Джон Барнс

При участии Бэн Брасгол.

Безопасное и надежное программное обеспечение

Ada 2012

Адаптировано для **SPARK 2014**

При поддержке

AdaCore

The GNAT Pro company

www.adacore.com

На основе редакции V.20150501

Вступление

Целью данной брошюры является продемонстрировать, как изучение языка Ада в целом и возможностей, введенных в редакциях Ада 2005, Ада 2012, в часности, 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 брощюру 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

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 всеобъемлящий 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.

Оглавление

1 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.

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.

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.

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.

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.

2 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.

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.

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.

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.

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.

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.

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.

«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.

3 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.

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 всеже 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.

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.

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
  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.

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:

$$Nfv\ (X) = 1000 \times (1 + X/100)^2 + 500 \times (1 + X/100)$$

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**.

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!

4 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.

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 С, 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 подрограмм 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 пакето 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. невложенные 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.

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).

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.

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;
        …
  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.

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.

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.

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;

  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.

5 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.

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;

  …  -- 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.

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 необнаруженными.

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.

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;
       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)
  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.

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 GNATstack, 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.

6 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.

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.

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.

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.

Контролируемые типы

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

1) объект создается;
2) объект прекращает существование;
3) 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;
  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.

7 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.

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.

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.

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.

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.

8 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.

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;
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.

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.

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].

9 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.

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#);
```

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.

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.

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.

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.

10 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 используюемых в управлении производством.

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 временны**ы**х условий.
- Необходим планировщик выполнения задач для эффективного использования ресурсов и попадания в установленные временны**ы**е 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.

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;
  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);
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.

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
       … -- 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.

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.

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.

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.

Время и планирования

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);
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.

11 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.

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.

(2) 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.

(3) При помощи тестирования. В этом случае мы запускаем какие-то примеры и размышляем над неожиданными результатами, пытаясь понять, что пошло не так.

(4) 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.

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.

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.

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.

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);

  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.

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.

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/**.

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
  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 с 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.

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.