Learning Ada

The complete contents of learn.adacore.com

LEARN.
ADACORE.COM

LEARN.
ADACORE.COM

Learning Ada

Release 2022-08

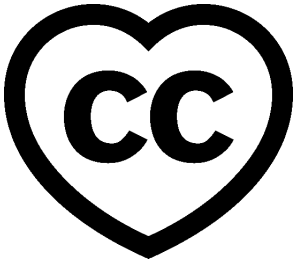
Various authors

Aug 22, 2022

CONTENTS

Part I

Введение в язык программирования Ada



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 Ади, 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 43) 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 73). 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 31). (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 109).

4.5 Переименование

Ранее мы упоминали, что *подпрограммы можно переименовывать* (page 29). 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)

```
12 begin
13    for I in My_Int loop
14       Put_Line (My_Int'Image (I));
15       --                ^ 'Image attribute
16       --                  converts a value
17       --                  to a String.
18    end loop;
19 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 154)). 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 29) и *пакетов* (page 40). 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)

```
17
18 end Dates;
```

Listing 3: dates.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Dates is
4
5     procedure Increase_Month (Some_Day : in out Date) is
6        --  Renaming components from
7        --  the Date record
8        M : Months  renames Some_Day.Month;
9        Y : Integer renames Some_Day.Year;
10
11       --  Renaming function (for Months
12       --  enumeration)
13       function Next (M : Months) return Months
14         renames Months'Succ;
15    begin
16       if M = December then
17          M := January;
18          Y := Y + 1;
19       else
20          M := Next (M);
21       end if;
22    end Increase_Month;
23
24    procedure Display_Month (Some_Day : Date) is
25       --  Renaming components from
26       --  the Date record
27       M : Months  renames Some_Day.Month;
28       Y : Integer renames Some_Day.Year;
29    begin
30       Put_Line ("Month: "
31                 & Months'Image (M)
32                 & ", Year:"
33                 & Integer'Image (Y));
34    end Display_Month;
35
36 end Dates;
```

Listing 4: main.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2  with Dates;       use Dates;
3
4  procedure Main is
5     D : Date := (1, January, 2000);
6  begin
7     Display_Month (D);
8
9     Put_Line ("Increasing month...");
10    Increase_Month (D);
11
12    Display_Month (D);
13 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 47)). 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 85).

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

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

procedure Greet is
   type My_Int is range 0 .. 1000;

   type My_Index   is range 1 .. 5;
   type Your_Index is range 1 .. 5;

   type My_Int_Array is array (My_Index) of My_Int;
   Tab : My_Int_Array := (2, 3, 5, 7, 11);
begin
   for I in Your_Index loop
      Put (My_Int'Image (Tab (I)));
   --                          ^ Compile time error
   end loop;
   New_Line;
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

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

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 237) 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 29), *пакеты* (page 40) 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 58).

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 47).

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)

```
22    --     Character value
23
24    C := Character'Val (65);
25    --  Assign the character at
26    --  position 65 in the
27    --  enumeration (which is 'A')
28
29    M := C;
30    --   ^ Invalid: C is of type Character,
31    --     and M is a My_Char
32
33    M := 'd';
34    --   ^ Invalid: 'd' is not a valid
35    --     literal for type My_Char
36 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 61). 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 35)), 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)

```
5
6     My_Colors : Color_Array (1 .. 5) :=
7       (Black, Red, Green, Blue, White);
8
9  begin
10    for C of My_Colors loop
11       Put_Line ("My_Color: "
12                 & Color'Image (C));
13    end loop;
14
15    New_Line;
16    Put_Line ("Reversing My_Color...");
17    New_Line;
18    Reverse_It (My_Colors);
19
20    for C of My_Colors loop
21       Put_Line ("My_Color: "
22                 & Color'Image (C));
23    end loop;
24
25 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)

```
procedure Show_Time_Task is
   package Aux renames Delay_Aux_Pkg;

   task T;

   task body T is
      Cycle : constant Time_Span :=
        Milliseconds (1000);
      Next  : Time := Aux.Get_Start_Time
                      + Cycle;

      Cnt   : Integer := 1;
   begin
      for I in 1 .. 5 loop
         delay until Next;

         Aux.Show_Elapsed_Time;
         Aux.Computational_Intensive_App;

         --  Calculate next execution time
         --  using a cycle of one second
         Next := Next + Cycle;

         Put_Line ("Cycle # "
                   & Integer'Image (Cnt));
         Cnt  := Cnt + 1;
      end loop;
      Put_Line ("Finished cycling");
   end T;

begin
   null;
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

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

procedure Show_Protected_Objects is

   protected Obj is
      --  Operations go here (only subprograms)
      procedure Set (V : Integer);
      function Get return Integer;
   private
      --  Data goes here
      Local : Integer := 0;
   end Obj;

   protected body Obj is
      --  procedures can modify the data
      procedure Set (V : Integer) is
      begin
         Local := V;
      end Set;

      --  functions cannot modify the data
      function Get return Integer is
      begin
         return Local;
      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

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Simple_Task is
4     task T;
5
6     task body T is
7     begin
8        Put_Line ("In task T");
9     end T;
10 begin
11    Put_Line ("In main");
12 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

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Simple_Task_Type is
4     task type TT;
5
6     task body TT is
7     begin
8        Put_Line ("In task type TT");
9     end TT;
10
11    A_Task : TT;
12 begin
13    Put_Line ("In main");
14 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 237);
- о дате и времени: Раздел *Даты и время* (page 221).

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

```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)

```

#include "test.h"

struct test {
  char name[80];
  char address[120];
};

static size_t
strlcpy(char *dst, const char *src, size_t dstsize)
{
  size_t len = strlen(src);
  if (dstsize) {
    size_t bl = (len < dstsize-1 ? len : dstsize-1);
    ((char*)memcpy(dst, src, bl))[bl] = 0;
  }
  return len;
}

struct test * test_create(void)
{
  return malloc (sizeof (struct test));
}

void test_destroy(struct test *t)
{
  if (t != NULL) {
    free(t);
  }
}

void test_reset(struct test *t)
{
  t->name[0]    = '\0';
  t->address[0] = '\0';
}

void test_set_name(struct test *t, char *name)
{
  strlcpy(t->name, name, sizeof(t->name));
}

void test_set_address(struct test *t, char *address)
{
  strlcpy(t->address, address, sizeof(t->address));
}

void test_display(const struct test *t)
{
  printf("Name:    %s\n", t->name);
  printf("Address: %s\n", t->address);
}
```

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)

```
   --  Dispatching: Calls Derived.Foo
   Foo (O4);
   --  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
```

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

```
with P; use P;

procedure Main is
   O1 : Derived := (A => 12);
   --  Declare an object of type Derived

   O2 : My_Class := My_Class (O1);

   O3 : My_Class'Class := O2;
begin
   Foo (O1);
   --  Non dispatching: Calls Derived.Foo
   Foo (O2);
   --  Non dispatching: Calls My_Class.Foo

   Foo (O3);
   --  Dispatching: Calls My_Class.Foo
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 109)), 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 обреботки. 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)

```
38    --
39    --  Using V.Insert to insert the element
40    --  into the vector
41    --
42    C := V.Find (10);
43    if C /= No_Element then
44       V.Insert (C, 9);
45    end if;
46
47    Show_Elements (V);
48
49 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

```
1  with Ada.Containers.Vectors;
2
3  with Ada.Text_IO; use Ada.Text_IO;
4
5  procedure Show_Remove_Vector_Element is
6     package Integer_Vectors is new
7       Ada.Containers.Vectors
8         (Index_Type   => Natural,
9          Element_Type => Integer);
10
11    use Integer_Vectors;
12
13    V : Vector := 20 & 10 & 0 & 13 & 10 & 13;
14    Idx : Extended_Index;
15    C   : Cursor;
16 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

```
with Ada.Containers; use Ada.Containers;
with Ada.Containers.Ordered_Sets;

with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Set_Ops is

   package Integer_Sets is new
     Ada.Containers.Ordered_Sets
       (Element_Type => Integer);

   use Integer_Sets;

   procedure Show_Elements (S : Set) is
   begin
      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_`

Part II

Безопасное и надежное программное обеспечение

Целью данной брошюры является продемонстрировать, как изучение языка Ада в целом и возможностей, введенных в редакциях Ада 2005, Ада 2012, в частности, поможет Вам разрабатывать безопасное и надежное программное обеспечение независимо от выбранного Вами языка реализации.

CHAPTER

TWENTYFIVE

ВСТУПЛЕНИЕ

Целью данной брошюры является продемонстрировать, как изучение языка Ада в целом и возможностей, введенных в редакциях Ада 2005, Ада 2012, в частности, поможет Вам разрабатывать безопасное и надежное программное обеспечение независимо от выбранного языка реализации. В конце концов, успешные реализации такого сорта программного обеспечения пишутся в духе Ады на любом языке! Спасибо Джону, постоянно демонстрирующему это в своих статьях, книгах и данной брошюре.

AdaCore посвящает эту брошюру памяти доктора Jean Ichbiah (1940-2007) — главному архитектору первой редакции языка Ада, заложившему надежный фундамент всех последующих редакций языка.

Франко Гасперони

Главный Исполнительный Директор, AdaCore

Париж, январь 2013

CHAPTER

TWENTYSIX

ПРЕДИСЛОВИЕ

Перед вами новая редакция брошюры «Безопасное и надежное программированное обеспечение». Она включает описание новых возможностей языка для поддержки контрактного программирования. Данные новшества были введены в ревизию языка Ада 2012. Они представляют особый интерес в областях, где существуют повышенные требования к безопасности и надежности программного обеспечения, особенно когда речь заходит о сертификации программ.

Я очень благодарен Бену Брасголу (Ben Brosgol) из AdaCore за помощь в подготовке новой редакции этой брошюры. Бен не только подготовил черновик новых разделов, но также исправил некоторые неясные, вводящие в заблуждения, а то и вовсе некорректные моменты в оригинале. К тому же он подготовил всеобъемлющий индекс, который, я уверен, оценят все читатели.

Джон Барнс.

Кавершам. Англия. Январь 2013

С момента публикации предыдущей редакции этой брошюры в 2013 году произошло важное событие: был выпущен SPARK 2014. SPARK представляет собой подмножество языка Ада, позволяющее применить методы формального анализа свойств программы, например, доказать отсутствие ошибок в коде. При этом остается возможность сочетать формальные методы с традиционными методами тестирования программ. Соответствующая глава «Сертификация с помощью SPARK» была значительно переработана, чтобы отразить изменения, введенные в SPARK 2014. Мы надеемся, что эта обновленная глава воодушевит читателей познакомиться поближе с этим интереснейшим новым языком и соответствующими технологиями.

CHAPTER

TWENTYSEVEN

БЕЗОПАСНЫЙ СИНТАКСИС

Часто сам синтаксис относят к скучным техническим деталям. Аргументируется это так — важно что именно вы скажете. При этом не так важно как вы это скажете. Это конечно же неверно. Ясность и однозначность очень важны при любом общении в цивилизованном мире.

Проводя аналогию, программа — это средство коммуникации между автором и читателем, будь то компилятор, другой член команды разработчиков, ответственный за контроль качества кода, либо любой другой человек. В самом деле, ведь чтение — это тоже способ общения. Ясный и однозначный синтаксис будет значительным подспорьем в подобном общении и, как мы убедимся далее, позволит избежать многих распространенных ошибок.

Важным свойством хорошо спроектированного синтаксиса является возможность находить ошибки в программе, вызванные типичными опечатками. Это позволит выявить их в момент компиляции вместо того, чтобы получить программу с непреднамеренным смыслом. Хотя тяжело предотвратить такие опечатки, как X вместо Y и + вместо *, многие опечатки, влияющие на структуру программы могут быть предотвращены. Заметим, однако, что полезно избегать коротких идентификаторов как раз по этой причине. К примеру, если финансовая программа оперирует курсом и временем, использовать в ней идентификаторы R и T чревато ошибками, поскольку легко ошибиться при наборе текста, ведь эти буквы находятся рядом на клавиатуре. В случае же использования идентификаторов Rate и Time возможные опечатки Tate, Rime будут легко выявлены компилятором.

27.1 Присваивание и проверка на равенство

Очевидно, что присваивание и проверка на равенство это разные действия. Если мы выполняем присваивание, мы изменяем состояние некоторой переменной. В то время как сравнение на равенство просто проверяет некоторое условие. Изменение состояния и проверка условия совершенно различные операции и осознавать это важно.

Во многих языках программирования запись этих операций может ввести в заблуждение.

Так в Fortran-е со дня его появления пишется:

```
X = X + 1
```

Но это весьма причудливая запись. В математике X никогда не равен X + 1. Данная инструкция в Fortran-е означает конечно же «заменить текущее значение переменной X старым значением, увеличенным на единицу». Но зачем использовать знак равенства таким способом, если в течении столетий он использовался для обозначения сравнения? (Знак равенства был предложен в 1550 году английским математиком Робертом Рекордом.) Создатели языка Algol 60 обратили внимание на эту проблему и использовали комбинацию двоеточия и знака равенства для записи операции присваивания:

```
X := X + 1;
```

Таким образом знак равенства можно использовать в операциях сравнения не теряя однозначности:

```
if X = 0 then ...
```

Язык C использует = для операции присваивания и, как следствие, вводит для операции сравнения двойной знак равенства (==). Это может запутать еще больше.

Вот фрагмент программы на C для управления воротами железной дороги:

```
if (the_signal == clean)
{
  open_gates(...);
  start_train(...);
}
```

Та же программа на языке Ада:

```
if The_Signal = Clean then
   Open_Gates(...);
   Start_Train(...);
end if;
```

Посмотрим, что будет, если C-программист допустит опечатку, введя единичный знак равенства вместо двойного:

```
if (the_signal = clean)
{
  open_gates(...);
  start_train(...);
}
```

Компиляция пройдет без ошибок, но вместо проверки the_signal произойдет присваивание значения clean переменной the_signal. Более того, в C нет различия между выражениями (вычисляющими значения) и присваиваниями (изменяющими состояние). Поэтому присваивание работает как выражение, а присвоенное значение затем используется в проверке условия. Если так случится, что clean не равно нулю, то условие будет считаться истинным, ворота откроются, the_signal примет значение clean и поезд уйдет в свой опасный путь. В противном случае, если clean закодирован как ноль, проверка не пройдет, ворота останутся заперты, а поезд — заблокированным. В любом случае все пойдет совсем не так, как нужно.

Ошибки связанные с использованием = для присваивания и == для проверки на равенство, то есть присваивания вместо выражений хорошо знакомы в C-сообществе. Для их выявления появились дополнительные правила оформления кода MISRA C[3] и средства анализа кода, такие как lint. Однако, следовало бы избежать подобных ловушек с самого момента разработки языка, что и было достигнуто в Аде.

Если Ада-программист по ошибке использует присваивание вместо проверки:

```
if The_Signal := Clean then  -- Ошибка
```

то программа просто не скомпилируется и все будет хорошо.

27.2 Группы инструкций

Часто необходимо сгруппировать последовательность из нескольких инструкций, например после проверки в условном операторе. Есть два типичных пути для этого:

- взять всю группу в скобки (как в C),
- закрыть последовательность чем-то, отвечающим "'if"' (как в Аде).

В C мы получим:

```
if (the_signal == clean)
{
  open_gates(...);
  start_train(...);
}
```

и, допустим, случайно добавим точку с запятой в конец первой строки. Получится следующее:

```
if (the_signal == clean) ;
{
  open_gates(...);
  start_train(...);
}
```

Теперь условие применяется только к пустому оператору, который неявно присутствует между условием и вновь введенным символом точки с запятой. Он практически невидим. И теперь не важно состояние the_signal, ворота все равно откроются и поезд поедет.

Вот для Ады соответствующая ошибка:

```
if The_Signal = Clean; then
   Open_Gates(...);
   Start_Train(...);
end if;
```

Это синтаксически неправильная запись, поэтому компилятор с легкостью ее обнаружит и предотвратит крушение поезда.

27.3 Именованное сопоставление

Еще одним свойством Ады синтаксической природы, помогающим избежать разнообразных ошибок, является именованная ассоциация, используемая в различных конструкциях. Хорошим примером может выступать запись даты, потому, что порядок следования ее составляющих различается в разных странах. К примеру 12 января 2008 года в Европе записывают, как 12/01/08, а в США обычно пишут 01/12/08 (не считая последних таможенных деклараций). В тоже время стандарт ISO требует писать вначале год — 08/01/12.

В C структура для хранения даты выглядит так:

```
struct date {
  int day, month, year;
};
```

что соответствует следующему типу в Аде:

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

В C мы можем написать:

```
struct date today = {1, 12, 8};
```

Но не имея перед глазами описания типа, мы не можем сказать, какая дата имеется в виду: 1 декабря 2008, 12 января 2008 или 8 декабря 2001.

В Аде есть возможность записать так:

```
Today : Date := (Day => 1, Month => 12, Year => 08);
```

Здесь используется именное сопоставление. Теперь не будет разночтений, даже, если мы запишем компоненты в другом порядке. (Заметьте, что в Аде допустимы лидирующие нули).

Следующая запись:

```
Today : Date := (Month => 12, Day => 1, Year => 08);
```

по-прежнему корректна и демонстрирует преимущество — нам нет нужды помнить порядок следования компонента записи.

Именованное сопоставление используется и в других конструкциях Ады. Подобные ошибки могут появляться при вызове подпрограмм, имеющих несколько параметров одного типа. Допустим, у нас есть функция вычисления индекса ожирения человека. Она имеет два параметра — высота и вес, заданные в фунтах и дюймах (либо в килограммах и метрах, для метрической системы измерений). В C мы имеем:

```
float index(float height, float weight) {
  ...
  return ...;
}
```

А в Аде:

```
function Index (Height, Weight : Float) return Float is
   ...
   return ...;
end;
```

Вызов функции index в C для автора примет вид:

```
my_index = index(68.0, 168.0);
```

Но по ошибке легко перепутать:

```
my_index = index(168.0, 68.0);
```

в результате вы становитесь очень тощим, но высоким гигантом! (По забавному совпадению оба числа оканчиваются на 68.0).

Такой нездоровой ситуации можно избежать в Аде, используя именованное сопоставление параметров в вызове:

```
My_Index := Index (Height => 68.0, Weight => 168.0);
```

Опять-таки, мы можем перечислять параметры в любом порядке, в каком пожелаем, ничего не случится, если мы забудем, в каком порядке они следуют в определении функции.

Именованное сопоставление очень ценная конструкция языка Ада. Хотя его использование не является обязательным, им стоит пользоваться как можно чаще, т. к. кроме защиты от ошибок, оно значительно улучшает читаемость программы.

27.4 Целочисленные литералы

Обычно целочисленные литералы не часто встречаются в программе, за исключением может быть 0 и 1. Целочисленные литералы должны в основном использоваться для инициализации констант. Но если они используются, их смысл должен быть очевиден для читателя. Использование подходящего основания исчисления (10, 8, 16 и пр.) и разделители групп цифр помогут избежать ошибок.

В Аде это сделать легко. Ясный синтаксис позволяет указать любое основание исчисления от 2 до 16 (по умолчанию конечно 10), например 16#2B# это целое число 43 по основанию 16. Символ подчеркивания используется для группировки цифр в значениях с большим количеством разрядов. Так, например, 16#FFFF_FFFF_FFFF_FFFF# вполне читаемая запись значения 2^64^-1.

Для сравнения тот же литерал в C (так же как в C++, Java и пр. родственных языках) выглядит как 0xFFFFFFFFFFFFFFFF, о который не трудно сломать глаза, пытаясь сосчитать сколько F оно содержит. Более того, C трактует лидирующие нули особым образом, в результате число 031 означает совсем не то, что поймет любой школьник, а 25.

CHAPTER

TWENTYEIGHT

БЕЗОПАСНЫЕ ТИПЫ ДАННЫХ

В данной главе речь пойдет не об ускорении набора текста программы, а о механизме, помогающем обнаружить еще больше ошибок и опечаток.

Этот специально разработанный и встроенный в язык механизм часто называют механизмом строгой типизации.

В ранних языках, таких как Fortran и Algol, все обрабатываемые данные имели числовой тип. В конце концов, ведь компьютер изначально способен обращаться только с числами, зачастую закодированными в бинарном виде целыми либо числами с плавающей точкой. В более поздних языках, начиная с Pascal, появилась возможность оперировать объектами на более абстрактном уровне. Так, использование перечислимых типов (в Pascal их называют скалярными) дает нам несомненное преимущество обращаться с цветами, как с цветами, хотя они в итоге будут обрабатываться компьютером, как числа.

Эта идея в языке Ада получила дальнейшее развитие, в то время как другие языки продолжают трактовать скалярные типы, как числовые, упуская ключевую идею абстракции, суть которой в разделении смыслового предназначения и машинного представления.

28.1 Использование индивидуальных типов

Допустим, мы наблюдаем за качеством производимой продукции и подсчитываем число испорченных изделий. Для этого мы считаем годные и негодные образцы. Нам нужно остановить производство, если количество негодных образцов привысит некоторый лимит, либо если мы изготовим заданное количество годных изделий. В C и C++ мы могли бы иметь следующие переменные:

```
int badcount, goodcount;
int b_limit, g_limit;
```

и далее:

```
badcount = badcount + 1;
...
if (badcount == b_limit) { ... };
```

Аналогично для годных образцов. Но, т. к. все это целые числа, ничто не помешает нам случайно написать по ошибке:

```
if (goodcount == b_limit) { ... };
```

где нам на самом деле нужно было бы написать g_limit. Это может быть результатом приема «скопировать и вставить», либо просто опечатка (b и g находятся близко на клавиатуре). Как бы то ни было, компилятор будет доволен, а мы — едва ли.

Так может случиться в любом языке. Но в Аде есть способ выразить наши действия более точно. Мы можем написать:

```
type Goods is new Integer;
type Bads is new Integer;
```

Эти определения вводят новые типы с теми же характеристиками, что и предопределенный тип Integer (например будут иметь операции + и -). Хотя реализация этих типов ничем не отличается, сами же типы различны. Теперь мы можем написать:

```
Good_Count, G_Limit : Goods;
Bad_Count, B_Limit : Bads;
```

Разделив таким образом понятия на две группы, мы достигнем того, что компилятор обнаружит ошибки, когда мы перепутаем сущности разных групп и предотвратит запуск некорректной программы. Следующий код не вызовет нареканий:

```
Bad_Count := Bad_Count + 1;
if Bad_Count = B_Limit then
```

Но следующая ошибка будет обнаружена

```
if Good_Count = B_Limit then --  Illegal
```

ввиду несовпадения типов.

Когда нам потребуется действительно смешать типы, например, чтобы сравнить количество годных и негодных образцов, мы можем использовать преобразование типов. Например:

```
if Good_Count = Goods (B_Limit) then
```

Другим примером может служить вычисление разницы счетчиков образцов:

```
Diff : Integer := Integer (Good_Count) - Integer (Bad_Count);
```

Аналогично можно избежать путаницы с типами с плавающей точкой. Например, имея дело с весом и ростом, как в примере из предыдущей главы, вместо того, чтобы писать:

```
My_Height, My_Weight : Float;
```

было бы лучше написать:

```
type Inches is new Float;
type Pounds is new Float;
My_Height : Inches := 68.0;
My_Weight : Pounds := 168.0;
```

И это позволит компилятору предотвратить путаницу.

28.2 Перечисления и целые

В главе «Безопасный синтаксис» мы обсуждали пример с железной дорогой, в котором была следующая проверка:

```
if (the_signal == clean) { ... }
```

```
if The_Signal = Clean then ... end if;
```

на языке C и Ада соответственно. На C переменная the_signal и соответствующие константы могут быть определены так:

```
enum signal {
  danger,
  caution,
  clear
};
enum signal the_signal;
```

Эта удобная запись на самом деле всего лишь сокращение для описания констант danger, caution и clear типа int. И сама переменная the_signal имеет тип int. Как следствие, ничего не помешает нам присвоить переменной the_signal абсолютно бессмысленное значение, например 4. В частности, такие значения могут возникать при использовании не инициализированных переменных. Хуже того, если мы в другой части программы оперируем химическими элементами и используем имена anion, cation, нам ничего не помешает перепутать cation и caution. Мы можем также использовать где-то женские имена betty и clare, либо названия оружия dagger и spear. И снова ничто не предотвратит опечатки типа dagger вместо danger и clare вместо clear.

В Аде мы пишем:

```
type Signal is (Danger, Caution, Clear);
The_Signal : Signal := Danger;
```

и путаница исключена, потому, что перечислимый тип в Аде - это совершенно отдельный тип и он не имеет отношения к целому типу. Если мы также где-то имеем:

```
type Ions is (Anion, Caution);
type Names is (Anne, Betty, Clare, ...);
type Weapons is (Arrow, Bow, Dagger, Spear);
```

то компилятор предотвратит путаницу этих понятий в момент компиляции. Более того, компилятор не даст присвоить Clear значение Danger, так как оба они литералы и присваивание для них также бессмысленно, как попытка поменять значение литерала 5 написав:

```
5 := 2 + 2;
```

На машинном уровне все перечисленные типы кодируются как целые и мы можем получить код для кодировки по умолчанию, используя атрибут Pos, когда нам это действительно необходимо:

```
Danger_Code : Integer := Signal'Pos (Danger);
```

Мы также можем ввести свою кодировку. Позже мы остановимся на этом в главе «Безопасная коммуникация».

Между прочим, один из важнейших типов Ады, Boolean, имеет следующее определение:

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

Результат операций сравнения, таких как The_Signal = Clear имеет тип Boolean. Также существуют предопределенные операции and, or, not для этого типа. В Аде невозможно использовать числовые значение вместо Boolean и наоборот. В то время, как в C, как мы помним, результат сравнения вернет значение целого типа, ноль означает ложно, а не нулевое значение — истинно. Снова возникает опасность в коде:

```
if (the_signal == clean) { ... }
```

Как ранее упоминалось, пропустив один знак равенства, вместо выражения сравнения мы получим присваивание. Поскольку целочисленный результат воспринимается в C как значение условия, ошибка остается необнаруженной. Таким образом, данный пример иллюстрирует сразу несколько возможных мест для ошибки:

- использование знака = для присваивания;
- допустимость присваивания там, где ожидается выражение;
- использование числовых значений вместо Boolean в условных выражениях.

Большинство из этого просочилось в C++. Ни одного пункта не присутствует в Аде.

28.3 Ограничения и подтипы

Довольно часто значение некоторой переменной должно находится в определенном диапазоне, чтобы иметь какой-то смысл. Хорошо бы иметь возможность обозначить это в программе, выразив, таким образом, наши представления об ограничениях окружающего мира в явном виде. Например, мой вес My_Weight не может быть меньше нуля и, я искренне надеюсь, никогда не привысит 300 фунтов. Таким образом мы можем определить:

```
My_Weight : Float range 0.0 .. 300.0;
```

а если мы продвинутые программисты и заранее определили тип Phounds, то:

```
My_Weight : Phounds range 0.0 .. 300.0;
```

Далее, если программа ошибочно рассчитает вес, не вписывающийся в заданный диапазон, и попытается присвоить его переменной My_Weight так:

```
My_Weight := Compute_Weight (...);
```

то во время исполнения будет возбуждено исключение Constraint_Error. Мы можем перехватить это исключение в каком-то месте программы и предпринять некоторые корректирующие действия. Если мы не сделаем этого, программа завершиться, выдав диагностическое сообщение с указанием, где произошло нарушение ограничения. Все это происходит автоматически — необходимые проверки компилятор сам вставит во всех нужных местах. (Придирчивый читатель, знакомый с языком Ада, заметит, что наша формулировка «программа завершится» нуждается в уточнении, поскольку верна только для последовательных программ. Ситуация в параллельном программировании несколько отличается от описанной, но это выходит за границы темы, обсуждаемой в этой главе.)

Идея ввести поддиапазоны впервые появилась в Pascal и была далее развита в Аде. Она не доступна в других языках, где нам бы пришлось повсюду вписывать свои проверки, и вряд ли мы бы стали этим озадачиваться. Как следствие, любые ошибки, приводящие к нарушению этих границ, становится обнаружить гораздо труднее. Если бы мы знали, что любое значения веса, которым оперирует программа, находится в указанном диапазоне, то вместо того, чтобы добавлять ограничение в определение каждой переменной, мы могли бы наложить его прямо на тип Pounds:

```
type Pounds is new Float range 0.0 .. 300.0;
```

С другой стороны, если некоторые значения веса в программе неограничены, а известно лишь, что значение веса человека находится в указанном диапазоне, то мы можем написать:

```
type Pounds is new Float;
subtype People_Pounds is Pounds range 0.0 .. 300.0;

My_Weight :  People_Pounds;
```

Аналогично мы можем накладывать ограничения на целочисленные и перечислимые типы. Так, при подсчете образцов мы подразумеваем, что их количество не будет меньше нуля или больше 1000. Тогда мы напишем:

```
type Goods is new Integer range 0 .. 1000;
```

Но если мы хотим лишь убедиться, что значения неотрицательные и не хотим накладывать ограничение на верхнюю границу диапазона, то мы напишем:

```
type Goods is new Integer range 0 .. Integer'Last;
```

где Integer'Last обозначает наибольшее значение типа Integer. Подмножества положительных и неотрицательных целых чисел используются повсеместно, поэтому для них Ада предоставляет стандартные подтипы:

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

Можно определить тип Goods:

```
type Goods is new Natural;
```

где ограничена только нижняя граница диапазона, что нам и нужно.

Пример ограничения для перечислимого типа может быть следующим:

```
type Day is (Monday, Tuesday, Wednesday, Thursday,
             Friday, Saturday, Sunday);
subtype Weekday is Day range Monday .. Friday;
```

Далее автоматические проверки предотвратят присваивание Sunday переменным типа Weekday.

Введение ограничений, подобных описанным выше, может показаться утомительным занятием, но это делает программу более понятной. Более того, это позволяет во время компиляции и во время исполнения убедиться, что наши предположения, выраженные в коде, действительно верны.

28.4 Предикаты подтипов

Подтипы в Аде очень полезны, они позволяют заранее обнаружить такие ошибки, которые в других языках могут остаться незамеченными и привести затем к краху программы. Но, при всей полезности, механизм подтипов несколько ограничен, т. к. разрешает указывать лишь непрерывные диапазоны значений для числовых и перечислимых типов.

Это подтолкнуло разработчиков языка Ада 2012 ввести предикаты подтипов, которые можно добавлять к определениям типов и подтипов. Как показала практика, необходимо иметь два различных механизма в зависимости от того, является ли предикат статическим или динамическим. Оба используют выражения типа Boolean, но статический предикат разрешает лишь некоторые типы выражений, в то время как динамический применим в более общем случае.

Допустим мы оперируем сезонами года и имеем следующее определение месяцев:

```
type Month is (Jan, Feb, Mar, Apr, May, Jun,
               Jul, Aug, Sep, Oct, Nov, Dec);
```

Мы хотим иметь отдельные подтипы для каждого сезона. Для северного полушария зима включает декабрь, январь и февраль. (С точки зрения солнцестояния и равноденствия зима длиться с 21 декабря по 21 марта, но, как по мне, март больше относится к весне, чем к зиме, а декабрь ближе к зиме, чем к осени.) Поэтому нам нужен подтип, включающий значения Dec , Jan и Feb. Мы не можем воспользоваться ограничением диапазона здесь, но можем использовать статический предикат следующим образом:

```
subtype Winter is Month
  with Static_Predicate => Winter in Dec | Jan | Feb;
```

Это гарантирует, что объекты типа Winter могут содержать только Dec , Jan и Feb. Заметьте, что имя подтипа (Winter) в выражении означает текущее значение подтипа.

Подобная синтаксическая конструкция со словом with введена в Ада 2012 и называется аспектом.

Данный аспект проверяется при инициализации переменной по умолчанию, присваивании, преобразовании типа, передаче параметра и т. д. Если проверка не проходит, то возбуждается исключение Assertion_Error. (Мы можем включить или отключить проверку предиката при помощи pragma Assertion_Policy; включает проверку аргумент с именем Check.)

Если условие проверки не является статическим, то необходимо использовать Dynamic_Predicatre аспект. Например:

```
type T is ...;
function Is_Good (X : T) return Boolean;
subtype Good_T is T
  with Dynamic_Predicate => Is_Good (Good_T);
```

Заметьте, что подтип с предикатом невозможно использовать в некоторых ситуациях, таких как ограничения индекса. Это позволяет избежать таких странных вещей, как массивы «с дырками». Однако подтипы со статическим предикатом можно использовать в for-циклах для перебора всех значений подтипа. Т.е. мы можем написать:

```
for M in Winter loop ...
```

В цикле M получит последовательно значения Jan, Feb, Dec, т. е. по порядку определения литералов перечислимого типа.

28.5 Массивы и ограничения

Массив - это множество элементов с доступом по индексу. Предположим, что у нас есть пара игральных костей, и мы хотим подсчитать, сколько раз выпало каждое возможное значение (от 2 до 12). Так как всего возможных значений 11, на C мы бы написали:

```
int counters[11];
int throw;
```

объявив таким образом 11 переменных с именами от counters[0] до counters[10] и целочисленную переменную throw.

При подсчете очередного значения мы бы написали:

```
throw = ...;
counters[throw-2] = counters[throw-2] + 1;
```

Заметьте, что нам пришлось уменьшить значение на 2, т. к. индексы массивов в C всегда отсчитываются от нуля (иначе говоря, нижняя граница массива — всегда ноль). Предположим, что-то пошло не так (или какой-то шутник подсунул нам кость с 7 точками, или используемый нами генератор случайных чисел был неправильно написан) и throw стало равно 13. Что произойдет? Программа на C не обнаружит ошибку. Просто высчитает, где могло бы располагаться counters[11] и прибавит туда единицу. Вероятнее всего, будет увеличено значение переменной throw, т. к. она объявлена сразу после массива. Дальше все пойдет непредсказуемо.

Этот пример демонстрирует печально известную проблему переполнения буфера. Она является причиной множества серьезных и трудно обнаруживаемых неисправностей. В конечном счете это может привести к появлению бреши в защите через которую проходят атаки вирусов на системы, такие как Windows. Мы обсудим это подробнее в главе 7 Безопасное управление памятью.

Давайте теперь рассмотрим аналогичную программу на Аде:

```
Counters : array (2 .. 12) of Integer;
Throw : Integer;
затем:
Throw := ...;
Counters (Throw) := Counters (Throw) + 1;
```

Во время исполнения программы на Аде выполняются проверки, запрещающие нам читать/писать элементы за границами массива, поэтому если Throw случайно станет 13, то будет возбуждено исключение Constraint_Error и мы избежим непредсказуемого поведения программы.

Заметим, что в Аде можно определить не только верхнюю, но и нижнюю границу массива. Нет нужды отсчитывать элементы от нуля. Массивы в реальных программах чаще имеют нижнюю границу равную единице, чем нулю. Задав нижнюю границу массива равную двум, мы получаем возможность в качестве индекса использовать непосредственно значение переменной Throw без необходимости вычитать соответствующее смещение, как в C версии.

Настоящая ошибка данной программы случается не в тот момент, когда происходит выход за пределы массива, а когда Throw выходит за корректный диапазон значений. Эту ситуацию можно выявить раньше, если наложить ограничение на Throw:

```
Throw : Integer range 2 .. 12;
```

и теперь исключение Constraint_Error будет возбуждено в момент, когда Throw станет 13. Как следствие, компилятор будет в состоянии определить, что значение Throw всегда попадает в границы массива и соответствующие проверки при доступе к массиву не нужны там, где в качестве индекса используется Throw. В итоге, мы можем избавиться от дублирования кода, написав:

```
subtype Dice_Range is Integer range 2 .. 12;
Throw : Dice_Range;
Counters : array (Dice_Range) of Integer;
```

Преимущество в том, что если нам в дальнейшем нужно будет поменять диапазон (например, добавив третью кость мы получим значения в диапазоне 3 .. 18), то это нужно будет сделать только в одном месте.

Значение проверок диапазона во время тестирования огромно. Но для программ в промышленной эксплуатации, при желании, эти проверки можно отключить. Подобные проверки применяются не только в Аде. Еще в 1962 компилятор Whetstone Algol 60 мог делать так же. Проверки диапазона определены в стандарте языка (как и в Java, C#).

Наверное стоит упомянуть, что мы можем давать имена и для типов-массивов. Их называют индексируемыми типами. Если у нас есть несколько множеств счетчиков, то будет лучше написать:

```
type Counter_Array is array (Dice_Range) of Integer;
Counters : Counter_Array;
Old_Counter : Counter_Array;
```

и затем, когда нам потребуется скопировать все элементы массива Counters в соответствующие элементы массива Old_Counters, мы просто напишем:

```
Old_Counters := Counters;
```

Именованные индексируемые типы есть далеко не во всех языках. Преимущество именованных типов в том, что они вводят явную абстракцию, как в примере с подсчетом годных и негодных образцов. Чем больше мы даем компилятору информации о том, что мы хотим сделать, тем больше у него возможностей проверить, что наша программа имеет смысл.

Все объекты типа Counter_Array имеют равное количество элементов, определенное типом Dice_Range. Соответственно такой тип называется ограниченным индексируемым типом. Иногда удобнее определить более гибкий тип для объектов, имеющих одинаковый тип индекса и тип элементов, но различное количество элементов. К примеру:

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

Тип Float_Array назвается неограниченным индексируемым типом. При создании объекта такого типа необходимо указать нижнюю и верхнюю границы при помощи ограничения либо задав начальное значение массива.

```
My_Array : Float_Array (1 .. N);
```

Любознательный читатель может спросить, что будет если верхняя граница меньше нижней, например, если N равно 0. Это вполне допустимо и приведет к созданию пустого массива. Интересно то, что верхняя граница может быть меньше чем нижняя граница подтипа индекса.

Неограниченные индексируемые типы очень полезны для аргументов, т. к. позволяют писать подпрограммы, обрабатывающие массивы любого размера. Мы рассмотрим примеры позже.

28.6 Установка начальных значений по умолчанию

Для устойчивой работы предикатов подтипа (а также инвариантов типа, как мы увидим далее) может потребоваться, чтобы объект при создании имел осмысленное начальное значение. Изначально язык Ада предлагал лишь частичное решение этого вопроса. Для значений ссылочных типов («указателей») гарантированно начальное значение в виде null. Для записей (record) программист может определить значение по умолчанию следующим образом:

```
type Font is (Arial, Bookman, Times_New_Roman);
type Size is range 1 .. 100;

type Formatted_Character is record
   C : Character;
   F : Font := Times_New_Roman;
   S : Size := 12;
end record;

FC : Formatted_Character;
-- Здесь FC.F = Times_New_Roman, FC.S = 12
-- FC.C не инициализировано
```

К начальным значениям можно относиться по-разному. Есть мнение, что иметь начальные значения (например, ноль) плохо, поскольку это может затруднить поиск плавающих ошибок. Контраргумент заключается в том, что это дает нам уверенность, что объект имеет согласованное начальное состояние, что может помочь предотвратить разного рода уязвимости.

Как бы то ни было, это довольно странно, что в ранних версиях Ады можно было задать значения по умолчанию для компонент записи, но нельзя было — для скалярных типов или

массивов. В версии Ада 2012 это было исправлено при помощи аспектов Default_Value и Default_Component_Value. Новая версия предыдущего примера может выглядеть так:

```
type Font is (Arial, Bookman, Times_New_Roman)
  with Default_Value => Times_New_Roman;
type Size is range 1 .. 100
  with Default_Value => 12;
```

При таком объявлении типов мы можем опустить начальные значения для компонент Formatted_Character:

```
type Formatted_Character is record
   C : Character;
   F : Font;    --  Times_New_Roman по умолчанию
   S : Size;    --  12 по умолчанию
end record;
```

Для массива можно указать значение по умолчанию для его компонент:

```
type Text is new String
  with Default_Component_Value =>
    Ada.Characters.Latin_1.Space;
```

Следует заметить, что в отличии от начальных значений компонент записи, здесь используются только статические значения.

28.7 «Вещественные ошибки»

Название этого раздела — дословный перевод термина real errors, обозначающего ошибки округления при работе с вещественными числами. В оригинале используется как каламбур.

Для операций над числами с плавающей точкой (используя такие типы, как real в Pascal, float в C и Float в Аде) используется отдельные вычислительные устройства процессора. При этом, само представление числа имеет относительную точность. Так в 32 разрядном слове под мантиссу может быть выделено 23 бита, один бит под знак и 8 бит под экспоненту. Это дает точность 23 двоичных цифры, т. е. примерно 7 десятичных.

При этом для больших чисел, таких как 123456.7 точность будет в одну десятую, а для маленьких, как 0.01234567 — восемь знаков после запятой, но в любом случае число значимых цифр всегда остается 7. Другими словами, точность связана с величиной значения.

Относительная точность подходит во многих случаях, но не во всех. Возьмем к примеру представление угла направления траектории корабля или ракеты. Допустим, мы хотим иметь точность в одну секунду. Полная окружность включает в себя 360 градусов, в каждом градусе 60 минут, в каждой минуте 60 секунд.

Если мы храним угол, как число с плавающей точкой:

```
float bearing;
```

тогда для значения 360 градусов точность будет примерно 8 секунд, что недостаточно, в то время как для 1 градуса — точность 1/45 секунды, что излишне. Мы могли бы хранить значение, как целое число секунд, используя целочисленный тип:

```
int bearingsecs;
```

Это бы сработало, но нам пришлось бы не забывать выполнять соответствующее масштабирование каждый раз при вводе и отображении значения.

Однако, настоящая проблема чисел с плавающей точкой в том, что точность операций, таких как сложение и вычитание, страдает от ошибок округления. Если мы находим разницу чисел приблизительно одной величины, мы получим существенную потерю точности. К тому же некоторые числа не имеют точного представления. К примеру, у нас есть шаговый двигатель с шагом 1/10 градуса. Мы отмеряем 10 шагов. Но так как 0.1 не имеет точного представления в двоичной форме, в результате мы никогда не получим ровно один градус. Таким образом, даже когда нам не требуется высокая точность, а точность используемого типа больше требуемой, суммарный эффект множества небольших вычислительных погрешностей может быть неограничен.

Ручное масштабирование для использования целочисленных типов допустимо в простых приложениях, но когда у нас несколько таких типов и нам приходится оперировать ими одновременно начинаются проблемы. Ситуация еще более усложняется, если применять для масштабирования более быстрые операции сдвига. Сложность результирующего кода легко может стать причиной ошибок и затрудняет поддержку. Ада среди тех немногих языков, которые предоставляют арифметику с фиксированной точкой. По своей сути, это автоматический вариант масштабирования целых чисел. Так, для шагового мотора мы могли бы определить:

```
type Angle is delata 0.1 range -360.0 .. 360.0;
for Angle'Small use 0.1;
```

Результат будет представлен в виде масштабированных (с коэффициентом 0.1) значений, хранимых в виде целых чисел. Но нам удобней думать о них как о соответствующих абстрактных величинах, таких как градусы и их десятые доли. Такая арифметика не страдает от ошибок округления.

Таким образом, Ада имеет две формы для вещественной арифметики:

- числа с плавающей точкой имеющие относительную погрешность;
- числа с фиксированной точкой, имеющие абсолютную погрешность.

Также поддерживается разновидность чисел с фиксированной точкой для десятичной арифметики — стандартная модель для финансовых расчетов.

Тема этого раздела довольно узкоспециализированная, но она иллюстрирует размах возможностей языка Ада и особое внимание к поддержке безопасных численных вычислений.

CHAPTER

TWENTYNINE

БЕЗОПАСНЫЕ УКАЗАТЕЛИ

Первобытный человек совершил гигантский шаг, открыв огонь. Он не только обеспечил себя теплом и едой, но также открыл себе путь к металлическим орудиям труда и далее к индустриальному обществу. Но огонь опасен при небрежном с ним обращении и может стать причиной ужасных бедствий. Для борьбы с диким огнем созданы специальные службы, что служит лишним доказательством серьезности вопроса.

Аналогично, гигантский шаг в разработке программного обеспечения произошел при изобретении понятия указателя или ссылки. Но небрежное обращение с указателями сродни игре с огнем. Указатели приносят неоспоримые преимущества, но при небрежном обращении немедленно следует катастрофа, например, «синий экран смерти», безвозвратная потеря данных, либо брешь в защите, через которую проникают вирусы.

Обычно в программном обеспечении высокой надежности использование указателей существенно ограничено. Ссылочные типы в Аде сочетают семантику указателей со множеством дополнительных ограничений в использовании. Это делает их пригодными для использования повсеместно, быть может, за редким исключением наиболее требовательных к безопасности областей.

29.1 Ссылки, указатели и адреса

Вместе с указателями появляются несколько возможностей совершить ошибки, такие как:

- Нарушение типа — создать объект одного типа и получить доступ к нему (через указатель) как если бы он был другого типа. В более общем виде, используя указатель, обратиться к объекту таким способом, что нарушается согласованность с семантическими свойствами этого объекта (например, присвоить значение константе или обойти ограничения диапазона).
- Висячие ссылки — доступ к объекту через указатель после того, как объект перестал существовать Это может быть указатель на локальную переменную после выхода из подпрограммы или на динамически созданный объект, уничтоженный затем через другой указатель.
- Исчерпание свободного пространства — ошибка создания объекта вследствие нехватки ресурсов. Что в свою очередь может быть результатом следующего:
- Созданные объекты становятся недоступными («мусором») и никогда не освобождаются;
- Фрагментация «кучи», когда суммарное количество свободного пространства достаточно, но нет ни одного непрерывного участка нужного размера;
- Необходимый размер «кучи» был недооценен;
- Постоянная утечка (все созданные объекты доступны, но создаются бесконечно, например объекты создаются и добавляются в связный список в бесконечном цикле).

Хотя детали разнятся, нарушения типов и висячие ссылки возможны также и в языках, где есть указатели на подпрограммы.

Исторически в языках применялись различные подходы для решения этих проблем. Ранние языки, такие как Fortran, COBOL и Algol 60 не предоставляли пользователю такое понятие как указатель вообще. Программы на всех языках используют адреса в базовых операциях, таких как вызов подпрограммы, но пользователи этих языков не могли оперировать адресами напрямую.

C (и C++) предоставляют указатели как на объекты созданные в «куче», так и на объекты в стеке, а также на функции. Хотя некоторые проверки все же присутствуют, в основном программист сам должен следить за корректным использованием указателей. Например, т. к. C представляет массив, как указатель на первый элемент и разрешает арифметику над указателями, программист может легко себе создать проблему.

Java и другие «чисто» объектно-ориентированные языки не раскрывают существование указателей приложению, но используют указатели и динамическое создание объектов как базовые понятия языка. Проверка типов сохраняется, и удается избежать висящих ссылок (т. к. нет способа явно вызвать free). Чтобы предотвратить исчерпание «кучи» более недоступными объектами, вводится автоматическая сборка «мусора». Это приемлемо для некоторого класса программ. Но довольно спорно в программах реального времени, особенно в областях требующих высокой надежности и безопасности.

Следует заметить, что «сборка мусора» сама по себе не может защитить от исчерпания свободного пространства: программа добавляющая объекты в связный список в бесконечном цикле в конце концов истратит всю память несмотря на все усилия сборщика мусора. (Бесконечный цикл не обязательно является ошибкой в программе, системы управления процессом и им подобные часто пишутся как программы не имеющие завершения. Для остановки такой программы требуются внешнее влияние, например, чтобы оператор повернул тумблер питания).

Сама история указателей в Аде довольно интересна. Изначально, в версии Ада 83, были доступны только указатели на динамически созданные объекты (т.е. не было указателей на объявленные объекты и на подпрограммы). Также предлагалась явная операция освобождения объектов (Unchecked_Deallocation). Таким образом предотвращалось нарушение типизации и появление висящих указателей на более недоступные локальные объекты. Вместе с тем, оставалась возможность получить висящий указатель при некорректном использовании Unchecked_Deallocation.

Введение Unchecked_Deallocation было неизбежно, так как единственная альтернатива — требовать реализациям предоставлять сборщик мусора — не вписывается в областях вычислений в реальном времени и высокой надежности, где ожидалось широкое распространение языка. Философия Ады такова - любое небезопасное действие должно быть четко обозначено. В самом деле, если мы используем Unchecked_Deallocation, нам необходимо указать его в спецификаторах использования (with Ada.Unchecked_Deallocation;), а затем настроить на нужный тип. (Концепции спецификаторов использования и настраиваемых модулей будет рассмотрена в следующем разделе). Такой сравнительно утяжеленный синтаксис одновременно предотвращает безалаберное использование и облегчает чтение и сопровождение кода, четко выделяя опасные места.

Ада 95 расширяет версию Ада 83 и разрешает иметь указатели на объявленные объекты и на подпрограммы. Версия Ада 2005 идет немного дальше, облегчая передачу указателя на подпрограмму в качестве параметра. Как при этом удается сохранить безопасность мы и рассмотрим в этой главе.

Еще одно замечание до того, как мы углубимся в детали. Поскольку термин указатель часто несет дополнительный низкоуровневый подтекст, в Аде используется термин ссылочный тип. Таким образом, делается акцент на том, что значения ссылочного типа позволяют нам ссылаться на другие объекты некоторого заранее известного типа (и являются своего рода динамическими именами этих объектов) и не должны восприниматься просто как

машинный адрес. В самом деле, на уровне реализации представление этих значений могут отличаться от физических указателей.

29.2 Ссылочные типы и строгая типизация

Используя возможности языка Ада 2005 мы можем объявить переменную Ref, чьи значения предоставляют доступ к объектам типа T:

```
Ref : access T;
```

Если мы не укажем начальное значение, то будет использовано специальное значение null. Ref может ссылаться на обычную переменную типа T (которая, однако, должна быть помечена как aliased):

```
Obj : aliased T;
...
Ref := Obj'Access;
```

Это аналогично следующей записи на языке C:

```
t* ref;
t obj;
ref = &obj;
```

Тип T в свою очередь может быть определен как:

```
type Date is record
   Day : Integer range 1 .. 31;
   Month : Integer range 1 .. 12;
   Year : Integer;
end record;
```

и далее мы можем написать:

```
Birthday: aliased Date :=(Day => 10, Month => 12, Year => 1815);
AD : access Date := Birthday'Access;
```

Можно обратиться к отдельным компонентам даты, используя AD:

```
The_Day : Integer := AD.Day;
```

Переменная AD также может ссылаться на динамически созданный объект, расположенный «в куче» (которая в Аде носит названия пул /storage pool/).

```
AD := new Date'(Day => 27, Month => 11, Year => 1852);
```

(Здесь использованы даты рождения и смерти графини Ады Лавлейс, в честь которой назван язык).

Типичным случаем использования ссылочных типов является связный список. Например, мы можем определить:

```
type Cell is record
   Next : access Cell;
   Value : Integer;
end record;
```

и затем мы сможем создать цепочку объектов типа Cell, связанных в список.

Часто удобно иметь именованный ссылочный тип:

```
type Date_Ptr is access all Date;
```

Здесь слово all означает, что этот тип служит для работы как с динамическими созданными объектами, так и с объявленными переменными, расположенными в стеке (которые помечены aliased).

Сама пометка aliased — весьма полезная «страховка». Она предупреждает программиста, что к объекту можно обратиться через ссылку (что помогает при беглом ознакомлении). Кроме того — это сигнал компилятору, что при оптимизации кода нужно учитывать возможность косвенного обращения к объекту через ссылочные значения.

Важным моментом является то, что ссылочный тип всегда связан с типом объектов, на которые он ссылается. Таким образом, всегда выполняется контроль типов при присваивании, передаче параметров и во всех других вариантах использования. Кроме того, ссылочное значение всегда имеет только легальные значения (в числе которых и null). При исполнении программы, при попытке доступа к объекту по ссылке Date_Ptr выполняется проверка, что значение не равно null и возбуждается исключение Constraint_Error, если это не так.

Мы можем явно задать, что ссылочное значение всегда будет отличным от null, записав определение переменной следующим образом:

```
WD : not null access Date := Wedding_Day'Access;
```

Естественно, сразу нужно указать начальное значение отличное от null. Использование этой возможности позволяет гарантировать, что упомянутое выше исключение никогда не произойдет.

Наконец, следует отметить, что ссылочное значение может указывать на компоненту составного типа, при условии, что сама компонента помечена как aliased. Например:

```
A : array (1..10) of aliased Integer := (1,2,3,4,5,6,7,8,9,10);
P : access Integer := A (4)'Access;
```

Но мы не можем использовать арифметику над ссылочным значением P, такую как P++ или P+1, чтобы обратиться к A (5), как это возможно в C. (На самом деле в Аде нет даже такого оператора, как ++.) Хорошо известно, что подобные действия в C легко приводят к ошибкам, т. к. ничто не мешает нам выйти за границы массива.

29.3 Ссылочные типы и контроль доступности

Только что мы рассмотрели, как строгая типизация в языке предотвращает доступ по ссылочному значению к объекту неправильного типа. Следующее требование — убедиться, что объект не прекратит свое существование, пока какой-либо объект ссылочного типа указывает на него. Для случая с определениями объектов это достигается механизмом контроля доступности (accessibility). Рассмотрим следующий код:

```
package Data is
   type Int_Ref is access all Integer;
   Ref1 : Int_Ref;
end Data;

with Data; use Data;

procedure P is
   K : aliased Integer;
   Ref2 : Int_Ref;
begin
```

(continues on next page)

(continued from previous page)

```
   Ref2 := K'Access;  --  Illegal
   Ref1 := Ref2;
end P;
```

Хотя это довольно искусственный пример, он в сжатом объеме демонстрирует основные интересные нам места. Пакет Data предоставляет ссылочный тип Int_Ref и объект Ref1 этого типа. Процедура P объявляет локальную переменную K и локальную ссылочную переменную Ref2 также типа Int_Ref. Затем делается попытка присвоить переменной Ref2 ссылку на K. Это запрещено. В самом присвоении Ref2 этого значения проблемы нет потому, что оба объекта (K и Ref2) прекратят свое существование одновременно при завершении вызова процедуры. Настоящая опасность в том, что в дальнейшем мы можем скопировать значение Ref2 в глобальную переменную, в данном случае Ref1, которая будет хранить ссылку на K даже после того, как K перестанет существовать.

Главное правило таково, что время жизни объекта, на который мы получаем ссылку (такого как K) должно быть как минимум такое же, как время жизни указанного ссылочного типа (в нашем случае Int_Ref). В нашем примере это не выполняется, поэтому попытка получить значение, ссылающееся на K незаконна.

Соответствующие правила сформулированы в терминах уровней доступности (accessibility levels), обозначающих вложенность конструкций, охватывающих данное определение. Таким образом, обозначенные правила опираются в основном на статические конструкции языка, проверяются компилятором в момент компиляции и не влекут дополнительных издержек в момент исполнения. Однако правила для параметров подпрограмм, имеющих анонимные ссылочные типы, имеют динамическую природу и проверяются в момент исполнения. Это плата за дополнительную гибкость, которую другим способом достичь не удается.

В столь коротком обзоре языка как этот, у нас нет возможности еще больше углубиться в детали. Достаточно сказать, что правила контроля доступности предотвращают висящие ссылки на объявляемые объекты, которые являются источником множества коварных и трудно устранимых ошибок в других «дырявых» языках.

29.4 Ссылки на подпрограммы

В Аде разрешены ссылки на процедуры и функции. Работают они аналогично ссылкам на объекты. Соответственно для них также выполняются проверки строгой типизации и контроль доступности. Например, используя возможности Ады 2005, мы можем написать:

```
A_Func : access function (X : Float) return Float;
```

После этого объект A_Func может хранить ссылки только на функции с параметром типа Float и возвращающие Float (такова, к примеру, предопределенная функция Sqrt).

И так мы можем написать:

```
A_Func := Sqrt'Access;
...
X : Float := A_Func (4.0);  --  косвенный вызов
```

Это приведет к вызову функции Sqrt с аргументом 4.0, а результат, видимо, будет 2.0.

Язык тщательно следит за совпадением параметров, поэтому мы не можем вызвать функцию с неверным количеством/типом параметров. Это же верно и для результата функции. Список параметров и результат функции составляют отдельное понятие, техническое название которого — профиль функции.

Теперь рассмотрим предопределенную функцию вычисления арктангенса Arctan. Она имеет два параметра и возвращает угол θ (в радианах) такой, что tan θ = Y/X.

```
function Arctan (X : Float; Y : Float) return Float;
```

Если мы напишем:

```
A_Func := Arctan'Access;  -- Ошибка
Z := A_Func (A);  --  косвенный вызов предотвращен
```

Компилятор отвергнет такой код потому, что профиль функции Arctan не совпадает с профилем A_Func. Это как раз то, что нужно, иначе функция Arctan достала бы два аргумента из стека, в то время как косвенный вызов по ссылке A_Func положит в стек лишь один аргумент (Это, на самом деле, зависит от ABI аппаратной платформы, не факт, что floats будут передаваться через стек). Результат получился бы совершенно бессмысленным.

Соответствующие проверки выполняются независимо от пересечения границ модулей компиляции (модули компиляции — программные единицы, компилируемые раздельно, мы остановимся на этом в главе «Безопасная архитектура»). Аналогичные проверки в C не работают в этом случае, что часто приводит к серьезным проблемам.

Более сложный случай возникает, когда одна подпрограмма передается в другую как параметр. Допустим, у нас есть функция для решения уравнения Fn(X) = 0, причем функция Fn сама передается как параметр:

```
function Solve (Trial : Float;
                Accuracy : Float;
                Fn : access function (X : Float) return Float)
   return Float;
```

Параметр Trial — это первое приближение, параметр Accuracy — требуемая точность, третий параметр Fn — функция из уравнения.

К примеру, мы инвестировали 1000 долларов сегодня и 500 долларов в течении года, какова должна быть процентная ставка, чтобы конечная чистая стоимость за два года была в точности 2000 долларов? Задавая процентную ставку как X, мы можем вычислить конечную чистую стоимость по формуле:

```
Nfv (X) = 1000 * (1 + X/100)2 + 500 * (1 + X/100)
```

Чтобы ответить на вопрос, определим функцию, которая вернет 0.0 когда стоимость достигнет 2000.0:

```
function Nvf_2000 (X : Float) return Float is
   Factor : constant Float := 1.0 + X / 100.0;
begin
   return 1000.0 * Factor ** 2 + 500.0 * Factor - 2000.0;
end Nvf_2000;
```

Затем можно сделать:

```
Answer : Float := Solve
   (Trial => 5.0, Accuracy => 0.01, Fn => Nvf_2000'Access);
```

Мы предлагаем решить уравнение, указав первое приближение в 5%, точность 0.01 и целевую функцию Nvf_2000. Предлагаем читателю найти решение, наше значение вы найдете в конце главы. (Термин «конечная чистая стоимость» хорошо известен финансовым специалистам.)

Мы хотели бы отметить, что в Аде будут проверяться типы параметров функции, даже когда она сама передается как параметр, благодаря тому, что профиль функции может иметь произвольную глубину вложенности. Многие языки имеют ограничение в один уровень.

Заметим, что параметр Fn имеет анонимный тип. Аналогично ссылкам на объекты, мы можем определить именованный ссылочный тип для подпрограмм. И можем заставить ссылочный тип хранить только не null значения. Т.е. можно написать:

```
A_Func : not null access function (X : Float) return Float
       := Sqrt'Access;
```

Плюс тут в том, что мы явно гарантируем, что A_Func не равен null в любом месте, где бы мы его не использовали.

В том случае, если Вы считаете использование произвольной функции (в данном случае Sqrt), как начального значения не равного null безвкусицей, можно определить специальную функцию для значения по умолчанию:

```
function Default (X : Float) return Float is
begin
   Put ("Value not set");
   return 0.0;
end Default;
...
A_Func : not null access function (X : Float) return Float
       := Default'Access;
```

Аналогично, нам необходимо добавить not null в профиль функции Solve:

```
function Solve
   (Trial : Float;
    Accuracy : Float;
    Fn : not null access function (X : Float) return Float)
   return Float;
```

Это гарантирует что аргумент Fn никогда не будет null.

29.5 Вложенные подпрограммы в качестве параметров

Как мы упоминали ранее, контроль доступности также работает и для ссылок на подпрограммы. Допустим функция Solve определена в пакете и использует именованный ссылочный тип для параметра Fn:

```
package Algorithms is
   type A_Function is
       not null access function (X : Float) return Float;
   function Solve
      (Trial : Float; Accuracy : Float; Fn : A_Function)
      return Float;
   ...
end Algorithms;
```

Допустим мы хотим обобщить пример с вычислением чистой стоимости, чтобы иметь возможность передавать целевое значение как параметр. Попробуем так:

```
with Algorithms; use Algorithms;

function Compute_Iterest (Target : Float) return Float is
   function Nfv_T (X : Float) return Float is
      Factor : constant Float := 1.0 + X/100.0;
   begin
      return 1000.0*Factor**2 + 500.0*Factor - Target;
   end Nfv_T;
begin
   return Solve (Trial => 5.0, Accuracy => 0.01,
             Fn => Nfv_T'Access);  --  Illegal
end Compute_Iterest;
```

Однако Nfv_T'Access нельзя использовать как значение параметра Fn, потому, что это нарушает правила контроля доступности. Проблема в том, что функция Nfv_T находится на более глубоком уровне вложенности, чем тип A_Function. (Так и должно быть, поскольку нам необходим доступ к параметру Target.) Если бы подобное было разрешено, мы могли бы присвоить это значение какой-нибудь глобальной переменной типа A_Function. После выхода из функции Compute_Interest функция Nfv_T будет недоступна для использования, но глобальная переменная все еще будет хранить ссылку на нее. Например:

```
Dodgy_Fn : A_Function := Default'Access; --  Глобальный объект

function Compute_Iterest (Target : Float) return Float is
   function Nfv_T (X : Float) return Float is
   ...
   end Nfv_T;
begin
   Dodgy_Fn := Nfv_T'Access;  --  Ошибка
   ...
end Compute_Iterest;
```

После завершения вызова мы делаем:

```
Answer := Dodgy_Fn (99.9); --  результат непредсказуем
```

Вызов Dodgy_Fn будет пытаться вызвать Nfv_T, но это невозможно, потому, что она локальна для Compute_Interest и будет пытаться обратиться к параметру Target, которого уже не существует. Если бы Ада не запрещала это делать, последствия были бы непредсказуемы. Заметьте, что при использовании анонимного типа для параметра Fn, мы могли бы передать вложенную функцию, как аргумент Solve, но тогда контроль доступности сработал бы при попытке присвоить это значение переменной Dodgy_Fn. Во время исполнения выполнилась бы проверка, что уровень вложенности Nfv_T больше, чем уровень вложенности типа A_Function и произошло бы возбуждение исключения Program_Error. Таким образом, правильным решением было бы определить:

```
package Algorithms is
   function Solve
      (Trial : Float;
       Accuracy : Float;
       Fn : not null access function (X : Float)
                     return Float)
      return Float;
end Algorithms;
```

и оставить функцию Compute_Interest в первоначальном варианте (удалив комментарий Ошибка, конечно).

Может показаться, что проблема лежит в том, что функция Nfv_T вложена в Compute_Iterest. Мы могли бы сделать ее глобальной и проблемы с доступностью исчезли бы. Но для этого нам пришлось бы передавать значение Target через какую-нибудь переменную в пакете верхнего уровня, в стиле COMMON-блоков языка FORTRAN. Мы не можем добавить ее в список параметров функции, т. к. список параметров должен совпадать со списком для Fn. Но передавать данные через глобальные переменные фактически порочная практика. Она нарушает принцип сокрытия информации, принцип абстракции, а также плохо стыкуется с многозадачностью. Заметим, что практика использования вложенных функций, когда функция получает доступ к нелокальным переменным (таким как Target) часто называется «замыканием».

Такие замыкания, другими словами передача указателя на вложенную подпрограмму как параметр времени исполнения, используются в некоторых местах стандартной библиотеки Ады, например для перебора элементов какого-нибудь контейнера.

Использовать вложенные подпрограммы в таких случаях естественно, т. к. они нуждаются в нелокальных данных. Подобный подход затруднен в «плоских» языках программирования,

таких как C, C++ и Java. В некоторых случаях можно использовать механизм наследования, но он менее ясен, что может повлечь проблемы при сопровождении кода.

Наконец, может потребоваться комбинировать алгоритмы, используя вложенность. Так, наш пакет Algorithms может содержать другие полезные вещи:

```
package Algorithms is
   function Solve
      (Trial : Float;
       Accuracy : Float;
       Fn : not null access function (X : Float)
                    return Float)
      return Float;

   function Integrate
      (Lo, Hi : Float;
       Accuracy : Float;
       Fn : not null access function (X : Float)
                    return Float)
      return Float;

   type Vector is array (Positive range <>) of Float;

   procedure Minimize
      (V : in out Vector; Accuracy : Float;
       Fn : not null access function (V : Vector)
                    return Float)
      return Float;
end Algorithms;
```

Функция Integrate подобна Solve. Она вычисляет определённый интеграл от данной функции на заданном интервале. Процедура Minimize несколько отличается. Она находит те значения элементов массива, на которых данная функция достигает минимума. Возможна ситуация, когда целевая функция минимизации является интегралом и использует V. Кому-то это может показаться притянутым за уши, но автор потратил первые несколько лет своей карьеры программиста, работая над подобными вещами в химической индустрии.

Код может иметь вид:

```
with Algorithms; use Algorithms;
procedure Do_It is
   function Cost (V: Vector) return Float is
      function F (X : Float) return Float is
         Result : Float;
      begin
         ... --  Вычислим Result используя V и X
         return Float;
      end F;
   begin
      return Integrate (0.0, 1.0, 0.01, F'Access);
   end Cost;
   A : Vector (1 .. 10);
begin
   ... -- perhaps read in or set trial values for the vector A
   Minimize (A, 0.01, Cost'Access);
   ... -- output final values of the vector A
end Do_It;
```

В Аде 2005 (и соответственно в Аде 2012) подобный подход срабатывает «на ура», как если бы вы делали это на Algol 60. В других языках подобное сделать трудно, либо требует использования небезопасных конструкций, которые могут привести к появлению висящих ссылок.

Дополнительные примеры использования ссылочных типов для подпрограмм можно найти в главе «Безопасная коммуникация».

И, наконец, искомая процентная ставка при которой 1000 долларов инвестиций с последующими 500 долларами за два года дадут 2000 чистой стоимости равна 18.6%. И это круто!

CHAPTER

THIRTY

БЕЗОПАСНАЯ АРХИТЕКТУРА

Если говорить о строительстве, то хорошей архитектурой считается та, что гарантирует требуемую прочность самым естественным и простейшим путем, предоставляя людям безопасную среду обитания. Красивым примером может служить Пантеон в Риме, чья сферическая форма обладает чрезвычайной прочностью и в тоже время предоставляет максимум свободного пространства. Многие кафедральные соборы не так хороши и нуждаются в дополнительных колоннах снаружи для поддержки стен. В 1624г. сэр Ганри Вутон подытожил эту тему следующими словами: «хорошее строение удовлетворяет трем условиям - удобство, прочность и красота».

Аналогично, хорошая архитектура в программировании должна обеспечивать безопасность функционирования отдельных компонент простейшим образом, при этом сохраняя прозрачность системы в целом. Она должна обеспечивать взаимодействие, где это необходимо, и препятствовать взаимному влиянию действий, не связанных друг с другом. Хороший язык программирования должен позволять писать эстетически красивые программы с хорошей архитектурой.

Возможно, здесь есть аналогия с архитектурой офисных помещений. Если выделить каждому отдельный кабинет, это будет препятствовать общению и обмену идеями. С другой стороны, полностью открытое пространство приведет к тому, что шум и другие отвлекающие факторы будут препятствовать продуктивной работе.

Структура программ на Аде базируется на идее пакетов, которые позволяют сгруппировать взаимосвязанные понятия и предоставляют очевидный способ сокрытия деталей реализации.

30.1 Спецификация и тело пакета

Ранние языки программирования, такие как FORTRAN, имели плоскую структуру, где все располагалось, в основном, на одном уровне. Как следствие, все данные (не считая локальных данных подпрограммы) были видимы всюду. Это похоже на единое открытое пространство в офисе. Похожую плоскую модель предлагает и язык C, хотя и предлагает некоторую дополнительную возможность инкапсуляции, предоставляя программисту возможность контролировать видимость подпрограмм за границами текущего файла.

Другие языки, например Algol и Pascal, имеют простую вложенную блочную структуру, напоминающую матрешку. Это немного лучше, но все равно напоминает единое открытое пространство, поделенное на отдельные мелкие офисы. Проблема взаимодействия все равно остается.

Рассмотрим стек из чисел в качестве простого примера. Мы хотим иметь протокол, позволяющий добавить элемент в стек, используя вызов процедуры Push, и удалить верхний элемент, используя функцию Pop. И, допустим, еще одну процедуру Clear для сброса стека в пустое состояние. Мы намерены предотвратить все другие способы модификации стека, чтобы сделать данный протокол независимым от метода его реализации.

Рассмотрим реализацию, написанную на Pascal-е. Для хранения данных используется массив, а для работы написаны три подпрограммы. Константа max ограничивает максимальный размер стека. Это позволяет нам избежать дублирования числа 100 в нескольких местах, на тот случай, если нам потребуется его изменить.

```
const max = 100;

var    top : 0 .. max;
   a : array [1..max] of real;

procedure Clear;
begin
   top := 0
end;

procedure Push(x : real);
begin
   top := top + 1;
   a[top] := x
end;

function Pop : real;
begin
   top := top - 1;
   Pop := a[top+1]
end;
```

Главная проблема тут в том, что max, top и a должны быть объявлены вне подпрограмм Push, Pop и Clear, чтобы мы могли иметь доступ к ним. Как следствие, в любом месте программы, где мы можем вызвать Push, Pop и Clear, мы также можем непосредственно поменять top и a, обойдя, таким образом протокол использования стека и получить несогласованное состояние стека.

Это может служить источником проблем. Если нам захочется вести учет количества изменений стека, то просто добавить счетчик в процедуры Push, Pop и Clear может оказаться недостаточно. При анализе большой программы, когда нам нужно найти все места, где стек изменялся, нам нужно отследить не только вызовы Push, Pop и Clear, но и обращения к переменным top и a.

Аналогичная проблема существует и в C и в Fortran. Эти языки пытаются ее преодолеть, используя механизм раздельной компиляции. Объекты, видимые из других единиц компиляции, помечаются специальной инструкцией extern либо при помощи заголовочного файла. Однако, проверка типов при пересечении границ модулей компиляции в этих языках работает гораздо хуже.

Язык Ада предлагает использовать механизм пакетов, чтобы защитить данные, используемые в Push, Pop и Clear от доступа извне. Пакет делится на две части — спецификацию, где описывается интерфейс доступный из других модулей, и тело, где располагается реализация. Другими словами, спецификация задает что делать, а тело — как это делать. Спецификация стека могла бы быть следующей:

```
package Stack is
   procedure Clear;
   procedure Push (X : Float);
   function Pop return Float;
end Stack;
```

Здесь описан интерфейс с внешним миром. Т.е. вне пакета доступны лишь три подпрограммы. Этой информации достаточно программисту, чтобы вызывать нужные подпрограммы, и достаточно компилятору, чтобы эти вызовы скомпилировать. Тело пакета может иметь следующий вид:

```
package body Stack is
   Max : constant := 100;
   Top : Integer range 0 .. Max := 0;
   A : array (1 .. Max) of Float;

   procedure Clear is
   begin
      Top := 0;
   end Clear;

   procedure Push (X : Float) is
   begin
      Top := Top + 1;
      A (Top) := X;
   end Push;

   function Pop return Float is
   begin
      Top := Top - 1;
      return A (Top + 1);
   end Pop;

end Stack;
```

Тело содержит полный текст подпрограмм, а также определяет скрытые объекты Max, Top и A. Заметьте, что начальное значение Top равняется нулю.

Для того, чтобы использовать сущности, описанные в пакете, клиентский код должен указать пакет в спецификаторе контекста (с помощью зарезервированного слова with) следующим образом:

```
with Stack;
procedure Some_Client is
   F : Float;
begin
   Stack.Clear;
   Stack.Push (37.4);
   ...
   F := Stack.Pop;
   ...
   Stack.Top := 5; -- Ошибка!
end Some_Client;
```

Теперь мы уверены, что требуемый протокол будет соблюден. Клиент (программный код, использующий пакет) не может ни случайно, ни намеренно взаимодействовать с деталями реализации стека. В частности, прямое присваивание значения Stack.Top запрещено, поскольку переменная Top не видима для клиента (т. к. о ней нет упоминания в спецификации пакета).

Обратите внимание на три составляющие этого примера: спецификацию пакета, тело пакета и клиента.

Существуют важные правила, касающиеся их компиляции. Клиент не может быть скомпилирован пока не будет предоставлена спецификация. Тело также не может быть скомпилировано без спецификации. Но подобных ограничений нет между телом и клиентом. Если мы решим изменить детали реализации и это не затрагивает спецификацию, то в перекомпиляции клиента нет нужды.

Пакеты и подпрограммы верхнего уровня (т. е. не вложенные в другие подпрограммы) могут быть скомпилированы раздельно. Их обычно называют библиотечными модулями и говорят, что они находятся на уровне библиотеки.

Заметим, что пакет Stack упоминается каждый раз при обращении к его сущностям. В

результате в коде клиента наглядно видно, что происходит. Но если постоянное повторение имени пакета напрягает, можно использовать спецификатор использования (с помощью служебного слова use):

```
with Stack; use Stack;
procedure Client is
begin
   Clear;
   Push (37.4);
   ...
end Client;
```

Однако, если используются два пакета, например Stack1 и Stack2, каждый из которых определяют подпрограмму Clear, и мы используем with и use для обоих, то код будет неоднозначным и компилятор не примет его. В этом случае достаточно указать необходимый пакет явно, например Stack2.Clear.

Подытожим вышесказанное. Спецификация определяет контракт между клиентом и пакетом. Тело пакета обязано реализовать спецификацию, а клиент обязан использовать пакет только указанным в спецификации образом. Компилятор проверяет, что эти обязательства выполняются. Мы вернемся к этому принципу позже в данной главе, а также в последней главе, когда рассмотрим поддержку контрактного программирования в Аде 2012 и идеи, лежащие в основе инструментария SPARK соответственно.

Дотошный читатель отметит, что мы полностью игнорировали ситуации переполнения (вызов Push при Top = Max) и исчерпания (вызов Pop при Top = 0) стека. Если одна из подобных неприятностей случиться, сработает проверка диапазона значения Top и будет возбужденно исключение Constraint_Error. Было бы хорошо включить предусловия для вызова подпрограмм Push и Pop в их спецификацию в явном виде. Тогда при использовании пакета программист бы знал, чего ожидать от вызова подпрограмм. Такая возможность появилась в Ада 2012, как часть поддержки технологии контрактного программирования, мы далее обсудим это.

Исключительно важным моментом здесь является то, что в Аде контроль строгой типизации не ограничен границами модулей компиляции. Написана ли программа как один компилируемый модуль или состоит из нескольких модулей, поделенных на разные файлы, производимые проверки будут совпадать в точности.

30.2 Приватные типы

Еще одна возможность пакета позволяет спрятать часть спецификации от клиента. Это делается при помощи так называемой приватной части. В примере выше пакет Stack реализует лишь один стек. Возможно, будет полезнее написать такой пакет, чтобы он позволял определить множество стеков. Чтобы достичь этого, нам нужно ввести понятие стекового типа. Мы могли бы написать

```
package Stacks is
   type Stack is private;
   procedure Clear (S : out Stack);
   procedure Push (S : in out Stack; X : in Float);
   procedure Pop (S : in out Stack; X : out Float);
private
   Max : constant := 100;
   type Vector is array (1 .. 100) of Float;
   type Stack is record
      A : Vector;
      Top : Integer range 0 .. Max := 0;
   end record;
end Stacks;
```

Это очевидное обобщение одно-стековой версии. Но стоит отметить, что в Аде 2012 появляется выбор, описать Pop либо как функцию, возвращающую Float, либо как процедуру с out параметром типа Float. Это возможно т. к., начиная с версии Ада 2012, функции могут иметь out и in out параметры. Несмотря на это, мы последовали традиционным путем и записали Pop как процедуру. По стилю вызовы Pop и Push будут единообразны, и тот факт, что вызов Pop имеет побочный эффект, будет более очевиден.

Тело пакета может выглядеть следующим образом:

```
package body Stacks is

   procedure Clear (S : out Stack) is
   begin
      S.Top := 0;
   end Clear;

   procedure Push (S : in out Stack; X : in Float) is
   begin
      S.Top := S.Top + 1;
      S.A (S.Top) := X;
   end Push;

   --  процедура Pop аналогично

end Stacks;
```

Теперь клиент может определить, сколько угодно стеков и работать с ними независимо:

```
with Stacks; use Stacks;
procedure Main is
   This_Stack : Stack;
   That_Stack : Stack;
begin
   Clear (This_Stack); Clear (That_Stack);
   Push (This_Stack, 37.4);
   ...
end Main;
```

Подробная информация о типе Stack дается в приватной части пакета и, хотя она видима читателю, прямой доступ к ней из кода клиента отсутствует. Таким образом, спецификация логически разделяется на две части — видимую (все что перед private) и приватную.

Изменения приватной части не приводят к необходимости исправлять исходный код клиента, однако модуль клиента должен быть перекомпилирован, поскольку его объектный код может измениться, хотя исходный текст и останется тем же.

Все необходимые перекомпиляции контролируются системой сборки и по желанию выполняются автоматически. Следует подчеркнуть, что это требования спецификации языка Ада, а не просто особенности конкретной реализации. Пользователю никогда не придется решать, нужно ли перекомпилировать модуль, таким образом, нет риска построить программу из несогласованных версий компилируемых модулей. Это большая проблема для языков, у которых нет точного механизма взаимодействия компилятора, системы сборки и редактора связей.

Также отметьте синтаксис указания режима параметров in, out и in out. Мы остановимся на них подробнее в главе «Безопасное создание объектов», где будет описана концепция потоков информации (information/data flow).

30.3 Контрактная модель настраиваемых модулей

Шаблоны — одна из важных возможностей таких языков программирования как C++ (а также недавно Java и C#). Им в Аде соответствуют настраиваемые модули (настраиваемые/обобщенные пакеты, generic packages). На самом деле, при создании шаблонов C++ были использованы идеи настраиваемых модулей Ады. Чтобы обеспечить безопасность типов данных, настраиваемые модули используют так называемую контрактную модель.

Мы можем расширить пример со стеком так, чтобы стало возможно определить стеки для произвольных типов и размеров (позже мы рассмотрим еще один способ сделать это). Рассмотрим следующий код

```
generic
   Max : Integer;
   type Item is private;
package Generic_Stacks is
   type Stack is private;
   procedure Clear (S : out Stack);
   procedure Push (S : in out Stack; X : in Item);
   procedure Pop (S : in out Stack; X : out Item);
private
   type Vector is array (1 .. 100) of Item;
   type Stack is record
      A : Vector;
      Top : Integer range 0 .. Max := 0;
   end record;
end Generic_Stacks;
```

Тело к этому пакету можно получить из предыдущего примера, заменив Float на Item.

Настраиваемый пакет - это просто шаблон. Чтобы использовать его в программе, нужно сначала выполнить его настройку, предоставив два параметра — Max и Item. В результате настройки получается реальный пакет. К примеру, если мы хотим работать со стеками целых чисел с максимальным размером 50 элементов, мы напишем:

```
package Integer_Stacks is new Generic_Stacks
   (Max => 50, Item => Integer);
```

Эта запись определяет пакет Integer_Stacks, который далее можно использовать как обычный. Суть контрактной модели в том, что если мы предоставляем параметры, отвечающие требованиям описания настраиваемого пакета, то при настройке мы гарантированно получим рабочий пакет, компилируемый без ошибок.

Другие языки не предоставляют этой привлекательной возможности. В C++, к примеру, некоторые несоответствия можно выявить только в момент сборки программы, а некоторые и вовсе могут остаться не обнаруженными, пока мы не запустим программу и не получим исключение.

Существуют разнообразные варианты настраиваемых параметров в языке Ада. Использованная ранее форма:

```
type Item is private;
```

позволяет использовать практически любой тип для настройки. Другая форма:

```
type Item is (<>);
```

обозначает любой скалярный тип. Сюда относятся целочисленные типы (такие как Integer и Long_Integer) и перечислимые типы (такие как Signal). Внутри настраиваемого модуля мы можем пользоваться всеми свойствами, общими для этих типов, и, несомненно, любой актуальный тип будет иметь эти свойства.

Контрактная модель настраиваемых модулей очень важна. Она позволяет вести разработку библиотек общего назначения легко и безопасно. Это достигается во многом благодаря тому, что пользователю нет необходимости разбираться с деталями реализации пакета, чтобы определить, что может пойти не так.

30.4 Дочерние модули

Общая архитектура системы (программы) на языке Ада может иметь иерархическую (древовидную) структуру, что облегчает сокрытие информации и упрощает модификацию. Дочерние модули могут быть общедоступными или приватными. Имея пакет с именем Parent мы можем определить общедоступный дочерний пакет следующим образом:

```
package Parent.Child is ...
```

а приватный как:

```
private package Parent.Child is ...
```

Оба варианта могут иметь тело и приватную часть, как обычно. Ключевая разница в том, что общедоступный дочерний модуль, по сути, расширяет спецификацию родителя (и таким образом видим всем клиентам), тогда как приватный модуль расширяет приватную часть и тело родителя (и таким образом не видим клиентам). У дочерних пакетов, в свою очередь, могут быть дочерние пакеты и так далее.

Среди правил, определяющих видимость имен, можно отметить следующее. Дочерний модуль не нуждается в спецификаторе контекста (with Parent), чтобы видеть объекты родителя. В тоже время, тело родителя может иметь спецификатор контекста для дочернего модуля, если нуждается в функциональности, предоставляемой им. Однако, поскольку спецификация родителя должна быть доступна к моменту компиляции дочернего модуля, спецификация родителя не может содержать «обычный» спецификатор контекста (with Child) для дочернего модуля. Мы обсудим это позже.

Согласно другому правилу, из видимой части приватного модуля видна приватная часть его родителя (в точности, как это происходит в теле пакета-родителя). Эта «дополнительная» видимость не нарушает родительскую инкапсуляцию, поскольку использовать приватные модули могут только те модули, которые и так видят приватную часть родителя. С другой стороны, в общедоступном модуле приватную часть родителя видно только из его приватной части и тела. Это обеспечивает инкапсуляцию данных родителя.

Особая форма спецификации контекста private with, которая была добавлена в Аде 2005, обеспечивает видимость перечисленных модулей в приватной части пакета. Это полезно, когда приватная часть общедоступного дочернего пакета нуждается в информации, предоставляемой приватным дочерним модулем. Допустим, у нас есть прикладной пакет App и два дочерних App.User_View и App.Secret_Details:

```
private package App.Secret_Details is
   type Inner is ...
   ...  -- различные операции для типа Inner
end App.Secret_Details;

private with App.Secret_Details;
package App.User_View is
   type Outer is private;
   ...     -- различные операции для типа Outer
      -- тип Inner не видим здесь
private
   type Outer is record
      X : App.Secret_Details.Inner;
```

(continues on next page)

(continued from previous page)

```
      ...
   end record;
   ...
end App.User_View;
```

Обычный спецификатор контекста (with App.Secret_Details;) не допустим в User_View, поскольку это бы позволило клиенту увидеть информацию из пакета Secret_Details через видимую часть пакета User_View. Все попытки обойти правила видимости в Аде тщательно заблокированы.

30.5 Модульное тестирование

Одна из проблем, встречающаяся при тестировании кода, заключается в том, чтобы предотвратить влияние тестов на поведение тестируемого кода. Это напоминает известный феномен квантовой механики, когда сама попытка наблюдения за такими частицами, как электрон, влияет на состояние этой частицы.

Тщательно разрабатывая архитектуру программного обеспечения, мы стараемся скрыть детали реализации, сохранив стройную абстракцию, например, используя приватные типы. Но при тестировании системы мы хотим иметь возможность тщательно проанализировать поведение скрытых деталей реализаций.

В качестве простейшего примера, мы хотим знать значение Top одного из стеков, объявленных с помощью пакета Stacks (в котором находится приватный тип Stack). У нас нет готовых средств сделать это. Мы могли бы добавить функцию Size в пакет Stacks, но это потребует модификации пакета и перекомпиляции пакета и всего клиентского кода. Возникает опасность внести ошибку в пакет при добавлении этой функции, либо позже, при удалении тестирующего кода (и это будет гораздо хуже).

Дочерние модули позволяют решить эту задачу легко и безопасно. Мы можем написать следующее

```
package Stacks.Monitor is
   function Size (S : Size) return Integer;
end Stacks.Monitor;

package body Stacks.Monitor is
   function Size (S : Size) return Integer is
   begin
      return S.Top;
   end Size;
end Stacks.Monitor;
```

Это возможно, так как тело дочернего модуля видит приватную часть родительского пакета. Теперь в тестах мы можем вызвать функцию Size как только нам понадобится. Когда мы убедимся, что наша программа корректна, мы удалим дочерний пакет целиком. Родительский пакет при этом не будет затронут.

30.6 Взаимозависимые типы

Эквивалент приватных типов есть во многих языках программирования, особенно в области ООП. По сути, операции, принадлежащие типу, это те, что объявлены рядом с типом в одном пакете. Для типа Stack такими операциями являются Clear, Push и Pop. Аналогичная конструкция в C++ выглядит так:

```
class Stack {
...   /* детали реализации стека */
public:
   void Clear();
   void Push(float);
   float Pop();
};
```

Подход C++ удобен тем, что использует один уровень при именовании — Stack, в то время, как в Аде мы используем два - имя пакета и имя типа, т.е. Stacks.Stack. Однако, этого, при желании, легко избежать, используя спецификатор использования — use Stacks. И, более того, появляется возможность выбора предпочитаемого стиля. Можно, например, назвать тип нейтрально — Object или Data, а затем ссылаться на него написав Stacks.Object или Stacks.Data.

С другой стороны, если в двух типах мы желаем для реализации использовать общую информацию, на языке Ада это может быть легко достигнуто:

```
package Twins is
   type Dum is private;
   type Dee is private;
   ...
private
   ...   --  Общая информация для реализации
end Twins;
```

где в приватной части мы определим Dum и Dee так, что они будут иметь свободный взаимный доступ к данным друг друга.

В других языках это может быть не так просто. Например, в C++ нужно использовать довольно спорный механизм friend. Подход, используемый в Аде, предотвращает некорректное использование данных и раскрытие данных реализации и является при этом симметричным.

Следующий пример демонстрирует взаимную рекурсию. Допустим, мы исследуем схемы из линий и точек, при этом каждая точка лежит на пересечении трех линий, а каждая линия проходит через три точки. (Это пример на самом деле не случаен, две из фундаментальных теорем проективной геометрии оперируют такими структурами). Это легко реализовать в одном пакете, используя ссылочные типы:

```
package Points_And_Lines is
   type Point is private;
   type Line is private;
   ...
private
   type Point is record
      L, M, N : access Line;
   end record;
   type Line is record
      P, Q, R : access Point;
   end record;
end Points_And_Lines;
```

Если мы решим расположить каждый тип в своем пакете, это тоже возможно, но необходимо использовать специальную форму спецификатора контекста limited with, введенную в

стандарте Ada 2005. (Два пакета не могут ссылаться друг на друга, используя обычный with, потому, что это вызовет циклическую зависимость между ними при инициализации). Мы напишем:

```
limited with Lines;
package Points is
   type Point is private;
   ...
private
   type Point is record
      L, M, N : access Lines.Line;
   end record;
end Points;
```

и аналогичный пакет для Lines. Данная форма спецификатора контекста обеспечивает видимость неполным описаниям типов из данного пакета. Грубо говоря, такая видимость пригодна лишь для использования в ссылочных типах.

30.7 Контрактное программирование

В примере со стеком, рассмотренном нами ранее, есть некоторые недочеты. Хотя он прекрасно иллюстрирует использование приватных типов для сокрытия деталей реализации, ничто в спецификации пакета не отражает того, что мы реализуем стек. Несмотря на то, что мы тщательно выбрали имена для операций со стеком, такие как Push, Pop и Clear, некто (по ошибке или злонамеренно) может использовать Push для извлечения элементов, а Pop — для их добавления. В целях повышения надежности и безопасности было бы полезно иметь механизм, который учитывает намерения автора в контексте семантики типа и подпрограмм, объявленных в пакете.

Такие средства были добавлены в стандарт Ада 2012. Они аналогичны предлагаемым в Eiffel средствам конрактного программирования. Воспользовавшись ими, программист указывает пред- и пост-условия для подпрограмм и инварианты приватных типов. Эти конструкции имеют форму логических условий. Специальные директивы компилятору включают проверку этих условий в момент исполнения. Чтобы привязать эти конструкции к соответствующему имени (пред/пост-условия к подпрограммам и инварианты к типам), используется новый синтаксис спецификации аспекта. Вкратце:

- Пред-условие проверяется в точке вызова подпрограммы и отражает обязательства вызывающего;
- Пост-условие проверяется при возврате из подпрограммы и отражает обязательства вызываемой подпрограммы;
- Инвариант типа эквивалентен пост-условию для каждой подпрограммы типа, видимой вне пакета, поэтому проверяется при выходе из каждой такой подпрограммы. Инвариант отражает «глобальное» состояние программы после выхода из любой такой подпрограммы.

Представленная ниже версия пакета Stacks иллюстрирует все три концепции. Чтобы получить нетривиальный инвариант типа, мы ввели новое условие — стек не должен иметь повторяющиеся элементы.

```
package Stacks is
   type Stack is private
      with Type_Invariant => Is_Unduplicated (Stack);

   function Is_Empty (S: Stack) return Boolean;
   function Is_Full (S: Stack) return Boolean;
   function Is_Unduplicated (S: Stack) return Boolean;
```

(continues on next page)

(continued from previous page)

```
   function Contains (S: Stack; X: Float) return Boolean;
   --  Внимание! Из Contains(S,X) следует Is_Empty(S)=False

   procedure Push (S: in out Stack; X: in Float)
      with
         Pre => not Contains(S,X) and not Is_Full(S),
         Post => Contains(S,X);

   procedure Pop (S: in out Stack; X: out Float)
      with
         Pre => not Is_Empty(S),
         Post => not Contains(S,X) and not Is_Full (S);

   procedure Clear (S: in out Stack)
      with
         Post => Is_Empty(S);

private
   ...
end Stacks;
```

Синтаксис спецификаций аспектов выглядит очевидным. Отсутствие пред-условия (как в Clear) эквивалентно пред-условию True.

Контракты (т. е. пред/пост-условия и инварианты) можно использовать по разному. В простейшем случае они уточняют намерения автора и выступают в роли формальной документации. Они также могут служить для генерации проверок соответствующих условий в момент исполнения. Директива Assertion_Policy управляет этим поведением. Она имеет форму

```
pragma Assertion_Policy(policy_identifier);
```

Когда policy_identifier равен Check, проверки генерируются, а если Ignore, то игнорируются. (Поведение по умолчанию, в отсутствии этой директивы, зависит от реализации компилятора.)

Третий способ использования контрактов способствует использованию формальных методов доказательства свойств программы, например, доказательство того, что код подпрограммы согласован с пред- и пост-условиями. Этот подход на примере языка SPARK мы обсудим в другой главе.

Стандарт Ada 2012 включает разнообразные конструкции, связанные с контрактным программированием, которые не попали в наш пример. Кванторные выражения (for all и for some) напоминают инструкции циклов. Функции-выражения позволяют поместить простейшую реализацию функций прямо в спецификацию пакета. Это выглядит логичным при задании пред/пост-условий, поскольку они являются частью интерфейса пакета. В Ada 2012 добавлены новые атрибуты: 'Old — позволяет в пост-условии сослаться на исходное значение формального параметра подпрограммы, а 'Result в пост-условии ссылается на результат, возвращаемый функцией. За более детальной информацией можно обратиться к стандарту языка или к разъяснениям стандарта (Rationale Ada 2012).

Следует отметить, что степень детализации условий контракта может значительно варьироваться. В нашем примере единственное требование к подпрограмме Push состоит в том, что элемент должен быть добавлен в стек. В то же время, семантика стека «последний вошел, первый вышел» более конкретна: элемент должен быть добавлен так, чтобы следующий вызов Pop его удалил. Аналогично, от Pop требуется, чтобы он вернул какой-то элемент. Если мы хотим быть более точными, необходимо указать, что этот элемент был добавлен последним. Подобные условия также можно написать, используя средства языка Ada 2012. Мы оставляем это читателю в качестве упражнения.

CHAPTER

THIRTYONE

БЕЗОПАСНОЕ ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Появившись в первый раз в языке Simula в 1960-х, объектно-ориентированное программирование (ООП) распространилось затем в исследовательских и академических кругах с такими языками как Smalltalk в конце 1980-х и заняло существующее прочное положение с появлением C++ и Java в начале 1990-х. Выдающимся свойством ООП считается его гибкость. Но гибкость в чем-то похожа на свободу, которую мы обсуждали во введении — злоупотребляя ею, можно допустить появление опасных ошибок.

Ключевая идея ООП в том, что доминирующая роль достается объектам, в то время как процедуры (методы) управления объектом являются его свойствами. Старый же метод, называемый структурным программированием, заключается в декомпозиции подпрограмм, когда организация программы определяется структурой подпрограмм, при этом объекты остаются пассивными сущностями.

Оба подхода имеют право на жизнь и своих сторонников. Существуют случаи, когда строгий ООП подход использовать неуместно.

Язык Ада достигает поразительного баланса. Мы с уверенностью можем назвать его методологически нейтральным в сравнении с языком Java, например, который является «чистым» ООП. В самом деле, идеи ООП проникли в язык Ада еще при его создании, в 1980-х, проявившись в концепции пакетов и приватных типов, как механизма скрытия деталей реализации и задач, как механизма абстракции активных сущностей. Стандарт Ада 95 включил главные свойства ООП, такие как наследование, полиморфизм, динамическое связывание и понятие «класса» как набора типов, связанных отношением наследования.

31.1 ООП вместо структурного программирования

Мы рассмотрим два примера для иллюстрации различных моментов. Мы выбрали их, исходя из легкости их понимания, что избавит нас от необходимости объяснять прикладную область. Примеры касаются геометрических объектов (коих существует великое множество) и людей (которых только две категории, мужчины и женщины).

Сначала рассмотрим геометрические объекты. Для простоты остановимся на объектах на плоскости. У каждого объекта есть позиция. Мы можем определить базовый объект со свойствами, общими для всех видов объектов:

```
type Object is tagged record
   X_Coord : Float;
   Y_Coord : Float;
end record;
```

Здесь ключевое слово tagged (тегированный, то есть каждому типу соответствует уникальный тег) отличает этот тип от обычных записей (таких как Date из главы 3) и означает, что тип можно в будущем расширить. Более того, объекты такого типа хранят в себе специальное поле-тег, и этот тег определяет тип объекта во время исполнения.

Объявив типы для специфических геометрических объектов, таких как окружность, треугольник, квадрат и прочие, мы получим различные значения тегов для каждого из типов. Компоненты X_Coord, Y_Coord задают центр объекта.

Мы можем объявить различные свойства объекта, такие как площадь и момент инерции. Каждый объект имеет эти свойства, но они зависят от формы объекта. Такие свойства можно определить при помощи функций и объявить их в том же пакете, что и тип. Таким образом, мы начнем с пакета:

```
package Geometry is
   type Object is abstract tagged record
      X_Coord, Y_Coord : Float;
   end record;

   function Area(Obj: Object) return Float is abstract;
   function Moment(Obj: Object) return Float is abstract;
end Geometry;
```

Здесь мы объявили тип и его операции, как абстрактные. На самом деле нам не нужны объекты типа Object. Объявив тип абстрактным, мы предотвратим создания таких объектов по ошибке. Нам нужны реальные объекты, такие как окружности, у которых есть свойства, такие как площадь. Если нам потребуется объект точка, мы объявим отдельный тип Point для этого. Объявив функции Area и Moment абстрактными, мы гарантируем, что каждый конкретный тип, такой как окружность, предоставит свой код для вычисления этих свойств.

Теперь мы готовы определить тип окружность. Лучше всего сделать это в дочернем пакете:

```
package Geometry.Circles is
   type Circle is new Object with record
      Radius : Float;
   end record;

   function Area(C: Circle) return Float;
   function Moment(C: Circle) return Float;
end Geometry.Circles;

with Ada.Numerics; use Ada.Numerics;  -- для доступа к π

package body Geometry.Circles is

   function Area(C: Circle) return Float is
   begin
      return π * C.Radius ** 2;
   end Area;

   function Moment(C: Circle) return Float is
   begin
      return 0.5 * C.Area * C.Radius ** 2;
   end Moment;
end Geometry.Circles;
```

В этом примере мы порождаем тип Circle от Object. Написав new Object, мы неявно наследуем все видимые операции типа Object пакета Geometry, если мы не переопределим их. Поскольку наши операции абстрактны, а сам тип Circle — нет, мы обязаны переопределить их явно. Определение типа расширяет тип Object новым компонентом Radius, добавляя его к определенным в Object компонентам (X_Coord и Y_Coord).

Заметим, что код вычисления площади и момента располагается в теле пакета. Как было объяснено в главе «Безопасная архитектура», это значит, что код можно изменить и перекомпилировать без необходимости перекомпиляции описания самого типа и всех модулей, использующих его.

Затем мы могли бы объявить тип для квадрата Square (с дополнительным компонентом

длинны стороны квадрата), треугольника Triangle (три компоненты соответствующие его сторонам) и так далее. При этом код для Object и Circle не нарушается.

Множество всевозможных типов из иерархии наследования от Object обозначаются как Object'Class и называются в терминологии Ады классом. Язык тщательно различает отдельные типы, такие как Circle от классов типа, таких как Object'Class. Это различие помогает избежать путаницы, которая может возникнуть в других языках. Если мы, в свою очередь, унаследуем тип от Circle, то будем иметь возможность говорить о классе Circle'Class.

В теле функции Moment продемонстрировано использование точечной нотации. Мы можем воспользоваться одной из следующих форм записи:

- C.Area – точечная нотация
- Area (C) – функциональная нотация

Точечная нотация появилась в стандарте Ада 2005 и смещает акцент в сторону ООП, указывая на то, что объект C доминирует над функцией Area. Теперь объявим несколько объектов, например:

```
A_Circle: Circle := (1.0, 2.0, Radius => 4.5);
My_Square: Square := (0.0, 0.0, Side => 3.7);
The_Triangle: Triangle := (1.0, 0.5, A=>3.0, B=>4.0, C=>5.0);
```

Процедура для печати свойств объекта может выглядеть так:

```
procedure Print(Obj: Object'Class) is  -- Obj is polymorfic
begin
   Put("Area is "); Put(Obj.Area);  -- dispatcing call of Area
   ...
end Print;

Print (A_Circle);
Print (My_Square);
```

Формальный параметр Obj — полиморфный, т. е. он может ссылаться на объекты различного типа (но только из иерархии растущей из Object) в различные моменты времени.

Процедура Print может принимать любой объект из класса Object'Class. Внутри процедуры вызов Area динамически диспетчеризируется, чтобы вызвать функцию Area, соответствующую конкретному типу параметра Obj. Это всегда безопасно, поскольку правила языка таковы, что любой объект в классе Object'Class будет иметь функцию Area. В тоже время сам тип Object объявлен абстрактным, следовательно нет способа создать объекты этого типа, поэтому не важно, что для этого типа нет функции Area, ведь ее вызов невозможен.

Аналогично мы могли бы объявить типы для людей. Например:

```
package People is

   type Person is abstract record
      Birthday: Date;
      Height: Inches;
      Weight: Pounds;
   end record;

   type Man is new Person with record
      Bearded: Boolean;  -- Имеет бороду
   end record;

   type Woman is new Person with record
      Births: Integer;  -- Кол-во рожденных детей
   end record;
```

(continues on next page)

(continued from previous page)

```
   ...  -- Различные операции
end People;
```

Поскольку все альтернативы заранее известны и новых типов не будет добавляться, мы могли бы использовать здесь запись с вариантами. Это будет в духе структурного программирования.

```
type Gender is (Male, Female);

type Person (Sex: Gender) is record
   Birthday: Date;
   Height: Inches;
   Weight: Pounds;
   case Sex is
      when Male =>
         Bearded: Boolean;
      when Female =>
         Births: Integer;
   end case;
end record;
```

Затем мы объявим различные необходимые операции для типа Person. Каждая операция может иметь в теле инструкцию case чтобы принять во внимание пол человека.

Это может выглядеть довольно старомодно и не так элегантно, тем не менее этот подход имеет свои значительные преимущества.

Если нам необходимо добавить еще одну операцию в ООП подходе, весь набор типов нужно будет исправить, т. к. в каждом типе нужно добавить реализацию новой операции. Если нужно добавить новый тип, то это не затронет уже существующие типы.

В случае со структурным программированием все выглядит с точностью до наоборот.

Если нам нужно добавить еще один тип в структурном подходе, весь код нужно будет исправить, т. к. в каждой операции нужно реализовать случай для нового типа. Если нужно добавить новую операцию, то это не затронет уже существующие операции.

ООП подход считается более надежным ввиду отсутствия case инструкций, которые тяжело сопровождать. Хотя это действительно так, но иногда сопровождение ООП еще тяжелее, если приходится добавлять новые операции, что требует доработки каждого типа в иерархии.

Ада поддерживает оба подхода и они оба безопасны в Аде.

31.2 Индикатор overriding

Одна из уязвимостей ООП проявляется в момент переопределения унаследованных операций. При добавлении нового типа, нам необходимо добавить новые версии соответствующих операций. Если мы не добавляем свою операцию, то будет использоваться унаследованная от родителя.

Опасность в том, что, добавляя новую операцию, мы можем ошибиться в написании:

```
function Aera(C: Circle) return Float;
```

либо в типе аргумента или результата:

```
function Area(C: Circle) return Integer;
```

В любом случае, старая операция остается не переопределенной, а вместо этого создается совершенно новая операция. Когда надклассовая операция вызовет Area, будет выполнена унаследованная версия, а вновь добавленный код будет полностью проигнорирован. Подобные ошибки трудно диагностируются — компиляция проходит без ошибок, программа запускается и работает, но результат зачастую совершенно неожиданный.

(Хотя Ада предлагает механизм абстрактных операций, как например Area у Object, это лишь дополнительная мера предосторожности. И этого недостаточно, если мы порождаем тип от не абстрактного типа, либо мы не слишком осторожны, чтобы объявить функцию Area абстрактной.)

Чтобы предотвратить подобные ошибки, мы можем воспользоваться синтаксической конструкцией, появившейся в стандарте Ада 2005:

```
overriding function Area(C: Circle) return Float;
```

В этом случае, если будет допущена ошибка, компилятор обнаружит ее. С другой стороны, если мы действительно добавляем новую операцию, мы укажем это так:

```
not overriding function Aera(C: Circle) return Float;
```

То, что данный индикатор не является обязательным, главным образом обусловлено необходимостью сохранения совместимости с предыдущими версиями языка.

Возможно, синтаксис not overriding индикатора неоправдано тяжеловесен, учитывая необходимость частого его использования. Идеальным было бы требовать использования overriding для всех переопределенных операций и только для них. Другими словами, не использовать not overriding вообще. Это позволило бы находить оба типа ошибок:

- Ошибки в написании и другие ошибки, приводящие к отсутствию переопределения операций, фиксируются благодаря наличию overriding;
- Случайное переопределение операции (например, в случае появления новой операции в родительском типе) обнаруживается ввиду отсутствия overriding.

Требование совместимости делает такой подход невозможным в общем случае. Однако такого эффекта можно достичь, используя опции компилятора GNAT фирмы AdaCore:

- -gnatyO включает появление предупреждений о переопределении операций без использования overriding;
- -gnatwe заставляет компилятор трактовать все предупреждения как ошибки.

Другие языки, типа C++ и Java, предоставляют меньше содействия в этом аспекте, оставляя подобные ошибки не обнаруженными.

31.3 Запрет диспетчеризации вызова подпрограмм

В условиях, когда надежность критически важна, динамическое связывание часто запрещено. Надежность повышается, если мы можем доказать, что поток управления соответствует заданному шаблону, например, отсутствует недостижимые участки кода. Традиционно это означает, что нам приходится следовать методикам структурного программирования, вводя в явном виде такие инструкции, как if и case.

Хотя диспетчеризация вызовов подпрограмм является ключевым свойством ООП, другие возможности (например повторное использование кода благодаря наследованию) также являются привлекательными. Таким образом, возможность расширять типы, но без использования диспетчеризации, все еще имеет значительную ценность. Использование точечной нотации вызова также имеет свои преимущества.

Существует механизм, который позволяет отключать некоторые возможности языка Ада в текущей программе. Речь идет о директиве компилятору Restrictions. В данном случае мы напишем:

```
pragma Restrictions(No_Dispatch);
```

Это гарантирует (в момент компиляции), что в программе отсутствует конструкции типа X'Class, а значит и диспетчеризация вызова невозможна.

Заметим, что это ограничение соответствует требованиям языка SPARK, о котором мы упоминали во введении, часто используемого в критических областях. Язык SPARK позволяет использовать расширения типов, но запрещает надклассовые операции и типы.

Когда используется ограничение No_Dispatch, реализация языка получает возможность избежать накладных расходов, связанных с ООП. Нет необходимости создания в каждом типе «виртуальных таблиц» для диспетчеризации вызовов. (Такие таблицы содержат адреса всех операций данного типа). Также нет необходимости в специальном поле тега в каждом объекте.

Здесь существуют также менее очевидные преимущества. При полном ООП подходе некоторые предопределенные операции (например операция сравнения) также имеет возможность диспетчеризации, что приводит к дополнительным расходам. Итоговый результат применения ограничения минимизирует документирование неактивного кода (это код, который присутствует в программе и соответствует требованиям к ПО, но никогда не исполняется) в целях сертификации по стандартам DO-178B и DO-178C.

31.4 Интерфейсы и множественное наследование

Иногда множественное наследование расценивается как Святой Грааль в индустрии ПО и как критерий оценки языков программирования. Здесь мы не будем отвлекаться на историю развития этого вопроса. Вместо этого мы остановимся на основных проблемах, связанных с этим механизмом.

Допустим, у нас есть возможность унаследовать тип от двух произвольных родительских типов. Вспомним знаменитый роман «Флатландия» Эдвина Эбботта (вышедший в 1884г.). Это сатира на социальную иерархию, в которой люди - это плоские геометрические фигуры. Рабочий класс - это треугольники, средний класс - другие полигоны, аристократы - окружности. Удивительно, но женщины там - двуугольники, т. е. просто отрезки прямой линии.

Используя объявленные ранее типы Object и Person, мы могли бы попытаться определить обитателей Флатландии как тип, унаследованный от обоих типов:

```
type Flatlander is
  new Geometry.Object and People.Person; -- illegal
```

Вопрос который теперь возникает: какие свойства унаследует новый тип? Мы ожидаем, что Flatlander унаследует компоненты X_Coord и Y_Coord от Object и Birthday от Person, хотя Heigh и Weight выглядит сомнительно для плоских персонажей. Конечно Area должен быть унаследован, потому, что Flatlander имеет площать, как и момент инерции.

Теперь должно быть ясно, какие проблемы могут возникнуть. Допустим, оба родительских типа имеют операцию с одним именем. Это весьма вероятно для широко распространенных имен типа Print, Make, Copy и т. д. Какую из них нужно наследовать? Что делать, если оба родителя имеют компоненты с одинаковыми именами? Подобные вопросы объязательно возникнут, если оба родителя имеют общий родительский тип.

Некоторые языки реализуют множественное наследование в такой форме, вводя сложные правила, чтобы урегулировать подобные вопросы. Примером могут служить C++ и Eiffel. Возможные решения включают переименование, явное указание имя родителя в

неоднозначных случаях, либо выбор родительского типа согласно позиции в списке. Некоторые из предлагаемых решений имеют субъективный оттенок/характер, для кого-то они очевидны, хотя других приводят в замешательство. Правила C++ дают широкую свободу программисту наделать ошибок.

Трудности возникают в основном двух видов: наследование компонент и наследование реализаций операций от более чем одного родителя. Но фактически проблем с наследованием спецификации (другими словами интерфейса) операции не бывает. Этим воспользовались в языке Java и этот путь оказался довольно успешным. Он также реализован и в языке Ада.

Таким образом, в Аде, начиная со стандарта Ada 2005, мы можем наследовать от более чем одного типа:

```
type T is new A and B and C with record
   ... -- дополнительные компоненты
end record;
```

но только первый тип в списке (A) может иметь компоненты и реальные операции. Остальные типы должны быть так называемыми интерфейсами (термин позаимствован у Java умышленно), т. е. абстрактными типами без компонент, у которых все операции либо абстрактные, либо null-процедуры. (Первый тип может также быть интерфейсом.)

Мы можем описать Object, как интерфейс:

```
package Geometry is

   type Object is interface;

   procedure Move(   Obj: in out Object;
             New_X, New_Y: Float) is abstract;
   function X_Coord(Obj: Object) return Float is abstract;
   function Y_Coord(Obj: Object) return Float is abstract;
   function Area(Obj: Object) return Float is abstract;
   function Moment(Obj: Object) return Float is abstract;
end Geometry;
```

Обратите внимание, что компоненты были удалены и заменены дополнительными операциями. Процедура Move позволяет передвигать объект, т. е. устанавливать новые координаты x и y. Функции X_Coord, Y_Coord возвращают текущую позицию.

Также следует заметить, что точечная нотация позволяет по прежнему обращаться к координатам, написав A_Circle.X_Coord и The_Triangle.Y_Coord, как если бы это были компоненты.

Теперь при определении конкретного типа мы должны предоставить реализацию всем этим операциям. Предположим:

```
package Geometry.Circles is

   type Circle is new Object with private; -- partial view
   procedure Move(C: in out Circle; New_X, New_Y: Float);
   function X_Coord(C: Circle) return Float;
   function Y_Coord(C: Circle) return Float;
   function Area(C: Circle) return Float;
   function Moment(C: Circle) return Float;

   function Radius(C: Circle) return Float;
   function Make_Circle(X, Y, R: Float) return Circle;
private

   type Circle is new Object with record
      X_Coord, Y_Coord: Float;
```

(continues on next page)

(continued from previous page)

```
      Radius : Float;
   end record;
end Geometry.Circles;

package body Geometry.Circles is

   procedure Move(C: in out Circle; New_X, New_Y: Float) is
   begin
      C.X_Coord := New_X;
      C.Y_Coord := New_Y;
   end Move;

   function X_Coord(C: Circle) return Float is
   begin
      return C.X_Coord;
   end X_Coord;

   -- Аналогично Y_Coord, а Area и Moment – как ранее
end Geometry.Circles;
```

Мы определили тип Circle как приватный, чтобы спрятать все его компоненты. Тем не менее, поскольку приватный тип унаследован от Object, он наследует все свойства Object. Обратите внимание, что мы добавили функции для создания окружности и получения его радиуса.

В основе программирования с использованием интерфейсов лежит идея, что мы обязаны предоставить реализацию операций, требуемых интерфейсом. Здесь множественное наследование не касается наследования существующих свойств, скорее это наследование контрактов, которым необходимо следовать. Таким образом, Ада позволяет множественное наследование интерфейсов, но только единичное наследование для реализаций.

Возвращаясь к Флатландии, мы можем определить:

```
package Flatland is

   type Flatlander is abstract new Person and Object
      with private;

   procedure Move(F: in out Flatlander; New_X,New_Y: Float);
   function X_Coord(F: Flatlander) return Float;
   function Y_Coord(F: Flatlander) return Float;
private

   type Flatlander is abstract new Person and Object
      with record
          X_Coord,  Y_Coord : Float := 0.0;
         ... -- остальные необходимые компоненты
      end record;
end Flatland;
```

Теперь тип Flatlander будет наследовать компоненту Birthday и прочие от типа Person и все реализации операций типа Person (мы не показываем их тут) и абстрактные операции типа Object. Удобно определить координаты, как компоненты типа Flatlander, чтобы легко реализовать такие операции, как Move, X_Coord, Y_Coord. Обратите внимание, что мы задали начальное значение этих компонент, как ноль, чтобы задать положение Flatlander по умолчанию.

Тело пакета будет следующим:

```
package body Flatland is

   procedure Move(F: in out Flatlander; New_X,New_Y: Float)
```

(continues on next page)

(continued from previous page)

```
   is
   begin
      F.X_Coord := New_X;
      F.Y_Coord := New_Y;
   end Move;

   function X_Coord(F: Flatlander) return Float is
   begin
      return F.X_Coord;
   end X_Coord;
   -- аналогично Y_Coord
end Flatland;
```

Сделав тип Flatlander абстрактным, мы избегаем необходимости немедленно предоставить реализацию всем операциям, например Area. Теперь мы можем объявить тип Square пригодным для Флатландии (когда роман вышел в печать, автор подписался псевдонимом A Square):

```
package Flatland.Squares is

   type Square is new Flatlander with record
      Size: Float;
   end record;

   function Area (S: Square) return Float;
   function Moment (S: Square) return Float;
end Flatland.Square;

package body Flatland.Squares is

   function Area (S: Square) return Float is
   begin
      return S.Size ** 2;
   end Area;

   function Moment (S: Square) return Float is
   begin
      return S.Area * S.Side ** 2 / 6.0;
   end Moment;

end Flatland.Square;
```

Таким образом, все операции в итоге получили реализацию. В демонстрационных целях мы сделали дополнительный компонент Size видимым, хотя тут можно использовать и приватный тип. Теперь мы можем определить Др. Эбботта как:

```
A_Square : Square := (Flatland with Side => 3.0);
```

и он будет иметь все свойства квадрата и человека. Обратите внимание на агрегат, который получает значения по умолчанию для приватных компонент, а дополнительные компоненты инициализирует явно.

Существуют другие важные свойства интерфейсов, которых мы коснемся лишь вскользь. Интерфейс может иметь в качестве операции null-процедуру. Такая процедура ничего не делает при вызове. Если два родителя имеют одну и туже операцию, то null-процедура переопределяет абстрактную. Если два родителя имеют одну и ту же операцию (с совпадающими параметрами и результатом), они сливаются в одну операцию, для которой требуется реализация. Если параметры и/или результат отличаются, то необходимо реализовать обе операции, т. к. они перегружены. Таким образом, правила сформулированы так, чтобы минимизировать сюрпризы и максимизировать выигрыш от множественного наследования.

31.5 Взаимозаменяемость

Этот раздел касается специализированной темы, которая может быть интересна при углубленном изучении.

Наследование можно рассматривать с двух перспектив. С точки зрения возможностей языка, это способность породить тип от родительского и унаследовать состояние (компоненты) и операции, сохранив при этом возможность добавлять новые компоненты и операции и переопределять унаследованные операции. С точки зрения моделирования либо теории типов, наследование это отношение («это есть») между подклассом и супер-классом: если класс S - это подкласс T, то любой объект класса S также является объектом класса T. Это свойство — основа полиморфизма. В терминах языка Ада, для любого тегового типа T, переменная типа T'Class может ссылаться на объект типа T или любого типа, унаследованного (прямо или косвенно) от T. Это значит, что любая операция, возможная для T, будет работать (как унаследованная либо переопределенная) и для объекта любого подкласса T.

Более формальная формулировка этого требования известна, как принцип подстановки Барбары Лисков, который выражен в терминах теории типов:

Пусть q(x) свойство объектов x типа T истинно. Тогда q(y) истинно для объектов типа S, где S является подтипом T. (Здесь «подтип» означает «подкласс».)

Хорошей практикой является проектирование иерархии классов так, чтобы выполнялся данный принцип. Если он нарушается, то становится возможным вызвать неподходящую операцию, используя динамическое связывание, что приведет к ошибке времени исполнения. Это возможно в том случае, когда наследование используется там, где два класса должны быть связаны менее строгим отношением.

Хотя сам принцип Лисков может показаться очевидным, его связь с контрактным программированием таковой не является. Напомним, что в методе вы можете дополнить спецификацию подпрограмм пред- и/или пост-условиями. Встает вопрос, если вы переопределяете операцию, налагает ли принцип Лисков дополнительные ограничения на пред- и пост-условия для новой версии подпрограммы? Ответ - «да»: пред-условия не могут быть усилены (например, вы не можете сформулировать пред-условие, добавив условие к родительскому через and оператор). Аналогично, пост-условие не может быть ослаблено.

На первый взгляд это противоречит тому, чего вы можете ожидать. Подкласс обычно ограничивает множество значений своего супер-класса. Поэтому накладывать более сильные пред-условия операции подкласса может показаться имеющим смысл. Но при ближайшем рассмотрении оказывается, что это нарушает принцип Лисков. С точки зрения вызывающего, для выполнения операции X.Op(...) полиморфной переменной X, имеющей тип T, необходимо обеспечить выполнение предусловия операции Op типа T. У автора этого кода нет возможности знать все возможные подклассы T. Если случится так, что X ссылается на объект типа T1, у которого предусловие Op сильнее, чем у T, то вызов не сможет состояться, поскольку проверка пред-условия не пройдет. Аналогичные рассуждения докажут, что пост-условие в подклассе не может быть слабее. Вызывающий ожидает выполнение пост-условия после вызова операции, если подкласс этого не гарантирует, это приведет к ошибке.

Дополнение в области ООП и связанных технологий к стандарту DO-178C (DO-332) останавливается на этом вопросе. Оно не требует подчинения принципу Лисков, вместо этого предлагает проверять «локальную согласованность типов». «Согласованность типов» означает выполнение принципа Лисков: операции подкласса не могут иметь более сильные пред-условия и более слабые пост-условия. «Локальная» означает, что необходимо анализировать только реально встречающийся в программе контекст. Например, если существует операция, у которой пред-условие более сильное, но она никогда не вызывается при диспетчеризации вызова, то это не вредит.

Стандарт DO-332 предлагает три подхода доказательства локальной согласованности типов, один на основе формальных методов и два на основе тестирования:

- формально проверить взаимозаменяемость;
- удостовериться, что каждый тип проходит все тесты всех своих родителей, которых он может замещать;
- для каждой точки динамического связывания протестировать, что каждый метод может быть вызван (пессимистическое тестирование).

Первый подход предполагает непосредственную проверку принципа Лисков. Ада 2012 поддерживает явным образом пред- и пост-условия, что помогает провести автоматический формальный анализ, используя специальный инструментарий. Это рекомендуемый подход, если возможно использовать формальные методы, например при помощи инструмента SPARK Pro, поскольку он обеспечивает наивысший уровень доверия.

Второй подход применим при использовании модульного тестирования. В этом контексте, каждая операция класса обладает набором тестов для проверки требований. Переопределенная операция обычно имеет расширенные требования по сравнению с изначальной, поэтому будет иметь больше тестов. Каждый класс тестируется отдельно при помощи всех тестов, принадлежащих ему методов. Идея в том, чтобы проверить принцип взаимозаменяемости для некоторого тегового типа, выполнив все тесты всех его родительских типов, используя объект данного тегового типа. Gnattest, инструмент для модульного тестирования из состава GNAT Pro, предоставляет необходимую поддержку для автоматизации процесса тестирования, в том числе и принципа Лисков.

Третий подход может быть наименее сложным в случае, когда диспетчеризирумые вызовы редки и иерархия типов неглубокая. В составе GNAT Pro, есть инструмент GNAT-stack, способный найти все диспетчеризируемые вызовы с указанием всех возможных вызываемых подпрограмм.

Более подробную информацию по этой теме можно найти в документации «ООП для надежных систем на языке Ада» от AdaCore.

CHAPTER

THIRTYTWO

БЕЗОПАСНОЕ СОЗДАНИЕ ОБЪЕКТОВ

Эта глава раскрывает некоторые аспекты управления объектами. Под объектами здесь мы понимаем как мелкие объекты в виде простых констант и переменных элементарного типа, например Integer, так и большие объекты из области ООП.

Язык Ада предоставляет развитые и гибкие инструменты в этой области. Эти инструменты, главным образом, не являются обязательными, но хороший программист использует их по-возможности, а хороший менеджер настаивает на их использовании по-возможности.

32.1 Переменные и константы

Как мы уже видели, мы можем определить переменные и константы, написав:

```
Top : Integer;  --  переменная

Max : constant Integer := 100;  --  константа
```

соответственно. Top - это переменная и мы можем присваивать ей новые значения, в то время как Max - это константа и ее значение не может быть изменено. Заметьте, что в определении константы задавать начальное значение необходимо. Переменным тоже можно задать начальное значение, но это не обязательно.

Преимущество констант в том, что их нельзя нечаянно изменить. Эта конструкция не только удобный «предохранитель». Она также помогает каждому, кто читает программу, сразу обозначая статус объекта. Важная деталь тут в том, что значение не обязательно должно быть статическим, т. е. известным в момент компиляции. В качестве примера приведем код вычисления процентных ставок:

```
procedure Nfv_2000 (X: Float) is
   Factor: constant Float := 1.0 + X/100.0;
begin
   return 1000.0 * Factor**2 + 500.0 * Factor - 2000.00;
end Nfv_2000;
```

Каждый вызов функции Nfv_2000 получает новое значение X и, следовательно, новое значение Factor. Но Factor не изменяется в течении одного вызова. Хотя этот пример тривиальный и легко видеть, что Factor не изменяется, необходимо выработать привычку использовать constant везде, где это возможно.

Параметры подпрограммы это еще один пример концепции переменных и констант.

Параметры бывают трех видов: in, in out и out. Если вид не указан в тексте, то, по умолчанию, используется in. В версиях языка вплоть до Ада 2005 включительно функции могли иметь параметры только вида in. Это ограничение имело методологический характер, чтобы склонить программиста не писать функции, имеющие побочные эффекты. На практике, однако, это ограничение на самом деле не работает. Создать функцию, имеющую побочные эффекты, легко, достаточно использовать ссылочный параметр или

присваивание нелокальной переменной. Более того, в случае, когда побочный эффект необходим, программист был лишен возможности обозначить это явно, использовав подходящий вид параметра. Учитывая вышесказанное, стандарт Ада 2012, наконец, разрешил функциям иметь параметры всех трех видов.

Параметр вида in это константа, получающая значение аргумента вызова подпрограммы. Таким образом, X, из примера с Nfv_2000, имеет вид in и поэтому является константой, т. е. мы не можем присвоить ей значение и у нас есть гарантия, что ее значение не меняется. Соответствующий аргумент может быть любым выражением требуемого типа.

Параметры вида in out и out являются переменными. Аргумент вызова также должен быть переменной. Различие между этими видами касается начального значения. Параметр вида in out принимает начальное значение аргумента, в то время как параметр вида out не имеет начального значения (либо оно задается начальным значением, определенным самим типом параметра, например null для ссылочных типов).

Примеры использования всех трех видов параметров мы встречали в определении процедур Push и Pop в главе «Безопасная Архитектура»:

```
procedure Push (S: in out Stack; X: in Float);

procedure Pop (S: in out Stack; X: out Float);
```

Правила, касающиеся аргументов подпрограмм, гарантируют, что свойство «константности» не нарушается. Мы не можем передать константы, такие как Factor, при вызове Pop, поскольку соответствующий параметр имеет вид out. В противном случае это дало бы возможность Pop поменять значение Factor.

Различие между константами и переменными также затрагивает ссылочные типы и объекты. Так, если мы имеем:

```
type Int_Ptr is access all Integer;

K: aliased Integer;

KP: Int_Ptr := K'Access;

CKP: constant Int_Ptr := K'Access;
```

то значение KP можно изменить, а значение CKP — нет. Хотя мы не можем заставить CKP ссылаться на другой объект, мы можем поменять значение K:

```
CKP.all := 47;  -- установить значение K равным 47
```

С другой стороны:

```
type Const_Int_Ptr is access constant Integer;

J: aliased Integer;

JP: Const_Int_Ptr := J'Access;

CJP: constant Const_Int_Ptr := J'Access;
```

где мы используем constant в описании типа. Это значит, мы не можем поменять значение объекта J, используя ссылки JP и CJP. Переменная JP может ссылаться на различные объекты, в то время, как CJP ссылается только на J.

Вид только-для-чтения и для чтения-записи

Иногда необходимо разрешить клиенту читать переменную без возможности изменять ее. Другими словами, нужно предоставить вид только-для-чтения для данной переменной. Это можно сделать при помощи отложенной константы и ссылочного типа:

```
package P is
   type Const_Int_Ptr is access constant Integer;
   The_Ptr: constant Const_Int_Ptr;  -- отложенная константа
private
   The_Variable: aliased Integer;
   The_Ptr: constant Const_Int_Ptr := The_Variable'Access;
   ...
end P;
```

Клиент может читать значение The_Variable, используя The_Ptr, написав

```
K := The_Ptr.all;  -- косвенное чтение The_Variable
```

Но, поскольку ссылочный тип описан, как access constant значение объекта не может быть изменено

```
The_Ptr.all := K;  -- ошибка, так нельзя изменить The_Variable
```

В то же время, любая подпрограмма, объявленная в пакете P, имеет непосредственный доступ к The_Variable и может изменять ее значение. Этот способ особенно полезен в случает таблиц, когда таблица вычисляется динамически, но клиент не должен иметь возможности менять ее.

Используя возможности стандарта Ада 2005, можно избежать введения ссылочного типа

```
package P is
   The_Ptr: constant access constant Integer;
private
   The_Variable: aliased Integer;
   The_Ptr: constant access constant Integer :=
          The_Variable'Access;
   ...
end P;
```

Ключевое слово constant встречается дважды в объявлении The_Ptr. Первое означает, что The_Ptr это константа. Второе — что нельзя изменить объект, на который ссылается The_Ptr.

32.2 Функция-конструктор

В таких языках, как C++, Java и C# есть специальный синтаксис для функций, создающих новые объекты. Такие функции называются конструкторами, их имена совпадают с именем типа. Введение аналогичного механизма в Аде отклонили, чтобы не усложнять язык. Конструкторы в других языках имеют дополнительную семантику (например, определяют, как вызвать конструктор родительского типа, в какой момент вызывать конструктор относительно инициализации по умолчанию и т.д.). Ада предлагает несколько конструкций, которые можно использовать вместо конструкторов, например, использование дискриминантов для параметризации инициализации, использование контролируемых типов с предоставляемой пользователем процедурой Initialize (к этому мы вернемся позже), обыкновенные функции, возвращающие значение целевого типа, поэтому необходимости в дополнительных средствах нет.

32.3 Лимитируемые типы

Типы, которые мы встречали до сих пор (Integer, Float, Date, Circle и др.), имеют различные операции. Некоторые из них предопределены, к ним относятся сравнение на равенство. Другие, такие как Area у типа Circle, определены пользователем. Операция присваивания также существует у всех перечисленных выше типов.

В некоторых случаях иметь операцию присваивания нежелательно. Этому может быть две главные причины

- тип может представлять некоторый ресурс, например права доступа, копирование которого нарушает политику безопасности
- тип может быт реализован с использованием ссылок и копирование затронет только ссылку, а не все данные.

Мы можем предотвратить присваивание, объявив тип limited. Для иллюстрации второй причины рассмотрим стек, реализованный в виде односвязного списка:

```
package Linked_Stacks is
   type Stack is limited private;
   procedure Clear(S: out Stack);
   procedure Push(S: in out Stack; X: in Float);
   procedure Pop(S: in out Stack; X: out Float);
private
   type Cell is record
      Next: access Cell;
      Value: Float;
   end record;

   type Stack is access all Cell;
end Linked_Stacks;
```

Тело пакета может быть следующим:

```
package body Linked_Stacks is

   procedure Clear(S: out Stack) is
   begin
      S := null;
   end Clear;

   procedure Push(S: in out Stack; X: in Float) is
   begin
      S := new Cell'(S, X);
   end Push;

   procedure Pop(S: in out Stack; X: out Float) is
   begin
      X := S.Value;
      S := Stack (S.Next);
   end Pop;

end Linked_Stacks;
```

Объявление стека как limited private запрещает операцию присваивания.

```
This_One, That_One: Stack;

This_One := That_One;  -- неверно, тип Stack лимитированный
```

Если бы такое присваивание сработало, это привело бы к тому, что This_One указывал бы на тот же список, что и That_One. Вызов Pop с This_One просто сдвинет его по списку

That_One вниз. Проблемы такого характера имеют общепринятое название — алиасинг, т. е. существование более чем одного способа сослаться на один и тот же объект. Зачастую это чревато плохими последствиями.

Объявление стека в данном примере работает успешно, благодаря тому, что он автоматически специализируется значением null, обозначающим пустой стек. Однако, бывают случаи, когда необходимо создать объект, инициализировав его конкретным значением (например, если мы объявляем константу). Мы не можем сделать это обычным способом:

```
type T is limited ...

...

X: constant T := Y;  --  ошибка, нельзя скопировать переменную
```

поскольку это повлечет копирование, что запрещено для лимитируемого типа.

Можно воспользоваться двумя конструкциями — агрегатами и функциями. Сначала рассмотрим агрегаты. Пусть наш тип представляет некий ключ, содержащий дату выпуска и некоторый внутренний код:

```
type Key is limited record
   Issued: Date;
   Code: Integer;
end record;
```

Поскольку тип лимитируемый, ключ нельзя скопировать (сейчас его содержимое видно, но мы это исправим позже). Но мы можем написать:

```
K: Key := (Today, 27);
```

так как, в этом случае, нет копирования целого ключа, вместо этого отдельные компоненты получают свои значения. Другими словами K строится по месту нахождения.

Более реалистично было бы объявить тип приватным. Тогда мы бы не имели возможности использовать агрегат, поскольку компоненты не были бы видимы. В этом случае мы бы использовали функцию, как конструктор:

```
package Key_Stuff is
   type Key is limited private;
   function Make_Key(...) return Key;
   ...
private
   type Key is limited record
      Issued: Date;
      Code: Integer;
   end record;
end Key_Stuff;

package body Key_Stuff is

   function Make_Key(...) return Key is
   begin
      return New_Key: Key do
         New_Key.Issued := Today;
         New_Key.Code := ...;
      end return;
   end Make_Key;
   ...
end Key_Stuff;
```

Клиент теперь может написать

```
My_Key: Key := Make_Key (...);  -- тут нет копирования
```

где параметры Make_Key используются для вычисления ключа.

Давайте обратим внимание на функцию Make_Key. Она содержит расширенную инструкцию возврата (return), которая начинается с объявления возвращаемого объекта New_Key. Когда результирующий тип функции лимитируемый, возвращаемый объект, на самом деле, строится сразу по месту своего нахождения (в нашем случае в переменной My_Key). Это выполняется аналогично тому, как заполняются компоненты при использовании агрегата. Поэтому копирования не выполняется.

В итоге в Аде мы имеем механизм инициализации клиентских объектов без использования копирования. Использование лимитируемых типов дает создателю ресурсов, таких как ключи, мощный механизм контроля их использования.

32.4 Контролируемые типы

Еще более мощным механизмом управлением объектами являются контролируемые типы. С их помощью можно написать код, исполняемый когда:

1. объект создается; 1. объект прекращает существование; 1. объект копируется, если он не лимитируемого типа.

В основе этого механизма лежат типы Controlled и Limited_Controlled, объявленные в пакете Ada.Finalization:

```
package Ada.Finalization is

   type Controlled is abstract tagged private;
   procedure Initialize(Object: in out Controlled) is null;
   procedure Adjust(Object: in out Controlled) is null;
   procedure Finalize(Object: in out Controlled) is null;

   type Limited_Controlled is abstract tagged private;
   procedure Initialize(Object: in out Limited_Controlled)       is null;
   procedure Finalize(Object: in out Limited_Controlled)
      is null;
private
   ...
end Ada.Finalization;
```

Синтаксис is null введен в Ада 2005 и упрощает определение поведения по умолчанию.

Основная идея (для нелимитируемых типов) состоит в том, что пользователь наследует свой тип от Controlled и переопределяет процедуры Initialize, Adjust и Finalize. Эти процедуры вызываются, когда объект создается, копируется и уничтожается, соответственно. Отметим, что вызов этих подпрограмм вставляется автоматически компилятором и программисту не нужно явно их вызывать. Аналогично обстоит дело с лимитируемыми типами. Различие в том, что используется Limited_Controlled тип, у которого отсутствует подпрограмма Adjust, поскольку копирование запрещено. Эти подпрограммы используются, чтобы реализовать сложную инициализацию, осуществлять глубокое копирование связных структур данных, освобождение памяти по окончании жизни объекта и тому подобной деятельности, специфичной для данного типа.

В качестве примера, снова рассмотрим стек, реализованный в виде связного списка, но предоставим возможность копирования. Напишем:

```
package Linked_Stacks is
   type Stack is limited private;
```

(continues on next page)

(continued from previous page)

```
   procedure Clear(S: out Stack);
   procedure Push(S: in out Stack; X: in Float);
   procedure Pop(S: in out Stack; X: out Float);
private

   type Cell is record
      Next: access Cell;
      Value: Float;
   end record;

   type Stack is new Controlled with record
      Header: access Cell;
   end record;

   overriding procedure Adjust(S: in out Stack);
end Linked_Stacks;
```

Тип Stack теперь приватный. Полное объявление типа - это теговый тип, порожденный от типа Controlled и имеющий компонент Header, аналогичный предыдущему объявлению стека. Это просто обертка. Клиент же не видит, что наш тип теговый и контролируемый. Чтобы присваивание работало корректно, мы переопределяем процедуру Adjust. Обратите внимание, что мы используем индикатор overriding, что заставляет компилятор проверить правильность параметров. Тело пакета может быть следующим:

```
package body Linked_Stacks is

   procedure Clear(S: out Stack) is
   begin
      S := (Controlled with Header => null);
   end Clear;

   procedure Push(S: in out Stack; X: in Float) is
   begin
      S.Header := new Cell'(S.Header, X);
   end Push;

   procedure Pop(S: in out Stack; X: out Float) is
   begin
      X := S.Header.Value;
      S.Header := S.Header.Next;
   end Pop;

   function Clone(L: access Cell) return access Cell is
   begin
      if L = null then
         return null;
      else
         return new Cell'(Clone(L.Next), L.Value);
      end if;
   end Clone;

   procedure Adjust (S: in out Stack) is
   begin
      S.Header := Clone (S.Header);
   end Adjust;
end Linked_Stacks;
```

Теперь присваивание будет работать как надо. Допустим, мы напишем:

```
This_One, That_One: Stack;

...

This_One := That_One;  --  автоматический вызов Adjust
```

Сперва выполняется побитовое копирование That_One в This_One, затем для This_One выполняется вызов Adjust, где вызывается рекурсивная функция Clone, которая и выполняет фактическое копирование. Часто такой процесс называют глубоким копированием. В результате This_One и That_One содержат одинаковые элементы, но их внутренние структуры никак не пересекаются.

Интересным моментом, также, может быть то, как в процедуре Clear устанавливается параметр S. Эта конструкция называется расширенным агрегатом. Первая часть агрегата — имя родительского типа, а часть после слова with предоставляет значения дополнительным компонентам, если такие имеются. Процедуры Push и Pop — тривиальны.

Читатель может спросить, что произойдет с занимаемой памятью после вызова процедуры Pop или Clear. Мы обсудим это в следующей главе, касающейся вопросов управления памятью.

Следует отметить, что процедуры Initialize и Finalize не переопределены и наследуются пустые процедуры от типа Controlled. Поэтому ничего дополнительного не выполняется в момент объявления стека. Это нам подходит, потому что компонента Header получает значение null по умолчанию, что нам и требуется. Аналогично, никаких действий не происходит, когда стек уничтожается, например при выходе из процедуры. Тут снова встает вопрос об освобождении памяти, к которому мы вернемся в следующей главе.

CHAPTER

THIRTYTHREE

БЕЗОПАСНОЕ УПРАВЛЕНИЕ ПАМЯТЬЮ

Компьютерная память, используемая программой, является критически важным ресурсом системы. Целостность его содержимого является необходимым условием здорового функционирования программы. Тут можно проследить аналогию с памятью человека. Когда память ненадежна, жизнь человека заметно ухудшается.

Есть две проблемы, которые связаны с вопросом управления памятью. Первая заключается в том, что информация может быть потеряна в случае, если она ошибочно перетирается другой информацией. Другая проблема в том, что память может быть утеряна после использования и, в конце концов, вся свободная память будет исчерпана, что приведет к невозможности сохранить нужную информацию. Эта проблема утечки памяти.

Утечка памяти является коварной проблемой, поскольку может не проявляться длительное время. Известны примеры из области управления химическим производством, когда казалось, что программа работала в течении нескольких лет. Ее перезапускали каждые три месяца по независимым причинам (перемещали кран, что приводило к остановке производства). Когда график перемещения крана изменился, программа должна была работать дольше, но в итоге сломалась после четырех месяцев непрерывной работы. Причина оказалась в утечке памяти, понемногу отгрызающей свободное пространство.

33.1 Переполнение буфера

Переполнение буфера - это общее название, используемое для обозначения нарушения информационной безопасности. Переполнение буфера может привести к искажению или чтению информации злоумышленником, либо случайно.

Эта проблема широко распространена в программах на C и C++ и зачастую связана с отсутствием проверок выхода за пределы массивов в этих языках. Мы встречались с подобной проблемой в главе «Безопасные типы данных» в примере с парой игральных костей.

Эта проблема не может возникнуть в Аде, поскольку в обычных условиях проверка индекса при обращении к массиву активирована. Эту проверку можно отключить, когда мы абсолютно уверены в поведении программы, но это может быть неблагоразумно, пока мы не доказали корректность программы формальным методом, например, используя инструментарий SPARK Examiner, который мы обсудим в главе 11.

Хотя, в подавляющем количестве случаев виновником переполнения буфера является отсутствие проверки индекса массива, другие свойства языка также могут его вызвать. Например, обозначение конца строки с помощью нулевого байта. Это приводит ко множеству мест в программе, где программист должен проверить этот маркер. Легко ошибиться, выписывая эти тесты так, чтобы они работали верно в любой ситуации. В результате это приводит к появлению узких мест, которые используются вирусами для проникновения в систему.

Другой распространенной причиной разрушения данных является использование некорректных значений указателей. Указатели в C трактуются как адреса и для них

разрешены арифметические операции. Как следствие, легко может возникнуть ситуация, когда значение указателя вычислено неверно. Запись по этому указателю разрушит какие-то данные.

В главе «Безопасные указатели» мы видели, что строгая типизация указателей и правила контроля доступности в Аде защищают нас от подобных ошибок, гарантируя, что объявленный объект не исчезнет, пока на него ссылаются другие объекты.

Таким образом фундаментальные свойства языка Ада защищают от случайной потери данных, связанной с разрушением содержимого памяти. Остаток этой главы мы посвятим проблеме утечки памяти.

33.2 Динамическое распределение памяти

Обычно языки программирования предоставляют три способа распределения памяти:

- глобальные данные существуют в течении всего времени работы программы, поэтому могут иметь постоянное положение в памяти и обычно распределяются статически;
- данные ,сохраненные в стеке, распределяются и освобождаются синхронно с вызовом подпрограмм;
- данные, распределенные динамически, время жизни которых не связанно с временем работы подпрограмм.

Секция common в Fortran — исторический пример глобального статического распределения, но подобные механизмы есть и в других языках. В Аде мы можем объявить:

```
package Calandar_Data is

   type Month is (Jan, Feb, Mar, ..., Nov, Dec);

   Days_In_Month: array (Month) of Integer :=
      (Jan => 31, Feb => 28, Mar => 31, Apr => 30,
       May => 31, Jun => 30, Jul => 31, Aug => 31,
       Sep => 30, Oct => 31, Nov => 30, Dec =>31);
end;
```

Память, выделяемая под Days_In_Month, будет, естественно, выделена в фиксированной глобальной области.

Стек - важный механизм распределения памяти во всех современных языках программирования. Отметим, что речь тут идет о механизме, связанном с реализацией распределения памяти, а не об объектах типа Stack из предыдущих глав. Стек используется для передачи параметров при вызове подпрограмм (в том числе передачи аргументов, хранения адреса возврата, сохранения промежуточных регистров, и т. д.), а также локальных переменных подпрограммы. В многозадачной программе, где несколько потоков управления исполняются параллельно, каждая задача имеет свой стек.

Вернемся к функции Nfv_2000 из примера вычисления процентных ставок:

```
procedure Nfv_2000 (X: Float) is
   Factor: constant Float := 1.0 + X/100.0;
begin
   return 1000.0 * Factor**2 + 500.0 * Factor - 2000.00;
end Nfv_2000;
```

Объект Factor обычно распределяется в стеке. Он создается при вызове функции и уничтожается при возврате. Все управление памятью происходит автоматически, благодаря механизму вызова/возврата подпрограмм. Отметим, что, хотя Factor является константой, он не является статическим объектом, потому что каждый вызов функции вычисляет для

него свое значение. Так как две задачи могут вызвать эту функцию одновременно, Factor нельзя распределить статически. Аналогично, параметр X также распределяется в стеке.

Теперь рассмотрим более сложный случай, когда подпрограмма объявляет локальный массив, чей размер неизвестен до момента исполнения. Например это может быть функция, возвращающая массив в обратном порядке:

```
function Rev (A: Vector) return Vector is
   Result: Vector(A'Range);
begin
   for K in A'Range loop
      Result (K) := A(A'First+A'Last-K);
   end loop;
   return Result;
end Rev;
```

где Vector объявлен, как неограниченный массив:

```
type Vector is array (Natural range <>) of Float;
```

Как объясняется в разделе «Массивы и ограничения» главы «Безопасные типы данных», эта запись означает, что Vector - это массив, но границы у разных объектов этого типа могут быть разные. Когда мы объявляем объект этого типа, мы должны предоставить границы. У нас может быть:

```
L: Integer := ...;    -- L может не быть статическим значением

My_Vector, Your_Vector: Vector (1 .. L);

...

Your_Vector := Rev (My_Vector);
```

В большинстве языков программирования нам бы пришлось распределить память для такого объекта динамически, поскольку размер объекта не известен заранее. На самом деле, это не является необходимым, поскольку стек может расти динамически, а память для локальных объектов всегда распределяется по принципу последний-зашел-первый-вышел. Такого рода требования возникают для простоты реализации языка. Приложив некоторые усилия во время дизайна и реализации языка, можно распределять такого рода объекты в стеке, сохранив при этом эффективность механизма вызова подпрограмм.

Хотя такое поведение не требуется согласно стандарту, все промышленные компиляторы всегда используют стек для хранения локальных данных. Эффективной техникой в этом случае является использование двух стеков, один для хранения адресов возврата и локальных данных фиксированного размера, а другой для данных переменного размера. Это позволит обрабатывать данные фиксированного размера столь же эффективно, но сохранит требуемую гибкость в распределении памяти. Кроме того, в Аде часто применяется контроль за исчерпанием стека. В этом случает при попытке превысить отведенный размер стека будет возбуждаться исключение Storage_Error.

Данный пример красиво реализуется в Аде. Реализация на C контрастирует своей сложностью ввиду того, что в C нет соответствующей абстракции массивов. Мы можем передать массив, как аргумент, но только при помощи указателя на массив. Кроме того, в C нельзя вернуть массив, как объект. Хотя мы можем определить функцию, которая переставляет элементы прямо в массиве, и требовать от пользователя создавать копию перед ее вызовом. При этом нужно быть осторожным, чтобы не испортить данные при перестановке. Проще будет разрешить пользователю передавать как указатель на аргумент, так и указатель на результат. Следующее затруднение состоит в том, что в C мы не можем определить размер массива. Нам придется размер передавать явно. Мы получаем еще один шанс допустить ошибку, передав значение, не соответствующее длине массива. В итоге мы получим

```
void rev(float* a, float* result, int length)

{

   for (k=0;k<length;k++)
      result[k]=a[length-k-1];
}

...

float my_vector[100], your_vector[100];

...

rev(my_vector, your_vector, 100);
```

Хотя эта глава посвящена управлению памятью, наверное стоит остановиться, чтобы перечислить риски и затруднения в этом коде на C.

- Массивы в C всегда индексируются, начиная с 0. Если прикладная область использует другую нумерацию, например с 1, может возникнуть путаница. В Аде нижняя граница присутствует всегда в явном виде.
- Длина массива должна передаваться отдельно, что создает риск получить неверную длину, либо перепутать длину и верхнюю границу массива. В Аде атрибуты массива неотделимы от массива.
- Адрес результата необходимо передавать отдельно. Появляется возможность перепутать два массива.
- Цикл для итерации по массиву нужно записывать явно, в то время как в Аде можно воспользоваться атрибутом 'Range.

Но мы отклонились от темы. Ключевой момент в том, что, если бы мы объявили локальный массив в C++, чей размер не задан статически:

```
void f(int n, ...)
{
   float a[]=new float[n];
```

то память под такой массив распределялась бы динамически, а не в стеке. В C пришлось бы использовать функцию malloc.

Основная опасность при динамическом распределении памяти в том, что она может быть потеряна после использования. Поскольку в языке Ада возможно создавать объекты произвольного размера в стеке, необходимость динамического распределения памяти значительно снижается, что повышает производительность и уменьшает риск утечки памяти.

33.3 Пулы памяти

Давайте рассмотрим динамическое распределение памяти. Для этого в Аде используются пулы памяти. Если мы создаем объект динамически, как например в процедуре Push из главы «Безопасное создание объектов»:

```
procedure Push(S: in out Stack; X: in Float) is
begin
   S := new Cell'(S, X);
end Push;
```

то память под новый Cell распределяется из пула памяти. Всегда существует стандартный пул памяти, но мы можем объявить и управлять своими собственными пулами памяти.

Первым языком программирования, избавившим программиста от необходимости управлением памятью, был LISP, благодаря использованию механизма сборки мусора. Этот механизм используется и в других языках, в том числе в Java и Python. Наличие сборщика мусора значительно упрощает программирование, но имеет свои проблемы. Например, сборщик мусора может приостанавливать исполнение программы непредсказуемым образом, что может привести к проблемам в системах реального времени. Программируя системы реального времени, необходимо тщательно контролировать распределение памяти, а также гарантировать время отклика программы, что может быть затруднительно при использовании сборщика мусора. Одной из причин, по которой в первый стандарт языка Ада 83 не включали средства ООП, было то, что автор языка, Жан Ишбиа, занимавшийся в свое время реализацией ООП языка Simula, был уверен в том, что ООП необходимо иметь сборщик мусора, а это неприемлемо для систем реального времени. Как впоследствии было продемонстрировано в C++ и Ада 95, язык может поддерживать ООП без сборщика мусора, если он предоставляет программисту развитые механизмы управления памятью.

Ада позволяет программисту выбрать один из следующих механизмов управления памятью:

- ручной режим. В этом случае программист освобождает память каждого объекта индивидуально.
- пул памяти. Объекты можно удалять, как каждый отдельно, так и весь пул целиком.
- сборщик мусора. Этот режим может быть реализован не во всех системах.

Чтобы удалить память, занимаемую некоторым объектом, нужно настроить предопределенную процедуру Unchecked_Deallocation. Для этого нужно предоставить именованный ссылочный тип. Вспомним тип Cell:

```
type Cell;

type Cell_Ptr is access all Cell;

type Cell is record
   Next: Cell_Ptr;
   Value: Float;
end record;
```

Обратите внимание, как здесь использовано неполное объявление типа, чтобы разорвать циклическую зависимость между типами. Теперь напишем:

```
procedure Free is new Unchecked_Deallocation (Cell, Cell_Ptr);
```

Чтобы удалить память, занимаемую объектом Cell, нужно вызвать процедуру Free и передать ей ссылку на удаляемый объект. Например, процедура Pop должна выглядеть так:

```
procedure Pop (S: in out Stack; X: out Float) is
   Old_S : Stack := S;
begin
   X := X.Value;
   S := S.Next;
   Free (Old_S);
end Pop;
```

Здесь мы используем Stack из примера с limited private, а не контролируемый тип.

Может показаться, что мы рискуем появлением висящих ссылок, поскольку могут быть другие ссылки, указывающие на удаленный объект. Но, в этом примере, с точки зрения пользователя, тип Stack лимитированный, следовательно ,пользователь не может сделать копию. Кроме того, пользователь не видит типов Cell и Cell_Ptr, поэтому не сможет вызвать Free. Это нам гарантирует корректность Pop. И, наконец, при настройке

Unchecked_Deallocation используется тип Cell_Ptr, что позволяет проверить тип аргумента при вызове Free.

Нам необходимо изменить и процедуру Clear. Простейший вариант такой:

```
procedure Clear (S: in out Stack) is
   Junk: Float;
begin
   while S /= null loop
      Pop (S, Junk);
   end loop;
end Clear;
```

Хотя эта техника позволяет гарантировать, что память очищается при вызове Pop и Clear, существует риск, что пользователь объявит стек и выйдет из его области видимости пока стек не пуст. Например

```
procedure Do_Something is
   A_Stack: Stack;
begin
   ... -- Используем стек
   ... -- Пуст ли стек при выходе?
end Do_Something;
```

Если стек не был пуст при выходе, то память будет потеряна. Мы не можем обременять пользователя заботой о таких деталях, поэтому мы должны сделать тип контролируемым, как было продемонстрировано в конце главы «Безопасное создание объектов». Мы переопределим процедуру Finalize так:

```
overriding procedure Finalize (S: in out Stack) is
begin
   Clear (S);
end Finalize;
```

Использовав индикатор overriding, мы заставляем компилятор проверить, что мы не ошиблись в написании Finalize или в формальных параметрах.

В Аде также есть возможность объявить свои пулы памяти. Это просто, но потребует слишком много места для описания всех подробностей здесь. Основная идея в том, что есть тип Root_Storage_Pool (это лимитированный контролируемый тип) и мы объявляем свой тип, наследуя от него

```
type My_Pool_Type (Size: Storage_Count) is
  new Root_Storage_Pool with private;

overriding procedure Allocate(...);

overriding procedure Deallocate(...);

-- также переопределим Initialize и Finalize
```

Процедура Allocate автоматически вызывается, когда создается новый объект при помощи new, а Deallocate — при вызове настройки Unchecked_Deallocation, такой как Free. Так мы реализуем необходимые действия по управлению памятью. Поскольку тип контролируемый, процедуры Initialize и Finalize автоматически вызываются при объявлении пула и его уничтожении.

Чтобы создать пул мы объявляем объект этого типа, как обычно. Наконец, необходимо привязать конкретный ссылочный тип к объекту-пулу.

```
Cell_Ptr_Pool: My_Pool_Type (1000); -- Размер пула — 1000

for Cell_Ptr'Storage_Pool use Cell_Ptr_Pool;
```

Важное преимущество пулов в том, что с их помощью можно уменьшить риск фрагментации памяти, если распределять объекты одного размера в одном пуле. Кроме того, мы можем написать свои алгоритмы распределения памяти или даже уплотнения, если захотим. Существует также возможность определить ссылочный тип локально, тогда и пул памяти можно определить локально и он будет автоматически удален по завершении подпрограммы, что исключит утечку памяти.

Пулы памяти были усовершенствованы в стандарте Ада 2012, где появились вложенные пулы. Мы не будем останавливаться здесь на этом, отметим лишь, что вложенные пулы — это части пула, уничтожаемые по отдельности.

Наконец, в качестве предохранителя от злоупотребления Unchecked_Deallocation, можно использовать тот факт, что Unchecked_Deallocation - это отдельный модуль компиляции. Следовательно, везде, где он используется, в начале текста будет:

```
with Unchecked_Deallocation;
```

Этот спецификатор контекста легко заметить при контроле качества программы.

33.4 Ограничения

Как мы уже знаем, есть общий механизм, гарантирующий отсутствие использования некоторых свойств языка, и это директива компилятору Restrictions. Если мы напишем:

```
pragma Restrictions(No_Dependence => Unchecked_Deallocation);
```

мы убедимся, что программа вообще не использует Unchecked_Deallocation — компилятор проверит это.

Существует около пятидесяти таких ограничений, которые контролируют различные аспекты программы. Многие из них узкоспециализированные и относятся к многозадачным программам. Другие касаются распределения памяти, например:

```
pragma Restrictions(No_Allocators);

pragma Restrictions(No_Implicit_Heap_Allocations);
```

Первый полностью запрещает использование конструкции new, как например new Cell'(...), а значит запрещает и динамическое распределение памяти вообще. Иногда, некоторые реализации используют динамическое распределение для хранения временных объектов. Это редкие случаи и второй вариант запрещает их появление.

CHAPTER

THIRTYFOUR

БЕЗОПАСНЫЙ ЗАПУСК

Мы можем тщательно написать программу, чтобы она правильно работала, но всё окажется бесполезно, если она не сможет корректно запуститься.

Машина, которая не заводится, никуда не годится, даже если она ездит как Rolls-Royce.

В случае с компьютерной программой, на старте необходимо убедиться, что все данные инициализированы правильно, зачастую это означает, что необходимо гарантировать, что операции инициализации выполняются в правильном порядке.

34.1 Предвыполнение

Типичная программа состоит из некоторого количества библиотечных пакетов P, Q, R и т. д. плюс главная подпрограмма M. При запуске программы пакеты предвыполняются, а затем вызывается главная подпрограмма. Предвыполнение пакета заключается в создании различных сущностей, объявленных в пакете на верхнем уровне. Но это не касается сущностей внутри подпрограмм, поскольку они создаются лишь в момент вызова этих подпрограмм.

Вернемся к примеру со стеком из главы «Безопасная Архитектура». Вкратце он выглядит так:

```
package Stack is
   procedure Clear;
   procedure Push(X: Float);
   function Pop return Float;
end Stack;

package body Stack is
   Max: constant := 100;
   Top: Integer range 0 .. Max := 0;
   A: array (1 .. Max) of Float;

   ... -- подпрограммы Clear, Push и Pop
end Stack;
```

Предвыполнение спецификации пакета не делает ничего, потому что не содержит объявлений объектов. Предвыполнение тела теоретически приводит к выделению памяти для целого Top и массива A. В данном случае, размер массива известен заранее, поскольку определяется константой Max, имеющей статическое значение. Следовательно, память под массив можно распределить заранее, до загрузки программы.

Но константа Max не обязательно должна иметь статическое значение. Она может принимать значение, например, вычисленное функцией:

```
Max: constant := Some_Function;

Top: Integer range 0 .. Max := 0;
```

(continues on next page)

(continued from previous page)

```
A: array (1 .. Max) of Float;
```

И тогда размер массива должен быть рассчитан при предвыполнении тела пакета. Если, по беспечности, мы объявим Max как переменную и забудем присвоить ей начальное значение:

```
Max: Integer;

Top: Integer range 0 .. Max := 0;

A: array (1 .. Max) of Float;
```

размер массива будет зависеть от случайного значения, которое примет Max. Если значение Max будет отрицательным, это приведет к возбуждению исключения Constraint_Error, а если Max будет слишком большим то будет исключение Storage_Error. (Отметим, что большинство компиляторов выдаст предупреждение, поскольку анализ потока данных обнаруживает ссылку на неинициализированную переменную.)

Следует также отметить, что мы инициализируем переменную Top нулем, чтобы пользователю не пришлось вызывать Clear перед первым вызовом Push или Pop.

Также мы можем добавить в тело пакета код для явной инициализации, например так:

```
package body Stack is
   Max: constant := 100;
   Top: Integer range 0 .. Max := 0;
   A: array (1 .. Max) of Float;

   ... -- подпрограммы Clear, Push и Pop

begin   -- далее явная инициализация
   Top := 0;
end Stack;
```

Явная инициализация может содержать любые инструкции. Она исполняется во время предвыполнения тела пакета и до момента, когда любая из подпрограмм пакета вызывается извне.

Может показаться, что всегда предоставлять значения по умолчанию для все переменных — это хорошая идея. В нашем примере значение «0» - весьма подходящее и соответствует состоянию стека после исполнения Clear. Но в некоторых случаях нет очевидного подходящего значения для инициализации. В этих случаях, использовать произвольное значение не разумно, поскольку это может затруднить обнаружение реальных ошибок. Мы еще вернемся к этому вопросу при обсуждении языка SPARK в заключительной главе.

В случае числовых значений, отсутствие значения по умолчанию не обязательно приведет к катастрофе. Но в случае ссылочного типа или других неявных представлений адреса в памяти, катастрофа более реальна. В языке Ада все ссылочные объекты будут иметь значение null по умолчанию, либо будут явно инициализированы.

Потенциальные ошибки в процессе инициализации тесно связаны с попыткой доступа до окончания предвыполнения. Рассмотрим код:

```
package P is
   function F return Integer;
   X: Integer := F;      -- возбудит Program_Error
end;
```

где тело F конечно находится в теле пакета P. Невозможно вызвать F, чтобы получить начальное значение X до того, как тело F будет предвыполнено, поскольку тело F может ссылаться на переменную X или любую другую переменную, объявленную после X, ведь они

все еще не инициализированы. Поэтому в Аде это приведет к возбуждению исключения Program_Error. В языке C подобные ошибки приведут к непредсказуемому результату.

34.2 Директивы компилятору, связанные с предвыполнением

Внутри одного модуля компиляции предвыполнение происходит в порядке объявления сущностей.

В случае программы, состоящей из нескольких модулей, каждый модуль предвыполняется после всех других, от которых он зависит. Так, тело предвыполняется после соответствующей спецификации, спецификация дочернего модуля — после спецификации родителя, все модули, указанные в спецификации контекста (без слова limited), предвыполняются до этого модуля.

Однако эти правила не полностью определяют порядок и их может быть недостаточно для гарантирования корректного поведения программы. Мы можем дополнить наш пример следующим образом:

```
package P is
   function F return Integer;
end P;

package body P is
   N: Integer := какое-то_значение;
   function F return Integer is
   begin
      return N;
   end F;
end P;


with P;
package Q is
   X: Integer := P.F;
end Q;
```

Здесь важно, чтобы тело пакета P предвыполнилось до спецификации Q, иначе получить начальное значение X будет невозможно. Предвыполнение тела P гарантирует корректную инициализацию N. Но вычисление его начального значения может потребовать вызова функций из других пакетов, а эти функции могут ссылаться на данные инициализированные в теле пакетов, содержащих эти функции. Поэтому необходимо убедиться, что не только тело P предвыполняется до спецификации Q, но и тела всех пакетов, от которых зависит P, также предвыполнены. Описанные выше правила не гарантируют этого поведения, как следствие на старте может быть возбуждено исключение Program_Error.

Мы можем потребовать желаемый порядок предвыполнения, вставив специальную директиву компилятору:

```
with P;
pragma Elaborate_All (P);

package Q is
   X: Integer := P.F;
end Q;
```

Отметим, что All в наименовании инструкции подчеркивает ее транзитивный характер. Как результат, предвыполнение пакета P (и всех его зависимостей) произойдет до предвыполнения кода Q.

Также существует инструкция Elaborate_Body, которую можно указать в спецификации, что приведет к тому, что тело предвыполнится сразу после спецификации. Во многих случаях этой инструкции достаточно, чтобы избежать проблем, связанных с порядком предвыполнения.

Здесь читатель может поинтересоваться, возможно ли решить проблему порядка предвыполнения, введя дополнительные простые правила в язык. Например, как если бы инструкция Elaborate_Body присутствовала всегда. К сожалению, это не сработает, поскольку полностью запретит взаимно рекурсивные пакеты (т. е. когда тело пакета P1 имеет спецификатор with P2; а P2 соответственно with P1).

Проблема нахождения правильного порядка предвыполнения может быть весьма сложной, особенно в больших системах. Иногда подобных проблем проще избежать, чем решать. Один из методов состоит в том, чтобы полностью отказаться от кода инициализации в телах пакетов, предоставив вместо этого главной подпрограмме явно вызывать процедуры для инициализации структур данных.

34.3 Динамическая загрузка

Вопрос динамической загрузки связан с вопросом запуска программы. Некоторые языки спроектированы так, чтобы создавать цельную согласованную программу, полностью собранную и загруженную к моменту исполнения. Среди них Ada, C и Pascal. Операционная система может выгружать из памяти и загружать обратно старницы с памятью программы, но это лишь детали реализации.

Другие языки созданы более динамическими. Они позволяют скомпилировать, загрузить и исполнить новый код прямо в процессе выполнения программы. COBOL, Java и C# среди них.

Для исполнения нового кода, в таких языках, как C, иногда используется механизм динамически загружаемых библиотек (DLL). Однако это не безопасно, поскольку при вызове подпрограмм отсутствует контроль параметров.

Подход, который можно предложить в Аде, использует механизм теговых типов для динамической загрузки. Смысл в том, что существующий код использует надклассовый тип (такой как Geometry.Object'Class) для вызова операций (таких как Area) любых конкретных новых типов (например пятиугольник, шестиугольник и пр.), и при этом создание новых типов не требует перекомпиляции существующего кода. Этого мы кратко касались в главе «Безопасное ООП». Этот механизм полностью безопасен с точки зрения строгой типизации.

Прекрасный пример, как с помощью этого подхода можно реализовать динамическую загрузку, можно найти в [7].

CHAPTER

THIRTYFIVE

БЕЗОПАСНАЯ КОММУНИКАЦИЯ

Программа, которая не взаимодействует с внешним миром каким-либо образом, бесполезна, хотя и очень безопасна. Можно сказать, что она находится в своего рода одиночном заключении. Преступник в одиночном заключении — безопасен, в том смысле, что он не может навредить другим людям, но в то же время он бесполезен для общества.

Поэтому, чтобы быть полезной, программа должна взаимодействовать. Даже если программа написана правильно и в ней нет внутренних изъянов, это не имеет особого значения, если ее взаимодействие с внешним миром само не является безопасным. Таким образом, безопасность коммуникации очень важна, поскольку именно через нее программа проявляет свои полезные свойства.

Тут, наверное, стоит обратиться к введению, где мы обсуждали понятия безопасности и надежности программы, определив их, как возможность программы причинить вред ее окружению и, наоборот, получить ущерб от окружения. Так вот, коммуникация программы является основой как для ее надежности, так и для безопасности.

35.1 Представление данных

Важным аспектом взаимодействия является отображение абстрактного программного обеспечения на конкретное исполняющее устройство. Большинство языков перекладывают этот вопрос на плечи реализации. Но Ада предоставляет развитые механизмы контроля над многими аспектами представления данных.

Например, данные некоторой записи должны располагается в памяти специфическим образом, чтобы соответствовать требуемой структуре файла. Допустим, речь идет о структуре Key из главы «Безопасное создание объектов»:

```
type Key is limited record
   Issued: Date;
   Code: Integer;
end record;
```

где тип Date, в свою очередь, имеет следующий вид:

```
type Date is record
   Day: Integer range 1 .. 31;
   Month: Integer range 1..12;
   Year : Integer;
end record;
```

Предположим, что наша целевая машина использует 32-х битные слова из четырех байт. Принимая во внимания ограничения диапазона, день и месяц свободно помещаются в один байт каждый и для года остается 16 бит (мы игнорируем «проблему 32768 года»). Вся запись красиво ложится в одно слово. Выразим это следующим образом:

```
for Date use record
   Day at 0 range 0 .. 7;
   Month at 1 range 0 .. 7;
   Year at 2 range 0 .. 15;
end record;
```

В случае с типом Key требуемая структура - это два слова и реализация почти наверняка выберет это представление по умолчанию. Но мы можем гарантировать это, написав:

```
for Key use record
   Issued at 0 range 0 .. 31;
   Code at 4 range 0 .. 31;
end record;
```

В качестве следующего примера рассмотрим тип Signal из главы «Безопасные типы данных»:

```
type Signal is (Danger, Caution, Clear);
```

Пока мы не укажем другого, компилятор будет использовать 0 для представления Danger, 1 для Caution и 2 для Clear. Но в реальной системе может потребоваться кодировать Danger как 1, Caution как 2 и Clear как 4. Мы можем потребовать использовать такую кодировку, написав спецификатор представления перечислимого типа:

```
for Signal use (Danger => 1, Caution => 2, Clear => 4);
```

Заметьте, что ключевое слово for здесь не имеет отношения к инструкции цикла for. Такой синтаксис был выбран в целях облегчения чтения программы.

Далее, допустим, мы хотим гарантировать, чтобы любой объект типа Signal занимал один байт. Это особенно актуально для компонент массивов и записей. Это может быть достигнуто с помощью следующей записи:

```
for Signal'Size use 8;
```

Для Ады 2012 возможна альтернативная запись в виде спецификации аспекта прямо в определении типа:

```
type Signal is (Danger, Caution, Clear)
   with Size => 8;
```

Развивая дальше этот пример, допустим, мы хотим чтобы переменная The_Signal этого типа находилась по адресу 0ACE. Мы можем добиться этого:

```
The_Signal: Signal;
for Signal'Address
   use System.Storage_Elements.To_Address (16#0ACE#);
```

Значение атрибута 'Address должно быть типа Address, который обычно является приватным. Для преобразования целочисленного значения к этому типу здесь вызывается функция To_Address из пакета System.Storage_Elements.

Эквивалентный способ с использованием аспекта в Ада 2012 выглядит так:

```
The_Signal: Signal
  with Address => System.Storage_Elements.To_Address (16#0ACE#);
```

35.2 Корректность данных

При получении данных из внешнего мира необходимо убедиться в их корректности. В большинстве случаев мы легко можем запрограммировать нужные проверки, но бывают и исключения.

Возьмем для примера опять тип Signal. Мы можем заставить компилятор использовать нужное нам кодирование. Но если по какой-то причине окажется, что значение не соответствует заданной схеме (например, пара бит были изменены из-за ошибки накопителя), то проверить это каким-либо условием будет невозможно, поскольку это выводит нас за область определения типа Signal. Но мы можем написать:

```
if not The_Signal'Valid then
```

Атрибут 'Valid применим к любому скалярному объекту. Он возвращает True, если объект содержит корректное, с точки зрения подтипа объекта, значение и False — в противном случае.

Мы можем поступить по-другому. Считать значение, например, как байт, проверить, что значение имеет корректный код, затем преобразовать его к типу Signal, использовав функцию Unchecked_Conversion. Объявим тип Byte и функцию преобразования:

```
type Byte is range 0 .. 255;
for Byte'Size use 8;

--  Или, в Ада 2012
type Byte is range 0 .. 255 with Size => 8;

function Byte_To_Signal is new
    Unchecked_Conversion(Byte, Signal);
```

затем:

```
Raw_Signal: Byte;
for Raw_Signal'Address use To_Address(16#ACE#);

--  Или, в Ада 2012
Raw_Signal: Byte with Address => To_Address(16#ACE#);
The_Signal: Signal;

case Raw_Signal is
   when 1 | 2 | 4 =>
      The_Signal := Byte_To_Signal (Raw_Signal);
   when others =>
      ... -- Значение не корректно
 end case;
```

Поскольку Byte - это обычный целочисленный тип, мы можем использовать любые арифметические операции, чтобы проверить полученное значение. При получении некорректного значения мы можем предпринять соответствующие действия, например запротоколировать его и т. д.

Отметим, что данный код не будет работать, если загружаемое значение меняется самопроизвольно. Значение может поменяться после проверки и до преобразования. Поэтому, сначала значение должно быть скопировано в локальную переменную.

35.3 Взаимодействие с другими языками

Многие большие современные системы написаны на нескольких языках программирования, каждый из которых больше подходит для своей части системы. Части с повышенными требованиями к безопасности и надежности могут быть написаны на Аде (например, с использованием SPARK), графический интерфейс на C++, какой-нибудь сложный математический анализ на Fortran, драйвера устройств на C, и т. д.

Многие языки имеют механизмы взаимодействия с другими языками (например, использование C из C++), хотя обычно они определены довольно неряшливо. Уникальность языка Ады в том, что он предоставляет четко определенные механизмы взаимодействия с другими языками в целом. В Аде есть средства взаимодействия с языками C, C++, Fortran и COBOL. В частности, Ада поддерживает внутреннее представление типов данных из этих языков, в том числе учитывает порядок строк и столбцов в матрицах Fortran и представление строк в C.

Во многоязычной системе имеет смысл использовать язык Ада, как центральный, чтобы воспользоваться преимуществами ее системы контроля типов.

Средства взаимодействия в своей основе используют директивы компилятору (pragma). Предположим, существует функция next_int на языке C и мы хотим вызвать ее из Ады. Достаточно написать:

```
function Next_Int return Interfaces.C.int;

pragma Import (C, Next_Int);
```

Эта директива указывает, что данная функция имеет соглашение о вызовах языка C, и что в Аде не будет тела для этой функции. В ней также возможно указать желаемое внешнее имя и имя для редактора связей, если необходимо. Предопределенный пакет Interfaces.C предоставляет объявления различных примитивных типов языка C. Использование этих типов позволяет нам абстрагироваться от того, как типы языка Ада соотносятся с типами языка C.

Аналогично, если мы хотим дать возможность вызывать процедуру Action, реализованную на языке Ада из C, мы сделаем имя этой процедуры доступным извне, написав:

```
procedure Action(X,Y: in Interfaces.C.int);

pragma Export (C, Action);
```

Ссылки на подпрограммы играют важную роль во взаимодействии с другими языками, особенно при программировании интерактивных систем. Предположим, мы хотим, чтобы процедура Action вызывалась из графического интерфейса при нажатии на кнопку мыши. Допустим, есть функция set_click на C, принимающая адрес подпрограммы, вызываемой при нажатии. На Аде мы выразим это так:

```
type Response is access procedure (X,Y: in Interfaces.C.int);
pragma Convention (C, Response);

procedure Set_Click(P: in Response);
pragma Import(C, Set_Click);

procedure Action(X,Y: in Interfaces.C.int);
pragma Convention(C, Action);
...
Set_Click(Action'Access);
```

В этом случае, мы не делаем имя процедуры Action видимым из программы на C, поскольку ее вызов будет происходить косвенно, но мы гарантируем, что соглашения о вызовах будут соблюдены.

35.4 Потоки ввода/вывода

При передаче значений различных типов во внешний мир и обратно мы можем столкнуться с некоторыми трудностями. Вывод значений достаточно прямолинеен, поскольку мы заранее знаем типы этих значений. Ввод может быть проблематичным, так как мы обычно не знаем, какой тип может прийти. Если файл содержит значения одного типа, то все просто, нужно лишь убедиться, что мы не перепутали файл. Настоящие затруднения начинаются, если файл содержит значения различных типов. Ада предлагает механизмы работы как с гомогенными файлами (например, файл из целых чисел или текстовый файл), так и с гетерогенными файлами. Механизм работы с последними использует потоки.

В качестве простого примера, рассмотрим файл, содержащий значения Integer, Float и Signal. Все типы имеют специальные атрибуты 'Read и 'Write для использования потоков. Для записи мы просто напишем:

```
S: Stream_Access := Stream(The_File);
...
Integer'Write(S, An_Integer);
Float'Write(S, A_Float);
Signal'Write(S, A_Signal);
```

и в результате получим смесь значений в файле The_File. Для экономии места опустим детали, скажем только, что S обозначает поток, связанный с файлом.

При чтении мы напишем обратное:

```
Integer'Read(S, An_Integer);
Float'Read(S, A_Float);
Signal'Read(S, A_Signal);
```

Если мы ошибемся в порядке, от получим исключение Data_Error в том случае, если прочитанные биты не отображают корректное значение нужного типа.

Если мы не знаем заранее, в каком порядке идут данные, мы должны создать класс, покрывающий все возможные типы. В нашем случае, объявим базовый тип:

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

выполняющий роль обвертки, затем серию типов для возможных вариантов:

```
type S_Integer is new Root with record
   Value: Integer;
end record;

type S_Float is new Root with record
   Value: Float;
end record;
```

и так далее. Запись примет следующий вид:

```
Root'Class'Output(S, (Root with An_Integer));
Root'Class'Output(S, (Root with A_Float));
Root'Class'Output(S, (Root with A_Signal));
```

Заметьте, что тут во всех случаях используется одна и та же подпрограмма. Она сначала записывает значение тега конкретного типа, а затем вызывает (с помощью диспетчеризации) соответствующий 'Write атрибут.

При чтении мы могли бы написать:

```
Next_Item: Root'Class := Root'Class'Input(S);
...
Process(Next_Item);
```

Процедура Root'Class'Input читает тег из потока и затем вызывает нужный 'Read атрибут для чтения всего объекта, а затем присваивает полученное значение переменной Next_Item. Далее мы можем обработать полученное значение, например, вызвав диспетчеризируемую подпрограмму Process. Пусть ее задачей будет присвоить полученное значение нужной переменной, согласно типу.

Для начала объявим абстрактную процедуру для базового типа:

```
procedure Process(X: in Root) is abstract;
```

затем конкретные варианты:

```
overriding procedure Process(X: S_Integer) is
begin
   An_Integer := X.Value; -- достаем значение из обвертки
end Process;
```

Безусловно, процедура Process может делать все, что мы захотим с полученным значением.

Возможно, данный пример выглядит немного искусственным. Его цель проиллюстрировать, что в Аде возможно обрабатывать элементы различных типов, сохранив при этом безопасную строго типизированную модель данных.

35.5 Фабрики объектов

Мы только что видели, как стандартный механизм потоков позволяет нам обрабатывать значения различных типов, поступающих из файла. Лежащий в основе механизм чтения тега и затем создания объекта соответствующего типа стал доступен пользователю в Аде 2005.

Допустим, мы обрабатываем геометрические объекты, которые мы обсуждали в главе «Безопасное ООП». Различные типы, такие как Circle, Square, Triangle и т. д. наследуются от базового типа Geometry.Object. Нам может понадобиться вводить эти объекты с клавиатуры. Для окружности нам понадобиться две координаты и радиус. Для треугольника — координаты и длины трех сторон. Мы могли бы объявить функцию Get_Object для создания объектов, например для окружности:

```
function Get_Object return Circle is
begin
   return C: Circle do
      Get(C.X_Coord); Get(C.Y_Coord); Get(C.Radius);
   end return;
end Get_Object;
```

Внутренние вызовы Get это стандартные процедуры для чтения значения примитивных типов с клавиатуры. Пользователь должен ввести какой-то код, чтобы обозначить, какого рода объект он хочет создать. Предположим, что ввод окружности обозначается строкой "Circle". Также предположим, что у нас уже есть функция для чтения строки Get_String.

Теперь нам остается только прочесть код и вызвать соответствующую процедуру Get_Object для создания объекта. Для этого мы воспользуемся предопределенной настраиваемой функцией, которая получает тег и возвращает сконструированный объект. Вот ее определение:

```
generic
   type T(<>) is abstract tagged limited private;
   with function Constructor return T is abstract;
function Generic_Dispatching_Constructor(The_Tag: Tag)
   return T'Class;
```

Эта настраиваемая функция имеет два параметра настройки. Первый определяет конструируемый класс типов (в нашем случае Geometry.Object, от которого происходят Circle, Square и Triangle). Второй - это диспетчеризируемая функция для создания объектов (в нашем случае Get_Object).

Теперь мы настроим ее и получим функцию создания геометрических объектов:

```
function Make_Object is new
  Generic_Dispatching_Constructor(Object, Get_Object);
```

Эта функция принимает тег типа создаваемого объекта, вызывает соответствующую Get_Object и возвращает полученный результат.

Мы могли бы определить ссылочную переменную для хранения вновь созданных объектов:

```
Object_Ptr: access Object'Class;
```

Если тег хранится в переменной Object_Tag (имеющую тип Tag из пакета Ada.Tags, там же объявлена и функция Generic_Dispatching_Constructor), то мы вызовем Make_Object так:

```
Object_Ptr := new Object'(Make_Object(Object_Tag));
```

и получим новый объект (например окружность), чьи координаты и радиус были введены с клавиатуры.

Чтобы закончить пример, нам осталось научиться преобразовывать строковые коды объектов, такие как "Circle", в теги. Простейший случай сделать это — определить атрибут 'External_Tag:

```
for Circle'External_Tag use "Circle";

for Triangle'External_Tag use "Triangle";
```

тогда прочесть строку и получить тег можно так:

```
Object_Tag: Tag := Internal_Tag(Get_String);
```

Конечно, использовать отдельную переменную Object_Tag не обязательно, поскольку мы можем объединить эти операции в одну:

```
Object_Ptr := new Object'(Make_Object(Internal_Tag(Get_String));
```

Напоследок следует заметить, что приведенный код немного упрощен. На самом деле настраиваемый конструктор имеет вспомогательный параметр, который мы опустили.

CHAPTER

THIRTYSIX

БЕЗОПАСНЫЙ ПАРАЛЛЕЛИЗМ

В реальной жизни многие процессы протекают одновременно. Люди делают несколько вещей одновременно с поразительной легкостью. Кажется, женщины преуспевают в этом лучше мужчин, возможно потому, что им нужно баюкать малыша, одновременно готовя еду и отгоняя тигра от пещеры. Мужчины же обычно концентрируются на одной проблеме за раз, ловят кролика на обед, либо ищут большую пещеру, либо, возможно, даже изобретают колесо.

Традиционно компьютер делает одно действие в каждый конкретный момент времени, а операционная система затем делает вид, будто несколько процессов выполняются одновременно. Положение дел меняется в наши дни, поскольку многие машины сейчас имеют несколько процессоров или ядер, но это все еще так, когда речь идет о огромном числе небольших систем, в том числе используемых в управлении производством.

36.1 Операционные системы и задачи

Степень параллельности, доступная в разных операционных системах, может быть совершенно разная. Операционные системы с поддержкой POSIX позволяют программисту создать несколько потоков управления. Эти потоки исполняют программу довольно независимо друг от друга и, таким образом, реализуют параллельное программирование.

На некоторых системах есть всего один процессор и он будет исполнять различные потоки в соответствии с некоторым алгоритмом планирования. Один из вариантов — просто выделять небольшой интервал времени каждому потоку по очереди. Более сложные алгоритмы (особенно для систем реального времени) используют приоритеты и предельные сроки, чтобы гарантировать, что процессор используется эффективно.

Другие системы имеют несколько процессоров. В этом случае некоторые потоки выполняются действительно параллельно. Здесь тоже используется планировщик, который распределяет процессоры для исполнения активных потоков по возможности эффективно.

В языках программирования параллельные процессы обычно называют потоки или задачи. Здесь мы используем второй вариант, который соответствует терминологии Ады. Существуют различные подходы к вопросу параллельности в разных языках. Одни предлагают многозадачные средства, встроенные непосредственно в язык. Другие просто предоставляют доступ к соответствующим примитивам операционной системы. Есть и такие, которые полностью игнорируют этот вопрос.

Ада, Java, C# и последние версии C++ среди тех языков, где поддержка параллельности встроена в язык. В C нет такой поддержки и программистам приходится пользоваться внешними библиотеками или напрямую вызывать сервисы операционной системы.

Есть, как минимум, три преимущества, когда язык имеет встроенные средства поддержки многозадачности:

- Встроенный синтаксис существенно облегчает написание корректных программ, поскольку язык может предотвратить появление множества видов ошибок. Здесь

опять проявляет себя принцип абстракции. Мы избегаем ошибок, скрывая различные низкоуровневые детали.

- При использовании средств операционной системы напрямую существенно затрудняется переносимость, поскольку одна система может значительно отличаться от другой.
- Операционные системы общего назначения не предоставляют средства, необходимые для различных приложений систем реального времени.

Многозадачной программе обычно нужны следующие условия:

- Необходимо предотвратить нарушение целостности данных, когда нескольким задачам требуется доступ к одним данным.
- Необходимо предоставить средства межзадачного взаимодействия для передачи данных от одной задачи к другой.
- Необходимы средства управления задачами с целью гарантирования специфичных временных условий.
- Необходим планировщик выполнения задач для эффективного использования ресурсов и попадания в установленные временные границы.

В этой главе мы кратко остановимся на этих вопросах и продемонстрируем надежные способы их решения, применяемые в языке Ада. Реализация многозадачности влечет целый спектр сложных моментов, поскольку написать корректную многозадачную программу намного труднее, чем чисто последовательную. Но сначала мы рассмотрим простую концепцию задачи в языке Ада и общую структуру программы.

В Аде программа может иметь множество задач, исполняющихся одновременно. Задача имеет две части, так же, как и пакет. Первая часть - это спецификация задачи. Она описывает интерфейс взаимодействия с другими задачами. Вторая часть - это тело задачи, где описывается, что собственно происходит. В простейшем случае спецификация задает лишь имя задачи и выглядит так:

```
task A; --  Спецификация задачи

task body A is      --  Тело задачи
begin
   ...   --  инструкции, определяющие что делать
end A;
```

Бывают случаи, когда удобно иметь несколько одинаковых задач, тогда мы объявляем задачный тип:

```
task type Worker;

task body Worker is ...
```

После чего мы можем объявить несколько задач так же, как мы объявляем объекты:

```
Tom, Dick, Harry: Worker;
```

Тут мы создали три задачи Tom, Dick, Harry. Мы можем объявлять массивы задач, делать компоненты записи и прочее. Задачи можно объявлять всюду, где можно объявлять объекты, например в пакете, в подпрограмме или даже в другой задаче. Не удивительно, что задачи имеют лимитированный тип, поскольку нет смысла присваивать одной задаче другую.

Главная подпрограмма всей программы вызывается из так называемой задачи окружения. Именно эта задача выполняет предвыполнение пакетов библиотечного уровня, как описано в главе «Безопасный запуск». Таким образом, программу с тремя пакетами A, B и C и главной процедурой Main можно представить, как:

```
task type Environment_Task;

task body Environment_Task is
   ... -- объявления пакетов A, B, C
   ... -- и главной процедуры Main
begin
   ... -- вызов процедуры Main
end Environment_Task;
```

Задача становится активной сразу после объявления. Она завершается, когда исполнение доходит до конца тела задачи. Существует важное правило, локальная задача (т. е. та, что объявлена внутри подпрограммы, блока или другой задачи) должна завершиться до того, как управление покинет охватывающий ее блок. Исполнение окружающего блока приостанавливается до тех пор, пока вложенная задача не завершится. Это правило предотвращает появление висящих ссылок на несуществующие более объекты.

36.2 Защищенные объекты

Допустим, три задачи - Tom, Dick и Harry используют общий стек для временного хранения данных. Время от времени одна из них кладет элемент в стек, затем, время от времени, одна из них (возможна та же, а возможно другая) достает данные из стека.

Три задачи исполняются параллельно, возможно на многопроцессорной машине, либо под управлением планировщика на однопроцессорной машине, в которой ОС выделяет кванты процессорного времени каждой задаче. Допустим второе и кванты длинной 10мс выделяются задачам по очереди.

Пусть задачи используют стек из главы «Безопасная архитектура». Пусть квант, выделенный задаче Harry, истекает при вызове Push, затем управление передается задаче Tom, вызывающей Pop. Говоря более конкретно, пусть Harry теряет управление сразу после увеличения переменной Top

```
procedure Push(X: Float) is
begin
   Top := Top + 1; -- после этого Harry теряет управление
   A(Top) := X;
end Push;
```

В этот момент Top уже имеет новое значение, но новое значение X еще не помещено в массив. Когда задача Tom вызовет Pop, она получит старое, скорее всего бессмысленное значение, которое должно было быть переписанным новым значением X. Когда задача Harry получит управление назад (допустим к этому моменту не было других операций со стеком), она запишет значение X в элемент массива, который находится за вершиной стека. Другими словами, значение X будет потеряно.

Еще хуже обстоят дела, когда задачи переключаются посреди выполнения инструкции языка. Например, Harry считал значение Top в регистр, но новое значение Top не успел сохранить, и тут переключился контекст. Далее, пусть Dick вызывает Push, таким образом увеличивает Top на единицу. Когда Harry продолжит исполнение, он заменит Top устаревшим значением. Таким образом, два вызова Push увеличивают Top лишь на 1, вместо 2.

Такое нежелательное поведение может быть предотвращено благодаря использованию защищенного объекта для хранения стека. Такая возможность появилась в стандарте Ада 95. Мы напишем:

```
protected Stack is
   procedure Clear;
```

(continues on next page)

(continued from previous page)

```
   procedure Push(X: in Float);
   procedure Pop(X: out Float);
private
   Max: constant := 100;
   Top: Integer range 0 .. Max := 0;
   A: Float_Array(1 .. Max);
end Stack;

protected body Stack is

   procedure Clear is
   begin
      Top := 0;
   end Clear;

   procedure Push(X: in Float) is
   begin
      Top := Top + 1;
      A(Top) := X;
   end Push;

   procedure Pop(X: out Float) is
   begin
      X := A(Top);
      Top := Top - 1;
   end Pop;

end Stack;
```

Отметьте, как package поменялось на protected, данные из тела пакета переместились в private часть, функция Pop превратилась в процедуру. Мы предполагаем, что тип Float_Array объявлен в другом месте, как array (Integer range <>) of Float.

Три процедуры Clear, Push и Pop называются защищенными операциями и вызываются аналогично обычным процедурам. Отличие состоит в том, что только одна задача может получить доступ к операциям объекта в один момент времени. Если задача, такая как Tom, пытается вызвать процедуру Pop, пока Harry исполняет Push, то Tom будет приостановлен, пока Harry не покинет Push. Это выполняется автоматически, без каких-то усилий со стороны программиста. Таким образом мы избежим несогласованности данных.

За кулисами защищенного объекта лежит механизм блокировок. Перед исполнением операции этого объекта, задача должна сначала захватить блокировку. Если другая задача уже захватила блокировку, первая задача будет ждать, пока другая задача закончит исполнять операцию и отпустит блокировку. (Блокировка может быть реализована с помощью примитивов операционной системы, но также возможны реализации с меньшими накладными расходами.)

Мы можем усовершенствовать наш пример, чтобы показать, как справиться с переполнением или опустошением стека. В первом варианте обе этих ситуации приводят к исключению Constraint_Error. В случае с Push, это происходит при попытке присвоить переменной Top значение Max+1; аналогичная проблема проявляется с Pop. При возбуждении исключения блокировка автоматически снимется, когда исключение завершит вызов процедуры.

Чтобы избежать переполненния и опустошения стека, мы используем барьеры:

```
protected Stack is

   procedure Clear;
   entry Push(X: in Float);
   entry Pop(X: out Float);
```

(continues on next page)

(continued from previous page)

```
private
   Max: constant := 100;
   Top: Integer range 0 .. Max := 0;
   A: Float_Array(1 .. Max);
end Stack;

protected body Stack is

   procedure Clear is
   begin
      Top := 0;
   end Clear;

   entry Push(X: in Float) when Top < Max is
   begin
      Top := Top + 1;
      A(Top) := X;
   end Push;

   entry Pop(X: out Float) when Top > 0 is
   begin
      X := A(Top);
      Top := Top - 1;
   end Pop;

end Stack;
```

Операции Push и Pop теперь входы (entry), а не процедуры, и у них появились барьеры, логические условия, такие как Top < Max. Вход не может принять исполнение, пока условие его барьера ложно. Заметьте, что это не значит, что такой вход нельзя вызвать. Просто вызывающая задача будет приостановлена до тех пор, пока условие не станет истинно. Например, если задача Harry пытается вызвать Push, когда стек заполнен, она должна дождаться, пока какая-нибудь другая задача (Tom или Dick) вызовет Pop и освободит верхний элемент. После этого исполнение Harry автоматически продолжится. Это произойдет без дополнительных действий программиста.

Заметьте, что вызов входа или защищенной процедуры выполняется так же, как и вызов обычной процедуры

```
Stack.Push (Z);
```

Подведем итог. Механизм защищенных объектов языка Ада обеспечивает эксклюзивный доступ к общим данным. В видимой части защищенного объекта объявляются защищенные операции, а защищаемые объектом компоненты объявляются в приватной части. Тело защищенного объекта содержит реализацию защищенных операций. Защищенные процедуры и входы предоставляют возможность читать/писать защищаемые данные, в то время, как защищенные функции — только читать. Это ограничение позволяет нескольким задачам читать общие данные одновременно (при использовании защищенных функций), но лишь одна задача может менять их. Из-за запрета защищенным функциям изменять данные нам пришлось переписать Pop как процедуру, хотя в изначальном варианте это была функция.

Аналогично задачам, мы можем объявить защищенный тип, чтобы использовать его как шаблон для создания защищенных объектов. Аналогично задачному типу, защищенный тип также является лимитированным.

Было бы поучительно рассмотреть, как мы запрограммировали бы этот пример, используя низкоуровневые примитивы. Исторически сложилось, что таким примитивом считается объект семафор, у которого определены две операции P (захватить) и V (освободить). Эффект операции P(sem) состоит в захвате блокировки, соответствующей sem, если блокировка свободна, в противном случае задача приостанавливается и помещается в очередь к sem.

Эффект V(sem) состоит в том, чтобы снять блокировку и разбудить одну из задач очереди, если она есть.

Чтобы обеспечить эксклюзивный доступ к данным, мы должны окружить каждую нашу операцию парой вызовов P и V. Например Push будет таким:

```
procedure Push(X: in Float) is
begin
   P(Stack_Lock); --  захватить блокировку
   Top := Top + 1;
   A(Top) := X;
   V(Stack_Lock); --  освободить блокировку
end Push;
```

Аналогично делается для подпрограмм Clear и Pop. Так обычно пишут многозадачный код на ассемблере. При этом есть множество возможностей совершить ошибку:

- Можно пропустить одну из операций P или V, нарушив баланс блокировок.
- Можно забыть поставить обе операции и оставить нужный код без защиты.
- Можно перепутать имя семафора.
- Можно случайно обойти вызов закрывающей операции V при исполнении.

Последняя ошибка могла бы возникнуть, например, если в варианте без барьеров, Push вызывается при заполненном массиве. Это приводит к возбуждению исключения Constraint_Error. Если мы не напишем обработчик исключения, где будем вызывать V, объект останется заблокированным навсегда.

Все эти трудности не возникают, если пользоваться защищенными объектами языка Ада, поскольку все низкоуровневые действия выполняются автоматически. Если действовать осторожно, можно обойтись семафорами в простых случаях, но очень сложно получить правильный результат в более сложных ситуациях, таких, как наш пример с барьерами. Сложно не только написать правильную программу, но также чрезвычайно сложно доказать, что программа корректна.

Входы с барьерами являются механизмом более высокого уровня, чем механизм «условных переменных», который можно найти в других языках программирования. Например, в языке Java, программист обязан явно вызывать wait, notify и notifyAll для переменных, отражающих состояния объекта, такие как «стек полон» и «стек пуст». Этот подход чреват ошибками и подвержен ситуации гонки приоритетов, в отличии от механизмов Ады.

36.3 Рандеву

Еще одним средством поддержи многозадачности является возможность непосредственного обмена данными между двумя задачами. В Аде это реализовано с помощью механизма рандеву. Две взаимодействующие задачи вступают в отношение клиент-сервер. Сервер должен быть известен клиенту, нуждающемуся в каком-то его сервисе. В то же время серверу безразлично, какого клиента он обслуживает.

Вот общий вид сервера, предоставляющего единственный сервис:

```
task Server is
   entry Some_Service(Format: in out Data);
end;

task body Server is
begin
   ...
   accept Some_Service(Format: in out Data) do
```

(continues on next page)

(continued from previous page)

```
      ... -- код предоставляющий сервис
   end Some_Service;
end Server;
```

По спецификации сервера видно, что он имеет вход Some_Service. Этот вход может быть вызван точно так же, как вход защищенного объекта. Отличие в том, что код, предоставляющий сервис, находится в соответствующей инструкции принятия (accept), которая исполняется лишь когда до нее доходит поток исполнения сервера. До этого момента вызывающая задача будет ожидать. Когда сервер достигнет инструкции принятия, она будет исполнена, используя любые параметры, переданные клиентом. Клиент будет ожидать окончания исполнения инструкции принятия, после чего все параметры out и in out получат новые значения.

Тело клиента может выглядеть так:

```
task body Client is
   Actual: Data;
begin
   ...
   Server.Some_Service (Actual);
   ...
end Client;
```

Каждый вход имеет соответствующую очередь. Если задача вызывает вход, а сервер в этот момент не ожидает на инструкции принятия, то задача ставится в очередь. С другой стороны, если сервер достигает инструкции принятия, а очередь пуста, то останавливается сервер. Инструкция принятия может находиться в любом месте в теле задачи, где допускаются инструкции, например, в одной из ветвей условной инструкции (if) или внутри цикла. Этот механизм очень гибкий.

Рандеву - это механизм высокого уровня (как и защищенные объекты), следовательно, его легко применять правильно. Соответствующий низкоуровневый механизм очередей тяжело использовать без ошибок.

Теперь приведем пример использования рандеву, в котором клиенту не нужно ожидать. Идея в том, что клиент передает серверу ссылку на вход, который необходимо вызвать, когда работа будет выполнена. Сначала мы объявим своего рода почтовый ящик, для обмена элементами некоторого типа Item, который определен где-то ранее:

```
task type Mailbox is
   entry Deposit(X: in Item);
   entry Collect(X: out Item);
end Mailbox;

task Mailbox is
   Local: Item;
begin
   accept Deposit(X: in Item) do
      Local := X;
   end;
   accept Collect(X: out Item) do
      X := Local;
   end;
end Mailbox;
```

Мы можем положить элемент в Mailbox, чтобы забрать его позже. Клиент передаст ссылку на почтовый ящик, куда сервер положит элемент, а клиент заберет его там позже. Нам понадобится ссылочный тип:

```
type Mailbox_Ref is access Mailbox;
```

Клиент и сервер будут следующего вида:

```
task Server is
   entry Request(Ref: in Mailbox_Ref; X: in Item);
end;

task body Server is
   Reply: Mailbox_Ref;
   Job: Item;
begin
   loop
      accept Request(Ref: in Mailbox_Ref; X: in Item) do
         Reply := Ref;
         Job := X;
      end;
      ...
      -- выполняем работу
      Reply.Deposit(Job);
   end loop;
end Server;

task Client;

task body Client is
   My_Box: Mailbox_Ref := new Mailbox;
   -- создаем задачу-почтовый ящик
   My_Item: Item;
begin
   Server.Request(My_Box, My_Item);
   ...
   -- занимаемся чем-то пока ждем
   My_Box.Collect(My_Item);
end Client;
```

На практике клиент мог бы время от времени проверять почтовый ящик. Это легко реализовать, используя условный вызов входа:

```
select
   My_Box.Collect (My_Item);
   -- успешно получили элемент
else
   -- элемент еще не готов
end select;
```

Почтовый ящик служит нескольким целям. Он отделяет операцию «положить элемент» от операции «взять элемент», что позволяет серверу сразу заняться следующим заданием. Кроме этого, он позволяет серверу ничего не знать о клиенте. Необходимость прямого вызова клиента привела бы к необходимости всегда иметь клиентов одного конкретного задачного типа, что совсем непрактично. Почтовый ящик позволяет нам очертить единственное необходимое свойство клиента — существование вызова Deposit.

36.4 Ограничения

Директива компилятору pragma Restrictions, используемая для запрета использования некоторых возможностей языка, уже упоминалась в главах «Безопасное ООП» и «Безопасное управление памятью».

Существует множество ограничений, касающихся многозадачности. Некоторые были известны еще со времен Ады 95, другие добавлены в Аде 2005 и 2012. Возможности Ады в этой области очень обширны. С их помощью можно создавать совершенно разные приложения реального времени. Но многие из них очень просты и не требуют использования всех возможностей языка. Вот некоторые примеры возможных ограничений:

- No_Task_Hierarchy
- No_Task_Termination
- Max_Entry_Queue_Length => n

Ограничение No_Task_Hierarchy предотвращает создание задач внутри других задач или подпрограмм, таким образом, все задачи будут в пакетах библиотечного уровня. Ограничение No_Task_Termination означает, что все задачи будут исполняться вечно, это часто встречается во многих управляющих приложениях, где каждая задача содержит бесконечный цикл, исполняющий какое-то повторяющееся действие. Следующее ограничение обуславливает максимальное количество задач, ожидающих на одном входе в любой момент времени.

Указание ограничений может дать возможность:

- использовать упрощенную версию библиотеки времени исполнения. В результате можно получить меньшую по объему и более быструю программу, что существенно в области встраиваемых систем.
- формально обосновать некоторые свойства программы, такие как детерминизм, отсутствие взаимных блокировок, способность уложится в указанные сроки исполнения.

Существует множество других ограничений, касающихся многозадачности, которые мы не рассмотрели.

36.5 Ravenscar

Особенно важная группа ограничений налагается профилем Ravenscar, который был разработан в середине 1990-х и стандартизирован, как часть языка Ада 2005. Чтобы гарантировать, что программа соответствует этому профилю, достаточно написать:

```
pragma Profile(Ravenscar);
```

Использование любой из запрещенных возможностей языка (на них мы остановимся далее) приведет к ошибке компиляции.

Главной целью профиля Ravenscar является ограничение многозадачных возможностей таким образом, чтобы эффект от программы стал предсказуемым. (Профиль был определен Международным Симпозиумом по вопросам Реального Времени языка Ада, который проходил дважды в отдаленной деревне Равенскар на побережье Йоркшира в северо-восточной Англии.)

Профиль определен как набор ограничений плюс несколько дополнительных директив компилятору, касающихся планировщика и других подобных вещей. В этот набор входят три ограничения, приведенные нами ранее — отсутствие вложенных задач, бесконечное

исполнение задач, ограничения на максимальный размер очереди входа в один элемент (т. е. только одна задача может ожидать на данном входе).

Оригинальная версия профиля Ravenscar предполагала исполнение программы на однопроцессорной машине. В Аде 2012 в профиль добавили семантику исполнения на многопроцессорной машине при условии, что задачи жестко закреплены за ЦПУ. Мы еще вернемся к этому вопросу.

Совместный эффект всех ограничений состоит в том, что становится возможным сформулировать утверждения о способности данной программы удовлетворять жестким требованиям в целях ее сертификации.

Никакой другой язык не предлагает таких средств обеспечения надежности, какие дает язык Ада с включенным профилем Ravenscar.

36.6 Безопасное завершение

В некоторых приложениях (например СУПР) задачи исполняются бесконечно, в других задачи доходят до своего конца и завершаются. В этих ситуациях вопросов, касающихся завершения задачи, не встает. Но бывают случаи, когда задача, предназначенная для бесконечной работы, должна быть завершена, например, по причине некоторой ошибки оборудования задача больше не нужна. Встает вопрос, как немедленно и безопасно завершить задачу. Увы, эти требования противоречат друг другу, и разработчику нужно искать компромисс. К счастью, язык Ада предлагает достаточно гибкие средства, чтобы разработчик смог реализовать нужный компромисс. Но даже если предпочтение отдается быстроте завершения, семантика языка гарантирует, что критические операции будут выполнены до того, как станет возможно завершить задачу.

Важным понятием в этом подходе является концепция региона отложенного прекращения. Это такой участок кода, исполнение которого должно дойти до конца, иначе существует риск разрушения разделяемых структур данных. Примерами могут служить тела защищенных операций и операторов принятия. Заметим, что при исполнении такого региона задача может быть вытеснена задачей с более высоким приоритетом.

Чтобы проиллюстрировать эти понятия, рассмотрим следующую версию задачи сервера:

```
task Server is
   entry Some_Service(Formal: in out Data);
end;

task body Server is
begin
   loop
      accept Some_Service(Format: in out Data) do
         ... -- код изменения значения Format
      end Some_Service;
   end loop;
end Server;
```

Задача будет исполнять периодически одну и ту же работу. Когда в ней больше не будет потребности, задача зависнет на инструкции принятия. Это напоминает анабиоз без перспективы пробуждения. Не очень приятная мысль и не очень приятный стиль программирования. Программа просто виснет, вместо того, чтобы изящно завершиться.

Есть несколько способов разрешить этот вопрос. Первый — объявить клиентскую задачу специально для того, чтобы выключить сервер, когда клиентских запросов больше не будет. Вот так можно описать задачу-палача:

```
task Grim_Reaper;

task body Grim_Reaper is
begin
   abort Server;
end Grim_Reaper;
```

Предназначение инструкции abort в том, чтобы прекратить указанную задачу (в нашем случае Server). Но это может быть рискованно - прекратить задачу немедленно, вне зависимости от того, что она делает в данный момент, будто вытащить вилку из розетки. Возможно, Server находится в процессе исполнения инструкции принятия входа Some_Service, и параметр может быть в несогласованном состоянии. Прекращение задачи привело бы к тому, что вызывающая задача получила бы искаженные данные, например, частично обработанный массив. Но, как сказано выше, инструкция принятия имеет «отложенное прекращение». Если будет попытка прекратить задачу в этот момент, то библиотека времени исполнения заметит это (формально, задача перейдет в аварийное состояние), но задача не будет завершена до тех пор, пока длится регион отложенного прекращения, т. е. в нашем случае до конца исполнения инструкции accept.

Даже если завершаемая задача не находится в регионе отложенного прекращения, эффект не обязательно будет мгновенным. Говоря коротко, завершаемая задача перейдет в аварийное состояние, в котором она рассматривается, как своего рода прокаженный. Если какая-либо задача неблагоразумно попытается взаимодействовать с этим несчастным (например, обратившись к одному из входов задачи), то получит исключение Tasking_Error. Наконец, если/когда прерванная задача достигнет любой точки планирования, такой как, вызов входа или инструкция принятия, то ее страданию прийдет конец и она завершится. (Для реализаций, поддерживающих Приложение Систем Реального Времени, требования к прекращению более жесткие: грубо говоря, аварийная задача завершится, как только окажется вне региона отложенного прекращения.)

Это может выглядеть немного гнетуще и сложно, и действительно, использование инструкции прекращения затрудняет написание программы, а применение формальных методов затрудняет еще больше. Популярный совет тут - «не делайте так». Можно использовать прекращение, когда нужно, например, сменить режим работы и завершить целое множество задач. В противном случае, лучше использовать одну из следующих техник, когда завершаемая задача сама решает, когда она готова уйти, т. е. завершение возможно только в особых точках.

Следующая техника использует специальный вход для передачи запроса на завершение. Приняв такой вызов, задача затем завершается обычным способом. Вот иллюстрация этой техники:

```
task Server is
   entry Some_Service(Formal: in out Data);
   entry Shutdown;
end;

task body Server is
begin
   loop
      select
         accept Shutdown;
         exit;
      or
         accept Some_Service(Format: in out Data) do
            ... -- код изменения значения Format
         end Some_Service;
      end select;
   end loop;
end Server;
```

В этой версии применяется форма инструкции select, охватывающая несколько альтернатив, каждая из которых начинается инструкцией принятия входа. При исполнении такой инструкции сначала проверяется, есть ли вызовы, ожидающие приема на этих входах. Если нет, то задача приостанавливается до тех пор, пока не появится такой вызов (в этом случае управление передается на соответствующую ветку). Если такой вызов один, управление передается на нужную ветку. Если несколько, то выбор ветки зависит от политики очередей входов (если реализация поддерживает Приложение Систем Реального Времени, политика, по умолчанию, основывается на приоритетах).

Такая форма инструкции select широко применяется в серверных задачах, когда задача имеет несколько входов, и порядок их вызова (либо количество) заранее не известен. В данном примере порядок вызова входов известен, сначала вызывается Some_Service, а затем Shutdown. Но мы не знаем, сколько раз вызовется Some_Service, поэтому нам понадобился бесконечный цикл.

Как и в предыдущем примере, нам нужна отдельная задача для завершения сервера. Но в этом случае, вместо инструкции прерывания задачи будет вызываться вход Shutdown:

```
task Grim_Reaper;

task body Grim_Reaper is
begin
   Server.Shutdown;
end Grim_Reaper;
```

При условии, что Grim_Reaper написан корректно, т. е. вызывает Shutdown после всех возможных вызовов Some_Service, подход с использованием Shutdown отвечает нашим требованиям к безопасной остановке задачи. Цена, которую мы платим за это — дополнительная задержка, поскольку завершение не происходит мгновенно.

Ада предлагает еще один подход к завершению, в котором нет необходимости какой-то из задач запускать процесс остановки. Идея состоит в том, что задача завершится автоматически (при содействии библиотеки времени исполнения), когда появится гарантия, что это можно сделать безопасно. Такую гарантию можно дать, если задача, имеющая один или несколько входов, висит на инструкции select и ни одна из ветвей этой инструкции не может быть вызвана. Чтобы это выразить, существует специальный вариант ветви инструкции select:

```
task Server is
   entry Some_Service(Formal: in out Data);
   entry Shutdown;
end;

task body Server is
begin
   loop
      select
         terminate;
      or
         accept Some_Service(Format: in out Data) do
            ... -- код изменения значения Format
         end Some_Service;
      end select;
   end loop;
end Server;
```

В этом случае, когда Server дойдет до инструкции select (или будет ожидать на ней), система времени исполнения посмотрит вокруг и определит, есть ли хоть одна живая задача, имеющая ссылку на Server (и таким образом имеющая возможность вызвать его вход). Если таких задач нет, то Server завершится безопасно и это произойдет автоматически. В этом случае нет возможности исполнить какой-либо код для очистки структур данных перед завершением. Но в стандарте Ада 2005 была добавлена возможность определить

обработчик завершения, который будет вызван в процессе завершения задачи.

К преимуществам этого подхода относится легкость понимания программы и надежность. Нет риска, как в других вариантах, забыть остановить задачу или остановить ее слишком рано. Недостаток - в существовании накладных расходов, которые несет библиотека времени исполнения при выполнении некоторых операций, даже если эта возможность не используется.

Подведем итог: Ада предоставляет различные возможности и поддерживает различные подходы к завершению задач. Инструкция прерывания имеет наименьшую задержку (хотя и ожидает выхода из регионов с отложенным прерыванием), но могут возникнуть проблемы, когда задача не находится в подходящем для завершения состоянии. Вход для запроса завершения решает эту проблему (задача завершается только когда примет запрос), но увеличивает задержку завершения. Наконец, подход со специальной альтернативой завершения наиболее безопасен (поскольку избавляет от ручного управления завершением задачи), но вводит дополнительные накладные расходы.

Среди возможностей, которые Ада избегает сознательно, возможность асинхронно возбудить исключение в другой задаче. Подобная возможность была, например, в изначальном варианте языка Java. Метод Thread.stop() (теперь считающийся устаревшим) позволяет легко разрушить разделяемые данные, оставив их в несогласованном состоянии. Исключения в Аде всегда выполняются синхронно и не имеют этой проблемы. Конечно, нужно аккуратно подходить к использованию исключений. Например, программист должен знать, что если он не обрабатывает исключения, то задача просто завершится при возникновении исключения, а исключение пропадет. Зато сложностей с асинхронными исключениями удалось избежать.

36.7 Время и планирование

Наверное, нельзя закончить главу о многозадачности в Аде, не остановившись на времени и планировании.

Есть инструкции для синхронизации исполнения программы с часами. Мы можем приостановить программу на некоторый промежуток времени (так называемая относительная задержка исполнения), либо до наступления нужного момента времени:

```
delay 2*Minutes;

delay until Next_Time;
```

предположим, что существуют объявления для Minutes и Next_Time. Небольшие относительные задержки могут быть полезны для интерактивного использования, в то время как задержка до наступления момента может быть использована для программирования периодических событий. Время можно измерять часами реального времени (они гарантируют некоторую точность), либо локальными часами, подверженными таким фактором, как переход на летнее время. В Аде также учитываются временные зоны и високосные секунды.

В стандарт Ада 2005 были добавлены несколько таймеров, при срабатывании которых вызывается защищенная процедура (обработчик). Есть три типа таймеров. Первый измеряет время ЦПУ, использованное конкретной задачей. Другой измеряет общий бюджет группы задач. Третий основан на часах реального времени. Установка обработчика выполняется по принципу процедуры Set_Handler.

Проиллюстрируем это на забавном примере варки яиц. Мы объявим защищенные объект Egg:

```
protected Egg is
   procedure Boil(For_Time: in Time_Span);
```

(continues on next page)

(continued from previous page)

```
private
   procedure Is_Done(Event: in out Timing_Event);
   Egg_Time: Timing_Event;
end Egg;

protected body Egg is

   procedure Boil(For_Time: in Time_Span) is
   begin
      Put_Egg_In_Water;
      Set_Handler(Egg_Done, For_Time, Is_Done'Access);
   end Boil;

   procedure Is_Done(Event: in out Timing_Event) is
   begin
      Ring_The_Pinger;
   end Is_Done;

end Egg;
```

Пользователь напишет так:

```
Egg.Boil(Minutes (10)); -- лучше сварить вкрутую

-- читаем пока яйцо варится
```

и будильник зазвенит, когда яйцо будет готово.

Планирование задается директивой компилятору Task_Dispatching_Policy(политика). В Аде 95 определена политика FIFO_Within_Priorities, а в Аде 2005, в Приложении Систем Реального Времени — еще несколько. С помощью этой директивы можно назначить политику всем задачам или задачам, имеющим приоритет из заданного диапазона. Перечислим существующие политики:

- FIFO_Within_Priorities — В пределах каждого уровня приоритета с этой политикой задачи исполняются по принципу первый-пришел-первый-вышел. Задачи с более высоким приоритетом могут вытеснять задачи с меньшим приоритетом.
- Non_Preemtive_FIFO_Within_Priotities — В пределах каждого уровня приоритета с этой политикой задачи исполняются, пока не окончат выполнение, будут заблокированы или выполнят инструкцию задержки (delay). Задача с более высоким приоритетом не может вытеснить их. Эта политика широко используется в приложениях с повышенными требованиями к безопасности.
- Round_Robin_Within_Priotities — В пределах каждого уровня приоритета с этой политикой задачам выделяются кванты времени заданной продолжительности. Эта традиционная политика применяется с первых дней появления параллельных программ.
- EDF_Across_Priotities — EDF это сокращение Earliest Deadline First. Общая идея следующая. На заданном диапазоне уровней приоритета каждая задача имеет крайний срок исполнения (deadline). Исполняется та, у которой это значение меньше. Это новая политика. Для ее использования разработан специальный математический аппарат.

Ада позволяет устанавливать и динамически изменять приоритеты задач и так называемые граничные приоритеты защищенных объектов. Это позволяет избежать проблемы инверсии приоритетов, как описано в [9].

Стандарт Ада 2012 ввел множество полезных усовершенствований, касающихся времени и планирования. Большинство из них не касаются темы этого буклета, но мы затронем здесь один из вопросов, теперь отраженный в стандарте — поддержка

многопроцессорных/многоядерных платформ. Эта проблема включает следующие возможности:

- Пакет System.Multiprocessors, где определена функция, возвращающая количество ЦПУ.
- Новый аспект, позволяющий назначить задачу данному ЦПУ.
- Дочерний пакет System.Multiprocessors.Dispatching_Domains позволяет выделить диапазон процессоров, как отдельный «домен диспетчеризации», а затем назначать задачи на исполнение этим доменом либо с помощью аспектов, либо при помощи вызова подпрограммы. После этого задача будет исполняться любым ЦПУ из заданного диапазона.
- Определение директивы компилятору Volatile поменяли. Теперь она гарантирует корректный порядок операций чтения и записи вместо требования указанной переменной находиться в памяти, как было раньше.

CHAPTER

THIRTYSEVEN

СЕРТИФИКАЦИЯ С ПОМОЩЬЮ SPARK

В областях с повышенными требованиями к безопасности и надежности приложение обязанно быть корректным. Эта корректность обеспечивается специальными формальными процедурами. Когда дело касается безопасности, ошибка в приложении может стоить человеческой жизни или катастрофы (или просто быть очень дорогой, как, например, ошибки в управлении космическими аппаратами и ракетами). Ошибка в особо надежных приложениях может привести к компрометации национальной безопасности, утрате коммерческой репутации или просто к краже.

Приложения можно классифицировать согласно последствий от сбоя программного обеспечения. Стандарты авиационной безопасности DO-178B [1] и DO-178C [2] предлагают следующую классификацию:

- Уровень E никакой: нет проблем. Пример — отказала развлекательная система? Это даже прикольно!
- Уровень D низкий: некоторое неудобство. Пример — отказала система автоматического туалета.
- Уровень C высокий: некоторые повреждения. Пример — жесткая посадка, порезы и ушибы.
- Уровень B опасный: есть жертвы. Пример — неудачная посадка с пожаром.
- Уровень A катастрофический: авиакатастрофа, все мертвы. Пример — поломка системы управления.

Следует отметить, хотя отказ системы развлечений и относится к уровню E, но, если пилот не может выключить ее (например, чтобы сделать важное объявление), то эта ошибка повышает уровень системы развлечений до D.

Для наиболее критичных приложений, где требуется сертификации в соответствующих органах, недостаточно иметь корректную программу. Необходимо также доказать, что программа корректна, что намного тяжелее. Это требует использования формальных математических методов. Эта и стало причиной возникновения языка SPARK.

Эта глава является кратким введением в SPARK 2014. Это наиболее свежая на данный момент версия языка. В ней используется синтаксис Ада 2012 для указания контрактов, таких как пред- и пост-условия. Таким образом, SPARK 2014 - это подмножество языка Ада 2012 с некоторыми дополнительными возможностями, облегчающими формальный анализ (эти возможности используют механизм аспектов, рассмотренный в главе «Безопасные типы данных»). Программа на Аде 2012 может иметь компоненты, написанные с использованием всех возможностей Ады, и другие компоненты на языке SPARK (которые будут иметь явную отметку об этом). Компоненты на SPARK могут быть скомпилированы стандартным Ада компилятором и будут иметь стандартную Ада семантику во время исполнения, но они лучше поддаются методам формального анализа, чем остальные компоненты.

Компоненты SPARK кроме компиляции стандартным Ада компилятором еще и анализируются с помощью инструментария SPARK. Этот инструментарий может статически удостовериться в исполнении контрактов (таких, как пред-условия и пост-условия), которые обычно проверяются во время исполнения. Как следствие, эти проверки времени

исполнения могут быть выключены. Инструментарий SPARK, используемый в GNAT, называется GNATprove. Далее в этой главе мы предполагаем использование для анализа именно этого инструмента.

Важно отметить, что SPARK можно использовать на разных уровнях. На самом простейшем Ада программа, удовлетворяющая подмножеству SPARK, может быть проанализирована без дополнительных усилий. Но мы можем укрепить описание программы, добавив различные аспекты относительно потока информации, что позволит инструментарию провести более скрупулезный анализ программы. В конце концов, пользователь сам выбирает, добавлять или нет данные аспекты, в зависимости от характера проекта, в том числе от требований к необходимому уровню безопасности ПО и политики верификации.

Мы начнем с более подробного рассмотрения основных концепций корректности и контрактов.

37.1 Контракты

Что мы подразумеваем под корректным ПО? Наверное, общим определением может быть такое — ПО, которое делает то, что предполагал его автор. Для простой одноразовой программы это может означать просто результат вычислений, в то время как для большого авиационного приложения это будет определиться текстом контракта между программистом и конечным пользователем.

Идея контрактов в области ПО далеко не нова. Если мы посмотрим на библиотеки, разработанные в начале 1960-х, в частности, в математической области, написанные на Algol 60 (этот язык пользовался популярностью при публикации подобных материалов в уважаемых журналах, типа Communications of the ACM и Computer Journal), мы увидим подробные требования к параметрам, ограничения на диапазон их значений и прочее. Суть здесь в том, что вводится контракт между автором подпрограммы и ее пользователем. Пользователь обязуется предоставить подходящие параметры, а подпрограмма обязуется вернуть корректный результат.

Деление программы на различные части - это хорошо известный подход, а сутью процесса программирования является определение этих составных частей и, таким образом, интерфейсов между ними. Это позволяет разрабатывать отдельные части независимо друг от друга. Если мы напишем каждую часть правильно (т. е. они будут выполнять каждая свою часть контракта, определяемого интерфейсом) и если мы правильно определили интерфейсы, то мы можем быть уверены, что, собрав все части вместе, получим функционирующую правильно систему.

Горький опыт говорит нам, что в жизни все не совсем так. Две вещи могут пойти не так: с одной стороны, определения интерфейсов зачастую не являются доскональными (есть дыры в контрактах), с другой стороны, индивидуальные компоненты работают неправильно или используются неправильно (контракты нарушаются). И, конечно, контракты могут диктовать совсем не то, что мы имели ввиду.

Любопытно, что есть несколько способов указать контракты на программные компоненты. В первую очередь, и исторически SPARK настоятельно рекомендует делать так, можно указывать контракты с самого начала. Обычно это называют «корректность с помощью построения», а также «декларативная верификация», когда каждый программный модуль содержит контракт в его спецификации. Контракт можно расценивать, как явно выраженные низкоуровневые требования к программному модулю. Если контракт противоречит коду модуля, то это будет обнаружено до начала исполнения программы. Годы использования SPARK в таких областях, как авиация, банковская сфера, управление железной дорогой, позволяют убедиться, что вероятность получить корректную программу повышается и, более того, общая стоимость разработки, включая фазы тестирования и интеграции, уменьшается.

Хотя такой подход эффективен для новых проектов, его применение может быть

затруднительно в случаях доработки существующего ПО. В связи с этим, SPARK поддерживает другой стиль разработки, называемый «порождающая верификация». Если код не содержит контрактов, GNATprove может синтезировать некоторые из них на основе тела модуля. Таким образом, проект может двигаться от порождающей верификации к декларативной по мере развития системы. Более того, как мы объясним далее, SPARK 2014 позволяет разработчикам комбинировать формальные методы с традиционными, основанными на тестировании.

Давайте теперь рассмотрим более подробно два вопроса, касающихся, во первых, исчерпывающего определения интерфейса, во вторых, проверки кода реализации на соответствие интерфейсу. Проще подойти к этому в терминах декларативной верификации, хотя эти же концепции также применимы к методу порождающей верификации.

В идеале, определение интерфейса должно скрывать все несущественные детали, но освещать все существенные. Другими словами, мы можем сказать, что определение интерфейса должно быть одновременно корректным и полным.

В качестве простого интерфейса рассмотрим интерфейс подпрограммы. Как мы уже упоминали, интерфейс должен описывать целиком контракт между пользователем и автором. Детали реализации нам не интересны. Чтобы различать эти два аспекта, полезно использовать два различных языка программирования. Некоторые языки представляют подпрограмму как некий неделимый кусок кода, в котором реализация не отделима от интерфейса. В этом кроется проблема. Это не только затрудняет проверку интерфейса, поскольку компилятор сразу требует код вместе с реализацией, но и поощряет программиста писать код одновременно с формулированием интерфейса, что вносит путаницу в процесс разработки.

Структура программы на Аде разделяет интерфейс (спецификацию) от реализации. Это верно как для отдельных подпрограмм, так и для их групп, объединенных в пакеты. Это главная причина того, почему Ада так хорошо подходит как основа для SPARK.

Как упоминалось ранее, иногда очень удобно, когда только часть программы написана на SPARK, а другие части на Аде. Части SPARK отмечаются с помощью аспекта SPARK_Mode. Он может указываться для отдельных подпрограмм, но удобнее указывать его для всего пакета. Например:

```
package P
   with SPARK_Mode is
   ...
```

В добавок к этому, этот режим можно включить при помощи директивы компилятору. Это удобно, если необходимо указать режим для всей программы (используя файл конфигурации gnat.adc):

```
pragma SPARK_Mode;
```

Дополнительную информацию об интерфейсе в SPARK можно передать при помощи механизма аспектов Ада 2012. Главная цель использования этих аспектов — увеличить количество информации об интерфейсе без предоставления лишних деталей о реализации. На самом деле, SPARK позволяет использовать информацию разного уровня детализации в зависимости от потребностей приложения.

Рассмотрим информацию предоставляемую следующей спецификацией:

```
procedure Add(X: in Integer);
```

Откровенно говоря, здесь ее совсем мало. Извесно только то, что процедура Add принимает единственный параметр типа Integer. Этого достаточно, чтобы компилятор имел возможность создать код для вызова процедуры. Но при этом совершенно ничего не известно о том, что процедура делает. Она может делать все, что угодно. Она вообще может не использовать значение X. Она может, к примеру, находить разность двух глобальных переменных и печатать результат в некоторый файл. Но теперь рассмотрим, что случиться,

если если мы добавим SPARK-конструкцию, определяющую, как используются глобальные переменные. Мы предполагаем, что процедура определена в пакете, для которого включен режим SPARK_Mode. Спецификация могла бы быть такой:

```
procedure Add(X: in Integer)
   with Global => (In_Out => Total);
```

Аспект Global указывает, что единственной переменной, доступной в этой процедуре, является Total. Кроме того, идентификатор In_Out говорит, что мы будем использовать начальное значение Total и вычислим ее новое значение. В SPARK есть дополнительные правила для параметров подпрограмм. Хотя в Аде мы вправе не использовать параметр X вообще, в SPARK параметры с режимом in должны быть использованы в теле подпрограммы. В противном случае мы получим предупреждение.

Теперь мы знаем намного больше. Вызов Add вычислит новое значение Total, использовав для этого начальное значение Total и значение X. Также известно, то Add не может изменить что-либо еще. Определенно, она не может ничего печатать или иметь другие побочные эффекты. (Анализатор обнаружит и отвергнет программу, если эти условия нарушаются.)

Безусловно, такой контракт нельзя считать завершенным, поскольку из него не следует, что используется операция сложения, чтобы получить новое значение Total. Указать это можно с помощью пост-условия:

```
procedure Add (X: in Integer)
   with Global => (In_Out => Total),
      Post   => (Total = Total'Old + X);
```

Пост-условие однозначно определяет, что новое значение Total равно сумме начального значения (обозначенного как 'Old) и значения X. Теперь спецификация завершена.

Также есть возможность указать пред-условие. Мы можем потребовать, чтобы X было положительным и при сложении не возникло переполнение. Это можно сделать следующим образом:

```
Pre => X > 0 and then Total <= Integer'Last - X
```

(Ограничение на положительные значения X лучше было бы выразить, использовав тип Positive в объявлении параметра X. Мы включили это в предусловие лишь в демонстрационных целях.)

Пред- и пост-условия, как и все аспекты SPARK, не являются обязательными. Если они не указаны явно, предполагается значение True.

Еще один аспект, который можно указать, это Depends. Он определяет зависимости между начальными значениями параметров и переменных и их конечными значениями. В случае с Add это выглядит так:

```
Depends => (Total => (Total, X))
```

Здесь просто сказано, что конечное значение Total зависит от его начального значения и значения X. Однако, в этом случае, это и так можно узнать из режима параметра и глобальной переменной, поэтому это не дает нам новой информации.

Как мы уже говорили, все SPARK-аспекты являются необязательными. Но, если они указаны, то будут статически верифицированы при анализе тела подпрограммы.

В случае с пред- и пост-условием будет сделана попытка доказать следующее:

Если предусловие истинно, то (1) не произойдет ошибок времени исполнения и (2) если подпрограмма завершится, то постусловие будет выполнено.

Если контракт не будет верифицирован (например, GNATProve не сможет доказать истинность пред- и пост-условий), то анализатор отвергнет модуль, но компилятор Ада все еще может скомпилировать его, превратив пред- и пост-условия в проверки времени

исполнения (если Assertion_Policy равна Check). Таким образом, использование общего синтаксиса для Ада 2012 и SPARK 2014 предоставляет значительную гибкость. Например, разработчик сначала может применять проверки во время исполнения, а затем, когда код и контракт будут отлажены, перейти к статической верификации.

Для критических систем данная статическая верификация очень важна. В этой области нарушение контракта во время исполнения недопустимо. Узнать, что условие не выполнено только в момент исполнения, это не то, что нам нужно. Допустим, у нас есть пред-условие для посадки самолета:

```
procedure Touchdown(...)
   with Pre => Undercarriage_Down; -- шасси выпущено
```

Узнать о том, что шасси не выпущено только в момент посадки самолета будет слишком поздно. На самом деле, мы хотим быть уверены, что программа была проанализирована заранее и гарантируется, что такая ситуация не возникнет.

Это подводит нас к следующему вопросу гарантий того, что реализация корректно исполняет интерфейсный контракт. Иногда это называют отладкой. Вообще есть четыре способа обнаружить ошибку:

1. С помощью компилятора. Такие ошибки обычно исправить легко, потому что компилятор говорит нам, что не так. 1. Во время исполнения с помощью проверок, определенных языком. Например, язык гарантирует, что мы не обращаемся к элементу за пределами массива. Обычно мы получаем сообщение об ошибке и место в программе, где она произошла. 1. При помощи тестирования. В этом случае мы запускаем какие-то примеры и размышляем над неожиданными результатами, пытаясь понять, что пошло не так. 1. При крахе программы. Обычно после краха остается очень мало данных и поиск причины может быть очень утомительным.

Первый тип, на самом деле, должен быть расширен, чтобы обозначать «до начала исполнения программы». Таким образом, он включает сквозной контроль программы и другие рецензирующие методы, в том числе статический анализ инструментарием типа GNATprove.

Очевидно, что эти четыре способа указаны в порядке возрастания трудности. Ошибки тем легче локализовать и исправить, чем раньше они обнаружены. Хорошие средства программирования позволяют переместить ошибки из категории с большим номером в категорию с меньшим. Хороший язык программирования предоставляет средства, позволяющие оградить себя от ошибок, которые трудно найти. Язык Ада хорош благодаря строгой типизации и проверкам времени исполнения. Например, правильное использование перечислимых типов превращает сложные ошибки типа 3 в простые ошибки типа 1, как мы продемонстрировали в главе «Безопасные типы данных».

Главная цель языка SPARK состоит в том, чтобы за счет усиления определения интерфейса (контрактов) переместить все ошибки из других категорий, в идеале, в категорию 1, для того, чтобы обнаружить их до момента запуска приложения. Например, аспект Globals предотвращает случайное изменение глобальных переменных. Аналогично, обнаружение потенциальных нарушений пред- и пост-условий выливается в ошибки 1-го типа. Однако, проверка, что такие нарушения невозможны, требует математических доказательств. Это не всегда просто, но GNATprove способен автоматизировать большую часть процесса доказательства.

37.2 SPARK — подмножество языка Ада

Ада - это довольно сложный язык программирования и использование всех его возможностей затрудняет полный анализ программы. Соответственно, подмножество языка Ада, используемое в SPARK, ограничивает набор доступных средств. В основном, это касается поведения во время исполнения. Например, отсутствуют ссылочные типы (а следовательно и динамическое распределение памяти) и обработка исключений.

Многозадачность, в ее полном варианте, имеет очень сложную семантику исполнения. Но при использовании Ravenscar-профиля она все же подлежит формальному анализу. Ravenscar вошел в версию SPARK 2005 и его планируется добавить в следующую версию SPARK 2014 (к моменту перевода брошуры уже добавлен).

Вот список некоторых ограничений, вводимых SPARK 2014:

- Все выражения (в том числе вызовы функций) не производят побочных эффектов. Хотя функции в Ада 2012 могут иметь параметры с режимом in out и out, в SPARK это запрещено. Функции не могут изменять глобальные переменные. Эти ограничения помогают гарантировать, что компилятор волен выбирать любой порядок вычисления выражений и подчеркивают различие между процедурами, чья задача изменять состояние системы, и функциями, которые лишь анализируют это состояние.
- Совмещение имен (aliasing) запрещается. Например, нельзя передавать в процедуру глобальный объект при помощи out или in out параметр, если она обращается к нему напрямую. Это ограничение делает результат работы более прозрачным и позволяет убедиться, что компилятор волен выбрать любой способ передачи параметра (по значению или по ссылке).
- Инструкция goto запрещена. Это ограничение облегчает статический анализ.
- Использование контролируемых типов запрещено. Контролируемые типы приводят к неявным вызовам подпрограмм, генерируемых компилятором. Отсутствие исходного кода для этих конструкций затрудняет использование формальных методов.

В дополнение к этим ограничениям, SPARK предписывает более строгую политику инициализации, чем Ада. Объект, передаваемый через in или in out параметр, должен быть полностью инициализирован до вызова процедуры, а out параметр — до возвращения из нее.

Несмотря на эти ограничения, подмножество языка, поддерживаемое SPARK, все еще довольно велико. Оно включает типы с дискриминантами (но без ссылочных дискриминантов), теговые типы и диспетчеризацию, типы с динамическими границами, отрезки массивов и конкатенацию, статические предикаты, условные и кванторные выражения, функции-выражения, настраиваемые модули, дочерние модули и субмодули. Рекурсия допускается, а стандартная библиотека контейнеров может быть использована для создания сложных структур данных в условиях отсутствия ссылочных типов и динамического распределения памяти. SPARK также предлагает средства взаимодействия с внешним окружением при помощи аспектов, касающихся «изменчивых» (volatile) переменных (т. е. переменных, подверженных асинхронному чтению и записи).

В критических областях, где использование таких средств, как динамическое распределение памяти нежелательно, ранние версии языка SPARK доказали свою исключительную полезность. Теперь же SPARK 2014 существенно расширил набор разрешенных возможностей языка.

37.3 Формальные методы анализа

В этой главе мы мы коротко подытожим некоторые аспекты механизма формального анализа, используемого инструментарием SPARK.

Во первых, есть две формы потокового анализа:

- Потоковый анализ зависимостей данных учитывает инициализацию переменных и зависимости между данными внутри подпрограмм.
- Потоковый анализ потоковых зависимостей учитывает взаимосвязи результатов подпрограмм и их входных данных.

Потоковый анализ не требует от пользователя больших усилий, чем указания аспектов глобальных переменных и аспектов зависимостей, и, разумеется, учета правил языка SPARK. Добавления этих аспектов позволяет провести более подробный потоковый анализ и обнаружить больше ошибок на ранней стадии.

Важной возможностью потокового анализа является обнаружение неинициализированных переменных. Чтобы воспользоваться этим, необходимо избегать задавать «мусорные» начальные значения по умолчанию, просто «на всякий случай», как мы уже оговаривали это в главе «Безопасный запуск», поскольку это затруднит нахождение ошибок потоковым анализом.

Далее, существуют формальные процессы верификации, касающиеся доказательств:

- Формальная верификация свойств надежности (т. е. гарантия отсутствия возбуждения предопределенных исключений);
- Формальная верификация функциональных свойств, основанных на контрактах, таких как пред-условия и пост-условия.

В случае с функциональными свойствами, которые кроме пред- и пост-условий включают в себя инварианты циклов и утверждения, касающиеся типов, анализатор генерирует предположения, которые необходимо доказать для гарантирования корректности программы. Эти предположения известны, как условия верификации (verification conditions, VC). А доказательство их называют исполнением условий верификации. За последние годы произошел значительный прогресс в области автоматического доказательства теорем, благодаря чему GNATprove способен автоматически исполнять множество условий верификации. На момент написания этого текста используются технологии Alt-Ergo и CVC4, но можно использовать и другие системы доказательств теорем.

Важно отметить, что даже в отсутствии пред- и пост-условий анализатор способен генерировать предположения, соответствующие проверкам времени исполнения языка, таким как проверка диапазона. Как мы демонстрировали в главе «Безопасные типы данных», подобные проверки автоматически вставляются, чтобы гарантировать, что переменная не получит значений вне диапазона значений согласно ее объявлению. Аналогичные проверки контролируют попытки чтения/записи элементов за границами массива. Доказательство этих предположений демонстрирует, что условия не нарушаются и программа не содержит ошибок, приводящих к возбуждению исключений в момент исполнения.

Заметим, что использование доказательств не является обязательным. SPARK и соответствующий инструментарий можно использовать на разных уровнях. Для некоторых приложений достаточно использовать потоковый анализ. Но для других может быть экономически целесообразно также провести и доказательство корректности. На самом деле, различные уровни анализа могут быть использованы для различных частей программы. Этого можно добиться, используя различные варианты аспекта SPARK_Mode.

37.4 Гибридная верификация

Проведение формальной верификации для всего кода программы может быть нецелесообразно по следующим причинам:

- Часть программы может использовать все возможности Ады (или вообще может быть написана на другом языке, например C), и, следовательно, не поддается формальному анализу. Например, спецификация пакета может иметь аспект SPARK_Mode, а тело пакета — нет. В этом случае информация о контрактах из спецификации пакета может быть использована инструментарием формального анализа, хотя анализ самого тела пакета будет невозможен. Для не-SPARK модулей необходимо использовать традиционные методы тестирования.
- Даже если при написании компонента используется SPARK подмножество языка, не всегда возможно выразить требуемые свойства формально или доказать их с учетом текущих возможностей применяемого инструментария. Аналогично, применение тестирования необходимо для демонстрации заданных свойств.

Следующие инструменты для поддержки такого рода «гибридной верификации» входят в состав GNAT технологии.

- Порождающая верификация. GNATprove может быть использован для анализа Ада модулей вне зависимости от режима SPARK, чтобы определить неявные зависимости данных. Этот подход, который ранее мы назвали «порождающая верификация», позволяет применить формальные методы при для уже существующей кодовой базы.
- Результаты GNATprove. GNATprove может отобразить свойства подпрограммы, которые он не может установить, что означает, например, возможность появления ошибок во время исполнения. Это может помочь пользователю в написании тестов, либо указать, где необходим дальнейший анализ кода.
- Опции компилятора. Некоторые из ключей компилятора позволяют получить дополнительные проверки времени исполнения для таких ошибок, как пересекающиеся параметры или недействительные скалярные объекты, в модулях, для которых невозможен формальный анализ.
- Инструмент GNATtest. При помощи аспектов (или директив компилятору), специфичных для GNAT, можно определить для данной подпрограммы «формальные тестовые случаи». Это совокупность требуемых условий на входе в подпрограмму и ожидаемых условий при выходе из нее. Инструмент GNATtest использует эту информацию для автоматизации построения набора тестов.

37.5 Примеры

В качестве примера, рассмотрим версию стека с указанием потоковых зависимостей (аспект Depends) без использования пред- и пост-условий:

```
package Stacks
   with Spark_Mode
is

   type Stack is private;

   function Is_Empty(S: Stack) return Boolean;
   function Is_Full(S: Stack) return Boolean;

   procedure Clear(S: out Stack)
      with Depends => (S => null);
```

(continues on next page)

(continued from previous page)

```
   procedure Push(S: in out Stack; X: in Float)
      with Depends => (S => (S, X));

   procedure Pop(S: in out Stack; X: out Float)
      with Depends => (S => S,
                       X => S);

private
   Max: constant := 100;
   type Top_Range is range 0 .. Max;
   subtype Index_Range is Top_Range range 1 .. Max;

   type Vector is array (Index_Range) of Float;
   type Stack is record
      A: Vector;
      Top: Top_Range;
   end record;
end Stacks;
```

Мы добавили функции Is_Full и Is_Empty, которые просто считывают состояние стека. Они не имеют аспектов.

Для остальных подпрограмм мы добавили аспекты Depends. Они позволяют указать, от каких аргументов зависит данный результат. Например, для Push указано (S => (S, X)), что означает, что конечное значение S зависит от начального значения S и начального значения X, что, в данном примере, можно вывести из режимов параметров. Но избыточность — это один из ключей достижения надежности. Если аспекты и режимы параметров противоречат друг другу, то это будет обнаружено автоматически при анализе, что, возможно, позволит найти ошибку в спецификации.

Объявления в приватной части были изменены, чтобы дать имена всем используемым подтипам, хотя это и не является обязательным в SPARK 2014.

На этом уровне не нужно вносить каких-либо изменений в тело пакета, поскольку контракты не требуются. Это подчеркивает тот факт, что SPARK касается в основном улучшения качества описания интерфейса.

Отличия появились также в том, что мы не присвоили начальное значение компоненте Top, а взамен требуем явного вызова Clear. При анализе клиентского SPARK-кода с помощью GNATprove будет проведен потоковый анализ, гарантирующий, что нет вызова процедур Push или Pop до вызова процедуры Clear. Этот анализ выполняется без исполнения программы. Если GNATprove не может доказать это, то в программе возможна ошибка. С другой стороны, если будет обнаружен вызов Push или Pop перед Clear, то это означает, что ошибка присутствует наверняка.

В таком коротком обзоре, как этот, невозможно привести какой-либо сложный пример анализа. Но мы приведем тривиальный пример, чтобы продемонстрировать идею. Следующий код:

```
procedure Exchange(X,Y: in out Float)
   with Depends => (X => Y, Y => X),
      Post => (X = Y'Old and Y = X'Old);
```

демонстрирует спецификацию процедуры, предназначенной для обмена значений двух параметров. Тело может выглядеть так:

```
procedure Exchange(X,Y: in out Float) is
   T: Float;
begin
   T := X; X := Y; Y := T;
end Exchange;
```

При анализе GNATprove создает условия верификации, выполнение которых необходимо доказать. В данном примере доказательство тривиально и выполняется автоматически. (Читатель может заметить, что доказательство сводится к проверке условия (x=x and y=y), которое, очевидно, истинно). В более сложных ситуациях GNATprove может не справиться с доказательством, тогда пользователь может предложить промежуточные утверждения, либо воспользоваться другим инструментарием для дальнейшего анализа.

37.6 Сертификация

Как было продемонстрировано в предыдущих главах, Ада - это превосходный язык для написания надежного ПО. Ада позволяет программисту обнаруживать ошибки на ранних стадиях процесса разработки. Еще больше ошибок можно выявить, используя SPARK даже без применения процедуры тестирования. Тестирование остается обязательным этапом разработки, несмотря на то, что это сложный и чреватый ошибками процесс.

В областях, где действуют наивысшие требования к безопасности и надежности, недостаточно иметь правильно работающую программу. Необходимо еще доказать, что она является таковой. Этот процесс доказательства называется сертификацией и выполняется согласно процедурам соответствующего органа сертификации. Примерами таких органов на территории США являются FAA в области безопасности и NSA в области надежности. SPARK обладает огромной ценностью в процессе разработки программ, подлежащих сертификации.

Может возникнуть впечатление, что использование SPARK увеличивает стоимость разработки. Однако, исследования систем, переданных в NSA, демонстрируют, что использование SPARK делает процесс разработки более дешевым по сравнению с обычными методами разработки. Несмотря на то, что необходимо потратить некоторые усилия на формулирование контрактов, это, в итоге, окупается за счет сокращения затрат на тестирование и исправление ошибок.

37.7 Дальнейший процесс

На момент написания этого текста две значительные разработки еще не завершены. Первая касается поддержки профиля Ravenscar в SPARK многозадачности. Вторая — это возможность указать уровни целостности различных компонент программы, которая позволит гарантировать, что поток информации удовлетворяет требованиям указанных уровней безопасности и надежности.

Узнать текущее состояние дел и получить всестороннюю документацию SPARK 2014 можно на сайте www.spark-2014.org/.

CHAPTER

THIRTYEIGHT

ЗАКЛЮЧЕНИЕ

Хочется надеяться, что эта брошюра была Вам интересна. Мы коснулись различных аспектов написания надежного ПО и надеемся, что смогли продемонстрировать особое положение языка Ада в этом вопросе. Напоследок мы коснемся некоторых моментов относительно разработки языков программирования.

Баланс между программным обеспечением и аппаратной частью сам по себе вызывает интерес. Аппаратные средства за последние пол-столетия развивались ошеломляющим темпом. Они сегодня не имеют ничего общего с тем, что было в 1960-х гг. По сравнению с аппаратными средствами, ПО тоже развивалось, но совсем чуть-чуть. Большинство языков программирования сейчас имеют лишь незначительные отличия от тех языков, что были в 1960-х. Я подозреваю, что основная причина этого заключается в том, что мы очень мало знаем о ПО, хотя считаем, что знаем о нем очень много. Человечество, вложив колоссальные средства в плохо написанное ПО, вдруг обнаружило, что никуда не может двинуться с ним дальше. В то же время новая аппаратура создается с такой скоростью, что это неизбежно приводит к ее замене. Несомненно, что любой может легко научиться, как написать небольшую программку, но чтобы создать хоть какую-нибудь аппаратуру требуется овладеть огромным объемом знаний.

Существуют две основные ветви развития языков. Одна берет начало от Algol 60 и CPL. Производные от этих языков часто упоминаются в этом буклете. Другая ветвь, охватывающая Fortran, COBOL и PL/I, также еще жива, хотя и весьма изолирована.

Algol 60 можно считать самым значительным шагом в развитии. (Когда-то существовал также его менее знаменитый предшественник Algol 58, от которого произошел язык Jovial, использовавшийся в военных сферах США.) Algol дал ощущение того, что написание ПО - это больше чем просто кодинг.

При создании Algol было сделано два больших шага вперед. Во первых, появилось понимание того, что присваивание не есть равенство. Это привело к появлению обозначения := для операции присваивания. Для обозначения различных управляющих структур стали использовать английские слова, в результате отпала необходимость в многочисленных инструкциях goto и метках, которые так затрудняли чтение программы на Fortran и autocode. На втором моменте стоит остановиться более подробно.

Сначала рассмотрим следующие две инструкции в Algol 60:

```
if X > 0 then
   Action(...);
Otherstuff(...);
```

Суть в том, что если X больше нуля, то вызывается процедура Action. Независимо от этого, мы далее вызываем Otherstuff. Т.е. действие условия здесь распространяется только на первую инструкцию после then. Если нам необходимо несколько инструкций, например, вызвать две процедуры This и That, то мы должны объединить их в составную инструкцию следующим образом:

```
if X > 0 then
begin
```

(continues on next page)

(continued from previous page)

```
   This(...);
   That(...);
end;
Otherstuff(...);
```

Здесь возникает опасность сделать две ошибки. Во первых, мы можем забыть добавить begin и end. Результат выйдет без ошибок, но процедура That будет вызываться в любом случае. Будет еще хуже, если мы нечаянно добавим лишнюю точку с запятой. Наверное, Algol 60 был первым языком, где использовалась точка с запятой как разделитель инструкций. Итак, мы получим:

```
if X > 0 then ;
begin
   This(...);
   That(...);
end;
Otherstuff(...);
```

К несчастью, в Algol 60 эта запись означает неявную пустую инструкцию после then. В результате, условие не будет влиять на вызовы подпрограмм This и That. Аналогичные проблемы в Algol 60 есть и для циклов.

Разработчики Algol 68 осознали эту проблему и ввели скобочную запись условной инструкции, т. е.:

```
if X > 0 then
   This(...);
   That(...);
fi;
Otherstuff(...);
```

Аналогичная запись появилась для циклов, где слову do соответствует od, а для case есть esac. Это полностью решает проблему. Теперь совершенно очевидно, что условная инструкция захватит оба вызова подпрограмм. Появление лишней точки с запятой после then приведет к синтаксической ошибке, которая легко будет обнаружена компилятором. Конечно, появление такой записи, как fi, od и esac выглядит причудливо, что может помешать серьезно отнестись к решаемой проблеме.

По какой-то причине разработчики языка Pascal проигнорировали этот здравый подход и оставили уязвимую запись инструкций из Algol 60. Они исправили свою ошибку гораздо позже, уже при проектировании Modula 2. К тому моменту Ада уже давно существовала.

Вероятно, в языке Ада впервые появилась удачная форма структурных инструкций, не использовавшая причудливую запись. На языке Ада мы бы написали:

```
if X > 0 then
   This(...);
   That(...);
end if;
Otherstuff(...);
```

Позже многие языки переняли такую безопасную форму записи. К таковым относится даже макро-язык для Microsoft Word for DOS и Visual Basic в составе Word for Windows.

Еще одним значимым языком можно считать CPL. Он был разработан в 1962г. и использовался в двух новых вычислительных машинах в университетах Кембриджа и Лондона.

CPL, (как и Algol 60) использовал := для обозначения присваивания и = для сравнения. Вот небольшой фрагмент кода на CPL:

```
§ let t, s, n = 1, 0, 1
  let x be real
  Read[x]
   t, s, n := tx/n, s + t, n + 1
   repeat until t<<1
  Write[s] §
```

Интересная особенность CPL заключается в использовании = (вместо :=) при задании начальных значений, ввиду того, что при этом не происходит изменения значений. В CPL был ряд интересных решений, например, параллельное присваивание и обработка списков. Но CPL остался лишь академической игрушкой и не был реализован.

Для группировки инструкций язык CPL использовал запись, аналогичную принятой в Algol 60. Например

```
if X > 0 then do
   § This(...)
     That(...) §
Otherstuff(...);
```

Для этого использовались странные символы параграфа и параграфа перечеркнутого вертикальной чертой.

Хотя сам язык CPL никогда не был реализован, в Кембридже изобрели его упрощенный вариант BCPL (Basic CPL). В отличии от имеющего строгую типизацию CPL, BCPL вообще не имел типов, а массивы были реализованы с помощью адресной арифметики. Так BCPL положил начало проблеме переполнения буфера, от которой мы страдаем по сей день.

BCPL преобразился в B, а затем в C, C++ и т. д. В BCPL использовалось обозначение := для операции присваивания, но по пути кто-то забыл, в чем смысл, и C остался с обозначением =. Поскольку знак = оказался занят, для операции сравнения пришлось ввести обозначение ==, а вместе с этим получить букет проблем, о котором мы писали в главе «Безопасный синтаксис».

Язык C унаследовал принцип группировки инструкций от CPL, но заменил его странные символы на фигурные скобки. То есть в C мы напишем

```
if (x > 0)
{
   this(...);
   that(...);
};
otherstuff(...);
```

Что же, в C от CPL практически ничего не осталось, ну разве что скобочки для группировки инструкций.

В заключение отметим, что использование знака равенства для обозначения присваивания однозначно осуждал Кристофер Страчи, один из авторов CPL. Много лет назад, на лекциях NATO, он сказал: «То, как люди учатся программировать, отвратительно. Они снова и снова учатся каламбурить. Они используют операции сдвига вместо умножения, запутывают код, используя битовые маски и числовые литералы, и вообще говорят одно, когда имеют в виду что-то совсем другое. Я думаю у нас не будет инженерного подхода к разработке ПО до тех пор, пока у нас не закрепятся профессиональные стандарты о том, как писать программы. А добиться этого можно лишь, начиная обучение программированию с того, как писать программы должным образом. Я убежден, что, в первую очередь, необходимо начать говорить именно то, что вы хотите сказать, а не что-то другое.»

Таков будет наш вывод. Мы должны научиться выражать, то что мы думаем. И язык Ада позволяет нам выразить это ясно, и в этом, в конечном счете, его сила.

CHAPTER

THIRTYNINE

СПИСОК ЛИТЕРАТУРЫ

[1] RTCA DO-178B / EUROCAE ED-12B. Software Considerations in Airborne Systems and Equipment Certification (December 1992).

[2] RTCA DO-178C / EUROCAE ED-12C. Software Considerations in Airborne Systems and Equipment Certification (December 2011).

[3] MISRA-C:2004, Guidelines for the use of the C language in critical systems (October 2004).

[4] Barbara Liskov and Jeannette Wing. "A behavioral notion of subtyping", ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 16, Issue 6 (November 1994), pp 1811-1841.

[5] RTCA DO-332 / EUROCAE ED-217. Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A (December 2011).

[6] AdaCore. High-Integrity Object-Oriented Programming Release 1.3 (July 2011), www.open-do.org/hi-oo-ada in Ada,

[7] Cyrille Comar and Pat Rogers. On Dynamic Plug-in Loading with Ada 95 and Ada 2005. AdaCore (2005). www.sigada.org/ada_letters/jun2005/Dynamic _plugin_loading_with-Pat Rogers.pdf

[8] ISO/IEC TR 24718:2004. Guide for the use of the Ada Ravenscar profile in high integrity systems (2004).

[9] Alan Burns and Andy Wellings. Concurrent and Real-Time programming in Ada 2005. Cambridge University Press (2006).

[10] Janet Barnes, Rod Chapman, Randy Johnson, James Widmaier, David Cooper and Bill Everett. Engineering the Tokeneer Enclave Protection Software. Published in ISSSE 06, the proceedings of the 1st IEEE International Symposium on Secure Software Engineering. IEEE (March 2006). www.sparkada.com.