИКСИРОВАННОЙ ЗАПЯТОЙ

11.1 Десятичные типы с фиксированной запятой

Мы уже видели, как определять типы с плавающей запятой. Однако в некоторых приложениях плавающая запятая не подходит, например, ошибка округления при двоичной арифметики неприемлема или, оборудование не поддерживает инструкции с плавающей запятой. В языке Ада есть десятичные типы с фиксированной запятой, которые позволяют программисту указать требуемую десятичную точность (количество цифр), а также коэффициент масштабирования (степень десяти) и, необязательно, диапазон. Фактически, значения такого типа будут представлены как целые числа, неявно масштабированные с указанной степенью 10. Это полезно, например, для финансовых приложений.

Синтаксис простого десятичного типа с фиксированной запятой:

```
type <type-name> is delta <delta-value> digits <digits-value>;
```

В этом случае **delta** и **digits** будут использоваться компилятором для вычисления диапазона значений.

Несколько атрибутов полезны при работе с десятичными типами:

Имя атрибута	Значение
First	Наименьшее значение типа
Last	Наибольшее значение типа
Delta	Значение минимального шага типа

В приведенном ниже примере мы объявляем два типа данных: `T3_D3` и `T6_D3`. Для обоих типов значение дельты одинаково: 0.001.

Listing 1: decimal_fixed_point_types.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Decimal_Fixed_Point_Types is
4     type T3_D3 is delta 10.0 ** (-3) digits 3;
5     type T6_D3 is delta 10.0 ** (-3) digits 6;
6  begin
7     Put_Line ("The delta    value of T3_D3 is "
8               & T3_D3'Image (T3_D3'Delta));
9     Put_Line ("The minimum  value of T3_D3 is "
10              & T3_D3'Image (T3_D3'First));
11    Put_Line ("The maximum  value of T3_D3 is "
12              & T3_D3'Image (T3_D3'Last));
13    New_Line;
14
15    Put_Line ("The delta    value of T6_D3 is "
16              & T6_D3'Image (T6_D3'Delta));
```

(continues on next page)

(continued from previous page)

```
17    Put_Line ("The minimum  value of T6_D3 is "
18              & T6_D3'Image (T6_D3'First));
19    Put_Line ("The maximum  value of T6_D3 is "
20              & T6_D3'Image (T6_D3'Last));
21 end Decimal_Fixed_Point_Types;
```

Runtime output

```
The delta    value of T3_D3 is  0.001
The minimum  value of T3_D3 is -0.999
The maximum  value of T3_D3 is  0.999

The delta    value of T6_D3 is  0.001
The minimum  value of T6_D3 is -999.999
The maximum  value of T6_D3 is  999.999
```

При запуске приложения мы видим, что значение дельты обоих типов действительно одинаково: 0.001. Однако, поскольку T3_D3 ограничен 3 цифрами, его диапазон составляет от -0,999 до 0,999. Для T6_D3 мы определили точность 6 цифр, поэтому диапазон от -999,999 до 999,999.

Аналогично определению типа с использованием синтаксиса диапазона (**range**), поскольку у нас есть неявный диапазон, скомпилированный код будет проверять, что переменные содержат значения, не выходящие за пределы диапазона. Кроме того, если результат умножения или деления десятичных типов с фиксированной запятой меньше, чем значение дельты, известное из контекста, фактический результат будет равен нулю. Например:

Listing 2: decimal_fixed_point_smaller.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Decimal_Fixed_Point_Smaller is
4     type T3_D3 is delta 10.0 ** (-3) digits 3;
5     type T6_D6 is delta 10.0 ** (-6) digits 6;
6     A : T3_D3 := T3_D3'Delta;
7     B : T3_D3 := 0.5;
8     C : T6_D6;
9  begin
10    Put_Line ("The value of A     is "
11              & T3_D3'Image (A));
12
13    A := A * B;
14    Put_Line ("The value of A * B is "
15              & T3_D3'Image (A));
16
17    A := T3_D3'Delta;
18    C := A * B;
19    Put_Line ("The value of A * B is "
20              & T6_D6'Image (C));
21 end Decimal_Fixed_Point_Smaller;
```

Build output

```
decimal_fixed_point_smaller.adb:7:04: warning: "B" is not modified, could be␣
 ↪declared constant [-gnatwk]
```

Runtime output

```
The value of A     is  0.001
The value of A * B is  0.000
The value of A * B is  0.000500
```

В этом примере, результат операции 0.001 * 0.5 будет 0.0005. Ввиду того, что это значение не может быть представленно типом T3_D3 ведь его дельта равна 0.001, реальное значение которое получит переменная A будет равно нулю. Однако, если тип имеет большую точность, то точности арифметических операций будет достаточно, и значение C будет равно 0.000500.

11.2 Обычные типы с фиксированной запятой

Обычные типы с фиксированной запятой похожи на десятичные типы с фиксированной запятой в том, что значения, по сути, являются масштабированными целыми числами. Разница между ними заключается в коэффициенте масштабирования: для десятичного типа с фиксированной запятой масштабирование, явно заданное дельтой (**delta**), всегда является степенью десяти.

Напротив, для обычного типа с фиксированной запятой масштабирование определяется значением small для типа, которое получается из указанного значения **delta** и, по умолчанию, является степенью двойки. Поэтому обычные типы с фиксированной запятой иногда называют двоичными типами с фиксированной запятой.

Note: Обычные типы с фиксированной запятой можно рассматривать как более близкие к реальному представлению на машине, поскольку аппаратная поддержка десятичной арифметики с фиксированной запятой не получила широкого распространения (изменение масштаба в десять раз), в то время как обычные типы с фиксированной запятой доступных используют широко распространенные инструкций целочисленного сдвига.

Синтаксис обычного типа с фиксированной запятой:

```
type <type-name> is
  delta <delta-value>
  range <lower-bound> .. <upper-bound>;
```

По умолчанию компилятор выберет коэффициент масштабирования, или small, то есть степень 2, не превышающую <delta-value>.

Например, можно определить нормализованный диапазон между -1.0 и 1.0 следующим образом:

Listing 3: normalized_fixed_point_type.adb

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

procedure Normalized_Fixed_Point_Type is
   D : constant := 2.0 ** (-31);
   type TQ31 is delta D range -1.0 .. 1.0 - D;
begin
   Put_Line ("TQ31 requires "
             & Integer'Image (TQ31'Size)
             & " bits");
   Put_Line ("The delta    value of TQ31 is "
             & TQ31'Image (TQ31'Delta));
   Put_Line ("The minimum  value of TQ31 is "
             & TQ31'Image (TQ31'First));
   Put_Line ("The maximum  value of TQ31 is "
             & TQ31'Image (TQ31'Last));
end Normalized_Fixed_Point_Type;
```

Runtime output

```
TQ31 requires  32 bits
The delta    value of TQ31 is  0.0000000005
The minimum  value of TQ31 is -1.0000000000
The maximum  value of TQ31 is  0.9999999995
```

В этом примере мы определяем 32-разрядный тип данных с фиксированной запятой для нормализованного диапазона. При запуске приложения мы замечаем, что верхняя граница близка к единице, но не ровна. Это типичный эффект типов данных с фиксированной запятой - более подробную информацию можно найти в этом обсуждении Q формата[16]. Мы также можем переписать этот код определения типа:

Listing 4: normalized_adapted_fixed_point_type.adb

```
procedure Normalized_Adapted_Fixed_Point_Type is
   type TQ31 is
     delta 2.0 ** (-31)
     range -1.0 .. 1.0 - 2.0 ** (-31);
begin
   null;
end Normalized_Adapted_Fixed_Point_Type;
```

Build output

```
normalized_adapted_fixed_point_type.adb:2:09: warning: type "TQ31" is not␣
↪referenced [-gnatwu]
```

Мы также можем использовать любой другой диапазон. Например:

Listing 5: custom_fixed_point_range.adb

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

procedure Custom_Fixed_Point_Range is
   type T_Inv_Trig is
     delta 2.0 ** (-15) * Pi
     range -Pi / 2.0 .. Pi / 2.0;
begin
   Put_Line ("T_Inv_Trig requires "
             & Integer'Image (T_Inv_Trig'Size)
             & " bits");
   Put_Line ("The delta    value of T_Inv_Trig is "
             & T_Inv_Trig'Image (T_Inv_Trig'Delta));
   Put_Line ("The minimum  value of T_Inv_Trig is "
             & T_Inv_Trig'Image (T_Inv_Trig'First));
   Put_Line ("The maximum  value of T_Inv_Trig is "
             & T_Inv_Trig'Image (T_Inv_Trig'Last));
end Custom_Fixed_Point_Range;
```

Build output

```
custom_fixed_point_range.adb:13:44: warning: static fixed-point value is not a␣
↪multiple of Small [-gnatwb]
```

Runtime output

```
T_Inv_Trig requires  16 bits
The delta    value of T_Inv_Trig is  0.00006
The minimum  value of T_Inv_Trig is -1.57080
The maximum  value of T_Inv_Trig is  1.57080
```

[16] https://en.wikipedia.org/wiki/Q_(number_format)

В этом примере мы определяем 16-разрядный тип с именем `T_Inv_Trig`, который имеет диапазон от −□/2 до □/2.

Для типов с фиксированной запятой доступны все общепринятые операции. Например:

Listing 6: fixed_point_op.adb

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

procedure Fixed_Point_Op is
   type TQ31 is
     delta 2.0 ** (-31)
     range -1.0 .. 1.0 - 2.0 ** (-31);

   A, B, R : TQ31;
begin
   A := 0.25;
   B := 0.50;
   R := A + B;
   Put_Line ("R is " & TQ31'Image (R));
end Fixed_Point_Op;
```

Runtime output

```
R is  0.7500000000
```

Как и ожидалось, R содержит 0,75 после сложения A и B.

На самом деле язык является более гибким, чем показано в этих примерах, поскольку на практике обычно необходимо умножать или делить значения различных типов с фиксированной запятой и получать результат, который может быть третьего типа. Подробная информация выходит за рамки данного вводного курса.

Следует также отметить, что, хотя подробности также выходят за рамки данного курса, можно явно указать значение `small` для обычного типа с фиксированной точкой. Это позволяет осуществлять недвоичное масштабирование, например:

```
type Angle is
  delta 1.0/3600.0
  range 0.0 .. 360.0 - 1.0 / 3600.0;
for Angle'Small use Angle'Delta;
```

CHAPTER

TWELVE

ИЗОЛЯЦИЯ

Одним из основных положений модульного программирования, а также объектно-ориентированного программирования, является инкапсуляция[17].

Инкапсуляция, вкратце, является концепцией, следуя которой разработчик программного обеспечения разделяет общедоступный интерфейс подсистемы и ее внутреннюю реализацию.

Это касается не только библиотек программного обеспечения, но и всего, где используются абстракции.

Ада несколько отличается от большинства объектно-ориентированных языков, тем что границы инкапсуляции в основном проходят по границам пакетов.

12.1 Простейшая инкапсуляция

Listing 1: encapsulate.ads

```
package Encapsulate is
   procedure Hello;

private

   procedure Hello2;
   --  Not visible from external units
end Encapsulate;
```

Listing 2: encapsulate.adb

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

package body Encapsulate is

   procedure Hello is
   begin
      Put_Line ("Hello");
   end Hello;

   procedure Hello2 is
   begin
      Put_Line ("Hello #2");
   end Hello2;

end Encapsulate;
```

[17] https://ru.wikipedia.org/wiki/Инкапсуляция_(программирование)

Listing 3: main.adb

```
with Encapsulate;

procedure Main is
begin
   Encapsulate.Hello;
   Encapsulate.Hello2;
   --  Invalid: Hello2 is not visible
end Main;
```

Build output

```
main.adb:6:15: error: "Hello2" is not a visible entity of "Encapsulate"
gprbuild: *** compilation phase failed
```

12.2 Абстрактные типы данных

С таким высокоуровневым механизмом инкапсуляции может быть неочевидно, как скрыть детали реализации одного типа. Вот как это можно сделать в Аде:

Listing 4: stacks.ads

```
package Stacks is
   type Stack is private;
   --  Declare a private type: You cannot depend
   --  on its implementation. You can only assign
   --  and test for equality.

   procedure Push (S   : in out Stack;
                   Val :        Integer);
   procedure Pop (S   : in out Stack;
                  Val :    out Integer);
private

   subtype Stack_Index is Natural range 1 .. 10;
   type Content_Type is array (Stack_Index)
     of Natural;

   type Stack is record
      Top     : Stack_Index;
      Content : Content_Type;
   end record;
end Stacks;
```

Listing 5: stacks.adb

```
package body Stacks is

   procedure Push (S   : in out Stack;
                   Val :        Integer) is
   begin
      --  Missing implementation!
      null;
   end Push;

   procedure Pop (S   : in out Stack;
                  Val :    out Integer) is
```

(continues on next page)

(continued from previous page)

```
   begin
      --  Dummy implementation!
      Val := 0;
   end Pop;

end Stacks;
```

В приведенном выше примере мы определяем тип для стека в публичной части (известной как *видимый раздел* спецификации пакета в Аде), но детали реализация этого типа скрыты.

Затем в личном разделе мы определяем реализацию этого типа. Мы также можем объявить там другие *вспомогательные* типы, которые будут использованы для описания основного публичного типа. Создание вспомогательных типов - это полезная и распространенная практика в Аде.

Несколько слов о терминологии:

- То, как выглядит тип стека `Stack` в видимом разделе, называется частичным представлением типа. Это то, к чему имеют доступ клиенты.
- То, как выглядит тип стека `Stack` из личного раздела или тела пакета, называется полным представлением типа. Это то, к чему имеют доступ разработчики.

С точки зрения клиента (указывающего пакет в **with**) важен только видимый раздел, и личного вообще может не существовать. Это позволяет очень легко просмотреть ту часть пакета, которая важна для нас.

```
--  No need to read the private part to use the package
package Stacks is
   type Stack is private;

   procedure Push (S   : in out Stack;
                   Val :        Integer);
   procedure Pop (S   : in out Stack;
                  Val :    out Integer);
private
   ...
end Stacks;
```

А вот как будет использоваться пакет `Stacks`:

```
--  Example of use
with Stacks; use Stacks;

procedure Test_Stack is
   S : Stack;
   Res : Integer;
begin
   Push (S, 5);
   Push (S, 7);
   Pop (S, Res);
end Test_Stack;
```

12.3 Лимитируемые типы

В Аде конструкция *лимитируемого типа* позволяет вам объявить тип, для которого операции присваивания и сравнения не предоставляются автоматически.

Listing 6: stacks.ads

```
package Stacks is
   type Stack is limited private;
   --  Limited type. Cannot assign nor compare.

   procedure Push (S   : in out Stack;
                   Val :        Integer);
   procedure Pop (S   : in out Stack;
                  Val :    out Integer);
private
   subtype Stack_Index is Natural range 1 .. 10;
   type Content_Type is
     array (Stack_Index) of Natural;

   type Stack is limited record
      Top     : Stack_Index;
      Content : Content_Type;
   end record;
end Stacks;
```

Listing 7: stacks.adb

```
package body Stacks is

   procedure Push (S   : in out Stack;
                   Val :        Integer) is
   begin
      --  Missing implementation!
      null;
   end Push;

   procedure Pop (S   : in out Stack;
                  Val :    out Integer) is
   begin
      --  Dummy implementation!
      Val := 0;
   end Pop;

end Stacks;
```

Listing 8: main.adb

```
with Stacks; use Stacks;

procedure Main is
   S, S2 : Stack;
begin
   S := S2;
   --  Illegal: S is limited.
end Main;
```

Build output

```
stacks.adb:10:19: warning: formal parameter "S" is not referenced [-gnatwf]
```

(continues on next page)

(continued from previous page)

```
main.adb:6:04: error: left hand of assignment must not be limited type
gprbuild: *** compilation phase failed
```

Это нужно, например, для тех типов данных, для которых встроенная операция присваивания работает неправильно (например, когда требуется многоуровневевое копирование).

Ада позволяет вам определить операторы сравнения = и /= для лимитируемых типов (или переопределить встроенные объявления для нелимитируемых).

Ада также позволяет вам предоставить собственную реализацию присваивания используя контролируемые типы[18]. Однако в некоторых случаях операция присваивания просто не имеет смысла; примером может служить `File_Type` из пакета `Ada.Text_IO`, который объявлен как лимитируемый тип, и поэтому все попытки присвоить один файл другому будут отклонены как незаконные.

12.4 Дочерние пакеты и изоляция

Ранее мы видели (в *разделе дочерние пакеты* (page 31)), что пакеты могут иметь дочерние пакеты. Изоляция играет важную роль в дочерних пакетах. В этом разделе обсуждаются некоторые правила касающиеся изоляции, действующие для дочерних пакетов.

Хотя личный раздел `P` предназначен для инкапсуляции информации, некоторые части дочернего пакета `P.C` могут иметь доступ к этому личному разделу `P`. В таких случаях информация из личного раздела `P` может затем использоваться так, как если бы она была объявлена в видимом разделе спецификации пакета. Говоря более конкретно, тело `P.C` и личный раздел пакета `P.C` имеют доступ к личному разделу `P`. Однако видимый раздел спецификации `P.C` имеет доступ только к видимому разделу спецификации `P`. В следующей таблице приводится сводная информация об этом:

Часть дочернего пакета	Доступ к личному разделу родительской спецификации
Спецификация: видимый раздел	Нет
Спецификация: личный раздел	Да
Тело	Да

В оставшейся части этого раздела показаны примеры того, как этот доступ к личной информации на самом деле работает для дочерних пакетов.

Давайте сначала рассмотрим пример, в котором тело дочернего пакета `P.C` имеет доступ к личному разделу спецификации его родителя `P`. В предыдущем примере исходного кода мы видели, что процедуру `Hello2`, объявленную в личном разделе пакета `Encapsulate` нельзя использовать в процедуре `Main`, поскольку ее там не видно. Однако это ограничение не распространяется на некоторые части дочерних пакетов. Фактически, тело дочернего пакета `Encapsulate.Child` имеет доступ к процедуре `Hello2` и она может быть вызванна оттуда, как вы можете видеть в реализации процедуры `Hello3` пакета `Child`:

Listing 9: encapsulate.ads

```
1 package Encapsulate is
2    procedure Hello;
3
4 private
```

(continues on next page)

[18] https://www.adaic.org/resources/add_content/standards/12rm/html/RM-7-6.html

(continued from previous page)

```
5
6     procedure Hello2;
7     --  Not visible from external units
8     --  But visible in child packages
9  end Encapsulate;
```

Listing 10: encapsulate.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Encapsulate is
4
5     procedure Hello is
6     begin
7        Put_Line ("Hello");
8     end Hello;
9
10    procedure Hello2 is
11    begin
12       Put_Line ("Hello #2");
13    end Hello2;
14
15 end Encapsulate;
```

Listing 11: encapsulate-child.ads

```
1  package Encapsulate.Child is
2
3     procedure Hello3;
4
5  end Encapsulate.Child;
```

Listing 12: encapsulate-child.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Encapsulate.Child is
4
5     procedure Hello3 is
6     begin
7        --  Using private procedure Hello2
8        --  from the parent package
9        Hello2;
10       Put_Line ("Hello #3");
11    end Hello3;
12
13 end Encapsulate.Child;
```

Listing 13: main.adb

```
with Encapsulate.Child;

procedure Main is
begin
   Encapsulate.Child.Hello3;
end Main;
```

Runtime output

```
Hello #2
Hello #3
```

Тот же механизм применяется к типам, объявленным в личном разделе родительского пакета. Например, тело дочернего пакета может получить доступ к компонентам записи, объявленной в личном разделе его родительского пакета. Рассмотрим пример:

Listing 14: my_types.ads

```
package My_Types is

   type Priv_Rec is private;

private

   type Priv_Rec is record
     Number : Integer := 42;
   end record;

end My_Types;
```

Listing 15: my_types-ops.ads

```
package My_Types.Ops is

   procedure Display (E : Priv_Rec);

end My_Types.Ops;
```

Listing 16: my_types-ops.adb

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

package body My_Types.Ops is

   procedure Display (E : Priv_Rec) is
   begin
      Put_Line ("Priv_Rec.Number: "
                & Integer'Image (E.Number));
   end Display;

end My_Types.Ops;
```

Listing 17: main.adb

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

with My_Types;     use My_Types;
with My_Types.Ops; use My_Types.Ops;
```

(continues on next page)

(continued from previous page)

```
5
6  procedure Main is
7     E : Priv_Rec;
8  begin
9     Put_Line ("Presenting information:");
10
11    --  The following code would trigger a
12    --  compilation error here:
13    --
14    --  Put_Line ("Priv_Rec.Number: "
15    --            & Integer'Image (E.Number));
16
17    Display (E);
18 end Main;
```

Runtime output

```
Presenting information:
Priv_Rec.Number:  42
```

В этом примере у нас нет доступа к компоненте `Number` типа записи `Priv_Rec` в процедуре `Main`. Вы можете увидеть это в вызове `Put_Line`, который был закомментирован в реализации `Main`. Попытка получить доступ к компоненте `Number` вызовет ошибку компиляции. Но у нас есть доступ к этой компоненте в теле пакета `My_Types.Ops`, поскольку это дочерний пакет пакета `My_Types`. Следовательно, тело `Ops` имеет доступ к объявлению типа `Priv_Rec`, которое находится в личном разделе его родительского пакета `My_Types`. По этой причине тот же вызов `Put_Line`, который вызовет ошибку компиляции в процедуре `Main`, отлично работает в процедуре `Display` пакета `My_Types.Ops`.

Такой рода правила изоляции для дочерних пакетов позволяют расширять функциональность родительского пакета и в то же время обеспечивают инкапсуляцию.

Как мы упоминали ранее, в дополнение к телу пакета личный раздел спецификации дочернего пакета `P.C` также имеет доступ к личному разделу спецификации его родителя `P`. Давайте посмотрим на пример, в котором мы объявляем объект личного типа `Priv_Rec` в личном разделе дочернего пакета `My_Types.Child` *и* напрямую инициализируем компоненту `Number` записи `Priv_Rec`:

```
package My_Types.Child is

private

   E : Priv_Rec := (Number => 99);

end My_Types.Ops;
```

Естественно, мы не смогли бы инициализировать этот компонент, если бы переместили это объявление в общедоступный (видимый) раздел того же дочернего пакета:

```
package My_Types.Child is

   E : Priv_Rec := (Number => 99);

end My_Types.Ops;
```

Объявление выше вызывает ошибку компиляции, поскольку тип `Priv_Rec` является личным. Поскольку видимый раздел `My_Types.Child` также виден за пределами дочернего пакета, Ада запрещает доступ к личной информации в этом разделе спецификации.

CHAPTER

THIRTEEN

НАСТРАИВАЕМЫЕ МОДУЛИ

13.1 Введение

Настраиваемые модули в Аде используются для метапрограммирования. Когда некоторые алгоритмы имеют достаточно много общего и отличаются лишь деталями, можно выделить абстрактный алгоритм воспользовавшись возможностями настраиваемых модулей.

Настраиваемыми могут быть лишь подпрограммы или пакеты. Объявление настраиваемого модуля начинается с ключевого слова **generic**. Например:

Listing 1: operator.ads

```
generic
   type T is private;
   --  Declaration of formal types and objects
--  Below, we could use one of the following:
--  <procedure | function | package>
procedure Operator (Dummy : in out T);
```

Listing 2: operator.adb

```
procedure Operator (Dummy : in out T) is
begin
   null;
end Operator;
```

13.2 Объявление формального типа

Формальные типы - это абстракции типа некоторого класса. Например, мы можем понадобиться создать алгоритм, который работает с любым целочисленным типом или даже с любым типом вообще, будь то числовой тип или нет. В следующем примере объявляется формальный тип T для процедуры Set.

Listing 3: set.ads

```
generic
   type T is private;
   --  T is a formal type that indicates that
   --  any type can be used, possibly a numeric
   --  type or possibly even a record type.
procedure Set (Dummy : T);
```

Listing 4: set.adb

```
procedure Set (Dummy : T) is
begin
   null;
end Set;
```

Объявление T как **private** указывает на то, что на его месте может быть любой определенный тип. Но также можно сузить условие, разрешив подстановку типов лишь некоторого класса. Вот несколько примеров:

Формальный тип	Формат
Любой тип	`type T is private;`
Любой дискретный тип	`type T is (<>);`
Любой тип с плавающей запятой	`type T is digits <>;`

13.3 Объявление формального объекта

Формальные объекты аналогичны параметрам подпрограммы. Они могут ссылаться на формальные типы, объявленные в формальной спецификации. Например:

Listing 5: set.ads

```
generic
   type T is private;
   X : in out T;
   --  X can be used in the Set procedure
procedure Set (E : T);
```

Listing 6: set.adb

```
procedure Set (E : T) is
   pragma Unreferenced (E, X);
begin
   null;
end Set;
```

Формальные объекты могут быть либо входными параметрами, либо иметь вид **in out**.

13.4 Определение тела настраиваемого модуля

Не нужно повторять ключевое слово **generic** при объявления тела настраиваемой подпрограммы или пакета. Для реализации мы используем синтаксис, как у обычного тела модуля, и используем объявленные выше формальные типы и объекты. Например:

Listing 7: set.ads

```
generic
   type T is private;
   X : in out T;
procedure Set (E : T);
```

Listing 8: set.adb

```
procedure Set (E : T) is
--  Body definition: "generic" keyword
--  is not used
begin
   X := E;
end Set;
```

13.5 Конкретизация настройки

Настраиваемую подпрограммы или пакеты нельзя использовать напрямую. Сначала они должны быть конкретезированы, что мы делаем с помощью ключевого слова **new**, как показано в следующем примере:

Listing 9: set.ads

```
generic
   type T is private;
   X : in out T;
   --  X can be used in the Set procedure
procedure Set (E : T);
```

Listing 10: set.adb

```
procedure Set (E : T) is
begin
   X := E;
end Set;
```

Listing 11: show_generic_instantiation.adb

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

procedure Show_Generic_Instantiation is

   Main    : Integer := 0;
   Current : Integer;

   procedure Set_Main is new Set (T => Integer,
                                  X => Main);
   --  Here, we map the formal parameters to
   --  actual types and objects.
   --
   --  The same approach can be used to
   --  instantiate functions or packages, e.g.:
   --
   --  function Get_Main is new ...
   --  package Integer_Queue is new ...

begin
   Current := 10;

   Set_Main (Current);
   Put_Line ("Value of Main is "
             & Integer'Image (Main));
end Show_Generic_Instantiation;
```

Runtime output

```
Value of Main is  10
```

В приведенном выше примере мы создаем экземпляр настраиваемой процедуры `Set`, сопоставляя формальные параметры `T` и `X` с фактическими, уже существующими, элементами, в данном случае типом **`Integer`** и переменной `Main`.

13.6 Настраиваемые пакеты

Предыдущие примеры мы сосредочились на настраиваемых подпрограммах. В этом разделе мы рассмотрим настраиваемые пакеты. Их синтаксис аналогичен: мы начинаем с ключевого слова **`generic`**, а далее следуют формальные объявления. Единственное отличие состоит в том, что вместо ключевого слова подпрограммы указывается **`package`**.

Вот пример:

Listing 12: element.ads

```
generic
   type T is private;
package Element is

   procedure Set (E : T);
   procedure Reset;
   function Get return T;
   function Is_Valid return Boolean;

   Invalid_Element : exception;

private
   Value : T;
   Valid : Boolean := False;
end Element;
```

Listing 13: element.adb

```
package body Element is

   procedure Set (E : T) is
   begin
      Value := E;
      Valid := True;
   end Set;

   procedure Reset is
   begin
      Valid := False;
   end Reset;

   function Get return T is
   begin
      if not Valid then
         raise Invalid_Element;
      end if;
      return Value;
   end Get;

   function Is_Valid return Boolean is (Valid);
end Element;
```

Listing 14: show_generic_package.adb

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

procedure Show_Generic_Package is

   package I is new Element (T => Integer);

   procedure Display_Initialized is
   begin
      if I.Is_Valid then
         Put_Line ("Value is initialized");
      else
         Put_Line ("Value is not initialized");
      end if;
   end Display_Initialized;

begin
   Display_Initialized;

   Put_Line ("Initializing...");
   I.Set (5);
   Display_Initialized;
   Put_Line ("Value is now set to "
             & Integer'Image (I.Get));

   Put_Line ("Reseting...");
   I.Reset;
   Display_Initialized;

end Show_Generic_Package;
```

Runtime output

```
Value is not initialized
Initializing...
Value is initialized
Value is now set to  5
Reseting...
Value is not initialized
```

В приведенном выше примере мы создали простой контейнер с именем `Element`, содержащий всего один элемент. Этот контейнер отслеживает, был ли элемент инициализирован или нет.

После написания определения пакета мы создаем экземпляр `I` пакета `Element`. Мы используем экземпляр, вызывая подпрограммы пакета (`Set`, `Reset` и `Get`).

13.7 Формальные подпрограммы

В дополнение к формальным типам и объектам мы также можем объявлять формальные подпрограммы или пакеты. Этот курс описывает только формальные подпрограммы; формальные пакеты обсуждаются в продвинутом курсе.

Мы используем ключевое слово **`with`** для объявления формальной подпрограммы. В приведенном ниже примере мы объявляем формальную функцию (`Comparison`), которая будет использоваться настраиваемой процедурой `Check`.

Listing 15: check.ads

```
generic
   Description : String;
   type T is private;
   with function Comparison (X, Y : T) return Boolean;
procedure Check (X, Y : T);
```

Listing 16: check.adb

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

procedure Check (X, Y : T) is
   Result : Boolean;
begin
   Result := Comparison (X, Y);
   if Result then
      Put_Line ("Comparison ("
                & Description
                & ") between arguments is OK!");
   else
      Put_Line ("Comparison ("
                & Description
                & ") between arguments is not OK!");
   end if;
end Check;
```

Listing 17: show_formal_subprogram.adb

```
with Check;

procedure Show_Formal_Subprogram is

   A, B : Integer;

   procedure Check_Is_Equal is new
     Check (Description => "equality",
            T           => Integer,
            Comparison  => Standard."=");
   --  Here, we are mapping the standard
   --  equality operator for Integer types to
   --  the Comparison formal function
begin
   A := 0;
   B := 1;
   Check_Is_Equal (A, B);
end Show_Formal_Subprogram;
```

Runtime output

```
Comparison (equality) between arguments is not OK!
```

13.8 Пример: конкретизация ввода/вывода

Ада предлагает настраиваемые пакеты ввода-вывода, которые могут быть конкретизированы для стандартных и произвольных типов. Одним из примеров является настраиваемый пакет `Float_IO`, который предоставляет такие процедуры, как `Put` и `Get`. Фактически, `Float_Text_IO` - доступный в стандартной библиотеке - является конкретизацией пакета `Float_IO` и определяется как:

```
with Ada.Text_IO;

package Ada.Float_Text_IO is new Ada.Text_IO.Float_IO (Float);
```

Его можно использовать непосредственно с любым объектом типа **`Float`**. Например:

Listing 18: show_float_text_io.adb

```
with Ada.Float_Text_IO;

procedure Show_Float_Text_IO is
   X : constant Float := 2.5;

   use Ada.Float_Text_IO;
begin
   Put (X);
end Show_Float_Text_IO;
```

Runtime output

```
 2.50000E+00
```

Создание экземпляров настраиваемых пакетов ввода-вывода может быть полезно для пользовательских типов. Например, давайте создадим новый тип `Price`, который должен отображаться с двумя десятичными цифрами после точки и без экспоненты.

Listing 19: show_float_io_inst.adb

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

procedure Show_Float_IO_Inst is

   type Price is digits 3;

   package Price_IO is new
     Ada.Text_IO.Float_IO (Price);

   P : Price;
begin
   --  Set to zero => don't display exponent
   Price_IO.Default_Exp  := 0;

   P := 2.5;
   Price_IO.Put (P);
   New_Line;

   P := 5.75;
   Price_IO.Put (P);
   New_Line;
end Show_Float_IO_Inst;
```

Runtime output

```
 2.50
 5.75
```

Регулируя значение `Default_Exp` экземпляра `Price_IO` для *удаления* экспоненты, мы можем контролировать, как отображаются переменные типа `Price`. В качестве примечания мы также могли бы написать:

```
-- [...]

   type Price is new Float;

   package Price_IO is new
     Ada.Text_IO.Float_IO (Price);

begin
   Price_IO.Default_Aft  := 2;
   Price_IO.Default_Exp  := 0;
```

В этом случае мы также изменяем `Default_Aft` чтобы при вызове `Put` получить две десятичные цифры после запятой.

В дополнение к настраиваемому пакету `Float_IO` в `Ada.Text_IO` доступны следующие настраиваемые пакеты:

- `Enumeration_IO` для перечислимых типов;
- `Integer_IO` для целочисленных типов;
- `Modular_IO` для модульных типов;
- `Fixed_IO` для типов с фиксированной запятой;
- `Decimal_IO` для десятичных типов.

Фактически, мы могли бы переписать пример выше, используя десятичные типы:

Listing 20: show_decimal_io_inst.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Decimal_IO_Inst is
4
5     type Price is delta 10.0 ** (-2) digits 12;
6
7     package Price_IO is new
8       Ada.Text_IO.Decimal_IO (Price);
9
10    P : Price;
11 begin
12    Price_IO.Default_Exp  := 0;
13
14    P := 2.5;
15    Price_IO.Put (P);
16    New_Line;
17
18    P := 5.75;
19    Price_IO.Put (P);
20    New_Line;
21 end Show_Decimal_IO_Inst;
```

Runtime output

```
          2.50
          5.75
```

13.9 Пример: АТД

Важным применением настраиваемых модулей является моделирование абстрактных типов данных (АТД). Фактически Ада предоставляет библиотеку с многочисленными АТД, использующими настраиваемые модули: `Ada.Containers` (описаны в разделе контейнеров).

Типичным примером АТД является стек:

Listing 21: stacks.ads

```
generic
   Max : Positive;
   type T is private;
package Stacks is

   type Stack is limited private;

   Stack_Underflow, Stack_Overflow : exception;

   function Is_Empty (S : Stack) return Boolean;

   function Pop (S : in out Stack) return T;

   procedure Push (S : in out Stack;
                   V :        T);

private

   type Stack_Array is
     array (Natural range <>) of T;

   Min : constant := 1;

   type Stack is record
      Container : Stack_Array (Min .. Max);
      Top       : Natural := Min - 1;
   end record;

end Stacks;
```

Listing 22: stacks.adb

```
package body Stacks is

   function Is_Empty (S : Stack) return Boolean is
     (S.Top < S.Container'First);

   function Is_Full (S : Stack) return Boolean is
     (S.Top >= S.Container'Last);

   function Pop (S : in out Stack) return T is
   begin
      if Is_Empty (S) then
         raise Stack_Underflow;
      else
         return X : T do
            X     := S.Container (S.Top);
            S.Top := S.Top - 1;
         end return;
      end if;
```

(continues on next page)

(continued from previous page)

```
   end Pop;

   procedure Push (S : in out Stack;
                   V :        T) is
   begin
      if Is_Full (S) then
         raise Stack_Overflow;
      else
         S.Top               := S.Top + 1;
         S.Container (S.Top) := V;
      end if;
   end Push;

end Stacks;
```

Listing 23: show_stack.adb

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

procedure Show_Stack is

   package Integer_Stacks is new
     Stacks (Max => 10,
             T   => Integer);
   use Integer_Stacks;

   Values : Integer_Stacks.Stack;

begin
   Push (Values, 10);
   Push (Values, 20);

   Put_Line ("Last value was "
             & Integer'Image (Pop (Values)));
end Show_Stack;
```

Runtime output

```
Last value was  20
```

В этом примере сначала создается настраиваемый пакет стека (`Stacks`), а затем он конкретизируется чтобы создаеть стек содержащи до 10 целых значений.

13.10 Пример: Обмен

Давайте рассмотрим простую процедуру, которая меняет местами переменные типа `Color`:

Listing 24: colors.ads

```
package Colors is
   type Color is (Black, Red, Green,
                  Blue, White);

   procedure Swap_Colors (X, Y : in out Color);
end Colors;
```

Listing 25: colors.adb

```
package body Colors is

   procedure Swap_Colors (X, Y : in out Color) is
      Tmp : constant Color := X;
   begin
      X := Y;
      Y := Tmp;
   end Swap_Colors;

end Colors;
```

Listing 26: test_non_generic_swap_colors.adb

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

procedure Test_Non_Generic_Swap_Colors is
   A, B, C : Color;
begin
   A := Blue;
   B := White;
   C := Red;

   Put_Line ("Value of A is "
             & Color'Image (A));
   Put_Line ("Value of B is "
             & Color'Image (B));
   Put_Line ("Value of C is "
             & Color'Image (C));

   New_Line;
   Put_Line ("Swapping A and C...");
   New_Line;
   Swap_Colors (A, C);

   Put_Line ("Value of A is "
             & Color'Image (A));
   Put_Line ("Value of B is "
             & Color'Image (B));
   Put_Line ("Value of C is "
             & Color'Image (C));
end Test_Non_Generic_Swap_Colors;
```

Runtime output

```
Value of A is BLUE
Value of B is WHITE
Value of C is RED

Swapping A and C...

Value of A is RED
Value of B is WHITE
Value of C is BLUE
```

В этом примере `Swap_Colors` можно использовать только для типа `Color`. Однако этот алгоритм теоретически можно использовать для любого типа, будь то перечислимый тип или составной тип записи с множеством элементов. Сам алгоритм такой же: отличается только тип. Если, например, мы хотим поменять местами переменные типа **`Integer`**, мы

не хотим дублировать реализацию. Следовательно, такой алгоритм - идеальный кандидат для абстракции с использованием настраиваемых модулей.

В приведенном ниже примере мы создадим настраиваемую версию `Swap_Colors` и назовем ее **`Generic_`**`Swap`. Эта настраиваемая версия может работать с любым типом благодаря объявлению формального типа `T`.

Listing 27: generic_swap.ads

```
generic
   type T is private;
procedure Generic_Swap (X, Y : in out T);
```

Listing 28: generic_swap.adb

```
procedure Generic_Swap (X, Y : in out T) is
   Tmp : constant T := X;
begin
   X := Y;
   Y := Tmp;
end Generic_Swap;
```

Listing 29: colors.ads

```
with Generic_Swap;

package Colors is

   type Color is (Black, Red, Green,
                  Blue, White);

   procedure Swap_Colors is new
     Generic_Swap (T => Color);

end Colors;
```

Listing 30: test_swap_colors.adb

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

procedure Test_Swap_Colors is
   A, B, C : Color;
begin
   A := Blue;
   B := White;
   C := Red;

   Put_Line ("Value of A is "
             & Color'Image (A));
   Put_Line ("Value of B is "
             & Color'Image (B));
   Put_Line ("Value of C is "
             & Color'Image (C));

   New_Line;
   Put_Line ("Swapping A and C...");
   New_Line;
   Swap_Colors (A, C);

   Put_Line ("Value of A is "
```

(continues on next page)

(continued from previous page)

```
            & Color'Image (A));
   Put_Line ("Value of B is "
            & Color'Image (B));
   Put_Line ("Value of C is "
            & Color'Image (C));
end Test_Swap_Colors;
```

Runtime output

```
Value of A is BLUE
Value of B is WHITE
Value of C is RED

Swapping A and C...

Value of A is RED
Value of B is WHITE
Value of C is BLUE
```

Как мы видим в примере, мы можем создать ту же процедуру `Swap_Colors`, что и в первой версии алгоритма, объявив ее как конкретизацию настраиваемой процедуры **`Generic`**_Swap. Мы сопоставляем формальный тип T с типом `Color`, указывая его в качестве аргумента конкретизации **`Generic`**_Swap.

13.11 Пример: Обратный порядок элементов

Предыдущий пример с алгоритмом обмена двух значений является одним из простейших примеров использования настраиваемых модулей. Теперь мы изучим алгоритм обращения элементов массива. Во-первых, давайте начнем с версии алгоритма без использования настраиваемых модулей, разработав версию конкретно для типа `Color`:

Listing 31: colors.ads

```
package Colors is

   type Color is (Black, Red, Green,
                  Blue, White);

   type Color_Array is
     array (Integer range <>) of Color;

   procedure Reverse_It (X : in out Color_Array);

end Colors;
```

Listing 32: colors.adb

```
package body Colors is

   procedure Reverse_It (X : in out Color_Array) is
   begin
      for I in X'First ..
               (X'Last + X'First) / 2 loop
         declare
            Tmp     : Color;
            X_Left  : Color
              renames X (I);
```

(continues on next page)

(continued from previous page)

```
            X_Right : Color
              renames X (X'Last + X'First - I);
         begin
            Tmp     := X_Left;
            X_Left  := X_Right;
            X_Right := Tmp;
         end;
      end loop;
   end Reverse_It;

end Colors;
```

Listing 33: test_non_generic_reverse_colors.adb

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

procedure Test_Non_Generic_Reverse_Colors is

   My_Colors : Color_Array (1 .. 5) :=
     (Black, Red, Green, Blue, White);

begin
   for C of My_Colors loop
      Put_Line ("My_Color: " & Color'Image (C));
   end loop;

   New_Line;
   Put_Line ("Reversing My_Color...");
   New_Line;
   Reverse_It (My_Colors);

   for C of My_Colors loop
      Put_Line ("My_Color: " & Color'Image (C));
   end loop;

end Test_Non_Generic_Reverse_Colors;
```

Runtime output

```
My_Color: BLACK
My_Color: RED
My_Color: GREEN
My_Color: BLUE
My_Color: WHITE

Reversing My_Color...

My_Color: WHITE
My_Color: BLUE
My_Color: GREEN
My_Color: RED
My_Color: BLACK
```

Процедура `Reverse_It` принимает массив цветов, начинает с обмена первого и последнего элементов массива и продолжает делать это со следующими элементами последовательно, пока не достигнет середины массива. В этот момент весь массив будет перевернут, как мы видим из выходных данных тестовой программы.

Чтобы абстрагироваться от этой процедуры, мы объявляем формальные типы для трех элементов алгоритма:

- тип компоненты массива (в примере - тип `Color`)
- диапазон, используемый для массива (в примере - целочисленный диапазон)
- фактический тип массива (в примере - тип `Color_Array`)

Это настраиваемая версия алгоритма:

Listing 34: generic_reverse.ads

```
generic
   type T is private;
   type Index is range <>;
   type Array_T is
     array (Index range <>) of T;
procedure Generic_Reverse (X : in out Array_T);
```

Listing 35: generic_reverse.adb

```
procedure Generic_Reverse (X : in out Array_T) is
begin
   for I in X'First ..
            (X'Last + X'First) / 2 loop
      declare
         Tmp     : T;
         X_Left  : T
           renames X (I);
         X_Right : T
           renames X (X'Last + X'First - I);
      begin
         Tmp     := X_Left;
         X_Left  := X_Right;
         X_Right := Tmp;
      end;
   end loop;
end Generic_Reverse;
```

Listing 36: colors.ads

```
with Generic_Reverse;

package Colors is

   type Color is (Black, Red, Green,
                  Blue, White);

   type Color_Array is
     array (Integer range <>) of Color;

   procedure Reverse_It is new
     Generic_Reverse (T       => Color,
                      Index   => Integer,
                      Array_T => Color_Array);

end Colors;
```

Listing 37: test_reverse_colors.adb

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

procedure Test_Reverse_Colors is
```

(continues on next page)

(continued from previous page)

```
   My_Colors : Color_Array (1 .. 5) :=
     (Black, Red, Green, Blue, White);

begin
   for C of My_Colors loop
      Put_Line ("My_Color: "
                & Color'Image (C));
   end loop;

   New_Line;
   Put_Line ("Reversing My_Color...");
   New_Line;
   Reverse_It (My_Colors);

   for C of My_Colors loop
      Put_Line ("My_Color: "
                & Color'Image (C));
   end loop;

end Test_Reverse_Colors;
```

Runtime output

```
My_Color: BLACK
My_Color: RED
My_Color: GREEN
My_Color: BLUE
My_Color: WHITE

Reversing My_Color...

My_Color: WHITE
My_Color: BLUE
My_Color: GREEN
My_Color: RED
My_Color: BLACK
```

Как упоминалось выше, мы выделили три параметра алгоритма:

- тип T абстрагирует элементы массива
- тип `Index` абстрагирует диапазон, используемый для массива
- тип `Array_T` абстрагирует тип массива и использует формальные объявления типов T и `Index`.

13.12 Пример: Тестовое приложение

В предыдущем примере мы сосредоточились только на абстрагировании самого алгоритма обращения массива. Однако мы могли бы аналогично абстрагировать наше небольшого тестового приложения. Это может быть полезно, если мы, например, решим протестировать другие процедуры, меняющие элементы массива.

Чтобы сделать это, мы снова должны выбрать элементы для абстрагирования. Поэтому мы объевляем следующие формальные параметры:

- S: строка, содержащая имя массива
- функция `Image`, преобразующая элемент типа T в строку

- процедура `Test`, которая выполняет некоторую операцию с массивом

Обратите внимание, что `Image` и `Test` являются примерами формальных подпрограмм, а S - примером формального объекта.

Вот версия тестового приложения, использующего общую процедуру `Perform_Test`:

Listing 38: generic_reverse.ads

```
generic
   type T is private;
   type Index is range <>;
   type Array_T is
     array (Index range <>) of T;
procedure Generic_Reverse (X : in out Array_T);
```

Listing 39: generic_reverse.adb

```
procedure Generic_Reverse (X : in out Array_T) is
begin
   for I in X'First ..
            (X'Last + X'First) / 2 loop
      declare
         Tmp     : T;
         X_Left  : T
           renames X (I);
         X_Right : T
           renames X (X'Last + X'First - I);
      begin
         Tmp     := X_Left;
         X_Left  := X_Right;
         X_Right := Tmp;
      end;
   end loop;
end Generic_Reverse;
```

Listing 40: perform_test.ads

```
generic
   type T is private;
   type Index is range <>;
   type Array_T is
     array (Index range <>) of T;
   S : String;
   with function Image (E : T) return String is <>;
   with procedure Test (X : in out Array_T);
procedure Perform_Test (X : in out Array_T);
```

Listing 41: perform_test.adb

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

procedure Perform_Test (X : in out Array_T) is
begin
   for C of X loop
      Put_Line (S & ": " & Image (C));
   end loop;

   New_Line;
   Put_Line ("Testing " & S & "...");
   New_Line;
   Test (X);
```

(continues on next page)

(continued from previous page)

```
13
14    for C of X loop
15       Put_Line (S & ": " & Image (C));
16    end loop;
17 end Perform_Test;
```

Listing 42: colors.ads

```
1  with Generic_Reverse;
2
3  package Colors is
4
5     type Color is (Black, Red, Green,
6                    Blue, White);
7
8     type Color_Array is
9       array (Integer range <>) of Color;
10
11    procedure Reverse_It is new
12      Generic_Reverse (T       => Color,
13                       Index   => Integer,
14                       Array_T => Color_Array);
15
16 end Colors;
```

Listing 43: test_reverse_colors.adb

```
1  with Colors;       use Colors;
2  with Perform_Test;
3
4  procedure Test_Reverse_Colors is
5
6     procedure Perform_Test_Reverse_It is new
7       Perform_Test (T       => Color,
8                     Index   => Integer,
9                     Array_T => Color_Array,
10                    S       => "My_Color",
11                    Image   => Color'Image,
12                    Test    => Reverse_It);
13
14    My_Colors : Color_Array (1 .. 5) :=
15      (Black, Red, Green, Blue, White);
16
17 begin
18    Perform_Test_Reverse_It (My_Colors);
19 end Test_Reverse_Colors;
```

Runtime output

```
My_Color: BLACK
My_Color: RED
My_Color: GREEN
My_Color: BLUE
My_Color: WHITE

Testing My_Color...

My_Color: WHITE
My_Color: BLUE
My_Color: GREEN
```

(continues on next page)

(continued from previous page)

```
My_Color: RED
My_Color: BLACK
```

В этом примере создается процедура, `Perform_Test_Reverse_It` как экземпляр настраиваемой процедуры (`Perform_Test`). Обратите внимание, что:

- Для формальной функции `Image` мы используем атрибут `'Image` типа `Color`
- Для формальной процедуры тестирования `Test` мы ссылаемся на процедуру `Reverse_Array` из пакета.

CHAPTER

FOURTEEN

ИСКЛЮЧЕНИЯ

Ада использует исключения для обработки ошибок. В отличие от многих других языков, в Аде принято говорить о *возбуждении*, а не о *выбрасывании* исключений и их *обработке*, а не *перехвате*.

14.1 Объявление исключения

Исключения в Аде - это не типы, а объекты, что может показаться вам необычным, если вы привыкли к тому, как работают исключения в Java или Python. Вот как вы объявляете исключение:

Listing 1: exceptions.ads

```
package Exceptions is
    My_Except : exception;
    --  Like an object. *NOT* a type !
end Exceptions;
```

Несмотря на то, что они являются объектами, каждый объявленный объект исключения вы используете как «класс» или «семейство» исключений. Ада не требует, чтобы подпрограмма при объявлении указывала каждое исключение, которое может быть возбуждено.

14.2 Возбуждение исключения

Чтобы возбудить исключение нашего только что объявленного класса исключения, сделайте следующее:

Listing 2: main.adb

```
with Exceptions; use Exceptions;

procedure Main is
begin
   raise My_Except;
   --  Execution of current control flow
   --  abandoned; an exception of kind
   --  "My_Except" will bubble up until it
   --  is caught.

   raise My_Except with "My exception message";
   --  Execution of current control flow
   --  abandoned; an exception of kind
   --  "My_Except" with associated string will
```

(continues on next page)

(continued from previous page)

```
15    --  bubble up until it is caught.
16 end Main;
```

Build output

```
main.adb:11:04: warning: unreachable code [enabled by default]
```

Runtime output

```

raised EXCEPTIONS.MY_EXCEPT : main.adb:5
```

14.3 Обработка исключения

Далее мы рассмотрим, как обрабатывать исключения, которые были возбуждены нами или библиотеками, которые мы вызываем. Изящная вещь в Аде заключается в том, что вы можете добавить обработчик исключений в любой блок операторов следующим образом:

Listing 3: open_file.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2  with Ada.Exceptions;  use Ada.Exceptions;
3
4  procedure Open_File is
5     File : File_Type;
6  begin
7     --  Block (sequence of statements)
8     begin
9        Open (File, In_File, "input.txt");
10    exception
11       when E : Name_Error =>
12       --       ^ Exception to be handled
13          Put ("Cannot open input file : ");
14          Put_Line (Exception_Message (E));
15          raise;
16          --  Reraise current occurence
17    end;
18 end Open_File;
```

Runtime output

```
Cannot open input file : input.txt: No such file or directory

raised ADA.IO_EXCEPTIONS.NAME_ERROR : input.txt: No such file or directory
```

В приведенном выше примере мы используем функцию `Exception_Message` из пакета `Ada.Exceptions`. Эта функция возвращает сообщение, связанное с исключением, в виде строки.

Вам не нужно вводить новый блок только чтобы обработать исключения: вы можете добавить его в блок операторов вашей текущей подпрограммы:

Listing 4: open_file.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2  with Ada.Exceptions;  use Ada.Exceptions;
3
4  procedure Open_File is
5     File : File_Type;
```

(continues on next page)

(continued from previous page)

```
begin
   Open (File, In_File, "input.txt");
--  Exception block can be added to any block
exception
   when Name_Error =>
      Put ("Cannot open input file");
end Open_File;
```

Build output

```
open_file.adb:2:09: warning: no entities of "Ada.Exceptions" are referenced [-
 ↪gnatwu]
open_file.adb:2:23: warning: use clause for package "Exceptions" has no effect [-
 ↪gnatwu]
```

Runtime output

```
Cannot open input file
```

Обратите внимание

Обработчики исключений имеют важное ограничение, с которым вам нужно быть осторожным: исключения, созданные в разделе описаний, не перехватываются обработчиками этого блока. Так, например, в следующем коде исключение не будет поймано.

Listing 5: be_careful.adb

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

procedure Be_Careful is
   function Dangerous return Integer is
   begin
      raise Constraint_Error;
      return 42;
   end Dangerous;

begin
   declare
      A : Integer := Dangerous;
   begin
      Put_Line (Integer'Image (A));
   exception
      when Constraint_Error =>
         Put_Line ("error!");
   end;
end Be_Careful;
```

Build output

```
be_careful.adb:2:09: warning: no entities of "Ada.Exceptions" are referenced [-
 ↪gnatwu]
be_careful.adb:2:23: warning: use clause for package "Exceptions" has no effect [-
 ↪gnatwu]
be_careful.adb:13:07: warning: "A" is not modified, could be declared constant [-
 ↪gnatwk]
```

Runtime output

```
raised CONSTRAINT_ERROR : be_careful.adb:7 explicit raise
```

Это также относится к блоку исключений верхнего уровня, который является частью текущей подпрограммы.

14.4 Предопределенные исключения

Ада имеет очень небольшое количество предопределенных исключений:

- `Constraint_Error` является основным, с которым вы можете столкнуться. Возбуждение исключения Constraint_Error происходит:
  - Когда происходит выход за границы массива или, в общем, любое нарушение ограничений
  - В случае переполнения
  - В случае обращения по пустой ссылке
  - В случае деления на 0
- `Program_Error` тоже может встетиться, но, вероятно, реже. Возбуждение исключения возникает в более сложных ситуациях, таких как проблемы с порядком предвыполнения и некоторые случаи обнаружения ошибочного выполнения.
- `Storage_Error` произойдет из-за проблем с памятью, таких как:
  - Недостаточно памяти (при распределении)
  - Недостаточно стека
- `Tasking_Error` сигнализирует об ошибках, связанных с задачами, такими как любая ошибка, возникающая во время активации задачи.

Не следует повторно использовать предопределенные исключения. Если вы будете так делать, то при возбужденнии одного из них не будет ясно, связано ли оно с работой встроенной языковой операции или нет.

CHAPTER

FIFTEEN

УПРАВЛЕНИЕ ЗАДАЧАМИ

Задачи и защищенные объекты позволяют реализовать параллельное исполнение в Аде. В следующих разделах эти концепции объясняются более подробно.

15.1 Задачи

Задачу можно рассматривать как приложение, которое выполняется *одновременно* с основным приложением. В других языках программирования задачи могут назваться потоками[19], а управление задачами можно назвать многопоточностью[20].

Задачи могут синхронизироваться с основным приложением, но также могут обрабатывать информацию полностью независимо от основного приложения. Здесь мы покажем, как это достигается.

15.1.1 Простая задача

Задачи объявляются с использованием ключевого слова **task**. Реализация задачи определяется в теле задачи (**task body**). Например:

Listing 1: show_simple_task.adb

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

procedure Show_Simple_Task is
   task T;

   task body T is
   begin
      Put_Line ("In task T");
   end T;
begin
   Put_Line ("In main");
end Show_Simple_Task;
```

Runtime output

```
In main
In task T
```

Здесь мы объявляем и реализуем задачу T. Как только запускается основное приложение, задача T запускается автоматически - нет необходимости вручную запускать эту задачу. Запустив приложение выше, мы видим, что выполняются оба вызова `Put_Line`.

[19] https://eru.wikipedia.org/wiki/Поток_выполнения
[20] https://eru.wikipedia.org/wiki/Поток_выполнения#Многопоточность

Обратите внимание, что:

- Основное приложение само по себе является задачей (основной задачей).
  - В этом примере подпрограмма `Show_Simple_Task` является основной задачей приложения.
- Задача T - это подзадача.
  - Каждая подзадача имеет задачу-родителя.
  - Поэтому основная задача - это также задача-родитель задачи T.
- Количество задач не ограничено одной: мы могли бы включить задачу T2 в приведенный выше пример.
  - Эта задача также запускается автоматически и выполняется *одновременно* как с задачей T, так и с основной задачей. Например:

Listing 2: show_simple_tasks.adb

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

      procedure Show_Simple_Tasks is
         task T;
         task T2;

         task body T is
         begin
            Put_Line ("In task T");
         end T;

         task body T2 is
         begin
            Put_Line ("In task T2");
         end T2;

      begin
         Put_Line ("In main");
      end Show_Simple_Tasks;
```

Runtime output

```
In task T
In main
In task T2
```

15.1.2 Простая синхронизация

Как мы сейчас видели, как только запускается основная задача, автоматически запускаются и ее подзадачи. Основная задача продолжает свою работу до тех пор, пока ей есть что делать. Однако после этого она не сразу завершится. Вместо этого задача ожидает завершения выполнения своих подзадач, прежде чем позволит себе завершиться. Другими словами, этот процесс ожидания обеспечивает синхронизацию между основной задачей и ее подзадачами. После этой синхронизации основная задача завершится. Например:

Listing 3: show_simple_sync.adb

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

procedure Show_Simple_Sync is
   task T;
```

(continues on next page)

(continued from previous page)

```
   task body T is
   begin
      for I in 1 .. 10 loop
         Put_Line ("hello");
      end loop;
   end T;
begin
   null;
   --  Will wait here until all tasks
   --  have terminated
end Show_Simple_Sync;
```

Runtime output

```
hello
hello
hello
hello
hello
hello
hello
hello
hello
hello
```

Тот же механизм используется для других подпрограмм, содержащих подзадачи: задача-родитель подпрограммы будет ждать завершения своих подзадач. Таким образом, этот механизм не ограничивается основным приложением, а также применяется к любой подпрограмме, вызываемой основным приложением или его подпрограммами.

Синхронизация также происходит, если мы вынесем задачу в отдельный пакет. В приведенном ниже примере мы объявляем задачу T в пакете `Simple_Sync_Pkg`.

Listing 4: simple_sync_pkg.ads

```
package Simple_Sync_Pkg is
   task T;
end Simple_Sync_Pkg;
```

Это соответствующее тело пакета:

Listing 5: simple_sync_pkg.adb

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

package body Simple_Sync_Pkg is
   task body T is
   begin
      for I in 1 .. 10 loop
         Put_Line ("hello");
      end loop;
   end T;
end Simple_Sync_Pkg;
```

Поскольку пакет указан в **with** основной процедуры, задача T, определенная в пакете, является частью основной задачи. Например:

Listing 6: test_simple_sync_pkg.adb

```
with Simple_Sync_Pkg;

```

(continues on next page)

(continued from previous page)

```
procedure Test_Simple_Sync_Pkg is
begin
   null;
   --  Will wait here until all tasks
   --  have terminated
end Test_Simple_Sync_Pkg;
```

Build output

```
test_simple_sync_pkg.adb:1:06: warning: unit "Simple_Sync_Pkg" is not referenced [-
↪gnatwu]
```

Runtime output

```
hello
hello
hello
hello
hello
hello
hello
hello
hello
hello
```

Опять же, как только основная задача достигает своего конца, она синхронизируется с задачей T из `Simple_Sync_Pkg` перед завершением.

15.1.3 Оператор задержки

Мы можем ввести задержку, используя ключевое слово **delay**. Это переводит задачу в спящий режим на время, указанное (в секундах) в операторе delay. Например:

Listing 7: show_delay.adb

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

procedure Show_Delay is

   task T;

   task body T is
   begin
      for I in 1 .. 5 loop
         Put_Line ("hello from task T");
         delay 1.0;
         --    ^ Wait 1.0 seconds
      end loop;
   end T;
begin
   delay 1.5;
   Put_Line ("hello from main");
end Show_Delay;
```

Runtime output

```
hello from task T
hello from task T
hello from main
```

(continues on next page)

(continued from previous page)

```
hello from task T
hello from task T
hello from task T
```

В этом примере мы заставляем задачу T ждать одну секунду после каждого вывода сообщения "hello". Кроме того, основная задача ожидает 1,5 секунды перед выводом своего сообщения "hello".

15.1.4 Синхронизация: рандеву

Единственный тип синхронизации, который мы видели до сих пор, - это тот, что происходит автоматически в конце основной задачи. Вы также можете определить пользовательские точки синхронизации (входы задач), используя ключевое слово **entry**. *Вход* задачи можно рассматривать как особый вид подпрограммы, вызов которой выполняется другой задачей и имеет синтаксис аналогичный синтаксу вызова процедуры.

При написании тела задачи вы определяете места, где задача будет принимать входы, используя ключевое слово **accept**. Задача выполняется до тех пор, пока не достигнет оператора **accept**, а затем ожидает синхронизации с вызывающей задачей. А именно:

- вызываемая задача ожидает в этот момент (в операторе принятия **accept**) и готова принять вызов соответствующей входа из вызывающей задачи;
- вызывающая задача вызывает вход задачи способом, аналогичным вызову процедуры, чтобы синхронизации с вызываемой задачей.

Эта синхронизация между задачами называется *рандеву*. Давайте посмотрим на пример:

Listing 8: show_rendezvous.adb

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

procedure Show_Rendezvous is

   task T is
      entry Start;
   end T;

   task body T is
   begin
      accept Start;
      --     ^ Waiting for somebody
      --       to call the entry

      Put_Line ("In T");
   end T;

begin
   Put_Line ("In Main");

   --  Calling T's entry:
   T.Start;
end Show_Rendezvous;
```

Runtime output

```
In Main
In T
```

В этом примере мы объявляем вход `Start` задачи T. В теле задачи мы реализуем этот вход с помощью оператора принятия **accept** `Start`. Когда задача T достигает этой точки, она

ожидает основную задачу. Эта синхронизация происходит в операторе `T.Start`. После завершения синхронизации основная задача и задача T снова выполняются одновременно, пока они не синхронизируются в последний раз, когда основная задача завершается.

Вход может использоваться для выполнения чего-то большего, чем простая синхронизация задач: он также может выполнять несколько операторов в течение времени синхронизации обеих задач. Мы делаем это с помощью блока **do ... end**. Для предыдущего примера мы бы просто написали **accept** Start **do** <операторы>; **end**;. Мы используем эту конструкцию в следующем примере.

15.1.5 Обрабатывающий цикл

Задача может исполнять оператор принятия входа не ограниченое число раз. Мы могли бы даже создать бесконечный цикл в задаче и принимать вызовы одного и того же входа снова и снова. Однако бесконечный цикл препятствует завершению задачи-родителя и заблокирует ее, когда она достигает своего конца. Поэтому цикл, содержащий оператор принятия **accept** в теле задачи, обычно используется в сочетании с оператором **select ... or terminate** (выбрать или завершить). Говоря упрощенно, этот оператор позволяет родительской задаче автоматически завершать выполнение подзадачи по достижении своего конца. Например:

Listing 9: show_rendezvous_loop.adb

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

procedure Show_Rendezvous_Loop is

   task T is
      entry Reset;
      entry Increment;
   end T;

   task body T is
      Cnt : Integer := 0;
   begin
      loop
         select
            accept Reset do
               Cnt := 0;
            end Reset;
            Put_Line ("Reset");
         or
            accept Increment do
               Cnt := Cnt + 1;
            end Increment;
            Put_Line ("In T's loop ("
                      & Integer'Image (Cnt)
                      & ")");
         or
            terminate;
         end select;
      end loop;
   end T;

begin
   Put_Line ("In Main");

   for I in 1 .. 4 loop
      --  Calling T's entry multiple times
      T.Increment;
   end loop;
```

(continues on next page)

(continued from previous page)

```
39
40    T.Reset;
41    for I in 1 .. 4 loop
42       --  Calling T's entry multiple times
43       T.Increment;
44    end loop;
45
46 end Show_Rendezvous_Loop;
```

Runtime output

```
In Main
In T's loop ( 1)
In T's loop ( 2)
In T's loop ( 3)
In T's loop ( 4)
Reset
In T's loop ( 1)
In T's loop ( 2)
In T's loop ( 3)
In T's loop ( 4)
```

В этом примере тело задачи содержит бесконечный цикл, который принимает вызовы входов `Reset` и `Increment`. Нам стоит отметить следующее:

- Блок **`accept`** `E` **`do`** `...` **`end`** используется для увеличения счетчика.
  - До тех пор, пока задача T выполняет блок **`do`** `...` **`end`**, основная задача ожидает завершения блока.
- Основная задача выполняет вызов входа `Increment` несколько раз в цикле от `1 .. 4`. Она также вызывает вход `Reset` перед вторым циклом.
  - Поскольку задача T содержит бесконечный цикл, она всегда принимает вызовы входов `Reset` и `Increment`.
  - Когда основная задача достигает конца, она проверяет статус задачи T. Несмотря на то, что задача T может принимать новые вызовы входов `Reset` или `Increment`, главная задача может завершить задачу T введу наличия **`or terminate`** ветви оператора **`select`**.

15.1.6 Циклические задачи

В предыдущем примере мы видели, как приостановить задачу на указанное время с помощью ключевого слова **`delay`**. Однако использования операторов задержки в цикле недостаточно, чтобы гарантировать регулярные интервалы между итерациями цикла. Допустим вызовы процедуры выполняющей интенсивные вычисления чередуется выполнением операторов задержки:

```
while True loop
   delay 1.0;
   --    ^ Wait 1.0 seconds
   Computational_Intensive_App;
end loop;
```

В этом случае мы не можем гарантировать, что после 10 выполнений оператора delay прошло ровно 10 секунд, поскольку процедура `Computational_Intensive_App` может влиять на длительность итерации. Во многих случаях этот дрейф не имеет значения, поэтому достаточно использовать ключевое слово **`delay`**.

Однако бывают ситуации, когда такой дрейф недопустим. В этих случаях нам нужно использовать оператор **delay until**, который принимает точное время окончания задержки, позволяя нам сохранить востоянные интервалы. Это полезно, например, в приложениях реального времени.

Вскоре мы увидим пример того, как можно получить этот временной дрейф и как оператор **delay until** позволяет обойти проблему. Но прежде чем мы это сделаем, мы рассмотрим пакет, содержащий процедуру, позволяющую нам измерять прошедшее время (`Show_Elapsed_Time`) и фиктивную процедуру `Computational_Intensive_App`, которая эмулируется с помощью простой задержки. Вот полный текст пакета:

Listing 10: delay_aux_pkg.ads

```
with Ada.Real_Time; use Ada.Real_Time;

package Delay_Aux_Pkg is

   function Get_Start_Time return Time
     with Inline;

   procedure Show_Elapsed_Time
     with Inline;

   procedure Computational_Intensive_App;
private
   Start_Time   : Time := Clock;

   function Get_Start_Time return Time is (Start_Time);

end Delay_Aux_Pkg;
```

Listing 11: delay_aux_pkg.adb

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

package body Delay_Aux_Pkg is

   procedure Show_Elapsed_Time is
      Now_Time     : Time;
      Elapsed_Time : Time_Span;
   begin
      Now_Time     := Clock;
      Elapsed_Time := Now_Time - Start_Time;
      Put_Line ("Elapsed time "
                & Duration'Image (To_Duration (Elapsed_Time))
                & " seconds");
   end Show_Elapsed_Time;

   procedure Computational_Intensive_App is
   begin
      delay 0.5;
   end Computational_Intensive_App;

end Delay_Aux_Pkg;
```

Используя этот вспомогательный пакет, теперь мы готовы написать наше приложение с дрейфом по времени:

Listing 12: show_time_task.adb

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

(continues on next page)

(continued from previous page)

```
3
4  with Delay_Aux_Pkg;
5
6  procedure Show_Time_Task is
7     package Aux renames Delay_Aux_Pkg;
8
9     task T;
10
11    task body T is
12       Cnt   : Integer := 1;
13    begin
14       for I in 1 .. 5 loop
15          delay 1.0;
16
17          Aux.Show_Elapsed_Time;
18          Aux.Computational_Intensive_App;
19
20          Put_Line ("Cycle # "
21                    & Integer'Image (Cnt));
22          Cnt  := Cnt + 1;
23       end loop;
24       Put_Line ("Finished time-drifting loop");
25    end T;
26
27 begin
28    null;
29 end Show_Time_Task;
```

Build output

```
show_time_task.adb:2:09: warning: no entities of "Ada.Real_Time" are referenced [-
↪gnatwu]
show_time_task.adb:2:21: warning: use clause for package "Real_Time" has no effect␣
↪[-gnatwu]
```

Runtime output

```
Elapsed time  1.035948169 seconds
Cycle #  1
Elapsed time  2.545618496 seconds
Cycle #  2
Elapsed time  4.050430835 seconds
Cycle #  3
Elapsed time  5.552350350 seconds
Cycle #  4
Elapsed time  7.082408585 seconds
Cycle #  5
Finished time-drifting loop
```

Запустив приложение, мы видим, что у нас уже есть разница во времени примерно в четыре секунды после трех итераций цикла из-за дрейфа, вызванного `Computational_Intensive_Applications`. Однако, используя оператор **`delay until`**, мы можем избежать этого дрейфа и получить регулярный интервал ровно итерации в одну секунду:

Listing 13: show_time_task.adb

```
1  with Ada.Text_IO;   use Ada.Text_IO;
2  with Ada.Real_Time; use Ada.Real_Time;
3
4  with Delay_Aux_Pkg;
5
```

(continues on next page)

(continued from previous page)

```
6  procedure Show_Time_Task is
7     package Aux renames Delay_Aux_Pkg;
8
9     task T;
10
11    task body T is
12       Cycle : constant Time_Span :=
13         Milliseconds (1000);
14       Next  : Time := Aux.Get_Start_Time
15                       + Cycle;
16
17       Cnt   : Integer := 1;
18    begin
19       for I in 1 .. 5 loop
20          delay until Next;
21
22          Aux.Show_Elapsed_Time;
23          Aux.Computational_Intensive_App;
24
25          --  Calculate next execution time
26          --  using a cycle of one second
27          Next := Next + Cycle;
28
29          Put_Line ("Cycle # "
30                    & Integer'Image (Cnt));
31          Cnt  := Cnt + 1;
32       end loop;
33       Put_Line ("Finished cycling");
34    end T;
35
36 begin
37    null;
38 end Show_Time_Task;
```

Runtime output

```
Elapsed time  1.004152324 seconds
Cycle #  1
Elapsed time  2.003440248 seconds
Cycle #  2
Elapsed time  3.030813293 seconds
Cycle #  3
Elapsed time  4.001581985 seconds
Cycle #  4
Elapsed time  5.032210132 seconds
Cycle #  5
Finished cycling
```

Теперь, как мы можем видеть, запустив приложение, оператор **`delay until`** гарантирует, что `Computational_Intensive_App` не нарушает регулярный интервал в одну секунду между итерациями.

15.2 Защищенные объекты

Когда несколько задач получают доступ к общим данным, может произойти повреждение этих данных. Например, данные могут оказаться несогласованными, если одна задача перезаписывает части информации, которые в то же время считываются другой задачей. Чтобы избежать подобных проблем и обеспечить скоординированный доступ к информации, мы используем *защищенные объекты*.

Защищенные объекты инкапсулируют данные и обеспечивают доступ к этим данным с помощью *защищенных операций*, которые могут быть подпрограммами или защищенными входами. Использование защищенных объектов гарантирует, что данные не будут повреждены из-за «состояния гонки» или другого одновременного доступа.

Важное замечание.

Защищенные объекты могут быть реализованы с помощью задач Ада. Фактически это был *единственный* возможный способ их реализации в Аде 83 (первый вариант стандарта языка Ада). Однако использование защищённых объектов гораздо проще, чем использование аналогичных механизмов, реализованных с использованием лишь задач. Поэтому предпочтительно использовать защищенные объекты, когда ваша основная цель - только защита данных.

15.2.1 Простой объект

Вы объявляете защищенный объект с помощью ключевого слова **protected**. Синтаксис аналогичен тому, который используется для пакетов: вы можете объявлять операции (например, процедуры и функции) в видимом разделе, а данные - в личном разделе. Соответствующая реализация операций включена в тело защищенного объекта (**protected body**). Например:

Listing 14: show_protected_objects.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Protected_Objects is
4
5     protected Obj is
6        --  Operations go here (only subprograms)
7        procedure Set (V : Integer);
8        function Get return Integer;
9     private
10       --  Data goes here
11       Local : Integer := 0;
12    end Obj;
13
14    protected body Obj is
15       --  procedures can modify the data
16       procedure Set (V : Integer) is
17       begin
18          Local := V;
19       end Set;
20
21       --  functions cannot modify the data
22       function Get return Integer is
23       begin
24          return Local;
25       end Get;
```

(continues on next page)

(continued from previous page)

```
   end Obj;

begin
   Obj.Set (5);
   Put_Line ("Number is: "
             & Integer'Image (Obj.Get));
end Show_Protected_Objects;
```

Runtime output

```
Number is:  5
```

В этом примере мы определяем две операции для `Obj`: `Set` и `Get`. Реализация этих операций находится в теле объекта `Obj`. Синтаксис, используемый для записи этих операций, такой же, как и для обычных процедур и функций. Реализация защищенных объектов проста - мы просто читаем и переписываем значение `Local` в этих подпрограммах. Для вызова этих операций в основном приложении мы используем точечную нотацию, например, `Obj.Get`.

15.2.2 Входы

В дополнение к защищенным процедурам и функциям вы также можете определить защищенные входы. Сделайте это, используя ключевое слово **`entry`**. Защищенные входы позволяют вам определить барьеры с помощью ключевого слова **`when`**. Барьеры - это условия, которые должны быть выполнены до того, как вход сможет начать выполнять свой код - мы говорим о *снятии* барьера при выполнении условия.

В предыдущем примере использовались процедуры и функции для определения операций с защищенными объектами. Однако при этом можно считать защищенную информацию (через `Obj.Get`) до того, как она будет установлена (через `Obj.Set`). Чтобы код имел детерминированное поведение, мы указали значение по умолчанию (0). Если, вместо этого, переписать функцию `Obj.Get` как *вход*, мы сможем установить барьер, гарантирующий, что ни одна задача не сможет прочитать информацию до того, как она будет записана.

В следующем примере реализован барьер для операции `Obj.Get`. Он также содержит две параллельно исполняющиеся подпрограммы (основная задача и задача T), которые пытаются получить доступ к защищаемому объекту.

Listing 15: show_protected_objects_entries.adb

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

procedure Show_Protected_Objects_Entries is

   protected Obj is
      procedure Set (V : Integer);
      entry Get (V : out Integer);
   private
      Local  : Integer;
      Is_Set : Boolean := False;
   end Obj;

   protected body Obj is
      procedure Set (V : Integer) is
      begin
         Local := V;
         Is_Set := True;
      end Set;

      entry Get (V : out Integer)
```

(continues on next page)

(continued from previous page)

```
         when Is_Set is
          --  Entry is blocked until the
          --  condition is true. The barrier
          --  is evaluated at call of entries
          --  and at exits of procedures and
          --  entries. The calling task sleeps
          --  until the barrier is released.
      begin
         V := Local;
         Is_Set := False;
      end Get;
   end Obj;

   N : Integer := 0;

   task T;

   task body T is
   begin
      Put_Line ("Task T will delay for 4 seconds...");
      delay 4.0;
      Put_Line ("Task T will set Obj...");
      Obj.Set (5);
      Put_Line ("Task T has just set Obj...");
   end T;
begin
   Put_Line ("Main application will get Obj...");
   Obj.Get (N);
   Put_Line ("Main application has just retrieved Obj...");
   Put_Line ("Number is: " & Integer'Image (N));

end Show_Protected_Objects_Entries;
```

Runtime output

```
Task T will delay for 4 seconds...
Main application will get Obj...
Task T will set Obj...
Task T has just set Obj...
Main application has just retrieved Obj...
Number is:  5
```

Как видим, запустив пример, основное приложение ждет, пока не произойдет запись в защищенный объект (по вызову `Obj.Set` в задаче `T`), прежде чем прочитать информацию (через `Obj.Get`). Поскольку в задаче `T` добавлена 4-секундная задержка, основное приложение также задерживается на 4 секунды. Только после этой задержки задача `T` записывает данные в объект и снимает барьер в `Obj.Get`, чтобы главное приложение могло затем возобновить обработку (после извлечения информации из защищенного объекта).

15.3 Задачные и защищенные типы

В предыдущих примерах мы определили единичные задачи и защищенные объекты. Однако мы можем описывать задачи и защищенные объекты, используя определения типов. Это позволяет нам, например, создавать несколько задач на основе только одного типа задач.

15.3.1 Задачные типы

Задачный типй - это обобщение задачи. Его объявление аналогично единичным задачам: вы заменяете **task** на **task type**. Разница между единичными задачами и задачными заключается в том, что задачные типы не создают фактические задачи и не запускают их автоматически. Вместо этого требуется объявление задачи. Именно так работают обычные переменные и типы: объекты создаются только с помощью определений переменных, но не определений типов.

Чтобы проиллюстрировать это, мы повторим наш первый пример:

Listing 16: show_simple_task.adb

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

procedure Show_Simple_Task is
   task T;

   task body T is
   begin
      Put_Line ("In task T");
   end T;
begin
   Put_Line ("In main");
end Show_Simple_Task;
```

Runtime output

```
In task T
In main
```

Теперь мы переписываем его, заменив задачу **task** T задачным типом **task type TT**. Объявляем задачу (A_Task) на основе задачного типа TT после её определения:

Listing 17: show_simple_task_type.adb

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

procedure Show_Simple_Task_Type is
   task type TT;

   task body TT is
   begin
      Put_Line ("In task type TT");
   end TT;

   A_Task : TT;
begin
   Put_Line ("In main");
end Show_Simple_Task_Type;
```

Runtime output

```
In task type TT
In main
```

Мы можем расширить этот пример и создать массив задач. Поскольку мы используем тот же синтаксис, что и для объявлений переменных, мы используем аналогичный синтаксис для задачного типа: `array (<>) of Task_Type`. Кроме того, мы можем передавать информацию отдельным задачам, определив начальный вход `Start`. Вот обновленный пример:

Listing 18: show_task_type_array.adb

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

procedure Show_Task_Type_Array is
   task type TT is
      entry Start (N : Integer);
   end TT;

   task body TT is
      Task_N : Integer;
   begin
      accept Start (N : Integer) do
         Task_N := N;
      end Start;
      Put_Line ("In task T: "
                & Integer'Image (Task_N));
   end TT;

   My_Tasks : array (1 .. 5) of TT;
begin
   Put_Line ("In main");

   for I in My_Tasks'Range loop
      My_Tasks (I).Start (I);
   end loop;
end Show_Task_Type_Array;
```

Runtime output

```
In main
In task T:  1
In task T:  2
In task T:  3
In task T:  4
In task T:  5
```

В этом примере мы объявляем пять задач в массиве `My_Tasks`. Мы передаем индекс в массиве отдельной задаче вызвав вход (`Start`). После синхронизации между отдельными подзадачами и основной задачей каждая подзадача одновременно вызывает `Put_Line`.

15.3.2 Защищенные типы

Защищенный тип - это обобщение защищенного объекта. Объявление аналогично объявлению для защищенных объектов: вы заменяете **protected** на **protected type**. Как и в случае задачных типов, защищенные типы требуют объявления объекта для создания реальных объектов. Опять же, это похоже на объявления переменных и позволяет создавать массивы (или другие составные объекты) из защищенных объектов.

Мы можем повторно использовать предыдущий пример и переписать его, чтобы использовать защищенный тип:

Listing 19: show_protected_object_type.adb

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

procedure Show_Protected_Object_Type is

   protected type Obj_Type is
      procedure Set (V : Integer);
      function Get return Integer;
   private
      Local : Integer := 0;
   end Obj_Type;

   protected body Obj_Type is
      procedure Set (V : Integer) is
      begin
         Local := V;
      end Set;

      function Get return Integer is
      begin
         return Local;
      end Get;
   end Obj_Type;

   Obj : Obj_Type;
begin
   Obj.Set (5);
   Put_Line ("Number is: "
             & Integer'Image (Obj.Get));
end Show_Protected_Object_Type;
```

Runtime output

```
Number is:  5
```

В этом примере вместо непосредственного определения объекта `Obj` мы сначала определяем защищенный тип `Obj_Type` а затем объявляем `Obj` как объект этого защищенного типа. Обратите внимание, что основное приложение не изменилось: мы по-прежнему используем `Obj.Set` и `Obj.Get` для доступа к защищенному объекту, как в исходном примере.

CHAPTER

SIXTEEN

КОНТРАКТНОЕ ПРОЕКТИРОВАНИЕ

Контракты используются в программировании чтобы сформулировать ожидания. Виды параметров подпрограммы можно рассматривать как простую форму контрактов. Когда спецификация подпрограммы `Op` объявляет параметр, вида **`in`**, вызывающий `Op` знает, что аргумент **`in`** не будет изменен `Op`. Другими словами, вызывающий ожидает, что `Op` не изменяет предоставляемый им аргумент, а просто читает информацию, хранящуюся в аргументе. Ограничения и подтипы являются другими примерами контрактов. В целом, эти спецификации улучшают согласованность приложения.

Контрактное программирование основывается на таких техниках, как указание пред- и постусловий, предикатов подтипов и инвариантов типов. Мы изучаем эти темы в этой главе.

16.1 Пред- и постусловия

Пред- и постусловия формализируют ожидания относительно входных и выходных параметров подпрограмм и возвращаемого значения функций. Если мы говорим, что некие требования должны быть выполнены перед вызовом подпрограммы `Op`, это предварительные условия. Точно так же, если определенные требования должны быть выполнены после вызова подпрограммы `Op`, это постусловия. Мы можем думать о предусловиях и постусловиях как об обещаниях между тем, кто вызывает и тем, кто принимает вызов подпрограммы: предусловие - это обещание от вызывающего к вызываемому, а постусловие - это обещание в обратном направлении.

Пред- и постусловия указываются с помощью спецификации аспекта в объявлении подпрограммы. Спуцификация **`with`** `Pre =>` <условие> определяет предварительное условие, а спецификация **`with`** `Post =>` <условие> определяет постусловие.

В следующем коде показан пример предварительных условий:

Listing 1: show_simple_precondition.adb

```
procedure Show_Simple_Precondition is

   procedure DB_Entry (Name : String;
                       Age  : Natural)
     with Pre => Name'Length > 0
   is
   begin
      --  Missing implementation
      null;
   end DB_Entry;
begin
   DB_Entry ("John", 30);

   --  Precondition will fail!
   DB_Entry ("",     21);
end Show_Simple_Precondition;
```

Runtime output

```
raised ADA.ASSERTIONS.ASSERTION_ERROR : failed precondition from show_simple_
↪precondition.adb:5
```

В этом примере мы хотим, чтобы поле имени в нашей базе данных не содержало пустой строки. Мы реализуем это требование, используя предварительное условие, требующее, чтобы длина строки, используемой для параметра `Name` процедуры `DB_Entry`, была больше нуля. Если процедура `DB_Entry` вызывается с пустой строкой для параметра `Name`, вызов завершится неудачно, поскольку предварительное условие не выполнено.

В наборе инструментов GNAT

GNAT обрабатывает предварительные и постусловия, генерируя код для проверки утверждений (assertion) во время выполнения программы. Однако по умолчанию проверка утверждения не включена. Следовательно, чтобы проверять предварительные и постусловия во время выполнения, вам необходимо включить проверку утверждений с помощью ключа *-gnata*.

Прежде чем мы перейдем к следующему примеру, давайте кратко обсудим кванторные выражения, которые весьма полезны для краткого написания предварительных и постусловий. Кванторные выражения возвращают логическое значение, указывающее, соответствуют ли элементы массива или контейнера ожидаемому условию. Они имеют форму: `(for all I in A'Range => <условие для A(I)>`, где `A` - это массив, а `I` - индекс. Кванторные выражения, использующие `for all`, проверяют, выполняется ли условие для каждого элемент. Например:

```
(for all I in A'Range => A (I) = 0)
```

Это кванторные выражение истинно только тогда, когда все элементы массива `A` имеют нулевое значение.

Другой вид кванторных выражений использует `for some`. Он выглядит примерно так: `(for some I in A'Range => <условие для A(I)>`. И в этом случае кванторное выражение проверяет, есть ли *некоторые* (some) элементы (отсюда и название) для которых условие истино.

Проиллюстрируем постусловия на следующем примере:

Listing 2: show_simple_postcondition.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Simple_Postcondition is
4
5     type Int_8 is range -2 ** 7 .. 2 ** 7 - 1;
6
7     type Int_8_Array is
8       array (Integer range <>) of Int_8;
9
10    function Square (A : Int_8) return Int_8 is
11      (A * A)
12      with Post => (if abs A in 0 | 1
13                    then Square'Result = abs A
14                    else Square'Result > A);
15
16    procedure Square (A : in out Int_8_Array)
17      with Post => (for all I in A'Range =>
18                      A (I) = A'Old (I) * A'Old (I))
```

(continues on next page)

(continued from previous page)

```
   is
   begin
      for V of A loop
         V := Square (V);
      end loop;
   end Square;

   V : Int_8_Array := (-2, -1, 0, 1, 10, 11);
begin
   for E of V loop
      Put_Line ("Original: "
                & Int_8'Image (E));
   end loop;
   New_Line;

   Square (V);
   for E of V loop
      Put_Line ("Square:   "
                & Int_8'Image (E));
   end loop;
end Show_Simple_Postcondition;
```

Runtime output

```
Original: -2
Original: -1
Original:  0
Original:  1
Original:  10
Original:  11

Square:    4
Square:    1
Square:    0
Square:    1
Square:    100
Square:    121
```

Мы объявляем 8-битный тип со знаком `Int_8` и массив элементов этого типа (`Int_8_Array`). Мы хотим убедиться, что после вызова процедуры `Square` каждый элемент массива `Int_8_Array` возведен в квадрат. Мы делаем это с помощью постусловия, используя выражение **`for all`**. Это постусловие использует атрибут `'Old` чтобы сослаться на исходное (до вызова) значение параметра.

Мы также хотим убедиться, что результат вызовов функции `Square` больше, чем аргумент этого вызова. Для этого мы пишем постусловие, применяя атрибут `'Result` к именем функции и сравниваем его с входным значением.

Мы можем использовать как предварительные, так и постусловия в объявлении одной подпрограммы. Например:

Listing 3: show_simple_contract.adb

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

procedure Show_Simple_Contract is

   type Int_8 is range -2 ** 7 .. 2 ** 7 - 1;

   function Square (A : Int_8) return Int_8 is
     (A * A)
```

(continues on next page)

(continued from previous page)

```
     with
          Pre  => (Integer'Size >= Int_8'Size * 2
                   and Integer (A) * Integer (A) <=
                       Integer (Int_8'Last)),
          Post => (if abs A in 0 | 1
                   then Square'Result = abs A
                   else Square'Result > A);

   V : Int_8;
begin
   V := Square (11);
   Put_Line ("Square of 11 is " & Int_8'Image (V));

   --  Precondition will fail...
   V := Square (12);
   Put_Line ("Square of 12 is " & Int_8'Image (V));
end Show_Simple_Contract;
```

Runtime output

```
Square of 11 is  121

raised ADA.ASSERTIONS.ASSERTION_ERROR : failed precondition from show_simple_
↪contract.adb:10
```

В этом примере мы хотим убедиться, что входной аргумент вызовова функции `Square` не вызовет переполнения типа `Int_8` в нашей функции. Мы делаем это путем преобразования входного значения в тип **`Integer`**, который используется для промежуточных вычислений, и проверяем, находится ли результат в допустимом для типа `Int_8` диапазоне. В этом примере у нас то же постусловие, что и в предыдущем.

16.2 Предикаты

Предикаты определяют ожидания касающиеся типов. Они похожи на предусловия и постусловия, но применяются к типам, а не к подпрограммам. Их условия проверяются для каждого объекта данного типа, что позволяет убедиться, что объект типа T соответствует требованиям, указанным для данного типа.

Есть два вида предикатов: статические и динамические. Проще говоря, статические предикаты используются для проверки объектов во время компиляции, а динамические предикаты используются для проверок во время выполнения. Обычно статические предикаты используются для скалярных типов и динамические предикаты для более сложных типов.

Статические и динамические предикаты указываются с помощью следующих спецификаций:

- **`with`** `Static_Predicate =>` `<свойство>`
- **`with`** `Dynamic_Predicate =>` `<свойство>`

Давайте воспользуемся следующим примером, чтобы проиллюстрировать динамические предикаты:

Listing 4: show_dynamic_predicate_courses.adb

```
with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
with Ada.Calendar;          use Ada.Calendar;
with Ada.Containers.Vectors;

```

(continues on next page)

(continued from previous page)

```
procedure Show_Dynamic_Predicate_Courses is

   package Courses is
      type Course_Container is private;

      type Course is record
         Name       : Unbounded_String;
         Start_Date : Time;
         End_Date   : Time;
      end record
        with Dynamic_Predicate =>
          Course.Start_Date <= Course.End_Date;

      procedure Add (CC : in out Course_Container;
                     C  :        Course);
   private
      package Course_Vectors is new
        Ada.Containers.Vectors
          (Index_Type   => Natural,
           Element_Type => Course);

      type Course_Container is record
         V : Course_Vectors.Vector;
      end record;
   end Courses;

   package body Courses is
      procedure Add (CC : in out Course_Container;
                     C  :        Course) is
      begin
         CC.V.Append (C);
      end Add;
   end Courses;

   use Courses;

   CC : Course_Container;
begin
   Add (CC,
        Course'(
          Name       => To_Unbounded_String
                          ("Intro to Photography"),
          Start_Date => Time_Of (2018, 5, 1),
          End_Date   => Time_Of (2018, 5, 10)));

   --  This should trigger an error in the
   --  dynamic predicate check
   Add (CC,
        Course'(
          Name       => To_Unbounded_String
                          ("Intro to Video Recording"),
          Start_Date => Time_Of (2019, 5, 1),
          End_Date   => Time_Of (2018, 5, 10)));

end Show_Dynamic_Predicate_Courses;
```

Runtime output

```
raised ADA.ASSERTIONS.ASSERTION_ERROR : Dynamic_Predicate failed at show_dynamic_
↪predicate_courses.adb:53
```

В этом примере пакет `Courses` определяет тип `Course` и тип `Course_Container`, объект которого содержит все курсы. Мы хотим обеспечить согласованность дат каждого курса, в частности, чтобы дата начала была не позже даты окончания. Чтобы обеспечить соблюдение этого правила, мы объявляем динамический предикат для типа `Course`, который будет действовать для каждого объекта. Предикат использует имя типа, так как будь-то это обычная переменная данного типа: такое имя обозначает экземпляр проверяемого объекта.

Обратите внимание, что в приведенном выше примере используются неограниченные строки и даты. Оба типа доступны в стандартной библиотеке Ада. Пожалуйста, обратитесь к следующим разделам для получения дополнительной информации:

- о типе неограниченной строки (`Unbounded_String`): раздел *Неограниченные строки* (page 233);
- о дате и времени: Раздел *Даты и время* (page 217).

Статические предикаты, как упоминалось выше, в основном используются для скалярных типов и проверяются во время компиляции. Они особенно полезны для представления несмежных элементов перечисления. Классический пример - список дней недели:

```
type Week is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
```

Мы можем легко создать подсписок рабочих дней в неделе, указав подтип (**subtype**) с диапазоном на основе `Week`. Например:

```
subtype Work_Week is Week range Mon .. Fri;
```

Диапазоны в Аде могут быть указаны только как непрерывные списки: они не позволяют нам выбирать определенные дни. Однако нам может понадобиться создать список, содержащий только первый, средний и последний день рабочей недели. Для этого мы используем статический предикат:

```
subtype Check_Days is Work_Week
  with Static_Predicate => Check_Days in Mon | Wed | Fri;
```

Давайте посмотрим на полный пример:

Listing 5: show_predicates.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Predicates is
4
5     type Week is (Mon, Tue, Wed, Thu,
6                   Fri, Sat, Sun);
7
8     subtype Work_Week is Week range Mon .. Fri;
9
10    subtype Test_Days is Work_Week
11      with Static_Predicate =>
12        Test_Days in Mon | Wed | Fri;
13
14    type Tests_Week is array (Week) of Natural
15      with Dynamic_Predicate =>
16        (for all I in Tests_Week'Range =>
17           (case I is
18                when Test_Days =>
19                   Tests_Week (I) > 0,
20                when others    =>
21                   Tests_Week (I) = 0));
22
```

(continues on next page)

(continued from previous page)

```
   Num_Tests : Tests_Week :=
                  (Mon => 3, Tue => 0,
                   Wed => 4, Thu => 0,
                   Fri => 2, Sat => 0,
                   Sun => 0);

   procedure Display_Tests (N : Tests_Week) is
   begin
      for I in Test_Days loop
         Put_Line ("# tests on "
                   & Test_Days'Image (I)
                   & " => "
                   & Integer'Image (N (I)));
      end loop;
   end Display_Tests;

begin
   Display_Tests (Num_Tests);

   --  Assigning non-conformant values to
   --  individual elements of the Tests_Week
   --  type does not trigger a predicate
   --  check:
   Num_Tests (Tue) := 2;

   --  However, assignments with the "complete"
   --  Tests_Week type trigger a predicate
   --  check. For example:
   --
   --  Num_Tests := (others => 0);

   --  Also, calling any subprogram with
   --  parameters of Tests_Week type
   --  triggers a predicate check. Therefore,
   --  the following line will fail:
   Display_Tests (Num_Tests);
end Show_Predicates;
```

Runtime output

```
# tests on MON =>  3
# tests on WED =>  4
# tests on FRI =>  2

raised ADA.ASSERTIONS.ASSERTION_ERROR : Dynamic_Predicate failed at show_
 ↪predicates.adb:58
```

Здесь у нас есть приложение, которое хочет проводить тесты только три дня в рабочей неделе. Эти дни указаны в подтипе `Test_Days`. Мы хотим отслеживать количество тестов, которые проводятся каждый день. Мы объявляем тип `Tests_Week` как массив, объект которого будет содержать количество тестов, выполняемых каждый день. Согласно нашим требованиям, эти тесты должны проводиться только в вышеупомянутые три дня; в другие дни никаких анализов проводить не следует. Это требование реализовано с помощью динамического предиката типа `Tests_Week`. Наконец, фактическая информация об этих тестах хранится в массиве `Num_Tests`, который является экземпляром типа `Tests_Week`.

Динамический предикат типа `Tests_Week` проверяется во время инициализации `Num_Tests`. Если у нас там будет несоответствующее значение, проверка не удастся. Однако, как мы видим в нашем примере, изменения отдельных элементов массива не приводят к выполнению проверки. Так происходит потому, что инициализация сложной структуры данных (например, массивов или записей) может требовать выполнения нескольких

операций присваивания. Однако, как только объект передается в качестве аргумента подпрограмме, динамический предикат проверяется, потому что подпрограмма требует, чтобы объект был согласован. Это происходит при вызове `Display_Tests` в последнем операторе нашего примера. Здесь проверка предиката вызовет ошибку потому, что предыдущее изменение приводит к несогласованному значению.

16.3 Инварианты типа

Инварианты типов - это еще один способ определить ожидания относительно типов. В то время как предикаты используются для всех типов, инварианты типов используются исключительно для определения ожиданий в отношении *личных типов*. Если тип T из пакета P имеет инвариант типа, результаты операций с объектами типа T всегда согласуются с этим инвариантом.

Инварианты типов указываются с помощью предложения `with Type_Invariant => <свойство>`. Подобно предикатам, *свойство* определяет условие, которое позволяет нам контролировать, соответствует ли объект типа T нашим требованиям. В этом смысле инварианты типов можно рассматривать как своего рода предикаты для личных типов. Однако есть некоторые отличия касающиеся проверок. Эти отличия приведены в следующей таблице:

Элемент	Проверки параметров подпрограммы	Проверки присвоения
Предикаты	По всем входящим **in** и выходящим **out** параметрам	При присваивании и явных инициализациях
Инварианты типов	По параметрам **out**, возвращенным из подпрограмм, объявленных в том же видимом разделе	При всех инициализациях

Мы могли бы переписать наш предыдущий пример и заменить динамические предикаты инвариантами типов. Это выглядело бы так:

Listing 6: show_type_invariant.adb

```
with Ada.Text_IO;           use Ada.Text_IO;
with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
with Ada.Calendar;          use Ada.Calendar;
with Ada.Containers.Vectors;

procedure Show_Type_Invariant is

   package Courses is
      type Course is private
        with Type_Invariant => Check (Course);

      type Course_Container is private;

      procedure Add (CC : in out Course_Container;
                     C  :        Course);

      function Init
        (Name                : String;
         Start_Date, End_Date : Time) return Course;

      function Check (C : Course) return Boolean;

   private
      type Course is record
```

(continues on next page)

(continued from previous page)

```
         Name       : Unbounded_String;
         Start_Date : Time;
         End_Date   : Time;
      end record;

      function Check (C : Course) return Boolean is
        (C.Start_Date <= C.End_Date);

      package Course_Vectors is new
        Ada.Containers.Vectors
          (Index_Type   => Natural,
           Element_Type => Course);

      type Course_Container is record
         V : Course_Vectors.Vector;
      end record;
   end Courses;

   package body Courses is
      procedure Add (CC : in out Course_Container;
                     C  :        Course) is
      begin
         CC.V.Append (C);
      end Add;

      function Init
        (Name                 : String;
         Start_Date, End_Date : Time) return Course is
      begin
         return
           Course'(Name       => To_Unbounded_String (Name),
                   Start_Date => Start_Date,
                   End_Date   => End_Date);
      end Init;
   end Courses;

   use Courses;

   CC : Course_Container;
begin
   Add (CC,
        Init (Name       => "Intro to Photography",
              Start_Date => Time_Of (2018, 5, 1),
              End_Date   => Time_Of (2018, 5, 10)));

   --  This should trigger an error in the
   --  type-invariant check
   Add (CC,
        Init (Name       => "Intro to Video Recording",
              Start_Date => Time_Of (2019, 5, 1),
              End_Date   => Time_Of (2018, 5, 10)));
end Show_Type_Invariant;
```

Build output

```
show_type_invariant.adb:1:09: warning: no entities of "Ada.Text_IO" are referenced␣
 ↪[-gnatwu]
show_type_invariant.adb:1:29: warning: use clause for package "Text_IO" has no␣
 ↪effect [-gnatwu]
```

Runtime output

```
raised ADA.ASSERTIONS.ASSERTION_ERROR : failed invariant from show_type_invariant.
 ↪adb:10
```

Основное отличие состоит в том, что в предыдущем примере тип `Course` был полностью описан в видимом разделе пакета `Courses`, но в этом примере он является личным типом.

CHAPTER

SEVENTEEN

ВЗАИМОДЕЙСТВИЕ С ЯЗЫКОМ C

Ада позволяет нам взаимодействовать с кодом на многих языках, включая C и C++. В этом разделе обсуждается, как взаимодействовать с C.

17.1 Многоязычный проект

По умолчанию при использовании **gprbuild** мы компилируем только исходные файлы Ада. Чтобы скомпилировать файлы C, нам нужно изменить файл проекта, используемый **gprbuild**. Мы добавляем запись `Languages`, как в следующем примере:

```
project Multilang is

   for Languages use ("ada", "c");

   for Source_Dirs use ("src");
   for Main use ("main.adb");
   for Object_Dir use "obj";

end Multilang;
```

17.2 Соглашение о типах

Для взаимодействия с типами данных, объявленными в приложении на C, необходимо указать аспект `Convention` в соответствующем объявлении типа Ада. В следующем примере вводим перечисленимый тип `C_Enum`, для соответствующего типа в исходном файле C:

Listing 1: show_c_enum.adb

```
1  procedure Show_C_Enum is
2
3     type C_Enum is (A, B, C)
4       with Convention => C;
5     --  Use C convention for C_Enum
6  begin
7     null;
8  end Show_C_Enum;
```

Для взаимодействия со встроенными типами C используется пакет `Interfaces.C`, содержащий большинство необходимых определений типов. Например:

Listing 2: show_c_struct.adb

```
with Interfaces.C; use Interfaces.C;

procedure Show_C_Struct is

   type c_struct is record
      a : int;
      b : long;
      c : unsigned;
      d : double;
   end record
     with Convention => C;

begin
   null;
end Show_C_Struct;
```

Здесь мы взаимодействуем со структурой C (`C_Struct`) и используем соответствующие типы данных C (**`int`**, **`long`**, **`unsigned`** и **`double`**). А вот объявление в C:

Listing 3: c_struct.h

```
struct c_struct
{
    int         a;
    long        b;
    unsigned    c;
    double      d;
};
```

17.3 Подпрограммы на других языках

17.3.1 Вызов подпрограмм C из Ады

Мы используем аналогичный подход при взаимодействии с подпрограммами, написанными на C. Рассмотрим следующее объявление в заголовочном файле C:

Listing 4: my_func.h

```
int my_func (int a);
```

Вот соответствующее определение функции на C:

Listing 5: my_func.c

```
#include "my_func.h"

int my_func (int a)
{
    return a * 2;
}
```

Мы можем связать этот код с кодом на Аде, указывая аспект `Import`. Например:

Listing 6: show_c_func.adb

```
with Interfaces.C; use Interfaces.C;
with Ada.Text_IO;  use Ada.Text_IO;

procedure Show_C_Func is

   function my_func (a : int) return int
     with
       Import        => True,
       Convention    => C;

   --  Imports function 'my_func' from C.
   --  You can now call it from Ada.

   V : int;
begin
   V := my_func (2);
   Put_Line ("Result is " & int'Image (V));
end Show_C_Func;
```

При необходимости можно использовать другое имя подпрограммы в Ада коде. Например, можно назвать функцию `Get_Value`:

Listing 7: show_c_func.adb

```
with Interfaces.C; use Interfaces.C;
with Ada.Text_IO;  use Ada.Text_IO;

procedure Show_C_Func is

   function Get_Value (a : int) return int
     with
       Import        => True,
       Convention    => C,
       External_Name => "my_func";

   --  Imports function 'my_func' from C and
   --  rename it to 'Get_Value'

   V : int;
begin
   V := Get_Value (2);
   Put_Line ("Result is " & int'Image (V));
end Show_C_Func;
```

17.3.2 Вызов Ада подпрограмм из C

Вы также можете вызывать Ада подпрограммы из C приложений. Это делается с помощью аспекта `Export`. Например:

Listing 8: c_api.ads

```
with Interfaces.C; use Interfaces.C;

package C_API is

   function My_Func (a : int) return int
     with
```

(continues on next page)

(continued from previous page)

```
      Export        => True,
      Convention    => C,
      External_Name => "my_func";

end C_API;
```

Вот соответствующее тело пакета с реализфцией этой функции:

Listing 9: c_api.adb

```
package body C_API is

   function My_Func (a : int) return int is
   begin
      return a * 2;
   end My_Func;

end C_API;
```

На стороне C мы делаем так, как если бы функция была написана на C: просто объявляем ее с помощью ключевого слова **extern**. Например:

Listing 10: main.c

```
#include <stdio.h>

extern int my_func (int a);

int main (int argc, char **argv) {

  int v = my_func(2);

  printf("Result is %d\n", v);

  return 0;
}
```

17.4 Внешние переменные

17.4.1 Использование глобальных переменных C в Аде

Чтобы использовать глобальные переменные из C, мы используем тот же метод, что и для подпрограмм: мы указываем аспекты `Import` и `Convention` для каждой переменной, которую мы хотим импортировать.

Давайте воспользуемся примером из предыдущего раздела. Мы добавим глобальную переменную (`func_cnt`) для подсчета количества вызовов функции (`my_func`):

Listing 11: test.h

```
extern int func_cnt;

int my_func (int a);
```

Переменная объявлена в файле C и увеличивается в `my_func`:

Listing 12: test.c

```
#include "test.h"

int func_cnt = 0;

int my_func (int a)
{
  func_cnt++;

  return a * 2;
}
```

В коде на Аде мы просто ссылаемся на внешнюю переменную:

Listing 13: show_c_func.adb

```
with Interfaces.C; use Interfaces.C;
with Ada.Text_IO;  use Ada.Text_IO;

procedure Show_C_Func is

   function my_func (a : int) return int
     with
       Import     => True,
       Convention => C;

   V : int;

   func_cnt : int
     with
       Import        => True,
       Convention    => C;
   --  We can access the func_cnt variable
   --  from test.c

begin
   V := my_func (1);
   V := my_func (2);
   V := my_func (3);
   Put_Line ("Result is " & int'Image (V));

   Put_Line ("Function was called "
             & int'Image (func_cnt)
             & " times");
end Show_C_Func;
```

Как мы видим, запустив приложение, значение счетчика - это количество вызовов `my_func`.

Мы можем использовать аспект `External_Name`, если хотим сослаться на имя отличное от наименования переменной в Аде, как мы это делали для подпрограмм.

17.4.2 Использование переменных Ада в C

Вы также можете использовать переменные, объявленные в файлах Ада, в приложениях C. Точно так же, как мы делали для подпрограмм, вы делаете это с помощью аспекта `Export`.

Давайте повторно воспользуемся прошлым примером и добавим счетчик, аналогично предыдущему примеру, но на этот раз будем увеличивать счетчик в Ада коде:

Listing 14: c_api.ads

```
with Interfaces.C; use Interfaces.C;

package C_API is

   func_cnt : int := 0
     with
       Export     => True,
       Convention => C;

   function My_Func (a : int) return int
     with
       Export        => True,
       Convention    => C,
       External_Name => "my_func";

end C_API;
```

Затем переменная увеличивается в `My_Func`:

Listing 15: c_api.adb

```
package body C_API is

   function My_Func (a : int) return int is
   begin
      func_cnt := func_cnt + 1;
      return a * 2;
   end My_Func;

end C_API;
```

В приложении C нам просто нужно объявить переменную и использовать ее:

Listing 16: main.c

```
#include <stdio.h>

extern int my_func (int a);

extern int func_cnt;

int main (int argc, char **argv) {

  int v;

  v = my_func(1);
  v = my_func(2);
  v = my_func(3);

  printf("Result is %d\n", v);

  printf("Function was called %d times\n", func_cnt);
```

(continues on next page)

(continued from previous page)

```
18
19    return 0;
20 }
```

Опять же, запустив приложение, мы видим, что значение счетчика - это количество вызовов `my_func`.

17.5 Автоматическое создание связок

В приведенных выше примерах мы вручную добавили аспекты в наш код Ада, чтобы они соответствовали исходному коду C, с которым мы взаимодействуем. Это называется созданием *связки*. Мы можем автоматизировать этот процесс, используя особый ключ компилятора для *дампа спецификации Ада*: `-fdump-ada-spec`. Мы проиллюстрируем это, вернувшись к нашему предыдущему примеру.

Это был наш заголовочный файл C:

Listing 17: my_func.c

```
1 extern int func_cnt;
2
3 int my_func (int a);
```

Чтобы создать связку на Аде, мы вызовем компилятор следующим образом:

```
gcc -c -fdump-ada-spec -C ./test.h
```

Результатом является файл спецификации Ада с именем `test_h.ads`:

Listing 18: test_h.ads

```
1  pragma Ada_2005;
2  pragma Style_Checks (Off);
3
4  with Interfaces.C; use Interfaces.C;
5
6  package test_h is
7
8     func_cnt : aliased int;  -- ./test.h:3
9     pragma Import (C, func_cnt, "func_cnt");
10
11    function my_func (arg1 : int) return int;  -- ./test.h:5
12    pragma Import (C, my_func, "my_func");
13
14 end test_h;
```

Теперь мы просто ссылаемся на этот пакет `test_h` в нашем приложении Ада:

Listing 19: show_c_func.adb

```
1  with Interfaces.C; use Interfaces.C;
2  with Ada.Text_IO;  use Ada.Text_IO;
3  with test_h;       use test_h;
4
5  procedure Show_C_Func is
6     V : int;
7  begin
8     V := my_func (1);
```

(continues on next page)

(continued from previous page)

```
   V := my_func (2);
   V := my_func (3);
   Put_Line ("Result is " & int'Image (V));

   Put_Line ("Function was called "
             & int'Image (func_cnt)
             & " times");
end Show_C_Func;
```

Вы можете указать имя родительского модуля создаваемых связок в качестве операнда для `fdump-ada-spec`:

```
gcc -c -fdump-ada-spec -fada-spec-parent=Ext_C_Code -C ./test.h
```

И получим файл `ext_c_code-test_h.ads`:

Listing 20: ext_c_code-test_h.ads

```
package Ext_C_Code.test_h is

   --  automatic generated bindings...

end Ext_C_Code.test_h;
```

17.5.1 Адаптация связок

Компилятор делает все возможное при создании связок для файла заголовка C. Однако мы получаем достаточно хороший результат и сгенерированные связки не всегда соответствуют нашим ожиданиям. Например, так может произойти при создании связок для функций, которые имеют указатели в качестве аргументов. В этом случае компилятор может использовать `System.Address` как тип одного или нескольких указателей. Хотя этот подход работает нормально (как мы увидим позже), обычно человек не так интерпретирует заголовочный файл C. Следующий пример иллюстрирует эту проблему.

Начнем с такого заголовочного файла C:

Listing 21: test.h

```
struct test;

struct test * test_create(void);

void test_destroy(struct test *t);

void test_reset(struct test *t);

void test_set_name(struct test *t, char *name);

void test_set_address(struct test *t, char *address);

void test_display(const struct test *t);
```

И соответствующей реализация на C:

Listing 22: test.c

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
```

(continues on next page)

(continued from previous page)

```
4
5   #include "test.h"
6
7   struct test {
8     char name[80];
9     char address[120];
10  };
11
12  static size_t
13  strlcpy(char *dst, const char *src, size_t dstsize)
14  {
15    size_t len = strlen(src);
16    if (dstsize) {
17      size_t bl = (len < dstsize-1 ? len : dstsize-1);
18      ((char*)memcpy(dst, src, bl))[bl] = 0;
19    }
20    return len;
21  }
22
23  struct test * test_create(void)
24  {
25    return malloc (sizeof (struct test));
26  }
27
28  void test_destroy(struct test *t)
29  {
30    if (t != NULL) {
31      free(t);
32    }
33  }
34
35  void test_reset(struct test *t)
36  {
37    t->name[0]    = '\0';
38    t->address[0] = '\0';
39  }
40
41  void test_set_name(struct test *t, char *name)
42  {
43    strlcpy(t->name, name, sizeof(t->name));
44  }
45
46  void test_set_address(struct test *t, char *address)
47  {
48    strlcpy(t->address, address, sizeof(t->address));
49  }
50
51  void test_display(const struct test *t)
52  {
53    printf("Name:    %s\n", t->name);
54    printf("Address: %s\n", t->address);
55  }
```

Далее мы создадим наши связки:

```
gcc -c -fdump-ada-spec -C ./test.h
```

Это создает следующую спецификацию в `test_h.ads`:

Listing 23: test_h.ads

```
pragma Ada_2005;
pragma Style_Checks (Off);

with Interfaces.C; use Interfaces.C;
with System;
with Interfaces.C.Strings;

package test_h is

   --  skipped empty struct test

   function test_create return System.Address;  -- ./test.h:5
   pragma Import (C, test_create, "test_create");

   procedure test_destroy (arg1 : System.Address);  -- ./test.h:7
   pragma Import (C, test_destroy, "test_destroy");

   procedure test_reset (arg1 : System.Address);  -- ./test.h:9
   pragma Import (C, test_reset, "test_reset");

   procedure test_set_name (arg1 : System.Address; arg2 : Interfaces.C.Strings.
↪chars_ptr);  -- ./test.h:11
   pragma Import (C, test_set_name, "test_set_name");

   procedure test_set_address (arg1 : System.Address; arg2 : Interfaces.C.Strings.
↪chars_ptr);  -- ./test.h:13
   pragma Import (C, test_set_address, "test_set_address");

   procedure test_display (arg1 : System.Address);  -- ./test.h:15
   pragma Import (C, test_display, "test_display");

end test_h;
```

Как мы видим, генератор связки полностью игнорирует объявление `struct test`, и все ссылки на структуру `test` заменяются адресами (`System.Address`). Тем не менее, эти связки достаточно хороши, чтобы позволить нам создать тестовое приложение на Ада:

Listing 24: show_automatic_c_struct_bindings.adb

```
with Interfaces.C;         use Interfaces.C;
with Interfaces.C.Strings; use Interfaces.C.Strings;
with Ada.Text_IO;          use Ada.Text_IO;
with test_h;               use test_h;

with System;

procedure Show_Automatic_C_Struct_Bindings is

   Name    : constant chars_ptr :=
     New_String ("John Doe");
   Address : constant chars_ptr :=
     New_String ("Small Town");

   T : System.Address := test_create;

begin
   test_reset (T);
   test_set_name (T, Name);
   test_set_address (T, Address);
```

(continues on next page)

(continued from previous page)

```
21
22    test_display (T);
23    test_destroy (T);
24 end Show_Automatic_C_Struct_Bindings;
```

Мы можем успешно собрать такую программу, используя автоматически сгенерированные связки, но они не идеальны. Вместо этого мы предпочли бы связки на Аде, которые соответствуют нашей (человеческой) интерпретации файла заголовка C. Это требует ручного анализа файла заголовка. Хорошая новость заключается в том, что мы можем использовать автоматически сгенерированные связки в качестве отправной точки и адаптировать их к нашим потребностям. Например, мы можем:

1. Определить тип `Test` на основе `System.Address` и использовать его во всех соответствующих функциях.
2. Удалить префикс `test_` во всех операциях с типом `Test`.

Вот итоговая версия спецификации:

Listing 25: adapted_test_h.ads

```
1  with Interfaces.C; use Interfaces.C;
2  with System;
3  with Interfaces.C.Strings;
4
5  package adapted_test_h is
6
7     type Test is new System.Address;
8
9     function Create return Test;
10    pragma Import (C, Create, "test_create");
11
12    procedure Destroy (T : Test);
13    pragma Import (C, Destroy, "test_destroy");
14
15    procedure Reset (T : Test);
16    pragma Import (C, Reset, "test_reset");
17
18    procedure Set_Name (T    : Test;
19                        Name : Interfaces.C.Strings.chars_ptr);  -- ./test.h:11
20    pragma Import (C, Set_Name, "test_set_name");
21
22    procedure Set_Address (T       : Test;
23                           Address : Interfaces.C.Strings.chars_ptr);
24    pragma Import (C, Set_Address, "test_set_address");
25
26    procedure Display (T : Test);  -- ./test.h:15
27    pragma Import (C, Display, "test_display");
28
29 end adapted_test_h;
```

И это соответствующее тело на Аде:

Listing 26: show_adapted_c_struct_bindings.adb

```
1  with Interfaces.C;         use Interfaces.C;
2  with Interfaces.C.Strings; use Interfaces.C.Strings;
3  with adapted_test_h;       use  adapted_test_h;
4
5  with System;
6
```

(continues on next page)

(continued from previous page)

```
procedure Show_Adapted_C_Struct_Bindings is

   Name    : constant chars_ptr :=
     New_String ("John Doe");
   Address : constant chars_ptr :=
     New_String ("Small Town");

   T : Test := Create;

begin
   Reset (T);
   Set_Name (T, Name);
   Set_Address (T, Address);

   Display (T);
   Destroy (T);
end Show_Adapted_C_Struct_Bindings;
```

Теперь мы можем использовать тип `Test` и его операции в понятной и удобочитаемой форме.

CHAPTER

EIGHTEEN

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Объектно-ориентированное программирование (ООП) - это большая и расплывчатая концепция в языках программирования, которая имеет тенденцию включать в себя множество различных элементов, потому, что разные языки часто реализуют свое собственное видение этой концепции, предлагая в чем-то сходные, а в чем-то отличающиеся реализации.

Но одна из моделей, можно сказать, «выиграла» битву за звания "истинного" объектно-ориентированного подхода, хотя бы, если судить только по популярности. Это модель, используется в языке программирования Java, и она очень похожа на модель, используемую в C ++. Вот некоторые важные характеристики:

- Производные типы и расширение типов: Большинство объектно-ориентированных языков позволяют пользователю добавлять новые поля в производные типы.
- Заменяемость подтипов: Объекты типа, производного от базового типа, в некоторых случаях, могут использоваться вместо объектов базового типа.
- Полиморфизм среды выполнения: Вызов подпрограммы, обычно называемой *методом*, привязанной к типу объекта, может диспечеризироваться во время выполнения программы в зависимости от конкретного типа объекта.
- Инкапсуляция: Объекты могут скрывать некоторые свои данные.
- Расширяемость: пользователи "извне" вашего пакета или даже всей вашей библиотеки могут создавать производные от ваших объектных типов и определять их поведение по своему.

Ада появилась еще до того, как объектно-ориентированное программирование стало таким уж популярным, как и сегодня. Некоторые механизмы и концепции из приведенного выше списка были в самой ранней версии Ада еще до того, как было добавлено то, что мы бы назвали поддержкой ООП:

- Как мы видели, инкапсуляция реализована в Аде не на уровне типа, а на уровне пакета.
- Заменяемость подтипов может быть реализована с использованием, ну да, подтипов, которые имеют полную и "разрешительную" (permissive) статическую модель замещаемости. Во время выполнения замена завершится неудачно, если динамические ограничения подтипа будут нарушены.
- Полиморфизм времени выполнения может быть реализован с использованием записей с вариантами.

Однако в этом списке нет расширения типов, если вы не считать записи с вариантами, и расширяемости.

Редакция Ада 1995 года добавила функцию, заполняющую пробелы, которая позволила людям проще программировать, следуя объектно-ориентированной парадигме. Эта функция называется *теговые типы*.

Note: Примечание: В Ада можно написать программу не создав даже одного тегового типа. Если вы предпочитаете такой стиль программирования или вам в данный момент не

нужны теговые типы, это нормально не использовать их, как в случае и со многими другими возможностями Ады.

Тем не менее, может оказаться, что они - наилучший способ выразить решение вашей задачи. А, раз это так, читайте дальше!

18.1 Производные типы

Прежде чем представить теговые типы, мы должны обсудить тему, которой мы уже касались, но на самом деле не углублялись в нее до сих пор:

Вы можете создать один или несколько новых типов из любого типа языка Ада. Производные типы встроены в язык.

Listing 1: newtypes.ads

```
package Newtypes is
   type Point is record
       X, Y : Integer;
   end record;

   type New_Point is new Point;
end Newtypes;
```

Наследование типов полезно для обеспечения строгой типизации, поскольку система типов рассматривает эти два типа как несовместимые.

Но этим дело не ограничивается: создавая производный тип вы наследуете от него многое. Вы наследуете не только представление данных, но также можете унаследовать и поведение.

Когда вы наследуете тип, вы также наследуете так называемые примитивные операции. *Примитивная операция* (или просто *примитив*) - это подпрограмма, привязанная к типу. Ада определяет примитивы как подпрограммы, определенные в той же области, что и тип.

Attention: Обратите внимание: подпрограмма станет примитивом этого типа только в том случае, если:

1. Подпрограмма объявляется в той же области, что и тип и
2. Тип и подпрограмма объявляются в пакете.

Listing 2: primitives.adb

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

procedure Primitives is
  package Week is
    type Days is (Monday, Tuesday, Wednesday,
                  Thursday, Friday,
                  Saturday, Sunday);

     --  Print_Day is a primitive
     --  of the type Days
    procedure Print_Day (D : Days);
  end Week;

  package body Week is
```

(continues on next page)

(continued from previous page)

```
15      procedure Print_Day (D : Days) is
16      begin
17         Put_Line (Days'Image (D));
18      end Print_Day;
19    end Week;
20
21    use Week;
22    type Weekend_Days is new
23      Days range Saturday .. Sunday;
24
25    --  A procedure Print_Day is automatically
26    --  inherited here. It is as if the procedure
27    --
28    --  procedure Print_Day (D : Weekend_Days);
29    --
30    --  has been declared with the same body
31
32    Sat : Weekend_Days := Saturday;
33  begin
34     Print_Day (Sat);
35  end Primitives;
```

Build output

```
primitives.adb:11:15: warning: procedure "Print_Day" is not referenced [-gnatwu]
primitives.adb:32:03: warning: "Sat" is not modified, could be declared constant [-
↪gnatwk]
```

Runtime output

```
SATURDAY
```

Этот вид наследования может быть очень полезным и не ограничивается типами записей (вы можете использовать его и для дискретных типов, как в примере выше), но он лишь внешне похож на объектно-ориентированное наследование:

- Записи не могут быть расширены с помощью этого механизма. Вы также не можете указать новое представление для нового типа: оно **всегда** будет то же, что и у базового типа. (*Прим. пер:* На самом деле, это не так и производные типы часто используются чтобы выполнять преобразование внутреннего представления типа.)
- Нет возможности для динамической диспетчеризации или полиморфизма. Объекты имеют фиксированный, статический тип.

Есть и другие различия, но перечислять их все здесь нет смысла. Просто помните, что эту форму наследования вы можете использовать, если хотите иметь только статически унаследованое поведение, избежать дублирования кода и использования композиции, и которое вам не подходит, если вам нужны какие-либо динамические возможности, которые обычно ассоциируются с ООП.

18.2 Теговые типы

Версия языка Ада 1995 года представила теговые типы, чтобы удовлетворить потребность в едином решении, которое позволяет программировать в объектно-ориентированном стиле, аналогичном тому, что был описан в начале этой главы.

Теговые типы очень похожи на обычные записи, за исключением того, что добавлена следующая функциональность:

- Типы имеют *тег*, хранящийся внутри каждого объекта и необходимый чтобы определить тип объекта во время выполнения[21].
- Для примитивов может приминяться диспечеризация. Примитив тегового типа - это то, что вы бы назвали *методом* в Java или C++. Если у вас есть тип, производный от базового типа с переопределенным примитивом, то при вызове примитива для объекта, какой примитив вызовется будет зависить от точного типа объекта в момент исполнения.
- Введены специальные правила, позволяющие теговому типу, производному от базового типа, быть статически совместимым с базовым типом.

Давайте посмотрим на наши первые объявления тегового типа:

Listing 3: p.ads

```
package P is
   type My_Class is tagged null record;
   --  Just like a regular record, but
   --  with tagged qualifier

   --  Methods are outside of the type
   --  definition:

   procedure Foo (Self : in out My_Class);
   --  If you define a procedure taking a
   --  My_Class argument in the same package,
   --  it will be a method.

   --  Here's how you derive a tagged type:

   type Derived is new My_Class with record
       A : Integer;
       --  You can add fields in derived types.
   end record;

   overriding procedure Foo (Self : in out Derived);
   --  The "overriding" qualifier is optional,
   --  but if it is present, it must be valid.
end P;
```

Listing 4: p.adb

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

package body P is
   procedure Foo (Self : in out My_Class) is
   begin
      Put_Line ("In My_Class.Foo");
   end Foo;

   procedure Foo (Self : in out Derived) is
```

(continues on next page)

[21] https://ru.wikipedia.org/wiki/Динамическая_идентификация_типа_данных

(continued from previous page)

```
   begin
      Put_Line ("In Derived.Foo, A = "
                & Integer'Image (Self.A));
   end Foo;
end P;
```

18.3 Надклассовые типы

Чтобы сохранить согласованность языка, необходимо было ввести новую нотацию, обозначающую: "Данный объект относится к этому теговому типу или любому его потомку".

В Аде мы называем это *надклассовым типом*. Он используется в ООП, как только вам понадобится полиморфизм. Например, вы не можете выполнить следующие действия:

Listing 5: main.adb

```
with P; use P;

procedure Main is

   O1 : My_Class;
   --  Declaring an object of type My_Class

   O2 : Derived := (A => 12);
   --  Declaring an object of type Derived

   O3 : My_Class := O2;
   --  INVALID: Trying to assign a value
   --  of type derived to a variable of
   --  type My_Class.
begin
   null;
end Main;
```

Build output

```
main.adb:11:21: error: expected type "My_Class" defined at p.ads:2
main.adb:11:21: error: found type "Derived" defined at p.ads:16
gprbuild: *** compilation phase failed
```

Это связано с тем, что объект типа T имеет в точности тип T, независимо от того, является T теговым или нет. То, что программист пытается тут сказать, это «Я хочу, чтобы O3 содержал объект типа `My_Class` или любого производного от `My_Class` типа». Вот как вы это делаете:

Listing 6: main.adb

```
with P; use P;

procedure Main is
   O1 : My_Class;
   --  Declare an object of type My_Class

   O2 : Derived := (A => 12);
   --  Declare an object of type Derived

   O3 : My_Class'Class := O2;
   --  Now valid: My_Class'Class designates
   --  the classwide type for My_Class,
```

(continues on next page)

(continued from previous page)

```
   --  which is the set of all types
   --  descending from My_Class (including
   --  My_Class).
begin
   null;
end Main;
```

Build output

```
main.adb:4:04: warning: variable "O1" is not referenced [-gnatwu]
main.adb:7:04: warning: "O2" is not modified, could be declared constant [-gnatwk]
main.adb:10:04: warning: variable "O3" is not referenced [-gnatwu]
```

Attention: Обратите внимание: Поскольку объект надклассового типа может быть размером с любого потомка его базового типа, то его размер заранее неизвестен. Таким образом, это неопределенный тип со всеми ожидаемыми ограничениями:

- Он не может быть использован для поля/компоненты записи
- Объект надклассового типа должен быть инициализирован немедленно (вы не можете ограничить такой тип каким-либо иным способом, кроме как путем его инициализации).

18.4 Операции диспетчеризации

Мы увидели, что можно переопределять операции в типах, производных от другого тегового типа. Конечной целью ООП является выполнение диспетчеризируемого вызова: когда вызываемый примитив (метод) определяется точным типом объекта.

Но если задуматься, переменная типа `My_Class` всегда содержит объект именно данного типа. Если требуется переменная, которая может содержать `My_Class` или любой производный тип, она должна иметь тип `My_Class'Class`.

Другими словами, чтобы сделать диспетчеризируемый вызов, вы должны сначала получить объект, который может иметь либо конкретный тип, либо любой тип, производным от этого конкретного типа, а именно объект надклассового типа.

Listing 7: main.adb

```
with P; use P;

procedure Main is
   O1 : My_Class;
   --  Declare an object of type My_Class

   O2 : Derived := (A => 12);
   --  Declare an object of type Derived

   O3 : My_Class'Class := O2;

   O4 : My_Class'Class := O1;
begin
   Foo (O1);
   --  Non dispatching: Calls My_Class.Foo
   Foo (O2);
   --  Non dispatching: Calls Derived.Foo
   Foo (O3);
```

(continues on next page)

(continued from previous page)

```
19    --  Dispatching: Calls Derived.Foo
20    Foo (O4);
21    --  Dispatching: Calls My_Class.Foo
22 end Main;
```

Runtime output

```
In My_Class.Foo
In Derived.Foo, A =  12
In Derived.Foo, A =  12
In My_Class.Foo
```

Внимание

Вы можете преобразовать объект типа `Derived` в объект типа `My_Class`. В Аде это называется *преобразованием представления* и полезно, например, если вы хотите вызвать родительский метод.

В том случае, когда объект действительно преобразуется в объект `My_Class`, что включает изменение его тега. Поскольку теговые объекты всегда передаются по ссылке, вы можете использовать этот вид преобразования для изменения состояния объекта: изменения в преобразованном объекте повлияют на оригинал. (*Прим. пер.:* Это не так, только преобразование представление позволяет менять оригинал.)

Listing 8: main.adb

```
1  with P; use P;
2
3  procedure Main is
4     O1 : Derived := (A => 12);
5     --  Declare an object of type Derived
6
7     O2 : My_Class := My_Class (O1);
8
9     O3 : My_Class'Class := O2;
10 begin
11    Foo (O1);
12    --  Non dispatching: Calls Derived.Foo
13    Foo (O2);
14    --  Non dispatching: Calls My_Class.Foo
15
16    Foo (O3);
17    --  Dispatching: Calls My_Class.Foo
18 end Main;
```

Runtime output

```
In Derived.Foo, A =  12
In My_Class.Foo
In My_Class.Foo
```

18.5 Точечная нотация

Вы также можете вызывать примитивы теговых типов с помощью нотации, более привычной объектно-ориентированным программистам. Учитывая приведенный выше примитив Foo, вы также можете написать указанную выше программу следующим образом:

Listing 9: main.adb

```
with P; use P;

procedure Main is
   O1 : My_Class;
   --  Declare an object of type My_Class

   O2 : Derived := (A => 12);
   --  Declare an object of type Derived

   O3 : My_Class'Class := O2;

   O4 : My_Class'Class := O1;
begin
   O1.Foo;
   --  Non dispatching: Calls My_Class.Foo
   O2.Foo;
   --  Non dispatching: Calls Derived.Foo
   O3.Foo;
   --  Dispatching: Calls Derived.Foo
   O4.Foo;
   --  Dispatching: Calls My_Class.Foo
end Main;
```

Runtime output

```
In My_Class.Foo
In Derived.Foo, A =  12
In Derived.Foo, A =  12
In My_Class.Foo
```

Если диспетчерезирующий параметр примитива является первым параметром, как в наших примерах, вы можете вызвать примитив, используя точечную нотацию. Все оставшиеся параметры передаются обычным образом:

Listing 10: main.adb

```
with P; use P;

procedure Main is
   package Extend is
      type D2 is new Derived with null record;

      procedure Bar (Self : in out D2;
                     Val  :        Integer);
   end Extend;

   package body Extend is
      procedure Bar (Self : in out D2;
                     Val  :        Integer) is
      begin
         Self.A := Self.A + Val;
      end Bar;
   end Extend;
```

(continues on next page)

(continued from previous page)

```
18
19    use Extend;
20
21    Obj : D2 := (A => 15);
22 begin
23    Obj.Bar (2);
24    Obj.Foo;
25 end Main;
```

Runtime output

```
In Derived.Foo, A =  17
```

18.6 Личные и лимитируемые типы с тегами

Ранее мы видели (в главе *Изоляция* (page 105)), что типы могут быть объявлены лимитируемыми или личными. Эти методы инкапсуляции также могут применяться к теговым типам, как мы увидим в этом разделе.

Это пример личного тегового типа:

Listing 11: p.ads

```
1 package P is
2    type T is tagged private;
3 private
4    type T is tagged record
5        E : Integer;
6    end record;
7 end P;
```

Это пример лимитируемого тегового типа:

Listing 12: p.ads

```
1 package P is
2    type T is tagged limited record
3        E : Integer;
4    end record;
5 end P;
```

Естественно, вы можете комбинировать как *лимитируемые*, так и *личные* типы и объявить лимитируемый личный теговый тип:

Listing 13: p.ads

```
1 package P is
2    type T is tagged limited private;
3
4    procedure Init (A : in out T);
5 private
6    type T is tagged limited record
7        E : Integer;
8    end record;
9 end P;
```

Listing 14: p.adb

```
package body P is

   procedure Init (A : in out T) is
   begin
      A.E := 0;
   end Init;

end P;
```

Listing 15: main.adb

```
with P; use P;

procedure Main is
  T1, T2 : T;
begin
  T1.Init;
  T2.Init;

  --  The following line doesn't work
  --  because type T is private:
  --
  --  T1.E := 0;

  --  The following line doesn't work
  --  because type T is limited:
  --
  --  T2 := T1;
end Main;
```

Обратите внимание, что код в процедуре `Main` имеет два оператора присваивания, которые вызывают ошибки компиляции, потому что тип T является лимитируемым личным. Фактически, вы не можете:

- присваивать `T1.E` непосредственно, потому что тип T является личным;
- присваивать T1 в T2, потому что тип T ограничен.

В этом случае нет различия между теговими типами и типами без тегов: эти ошибки компиляции также могут возникать и для нетеговых типов.

18.7 Надклассовые ссылочные типы

В этом разделе мы обсудим полезный шаблон для объектно-ориентированного программирования в Аде: надклассовые ссылочные типы. Начнем с примера, в котором мы объявляем теговый тип T и производный тип T_New:

Listing 16: p.ads

```
package P is
   type T is tagged null record;

   procedure Show (Dummy : T);

   type T_New is new T with null record;

```

(continues on next page)

(continued from previous page)

```
8     procedure Show (Dummy : T_New);
9  end P;
```

Listing 17: p.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body P is
4
5     procedure Show (Dummy : T) is
6     begin
7        Put_Line ("Using type "
8                  & T'External_Tag);
9     end Show;
10
11    procedure Show (Dummy : T_New) is
12    begin
13       Put_Line ("Using type "
14                 & T_New'External_Tag);
15    end Show;
16
17 end P;
```

Обратите внимание, как мы используем пустые записи для типов T и T_New. Хотя эти типы на самом деле не имеют каких-либо компонент, мы по-прежнему можем использовать их для демонстрации диспетчеризации. Также обратите внимание, что в приведенном выше примере используется атрибут 'External_Tag в реализации процедуры Show для получения строки с названием соответствующего тегового типа.

Как мы видели ранее, мы должны использовать надклассовый тип для создания объектов, которые могут выполнять диспетчерезируемые вызовы. Другими словами, будут диспетчеризироваться объекты типа T'Class. Например:

Listing 18: dispatching_example.adb

```
1  with P; use P;
2
3  procedure Dispatching_Example is
4    T2         :          T_New;
5    T_Dispatch : constant T'Class := T2;
6  begin
7    T_Dispatch.Show;
8  end Dispatching_Example;
```

Runtime output

```
Using type P.T_NEW
```

Более полезным приложением является объявление массива объектов, для которых будет выполняться диспетчеризация. Например, мы хотели бы объявить массив T_Arr, перебрать в цикле этот массив и выполнить диспетчеризацию в соответствии с фактическим типом каждого отдельного элемента массива:

```
for I in T_Arr'Range loop
   T_Arr (I).Show;
   --  Call Show procedure according
   --  to actual type of T_Arr (I)
end loop;
```

Однако непосредственно объявить массив с элементами T'Class невозможно:

Listing 19: classwide_compilation_error.adb

```
with P; use P;

procedure Classwide_Compilation_Error is
  T_Arr  : array (1 .. 2) of T'Class;
  --                         ^
  --               Compilation Error!
begin
  for I in T_Arr'Range loop
     T_Arr (I).Show;
  end loop;
end Classwide_Compilation_Error;
```

Build output

```
classwide_compilation_error.adb:4:31: error: unconstrained element type in array␣
 ↪declaration
gprbuild: *** compilation phase failed
```

В самом деле, компилятор не может знать, какой тип фактически будет использоваться для каждого элемента массива. Но, если мы используем динамическое распределение памяти используя ссылочные типы, мы сможем выделять объекты разных типов для отдельных элементов массива `T_Arr`. Мы делаем это с помощью надклассовых ссылочных типов, которые имеют следующий формат:

```
type T_Class is access T'Class;
```

Мы можем переписать предыдущий пример, используя тип `T_Class`. В этом случае динамически выделяемые объекты этого типа будут диспетчеризироваться в соответствии с фактическим типом, используемым во время выделения. Также давайте добавим процедуру `Init`, которая не будет переопределена для производного типа `T_New`. Это адаптированный код:

Listing 20: p.ads

```
package P is
   type T is tagged record
       E : Integer;
   end record;

   type T_Class is access T'Class;

   procedure Init (A : in out T);

   procedure Show (Dummy : T);

   type T_New is new T with null record;

   procedure Show (Dummy : T_New);

end P;
```

Listing 21: p.adb

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

package body P is

   procedure Init (A : in out T) is
```

(continues on next page)

(continued from previous page)

```
   begin
      Put_Line ("Initializing type T...");
      A.E := 0;
   end Init;

   procedure Show (Dummy : T) is
   begin
      Put_Line ("Using type "
                & T'External_Tag);
   end Show;

   procedure Show (Dummy : T_New) is
   begin
      Put_Line ("Using type "
                & T_New'External_Tag);
   end Show;

end P;
```

Listing 22: main.adb

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

procedure Main is
  T_Arr  : array (1 .. 2) of T_Class;
begin
  T_Arr (1) := new T;
  T_Arr (2) := new T_New;

  for I in T_Arr'Range loop
     Put_Line ("Element # "
               & Integer'Image (I));

     T_Arr (I).Init;
     T_Arr (I).Show;

     Put_Line ("-----------");
  end loop;
end Main;
```

Runtime output

```
Element #  1
Initializing type T...
Using type P.T
-----------
Element #  2
Initializing type T...
Using type P.T_NEW
-----------
```

В этом примере первый элемент (`T_Arr (1)`) имеет тип T, а второй элемент - тип `T_New`. При запуске примера процедура `Init` типа T вызывается для обоих элементов массива `T_Arr`, в то время, как вызов процедуры Show выберет нужную процедуру в соответствии с типом каждого элемента `T_Arr`.

CHAPTER

NINETEEN

СТАНДАРТНАЯ БИБЛИОТЕКА: КОНТЕЙНЕРЫ

В предыдущих главах мы использовали массивы в качестве стандартного способа, который группирует нескольких объектов определенного типа данных. Во многих случаях массивы достаточно хороши для работы с групрой объектов. Однако бывают ситуации, которые требуют большей гибкости и более совершенные операций. Для этих случаев Ада предоставляет поддержку контейнеров - таких как векторы и множества - в своей стандартной библиотеке.

Здесь мы представляем введение в контейнеры. Список всех контейнеров, имеющихся в Аде, см. в Приложении B.

19.1 Векторы

В следующих разделах мы представляем общий обзор векторов, включая создание экземпляров, инициализацию и операции с элементами вектора и самими векторами.

19.1.1 Создание экземпляра

Вот пример, показывающий настройку и объявление вектора V:

Listing 1: show_vector_inst.adb

```
with Ada.Containers.Vectors;

procedure Show_Vector_Inst is

   package Integer_Vectors is new
     Ada.Containers.Vectors
       (Index_Type   => Natural,
        Element_Type => Integer);

   V : Integer_Vectors.Vector;
begin
   null;
end Show_Vector_Inst;
```

Build output

```
show_vector_inst.adb:10:04: warning: variable "V" is not referenced [-gnatwu]
```

Контейнеры основаны на настраиваемых пакетах, поэтому мы не можем просто объявить вектор, как если бы объявляли массив определенного типа:

```
A : array (1 .. 10) of Integer;
```

Вместо этого нам сначала нужно создать экземпляр одного из этих пакетов. Мы используем пакет контейнера (в данном случае `Ada.Containers.Vectors`) и настраиваем его, чтобы создать экземпляр настраиваемого пакета для желаемого типа. Только затем мы сможем объявить вектор, используя тип из созданного пакета. Такая настройка необходима для любого типа контейнера стандартной библиотеки.

При настройки экземпляра `Integer_Vectors` мы указываем, что вектор содержит элементы типа **`Integer`**, указывая его как `Element_Type`. Подставляя для `Index_Type` тип **`Natural`**, мы указываем, что допустимый диапазон индекса включает все натуральные числа. При желании мы могли бы использовать более ограниченный диапазон.

19.1.2 Инициализация

Один из способов инициализации вектора - это конкатенация элементов. Мы используем оператор &, как показано в следующем примере:

Listing 2: show_vector_init.adb

```
with Ada.Containers; use Ada.Containers;
with Ada.Containers.Vectors;

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Vector_Init is

   package Integer_Vectors is new
     Ada.Containers.Vectors
       (Index_Type   => Natural,
        Element_Type => Integer);

   use Integer_Vectors;

   V : Vector := 20 & 10 & 0 & 13;
begin
   Put_Line ("Vector has "
             & Count_Type'Image (V.Length)
             & " elements");
end Show_Vector_Init;
```

Build output

```
show_vector_init.adb:15:04: warning: "V" is not modified, could be declared␣
↪constant [-gnatwk]
```

Runtime output

```
Vector has  4 elements
```

Мы указываем **`use`** `Integer_Vectors`, чтобы получить прямой доступ к типам и операциям из созданного пакета. Кроме того, пример знакомит нас с еще одной операцией вектора: `Length`, она возвращает количество элементов в векторе. Мы можем использовать точечную нотацию, потому что `Vector` - это теговый тип, и это позволяет нам писать, как `V.Length`, так и `Length (V)`.

19.1.3 Добавление элементов

Вы добавляете элементы в вектор с помощью операций Prepend и Append. Как следует из названий, эти операции добавляют элементы в начало или конец вектора соответственно. Например:

Listing 3: show_vector_append.adb

```
with Ada.Containers; use Ada.Containers;
with Ada.Containers.Vectors;

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Vector_Append is

   package Integer_Vectors is new
     Ada.Containers.Vectors
       (Index_Type   => Natural,
        Element_Type => Integer);

   use Integer_Vectors;

   V : Vector;
begin
   Put_Line ("Appending some elements to the vector...");
   V.Append (20);
   V.Append (10);
   V.Append (0);
   V.Append (13);
   Put_Line ("Finished appending.");

   Put_Line ("Prepending some elements to the vector...");
   V.Prepend (30);
   V.Prepend (40);
   V.Prepend (100);
   Put_Line ("Finished prepending.");

   Put_Line ("Vector has "
             & Count_Type'Image (V.Length)
             & " elements");
end Show_Vector_Append;
```

Runtime output

```
Appending some elements to the vector...
Finished appending.
Prepending some elements to the vector...
Finished prepending.
Vector has  7 elements
```

В этом примере элементы помещаются в вектор в следующей последовательности: (100, 40, 30, 20, 10, 0, 13).

Справочное руководство указывает, что сложность наихудшего случая должна быть:

- O(log N) для операции Append и
- O(N log N) для операции Prepend.

19.1.4 Доступ к первому и последнему элементам

Мы получаем доступ к первому и последнему элементам вектора с помощью функций `First_Element` и `Last_Element`. Например:

Listing 4: show_vector_first_last_element.adb

```
with Ada.Containers; use Ada.Containers;
with Ada.Containers.Vectors;

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Vector_First_Last_Element is

   package Integer_Vectors is new
     Ada.Containers.Vectors
       (Index_Type   => Natural,
        Element_Type => Integer);

   use Integer_Vectors;

   function Img (I : Integer)    return String
     renames Integer'Image;
   function Img (I : Count_Type) return String
     renames Count_Type'Image;

   V : Vector := 20 & 10 & 0 & 13;
begin
   Put_Line ("Vector has "
             & Img (V.Length)
             & " elements");

   --  Using V.First_Element to
   --  retrieve first element
   Put_Line ("First element is "
             & Img (V.First_Element));

   --  Using V.Last_Element to
   --  retrieve last element
   Put_Line ("Last element is "
             & Img (V.Last_Element));
end Show_Vector_First_Last_Element;
```

Build output

```
show_vector_first_last_element.adb:20:04: warning: "V" is not modified, could be␣
 ↪declared constant [-gnatwk]
```

Runtime output

```
Vector has  4 elements
First element is  20
Last element is  13
```

Вы можете поменять местами элементы с помощью процедуры Swap, передав ей *курсоры* на первый и последний элементы вектора полученные функциями `First` и `Last`. Курсор используются при переборе элементов контейнера и для указания на отдельные его элементы.

С помощью этих операций мы можем написать код, чтобы поменять местами первый и последний элементы вектора:

Listing 5: show_vector_first_last_element.adb

```
with Ada.Containers; use Ada.Containers;
with Ada.Containers.Vectors;

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Vector_First_Last_Element is

   package Integer_Vectors is new
     Ada.Containers.Vectors
       (Index_Type   => Natural,
        Element_Type => Integer);

   use Integer_Vectors;

   function Img (I : Integer) return String
     renames Integer'Image;

   V : Vector := 20 & 10 & 0 & 13;
begin
   --  We use V.First and V.Last to retrieve
   --  cursor for first and last elements.
   --  We use V.Swap to swap elements.
   V.Swap (V.First, V.Last);

   Put_Line ("First element is now "
             & Img (V.First_Element));
   Put_Line ("Last element is now "
             & Img (V.Last_Element));
end Show_Vector_First_Last_Element;
```

Runtime output

```
First element is now  13
Last element is now  20
```

19.1.5 Итерация

Самый простой способ перебрать элементы контейнера - использовать цикл `for E of Our_Container`. Это дает нам ссылку (E) на элемент в текущей позиции. Затем мы можем использовать E непосредственно. Например:

Listing 6: show_vector_iteration.adb

```
with Ada.Containers.Vectors;

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Vector_Iteration is

   package Integer_Vectors is new
     Ada.Containers.Vectors
       (Index_Type   => Natural,
        Element_Type => Integer);

   use Integer_Vectors;

   function Img (I : Integer) return String
```

(continues on next page)

(continued from previous page)

```
     renames Integer'Image;

   V : Vector := 20 & 10 & 0 & 13;
begin
   Put_Line ("Vector elements are: ");

   --
   --  Using for ... of loop to iterate:
   --
   for E of V loop
      Put_Line ("- " & Img (E));
   end loop;

end Show_Vector_Iteration;
```

Build output

```
show_vector_iteration.adb:17:04: warning: "V" is not modified, could be declared␣
↪constant [-gnatwk]
```

Runtime output

```
Vector elements are:
-  20
-  10
-  0
-  13
```

Этот код отображает каждый элемент вектора V.

Поскольку у нас есть ссылка, мы можем отобразить не только значение элемента, но и изменить его. Например, мы могли бы легко записать цикл, чтобы добавить единицу к каждому элементу вектора V:

```
for E of V loop
   E := E + 1;
end loop;
```

Мы также можем использовать индексы для доступа к элементам вектора. Формат аналогичен циклу по элементам массива: мы используем в цикле `for I in <range>`. Диапазон строим из `V.First_Index` и `V.Last_Index`. Мы можем обратиться к текущему элементу, используя операцию индексирования массива: `V (I)`. Например:

Listing 7: show_vector_index_iteration.adb

```
with Ada.Containers.Vectors;

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Vector_Index_Iteration is

   package Integer_Vectors is new
     Ada.Containers.Vectors
       (Index_Type   => Natural,
        Element_Type => Integer);

   use Integer_Vectors;

   V : Vector := 20 & 10 & 0 & 13;
begin
   Put_Line ("Vector elements are: ");
```

(continues on next page)

(continued from previous page)

```
   --
   --  Using indices in a "for I in ..." loop
   --  to iterate:
   --
   for I in V.First_Index .. V.Last_Index loop
      --  Displaying current index I
      Put ("- ["
           & Extended_Index'Image (I)
           & "] ");

      Put (Integer'Image (V (I)));

      --  We could also use the V.Element (I)
      --  function to retrieve the element at
      --  the current index I

      New_Line;
   end loop;

end Show_Vector_Index_Iteration;
```

Build output

```
show_vector_index_iteration.adb:14:04: warning: "V" is not modified, could be␣
↪declared constant [-gnatwk]
```

Runtime output

```
Vector elements are:
- [ 0]  20
- [ 1]  10
- [ 2]  0
- [ 3]  13
```

Здесь, помимо вывода элементов вектора, мы также печатаем каждый индекс `I`, точно так же, как то, что мы можем сделать для индексов массива. Получить элемент можено, как с помощью краткой формы `V (I)`, так и с помощью вызова функции `V.Element (I)`, но не как `V.I`.

Как упоминалось в предыдущем разделе, курсоры можно использовать для итерации по контейнерам. Для этого используйте функцию `Iterate`, которая выдает курсоры для каждой позиции в векторе. Соответствующий цикл имеет формат `for C in V.Iterate loop`. Как и в предыдущем примере с использованием индексов, можно снова получить доступ к текущему элементу, используя курсор в качестве индекса массива `V (C)`. Например:

Listing 8: show_vector_cursor_iteration.adb

```
with Ada.Containers.Vectors;

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Vector_Cursor_Iteration is

   package Integer_Vectors is new
     Ada.Containers.Vectors
       (Index_Type   => Natural,
        Element_Type => Integer);

   use Integer_Vectors;

```

(continues on next page)

(continued from previous page)

```
   V : Vector := 20 & 10 & 0 & 13;
begin
   Put_Line ("Vector elements are: ");

   --
   --  Use a cursor to iterate in a loop:
   --
   for C in V.Iterate loop
      --  Using To_Index function to retrieve
      --  the index for the cursor position
      Put ("- ["
           & Extended_Index'Image (To_Index (C))
           & "] ");

      Put (Integer'Image (V (C)));

      --  We could use Element (C) to retrieve
      --  the vector element for the cursor
      --  position

      New_Line;
   end loop;

   --  Alternatively, we could iterate with a
   --  while-loop:
   --
   --  declare
   --     C : Cursor := V.First;
   --  begin
   --     while C /= No_Element loop
   --        some processing here...
   --
   --        C := Next (C);
   --     end loop;
   --  end;

end Show_Vector_Cursor_Iteration;
```

Build output

```
show_vector_cursor_iteration.adb:14:04: warning: "V" is not modified, could be␣
 ↪declared constant [-gnatwk]
```

Runtime output

```
Vector elements are:
- [ 0]  20
- [ 1]  10
- [ 2]  0
- [ 3]  13
```

Мы также могли бы использовать более длинную форму `Element (C)`, вместо `V (C)`, для доступа к элементу в цикле. В этом примере мы используем функцию `To_Index` для получения индекса, соответствующего текущему курсору.

Как показано в комментариях после цикла, мы также можем использовать цикл **`while ... loop`** для прохода по вектору. В этом случае мы должны начать с курсора первого элемента (полученного с помощью вызова `V.First`), а затем вызывать `Next (C)`, чтобы получить курсор для последующих элементов. Функция `Next (C)` возвращает `No_Element`, когда курсор достигает конца вектора. Используя курсор вы можете изменять элементы вектора непосредственно.

Вот как это выглядит при использовании, как индексов, так и курсоров:

```
--  Modify vector elements using index
for I in V.First_Index .. V.Last_Index loop
   V (I) := V (I) + 1;
end loop;

--  Modify vector elements using cursor
for C in V.Iterate loop
   V (C) := V (C) + 1;
end loop;
```

Справочное руководство требует, чтобы сложность доступа к элементу в наихудшем случае составляла O (log N).

Другой способ изменения элементов вектора - использование *процедуры обработки*, которая получает отдельный элемент и выполняет над ним некоторую работу. Вы можете вызвать `Update_Element`, передав курсор и ссылку на процедуру обработки. Например:

Listing 9: show_vector_update.adb

```
with Ada.Containers.Vectors;

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Vector_Update is

   package Integer_Vectors is new
     Ada.Containers.Vectors
       (Index_Type   => Natural,
        Element_Type => Integer);

   use Integer_Vectors;

   procedure Add_One (I : in out Integer) is
   begin
      I := I + 1;
   end Add_One;

   V : Vector := 20 & 10 & 12;
begin
   --
   --  Use V.Update_Element to process elements
   --
   for C in V.Iterate loop
      V.Update_Element (C, Add_One'Access);
   end loop;

end Show_Vector_Update;
```

Build output

```
show_vector_update.adb:3:09: warning: no entities of "Ada.Text_IO" are referenced␣
 ↪[-gnatwu]
show_vector_update.adb:3:19: warning: use clause for package "Text_IO" has no␣
 ↪effect [-gnatwu]
```

19.1.6 Поиск и изменение элементов

Вы можете найти определенный элемент в векторе и получить его индекс. Функция `Find_Index` вернет индекс первого элемента, соответствующего искомому значению. В качестве альтернативы вы можете использовать `Find`, чтобы получить курсор, ссылающийся на этот элемент. Например:

Listing 10: show_find_vector_element.adb

```
with Ada.Containers.Vectors;

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Find_Vector_Element is

   package Integer_Vectors is new
     Ada.Containers.Vectors
       (Index_Type   => Natural,
        Element_Type => Integer);

   use Integer_Vectors;

   V : Vector := 20 & 10 & 0 & 13;
   Idx : Extended_Index;
   C   : Cursor;
begin
   --  Using Find_Index to retrieve the index
   --  of element with value 10
   Idx := V.Find_Index (10);
   Put_Line ("Index of element with value 10 is "
             & Extended_Index'Image (Idx));

   --  Using Find to retrieve the cursor for
   --  the element with value 13
   C   := V.Find (13);
   Idx := To_Index (C);
   Put_Line ("Index of element with value 13 is "
             & Extended_Index'Image (Idx));
end Show_Find_Vector_Element;
```

Build output

```
show_find_vector_element.adb:14:04: warning: "V" is not modified, could be␣
↪declared constant [-gnatwk]
```

Runtime output

```
Index of element with value 10 is  1
Index of element with value 13 is  3
```

Как мы видели в предыдущем разделе, мы можем осуществлять прямой доступ к элементам вектора используя индекс или курсор. Но, если мы пытаемся получить доступ к элементу с недопустимым индексом или курсором, будет возбуждено исключение, поэтому мы сначала должны проверить, действителен ли индекс или курсор, прежде чем использовать его для доступа к элементу. В нашем примере `Find_Index` или `Find` могли не найти элемент в векторе. Мы проверяем эту ситуацию, сравнивая индекс с `No_Index` или курсора с `No_Element`. Например:

```
--  Modify vector element using index
if Idx /= No_Index then
   V (Idx) := 11;
```

(continues on next page)

(continued from previous page)

```
end if;

--  Modify vector element using cursor
if C /= No_Element then
   V (C) := 14;
end if;
```

Вместо того, чтобы писать `V (C) := 14`, мы могли бы использовать более длинную форму `V.Replace_Element (C, 14)`.

19.1.7 Вставка элементов

В предыдущих разделах мы видели примеры того, как добавлять элементы в вектор:

- с помощью оператора конкатенации (&) при объявлении вектора, или
- вызвав процедуры `Prepend` и `Append`.

Вам может потребоваться вставить элемент в определенное место, например, перед определенным элементом в векторе. Вы делаете это, вызывая `Insert`. Например:

Listing 11: show_vector_insert.adb

```
with Ada.Containers; use Ada.Containers;
with Ada.Containers.Vectors;

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Vector_Insert is

   package Integer_Vectors is new
     Ada.Containers.Vectors
       (Index_Type   => Natural,
        Element_Type => Integer);

   use Integer_Vectors;

   procedure Show_Elements (V : Vector) is
   begin
      New_Line;
      Put_Line ("Vector has "
                & Count_Type'Image (V.Length)
                & " elements");

      if not V.Is_Empty then
         Put_Line ("Vector elements are: ");
         for E of V loop
            Put_Line ("- " & Integer'Image (E));
         end loop;
      end if;
   end Show_Elements;

   V : Vector := 20 & 10 & 12;
   C : Cursor;
begin
   Show_Elements (V);

   New_Line;
   Put_Line ("Adding element with value 9 (before 10)...");

```

(continues on next page)

(continued from previous page)

```
   --
   --  Using V.Insert to insert the element
   --  into the vector
   --
   C := V.Find (10);
   if C /= No_Element then
      V.Insert (C, 9);
   end if;

   Show_Elements (V);

end Show_Vector_Insert;
```

Runtime output

```
Vector has  3 elements
Vector elements are:
-  20
-  10
-  12

Adding element with value 9 (before 10)...

Vector has  4 elements
Vector elements are:
-  20
-  9
-  10
-  12
```

В этом примере мы ищем элемент со значением 10. Если он найден, перед ним вставляется элемент со значением 9.

19.1.8 Удаление элементов

Вы можете удалить элементы из вектора, передав соответствующий индекс или курсор в процедуру удаления `Delete`. Если мы объединим это с функциями `Find_Index` и `Find` из предыдущего раздела, мы сможем написать программу, которая ищет определенный элемент и удаляет его, если он найден:

Listing 12: show_remove_vector_element.adb

```
with Ada.Containers.Vectors;

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Remove_Vector_Element is
   package Integer_Vectors is new
     Ada.Containers.Vectors
       (Index_Type   => Natural,
        Element_Type => Integer);

   use Integer_Vectors;

   V : Vector := 20 & 10 & 0 & 13 & 10 & 13;
   Idx : Extended_Index;
   C   : Cursor;
begin
```

(continues on next page)

(continued from previous page)

```
   --  Use Find_Index to retrieve index of
   --  the element with value 10
   Idx := V.Find_Index (10);

   --  Checking whether index is valid
   if Idx /= No_Index then
      --  Removing element using V.Delete
      V.Delete (Idx);
   end if;

   --  Use Find to retrieve cursor for
   --  the element with value 13
   C := V.Find (13);

   --  Check whether index is valid
   if C /= No_Element then
      --  Remove element using V.Delete
      V.Delete (C);
   end if;

end Show_Remove_Vector_Element;
```

Build output

```
show_remove_vector_element.adb:3:09: warning: no entities of "Ada.Text_IO" are␣
 ↪referenced [-gnatwu]
show_remove_vector_element.adb:3:19: warning: use clause for package "Text_IO" has␣
 ↪no effect [-gnatwu]
```

Мы можем расширить этот подход, чтобы удалить все элементы, соответствующие определенному значению. Нам просто нужно продолжать поиск элемента в цикле, пока мы не получим недопустимый индекс или курсор. Например:

Listing 13: show_remove_vector_elements.adb

```
with Ada.Containers; use Ada.Containers;
with Ada.Containers.Vectors;

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Remove_Vector_Elements is

   package Integer_Vectors is new
     Ada.Containers.Vectors
       (Index_Type   => Natural,
        Element_Type => Integer);

   use Integer_Vectors;

   procedure Show_Elements (V : Vector) is
   begin
      New_Line;
      Put_Line ("Vector has "
                & Count_Type'Image (V.Length)
                & " elements");

      if not V.Is_Empty then
         Put_Line ("Vector elements are: ");
         for E of V loop
            Put_Line ("- " & Integer'Image (E));
         end loop;
```

(continues on next page)

(continued from previous page)

```
      end if;
   end Show_Elements;

   V : Vector := 20 & 10 & 0 & 13 & 10 & 14 & 13;
begin
   Show_Elements (V);

   --
   --  Remove elements using an index
   --
   declare
      E : constant Integer := 10;
      I : Extended_Index;
   begin
      New_Line;
      Put_Line ("Removing all elements with value of "
                & Integer'Image (E) & "...");
      loop
         I := V.Find_Index (E);
         exit when I = No_Index;
         V.Delete (I);
      end loop;
   end;

   --
   --  Remove elements using a cursor
   --
   declare
      E : constant Integer := 13;
      C : Cursor;
   begin
      New_Line;
      Put_Line ("Removing all elements with value of "
                & Integer'Image (E) & "...");
      loop
         C := V.Find (E);
         exit when C = No_Element;
         V.Delete (C);
      end loop;
   end;

   Show_Elements (V);
end Show_Remove_Vector_Elements;
```

Runtime output

```
Vector has  7 elements
Vector elements are:
-  20
-  10
-  0
-  13
-  10
-  14
-  13

Removing all elements with value of  10...

Removing all elements with value of  13...
```

(continues on next page)

(continued from previous page)

```
Vector has  3 elements
Vector elements are:
-  20
-  0
-  14
```

В этом примере мы удаляем из вектора все элементы со значением 10, получая их индекс. Точно так же мы удаляем все элементы со значением 13 использую курсор.

19.1.9 Другие операции

Мы видели некоторые операции с элементами вектора. Здесь мы продемонстрируем операции с вектором в целом. Наиболее заметным является объединение нескольких векторов, но мы также увидим такие операции с векторами, как сортировка и слияния отсортированных массивов, которые рассматривают вектор, как последовательность элементов, при этом учитывают отношения элементов друг с другом.

Мы выполняем конкатенацию векторов с помощью оператора `&` для векторов. Рассмотрим два вектора `V1` и `V2`. Мы можем объединить их, выполнив `V := V1 & V2`. Результирующий вектор содержится в `V`.

Настраиваемый пакет `Generic_Sorting` является дочерним пакетом `Ada.Containers.Vectors`. Он содержит операции сортировки и объединения. Поскольку это настраиваемый пакет, вы не можете использовать его непосредственно, но должны сначала настроить его. Чтобы использовать эти операции с вектором целочисленных значений (`Integer_Vectors` в нашем примере), вам необходимо настроить пакет дочерний к `Integer_Vectors`. Следующий пример поясняет, как это сделать.

После настройки `Generic_Sorting` мы делаем все операции доступными с помощью спецификатора **`use`**. Затем мы можем вызвать `Sort`, чтобы отсортировать вектор, и `Merge`, чтобы объединить один вектор с другим.

В следующем примере представлен код, который работает тремя векторами (`V1`, `V2`, `V3`) и использует операций конкатенации, сортировки и слияния:

Listing 14: show_vector_ops.adb

```
1  with Ada.Containers; use Ada.Containers;
2  with Ada.Containers.Vectors;
3
4  with Ada.Text_IO; use Ada.Text_IO;
5
6  procedure Show_Vector_Ops is
7
8     package Integer_Vectors is new
9       Ada.Containers.Vectors
10        (Index_Type   => Natural,
11         Element_Type => Integer);
12
13    package Integer_Vectors_Sorting is new Integer_Vectors.Generic_Sorting;
14
15    use Integer_Vectors;
16    use Integer_Vectors_Sorting;
17
18    procedure Show_Elements (V : Vector) is
19    begin
20       New_Line;
21       Put_Line ("Vector has "
22                 & Count_Type'Image (V.Length)
```

(continues on next page)

(continued from previous page)

```
                & " elements");

      if not V.Is_Empty then
         Put_Line ("Vector elements are: ");
         for E of V loop
            Put_Line ("- " & Integer'Image (E));
         end loop;
      end if;
   end Show_Elements;

   V, V1, V2, V3 : Vector;
begin
   V1 := 10 & 12 & 18;
   V2 := 11 & 13 & 19;
   V3 := 15 & 19;

   New_Line;
   Put_Line ("---- V1 ----");
   Show_Elements (V1);

   New_Line;
   Put_Line ("---- V2 ----");
   Show_Elements (V2);

   New_Line;
   Put_Line ("---- V3 ----");
   Show_Elements (V3);

   New_Line;
   Put_Line ("Concatenating V1, V2 and V3 into V:");

   V := V1 & V2 & V3;

   Show_Elements (V);

   New_Line;
   Put_Line ("Sorting V:");

   Sort (V);

   Show_Elements (V);

   New_Line;
   Put_Line ("Merging V2 into V1:");

   Merge (V1, V2);

   Show_Elements (V1);

end Show_Vector_Ops;
```

Runtime output

```
---- V1 ----

Vector has  3 elements
Vector elements are:
-  10
-  12
-  18
```

(continues on next page)

(continued from previous page)

```
---- V2 ----

Vector has  3 elements
Vector elements are:
-  11
-  13
-  19

---- V3 ----

Vector has  2 elements
Vector elements are:
-  15
-  19

Concatenating V1, V2 and V3 into V:

Vector has  8 elements
Vector elements are:
-  10
-  12
-  18
-  11
-  13
-  19
-  15
-  19

Sorting V:

Vector has  8 elements
Vector elements are:
-  10
-  11
-  12
-  13
-  15
-  18
-  19
-  19

Merging V2 into V1:

Vector has  6 elements
Vector elements are:
-  10
-  11
-  12
-  13
-  18
-  19
```

Справочное руководство требует, чтобы худшей сложностью вызова для сортировки `Sort` было O(Nsup:*2*) и средняя сложность должна быть лучше, чем O(Nsup:*2*).

19.2 Множества

Множества это другим вид контейнеров. В то время как векторы позволяют вставлять дублирующиеся элементы, множества гарантируют, что дублированных элементов не будет.

В следующих разделах мы рассмотрим операции, которые вы можете выполнять с множествами. Однако, поскольку многие операции с векторами аналогичны тем, которые используются для множеств, мы рассмотрим их здесь лишь кратко. Пожалуйста, обратитесь к разделу о векторах для более подробного обсуждения.

19.2.1 Инициализация и итерация

Чтобы инициализировать множество, вы можете вызвать процедуру `Insert`. Делая это, вы должны убедиться, что не вставляются повторяющиеся элементы: если вы попытаетесь вставить дубликат, вы получите исключение. Если вы не уверены, что нет дубликатов, вы можете воспользоваться другими вариантами:

- версия `Insert`, которая возвращает логическое значение, указывающее, была ли вставка успешной;
- процедура `Include`, которая молча игнорирует любую попытку вставить повторяющийся элемент.

Чтобы перебрать множество, вы можете использовать цикл `for E of S`, аналогично векторам. Вы получаете ссылку на элемент в множестве.

Посмотрим на пример:

Listing 15: show_set_init.adb

```
with Ada.Containers; use Ada.Containers;
with Ada.Containers.Ordered_Sets;

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Set_Init is

   package Integer_Sets is new
     Ada.Containers.Ordered_Sets
       (Element_Type => Integer);

   use Integer_Sets;

   S : Set;
   --  Same as:  S : Integer_Sets.Set;
   C : Cursor;
   Ins : Boolean;
begin
   S.Insert (20);
   S.Insert (10);
   S.Insert (0);
   S.Insert (13);

   --  Calling S.Insert(0) now would raise
   --  Constraint_Error because this element
   --  is already in the set. We instead call a
   --  version of Insert that doesn't raise an
   --  exception but instead returns a Boolean
   --  indicating the status

```

(continues on next page)

(continued from previous page)

```
   S.Insert (0, C, Ins);
   if not Ins then
      Put_Line ("Inserting 0 into set was not successful");
   end if;

   --  We can also call S.Include instead
   --  If the element is already present,
   --  the set remains unchanged
   S.Include (0);
   S.Include (13);
   S.Include (14);

   Put_Line ("Set has "
             & Count_Type'Image (S.Length)
             & " elements");

   --
   --  Iterate over set using for .. of loop
   --
   Put_Line ("Elements:");
   for E of S loop
       Put_Line ("- " & Integer'Image (E));
   end loop;
end Show_Set_Init;
```

Runtime output

```
Inserting 0 into set was not successful
Set has  5 elements
Elements:
-  0
-  10
-  13
-  14
-  20
```

19.2.2 Операции с элементами

В этом разделе мы кратко рассмотрим следующие операции над множествами:

- `Delete` и `Exclude`, чтобы удалить элементы;
- `Contains` и `Find`, чтобы проверить наличие элементов.

Чтобы удалить элементы, вы вызываете процедуру `Delete`. Однако, аналогично описанной выше процедуре `Insert`, `Delete` возбуждает исключение, если элемент, подлежащий удалению, отсутствует в множестве. Если элемент может отсутствовать в момент удаления и вам не нужна проверка, то вы можете вызвать процедуру `Exclude`, которая молча игнорирует любую попытку удалить несуществующий элемент.

Функция `Contains` возвращает логическое значение Boolean, указывающее, содержится ли значение в множестве. `Find` также ищет элемент в множестве, но возвращает курсор на элемент или `No_Element`, если элемент не существует. Вы можете использовать любую из этих функций для проверки наличия элементов в множестве.

Давайте рассмотрим пример, в котором используются эти операции:

Listing 16: show_set_element_ops.adb

```
with Ada.Containers; use Ada.Containers;
with Ada.Containers.Ordered_Sets;

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Set_Element_Ops is

   package Integer_Sets is new
     Ada.Containers.Ordered_Sets
       (Element_Type => Integer);

   use Integer_Sets;

   procedure Show_Elements (S : Set) is
   begin
      New_Line;
      Put_Line ("Set has "
                & Count_Type'Image (S.Length)
                & " elements");
      Put_Line ("Elements:");
      for E of S loop
         Put_Line ("- " & Integer'Image (E));
      end loop;
   end Show_Elements;

   S : Set;
begin
   S.Insert (20);
   S.Insert (10);
   S.Insert (0);
   S.Insert (13);

   S.Delete (13);

   --  Calling S.Delete (13) again raises
   --  Constraint_Error because the element
   --  is no longer present in the set, so
   --  it can't be deleted. We can call
   --  V.Exclude instead:
   S.Exclude (13);

   if S.Contains (20) then
      Put_Line ("Found element 20 in set");
   end if;

   --  Alternatively, we could use S.Find
   --  instead of S.Contains
   if S.Find (0) /= No_Element then
      Put_Line ("Found element 0 in set");
   end if;

   Show_Elements (S);
end Show_Set_Element_Ops;
```

Runtime output

```
Found element 20 in set
Found element 0 in set

Set has  3 elements
```

(continues on next page)

(continued from previous page)

```
Elements:
-  0
-  10
-  20
```

В дополнение к упорядоченным множествам, используемым в приведенных выше примерах, стандартная библиотека также предлагает хешированные множества. Справочное руководство требует следующей средней сложности каждой операции:

Операции	`Ordered_Sets`	`Hashed_Sets`
• Insert • Include • Replace • Delete • Exclude • Find	O((log N)^2) или лучше	O(log N)
Подпрограмма с использованием курсора	O(1)	O(1)

19.2.3 Другие операции

Предыдущие разделы в основном касались операций с отдельными элементами множествоа. Но Ада также предоставляет типичные операции над множествами: объединение, пересечение, разность и симметричная разность. В отличие от некоторых векторных операций, которые мы видели раньше (например, слияния - `Merge`), здесь вы можете использовать общепринятые операторы, такие как `-`. В следующей таблице перечислены операции и связанный с ними оператор:

Операции над множеством	Оператор
Объединение	**`or`**
Пересечение	**`and`**
Разность	`-`
Симметричная разность	**`xor`**

В следующем примере используются эти операторы:

Listing 17: show_set_ops.adb

```
1  with Ada.Containers; use Ada.Containers;
2  with Ada.Containers.Ordered_Sets;
3
4  with Ada.Text_IO; use Ada.Text_IO;
5
6  procedure Show_Set_Ops is
7
8     package Integer_Sets is new
9       Ada.Containers.Ordered_Sets
10        (Element_Type => Integer);
11
12    use Integer_Sets;
13
14    procedure Show_Elements (S : Set) is
15    begin
16       Put_Line ("Elements:");
```

(continues on next page)

(continued from previous page)

```
      for E of S loop
         Put_Line ("- " & Integer'Image (E));
      end loop;
   end Show_Elements;

   procedure Show_Op (S       : Set;
                      Op_Name : String) is
   begin
      New_Line;
      Put_Line (Op_Name
                & "(set #1, set #2) has "
                & Count_Type'Image (S.Length)
                & " elements");
   end Show_Op;

   S1, S2, S3 : Set;
begin
   S1.Insert (0);
   S1.Insert (10);
   S1.Insert (13);

   S2.Insert (0);
   S2.Insert (10);
   S2.Insert (14);

   S3.Insert (0);
   S3.Insert (10);

   New_Line;
   Put_Line ("---- Set #1 ----");
   Show_Elements (S1);

   New_Line;
   Put_Line ("---- Set #2 ----");
   Show_Elements (S2);

   New_Line;
   Put_Line ("---- Set #3 ----");
   Show_Elements (S3);

   New_Line;
   if S3.Is_Subset (S1) then
      Put_Line ("S3 is a subset of S1");
   else
      Put_Line ("S3 is not a subset of S1");
   end if;

   S3 := S1 and S2;
   Show_Op (S3, "Intersection");
   Show_Elements (S3);

   S3 := S1 or S2;
   Show_Op (S3, "Union");
   Show_Elements (S3);

   S3 := S1 - S2;
   Show_Op (S3, "Difference");
   Show_Elements (S3);

   S3 := S1 xor S2;
   Show_Op (S3, "Symmetric difference");
```

(continues on next page)

(continued from previous page)

```
78    Show_Elements (S3);
79
80 end Show_Set_Ops;
```

Runtime output

```
---- Set #1 ----
Elements:
-  0
-  10
-  13

---- Set #2 ----
Elements:
-  0
-  10
-  14

---- Set #3 ----
Elements:
-  0
-  10

S3 is a subset of S1

Intersection(set #1, set #2) has  2 elements
Elements:
-  0
-  10

Union(set #1, set #2) has  4 elements
Elements:
-  0
-  10
-  13
-  14

Difference(set #1, set #2) has  1 elements
Elements:
-  13

Symmetric difference(set #1, set #2) has  2 elements
Elements:
-  13
-  14
```

19.3 Отображения для неопределенных типов

В предыдущих разделах были представлены контейнеры для элементов определенных типов. Хотя большинство примеров в этих разделах использовали целочисленный тип **Integer** как тип элемента контейнера, контейнеры также могут использоваться с неопределенными типами, примером которых является тип **String**. Однако неопределенные типы требуют другого вида контейнеров, разработанных специально для них.

Мы также изучим другой класс контейнеров: отображения. Они связывают ключ с определенным значением. Примером отображения является связь «один к одному» между

человеком и его возрастом. Если мы считаем имя человека ключевым, то значение - возраст человека.

19.3.1 Хэшированные отображения

Хэшированные отображения - это отображения, которые используют хэш ключа. Сам хэш вычисляется с помощью предоставленной вами функции.

На других языках

Хэшированные отображения похожи на словари в Python и хэши в Perl. Одно из основных отличий заключается в том, что эти скриптовые языки позволяют использовать разные типы для значений, содержащихся в одном отображении, в то время как в Аде, тип ключа и тип значение указываются в настройке пакета и остаются постоянными для этого конкретного отображения. У вас не может быть отображения, содержащего два элемента или два ключа разног типов. Если вы хотите использовать несколько типов, вы должны создать разные отображения для каждого и использовать только одино из них.

При создании настройке хешированного отображения `Ada.Containers.Indefinite_Hashed_Maps` мы указываем следующие элементы:

- `Key_Type`: тип ключа
- `Element_Type`: тип элемента
- `HashKey_Type`
- `Equivalent_Keys`: оператор равенства (например, `=`), который указывает, должны ли два ключа считаться равными.
  - Если тип, указанный в `Key_Type`, имеет стандартный оператор, вы можете использовать его. В примере мы так и делаем. Мы указываем этот оператор как значение `Equivalent_Keys`.

В следующем примере мы будем использовать строку в качестве типа ключа. Мы будем использовать функцию `Hash`, предоставляемую стандартной библиотекой для строк (в пакете `Ada.Strings`), и стандартный оператор равенства.

Вы добавляете элементы в хешированное отображение, вызывая `Insert`. Если элемент уже содержится в отображении `M`, вы можете получить к нему доступ непосредственно, используя его ключ. Например, вы можете изменить значение элемента, написав `M("My_Key") := 10`. Если ключ не найден, возбуждается исключение. Чтобы проверить, доступен ли ключ, используйте функцию `Contains` (как мы видели выше в разделе о множествоах).

Посмотрим на пример:

Listing 18: show_hashed_map.adb

```
with Ada.Containers.Indefinite_Hashed_Maps;
with Ada.Strings.Hash;

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Hashed_Map is

   package Integer_Hashed_Maps is new
     Ada.Containers.Indefinite_Hashed_Maps
       (Key_Type        => String,
        Element_Type    => Integer,
        Hash            => Ada.Strings.Hash,
```

(continues on next page)

(continued from previous page)

```
        Equivalent_Keys => "=");

   use Integer_Hashed_Maps;

   M : Map;
   --  Same as:
   --
   --  M : Integer_Hashed_Maps.Map;
begin
   M.Include ("Alice", 24);
   M.Include ("John",  40);
   M.Include ("Bob",   28);

   if M.Contains ("Alice") then
      Put_Line ("Alice's age is "
                & Integer'Image (M ("Alice")));
   end if;

   --  Update Alice's age
   --  Key must already exist in M.
   --  Otherwise an exception is raised.
   M ("Alice") := 25;

   New_Line; Put_Line ("Name & Age:");
   for C in M.Iterate loop
      Put_Line (Key (C) & ": "
                & Integer'Image (M (C)));
   end loop;

end Show_Hashed_Map;
```

Runtime output

```
Alice's age is  24

Name & Age:
John:  40
Bob:  28
Alice:  25
```

19.3.2 Упорядоченные отображения

Упорядоченные отображения имеют много общих черт с хэшированными отображениями. Основными отличиями являются:

- Хэш-функция не нужна. Вместо этого вы должны предоставить функцию сравнения (< operator), которую упорядоченное отображение будет использовать для сравнения элементов и обеспечения быстрого доступа (сложность O(log N)), используя двоичный поиск.
  - Если тип, указанный в `Key_Type`, имеет стандартный оператор <, вы можете использовать его аналогично тому, как мы это делали для `Equivalent_Keys` выше для хэшированных отображений.

Давайте посмотрим на пример:

Listing 19: show_ordered_map.adb

```
with Ada.Containers.Indefinite_Ordered_Maps;

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Ordered_Map is

   package Integer_Ordered_Maps is new
     Ada.Containers.Indefinite_Ordered_Maps
       (Key_Type        => String,
        Element_Type    => Integer);

   use Integer_Ordered_Maps;

   M : Map;
begin
   M.Include ("Alice", 24);
   M.Include ("John",  40);
   M.Include ("Bob",   28);

   if M.Contains ("Alice") then
      Put_Line ("Alice's age is "
                & Integer'Image (M ("Alice")));
   end if;

   --  Update Alice's age
   --  Key must already exist in M
   M ("Alice") := 25;

   New_Line; Put_Line ("Name & Age:");
   for C in M.Iterate loop
      Put_Line (Key (C) & ": "
                & Integer'Image (M (C)));
   end loop;

end Show_Ordered_Map;
```

Runtime output

```
Alice's age is  24

Name & Age:
Alice:  25
Bob:  28
John:  40
```

Вы можете увидеть большое сходство между примерами, приведенными выше, и примерами из предыдущего раздела. Фактически, поскольку оба типа отображений имеют много общих операций, нам не нужно было вносить существенные изменения, когда мы изменили наш пример, чтобы использовать упорядоченные отображения вместо хешированных. Основное различие видно, когда мы запускаем примеры: вывод для хешированных отображений обычно неупорядочен, но вывод для упорядоченных отображений всегда упорядочен, как следует из его имени.

19.3.3 Сложность

Хэшированные отображения, как правило, являются самой быстрой структурой данных, доступной в Аде, если необходимо связать неоднородные ключи со значениями и быстро находить их. В большинстве случаев они немного быстрее упорядоченных отображений. Так что, если вам не важен порядок, используйте хэшированные отображения.

Справочное руководство требует следующей средней сложности операций:

Операции	`Ordered_Maps`	`Hashed_Maps`
• Insert • Include • Replace • Delete • Exclude • Find	O((log N)^2) or better	O(log N)
Подпрограмма с использованием курсора	O(1)	O(1)

CHAPTER

TWENTY

СТАНДАРТНАЯ БИБЛИОТЕКА: ДАТА И ВРЕМЯ

Стандартная библиотека поддерживает обработку дат и времени с использованием двух подходов:

- *Календарный* подход, подходящий для обработки дат и времени в целом;
- Подход *реального времени*, который лучше подходит для приложений реального времени, требующих повышенной точности, например, благодаря доступу к абсолютным часам и обработке интервалов времени. Следует отметить, что этот подход поддерживает только время, но не даты.

Эти два подхода представлены в следующих разделах.

20.1 Обработка даты и времени

Пакет `Ada.Calendar` поддерживает обработку дат и времени. Рассмотрим простой пример:

Listing 1: display_current_time.adb

```
with Ada.Calendar;            use Ada.Calendar;
with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;
with Ada.Text_IO;             use Ada.Text_IO;

procedure Display_Current_Time is
   Now : Time := Clock;
begin
   Put_Line ("Current time: " & Image (Now));
end Display_Current_Time;
```

Build output

```
display_current_time.adb:6:04: warning: "Now" is not modified, could be declared␣
↪constant [-gnatwk]
```

Runtime output

```
Current time: 2022-08-22 21:14:24
```

В этом примере отображаются текущая дата и время, которые извлекаются при вызове функции `Clock`. Мы вызываем функцию `Image` из пакета `Ada.Calendar.Formatting`, чтобы получить строку (`String`) для текущей даты и времени. Вместо этого мы могли бы получить каждую компоненту с помощью функции `Split`. Например:

Listing 2: display_current_year.adb

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

procedure Display_Current_Year is
   Now         : Time := Clock;

   Now_Year    : Year_Number;
   Now_Month   : Month_Number;
   Now_Day     : Day_Number;
   Now_Seconds : Day_Duration;
begin
   Split (Now,
          Now_Year,
          Now_Month,
          Now_Day,
          Now_Seconds);

   Put_Line ("Current year  is: "
             & Year_Number'Image (Now_Year));
   Put_Line ("Current month is: "
             & Month_Number'Image (Now_Month));
   Put_Line ("Current day   is: "
             & Day_Number'Image (Now_Day));
end Display_Current_Year;
```

Build output

```
display_current_year.adb:5:04: warning: "Now" is not modified, could be declared␣
↪constant [-gnatwk]
```

Runtime output

```
Current year  is:  2022
Current month is:  8
Current day   is:  22
```

Здесь мы получаем каждый элемент и отображаем его отдельно.

20.1.1 Задержка с использованием даты

Вы можете приостановить приложение, чтобы оно перезапустилось в определенную дату и время. Мы видели нечто подобное в главе о задачах. Вы делаете это с помощью оператора **delay until**. Например:

Listing 3: display_delay_next_specific_time.adb

```
with Ada.Calendar;            use Ada.Calendar;
with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;
with Ada.Calendar.Time_Zones; use Ada.Calendar.Time_Zones;
with Ada.Text_IO;             use Ada.Text_IO;

procedure Display_Delay_Next_Specific_Time is
   TZ   : Time_Offset := UTC_Time_Offset;
   Next : Time        :=
     Ada.Calendar.Formatting.Time_Of
       (Year        => 2018,
        Month       => 5,
        Day         => 1,
```

(continues on next page)

(continued from previous page)

```
        Hour        => 15,
        Minute      => 0,
        Second      => 0,
        Sub_Second  => 0.0,
        Leap_Second => False,
        Time_Zone   => TZ);

   --  Next = 2018-05-01 15:00:00.00
   --         (local time-zone)
begin
   Put_Line ("Let's wait until...");
   Put_Line (Image (Next, True, TZ));

   delay until Next;

   Put_Line ("Enough waiting!");
end Display_Delay_Next_Specific_Time;
```

Build output

```
display_delay_next_specific_time.adb:7:04: warning: "TZ" is not modified, could be␣
 ↪declared constant [-gnatwk]
display_delay_next_specific_time.adb:8:04: warning: "Next" is not modified, could␣
 ↪be declared constant [-gnatwk]
```

Runtime output

```
Let's wait until...
2018-05-01 15:00:00.00
Enough waiting!
```

В этом примере мы указываем дату и время, инициализируя `Next` с помощью вызова `Time_Of`, функции, принимающей различные компоненты даты (год, месяц и т. д.) и возвращающей элемент типа `Time`. Поскольку указанная дата находится в прошлом, задержка **`delay until`** не даст заметного эффекта. Если мы укажем дату в будущем, программа будет ждать наступления этой конкретной даты и времени.

Здесь мы переводим время в местный часовой пояс. Если мы не указываем часовой пояс, по умолчанию используется всемирное координированное время (*Coordinated Universal Time* сокращенно UTC). Получив смещение времени к UTC с помощью вызова `UTC_Time_Offset` из пакета `Ada.Calendar.Time_Zones`, мы можем инициализировать `TZ` и использовать его при вызове `Time_Of`. Это все, что нам нужно сделать, чтобы информация, предоставляемая `Time_Of`, относилась к местному часовому поясу.

Мы могли бы добиться аналогичного результата, инициализировав `Next` с помощью значения типа **`String`**. Мы можем сделать это использовав вызов `Value` из пакета `Ada.Calendar.Formatting`. Вот модифицированный код:

Listing 4: display_delay_next_specific_time.adb

```
with Ada.Calendar;            use Ada.Calendar;
with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;
with Ada.Calendar.Time_Zones; use Ada.Calendar.Time_Zones;
with Ada.Text_IO;             use Ada.Text_IO;

procedure Display_Delay_Next_Specific_Time is
   TZ   : Time_Offset := UTC_Time_Offset;
   Next : Time        :=
     Ada.Calendar.Formatting.Value
       ("2018-05-01 15:00:00.00", TZ);

```

(continues on next page)

(continued from previous page)

```
   --  Next = 2018-05-01 15:00:00.00
   --         (local time-zone)
begin
   Put_Line ("Let's wait until...");
   Put_Line (Image (Next, True, TZ));

   delay until Next;

   Put_Line ("Enough waiting!");
end Display_Delay_Next_Specific_Time;
```

Build output

```
display_delay_next_specific_time.adb:7:04: warning: "TZ" is not modified, could be␣
 ↪declared constant [-gnatwk]
display_delay_next_specific_time.adb:8:04: warning: "Next" is not modified, could␣
 ↪be declared constant [-gnatwk]
```

Runtime output

```
Let's wait until...
2018-05-01 15:00:00.00
Enough waiting!
```

В этом примере мы снова используем TZ в вызове `Value`, чтобы привести время ввода в соответствие с текущим часовым поясом.

В приведенных выше примерах мы приостанавливались до определенной даты и времени. Как мы видели в главе о задачах, мы могли бы вместо этого указать задержку относительно текущего времени. Например, мы можем задержать на 5 секунд, используя текущее время:

Listing 5: display_delay_next.adb

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

procedure Display_Delay_Next is
   D    : Duration := 5.0;
   --                 ^ seconds
   Now  : Time     := Clock;
   Next : Time     := Now + D;
   --                       ^ use duration to
   --                         specify next point
   --                         in time
begin
   Put_Line ("Let's wait "
             & Duration'Image (D) & " seconds...");
   delay until Next;
   Put_Line ("Enough waiting!");
end Display_Delay_Next;
```

Build output

```
display_delay_next.adb:5:04: warning: "D" is not modified, could be declared␣
 ↪constant [-gnatwk]
display_delay_next.adb:7:04: warning: "Now" is not modified, could be declared␣
 ↪constant [-gnatwk]
display_delay_next.adb:8:04: warning: "Next" is not modified, could be declared␣
 ↪constant [-gnatwk]
```

Runtime output

```
Let's wait  5.000000000 seconds...
Enough waiting!
```

Здесь мы указываем продолжительность 5 секунд в D, добавляем ее к текущему времени из Now и сохраняем сумму в Next. Затем мы используем его в операторе **delay until**.

20.2 Режим реального времени

В дополнение к Ada.Calendar стандартная библиотека также поддерживает операции со временем для приложений реального времени. Они включены в пакет Ada.Real_Time. Этот пакет также включает тип Time. Однако в пакете Ada.Real_Time тип Time используется для представления абсолютных часов и обработки промежутков времени. Это контрастирует с Ada.Calendar, который использует тип Time для представления даты и времени.

В предыдущем разделе мы использовали тип Time из Ada.Calendar и оператор **delay until**, чтобы отложить приложение на 5 секунд. Вместо этого мы могли бы использовать пакет Ada.Real_Time. Давайте изменим этот пример:

Listing 6: display_delay_next_real_time.adb

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

procedure Display_Delay_Next_Real_Time is
   D     : Time_Span := Seconds (5);
   Next  : Time      := Clock + D;
begin
   Put_Line ("Let's wait "
             & Duration'Image (To_Duration (D))
             & " seconds...");
   delay until Next;
   Put_Line ("Enough waiting!");
end Display_Delay_Next_Real_Time;
```

Build output

```
display_delay_next_real_time.adb:5:04: warning: "D" is not modified, could be␣
 ↪declared constant [-gnatwk]
display_delay_next_real_time.adb:6:04: warning: "Next" is not modified, could be␣
 ↪declared constant [-gnatwk]
```

Runtime output

```
Let's wait  5.000000000 seconds...
Enough waiting!
```

Основное отличие состоит в том, что D теперь является переменной типа Time_Span, определенной в пакете Ada.Real_Time. Мы вызываем функцию Seconds для инициализации D, но мы могли бы получить более тончное значение, вызвав вместо этого Nanoseconds. Кроме того, нам нужно сначала преобразовать D в тип **Duration** с помощью функции To_Duration, прежде чем мы сможем его напечатать.

20.2.1 Анализ производительности

Одним из интересных приложений, использующих пакет `Ada.Real_Time`, является анализ производительности. Мы уже использовали этот пакет в предыдущем разделе при обсуждении задач. Давайте рассмотрим пример анализа производительности:

Listing 7: display_benchmarking.adb

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

procedure Display_Benchmarking is

   procedure Computational_Intensive_App is
   begin
      delay 5.0;
   end Computational_Intensive_App;

   Start_Time, Stop_Time : Time;
   Elapsed_Time          : Time_Span;

begin
   Start_Time := Clock;

   Computational_Intensive_App;

   Stop_Time    := Clock;
   Elapsed_Time := Stop_Time - Start_Time;

   Put_Line ("Elapsed time: "
             & Duration'Image (To_Duration (Elapsed_Time))
             & " seconds");
end Display_Benchmarking;
```

Runtime output

```
Elapsed time:  5.007097508 seconds
```

В этом примере определяется фиктивное приложение `Computational_Intensive_App`, реализованное с использованием простого оператора задержки **`delay`**. Мы инициализируем `Start_Time` и `Stop_Time` по текущим на тот момент часам и вычисляем прошедшее время. Запустив эту программу, мы видим, что время составляет примерно 5 секунд, что соответствует реботе оператора задержки **`delay`**.

Аналогичное приложение - это анализ затраченного процессорного времени. Мы можем реализовать это с помощью пакета `Execution_Time`. Давайте изменим предыдущий пример, чтобы измерить процессорное время:

Listing 8: display_benchmarking_cpu_time.adb

```
with Ada.Text_IO;        use Ada.Text_IO;
with Ada.Real_Time;      use Ada.Real_Time;
with Ada.Execution_Time; use Ada.Execution_Time;

procedure Display_Benchmarking_CPU_Time is

   procedure Computational_Intensive_App is
   begin
      delay 5.0;
   end Computational_Intensive_App;

   Start_Time, Stop_Time : CPU_Time;
```

(continues on next page)

(continued from previous page)

```
   Elapsed_Time          : Time_Span;

begin
   Start_Time := Clock;

   Computational_Intensive_App;

   Stop_Time    := Clock;
   Elapsed_Time := Stop_Time - Start_Time;

   Put_Line ("CPU time: "
             & Duration'Image (To_Duration (Elapsed_Time))
             & " seconds");
end Display_Benchmarking_CPU_Time;
```

Runtime output

```
CPU time:  0.000125123 seconds
```

В этом примере `Start_Time` и `Stop_Time` имеют тип `CPU_Time` вместо `Time`. Однако мы по-прежнему вызываем функцию `Clock` для инициализации обеих переменных и вычисления прошедшего времени так же, как и раньше. Запустив эту программу, мы видим, что время процессора значительно ниже, чем те 5 секунд, которые мы видели раньше. Это связано с тем, что оператор задержки **`delay`** не требует много времени процессора. Результаты будут другими, если мы изменим реализацию `Computational_Intensive_App` для использования математических функций в длинном цикле. Например:

Listing 9: display_benchmarking_math.adb

```
with Ada.Text_IO;         use Ada.Text_IO;
with Ada.Real_Time;       use Ada.Real_Time;
with Ada.Execution_Time; use Ada.Execution_Time;

with Ada.Numerics.Generic_Elementary_Functions;

procedure Display_Benchmarking_Math is

   procedure Computational_Intensive_App is
      package Funcs is new Ada.Numerics.Generic_Elementary_Functions
        (Float_Type => Long_Long_Float);
      use Funcs;

      X : Long_Long_Float;
   begin
      for I in 0 .. 1_000_000 loop
         X := Tan (Arctan
                   (Tan (Arctan
                      (Tan (Arctan
                         (Tan (Arctan
                            (Tan (Arctan
                               (Tan (Arctan
                                  (0.577))))))))))));
      end loop;
   end Computational_Intensive_App;

   procedure Benchm_Elapsed_Time is
      Start_Time, Stop_Time : Time;
      Elapsed_Time          : Time_Span;

   begin
```

(continues on next page)

(continued from previous page)

```
      Start_Time := Clock;

      Computational_Intensive_App;

      Stop_Time    := Clock;
      Elapsed_Time := Stop_Time - Start_Time;

      Put_Line ("Elapsed time: "
                & Duration'Image (To_Duration (Elapsed_Time))
                & " seconds");
   end Benchm_Elapsed_Time;

   procedure Benchm_CPU_Time is
      Start_Time, Stop_Time : CPU_Time;
      Elapsed_Time          : Time_Span;

   begin
      Start_Time := Clock;

      Computational_Intensive_App;

      Stop_Time    := Clock;
      Elapsed_Time := Stop_Time - Start_Time;

      Put_Line ("CPU time: "
                & Duration'Image (To_Duration (Elapsed_Time))
                & " seconds");
   end Benchm_CPU_Time;
begin
   Benchm_Elapsed_Time;
   Benchm_CPU_Time;
end Display_Benchmarking_Math;
```

Build output

```
display_benchmarking_math.adb:14:07: warning: variable "X" is assigned but never␣
 ↪read [-gnatwm]
```

Runtime output

```
Elapsed time:  0.871429605 seconds
CPU time:  0.876544727 seconds
```

Теперь, когда наша фиктивная `Computational_Intensive_App` включает математические операции, требующие значительного времени ЦПУ, измеренное затраченное время и время ЦПУ намного ближе друг к другу, чем раньше.

CHAPTER

TWENTYONE

СТАНДАРТНАЯ БИБЛИОТЕКА: СТРОКИ

В предыдущих главах мы видели примеры исходного кода с использованием типа **String**, который является строковым типом фиксированной длины - по сути, это массив символов. Во многих случаях этого типа данных достаточно для работы с текстовой информацией. Однако бывают ситуации, когда требуется более сложная обработка текста. Ада предлагает альтернативные подходы для этих случаев:

- *Ограниченные строки*: аналогично строкам фиксированной длины, ограниченные строки имеют максимальную длину, которая устанавливается при их создании. Однако ограниченные строки не являются массивами символов. В любой момент они могут содержать строку различной длины - при условии, что эта длина меньше или равна максимальной длине.
- *Неограниченные строки*: подобно ограниченным строкам, неограниченные строки могут содержать строки различной длины. Однако, помимо этого, у них нет максимальной длины. В этом смысле они очень гибкие.

В следующих разделах представлен обзор различных типов строк и общих операций для типов строк.

21.1 Операции со строками

Операции со стандартными строками (фиксированной длины) доступны в пакете `Ada.Strings.Fixed`. Как упоминалось ранее, стандартные строки представляют собой массивы элементов типа **Character** с *фиксированной длиной*. Вот почему этот дочерний пакет называется фиксированным (`Fixed`).

Одна из самых простых операций - это подсчет количества подстрок, доступных в строке (**Count**), и поиск их соответствующих индексов (`Index`). Давайте посмотрим на пример:

Listing 1: show_find_substring.adb

```
with Ada.Strings.Fixed; use Ada.Strings.Fixed;
with Ada.Text_IO;       use Ada.Text_IO;

procedure Show_Find_Substring is

   S   : String := "Hello" & 3 * " World";
   P   : constant String := "World";
   Idx : Natural;
   Cnt : Natural;
begin
   Cnt := Ada.Strings.Fixed.Count
     (Source  => S,
      Pattern => P);

   Put_Line ("String: " & S);
```

(continues on next page)

(continued from previous page)

```
   Put_Line ("Count for '" & P & "': "
             & Natural'Image (Cnt));

   Idx := 0;
   for I in 1 .. Cnt loop
      Idx := Index
        (Source  => S,
         Pattern => P,
         From    => Idx + 1);

      Put_Line ("Found instance of '"
                & P & "' at position: "
                & Natural'Image (Idx));
   end loop;

end Show_Find_Substring;
```

Build output

```
show_find_substring.adb:6:04: warning: "S" is not modified, could be declared␣
 ↪constant [-gnatwk]
```

Runtime output

```
String: Hello World World World
Count for 'World':  3
Found instance of 'World' at position:  7
Found instance of 'World' at position:  13
Found instance of 'World' at position:  19
```

Мы инициализируем строку S умножением. Запись `"Hello" & 3 * " World"` создает строку `Hello World World World`. Затем мы вызываем функцию `Count`, чтобы получить количество экземпляров слова `World` в S .Затем мы вызываем функцию `Index` в цикле, чтобы найти индекс каждого экземпляра слова `World` в S.

В этом примере искались вхождения определенной подстроки. В следующем примере мы извлекаем все слова в строке. Мы делаем это с помощью `Find_Token`, определив пробелы в качестве разделителей. Например:

Listing 2: show_find_words.adb

```
with Ada.Strings;       use Ada.Strings;
with Ada.Strings.Fixed; use Ada.Strings.Fixed;
with Ada.Strings.Maps;  use Ada.Strings.Maps;
with Ada.Text_IO;       use Ada.Text_IO;

procedure Show_Find_Words is

   S   : String := "Hello" & 3 * " World";
   F   : Positive;
   L   : Natural;
   I   : Natural := 1;

   Whitespace : constant Character_Set :=
     To_Set (' ');
begin
   Put_Line ("String: " & S);
   Put_Line ("String length: "
             & Integer'Image (S'Length));

   while I in S'Range loop
```

(continues on next page)

(continued from previous page)

```
      Find_Token
        (Source  => S,
         Set     => Whitespace,
         From    => I,
         Test    => Outside,
         First   => F,
         Last    => L);

      exit when L = 0;

      Put_Line ("Found word instance at position "
                & Natural'Image (F)
                & ": '" & S (F .. L) & "'");
      --   & "-" & F'Img & "-" & L'Img

      I := L + 1;
   end loop;
end Show_Find_Words;
```

Build output

```
show_find_words.adb:8:04: warning: "S" is not modified, could be declared constant␣
↪[-gnatwk]
```

Runtime output

```
String: Hello World World World
String length:  23
Found word instance at position  1: 'Hello'
Found word instance at position  7: 'World'
Found word instance at position  13: 'World'
Found word instance at position  19: 'World'
```

Мы передаем множество символов, которые будут использоваться в качестве разделителей для процедуры `Find_Token`. Это множество является значением типа `Character_Set` из пакета `Ada.Strings.Maps`. Мы вызываем функцию `To_Set` (из того же пакета), чтобы инициализировать множество `Whitespace`, а затем вызываем `Find_Token`, чтобы перебрать все возможные индексы и найти начальный индекс каждого слова. Мы передаем `Outside` в параметр `Test` процедуры `Find_Token`, чтобы указать, что мы ищем индексы, которые находятся за пределами множества `Whitespace`, то есть фактические слова. `First` и `Last` параметры `Find_Token` являются выходными параметрами, которые указывают допустимый диапазон подстроки. Мы используем эту информацию для отображения строки (`S (F .. L)`).

Операции, которые мы рассмотрели до сих пор, читают строки, но не изменяют их. Далее мы обсудим операции, которые изменяют содержимое строк:

Operation	Description
Insert	Вставка подстроки в строку
Overwrite	Замена подстроки в строке
Delete	Удаление подстроки
Trim	Удаление пробелов из строки

Все эти операции доступны как в виде функций, так и в виде процедур. Функции создают новую строку, но процедуры выполняют операции по месту. Процедура вызовет исключение, если ограничения строки будут нарушены. Например, если у нас есть строка S, содержащая 10 символов, вставка в нее строки с двумя символами (например, `"!!"`) дает строку, содержащую 12 символов. Поскольку она имеет фиксированную длину, мы не можем увеличить его размер. Одно из возможных решений в этом случае - указать, что при вставке

подстроки следует применять усечение. Это сохраняет длину S. Давайте посмотрим на пример, в котором используются как функции, так и версии процедур `Insert`, `Overwrite` и `Delete`:

Listing 3: show_adapted_strings.adb

```
with Ada.Strings;       use Ada.Strings;
with Ada.Strings.Fixed; use Ada.Strings.Fixed;
with Ada.Text_IO;       use Ada.Text_IO;

procedure Show_Adapted_Strings is

   S   : String := "Hello World";
   P   : constant String := "World";
   N   : constant String := "Beautiful";

   procedure Display_Adapted_String
     (Source   : String;
      Before   : Positive;
      New_Item : String;
      Pattern  : String)
   is
      S_Ins_In : String := Source;
      S_Ovr_In : String := Source;
      S_Del_In : String := Source;

      S_Ins : String :=
        Insert (Source,
                Before,
                New_Item & " ");
      S_Ovr : String :=
        Overwrite (Source,
                   Before,
                   New_Item);
      S_Del : String :=
        Trim (Delete (Source,
                      Before,
                      Before + Pattern'Length - 1),
              Ada.Strings.Right);
   begin
      Insert (S_Ins_In,
              Before,
              New_Item,
              Right);

      Overwrite (S_Ovr_In,
                 Before,
                 New_Item,
                 Right);

      Delete (S_Del_In,
              Before,
              Before + Pattern'Length - 1);

      Put_Line ("Original:  '"
                & Source & "'");

      Put_Line ("Insert:    '"
                & S_Ins  & "'");
      Put_Line ("Overwrite: '"
                & S_Ovr  & "'");
      Put_Line ("Delete:    '"
```

(continues on next page)

(continued from previous page)

```
                & S_Del  & "'");

      Put_Line ("Insert    (in-place): '"
                & S_Ins_In & "'");
      Put_Line ("Overwrite (in-place): '"
                & S_Ovr_In & "'");
      Put_Line ("Delete    (in-place): '"
                & S_Del_In & "'");
   end Display_Adapted_String;

   Idx : Natural;
begin
   Idx := Index
     (Source  => S,
      Pattern => P);

   if Idx > 0 then
      Display_Adapted_String (S, Idx, N, P);
   end if;
end Show_Adapted_Strings;
```

Build output

```
show_adapted_strings.adb:7:04: warning: "S" is not modified, could be declared␣
 ↪constant [-gnatwk]
show_adapted_strings.adb:21:07: warning: "S_Ins" is not modified, could be␣
 ↪declared constant [-gnatwk]
show_adapted_strings.adb:25:07: warning: "S_Ovr" is not modified, could be␣
 ↪declared constant [-gnatwk]
show_adapted_strings.adb:29:07: warning: "S_Del" is not modified, could be␣
 ↪declared constant [-gnatwk]
```

Runtime output

```
Original:  'Hello World'
Insert:    'Hello Beautiful World'
Overwrite: 'Hello Beautiful'
Delete:    'Hello'
Insert    (in-place): 'Hello Beaut'
Overwrite (in-place): 'Hello Beaut'
Delete    (in-place): 'Hello      '
```

В этом примере мы ищем индекс подстроки `World` и выполняем операции с этой подстрокой внутри внешней строки. Процедура `Display_Adapted_String` использует обе версии операций. Для процедурной версии `Insert` и `Overwrite` мы применяем усечение к правой стороне строки (`Right`). Для процедуры `Delete` мы указываем диапазон подстроки, которая заменяется пробелами. Для функциональной версии `Delete` мы также вызываем `Trim`. Эта функция которая обрезает конечные пробелы.

21.2 Ограничение строк фиксированной длины

Использование строк фиксированной длины обычно достаточно хорошо для строк, которые инициализируются при их объявлении. Однако, как видно из предыдущего раздела, процедурные операции со строками вызывают трудности при работе со строками фиксированной длины, поскольку строки фиксированной длины представляют собой массивы символов. В следующем примере показано, насколько громоздкой может быть инициализация строк фиксированной длины, если она не выполняется в объявлении:

Listing 4: show_char_array.adb

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

procedure Show_Char_Array is
   S : String (1 .. 15);
   --  Strings are arrays of Character
begin
   S := "Hello          ";
   --  Alternatively:
   --
   --  #1:
   --      S (1 .. 5)      := "Hello";
   --      S (6 .. S'Last) := (others => ' ');
   --
   --  #2:
   --      S := ('H', 'e', 'l', 'l', 'o',
   --            others => ' ');

   Put_Line ("String: " & S);
   Put_Line ("String Length: "
             & Integer'Image (S'Length));
end Show_Char_Array;
```

Runtime output

```
String: Hello
String Length:  15
```

В этом случае мы не можем просто написать `S := "Hello"`, потому что результирующий массив символов для константы `Hello` имеет длину, отличную от длины строки `S` .Следовательно, нам необходимо включить конечные пробелы, чтобы соответствовать длине S. Как показано в примере, мы могли бы использовать точный диапазон для инициализации (`S (1 .. 5)`) или использовать явный массив отдельных символов.

Когда строки инициализируются или обрабатываются во время выполнения, обычно лучше использовать ограниченные или неограниченные строки. Важной особенностью этих типов является то, что они не являются массивами, поэтому описанные выше трудности не применяются. Начнем с ограниченных строк.

21.3 Ограниченные строки

Ограниченные строки определены в пакете `Ada.Strings.Bounded.Generic_Bounded_Length`. Поскольку это общий пакет, вам необходимо создать его экземпляр и установить максимальную длину ограниченной строки. Затем вы можете объявить ограниченные строки типа `Bounded_String`.

Строки как ограниченной, так и фиксированной длины имеют максимальную длину, которую они могут вместить. Однако ограниченные строки не являются массивами, поэтому их инициализация во время выполнения намного проще. Например:

Listing 5: show_bounded_string.adb

```
with Ada.Strings;         use Ada.Strings;
with Ada.Strings.Bounded;
with Ada.Text_IO;         use Ada.Text_IO;

procedure Show_Bounded_String is
   package B_Str is new
     Ada.Strings.Bounded.Generic_Bounded_Length (Max => 15);
   use B_Str;

   S1, S2 : Bounded_String;

   procedure Display_String_Info (S : Bounded_String) is
   begin
      Put_Line ("String: " & To_String (S));
      Put_Line ("String Length: "
                & Integer'Image (Length (S)));
      --  String:
      --           S'Length => ok
      --  Bounded_String:
      --           S'Length => compilation error:
      --                       bounded strings are
      --                       not arrays!

      Put_Line ("Max.   Length: "
                & Integer'Image (Max_Length));
   end Display_String_Info;
begin
   S1 := To_Bounded_String ("Hello");
   Display_String_Info (S1);

   S2 := To_Bounded_String ("Hello World");
   Display_String_Info (S2);

   S1 := To_Bounded_String
     ("Something longer to say here...",
      Right);
   Display_String_Info (S1);
end Show_Bounded_String;
```

Runtime output

```
String: Hello
String Length:  5
Max.   Length:  15
String: Hello World
String Length:  11
Max.   Length:  15
String: Something longe
```

(continues on next page)

(continued from previous page)

```
String Length:  15
Max.   Length:  15
```

Используя ограниченные строки, мы можем легко назначить S1 и S2 несколько раз во время выполнения. Мы используем функции To_Bounded_String и To_String для преобразования в соответствующем направлении между строками фиксированной длины и ограниченными строками. Вызов To_Bounded_String возбуждает исключение, если длина входной строки больше максимальной длинны ограниченной строки. Чтобы этого избежать, мы можем использовать параметр усечения (Right в нашем примере).

Ограниченные строки не являются массивами, поэтому нельзя использовать атрибут 'Length, как это было для строк фиксированной длины. Вместо этого вызывается функция Length, которая возвращает длину ограниченной строки. Константа Max_Length представляет максимальную длину ограниченной строки, заданную при настройке экземпляра пакета.

После инициализации ограниченной строки мы можем с ней работать. Например, мы можем добавить строку в ограниченную строку с помощью Append или выполнить конкатенацию ограниченных строк с помощью оператора &. Вот так:

Listing 6: show_bounded_string_op.adb

```
with Ada.Strings;          use Ada.Strings;
with Ada.Strings.Bounded;
with Ada.Text_IO;          use Ada.Text_IO;

procedure Show_Bounded_String_Op is
   package B_Str is new
     Ada.Strings.Bounded.Generic_Bounded_Length (Max => 30);
   use B_Str;

   S1, S2 : Bounded_String;
begin
   S1 := To_Bounded_String ("Hello");
   --  Alternatively:
   --
   --  A := Null_Bounded_String & "Hello";

   Append (S1, " World");
   --  Alternatively: Append (A, " World", Right);

   Put_Line ("String: " & To_String (S1));

   S2 := To_Bounded_String ("Hello!");
   S1 := S1 & " " & S2;
   Put_Line ("String: " & To_String (S1));
end Show_Bounded_String_Op;
```

Runtime output

```
String: Hello World
String: Hello World Hello!
```

Мы можем инициализировать ограниченную строку пустой строкой, используя константу Null_Bounded_String. Кроме того, можно использовать процедуру Append и указать режим усечения, как в случае с функцией To_Bounded_String.

21.4 Неограниченные строки

Неограниченные строки определяются в пакете `Ada.Strings.Unbounded`. Это *не* настраиваемый пакет, поэтому нам не нужно создавать его экземпляр перед использованием `Unbounded_String` типа. Как можно вспомнить из предыдущего раздела, для ограниченных строк требуется настройка пакета.

Неограниченные строки похожи на ограниченные строки. Главное отличие состоит в том, что они могут содержать строки любого размера и подстраиваться в соответствии с входной строкой: если мы назначим, например, 10-символьную строку неограниченной строке, а позже назначим 50-символьную строку, внутренние операции в контейнере обеспечат выделение памяти для хранения новой строки. В большинстве случаев разработчикам не нужно беспокоиться об этих операциях. Кроме того, усечение не требуется.

Инициализация неограниченных строк очень похожа на ограниченные строки. Рассмотрим пример:

Listing 7: show_unbounded_string.adb

```
with Ada.Strings;           use Ada.Strings;
with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
with Ada.Text_IO;           use Ada.Text_IO;

procedure Show_Unbounded_String is
   S1, S2 : Unbounded_String;

   procedure Display_String_Info (S : Unbounded_String) is
   begin
      Put_Line ("String: " & To_String (S));
      Put_Line ("String Length: "
                & Integer'Image (Length (S)));
   end Display_String_Info;
begin
   S1 := To_Unbounded_String ("Hello");
   --  Alternatively:
   --
   --  A := Null_Unbounded_String & "Hello";

   Display_String_Info (S1);

   S2 := To_Unbounded_String ("Hello World");
   Display_String_Info (S2);

   S1 := To_Unbounded_String ("Something longer to say here...");
   Display_String_Info (S1);
end Show_Unbounded_String;
```

Runtime output

```
String: Hello
String Length:  5
String: Hello World
String Length:  11
String: Something longer to say here...
String Length:  31
```

Как и ограниченные строки, мы можем назначать `S1` и `S2` несколько раз во время выполнения и использовать функции `To_Unbounded_String` и `To_String` для преобразования строк фиксированной длины в неограниченные строками и обратно. В этом случае усечение не нужно.

Как и для ограниченных строк, можно использовать процедуру `Append` и оператор `&` для

неограниченных строк. Например:

Listing 8: show_unbounded_string_op.adb

```
with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
with Ada.Text_IO;           use Ada.Text_IO;

procedure Show_Unbounded_String_Op is
   S1, S2 : Unbounded_String := Null_Unbounded_String;
begin
   S1 := S1 & "Hello";
   S2 := S2 & "Hello!";

   Append (S1, " World");
   Put_Line ("String: " & To_String (S1));

   S1 := S1 & " " & S2;
   Put_Line ("String: " & To_String (S1));
end Show_Unbounded_String_Op;
```

Runtime output

```
String: Hello World
String: Hello World Hello!
```

CHAPTER

TWENTYTWO

СТАНДАРТНАЯ БИБЛИОТЕКА: ФАЙЛЫ И ПОТОКИ

Ада предлагает различные средства для ввода/вывода файлов (I/O):

- *Текстовый* ввод-вывод в текстовом формате, включая отображение информации на консоли.
- *Последовательный* ввод-вывод в двоичном формате последовательным образом для конкретного типа данных.
- *Прямой* ввод-вывод в двоичном формате для конкретного типа данных, но также поддерживающий доступ к любой позиции файла.
- *Потоковый* ввод-вывод информации для нескольких типов данных, включая объекты неограниченных типов, с использованием файлов в двоичном формате.

В этой таблице представлено краткое описание функций, которые мы только что видели:

Опция ввода- вывода файлов	Формат	Произвольный доступ	Типы данных
Text I/O	текст		строковый тип
Sequential I/O	двоичный		одиночный тип
Direct I/O	двоичный	да	одиночный тип
Stream I/O	двоичный	да	несколько типов

В следующих разделах мы подробно обсудим эти средства ввода-вывода.

22.1 Текстовый ввод-вывод

В большинстве примеров этого курса мы использовали процедуру `Put_Line` для отображения информации на консоли. Однако эта процедура также принимает параметр **`File_Type`**. Например, вы можете выбрать между стандартным выводом (Standard_Output) и выводом стандартной ошибкой (Standard_Error), явно задав этот параметр:

Listing 1: show_std_text_out.adb

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

procedure Show_Std_Text_Out is
begin
   Put_Line (Standard_Output, "Hello World #1");
   Put_Line (Standard_Error,  "Hello World #2");
end Show_Std_Text_Out;
```

Runtime output

```
Hello World #1
Hello World #2
```

Вы также можете использовать этот параметр для записи информации в любой текстовый файл. Чтобы создать новый файл для записи, используйте процедуру `Create` для инициализации объекта `File_Type`, который впоследствии можно передать в `Put_Line` (вместо, например, `Standard_Output`). После того, как вы закончите запись информации, вы можете закрыть файл, вызвав процедуру `Close`.

Вы используете аналогичный метод для чтения информации из текстового файла. Однако при открытии файла вы должны указать, что это входной файл (`In_File`), а не выходной файл. Кроме того, вместо вызова процедуры `Put_Line` вы вызываете функцию `Get_Line` для чтения информации из файла.

Давайте посмотрим на пример, который записывает информацию в новый текстовый файл, а затем считывает ее обратно из того же файла:

Listing 2: show_simple_text_file_io.adb

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

procedure Show_Simple_Text_File_IO is
   F         : File_Type;
   File_Name : constant String := "simple.txt";
begin
   Create (F, Out_File, File_Name);
   Put_Line (F, "Hello World #1");
   Put_Line (F, "Hello World #2");
   Put_Line (F, "Hello World #3");
   Close (F);

   Open (F, In_File, File_Name);
   while not End_Of_File (F) loop
      Put_Line (Get_Line (F));
   end loop;
   Close (F);
end Show_Simple_Text_File_IO;
```

Runtime output

```
Hello World #1
Hello World #2
Hello World #3
```

В дополнение к процедурам `Create` и `Close` стандартная библиотека также включает процедуру `Reset`, которая, как следует из названия, сбрасывает (стирает) всю информацию из файла. Например:

Listing 3: show_text_file_reset.adb

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

procedure Show_Text_File_Reset is
   F         : File_Type;
   File_Name : constant String := "simple.txt";
begin
   Create (F, Out_File, File_Name);
   Put_Line (F, "Hello World #1");
   Reset (F);
   Put_Line (F, "Hello World #2");
   Close (F);

   Open (F, In_File, File_Name);
   while not End_Of_File (F) loop
      Put_Line (Get_Line (F));
```

(continues on next page)

(continued from previous page)

```
   end loop;
   Close (F);
end Show_Text_File_Reset;
```

Runtime output

```
Hello World #2
```

Запустив эту программу, мы замечаем, что, хотя мы записали первую строку (`"Hello World #1"`) в файл, она была удалена вызовом процедуры сброса `Reset`.

В дополнение к открытию файла для чтения или записи, вы также можете открыть существующий файл для добавления в конец. Сделайте это, вызвав процедуру `Open` с параметром `Append_File`.

При вызове процедуры открытия `Open` возбуждает исключение, если указанный файл не найден. Поэтому вы должны обрабатывать исключения в этом контексте. В следующем примере удаляется файл, а затем предпринимается попытка открыть тот же файл для чтения:

Listing 4: show_text_file_input_except.adb

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

procedure Show_Text_File_Input_Except is
   F         : File_Type;
   File_Name : constant String := "simple.txt";
begin
   --  Open output file and delete it
   Create (F, Out_File, File_Name);
   Delete (F);

   --  Try to open deleted file
   Open (F, In_File, File_Name);
   Close (F);
exception
   when Name_Error =>
      Put_Line ("File does not exist");
   when others =>
      Put_Line ("Error while processing input file");
end Show_Text_File_Input_Except;
```

Runtime output

```
File does not exist
```

В этом примере файл создается вызовом `Create`, а затем удаляется вызовом `Delete`. После вызова функции `Delete` мы больше не можем использовать объект **`File_Type`**. После удаления файла мы пытаемся открыть несуществующий файл, что возбуждает исключение `Name_Error`.

22.2 Последовательный ввод-вывод

В предыдущем разделе представлена подробная информация о вводе/выводе текстовых файлов. Здесь мы обсудим выполнение операций ввода-вывода для файловов двоичного формата. Первый пакет, который мы рассмотрим - это `Ada.Sequential_IO`. Поскольку этот пакет является настраиваемым, необходимо его настроить на тип данных ввод/вывлд которого мы будем производить. После этого можно использовать те же процедуры, что и в предыдущем разделе: `Create`, `Open`, `Close`, `Reset` и `Delete`. Однако вместо вызова процедур `Get_Line` и `Put_Line` следует вызвать процедуры `Read` и `Write`.

В следующем примере создается настройка пакета `Ada.Sequential_IO` для типов с плавающей запятой:

Listing 5: show_seq_float_io.adb

```
with Ada.Text_IO;
with Ada.Sequential_IO;

procedure Show_Seq_Float_IO is
   package Float_IO is
     new Ada.Sequential_IO (Float);
   use Float_IO;

   F         : Float_IO.File_Type;
   File_Name : constant String := "float_file.bin";
begin
   Create (F, Out_File, File_Name);
   Write (F,  1.5);
   Write (F,  2.4);
   Write (F,  6.7);
   Close (F);

   declare
      Value : Float;
   begin
      Open (F, In_File, File_Name);
      while not End_Of_File (F) loop
         Read (F, Value);
         Ada.Text_IO.Put_Line (Float'Image (Value));
      end loop;
      Close (F);
   end;
end Show_Seq_Float_IO;
```

Runtime output

```
 1.50000E+00
 2.40000E+00
 6.70000E+00
```

Мы используем один и тот же подход для чтения и записи сложной структурной информации. В следующем примере используется тип запись, включающая логическое значение типа Boolean и значение с плавающей запятой типа Float:

Listing 6: show_seq_rec_io.adb

```
with Ada.Text_IO;
with Ada.Sequential_IO;

procedure Show_Seq_Rec_IO is
   type Num_Info is record
```

(continues on next page)

(continued from previous page)

```
      Valid : Boolean := False;
      Value : Float;
   end record;

   procedure Put_Line (N : Num_Info) is
   begin
      if N.Valid then
         Ada.Text_IO.Put_Line ("(ok,      "
                               & Float'Image (N.Value) & ")");
      else
         Ada.Text_IO.Put_Line ("(not ok,  -----------)");
      end if;
   end Put_Line;

   package Num_Info_IO is new Ada.Sequential_IO (Num_Info);
   use Num_Info_IO;

   F         : Num_Info_IO.File_Type;
   File_Name : constant String := "float_file.bin";
begin
   Create (F, Out_File, File_Name);
   Write (F,  (True,  1.5));
   Write (F,  (False, 2.4));
   Write (F,  (True,  6.7));
   Close (F);

   declare
      Value : Num_Info;
   begin
      Open (F, In_File, File_Name);
      while not End_Of_File (F) loop
         Read (F, Value);
         Put_Line (Value);
      end loop;
      Close (F);
   end;
end Show_Seq_Rec_IO;
```

Runtime output

```
(ok,      1.50000E+00)
(not ok,  -----------)
(ok,      6.70000E+00)
```

Как показывает пример, мы можем использовать тот же подход, который мы использовали для типов с плавающей запятой, для выполнения файлового ввода-вывода для этой записи. После того, как мы создадим экземпляр пакета `Ada.Sequential_IO` для типа записи, операции ввода-вывода файлов будут выполняться таким же образом.

22.3 Прямой ввод-вывод

Прямой ввод-вывод доступен в пакете `Ada.Direct_IO`. Этот механизм похож на только что представленный последовательный ввод-вывод, но позволяет нам получить доступ к любой позиции в файле. Создание пакета и большинство операций очень похожи на последовательный ввод-вывод. Чтобы переписать приложение `Show_Seq_Float_IO` из предыдущего раздела для использования пакета `Ada.Direct_IO`, нам просто нужно заменить `Ada.Sequential_IO` на `Ada.Direct_IO` в настройке пакета. Это новый исходный код:

Listing 7: show_dir_float_io.adb

```
with Ada.Text_IO;
with Ada.Direct_IO;

procedure Show_Dir_Float_IO is
   package Float_IO is new Ada.Direct_IO (Float);
   use Float_IO;

   F         : Float_IO.File_Type;
   File_Name : constant String := "float_file.bin";
begin
   Create (F, Out_File, File_Name);
   Write (F,  1.5);
   Write (F,  2.4);
   Write (F,  6.7);
   Close (F);

   declare
      Value : Float;
   begin
      Open (F, In_File, File_Name);
      while not End_Of_File (F) loop
         Read (F, Value);
         Ada.Text_IO.Put_Line (Float'Image (Value));
      end loop;
      Close (F);
   end;
end Show_Dir_Float_IO;
```

Runtime output

```
 1.50000E+00
 2.40000E+00
 6.70000E+00
```

В отличие от последовательного ввода-вывода, прямой ввод-вывод позволяет получить доступ к любой позиции в файле. Однако он не предлагает возможность добавлять информацию в конец файла. Вместо этого он предоставляет режим `Inout_File`, позволяющий читать и записывать в файл через один и тот же объект **`File_Type`**.

Чтобы получить доступ к нужной позиции в файле, вызовите процедуру `Set_Index` и она установит новую позицию / индекс. Вы можете использовать функцию `Index` чтобы узать текущий индекс. Посмотрим на пример:

Listing 8: show_dir_float_in_out_file.adb

```
with Ada.Text_IO;
with Ada.Direct_IO;

procedure Show_Dir_Float_In_Out_File is
```

(continues on next page)

(continued from previous page)

```
   package Float_IO is new Ada.Direct_IO (Float);
   use Float_IO;

   F         : Float_IO.File_Type;
   File_Name : constant String := "float_file.bin";
begin
   --  Open file for input / output
   Create (F, Inout_File, File_Name);
   Write (F,  1.5);
   Write (F,  2.4);
   Write (F,  6.7);

   --  Set index to previous position and overwrite value
   Set_Index (F, Index (F) - 1);
   Write (F,  7.7);

   declare
      Value : Float;
   begin
      --  Set index to start of file
      Set_Index (F, 1);

      while not End_Of_File (F) loop
         Read (F, Value);
         Ada.Text_IO.Put_Line (Float'Image (Value));
      end loop;
      Close (F);
   end;
end Show_Dir_Float_In_Out_File;
```

Runtime output

```
 1.50000E+00
 2.40000E+00
 7.70000E+00
```

Запустив этот пример, мы видим, что файл содержит значение `7.7` и не содержит значения `6.7`, которое мы записали сначала. Мы перезаписали значение, установив индекс на предыдущую позицию перед выполнением следующей операции записи.

В этом примере мы использовали режим `Inout_File`. Используя этот режим, мы просто вернули индекс в начальное положение перед чтением из файла (`Set_Index (F, 1)`) вместо того, чтобы закрывать файл и повторно открывать его для чтения.

22.4 Потоковый ввод-вывод

Все предыдущие подходы к файловому вводу-выводу в двоичном формате (последовательный и прямой ввод-вывод) работают с одним типом данных (тем, на который мы их настраиваем). Вы можете использовать эти подходы для записи объектов одного типа данных, хотя это может быть массивы или записи (потенциально со многими полями), но если вам нужно создать или обработать файлы, которые включают разные типы данных, либо объекты неограниченного типа, этих средств недостаточно. Вместо этого вы должны использовать потоковый ввод-вывод.

Потоковый ввод-вывод имеет некоторые общие черты с предыдущими подходами. Мы по-прежнему используем процедуры `Create`, `Open` и `Close`. Однако вместо прямого доступа к файлу через объект `File_Type` вы используете тип `Stream_Access`. Для чтения и записи

информации вы используете атрибуты 'Read или 'Write тех типов данных, которые вы читаете или пишете.

Давайте посмотрим на версию процедуры Show_Dir_Float_IO из предыдущего раздела. Процедура использует потоковый ввод-вывод вместо прямого ввода-вывода:

Listing 9: show_float_stream.adb

```
with Ada.Text_IO;
with Ada.Streams.Stream_IO; use Ada.Streams.Stream_IO;

procedure Show_Float_Stream is
   F         : File_Type;
   S         : Stream_Access;
   File_Name : constant String := "float_file.bin";
begin
   Create (F, Out_File, File_Name);
   S := Stream (F);

   Float'Write (S, 1.5);
   Float'Write (S, 2.4);
   Float'Write (S, 6.7);

   Close (F);

   declare
      Value : Float;
   begin
      Open (F, In_File, File_Name);
      S := Stream (F);

      while not End_Of_File (F) loop
         Float'Read (S, Value);
         Ada.Text_IO.Put_Line (Float'Image (Value));
      end loop;
      Close (F);
   end;
end Show_Float_Stream;
```

Runtime output

```
 1.50000E+00
 2.40000E+00
 6.70000E+00
```

После вызова Create мы получаем соответствующее значение Stream_Access, вызывая функцию Stream (поток). Затем мы используем этот поток для записи информации в файл используя атрибут 'Write типа **Float**. После закрытия файла и повторного открытия его для чтения мы снова получаем значение Stream_Access и используем его для чтения информации из файла с помощью атрибута 'Read типа **Float**.

Вы можете использовать потоки для создания и обработки файлов, содержащих разные типы данных в одном файле. Вы также можете читать и записывать неограниченные типы данных, такие как строки. Однако при использовании неограниченных типов данных вы должны вызывать атрибуты 'Input и 'Output неограниченного типа данных: эти атрибуты записывают информацию о границах или дискриминантах в дополнение к фактическим данным объекта.

В следующем примере показан файловый ввод-вывод, который смешивает как строки разной длины, так и значения с плавающей запятой:

Listing 10: show_string_stream.adb

```
with Ada.Text_IO;
with Ada.Streams.Stream_IO; use Ada.Streams.Stream_IO;

procedure Show_String_Stream is
   F         : File_Type;
   S         : Stream_Access;
   File_Name : constant String := "float_file.bin";

   procedure Output (S  : Stream_Access;
                     FV : Float;
                     SV : String) is
   begin
      String'Output (S, SV);
      Float'Output (S,  FV);
   end Output;

   procedure Input_Display (S : Stream_Access) is
      SV : String := String'Input (S);
      FV : Float  := Float'Input (S);
   begin
      Ada.Text_IO.Put_Line (Float'Image (FV)
                            & " --- " & SV);
   end Input_Display;

begin
   Create (F, Out_File, File_Name);
   S := Stream (F);

   Output (S, 1.5, "Hi!!");
   Output (S, 2.4, "Hello world!");
   Output (S, 6.7, "Something longer here...");

   Close (F);

   Open (F, In_File, File_Name);
   S := Stream (F);

   while not End_Of_File (F) loop
      Input_Display (S);
   end loop;
   Close (F);

end Show_String_Stream;
```

Build output

```
show_string_stream.adb:18:07: warning: "SV" is not modified, could be declared␣
 ↪constant [-gnatwk]
show_string_stream.adb:19:07: warning: "FV" is not modified, could be declared␣
 ↪constant [-gnatwk]
```

Runtime output

```
 1.50000E+00 --- Hi!!
 2.40000E+00 --- Hello world!
 6.70000E+00 --- Something longer here...
```

Когда вы используете потоковый ввод-вывод, в файл не записывается никакая информация, указывающая тип данных, которые вы записали. Если файл содержит данные разных типов, при чтении файла вы должны ссылаться на типы в том же порядке, что и при его

написании. В противном случае полученная информация будет повреждена. К сожалению, строгая типизация данных в этом случае вам не поможет. Написание простых процедур для файлового ввода-вывода (как в приведенном выше примере) может помочь обеспечить согласованность формата файла.

Как и прямой ввод-вывод, поддержка потокового ввода-вывода также позволяет получить доступ к любому месту в файле. Однако при этом нужно быть предельно осторожным, чтобы положение нового индекса соответствовало ожидаемым типам данных.

CHAPTER

TWENTYTHREE

СТАНДАРТНАЯ БИБЛИОТЕКА: NUMERICS

Стандартная библиотека обеспечивает поддержку широко распространенных математических операций для типов с плавающей запятой, комплексных чисел и матриц. В нижеследующих разделах приводится краткое введение в эти математические операции.

23.1 Элементарные функции

Пакет `Ada.Numerics.Elementary_Functions` обеспечивает общепринятые операции для типов с плавающей точкой, такие как квадратный корень, логарифм и тригонометрические функции (типа sin, cos). Например:

Listing 1: show_elem_math.adb

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

with Ada.Numerics.Elementary_Functions;
use  Ada.Numerics.Elementary_Functions;

procedure Show_Elem_Math is
   X : Float;
begin
   X := 2.0;
   Put_Line ("Square root of "
             & Float'Image (X)
             & " is "
             & Float'Image (Sqrt (X)));

   X := e;
   Put_Line ("Natural log of "
             & Float'Image (X)
             & " is "
             & Float'Image (Log (X)));

   X := 10.0 ** 6.0;
   Put_Line ("Log_10      of "
             & Float'Image (X)
             & " is "
             & Float'Image (Log (X, 10.0)));

   X := 2.0 ** 8.0;
   Put_Line ("Log_2       of "
             & Float'Image (X)
             & " is "
             & Float'Image (Log (X, 2.0)));

```

(continues on next page)

(continued from previous page)

```
34    X := Pi;
35    Put_Line ("Cos         of "
36              & Float'Image (X)
37              & " is "
38              & Float'Image (Cos (X)));
39
40    X := -1.0;
41    Put_Line ("Arccos      of "
42              & Float'Image (X)
43              & " is "
44              & Float'Image (Arccos (X)));
45 end Show_Elem_Math;
```

Runtime output

```
Square root of  2.00000E+00 is  1.41421E+00
Natural log of  2.71828E+00 is  1.00000E+00
Log_10      of  1.00000E+06 is  6.00000E+00
Log_2       of  2.56000E+02 is  8.00000E+00
Cos         of  3.14159E+00 is -1.00000E+00
Arccos      of -1.00000E+00 is  3.14159E+00
```

Здесь мы используем стандартные константы `e` и `Pi` из пакета `Ada.Numerics`.

Пакет `Ada.Numerics.Elementary_Functions` предоставляет операции для типа **`Float`**. Аналогичные пакеты есть для типов **`Long_Float`** и **`Long_Long_Float`**. Например, пакет `Ada.Numerics.Long_Elementary_Functions` предлагает тот же набор операций для типа **`Long_Float`**. Кроме того, пакет `Ada.Numerics.Generic_Elementary_Functions` - это настраиваемая версия пакета, которую можно использовать для пользовательских типов с плавающей запятой. Фактически, пакет `Elementary_Functions` можно определить следующим образом:

```
package Elementary_Functions is new
  Ada.Numerics.Generic_Elementary_Functions (Float);
```

23.2 Генерация случайных чисел

Пакет `Ada.Numerics.Float_Random` предоставляет простой генератор случайных чисел с диапазоном от 0,0 до 1,0. Чтобы использовать его, объявите генератор `G`, который вы затем передадите в `Random`. Например:

Listing 2: show_float_random_num.adb

```
1  with Ada.Text_IO;  use Ada.Text_IO;
2  with Ada.Numerics.Float_Random; use Ada.Numerics.Float_Random;
3
4  procedure Show_Float_Random_Num is
5     G : Generator;
6     X : Uniformly_Distributed;
7  begin
8     Reset (G);
9
10    Put_Line ("Some random numbers between "
11              & Float'Image (Uniformly_Distributed'First)
12              & " and "
13              & Float'Image (Uniformly_Distributed'Last)
14              & ":");
```

(continues on next page)

(continued from previous page)

```
   for I in 1 .. 15 loop
      X := Random (G);
      Put_Line (Float'Image (X));
   end loop;
end Show_Float_Random_Num;
```

Runtime output

```
Some random numbers between  0.00000E+00 and  1.00000E+00:
 9.29413E-01
 6.83346E-01
 2.30866E-01
 9.41220E-01
 7.81104E-01
 5.04554E-01
 2.73398E-01
 9.05950E-01
 6.31678E-01
 7.84824E-01
 4.11724E-01
 9.77724E-01
 5.53477E-01
 8.14905E-01
 2.13336E-01
```

Стандартная библиотека также включает генератор случайных чисел для дискретных чисел, который является частью пакета `Ada.Numerics.Discrete_Random`. Поскольку это настраиваемый пакет, необходимо создать его экземпляр для требуемого дискретного типа. Это позволяет вам задать диапазон генератора. В следующем примере создается приложение, отображающее случайные целые числа от 1 до 10:

Listing 3: show_discrete_random_num.adb

```
with Ada.Text_IO;  use Ada.Text_IO;
with Ada.Numerics.Discrete_Random;

procedure Show_Discrete_Random_Num is

   subtype Random_Range is Integer range 1 .. 10;

   package R is new
     Ada.Numerics.Discrete_Random (Random_Range);
   use R;

   G : Generator;
   X : Random_Range;
begin
   Reset (G);

   Put_Line ("Some random numbers between "
             & Integer'Image (Random_Range'First)
             & " and "
             & Integer'Image (Random_Range'Last)
             & ":");

   for I in 1 .. 15 loop
      X := Random (G);
      Put_Line (Integer'Image (X));
   end loop;
end Show_Discrete_Random_Num;
```

Runtime output

```
Some random numbers between  1 and  10:
 1
 1
 9
 4
 3
 9
 4
 4
 6
 6
 8
 1
 2
 2
 7
```

Здесь пакет R создается с типом Random_Range с ограничением диапазона от 1 до 10. Это позволяет нам контролировать диапазон получаемых случайных чисел. Мы могли бы легко изменить приложение для получения случайных целых чисел от 0 до 20, изменив спецификацию подтипа Random_Range. Аналогично мы можем использовать типы с плавающей запятой и с фиксированной запятой.

23.3 Комплексные числа

Пакет Ada.Numerics.Complex_Types обеспечивает поддержку комплексных чисел, а пакет Ada.Numerics.Complex_Elementary_Functions обеспечивает поддержку общепринятые операций с типами комплексных чисел, аналогично пакету Ada.Numerics.Elementary_Functions. Наконец, вы можете использовать пакет Ada.Text_IO.Complex_IO для выполнения операций ввода-вывода над комплексными числами. В следующем примере мы объявляем переменные типа Complex и инициализируем их с помощью агрегата:

Listing 4: show_elem_math.adb

```
1  with Ada.Text_IO;  use Ada.Text_IO;
2  with Ada.Numerics; use Ada.Numerics;
3
4  with Ada.Numerics.Complex_Types;
5  use  Ada.Numerics.Complex_Types;
6
7  with Ada.Numerics.Complex_Elementary_Functions;
8  use  Ada.Numerics.Complex_Elementary_Functions;
9
10 with Ada.Text_IO.Complex_IO;
11
12 procedure Show_Elem_Math is
13
14    package C_IO is new
15      Ada.Text_IO.Complex_IO (Complex_Types);
16    use C_IO;
17
18    X, Y  : Complex;
19    R, Th : Float;
20 begin
21    X := (2.0, -1.0);
22    Y := (3.0,  4.0);
```

(continues on next page)

(continued from previous page)

```
23
24    Put (X);
25    Put (" * ");
26    Put (Y);
27    Put (" is ");
28    Put (X * Y);
29    New_Line;
30    New_Line;
31
32    R  := 3.0;
33    Th := Pi / 2.0;
34    X  := Compose_From_Polar (R, Th);
35    --  Alternatively:
36    --  X := R * Exp ((0.0, Th));
37    --  X := R * e ** Complex'(0.0, Th);
38
39    Put ("Polar form:    "
40         & Float'Image (R)  & " * e**(i * "
41         & Float'Image (Th) & ")");
42    New_Line;
43
44    Put ("Modulus     of ");
45    Put (X);
46    Put (" is ");
47    Put (Float'Image (abs (X)));
48    New_Line;
49
50    Put ("Argument    of ");
51    Put (X);
52    Put (" is ");
53    Put (Float'Image (Argument (X)));
54    New_Line;
55    New_Line;
56
57    Put ("Sqrt        of ");
58    Put (X);
59    Put (" is ");
60    Put (Sqrt (X));
61    New_Line;
62 end Show_Elem_Math;
```

Runtime output

```
( 2.00000E+00,-1.00000E+00) * ( 3.00000E+00, 4.00000E+00) is ( 1.00000E+01, 5.
↪00000E+00)

Polar form:     3.00000E+00 * e**(i *  1.57080E+00)
Modulus     of (-1.31134E-07, 3.00000E+00) is  3.00000E+00
Argument    of (-1.31134E-07, 3.00000E+00) is  1.57080E+00

Sqrt        of (-1.31134E-07, 3.00000E+00) is ( 1.22474E+00, 1.22474E+00)
```

Как видно из этого примера, все общепринятые операции, такие, как * и +, доступны для комплексных типов. У вас также есть типичные операции комплексных чисел, такие как Argument и Exp. Помимо инициализации комплексных чисел в декартовой форме с использованием агрегатов, вы можете получать их из полярной формы, вызвав функцию Compose_From_Polar.

Пакеты Ada.Numerics.Complex_Types и Ada.Numerics.Complex_Elementary_Functions предоставляют операции для типа **Float**. Подобные пакеты доступны для типов **Long_Float** и **Long_Long_Float**. Кроме того, с помощью настраиваемых пакетов Ada.Numerics.

Generic_Complex_Types и Ada.Numerics.Generic_Complex_Elementary_Functions можно создавать комплексные типы и их функции из пользовательских или предопределенных типов с плавающей запятой. Например:

```
with Ada.Numerics.Generic_Complex_Types;
with Ada.Numerics.Generic_Complex_Elementary_Functions;
with Ada.Text_IO.Complex_IO;

procedure Show_Elem_Math is

   package Complex_Types is new
     Ada.Numerics.Generic_Complex_Types (Float);
   use Complex_Types;

   package Elementary_Functions is new
     Ada.Numerics.Generic_Complex_Elementary_Functions
       (Complex_Types);
   use Elementary_Functions;

   package C_IO is new Ada.Text_IO.Complex_IO
     (Complex_Types);
   use C_IO;

   X, Y  : Complex;
   R, Th : Float;
```

23.4 Работа с векторами и матрицами

Пакет Ada.Numerics.Real_Arrays обеспечивает поддержку векторов и матриц. Он включает в себя типичные операции с матрицами, такие как обращение, нахождение определителя и собственного значения, а также более простые операции, такие как сложение и умножение матриц. Вы можете объявлять векторы и матрицы, используя типы Real_Vector и Real_Matrix соответственно.

В следующем примере используются некоторые операции из пакета Ada.Numerics.Real_Arrays:

Listing 5: show_matrix.adb

```
1  with Ada.Text_IO;  use Ada.Text_IO;
2
3  with Ada.Numerics.Real_Arrays;
4  use  Ada.Numerics.Real_Arrays;
5
6  procedure Show_Matrix is
7
8     procedure Put_Vector (V : Real_Vector) is
9     begin
10       Put ("    (");
11       for I in V'Range loop
12          Put (Float'Image (V (I)) & " ");
13       end loop;
14       Put_Line (")");
15    end Put_Vector;
16
17    procedure Put_Matrix (M : Real_Matrix) is
18    begin
19       for I in M'Range (1) loop
20          Put ("    (");
```

(continues on next page)

(continued from previous page)

```
         for J in M'Range (2) loop
            Put (Float'Image (M (I, J)) & " ");
         end loop;
         Put_Line (")");
      end loop;
   end Put_Matrix;

   V1       : Real_Vector := (1.0, 3.0);
   V2       : Real_Vector := (75.0, 11.0);

   M1       : Real_Matrix :=
                ((1.0, 5.0, 1.0),
                 (2.0, 2.0, 1.0));
   M2       : Real_Matrix :=
                ((31.0, 11.0, 10.0),
                 (34.0, 16.0, 11.0),
                 (32.0, 12.0, 10.0),
                 (31.0, 13.0, 10.0));
   M3       : Real_Matrix := ((1.0, 2.0),
                              (2.0, 3.0));
begin
   Put_Line ("V1");
   Put_Vector (V1);
   Put_Line ("V2");
   Put_Vector (V2);
   Put_Line ("V1 * V2 =");
   Put_Line ("    "
             & Float'Image (V1 * V2));
   Put_Line ("V1 * V2 =");
   Put_Matrix (V1 * V2);
   New_Line;

   Put_Line ("M1");
   Put_Matrix (M1);
   Put_Line ("M2");
   Put_Matrix (M2);
   Put_Line ("M2 * Transpose(M1) =");
   Put_Matrix (M2 * Transpose (M1));
   New_Line;

   Put_Line ("M3");
   Put_Matrix (M3);
   Put_Line ("Inverse (M3) =");
   Put_Matrix (Inverse (M3));
   Put_Line ("abs Inverse (M3) =");
   Put_Matrix (abs Inverse (M3));
   Put_Line ("Determinant (M3) =");
   Put_Line ("    "
             & Float'Image (Determinant (M3)));
   Put_Line ("Solve (M3, V1) =");
   Put_Vector (Solve (M3, V1));
   Put_Line ("Eigenvalues (M3) =");
   Put_Vector (Eigenvalues (M3));
   New_Line;
end Show_Matrix;
```

Build output

```
show_matrix.adb:28:04: warning: "V1" is not modified, could be declared constant [-
↪gnatwk]
show_matrix.adb:29:04: warning: "V2" is not modified, could be declared constant [-
↪gnatwk]
```

(continues on next page)

(continued from previous page)

```
show_matrix.adb:31:04: warning: "M1" is not modified, could be declared constant [-
↪gnatwk]
show_matrix.adb:34:04: warning: "M2" is not modified, could be declared constant [-
↪gnatwk]
show_matrix.adb:39:04: warning: "M3" is not modified, could be declared constant [-
↪gnatwk]
```

Runtime output

```
V1
    ( 1.00000E+00  3.00000E+00 )
V2
    ( 7.50000E+01  1.10000E+01 )
V1 * V2 =
     1.08000E+02
V1 * V2 =
    ( 7.50000E+01  1.10000E+01 )
    ( 2.25000E+02  3.30000E+01 )

M1
    ( 1.00000E+00  5.00000E+00  1.00000E+00 )
    ( 2.00000E+00  2.00000E+00  1.00000E+00 )
M2
    ( 3.10000E+01  1.10000E+01  1.00000E+01 )
    ( 3.40000E+01  1.60000E+01  1.10000E+01 )
    ( 3.20000E+01  1.20000E+01  1.00000E+01 )
    ( 3.10000E+01  1.30000E+01  1.00000E+01 )
M2 * Transpose(M1) =
    ( 9.60000E+01  9.40000E+01 )
    ( 1.25000E+02  1.11000E+02 )
    ( 1.02000E+02  9.80000E+01 )
    ( 1.06000E+02  9.80000E+01 )

M3
    ( 1.00000E+00  2.00000E+00 )
    ( 2.00000E+00  3.00000E+00 )
Inverse (M3) =
    (-3.00000E+00  2.00000E+00 )
    ( 2.00000E+00 -1.00000E+00 )
abs Inverse (M3) =
    ( 3.00000E+00  2.00000E+00 )
    ( 2.00000E+00  1.00000E+00 )
Determinant (M3) =
    -1.00000E+00
Solve (M3, V1) =
    ( 3.00000E+00 -1.00000E+00 )
Eigenvalues (M3) =
    ( 4.23607E+00 -2.36068E-01 )
```

Если вы не указываете размеры матрицы, они автоматически определяются на основе агрегата, используемого для инициализации. Хотя вы также можете использовать явные диапазоны. Например:

```
M1       : Real_Matrix (1 .. 2, 1 .. 3) :=
             ((1.0, 5.0, 1.0),
              (2.0, 2.0, 1.0));
```

Пакет `Ada.Numerics.Real_Arrays` предоставляет операции для типа **`Float`**. Аналогичные пакеты доступны для типов **`Long_Float`** и **`Long_Long_Float`**. Кроме того, настраиваемый пакет `Ada.Numerics.`**`Generic`**`_Real_Arrays` можно использовать для пользовательских

типов с плавающей запятой. Например, пакет `Real_Arrays` может быть определен следующим образом:

```
package Real_Arrays is new
  Ada.Numerics.Generic_Real_Arrays (Float);
```

CHAPTER

TWENTYFOUR

ПРИЛОЖЕНИЯ

24.1 Приложение A: Формальные типы настройки

Следующие таблицы содержат примеры доступных формальных типов настраиваемых модулей:

Формальный тип	Фактический тип
Неполный тип **Формат**: `type T;`	Любой тип
Дискретный тип **Формат**: `type T is (<>);`	Любой целочисленный тип, модульный тип или перечислимый тип
Тип диапазона **Формат**: `type T is range <>;`	Любой целочисленный тип со знаком
Модульный тип **Формат**: `type T is mod <>;`	Любой модульный тип
Тип с плавающей запятой **Формат**: `type T is digits <>;`	Любой тип с плавающей запятой
Двоичный тип с фиксированной запятой **Формат**: `type T is delta <>;`	Любой двоичный тип с фиксированной запятой
Десятичный тип с фиксированной запятой **Формат**: `type T is delta <> digits <>;`	Любой десятичный тип с фиксированной запятой
Определенный нелимитируемый личный тип **Формат**: `type T is private;`	Любой нелимитируемый, определенный тип
Нелимитируемый личный тип с дискриминантом **Формат**: `type T (D : DT) is private;`	Любой нелимитируемый тип с дискриминантом
Ссылочный тип **Формат**: `type A is access T;`	Любой ссылочный тип для типа T
Определенный производный тип **Формат**: `type T is new B;`	Любой конкретный тип, производный от базового типа B
Лимитируемый частный тип **Формат**: `type T is limited private;`	Любой определенный тип, лимитируемый или нет
Неполное описание тегового типа **Формат**: `type T is tagged;`	Любой конкретный, определенный, теговый тип
Определенный теговый личный тип **Формат**: `type T is tagged private;`	Любой конкретный, определенный, теговый тип
Определенный теговый лимитируемый личный тип **Формат**: `type T is tagged limited private;`	Любой конкретный определенный теговый тип лимитируемый или нет.

continues on next page

Table 1 – continued from previous page

Формальный тип	Фактический тип
Определенный абстрактный теговый личный тип **Формат**: `type T is abstract tagged private;`	Любой нелимитируемый, определенный теговый тип абстрактный или конкретный
Определенный абстрактный теговый лимитируемый личный тип **Формат**: `type T is abstract tagged limited private;`	Любой определенный теговый тип, лимитир-мый или нет, абстрактный или конкретный
Определенный производный теговый тип **Формат**: `type T is new B with private;`	Любой конкретный теговый тип, производный от базового типа B
Определенный абстрактный производный теговый тип **Формат**: `type T is abstract new B with private;`	Любой теговый тип, производный от базового типа B абстрактный или конкретный
Тип массива **Формат**: `type A is array (R) of T;`	Любой тип массива с диапазоном R, содержащий элементы типа T
Интерфейсный тип **Формат**: `type T is interface;`	Любой интерфейсный тип
Лимитируемый интерфейсный тип **Формат**: `type T is limited interface;`	Любой лимитируемый интерфейсный тип
Задачный интерфейсный тип **Формат**: `type T is task interface;`	Любой задачный интерфейсный тип
Синхронизированный интерфейсный тип **Формат**: `type T is synchronized interface;`	Любой синхронизированный интерфейсный тип
Защищенный интерфейсный тип **Формат**: `type T is protected interface;`	Любой защищенный интерфейсный тип
Производный интерфейсный тип **Формат**: `type T is new B and I with private;`	Любой тип, производный от базового типа B и интерфейса I
Производный тип с несколькими интерфейсами **Формат**: `type T is new B and I1 and I2 with private;`	Любой тип, производный от базового типа B и интерфейсов I1 и I2
Абстрактный производный интерфейсный тип **Формат**: `type T is abstract new B and I with private;`	Любой тип, производный от абстрактного базового типа B и интерфейса I
Лимитируемый производный интерфейсный тип **Формат**: `type T is limited new B and I with private;`	Любой тип, производный от лимитируемого базового типа B и лимитируемого интерфейса
Абстрактный лимитируемый производный интерфейсный тип **Формат**: `type T is abstract limited new B and I with private;`	Любой тип, производный от абстрактного лимитиру емого базового типа B и лимитируемого интерфейса
Синхронизированный интерфейсный тип **Формат**: `type T is synchronized new SI with private;`	Любой тип, производный от синхронизированного интерфейсного типы SI
Абстрактный синхронизированный интерфейсный тип **Формат**: `type T is abstract synchronized new SI with private;`	Любой тип, производный от синхронизированного интерфейса SI

24.1.1 Неопределенные версии типов

Многие из приведенных выше примеров могут быть использованы для формальных неопределенных типов:

Формальный тип	Фактический тип
Неопределенный неполный тип **Формат**: `type T (<>);`	Любой тип
Неопределенный нелимитируемый личный тип **Формат**: `type T (<>) is private;`	Любой нелимитируемый тип неопределенный или определенный
Неопределенный лимитируемый личный тип **Формат**: `type T (<>) is limited private;`	Любой тип, лимитируемый или нет, неопределенный или определенный
Неполный неопределенный личный теговый тип **Формат**: `type T (<>) is tagged;`	Любой конкретный теговый тип, неопределенный или определенный
Неопределенный теговый личный тип **Формат**: `type T (<>) is tagged private;`	Любой конкретный, нелимитируемый теговый тип, неопределенный или определенный
Неопределенный теговый лимитируемый личный типми **Формат**: `type T (<>) is tagged limited private;`	Любой конкретный теговый тип, лимитируемый или нет, неопределенный или определенный
Неопределенный абстрактный теговый личный тип **Формат**: `type T (<>) is abstract tagged private;`	Любой нелимитируемый тег овый тип, неопределенный или определенный, абстрактный или конкретный
Неопределенный абстрактный теговый лимитируемый личный тип **Формат**: `type T (<>) is abstract tagged limited private;`	Любой теговый тип, лимитируемый или нет, неопределенный или определенный,абстрактный или конкретный
Неопределенный производный теговый тип **Формат**: `type T (<>) is new B with private;`	Любой теговый тип, производный от базового типа B, неопределенный или определенный
Неопределенный абстрактный производный теговый тип **Формат**: `type T (<>) is abstract new B with private;`	Любой тегоый тип, производный от базового типа B, неопределенный или определенный абстрактный или конкретный

Те же примеры могут также содержать дискриминанты. В этом случае, `(<>)` заменяется списком дискриминантов, например: `(D: DT)`.

24.2 Приложение B: Контейнеры

В следующей таблице показаны все контейнеры, доступные в Аде, включая их версии (стандартный, ограниченный, неограниченный, неопределенный):

Категория	Контейнер	Std	Bounded	Un-bounded	Indefi-nite
Вектор	`Vectors`	Y	Y		Y
Список	`Doubly Linked Lists`	Y	Y		Y
Отображение	`Hashed Maps`	Y	Y		Y
Отображение	`Ordered Maps`	Y	Y		Y
Множество	`Hashed Sets`	Y	Y		Y
Множество	`Ordered Sets`	Y	Y		Y
Дерево	`Multiway Trees`	Y	Y		Y
Универсальн	`Holders`				Y
Очередь	`Synchronized Queue Inter-faces`	Y			
Очередь	`Synchronized Queues`		Y	Y	
Очередь	`Priority Queues`		Y	Y	

Note: Чтобы получить имя пакета контейнера, замените пробел на _ в его названии. (Например, пакет для `Hashed Maps` называется `Hashed_Maps`.)

В следующей таблице представлены префиксы, применяемые к имени контейнера, которые зависят от его версии. Как указано в таблице, стандартная версия не имеет связанного с ней префикса.

Версия	Префикс именования
Std	
Ограниченный	`Bounded_`
Неограниченный	`Unbounded_`
Неопределенный	`Indefinite_`