Quentin Ochem (Квентин Окем)

Ada для программистов C++ или Java

Выпуск 1.0
23 июля 2013

Статья любезно предоставлена компанией AdaCore.
Перевод и дополнения выполнил Киркоров Сергей.

Минск 2016

Содержание

Первая глава. ПРЕДИСЛОВИЕ

В наше время может показаться, что разговор о языках программирования – это достаточно избитая тема. Технические войны прошлого десятилетия утихли, и сегодня мы видим множество высокоуровневых и прочно укрепившихся языков, предлагающих функциональность, которая может удовлетворить потребности любого программиста.

В качестве современных примеров можно привести языки Python, Java, C++, C# и Visual Basic. Они действительно позволяют очень быстро писать программный код, обладают высоким уровнем гибкости и предоставляют доступ к динамическим функциям, а компиляторы некоторых из них даже способны содействовать работе, предугадывая возможные намерения разработчика и предлагая варианты последующих действий на основе этих заключений.

Тем не менее, в попытке достижения универсальности для охвата всего рынка программного обеспечения, вышеупомянутые языки стали малопригодными для работы с системами, имеющими высокие требования к надежности и безопасности. Языки, используемые для написания программных приложений для систем самолетов, ракет, спутников, поездов и других устройств, выход из строя которых может нести угрозу человеческой жизни или сохранности критически важных объектов, должны соответствовать высоким стандартам, используемым при разработке программного обеспечения, гарантирующим надежность и целостность систем.

Концепция верификации – практика демонстрации того, что система работает и поддерживает производительность должным образом – является ключевой в таком роде программных оболочек. Верификация представляет собой комплексный процесс, состоящий из анализа, тестирования, статического анализа и методов формально–логического доказательства. Растущая зависимость современных систем от программного обеспечения и усложнение их структуры значительно усложняет эту задачу. Технологии и методики, которые еще десять или пятнадцать лет назад были вполне приемлемы, не соответствуют современным стандартам. К счастью, уровень развития аналитических технологий и доказательных методик также продвинулся вперед.

Последние версии языка программирования Ada, Ada 2005 и Ada 2012, делают возможным обеспечение повышенного уровня надежности и безопасности программного обеспечения. С момента своего появления в 1980–ых гг. язык Ada был разработан специально для разработки программного обеспечения для систем с повышенными требованиями к безопасности и надежности, и по сей день остается пригодным для реализации критически важных встраиваемых либо собственных

приложений. И в последнее время этому уделяется всё больше внимания. Каждая версия языка становилась выразительней. В частности, Ada 2012 предоставила новые возможности для контрактного программирования, которые ценны для любого проекта, где верификация является частью жизненного цикла разработки. Наряду с этими улучшениями языка, компилятор Ada и технологии инструментария на протяжении последних нескольких лет выдерживают темп развития как у широко распространённых средств вычислительной обработки данных. Среды разработки Ada доступны на большом количестве платформ и используются для самых ресурсоёмких приложений и сохранили совместимость с вычислительными разработками за прошлые несколько лет.

Не секрет, что в компании AdaCore развитие языка Ada – приоритетное направление, но AdaCore не будет настаивать, что Ada является идеальным решением; язык Ada не является решением для всего многообразия задач, как и любой другой язык программирования. В некоторых областях информационных технологий имеет смысл использовать другие языки, для которых существуют специфические библиотеки или среды разработки. Например, C++ и Java считаются хорошим выбором для разработки приложений для ПК или приложений, где уменьшение времени разработки является главной целью. В других областях, таких как веб–программирование или системное администрирование, предпочтение отдают другим формальным системам, таким как скриптовые и интерпретируемые языки. Чтобы выполнить требования, предъявляемые к программным средствам, ключевым моментом является выбор надлежащего технического подхода, с точки зрения языка и инструментария разработки программного обеспечения. Сила языка программирования Ada проявляется в областях, где обеспечение надежности и безопасности с высоким уровнем доверия к программным средствам является главной задачей.

Изучение нового языка не должно быть сложным. Парадигмы программирования не очень развились, начиная с объектно–ориентированного программирования, и те же парадигмы присутствуют, так или иначе, во многих широко используемых языках. Этот документ, таким образом, даст Вам обзор языка Ada, используя аналогии с C++ и Java. Это те языки, которые Вы, вероятно, будете, уже знать. Никакие предварительные знания Ada не требуются. Если Вы работаете над проектом, в котором теперь применяете язык программирования Ada и нуждайтесь в большем количестве примеров, или если Вы интересуетесь обучением программирования в Ada, или если Вы должны выступить с оценкой возможных языков, которые будут использоваться для новой разработки, это руководство для Вас.

Этот документ был подготовлен Квентином Окемом, с дополнениями и обзором от Ричарда Кеннера, Альберта Ли, и Бен Бросгол.

Вторая глава. ОСНОВНЫЕ ПОНЯТИЯ

Ada реализует подавляющее большинство концепций программирования, которые приняты в C++ и Java: классы, наследование, шаблоны (обобщения), и т.д. Его синтаксис мог бы казаться странным, все же. Этот синтаксис не получен из популярного C стиля нотации с его вполне достаточным использованием скобок; скорее это использует более описательный синтаксис, который близок к языку Pascal. Во многих отношениях, Ada более простой язык — синтаксис нацелен больше на облегчение понимания кода программы при его чтении, чем на быстрое его создания и записи кода заумным способом с минимальным количеством символов. Например, полное написание слов таких как, **`begin`**, и **`end`** используются вместо изогнутых фигурных скобок. Условия записываются, используя **`if`**, **`then`**, **`elsif`**, **`else`**, и **`end if`**. Оператор присваивания Ada не делает удвоение символа в выражение, однозначно устранив любые опечатки в выражении, приводящие к изменению алгоритма или побочным эффектам, которое получается при наборе **`=`** там, где должен быть **`==`**.

Все языки обеспечивают один или несколько способов выразить комментарии. В Ada два последовательных дефиса **`--`** отмечают запуск комментарий, который продолжается до конца строки. Это точно то же, как использующий двойной слеш **`//`** для комментариев в C++ и Java. Нет никакого эквивалента **`/*... */`** блочных комментариев, в Ada вместо этого используется два последовательных дефиса многократно, для каждой строки комментария.

Компиляторы Ada более строги с проверкой типа и диапазона, чем для большей части программистов использующие C++ и Java. Большинство начинающих программистов языка Ada встречаются с множеством предупреждений и сообщений об ошибках, так как привыкли кодировать, по их мнению, более креативно. Эти сообщения помогают обнаруживать проблемы и уязвимости во время компиляции – в начале цикла разработки. Кроме того, динамичные проверки (такие как граничные проверки массива) обеспечивают проверку, которая не могла быть сделана во время компиляции. Динамические проверки производятся во время выполнения кода, подобно тому, как это сделано в Java.

Идентификаторы языка Ada и зарезервированные слова нечувствительны к регистру. Написание идентификаторов **`VAR`**, **`var`** и **`VaR`** рассматривают как одни и те же; аналогично **`begin`**, **`BEGIN`**, **`Begin`** и т.д. Определенные знаки языка, такие как согласованные символы, греческие или российские буквы, и Азиатские алфавиты, приемлемы для использования. Идентификаторы могут включать буквы, цифры и подчеркивания, но должны всегда запускаться с буквой. Есть 73 зарезервированных слова в языке Ada, которые не могут использоваться в качестве идентификаторов.

Зарезервированные слова языка Ada приведены в таблице 1:

Таблица 1.

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

Язык программирования Ada разработан, чтобы исходный код программы был переносимым. Компиляторы Ada должны следовать точно определенному международному (ISO) стандарту спецификации языка с четко задокументированными областями свободы поведение, которая зависит от реализации разработчиком компилятора. Это дает возможность написать независимое от реализации компилятора приложение в Ada и удостовериться, что у него будет тот же эффект независимо от платформы и компиляторов.

Программирования на языке Ada есть действительное воплощение основного намерения – парадигмы многократного использования исходных текстов языка, которая обеспечивается поддержкой для программиста или другими словами иметь такой функционал как проверка контракта во время выполнения, управление задачами, объектно–ориентированное программирование и обобщения. Язык Ada используется для эффективного программирования в драйверах устройств, обработчиках прерываний и других низкоуровневых функциях. Сегодня программные модули, написанные на языке Ada можно найти в устройствах с высокими требованиями к временным пределам на реакцию событий в системе и скорости их обработки, при ограничении выделенной памяти и потребляемой мощности. Но язык Ada также используется для программирования больших интегрированных систем, работающих на рабочих станциях, серверах и суперкомпьютерах.

Третья глава. СТРУКТУРА ЕДИНИЦЫ КОМПИЛЯЦИИ

Стиль программирования на C++ обычно способствует использованию двух отличных файлов: заголовочные файлы раньше определяли спецификации (.h, .hxx, .hpp), и файлы реализации, которые содержат исполняемый код (.c, .cxx, .cpp). Однако различие между файлами спецификаций и реализаций компилятор не осуществляет, но есть возможность обойти это ограничение. Потребность в модульности может возникнуть при необходимости реализации, например, шаблонов и встраиваемого кода.

Компиляторы Java предполагают, что и реализация и спецификация будут в том же .java файле. (Да, шаблоны разработки позволяют использовать интерфейсы, чтобы разделить спецификацию от реализации до некоторой степени, но это за пределами объема этого описания.)

Ада поверхностно подобна случаю C++: единицы компиляции Ады обычно разделены на две части, спецификацию и тело. Однако то, что входит в те файлы, более предсказуемо и для компилятора и для программиста.

В GNAT единицы компиляции сохранены в файлах с .ads расширением для спецификаций и с .adb расширением для реализаций.

Для наглядности, приведем примеры известного приложения “Привет Мир” на трех языках:

[Ada]

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

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

[C++]

```
#include <iostream>
using namespace std;

int main(int argc, const char* argv[]) {
     cout << "Hello World" << endl;
}
```

[Java]

```
public class Main {
     public static void main(String [] argv) {
          System.out.println ("Hello World");
     }
```

```
}
```

Первая строка Ады, которую мы видим, это **with,** объявляя, что модуль (в этом случае, подпрограмма **Main**) потребует служб пакета *Ada.Text_IO*. Это отличается от того, как **#include** работает в C++, в котором он не делает ничего, в логическом смысле, а выполняет только вставить и затем скомпилировать код в **Main**. Спецификатор **with** указывает компилятору, что предопределенный интерфейс *Ada.Text_IO* делает видимым код содержащий средства ввода–вывода, чтобы ссылаться на него при кодировании в модуле **Main**. Обратите внимание на то, что у этой конструкции нет прямого аналога в Java, где весь CLASSPATH (ПУТЬ К КЛАССУ) всегда доступен. Кроме того, имя, “Main” для основной подпрограммы, было выбрано для непротиворечивости с C++ и стилем Java, так как в Ada это имя может быть зависеть только от выбора программиста.

Спецификатор **use** – эквивалент использования **using namespace** (пространства имен) в C++ или **import** (импорта) в Java (хотя не было необходимости использовать **import** в примере для Java). Это позволяет Вам опускать полное имя пакета при обращении к модулям объявленных посредством **with**. Без спецификатора **use**, любая ссылка на элементы *Ada.Text_IO* должна была бы быть полностью определена с именем пакета. Строка *Put_Line* тогда должна быть:

```
Ada.Text_IO.Put_Line ("Hello World");
```

У зарезервированного слова **package** (пакет) есть различные значения в Ada и Java. В Java пакет используется в качестве пространства имен для классов. В Ada это часто – единица компиляции. В результате Ada позволяет иметь намного больше пакетов, чем Java.

У спецификаций пакета Ada следующая структура:

```
package Package_Name is
    -- public declarations
private
    -- private declarations
end Package_Name;
```

Реализация тела пакета (записанном в .adb файле) имеет следующую структуру:

```
package body Package_Name is
    -- implementation
end Package_Name;
```

Зарезервированное слово **private** используется, чтобы отметить начало частной секции спецификации пакета. Разделяя спецификацию пакета на частные и общедоступные части, становится возможным сделать объект доступным для использования при сокрытии его реализации. Например, общее использование объявляет record (запись – Ada's struct), чьи поля только видимы к его пакету а не к вызывающей стороне. Это позволяет вызывающей стороне относиться к объектам того типа, но не изменять любое свое содержание непосредственно.

Тело пакета содержит код реализации и только доступно для внешнего кода через объявления в спецификации пакета.

Объект, объявленный в приватной части пакета в Ada, примерно эквивалентен защищенному члену класса Java или C++. Объект, объявленный в теле пакета Ады, примерно эквивалентен приватному члену класса Java или C++.

Четвертая глава. ОПЕРАТОРЫ, ОБЪЯВЛЕНИЯ, И УПРАВЛЯЮЩИЕ СТРУКТУРЫ

1. Операторы и объявления

Следующие примеры кода все эквивалентны, и иллюстрируют использование комментариев и работу с целочисленными переменными:

[Ada]

```ada
--
-- Ada program to declare and modify Integers
--
procedure Main is
      -- Variable declarations
      A, B : Integer := 0;
      C : Integer := 100;
      D : Integer;
begin
      -- Ada uses a regular assignment statement for incrementation.
      A := A + 1;

      -- Regular addition
      D := A + B + C;
end Main;
```

[C++]

```cpp
/*
* C++ program to declare and modify ints
*/
int main(int argc, const char* argv[]) {
      // Variable declarations
      int a = 0, b = 0, c = 100, d;

      // C++ shorthand for incrementation
      a++;

      // Regular addition
      d = a + b + c;
}
```

```
[Java]
/*
* Java program to declare and modify ints
*/
public class Main {
      public static void main(String [] argv) {
            // Variable declarations
            int a = 0, b = 0, c = 100, d;

            // Java shorthand for incrementation
            a++;

            // Regular addition
            d = a + b + c;
      }
}
```

Операторы завершены точками с запятой на всех трех языках. В Ada блоки кода окружены зарезервированными словами, начинаются и заканчиваются, а не изогнутыми фигурными скобками. В C++ и коде Java можем использовать стиль комментария как многострочный так и одной строки, и только однострочный комментарий в коде Ada.

Синтаксис языка Ada требует объявления переменных производить в определенной секции. Эту секцию называют *declarative part* (декларативная секция). Декларативная секция как видно в этом примере – это перед ключевым словом **begin**. Объявления переменной начинается в Ada с идентификатора переменной, в противоположность C++ и Java, где объявление переменной начинается с типа переменной (также необходимо отметить использование Ada “**:**” разделитель). Определение инициализаторов отличается также: в Ada выражение инициализации может примениться к многократным переменным (но будет вычисляться отдельно для каждого), тогда как в C++ и Java каждая переменная инициализирована индивидуально. На всех трех языках, если Вы используете функцию в качестве инициализатора, и эта функция возвращает различные значения при каждом вызове, то каждая переменная будет инициализирована к различному значению.

Давайте идти дальше к обязательным операторам. Ada не обеспечивает краткие выражения инкрементных операций: **++** , или декрементных операций: **--**. Необходимо использовать полный оператор присваивания. Чтобы выполнить присвоение значения в Ada используется символ **:=**. В отличие от C++ и Java где используется символ **=** и не может использоваться **:=** в качестве части выражения. Так, оператор как **`A: = B: = C;`** не понятен компилятору Ada, и ни один компилятор не понимает оператор как “**`if A := B then...`**”. Компиляторы Ada, C++ и Java выдадут ошибки времени компиляции.

Вы можете вложить блок кода во внешнем блоке, если Вы хотите создать внутренний локальный блок кода с локальными переменными:

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

procedure Main is
begin
      Put_Line ("Before the inner block");

      declare
            Alpha : Integer := 0;
      begin
            Alpha := Alpha + 1;
            Put_Line ("Now inside the inner block");
      end;

      Put_Line ("After the inner block");
end Main;
```

Если у Вас нет объявлений, то нормально иметь пустую декларативную секцию, или опустить декларативную секцию полностью – иметь только секцию с исполняемым внутренним блоком, которая начинается с **`begin`**. Однако, не нормально иметь пустую последовательность операторов. Вы должны, по крайней мере, обеспечить нуль оператор: " **`null;`** ", который ничего не делает и указывает, что пропуск операторов намеренный.

2. Условные операторы

Использование оператора **`if`**:

```
[Ada]

if Variable > 0 then
      Put_Line (" > 0 ");
elsif Variable < 0 then
      Put_Line (" < 0 ");
else
      Put_Line (" = 0 ");
end if;

[C++]

if (Variable > 0)
      cout << " > 0 " << endl;
else if (Variable < 0)
      cout << " < 0 " << endl;
else
      cout << " = 0 " << endl;

[Java]

if (Variable > 0)
      System.out.println (" > 0 ");
else if (Variable < 0)
      System.out.println (" < 0 ");
else
      System.out.println (" = 0 ");
```

В Ada, все, что появляется между ключевыми словами **`if`** и **`then`** – это условное выражение, нет никакой необходимости в круглых скобках.

Операторы сравнения – те же, за исключением операторов равенства (**=**) и неравенства (**/=**). Для выполнения логических операций английские слова **not**, **and** и **or** заменены символом **!**, **&**, и **|**, соответственно.

При записи булевых выражений, более естественно воспринимаются **&** и **|**, чем используемые в C++ и Java **&&** и **||**. Различие в том, что **&&** и **||** являются операторами сокращенного вычисления логического выражения, которые оценивают условия только по мере необходимости, а **&** и **|** безоговорочно оценит все условия. В Ada, **and** и **or** оценит все условия; и затем по результату сокращенного вычисления логического выражения выполнит соответствующий код.

Вот то, на что похожи операторы **switch/case**:

[Ada]

```
case Variable is
      when 0 =>
            Put_Line ("Zero");
      when 1 .. 9 =>
            Put_Line ("Positive Digit");
      when 10 | 12 | 14 | 16 | 18 =>
            Put_Line ("Even Number between 10 and 18");
      when others =>
            Put_Line ("Something else");
end case;
```

[C++]

```
switch (Variable) {
      case 0:
            cout << "Zero" << endl;
            break;
      case 1: case 2: case 3: case 4: case 5:
      case 6: case 7: case 8: case 9:
            cout << "Positive Digit" << endl;
            break;
      case 10: case 12: case 14: case 16: case 18:
            cout << "Even Number between 10 and 18" << endl;
            break;
      default:
            cout << "Something else";
}
```

```
[Java]
switch (Variable) {
      case 0:
            System.out.println ("Zero");
            break;
      case 1: case 2: case 3: case 4: case 5:
      case 6: case 7: case 8: case 9:
            System.out.println ("Positive Digit");
            break;
      case 10: case 12: case 14: case 16: case 18:
            System.out.println ("Even Number between 10 and 18");
            break;
      default:
            System.out.println ("Something else");
}
```

В Ada строки **case** и **end case** окружают целый оператор выбора, и каждый случай условия начинается с **when**. Так, при программировании в Ada замените **switch** на **case** и **case** замените на **when**.

Операторы выбора в Ada требуют использования дискретных типов (целые числа или типы перечисления), и требуют, чтобы все возможные случаи были покрыты оператором **when**. Если не все случаи будут обработаны, или если двойные случаи будут существовать, то программа не скомпилирует. Случай по умолчанию, значение по **default**: в C++ и Java, может быть определен в Ada , используя “ **when others=>** “.

В Ada инструкция “ **break** “ присутствует неявно, выполнение программы никогда не будет проваливаться к последующим случаям. Чтобы объединить случаи, Вы можете определить использование диапазонов “ **..** “ и перечислите непересекающееся использование значений “ **|** “, который аккуратно заменяет многократные операторы выбора, замеченные в версиях Java и C++.

3. Циклы

В Ada циклы всегда запускаются с зарезервированного слова **loop** и заканчиваются **end loop**. Чтобы выйти из цикла, используйте **exit**. В C++ и Java, эквивалентом является **break**. Оператор, который может определить завершающееся условие, имеет синтаксис ” **exit when** “. Ключевому слову **loop**, которое открывает блок цикла, может предшествовать **while** или **for**.

Цикл с условием **while** – самый простой и очень похож во всех трех языках:

```
[Ada]
while Variable < 10_000 loop
      Variable := Variable * 2;
end loop;
```

[C++]

```
while (Variable < 10000) {
      Variable = Variable * 2;
}
```

[Java]

```
while (Variable < 10000) {
      Variable = Variable * 2;
}
```

Оператор цикла Ada – **for**, однако, очень отличается от **for** в C++ и Java. Этот оператор цикла, всегда постепенно увеличивает или постепенно уменьшает индекс цикла в дискретном диапазоне. Индекс цикла (или “параметр цикла” в языке Ada) локален для кода внутри цикла и неявно постепенно увеличивает или постепенно уменьшает свое значение при каждой итерации оператора цикла; программа не может непосредственно изменить значение индекса цикла. Тип параметра цикла получен из диапазона. Диапазон всегда дается в порядке возрастания, даже если цикл выполняет итерации в порядке убывания. Если связанный запуск больше, чем связанное окончание, интервал, как полагают, пуст, и содержание цикла не будет выполняться. Чтобы определить итерацию цикла в порядке убывания, используется зарезервированное слово **reverse**. Вот примеры циклов, демонстрирующие оба направления:

[Ada]

```
-- Outputs 0, 1, 2, ..., 9
for Variable in 0 .. 9 loop
      Put_Line (Integer'Image (Variable));
end loop;

-- Outputs 9, 8, 7, ..., 0
for Variable in reverse 0 .. 9 loop
      Put_Line (Integer'Image (Variable));
end loop;
```

[C++]

```
// Outputs 0, 1, 2, ..., 9
for (int Variable = 0; Variable <= 9; Variable++) {
      cout << Variable << endl;
}

// Outputs 9, 8, 7, ..., 0
for (int Variable = 9; Variable >=0; Variable--) {
      cout << Variable << endl;
}
```

[Java]

```
// Outputs 0, 1, 2, ..., 9
for (int Variable = 0; Variable <= 9; Variable++) {
      System.out.println (Variable);
}

// Outputs 9, 8, 7, ..., 0
for (int Variable = 9; Variable >= 0; Variable--) {
      System.out.println (Variable);
}
```

Ada использует Integer type's 'Image – это Атрибут изображения Целочисленного типа, чтобы преобразовать численное значение в Последовательность String. Нет никакого неявного преобразования между Целым числом *Integer* и Последовательностью *String*, которая есть в C++ и Java. У нас будет более всесторонний взгляд на такие атрибуты позже.

Просто выразить итерацию по содержанию контейнера (например, массив, список или карта) в Ada и Java.

Например, предполагая, что *Int_List* определен как массив Целочисленных значений, Вы можете использовать:

[Ada]

```
for I of Int_List loop
      Put_Line (Integer'Image (I));
end loop;
```

[Java]

```
for (int i : Int_List) {
      System.out.println (i);
}
```

Пятая глава. СИСТЕМА ТИПОВ

1. Строгая типизация

Одна из основных характеристик языка программирования Ada – свой строгий контроль типов (т.е., относительное отсутствие неявных преобразований типов). Это может потребовать некоторых усилий, чтобы привыкнуть к этой строгой типизации. Например, Вы не можете разделить целое число на число с плавающей запятой. Вы должны выполнить операцию деления, используя значения того же типа, таким образом, одно значение должно быть явно преобразовано, чтобы соответствовать типу другого (в этом случае, более вероятное преобразование от целого числа к числу с плавающей запятой, чтобы получить в результате деления число с плавающей запятой). Ада разработана, чтобы гарантировать, что то, что сделано программой – это то, что подразумевалось программистом, оставляя как можно меньше неопределенности для интерпретации компилятором. Давайте взглянем на следующий пример:

[Ada]

```
procedure Strong_Typing is
      Alpha : Integer := 1;
      Beta : Integer := 10;
      Result : Float;
begin
      Result := Float (Alpha) / Float (Beta);
end Strong_Typing;
```

[C++]

```
void weakTyping (void) {
      int alpha = 1;
      int beta = 10;
      float result;

      result = alpha / beta;
}
```

[Java]

```
void weakTyping () {
      int alpha = 1;
      int beta = 10;
      float result;

      result = alpha / beta;
}
```

Эти три программы эквивалентны? Может казаться, что Ada просто добавляет дополнительную сложность, вынуждая Вас сделать

преобразование из Целого числа в число с плавающей запятой явным образом. Фактически это значительно изменяет поведение вычисления.

В то время как код Ada выполняет работу с плавающей точкой 1.0 / 10.0 и хранит 0.1 в Результате: `Result` , C++ и версии Java вместо этого хранят 0.0 в результате: `result`. Это вызвано тем, что C++ и версии Java выполняют целочисленную операцию между двумя целочисленными переменными: 1 / 10 в результате получаем 0. Лишь после целочисленного деления результат преобразован в число типа плавающей запятой и сохранен. Очень трудно определение местоположение Ошибки этого вида в сложных частях кода. Систематическая спецификация того, как работа должна быть интерпретирована, помогает избегать этого класса ошибок. Если целочисленное деление было фактически предназначено в случае Ada, все еще необходимо явно преобразовать конечный результат к типу плавающей запятой *Float* :

```
-- Perform an Integer division then convert to Float
Result := Float (Alpha / Beta);
```

В Ada литерал с плавающей точкой должен быть записан и с интегральной и с десятичной частью. 10 не допустимый литерал для значения с плавающей точкой, в то время как 10.0 правильный.

2. Предопределенные типы языка

Основные скалярные типы, предопределенные Ada–ой, являются *Integer*, *Float, Boolean* и *Character*. Они соответствуют int, float, bool/boolean, and char соответственно. Имена для этих типов не зарезервированные слова; они – регулярные идентификаторы.

3. Типы определенные приложением

Система типов Ada поощряет программистов думать о данных на высоком уровне абстракции. Компилятор будет иногда выводить простую эффективную машинную команду для полной строки исходного кода (и некоторые инструкции могут быть устранены полностью). Осторожный программист беспокоится о том, что работа действительно целесообразная в реальном мире, была бы выполнена, так и беспокоится по поводу производительности при выполнении этой работы.

Следующий пример ниже определяет две различных метрики: площадь и расстояние. Смешивание этих двух метрик должно быть сделано с большой осторожностью, поскольку определенные операции не целесообразны, как добавление площади к расстоянию. Другие знания из прикладной области требуют другой ожидаемой семантики; например, умножение двух расстояний. Чтобы помочь избегать ошибок, Ada требует, чтобы каждый из бинарных операторов “+”, “–”, “*”, и “/” для целого числа или типов с

плавающей точкой взял операнды того же типа и возвратил значение того же типа.

```
procedure Main is
      type Distance is new Float;
      type Area is new Float;
      D1 : Distance := 2.0;
      D2 : Distance := 3.0;
      A : Area;
begin
      D1 := D1 + D2; -- OK
      D1 := D1 + A; -- NOT OK: incompatible types for "+" operator
      A := D1 * D2; -- NOT OK: incompatible types for ":=" assignment
      A := Area (D1 * D2); -- OK
end Main;
```

Даже при том, что типы Расстояние (`Distance`) и типы Площади (`Area`), в примере выше, – это просто тип с плавающей запятой (`Float`), компилятор не позволяет произвольное смешивание значений этих различных типов. Явное преобразование необходимо (которое не означает обязательное добавление какого–либо дополнительного объектного кода).

Предопределенные правила Ada не совершенны. Признается существование некоторых проблематичных случаев (например, умножая две переменные типа Расстояние, может быть присвоено переменной типа Расстояние), или запрет на некоторые полезные случаи (например, умножая две переменные типа Расстояние, должен получить тип Площадь). Эти ситуации могут быть обработаны через другие механизмы. Предопределенная операция может быть идентифицирована как абстрактная (`abstract`), чтобы сделать ее недоступной; перегрузка операции может быть использована, чтобы дать новые интерпретации к существующим операторным символам. Например, позволяя оператору возвращать значение из определенного типа, отличного от типа операндов. В более общем случае, GNAT предоставил средство, которое помогает выполнять проверку размерности.

Тип Перечисления Ады аналогичен *enum* в C++ и Java.

```
[Ada]
type Day is
      (Monday,
      Tuesday,
      Wednesday,
      Thursday,
      Friday,
      Saturday,
      Sunday);
```

[C++]

```
enum Day {
      Monday,
      Tuesday,
      Wednesday,
      Thursday,
      Friday,
      Saturday,
      Sunday};
```

[Java]

```
enum Day {
      Monday,
      Tuesday,
      Wednesday,
      Thursday,
      Friday,
      Saturday,
      Sunday}
```

Но даже при том, что такие перечисления могут быть реализованы, используя машинное слово на уровне языка, Ada не перепутает, что *Monday* имеет тип *Day* и не тип *Integer*. Все же, вы можете сравнить *Day* с другим *Day*. Для того чтобы определить реализацию перечислений как числовые значения, которые соответствуют перечислимым величинам в C++, Вы включаете их в исходный перечислимый оператор:

[C++]

```
enum Day {
      Monday = 10,
      Tuesday = 11,
      Wednesday = 12,
      Thursday = 13,
      Friday = 14,
      Saturday = 15,
      Sunday = 16};
```

Но в Ada Вы должны использовать оба определения, как для типа *Day*, а так и отдельный пункт представления для него, как приведено в примере:

[Ada]

```
for Day use
      (Monday => 10,
      Tuesday => 11,
      Wednesday => 12,
      Thursday => 13,
      Friday => 14,
      Saturday => 15,
      Sunday => 16);
```

4. Диапазоны Типа

Контракты могут быть связаны с типами и переменными, чтобы совершенствовать значения и определить то, что считают допустимыми значениями. Наиболее распространенный вид контракта – ограничение диапазона, начатое с зарезервированного слова диапазона, например:

```
procedure Main is
     type Grade is range 0 .. 100;

     G1, G2 : Grade;
     N : Integer;
begin
     ... -- Initialization of N
     G1 := 80; -- OK
     G1 := N; -- Illegal (type mismatch)
     G1 := Grade (N); -- Legal, run-time range check
     G2 := G1 + 10; -- Legal, run-time range check
     G1 := (G1 + G2)/2; -- Legal, run-time range check
end Main;
```

В вышеупомянутом примере *Grade* – новый целочисленный тип, связанный с проверкой диапазона. Проверки диапазона динамичные и предназначаются, чтобы осуществить свойство, что ни у какого объекта данного типа не может быть значения вне указанного диапазона. В этом примере первое присвоение на G1 корректно и не вызовет во время выполнения excepгion. Присвоение N к G1 недопустимо, так как *Grade* – другой тип, чем *Integer* число. Преобразование N в *Grade* делает присвоение законным, и проверка диапазона на преобразовании подтверждает, что значение в 0.. 100. Присвоение G1+10 к G2 законно, так как + для *Grade* возвращается *Grade* (обратите внимание на то, что литеральные 10 интерпретируются как значение *Grade* в этом контексте), и снова производится проверка диапазона.

Заключительное присвоение иллюстрирует интересный, но тонкий тезис. Подвыражение G1 + G2 может быть вне диапазона *Grade*, но конечный результат будет в диапазоне. Тем не менее, в зависимости от представления, выбранного для *Grade*, дополнение может переполниться. Если компилятор представляет значения *Grade* как подписанные 8–разрядные целые числа (т.е., числа машины в диапазоне–128.. 127), тогда сумма G1+G2 может превысить 127, приведя к целочисленному переполнению. Чтобы предотвратить это, Вы можете использовать явные преобразования и выполнить вычисление в достаточно большом целочисленном типе, например:

```
G1 := Grade (Integer (G1) + Integer (G2)) / 2);
```

Проверки диапазона полезны для обнаружения ошибок как можно раньше. Однако может быть некоторое влияние на производительность.

Современные компиляторы действительно знают, как удалить избыточные проверки, и Вы можете деактивировать эти проверки в целом, если у Вас есть достаточная уверенность, что Ваш код будет функционировать правильно.

Типы могут быть получены из представления любого другого типа. Новый производный тип может быть связан с новыми ограничениями и операциями. Возвращаясь к примеру с типом *Day*, можно записать:

```
type Business_Day is new Day range Monday .. Friday;
type Weekend_Day is new Day range Saturday .. Sunday;
```

Так как это новые типы, неявные преобразования не позволены. В этом случае более естественно создать новый набор ограничений для того же типа, вместо того, чтобы делать абсолютно новые типы. Это и есть идея за создание 'подтипов' в Ada. Подтип, по своей сути, является той же самой сущностью, что и оригинальный тип, но при этом он может иметь ограниченный диапазон значений оригинального типа. Например:

```
subtype Business_Day is Day range Monday .. Friday;
subtype Weekend_Day is Day range Saturday .. Sunday;
subtype Dice_Throw is Integer range 1 .. 6;
```

Эти объявления не создают новые типы, просто новые имена для ограниченных диапазонов их базовых типов.

Значение подтипа могут использоваться везде, где могут использоваться значения оригинального типа, а также значения других подтипов этого оригинального типа. При этом, любая попытка присваивания переменной такого подтипа значения выходящего за границы указанного для этого подтипа диапазона допустимых значений будет приводить к исключительной ситуации *Constraint_Error* (проще говоря – ошибке программы). Такой подход облегчает обнаружение ошибок, а также, позволяет отделить обработку ошибок от основного алгоритма программы. Например:

```
subtype Data is Integer;
subtype Age is Data range 0 . . 140;
My_Age : Age;
Height : Integer;

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

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

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

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

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

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

5. Обобщенные контракты типа: предикаты подтипа

Проверки диапазона – специальная форма контрактов типа; более общий метод обеспечен предикатами подтипа Ada, представленными в Ada 2012. Предикат подтипа – булево выражение, определяющее условия, которые требуются для данного типа или подтипа. Например, подтип *Dice_Throw*, показанный выше, может быть определен следующим образом:

```
subtype Dice_Throw is Integer
      with Dynamic_Predicate => Dice_Throw in 1 .. 6;
```

Придаточное предложение начинается с *with* и представляет собой Ada `aspect' (далее 'аспект'), который является дополнительной информацией для обеспечения таких объектов как типы и подтипы. Аспект Dynamic_Predicate – самая общая форма. В выражении предиката, имя *(sub)type* относится к

текущему значению переменной данного *(sub)type*. Условия предиката проверяются при присвоении, передаче параметров, и в нескольких других контекстах. Есть форма “Static_Predicate”, которая представляют некоторую оптимизацию, и ограничения на форме этих предикатов. Рассмотрение этой формы предикатов за пределами объема этого документа.

Конечно, предикаты полезны не только в просто выражения диапазонов. Они могут использоваться, чтобы представлять типы с произвольными ограничениями, в определенных типах с разрывами, например:

```
type Not_Null is new Integer
     with Dynamic_Predicate => Not_Null /= 0;
type Even is new Integer
     with Dynamic_Predicate => Even mod 2 = 0;
```

6. Атрибуты

Атрибуты запускаются с единственного апострофа (“галочка”), и они позволяют Вам запрашивать свойства и выполнять определенные действия над объявленными объектами, такими как типы, объекты и подпрограммы. Например, Вы можете определить первые и последние границы скалярных типов, получить размеры объектов и типов, и преобразовать значения в строку и строку в значение. Этот раздел обеспечивает обзор того, как работают атрибуты. Для получения дополнительной информации о многих атрибутах, определенных языком, Вы можете обратиться непосредственно к Справочнику – Ada Language Reference Manual.

Атрибуты *‘Image* и *‘Value* позволяют Вам преобразовывать скалярную величину в *String* и наоборот. Например:

```
declare
     A : Integer := 99;
begin
     Put_Line (Integer’Image (A));
     A := Integer’Value ("99");
end;
```

Определенные атрибуты обеспечены только для определенных видов типов. Например, атрибуты *‘Val* и *‘Pos* для типа перечисления с назначенными дискретными значениями в соответствии с их позиции в типе перечисления. Пример одного из способов переместиться к следующему символу таблицы ASCII:

```
[Ada]
declare
      C : Character := 'a';
begin
      C := Character'Val (Character'Pos (C) + 1);
end;
```

Более краткий способ получить следующее значение в Ada состоит в том, чтобы использовать атрибут *'Succ*:

```
declare
      C : Character := 'a';
begin
      C := Character'Succ (C);
end;
```

Вы можете получить предыдущее значение, используя атрибут *'Pred*. Вот эквивалент в C++ и Java:

```
[C++]
char c = 'a';
c++;

[Java]
char c = 'a';
c++;
```

Другие интересные примеры это атрибуты *'First* и *'Last* , которые, соответственно, возвращают первые и последние значения скалярного типа. Используя 32–разрядные целые числа (32–bit integers), например, *Integer'First* возвращает 2^{31}, и *Integer'Last* возвращает $2^{31} – 1$.

7. Массивы и строки

Массивы C++ это указатели со смещениями, но для Ada и Java это не так. Массивы на последних двух языках не взаимозаменяемы с операциями на указателях и типы массивов считаются самостоятельными сущностью языка. Массивы в Ada выделили семантику, такую как доступность границ массива во времени выполнения. Поэтому, не обработанные переполнения массива невозможны, если проверки не подавлены. Любой дискретный тип может служить индексом массива, и Вы можете определить обе и начало и окончание границ – нижняя граница должна не обязательно быть 0. Тип массива должны быть явно предварительно объявлен до объявления объекта этого типа массива.

Вот пример объявления массива 26 знаков и инициализация значениями от 'a' до ‘z’:

```
[Ada]
declare
      type Arr_Type is array (Integer range <>) of Character;
      Arr : Arr_Type (1 .. 26);
      C : Character := 'a';
begin
      for I in Arr'Range loop
            Arr (I) := C;
            C := Character'Succ (C);
      end loop;
end;

[C++]
char Arr [26];
char C = 'a';
for (int I = 0; I < 26; ++I) {
      Arr [I] = C;
      C = C + 1;
}

[Java]
char [] Arr = new char [26];
char C = 'a';
for (int I = 0; I < Arr.length; ++I) {
      Arr [I] = C;
      C = C + 1;
}
```

В C++ и Java, только размер массива дан во время объявления. В Аде индексные диапазоны массива определены, используя две величины (верхнею и нижнею границу) дискретного типа. В этом примере описание типа массива определяет использование Целого числа как индексный тип, но не обеспечивает ограничений (использование <>, объявленный ‘box’, чтобы не определять “ограничения”). Ограничения определены в объектном объявлении, как диапазон от 1 до 26, включительно. У массивов есть атрибут, называется *‘Range*. В нашем примере *Arr’Range* может также быть выражен как *Arr’Firs t.. Arr’Last*; оба выражения приведут к диапазону " 1 .. 26 ". Таким образом, атрибут *‘Range* предоставляет границы для нашего *for* цикла. Нет никакого риска неправильного присвоения ни одной из границ цикла, как можно было бы сделать в C++, где *“I <= 26”* может быть определено как условие конца цикла.

Как в C++, так и в Ada, *Strings* – массивы *Characters*. Строковый класс C++ или Java – эквивалент типа Ada *Ada.Strings.Unbounded_String*, который предлагает дополнительные возможности в обмен на дополнительные накладные расходы. Строки Ada, не разграничены со специальным знаком ‘\0’, как они определяются в C++. В этом нет необходимости, потому что Ада

использует границы массива, чтобы определить, где строка начинается и оканчивается.

Предопределенный тип *String* Ada очень прост в использовании:

```
My_String : String (1 .. 26);
```

В отличие от C++ и Java, Ada не предлагает escape–последовательности, такие как '\n'. Вместо этого явные значения от пакета ASCII должны быть связаны (через оператора связи, &). Здесь, например, демонстрируется то, как инициализировать строку текста, заканчивающегося новой строкой:

```
My_String : String := "This is a line with a end of line" & ASCII.LF;
```

Как Вы видите здесь, что нет никакой необходимости в ограничениях для этого переменного определения. Данное начальное значение позволяет автоматическое определение границ My_String.

Ada предлагает высокоуровневые операции для копирования, разделения и присвоения значений к массивам. Мы начнем рассмотрение с операций присвоения. В C++ или Java, оператор присваивания не делает копию значения массива, он только копирует адрес или ссылку на целевую переменную. В Ada дублировано фактическое содержание массива. Чтобы получить вышеупомянутое поведение, фактические типы указателей должны были бы определяться и использоваться.

[Ada]

```
declare
      type Arr_Type is array (Integer range <>) of Integer
      A1 : Arr_Type (1 .. 2);
      A2 : Arr_Type (1 .. 2);
begin
      A1 (1) := 0;
      A1 (2) := 1;

      A2 := A1;
end;
```

[C++]

```
int A1 [2];
int A2 [2];

A1 [0] = 0;
A1 [1] = 1;

for (int i = 0; i < 2; ++i) {
      A2 [i] = A1 [i];
}
```

[Java]

```
int [] A1 = new int [2];
int [] A2 = new int [2];

A1 [0] = 0;
A1 [1] = 1;

A2 = Arrays.copyOf(A1, A1.length);
```

Во всех примерах выше, у источника и целевых массивов должно быть точно то же число элементов. Ada позволяет Вам легко определять часть, или разделять последовательность элементов массива. Таким образом, Вы можете записать следующее:

[Ada]

```
declare
      type Arr_Type is array (Integer range <>) of Integer
      A1 : Arr_Type (1 .. 10);
      A2 : Arr_Type (1 .. 5);
begin
      A2 (1 .. 3) := A1 (4 .. 6);
end;
```

Это присваивает 4–е, 5–е, и 6–е элементы A1 в 1–е, 2–е, и 3–и элементы A2. Обратите внимание на то, что только длина имеет здесь значение: индексы не должны быть равными; значения индексов скользят автоматически.

Ada также предлагает операции сравнения высокого уровня, которые сравнивают содержание массивов, в противоположность сравнения их адресов (например как в C++):

[Ada]

```
declare
      type Arr_Type is array (Integer range <>) of Integer;
      A1 : Arr_Type (1 .. 2);
      A2 : Arr_Type (1 .. 2);
begin
      if A1 = A2 then
```

[C++]

```
int A1 [2];
int A2 [2];

bool eq = true;

for (int i = 0; i < 2; ++i) {
      if (A1 [i] != A2 [i]) {
            eq = false;
      }
}

if (eq) {
```

```
[Java]
int [] A1 = new int [2];
int [] A2 = new int [2];
if (A1.equals (A2)) {
```

Вы можете присвоиться ко всем элементам массива на каждом языке по–разному. В Ada число элементов, чтобы присвоиться может быть определено, смотря на правую сторону, левую сторону или обе стороны присвоения. Когда границы известны на левой стороне, возможно использовать другие выражение, чтобы определить значение по умолчанию для всех неуказанных элементов матрицы. Поэтому, Вы можете записать:

```
declare
      type Arr_Type is array (Integer range <>) of Integer;
      A1 : Arr_Type := (1, 2, 3, 4, 5, 6, 7, 8, 9);
      A2 : Arr_Type (-2 .. 42) := (others => 0);
begin
      A1 := (1, 2, 3, others => 10);

      -- use a slice to assign A2 elements 11 .. 19 to 1
      A2 (11 .. 19) := (others => 1);
end;
```

8. Разнородные структуры данных

В Аде нет никакого различия между struct и классом, как это есть в C++. Все структуры гетерогенных данных – записи. Вот некоторые простые записи:

```
[Ada]
declare
      type R is record
            A, B : Integer;
            C : Float;
      end record;

      V : R;
begin
      V.A := 0;
end;

[C++]
struct R {
      int A, B;
      float C;
};

R V;
V.A = 0;
```

```
[Java]
class R {
      public int A, B;
      public float C;
}
R V = new R ();
V.A = 0;
```

Ada позволяет спецификацию значений по умолчанию для полей точно так же, как C++ и Java. Определенные значения могут принять форму упорядоченного списка значений, именованного списка значений или неполного списка, сопровождаемого `others => <>`, чтобы определить, что поля, не перечисленные, примут свои значения по умолчанию. Например:

```
type R is record
      A, B : Integer := 0;
      C : Float := 0.0;
end record;
V1 : R => (1, 2, 1.0);
V2 : R => (A => 1, B => 2, C => 1.0);
V3 : R => (C => 1.0, A => 1, B => 2);
V3 : R => (C => 1.0, others => <>);
```

9. Указатели

Указатели, ссылки и типы доступа отличаются значительно способами реализаций посредством языков, которые мы исследуем. В C++ указатели являются неотъемлемой частью основного понимания языка от манипулирования массивом до надлежащего объявления и использования параметров функции. Java идет идеологически на шаг вперед: все – есть ссылка, за исключением типов примитивов как скаляры. Проект Ada выбрал другое направление: он делает это за счет большего доступного функционала, не требуя явного использования указателей.

Мы будем продолжать этот раздел, объясняя различие между объектами, выделенными на стеке и объектах, выделенных на "куче" (heap), используя следующий пример:

```
[Ada]
declare
      type R is record
            A, B : Integer;
      end record;
      V1, V2 : R;
begin
      V1.A := 0;
      V2 := V1;
      V2.A := 1;
end;
```

[C++]

```
struct R {
      int A, B;
};

R V1, V2;
V1.A = 0;
V2 = V1;
V2.A = 1;
```

[Java]

```
public class R {
      public int A, B;
}

R V1, V2;
V1 = new R ();

V1.A = 0;
V2 = V1;
V2.A = 1;
```

Есть принципиальное различие между семантикой Ada, C++ и для Java. В Аде и C++, объекты выделены в стеке и непосредственно получают доступ. V1 и V2 – два различных объекта, и оператор присваивания копирует значение V1 в V2. В Java V1 и V2 – две 'ссылки' на объекты класса R. Обратите внимание на то, что, когда V1 и V2 объявлены, никакой фактический объект класса R все еще не существует в памяти. Он должен быть создан позже оператором `new` – средство выделения/размещения в памяти "кучи". После присвоения V2 = V1, в памяти есть только один объект R, так как присвоение – ссылочное присвоение, а не присвоение значения. В конце кода Java V1 и V2 – это две ссылки на один и тот же объект, и оператор V2. = 1 изменяет поле этого единственного объекта размещенного в памяти "кучи, в то время как в случае Ada и C++ : V1 и V2 – это два отличных объекта.

Чтобы получить подобное поведение в Аде, Вы можете использовать указатели. Это может быть сделано через Ada's способ ‘`access type`’. Например:

[Ada]

```
declare
      type R is record
            A, B : Integer;
      end record;
      type R_Access is access R;

      V1 : R_Access;
      V2 : R_Access;
begin
      V1 := new R;
      V1.A := 0;
      V2 := V1;
      V2.A := 1;
end;
```

```
[C++]
struct R {
      int A, B;
};
R * V1, * V2;
V1 = new R ();
V1->A = 0;
V2 = V1;
V2->A = 0;
```

Для тех, которые происходят из мира Java: в Ada нет никакого сборщика "мусора", таким образом, объекты, созданные оператором *new*, должны быть явно освобождены.

Разыменование в определенных ситуациях выполняется автоматически, например, когда ясно, что требуемый тип является разыменованным объектом, а не самим указателем, или при доступе к участникам записи через указатель. Чтобы явно разыменовать переменную доступа, добавьте *.all*. Эквивалент C++ синтаксиса *V1->A* , в Ada может быть записано или как V1.A, или V1.all.A.

Указатели на скалярные объекты в Ada и C++ похожи:

```
[Ada]
procedure Main is
      type A_Int is access Integer;
      Var : A_Int := new Integer;
Begin
      Var.all := 0;
end Main;

[C++]
int main (int argc, char *argv[]) {
      int * Var = new int;
      *Var = 0;
}
```

Инициализация может быть выполнена посредством добавления *'(value)*:

```
Var : A_Int := new Integer'(0);
```

Когда используются указатели Ada то ссылка производится на объекты размещенные в стеке, чтобы избежать этого объекты должны быть объявлены как **aliased**. Это заставит компилятор реализовать объект используя область памяти, вместо того, чтобы использовать регистры или устранить его полностью через оптимизацию. Тип доступа должен быть объявлен как **access all** (если объект, на который ссылаются, должен быть присвоен), или **access constant** (если объект, на который ссылаются – константа). Атрибут *'Access* работает аналогично как в C++

оператор **&**, чтобы получить указатель на объект, но с проверкой доступности объема “`scope accessibility`” , чтобы предотвращать ссылки на объекты, которые не соответствуют объему памяти для данного типа объекта на который они ссылаются. Например:

[Ada]

```ada
type A_Int is access all Integer;
Var : aliased Integer;
Ptr : A_Int := Var'Access;
```

[C++]

```cpp
int Var;
int * Ptr = &Var;
```

Чтобы освободить объекты от "кучи" в Ada, необходимо использовать подпрограмму освобождения, которая принимает определенный тип доступа. Есть универсальная процедура, которая должна быть настроена на необходимый тип объекта, чтобы соответствовать Вашим потребностям – она вызывает *`Ada.Unchecked_Deallocation`*. Чтобы создать Ваш специализированный *`deallocator`* (т.е. дистанцироваться от этого обобщения), Вы должны обеспечить тип объекта, а также тип доступа следующим образом:

[Ada]

```ada
with Ada.Unchecked_Deallocation;
procedure Main is
      type Integer_Access is access all Integer;
      procedure Free is new Ada.Unchecked_Deallocation (Integer,
      Integer_Access);
      My_Pointer : Integer_Access := new Integer;
begin
      Free (My_Pointer);
end Main;
```

[C++]

```cpp
int main (int argc, char *argv[]) {
      int * my_pointer = new int;
      delete my_pointer;
}
```

Глава шестая. ФУНКЦИИ И ПРОЦЕДУРЫ

1. Основная форма

Подпрограммы в C++ и Java всегда выражаются как функции (методы), которые могут или могут не возвратить значение. Язык программирования Ada явно дифференцируется между функциями и процедурами. Функции должны возвратить значение, и процедуры не должны. Язык программирования Ada использует более общий термин “подпрограмма”, чтобы относиться к функциям и к процедурам.

Параметры могут быть переданы в трех отличных режимах: в **`in`**, которое является значением по умолчанию, для входных параметров, значение которых обеспечено вызывающей стороной и не может быть изменено подпрограммой; в **`out`** , для выходных параметров, без начального значения, чтобы быть присвоенным подпрограммой и возвращенным к вызывающей стороне; в **`in out`** параметр с начальным значением, обеспеченным вызывающей стороной, которая может быть изменена подпрограммой и возвращена к вызывающей стороне (более или менее эквивалент непостоянной ссылки в C++). Язык программирования Ada также обеспечивает параметром **`access`**, в действительности явный индикатор передачи ссылкой.

В Ada программист определяет, как параметр будет использоваться, и в целом компилятор решает, как это будет передано (т.е. копией или ссылкой). (Есть некоторые исключения к “в целом”. Например, параметры скалярных типов всегда передаются копией для всех трех режимов.) C++ реализует передачу параметров таким образом, чтобы программист определил, как передать параметр. Язык Java вынуждает параметры типа примитива быть переданными копией, а все другие параметрами, которые будут переданы ссылкой. Поэтому не очевидны отображения между Адой и Java в 1:1. Пример – это попытка показать эти различия:

```
[Ada]
procedure Proc
     (Var1 : Integer;
     Var2 : out Integer;
     Var3 : in out Integer);
function Func (Var : Integer) return Integer;
```

```
procedure Proc
      (Var1 : Integer;
      Var2 : out Integer;
      Var3 : in out Integer)
is
begin
      Var2 := Func (Var1);
      Var3 := Var3 + 1;
end Proc;

function Func (Var : Integer) return Integer
is
begin
      return Var + 1;
end Func;
```

```
[C++]

void Proc
      (int Var1,
      int & Var2,
      int & Var3);

int Func (int Var);

void Proc
      (int Var1,
      int & Var2,
      int & Var3) {

      Var2 = Func (Var1);
      Var3 = Var3 + 1;
}

int Func (int Var) {
      return Var + 1;
}
```

```
[Java]

public class ProcData {
      public int Var2;
      public int Var3;

      public void Proc (int Var1) {
            Var2 = Func (Var1);
            Var3 = Var3 + 1;
      }
}

int Func (int Var) {
      return Var + 1;
}
```

Первые два объявления для Proc и Func – спецификации подпрограмм, которые обеспечиваются позже. Несмотря на то, что здесь это не обязательно, чтобы упростить читаемость программы, отдельная спецификация подпрограмм, все еще считается хорошим стилем программирования – это отдельно определять спецификации от реализации. В Ada и C++, не может использоваться функция, которая еще не была до

этого определена. Здесь в примере, Proc может вызвать Func, потому что его спецификация была объявлена. В Java разрешается иметь объявление подпрограммы позже.

Параметры в объявлениях подпрограммы Ada разделены точками с запятой, потому что запятые зарезервированы для перечисления многократных параметров того же типа. Синтаксис объявления параметра совпадает с синтаксисом объявления переменной, включая значения по умолчанию для параметров. Если нет никаких параметров, круглые скобки должны быть опущены полностью и в объявлении подпрограммы и в вызове подпрограммы.

2. Перегрузка

Различные подпрограммы могут совместно использовать то же имя подпрограммы. Это называется **“overloading”** – перезагрузка процедуры. Пока сигнатура подпрограммы (имя подпрограммы, типы параметра и типы возврата) отличаются, компилятор будет в состоянии разрешить проблему, – какую версию процедуры вызывать. Например:

```
function Value (Str : String) return Integer;
function Value (Str : String) return Float;
V : Integer := Value ("8");
```

Компилятор Ada знает, что присвоение на V требует *Integer*. Поэтому выбирает функцию *Value*, которая возвращает *Integer*, чтобы удовлетворить это требование.

Операторы в Ada можно рассматривать как функции также. Это позволяет Вам определять локальные операторы, которые переопределяют операторы, определенны во внешнем контексте, и предоставляют перегруженные операторы, которые выполняют операции на различных типах или сравнивают различные типы. Чтобы выразить оператора как функцию, включите его в кавычки:

```
[Ada]
function "=" (Left : Day; Right : Integer) return Boolean;

[C++]
bool operator = (Day Left, int Right);
```

3. Контракты подпрограмм

Вы можете выразить ожидаемые входы и выходы подпрограмм, определив контракты подпрограммы. Компилятор может тогда проверить на действительное выполнение условия, когда подпрограмму вызывают и/или может проверить, что имеет смысл возвращаемое значение. Язык

программирования Ada позволяет определять контракт: ожидаемых входов подпрограммы в виде **Pre** условия и выходы подпрограммы как **Post** условия. Это средство было введено в стандарт Ada 2012. Вот похожий пример:

```
function Divide (Left, Right : Float) return Float
     with Pre => Right /= 0.0,
          Post => Divide'Result * Right < Left + 0.0001
               and then Divide'Result * Right > Left - 0.0001;
```

Вышеупомянутый пример добавляет **Pre** условие, заявляя, что *`Right`* не может быть равно 0.0. В то время как стандарт IEEE разрешает деление на ноль чисел с плавающей запятой, Вы, возможно, решили, что использование результата могло все еще привести к проблемам в особом применении. Составление "контракта" помогает обнаружить это как можно раньше. Эта декларация также обеспечивает **Post** условие на результат подпрограммы.

Постусловия могут также быть выражены относительно значения ввода:

```
procedure Increment (V : in out Integer)
     with Pre => V < Integer'Last,
          Post => V = V'Old + 1;
```

`V'Old` в постусловии представляет значение, которое V имело прежде, чем ввести *`Increment`*.

Глава седьмая. ПАКЕТЫ

1. Объявление Protection

Пакет – основная единица модульности языка Ada, как класс для Java и как пара – заголовок и реализация для C++. Пакет Ada содержит три части, которые, для GNAT, разделены на два файла: файлы .ads содержат общедоступные и частные спецификации Ada, и .adb файлы содержат реализацию или тела пакета Ada.

Java не обеспечивает средств чисто разделить спецификацию методов от их реализации: они все появляются в том же файле. Вы можете использовать интерфейсы, чтобы эмулировать наличие отдельных спецификаций, но это требует использования методов ООП, которое не всегда практично.

Языки программирования Ada и C++ действительно предлагают разделение между спецификациями и реализациями, независимого от ООП.

```
package Package_Name is
      -- public specifications
private
      -- private specifications
end Package_Name;

package body Package_Name is
      -- implementation
end Package_Name;
```

Частные (`Private`) типы полезны для предотвращения пользователей типов пакета от зависимости деталей реализации типов. Ключевое слово `Private` разделяет спецификацию пакета на “общедоступные” и “частные” части. Это несколько походит на разделение в C++ конструкции класса на различные разделы с различными свойствами видимости. В Java инкапсуляция должна быть сделана поле за полем, но в Ada может быть скрыто все определение типа. Например:

```
package Types is
      type Type_1 is private;
      type Type_2 is private;
      type Type_3 is private;
      procedure P (X : Type_1);
      ...
private
      procedure Q (Y : Type_1);
      type Type_1 is new Integer range 1 .. 1000;
      type Type_2 is array (Integer range 1 .. 1000) of Integer;
      type Type_3 is record
            A, B : Integer;
      end record;
end Types;
```

Подпрограммы, объявленные выше `private` разделителя (такие как *P*), будут видимы пользователю пакета, а те, что ниже (такие как *Q*) не будут. У тела реализации пакета языка Ada, есть доступ к обеим частям.

2. Иерархия пакетов

Пакеты Ada могут быть организованы в иерархии. Дочерний модуль может быть объявлен следующим образом:

```
-- root-child.ads
package Root.Child is
      -- package spec goes here
end Root.Child;

-- root-child.adb
package body Root.Child is
      -- package body goes here
end Root.Child;
```

Здесь, *Root.Child* – дочерний пакет *Root*. У Общедоступной части *Root.Child* есть доступ к общедоступной части *Root*. У приватной части элемента *Child* есть доступ к приватной части *Root*, это является одним из основных преимуществ дочерних пакетов. Однако между этими двумя телами нет никакого отношения видимости. Один распространенный способ использовать эту возможность состоит в том, чтобы определить подсистемы вокруг иерархической схемы именования.

3. Использование сущностей из пакетов

Сущности, объявленные в видимой части спецификации пакета, могут быть сделаны доступными с использованием пункта **`with`** , который ссылается на пакет. Пункт **with** подобен директиве C++ *#include*. Видимость неявна в Java: Вы можете всегда получать доступ ко всем классам, расположенным в Вашем *CLASSPATH* (ПУТИ К КЛАССУ). После пункта **`with`**, объекты к которым обращаются, должны быть снабжены префиксом – именем их пакета, как пространство имен C++ или пакет Java. Этот префикс может быть опущен, если пункт **`use`** используется, подобен используемому пространству имен в C++ **`using namespace`** (пространство имен) или **`import`** (импорт) Java.

[Ada]

```ada
-- pck.ads
package Pck is
      My_Glob : Integer;
end Pck;
```

```ada
-- main.adb
with Pck;
procedure Main is
begin
      Pck.My_Glob := 0;
end Main;
```

[C++]

```cpp
// pck.h
namespace pck {
      extern int myGlob;
}
```

```cpp
// pck.cpp
namespace pck {
      int myGlob;
}
```

```cpp
// main.cpp
#include "pck.h"
int main (int argc, char ** argv) {
      pck::myGlob = 0;
}
```

[Java]

```java
// Globals.java
package pck;
public class Globals {
      public static int myGlob;
}
```

```java
// Main.java
public class Main {
      public static void main (String [] argv) {
            pck.Globals.myGlob = 0;
      }
}
```

Глава восьмая. КЛАССЫ И ОБЪЕКТНО–ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

1. Примитивные подпрограммы

Примитивные подпрограммы в Ada – просто подпрограммы, которые имеют право на наследование / деривацию (inheritance / derivation). Они – эквивалент функций членства C++ и методов экземпляра Java. В то время как в C++ и Java эти подпрограммы расположены во вложенном контексте типа, в Ada они просто объявлены в том же контексте как тип. Нет никакой синтаксической индикации, что подпрограмма – примитив типа.

Способом определить, является ли P примитивом типа T, состоит в том, что если (1) примитив типа объявлен в том же объеме как T, и (2) примитив типа содержит, по крайней мере, один параметр типа T или возвращает результат типа T.

В C++ или Java, ссылка self (ссылка на себя) неявно объявлена. Есть возможность явно объявить ссылку self, и это требуется в определенных ситуациях, но обычно это опущено. В Ada ссылка self, названная ‘параметром управления’ (‘controlling parameter’), должна быть явно определена в списке параметров подпрограммы. В то время как это может быть любой параметр в профиле с любым именем, мы будем фокусироваться на типичном случае, где первый параметр используется в качестве 'self' параметра. Перечисление параметра управления сначала также включает использование префиксной нотации ООП, которая удобна.

Класс в C++ или Java соответствует теговому типу в Ada. Вот пример объявления Ada теговым типом с двумя параметрами и некоторых примитивов с диспетчеризацией и без диспетчеризации с эквивалентными примерами в C++ и Java:

[Ada]

```
type T is tagged record
     V, W : Integer;
end record;
type T_Access is access all T;
function F (V : T) return Integer;
procedure P1 (V : access T);
procedure P2 (V : T_Access);
```

[C++]

```
class T {

public:

int V, W;

      int F (void);

void P1 (void);

};

void P2 (T * v);
```

```
[Java]

public class T {

public int V, W;

public int F (void) {};

public void P1 (void) {};

public static void P2 (T v) {};

      }
```

Обратите внимание на то, что P2 не примитив T – у этого P2 нет параметров типа T. Его параметр имеет тип T_Access, который является другим типом.

После объявления примитивы можно вызвать также как любую подпрограмму с определением каждого необходимого параметра, или вызвать, используя префиксную нотацию. Например:

```
[Ada]

declare

      V : T;

begin

      V.P1;

end;
```

```
[C++]

{

      T v;

      v.P1 ();

}
```

```
[Java]

{

      T v = new T ();

      v.P1 ();

}
```

2. Наследование (Derivation) и динамическая диспетчеризация (Dispatch)

Несмотря на синтаксические различия, наследование в Ada подобна деривации (наследованию) в C++ или Java. Например, вот иерархия типа, где дочерний класс переопределяет метод и добавляет новый метод:

```
[Ada]

type Root is tagged record
      F1 : Integer;
end record;

procedure Method_1 (Self : Root);

type Child is new Root with record
      F2 : Integer;
end Child;

overriding
procedure Method_1 (Self : Child);

procedure Method_2 (Self : Child);

[C++]

class Root {
      public:
            int f1;
      virtual void method1 ();
};

class Child : public Root {
      public:
            int f2;
            virtual void method1 ();
            virtual void method2 ();
};

[Java]

public class Root {
      public int f1;
      public void method1 ();
}
```

```
public class Child extends Root {
      public int f2;
      @Override
      public void method1 ();
      public void method2 ();
}
```

Как и в Java, примитивы Ada на теговых типах всегда подвергаются диспетчеризации; нет никакой потребности отметить их как **virtual** (виртуальный). Также как Java, есть дополнительное **overriding** (переопределение) ключевого слова, чтобы гарантировать, что метод действительно переопределяет что–то от родительского типа.

В отличие от многих других языков ООП, Ada дифференцируется между ссылкой на определенный теговый тип и ссылкой на всю теговую иерархию типа. В то время как *Root* (Корень) используется, чтобы означать определенный тип, *Root'Class* – тип всего класса – относится к тому типу или к любому из его потомков. Метод, используя параметр такого типа не может быть переопределен и должен быть передан параметр, тип которого имеет любого из потомков *Root* (включая сам *Root*).

Затем, мы будем смотреть на то, как каждый язык находит, что надлежащий метод вызывает в иерархии классов ОО; т.е. их правила диспетчеризации. В Java вызовы к нечастным методам экземпляра всегда диспетчеризируют. Единственный случай, где статический выбор метода экземпляра возможен, при вызове от метода до **super** (супер) версии.

В C++, по умолчанию, вызовы к виртуальным методам всегда диспетчеризируют. Одна частая ошибка состоит в том, чтобы использовать параметр копией, надеясь, что диспетчеризация достигнет реального объекта. Например:

```
void proc (Root p) {
      p.method1 ();
}
Root * v = new Child ();
proc (*v);
```

В вышеупомянутом коде, *p.method1 ()* не диспетчеризирует. Вызов к *proc* делает копию *Root* части *v*, таким образом, внутри *proc*, **p.method1* ()* относится к **method1* ()* root объекта. Намеченное поведение может быть определено при помощи ссылки вместо копии:

```
void proc (Root & p) {
      p.method1 ();
}
Root * v = new Child ();
proc (*v);
```

В Ada теговые типы всегда передаются ссылкой, но диспетчеризация только происходит на типах всего класса. Следующий код Ada эквивалентен последнему примеру C++:

```
declare
      procedure Proc (P : Root'Class) is
      begin
            P.Method_1;
      end;

      type Root_Access is access all Root'Class;
      V : Root_Access := new Child;
begin
      Proc (V.all);
end;
```

Диспетчеризация из примитивов может стать хитрой. Давайте рассмотрим вызов к *Method_1* в реализации *Method_2*. Первая реализация, которая могла бы прийти на ум:

```
procedure Method_2 (P : Root) is
begin
      P.Method_1;
end;
```

Однако Method_2 вызывают с параметром, который имеет тип определенный как *Root* . Более точно это – определенное представление дочернего элемента. Так как, этот вызов не диспетчеризирует; то он всегда будет вызывать *Method_1* Корня (*Root*), даже если в объект передавался, дочерний элемент *Root*. Чтобы фиксировать это, необходимо преобразование представления:

```
[Ada]
procedure Method_2 (P : Root) is
begin
      Root'Class (P).Method_1;
end;

[C++]
void proc (Root & p) {
      p.Root::method1 ();
}
```

3. Конструкторы и деструкторы (Constructors and Destructors)

У Ada нет конструкторов и деструкторов реализованных тем же способом как C++ и Java, но в Ada есть аналогичная функциональность в форме инициализации по умолчанию и завершения.

Инициализация по умолчанию может быть определена для компонента record и произойдет, если переменной типа record (записи) не присваивается значение в время инициализации. Например:

```
type T is tagged record
      F : Integer := Compute_Default_F;
end record;

function Compute_Default_F return Integer is
begin
      Put_Line ("Compute");
      return 0;
end Compute_Default_F;

V1 : T;
V2 : T := (F => 0);
```

В объявлении *V1 T.F* получает значение, вычисленное подпрограммой *Compute_Default_F*. Это – часть инициализации по умолчанию. *V2* инициализирован явно (вручную) и таким образом не будет использовать инициализацию по умолчанию.

Для дополнительной выразительной мощи языка, Ada обеспечивает тип по имени *Ada.Finalization.Controlled*, который позволяет Вам получить свой собственный тип. Затем переопределяя процедуру *Initialize* , Вы можете создать конструктора для типа:

```
type T is new Ada.Finalization.Controlled with record
      F : Integer;
end record;

procedure Initialize (Self : in out T) is
begin
      Put_Line ("Compute");
      Self.F := 0;
end Initialize;

V1 : T;
V2 : T := (V => 0);
```

Снова, эта подпрограмма инициализации по умолчанию только требуется *V1*; *V2* инициализирован явным образом – вручную. Кроме того, в отличие от C++ или конструктора Java, *Initialize* – нормальная подпрограмма и не выполняет дополнительной инициализации, такой как вызов подпрограмм инициализации родителя.

При получении из *Controlled* также возможно переопределить подпрограмму *Finalize*, которая аналогична роли деструктора и вызывается для объектного завершения. Как *Initialize*, это – обычная подпрограмма. Не ожидайте, что любые другие программы финализации будут автоматически вызваны для Вас.

Управляемые типы также обеспечивают функциональность, которая по существу позволяет переопределять значение работы присвоения, и полезна для определения типов, которые управляют их собственным

восстановлением хранения (например, реализовывая стратегию восстановления подсчета ссылок).

4. Инкапсулирование

В то время как инкапсулирование сделано на уровне класса для C++ и Java, инкапсуляция Ada происходит на уровне пакета и предназначается для всех объектов языка, в противоположность только методам и атрибутам для C++ и Java. Например:

```
[Ada]
package Pck is
      type T is tagged private;
      procedure Method1 (V : T);
private
      type T is tagged record
            F1, F2 : Integer;
      end record;
      procedure Method2 (V : T);
end Pck;

[C++]
class T {
      public:
            virtual void method1 ();
      protected:
            int f1, f2;
            virtual void method2 ();
};

[Java]
public class T {
      public void method1 ();
      protected int f1, f2;
      protected void method2 ();
}
```

Здесь пример демонстрирует, как отобразить эти понятия между языками, посредством использование кода C++ и Java защищенных частей кода и использование приватной части кода Ada. Действительно, у приватной части кода дочернего пакета Ada была бы видимость приватной части кода ее родителей, имитируя понятие защищенных. Только объекты, объявленные в **package body**, полностью изолированы от доступа.

5. Абстрактные типы и интерфейсы

Ada, C++ и Java всеми языками предложена подобная функциональность с точки зрения абстрактных классов или чистые виртуальные классы. В языках Ada и Java необходимо явно определить, абстрактны ли теговый тип или класс, тогда как в C++ присутствие чистой

виртуальной функции неявно делает класс абстрактным базовым классом. Например:

[Ada]

```
package P is
      type T is abstract tagged private;
      procedure Method (Self : T) is abstract;
private
      type T is abstract tagged record
            F1, F2 : Integer;
      end record;
end P;
```

[C++]

```
class T {
      public:
            virtual void method () = 0;
      protected:
            int f1, f2;
};
```

[Java]

```
public abstract class T {
      public abstract void method1 ();
      protected int f1, f2;
};
```

Все абстрактные методы должны быть реализованы при реализации конкретного типа на основе абстрактного типа.

Ada не предлагает множественное наследование тем же путем, как делает C++, но Ada действительно поддерживает подобное Java понятие интерфейсов. Интерфейс походит на чистый виртуальный класс C++ без атрибутов и только абстрактных элементов. В то время как теговый тип Ada может наследоваться самое большее от одного тегового типа, он может реализовать многократные интерфейсы. Например:

[Ada]

```
type Root is tagged record
      F1 : Integer;
end record;
procedure M1 (Self : Root);

type I1 is interface;
procedure M2 (Self : I1) is abstract;

type I2 is interface;
procedure M3 (Self : I2) is abstract;

type Child is new Root and I1 and I2 with record
      F2 : Integer;
end record;

-- M1 implicitly inherited by Child
procedure M2 (Self : Child);
```

```
procedure M3 (Self : Child);
```

```
[C++]
class Root {
      public:
            virtual void M1();
            int f1;
};
class I1 {
      public:
            virtual void M2 () = 0;
};
class I2 {
      public
            virtual void M3 () = 0;
};
class Child : public Root, I1, I2 {
      public:
            int f2;
            virtual void M2 ();
            virtual void M3 ();
};
```

```
[Java]
public class Root {
      public void M1();
      public int f1;
}
public interface I1 {
      public void M2 () = 0;
}
public class I2 {
      public void M3 () = 0;
}
public class Child extends Root implements I1, I2 {
      public int f2;
      public void M2 ();
      public void M3 ();
}
```

6. Инвариант

Любой частный тип в Ada может быть связан с контрактом Type_Invariant. Инвариант – свойство типа, который должен всегда быть истиной после возврата из любой из его примитивных подпрограмм. (Инвариант не мог бы сохраняться во время выполнения примитивных подпрограмм, но будет истиной после возврата.) Разобраться нам в этом позволит следующий пример:

```
package Int_List_Pkg is
      type Int_List (Max_Length : Natural) is private
            with Type_Invariant => Is_Sorted (Int_List);
```

```
      function Is_Sorted (List : Int_List) return Boolean;
      type Int_Array is array (Positive range <>) of Integer;
      function To_Int_List (Ints : Int_Array) return Int_List;
      function To_Int_Array (List : Int_List) return Int_Array;
      function "&" (Left, Right : Int_List) return Int_List;
      ... -- Other subprograms
private
      type Int_List (Max_Length : Natural) is record
            Length : Natural;
            Data : Int_Array (1..Max_Length);
      end record;
      function Is_Sorted (List : Int_List) return Boolean is
            (for all I in List.Data'First .. List.Length-1 =>
            List.Data (I) <= List.Data (I+1));
end Int_List_Pkg;
package body Int_List_Pkg is
      procedure Sort (Ints : in out Int_Array) is
      begin
            ... Your favorite sorting algorithm
      end Sort;
      function To_Int_List (Ints : Int_Array) return Int_List is
            List : Int_List :=
                  (Max_Length => Ints'Length,
                  Length => Ints'Length,
                  Data => Ints);
      begin
            Sort (List.Data);
            return List;
      end To_Int_List;
      function To_Int_Array (List : Int_List) return Int_Array is
      begin
            return List.Data;
      end To_Int_Array;
      function "&" (Left, Right : Int_List) return Int_List is
            Ints : Int_Array := Left.Data & Right.Data;
      begin
            Sort (Ints);
            return To_Int_List (Ints);
      end "&";
... -- Other subprograms
end Int_List_Pkg;
```

Функция Is_Sorted проверяет, что тип остается непротиворечивым. Ее вызовут при выходе из каждого примитива, определенных выше. Допустимо, если условия инварианта не соблюдают во время выполнения примитива. В To_Int_List, например, если исходный массив не находится в сортированном порядке, инвариант не будет удовлетворен при “начинании”, но это будет проверено в конце.

Глава девятая. ОБОБЩЕНИЯ (GENERIC formalism)

Ada, C++ и Java у всех есть поддержка обобщений или шаблонов, но на различных наборах объектов языка. Шаблон C++ может быть применен к классу или функции. Так может и обобщение Java. Обобщение в Ada может быть или пакетом или подпрограммой.

1. Обобщение для подпрограммы

Функция, которая подобна через все три языка, является подпрограммой. Например подпрограмма обмена (swap) двух объектов:

[Ada]

```
generic
      type A_Type is private;
procedure Swap (Left, Right : in out A_Type) is
      Temp : A_Type := Left;
begin
      Left := Right;
      Right := Temp;
end Swap;
```

[C++]

```
template <class AType>
AType swap (AType & left, AType & right) {
      AType temp = left;
      left = right;
      right = temp;
}
```

[Java]

```
public <AType> void swap (AType left, AType right) {
      AType temp = left;
      left = right;
      right = temp;
}
```

И примеры использования этих обобщений:

[Ada]

```
declare
      type R is record
            F1, F2 : Integer;
      end record;

      procedure Swap_R is new Swap (R);
      A, B : R;
begin
      ...
      Swap_R (A, B);
```

```
end;
```

```
[C++]
class R {
      public:
            int f1, f2;
};

R a, b;

...
swap (a, b);
```

```
[Java]
public class R {
      public int f1, f2;
}

R a = new R(), b = new R();

...
swap (a, b);
```

Шаблон C++ и обобщение Java оба становятся применимыми после определения. Обобщение Ada требует явной инсталляции (настройки), используя локальное имя и параметры обобщения.

2. Обобщенные пакеты

Затем, мы собираемся создать универсальный модуль, содержащий данные и подпрограммы. В Java или C++, это сделано через класс, в то время как в Ada, это – ‘обобщенный пакет’. Модель Ada и C++ существенно отличается от модели Java. Действительно, после инсталляции (настройки), Ada и C++ обобщенные данные дублированы; т.е. если они будут содержать глобальные переменные (Ada) или статические атрибуты (C++), то у каждого экземпляра будет своя собственная копия переменной, должным образом введенной и независимой от других. В Java обобщения – только механизм, чтобы указать компилятору, как делать проверки непротиворечивости, но все экземпляры фактически совместно используют те же данные, где обобщенные параметры заменены java.lang.Object. Давайте посмотрим на следующий пример:

```
[Ada]
generic
      type T is private;
package Gen is
      type C is tagged record
            V : T;
      end record;

      G : Integer;
end Gen;
```

```
[C++]
```

```
template <class T>
class C{
      public:
            T v;
      static int G;
};
```

[Java]

```
public class C <T> {
      public T v;
      public static int G;
}
```

Во всех трех случаях есть переменная экземпляра (v) и статическая переменная (G). Давайте теперь посмотрим на поведение (и синтаксис) этих трех инициализаций (настроек):

[Ada]

```
declare
      package I1 is new Gen (Integer);
      package I2 is new Gen (Integer);
      subtype Str10 is String (1..10);
      package I3 is new Gen (Str10);
begin
      I1.G := 0;
      I2.G := 1;
      I3.G := 2;
end;
```

[C++]

```
C <int>::G = 0;
C <int>::G = 1;
C <char *>::G = 2;
```

[Java]

```
C.G = 0;
C.G = 1;
C.G = 2;
```

В случае Java мы получаем доступ к обобщенному объекту непосредственно, не используя параметрический тип. Это вызвано тем, что есть действительно только один экземпляр *C* с каждым экземпляром, совместно использующим ту же глобальную переменную *G*. В C++ экземпляры неявны, таким образом, нет возможности создать два различных экземпляра с теми же параметрами. Первые два присвоения управляют той же глобальной переменной, в то время как третий управляет различным экземпляром. В случае Ada эти три экземпляра явно создают, называют и ссылаются индивидуально.

3. Параметры для обобщений

Ada предлагает большое разнообразие универсальных параметров, которое трудно перевести на другие языки. Параметры использовали во

время инициализации – и как следствие те, на которых обобщенный модуль может полагаться – могут быть переменные, типы или подпрограммы с определенными свойствами. Например, следующее обобщение обеспечивает алгоритм сортировки для любого вида массива:

```
generic
      type Component is private;
      type Index is (<>);
      with function "<" (Left, Right : Component) return Boolean;
      type Array_Type is array (Index range <>) of Component;
procedure Sort (A : in out Array_Type);
```

Вышеупомянутое объявление утверждает, что нам нужен тип *Component* (Компонент), дискретный тип *Index* (Индекс), подпрограмма сравнения (“<”), и определение массива (*Array_Type*). Учитывая их, возможно, записать алгоритм, который может сортировать любой Array_Type. Отметьте использование с зарезервированным словом (создается новый тип) перед именем функции, чтобы дифференцироваться между обобщенным параметром и начало обобщенной подпрограммы (чтобы различать подпрограммы, с обобщенным типом и инициализированным типом).

Вот неисчерпывающий обзор вида ограничений, которые могут быть помещены на типы:

```
type T is private; -- T is a constrained type, such as Integer
type T (<>) is private; -- T can be an unconstrained type, such as
                        --String
type T is tagged private; -- T is a tagged type
type T is new T2 with private; -- T is an extension of T2
type T is (<>); -- T is a discrete type
type T is range <>; -- T is an integer type
type T is digits <>; -- T is a floating point type
type T is access T2; -- T is an access type, T2 is its designated type
```

Глава десятая. ИСКЛЮЧЕНИЯ (EXCEPTIONS formalism)

Исключения – механизм для контакта со случаями во время выполнения, которые редки, которые обычно соответствуют ошибкам (такой, как неправильно сформировано входные данные), и чье возникновение вызывает безусловную передачу управления.

1. Стандартные Исключения

По сравнению с Java и C++, понятие исключения Ada очень просто. Исключение в Ada – объект, “тип” которого – исключение, в противоположность классам в Java или любому типу в C++. Единственной частью пользовательских данных, которые могут быть связаны за исключением Ada, является Строка. В основном исключение в Ada может быть повышено, и оно может быть обработано; информация, связанная с возникновением исключения, может быть опрошена обработчиком.

Ada делает интенсивное использование исключений специально для отказов проверки непротиворечивости данных во время выполнения. Они включают, но не ограничены, проверив по диапазонам типа и границам массива, нулевым указателям, различному виду свойств параллелизма и функциям, не возвратив значение. Например, следующая часть кода повысит исключение Constraint_Error:

```
procedure P is
     V : Positive;
begin
     V := -1;
end P;
```

В выше приведенном коде мы пытаемся присвоить отрицательную величину переменной, в то время как ей объявляют тип позволяющий присваивать ей только положительные значения. Проверка диапазона имеет место во время выполнения присвоения, и отказ вызывает исключение Constraint_Error в этой точке. (Обратите внимание на то, что компилятор может дать предупреждение, что значение вне диапазона и что ошибка появится как исключение на этапе выполнения.) До тех пор нет никакого локального обработчика, исключение будет распространено к вызывающей стороне; если P будет основной процедурой, то программа будет завершена.

Java и C++ создается специальный try блок исключения и catch блок исключения в том месте где производится попытке выполнить код. Весь код Ada уже неявно в блоках попытки, и исключения передается на более высокий уровень или обрабатывается.

[Ada]

```
begin
      Some_Call;
exception
      when Exception_1 =>
            Put_Line ("Error 1");
      when Exception_2 =>
            Put_Line ("Error 2");
      when others =>
            Put_Line ("Unknown error");
end;
```

[C++]

```
try {
      someCall ();
} catch (Exception1) {
      cout << "Error 1" << endl;
} catch (Exception2) {
      cout << "Error 2" << endl;
} catch (...) {
      cout << "Unknown error" << endl;
}
```

[Java]

```
try {
      someCall ();
} catch (Exception1 e1) {
      System.out.println ("Error 1");
} catch (Exception2 e2) {
      System.out.println ("Error 2");
} catch (Throwable e3) {
      System.out.println ("Unknown error");
}
```

Повышение (Raising) и выдача исключений, одновременно с обработкой исключений допустимо на всех трех языках.

2. Пользовательские исключения

Пользовательские декларации исключений напоминают декларации объектов, и они могут быть созданы в Аде, используя ключевое слово исключение:

```
My_Exception : exception;
```

Ваши исключения тогда можно поднимать с помощью ***raise*** заявления, необязательно сопровождается сообщения с нижеследующим зарезервированным словом ***with***:

[Ada]

```
raise My_Exception with "Some message";
```

[C++]

```
throw My_Exception ("Some message");
```

```
[Java]
throw new My_Exception ("Some message");
```

Исключения определенные в языке Ada также могут быть подняты таким же образом:

```
raise Constraint_Error;
```

Глава одиннадцатая. ПАРАЛЛЕЛИЗМ (CONCURRENCY formalism)

1. Задачи

Java и Ada оба предоставляют поддержку для параллелизма на уровне языка. Язык C++ добавил средство параллелизма в своей новой версии, C++ 11, но мы предполагаем, что большинство программистов на C++ еще не знакомо с этими новыми функциями. Мы, таким образом, обеспечиваем следующий дополнительный API для C++, который подобен классу Потока Java (Java Thread class):

```
class Thread {
      public:
            virtual void run (); // code to execute
            void start (); // starts a thread and then call run ()
            void join (); // waits until the thread is finished
};
```

Каждый из следующих примеров выведет на экран эти 26 букв алфавита дважды, используя два параллельных потока/задачи. Так как нет никакой синхронизации между двумя потоками управления ни в одном из примеров, вывод может быть прерывистый для каждого из потока (вкраплен вывод другого потока).

```
[Ada]
procedure Main is -- implicitly called by the environment task
      task My_Task;

      task body My_Task is
      begin
            for I in 'A' .. 'Z' loop
                  Put_Line (I);
            end loop;
      end My_Task;
begin
      for I in 'A' .. 'Z' loop
            Put_Line (I);
      end loop;
end Main;

[C++]
class MyThread : public Thread {
      public:

      void run () {
            for (char i = 'A'; i <= 'Z'; ++i) {
                  cout << i << endl;
            }
      }
```

```
};
int main (int argc, char ** argv) {
      MyThread myTask;
      myTask.start ();
      for (char i = 'A'; i <= 'Z'; ++i) {
            cout << i << endl;
      }
      myTask.join ();
      return 0;
}
```

[Java]

```
public class Main {
      static class MyThread extends Thread {
            public void run () {
                  for (char i = 'A'; i <= 'Z'; ++i) {
                        System.out.println (i);
                  }
            }
      }
      public static void main (String args) {
            MyThread myTask = new MyThread ();
            myTask.start ();
            for (char i = 'A'; i <= 'Z'; ++i) {
                  System.out.println (i);
            }
            myTask.join ();
      }
}
```

Любое число задач Ada может быть объявлено в любой декларативной области. Объявление задачи очень похоже на объявление пакета или процедуру. Они все запускают автоматически, когда управление достигает ***begin***. Блок не выйдет, пока все последовательности операторов, определенных в том объеме, включая тех в задачах, не были завершены.

Тип задачи – обобщение объекта задачи; у каждого объекта типа задачи есть одно и то же поведение. Заявленный объект типа задачи запущен в объеме, где это объявлено, и управление не оставляет тот объем, пока задача не завершилась.

Тип задачи Ada несколько походит на подкласс Потока Java (Java Thread subclass), но в Java всегда динамично выделяются экземпляры такого подкласса. В Ada экземпляр типа задачи может или быть объявлен или динамично выделен.

Типы задачи могут быть параметризованные; параметр служит той же цели как параметр конструктору в Java. Следующий пример создает 10 задач, каждая из которых выводит на экран подмножество алфавита, содержавшего между параметром и Символом ‘Z’. Как с более ранним примером, с тех пор нет никакой синхронизации среди задач, вывод может быть вкраплен в зависимости от алгоритма планирования задач реализации.

[Ada]

```ada
task type My_Task (First : Character);

task body My_Task (First : Character) is
begin
      for I in First .. 'Z' loop
            Put_Line (I);
      end loop;
end My_Task;

procedure Main is
      Tab : array (0 .. 9) of My_Task ('G');
begin
      null;
end Main;
```

[C++]

```cpp
class MyThread : public Thread {
      public:

      char first;

      void run () {
            for (char i = first; i <= 'Z'; ++i) {
                  cout << i << endl;
            }
      }
};

int main (int argc, char ** argv) {
      MyThread tab [10];

      for (int i = 0; i < 9; ++i) {
            tab [i].first = 'G';
            tab [i].start ();
      }

      for (int i = 0; i < 9; ++i) {
            tab [i].join ();
      }

      return 0;
}
```

[Java]

```java
public class MyThread extends Thread {
      public char first;

      public MyThread (char first){
            this.first = first;
      }

      public void run () {
            for (char i = first; i <= 'Z'; ++i) {
                  cout << i << endl;
            }
      }
}

public class Main {
      public static void main (String args) {
            MyThread [] tab = new MyThread [10];
```

```
            for (int i = 0; i < 9; ++i) {
                  tab [i] = new MyThread ('G');
                  tab [i].start ();
            }
            for (int i = 0; i < 9; ++i) {
                  tab [i].join ();
            }
      }
}
```

В Ada задача может быть выделена на "куче" в противоположность стеку. Задача тогда запустится, как только она была выделена и завершается, когда его работа завершена. Эта модель – вероятно, та, которая наибольшей степени соответствует Java:

```
[Ada]
type Ptr_Task is access My_Task;
procedure Main is
      T : Ptr_Task;
begin
      T := new My_Task ('G');
end Main;

[C++]
int main (int argc, char ** argv) {
      MyThread * t = new MyThread ();
      t->first = 'G';
      t->start ();
      return 0;
}

[Java]
public class Main {
      public static void main (String args) {
            MyThread t = new MyThread ('G');
            t.start ();
      }
}
```

2. Рандеву

Рандеву – синхронизация между двумя задачами, позволяя им обмениваться данными и координатным выполнением. Средство рандеву Ada не может быть смоделировано с C++ или Java без сложного машинного оборудования. Поэтому, этот раздел просто покажет примеры, записанные в Ada.

Давайте рассмотрим следующий пример:

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Main is
      task After is
            entry Go;
```

```
      end After ;

      task body After is
      begin
            accept Go;
            Put_Line ("After");
      end After;

begin
      Put_Line ("Before");
      After.Go;
end;
```

Запись (*entry*) ***Go***, объявленная в ***After***, является внешним интерфейсом к задаче. В теле задачи оператор "принятия" (**accept**) вызывает ожидание задачи запроса к записи. В примере, это конкретное определение пары "запись" (*entry*) и "принятие" (*accept*) не делает больше, чем заставить задачу ожидать до момента вызова ***After.Go*** в ***Main***. Так, даже при том, что эти две задачи запускаются одновременно и выполняются независимо, они могут скоординировать свое выполнение через ***Go***. Затем, после рандеву, они обе продолжают выполнение независимо.

Пара запись/принимать (entry/accept) может брать/передавать (take/pass) параметры, и оператор принятия может содержать последовательность операторов; в то время как эти операторы выполняются, вызывающая сторона блокирована.

Давайте посмотрим на более амбициозный пример. Рандеву ниже принимает параметры и выполняет некоторый код:

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

procedure Main is

      task After is
            entry Go (Text : String);
      end After ;

      task body After is
      begin
            accept Go (Text : String) do
                  Put_Line ("After: " & Text);
            end Go;
      end After;

begin
      Put_Line ("Before");
      After.Go ("Main");;
end;
```

В вышеупомянутом примере Put_Line помещен в оператор "принятия" (**accept**). Вот возможная трассировка выполнения, если предположить что однопроцессорный компьютер:

1. При **begin** процедуры *Main* запущена задача *After*, и основная процедура приостановлена.

2. *After* достигает оператора **accept** и приостанавливается, так как нет никакого незаконченного запроса к записи (**entry**) *Go*.

3. Основная процедура пробуждена и выполняет вызов Put_Line, выводя на экран строку “ Before”.

4. Основная процедура вызывает запись *Go*. Так как *After* приостановлена на его операторе **accept** для данной **entry**, вызов выполняется успешно.

5. Основная процедура приостановлена, и задача *After*, пробуждена, чтобы выполнить тело оператора **accept**. Фактический “ Main ” параметр передан к оператору **accept**, и вызов Put_Line выполняется. В результате строка “ After: Main ” выведена на экран.

6. Когда оператор **accept** завершен, обе и *After* задача и основная процедура будут готовы работать. Предположим, что процедуре Main дают процессор. Это достигает своего конца, но локальная задача *After* еще не завершилась. Основная процедура приостановлена.

7. После продолжения задачи *After,* она завершается, так как достигает своего оператора **end**. Основная процедура возобновляет свое выполнение, и она также может завершиться, так как ее зависимая задача завершилась.

Вышеупомянутое описание – концептуальная модель; на практике реализация может выполнить различную оптимизацию, чтобы избежать ненужных контекстных переключений.

3. Выборочное рандеву

Оператор **accept** отдельно может только ожидать единственного события (вызов) за один раз. Оператор **select** позволяет задаче прислушаться к многократным событиям одновременно, и затем иметь дело с первым событием, которое произошло. Эта функция проиллюстрирована задачей ниже, которая поддерживает целочисленное значение, которое изменяется посредством взаимодействия с другими задачами. Другие задачи вызывают *Increment*, *Decrement*, и *Get*:

```
task Counter is
      entry Get (Result : out Integer);
      entry Increment;
      entry Decrement;
end Counter;

task body Counter is
      Value : Integer := 0;
begin
      loop
            select
                  accept Increment do
                        Value := Value + 1;
                  end Increment;
            or
                  accept Decrement do
                        Value := Value - 1;
                  end Decrement;
            or
                  accept Get (Result : out Integer) do
```

```
                    Result := Value;
                end Get;
            or
                delay 1.0 * Minute;
                exit;
            end select;
        end loop;
end Counter;
```

Когда поток оператора задачи достигнет (**select**) выбора, он будет ожидать всех четырех событий – трех записей и задержки – параллельно. Если задержка одной минуты будет превышена, то задача выполнит операторы после оператора задержки (и в этом случае выйдет из цикла, в действительности завершая задачу). Иначе Приняв (**accept**) выбор, тела для *Increment*, *Decrement*, или *Get*, записи будут выполняться, как их вызывают. Эти четыре раздела избранного оператора взаимоисключающие: при каждой итерации цикла только один будет вызван. Это – критическая точка; если задача была записана как пакет с процедурами для различных операций, то “состояние состязания” могло произойти, где многократные задачи, одновременно вызывая, скажем, Инкремент, заставят значение только быть постепенно увеличенным один раз. В версии компилятора с управлением задачами, если нескольких задач одновременно вызывают Increment, то значение будет увеличиваться для каждой из задач только один раз, когда оно будет принято.

Более конкретно, каждая запись имеет ассоциированную с ней очередь ожидающих вызывающих абонентов. Если задача вызывает одну из записей, и Счетчик (*Counter*) не готов принять вызов (т.е., если Счетчик (*Counter*) не приостановлен в избранном операторе), тогда, задача вызова приостановлена и помещена в очередь записи, которую она вызывает. С точки зрения *Counter* задачи при любой итерации цикла есть несколько возможностей:

• Нет никакого вызова, ожидающего ни на одной из записей. В этом случае приостановлен Счетчик. Это будет пробуждено первым из двух событий: запрос к одной из его записей (который будет тогда сразу принят), или истечение одной минутной задержки (чей эффект был отмечен выше).

• Есть вызов, ожидающий на точно одной из записей. В этом случае управляют передачами в избранное ответвление с оператором принятия для той записи. Выбор, который вызывающая сторона принимает, если больше чем один, зависит от политики организации очередей, которая может быть определена через прагму, определенную в Приложении Систем реального времени стандарта Ada; значение по умолчанию – Метод "первым пришел – первым вышел".

• Есть вызовы, ожидающие больше чем на одной записи. В этом случае одна из записей с незаконченными вызывающими сторонами выбрана, и затем одна из вызывающих сторон выбрана, чтобы быть исключенной из очереди (выбор зависит от политики организации очередей).

4. Защищенные объекты

Несмотря на то, что рандеву может быть использован для реализации взаимно исключающего доступа к общему объекту данных, альтернативный (и обычно предпочтительно) стиль программирования через защищенный объект (*protected object*), эффективно реализуемого механизма, который делает эффект более явным. У защищенного объекта есть открытый интерфейс (его защищенные операции) для доступа и управления компонентами объекта (его приватная часть). Взаимное исключение осуществлено через концептуальную блокировку на объекте, и инкапсуляция гарантирует, что единственный внешний доступ к компонентам через защищенные операции.

Два вида операций могут быть выполнены на таких объектах: операции чтения–записи процедурами или записями, и операции только для чтения функциями. Механизм блокировки реализован так, чтобы было возможно выполнить параллельные операции чтения, но не параллельную запись или операции чтения–записи.

Давайте повторно реализуем наш более ранний пример управления задачами с защищенным объектом под названием Счетчик (*Counter*):

```
protected Counter is
      function Get return Integer;
      procedure Increment;
      procedure Decrement;
private
      Value : Integer := 0;
end Counter;

protected body Counter is
      function Get return Integer is
      begin
            return Value;
      end Get;

      procedure Increment is
      begin
            Value := Value + 1;
      end Increment;

      procedure Decrement is
      begin
            Value := Value - 1;
      end Decrement;
end Counter;
```

Наличие двух абсолютно различных способов реализовать ту же парадигму могло бы казаться сложным. Однако на практике фактическая проблема обычно стимулирует решение выбора между активной структурой (задача) или пассивной структурой (защищенный объект).

К защищенному объекту можно получить доступ через префиксную нотацию:

```
Counter.Increment;
Counter.Decrement;
Put_Line (Integer'Image (Counter.Get));
```

Защищенный объект может быть похожим на пакет синтаксически, так как он содержит объявления, к которым можно получить доступ, внешне используя префиксную нотацию. Однако объявление защищенного объекта чрезвычайно ограничено; например, никакие общедоступные данные не позволены, никакие типы не могут быть объявлены внутри и т.д. И помимо синтаксических различий, есть критическое семантическое различие: у защищенного объекта есть концептуальная блокировка, которая гарантирует взаимное исключение; нет такой блокировки для пакета.

Также как типы задачи, возможно, объявить защищенные типы, которые можно несколько раз инициализировать:

```
declare
      protected type Counter is
            -- as above
      end Counter;

      protected body Counter is
            -- as above
      end Counter;

      C1 : Counter;
      C2 : Counter;
begin
      C1.Increment;
      C2.Decrement;
      ...
end;
```

Защищенные объекты и типы могут объявить подобную процедуре работу, известную как “запись”. Запись несколько подобна процедуре, но включает так называемое условие барьера (*barrier condition*), которое должно быть истиной для вызова записи, чтобы успешно выполниться. Вызов защищенной записи является, таким образом, процессом в два шага: во–первых, получите блокировку на объекте, и затем оцените условие барьера. Если условие будет истиной, то тогда вызывающая сторона выполнит код выполняющий запись. Если условие – ложь, то вызывающая сторона помещена в очередь для записи и оставляет блокировку. Условия барьера (для записей с непустыми очередями) переоцениваются после завершения защищенных процедур и защищенных записей.

Вот пример, иллюстрирующий защищенные записи: защищенный тип, который моделирует двоичный семафор / персистентный сигнал.

```
protected type Binary_Semaphore is
      entry Wait;
      procedure Signal;
private
      Signaled : Boolean := False;
end Binary_Semaphore;
```

```
protected body Binary_Semaphore is
      entry Wait when Signaled is
      begin
            Signaled := False;
      end Wait;

procedure Signal is
      begin
            Signaled := True;
      end Signal;
end Binary_Semaphore;
```

Функции параллелизма Ada обеспечивают гораздо дальше общность, чем, что было представлено здесь. За дополнительной информацией обращайтесь к одному из разделов в References.

Глава двенадцатая. НИЗКОУРОВНЕВОЕ ПРОГРАММИРОВАНИЕ

1. Уровни представления

Мы видели в предыдущих главах, как Ada может использоваться, чтобы описать семантику высокого уровня и архитектуру. Красота языка, однако, состоит в том, что он может использоваться во всех случаях вплоть до самой низко–уровневой разработки, включая встроенный ассемблерный код или управление битами данных и уровня регистров устройств (не выходя за рамки стандарта).

Одна очень интересная функция языка – то, что, в отличие от C, например, нет никаких ограничений представления данных, если не определено разработчиком. Это означает, что компилятор свободен, чтобы выбрать лучший компромисс с точки зрения представления по сравнению с производительностью. Давайте начнем со следующего примера:

```
[Ada]
type R is record
      V : Integer range 0 .. 255;
      B1 : Boolean;
      B2 : Boolean;
end record
with Pack;

[C++]
struct R {
      unsigned int v:8;
      bool b1;
      bool b2;
};

[Java]
public class R {
      public byte v;
      public boolean b1;
      public boolean b2;
}
```

Приведенный в примере выше код Ada и C++, в обоих случаях оптимизирован для создания объекта, который должен занимать как можно меньше памяти. Управление размером данных не возможно в Java, но язык действительно определяет размер значений для типов примитивов.

Несмотря на то, что код C++ эквивалентен коду Ada в этом определенном примере, есть интересное семантическое различие. В C++ должно быть определено число битов, требуемых каждым полем. Здесь, мы утверждаем, что v составляет только 8 битов, эффективно представляя

значения от 0 до 255. В Ada это наоборот: разработчик определяет диапазон требуемых значений, и компилятор решает, как отобразить код, с оптимизацией по скорости доступа или по размеру занимаемого в памяти. Аспект **Pack**, объявленный в конце записи, определяет, что компилятор должен оптимизировать для размера даже за счет уменьшенной скорости в доступе к компонентам записи.

Другие пункты представления могут быть определены также, вместе с проверками непротиворечивости времени компиляции между требованиями с точки зрения доступных значений и указанных размеров. Это особенно полезно, когда определенное расположение необходимо; например, при взаимодействии через интерфейс с аппаратными средствами, драйвером или протоколом связи. Вот то, как определить определенный формат данных на основе предыдущего примера:

```
type R is record
     V : Integer range 0 .. 255;
     B1 : Boolean;
     B2 : Boolean;
end record;

for R use record
     -- Occupy the first bit of the first byte.
     B1 at 0 range 0 .. 0;

     -- Occupy the last 7 bits of the first byte,
     -- as well as the first bit of the second byte.
     V at 0 range 1 .. 8;

     -- Occupy the second bit of the second byte.
     B2 at 1 range 1 .. 1;
end record;
```

Мы опускаем директиву **with** *Pack* и вместо этого используем рекордный пункт представления после рекордного объявления. Компилятор предписан распространить объекты типа *R* через два байта. Расположение, которое мы определяем здесь, довольно неэффективно, чтобы работать на любой машине, но у Вас может быть конструкция компилятора, которая реализует наиболее эффективные методы для доступа, вместо того, чтобы кодировать вручную Ваши собственные машинно–зависимые методы разрядного уровня.

2. Встроенный ассемблерный код

При выполнении низкоуровневой разработки, такой как в драйвере ядра или на уровне драйвера оборудования, могут быть моменты, когда необходимо реализовать функциональность с ассемблерным кодом.

У каждого компилятора Ada есть свои собственные соглашения для встраивания ассемблерного кода, на основе аппаратной платформы и поддерживаемого ассемблера (ассемблеров). Наши примеры здесь будут работать с GNAT и GCC на x86 архитектуре.

Все x86 процессоры, совместимые с Intel спецификацией Pentium имеют инструкцию *rdtsc*, которая сообщает нам число циклов, начиная с последнего сброса процессора. Эта инструкция не имеет аргументов и помещает 64 разделения битового значения без знака в регистрах eax и edx (регистры имеют 32 разряда).

GNAT обеспечивает подпрограмму под названием *System.Machine_Code.Asm*, которая может использоваться для вставки ассемблерного кода. Вы можете определить строку, чтобы передать ее к ассемблеру, а также переменным исходного уровня (переменные определенные в исходном тексте программы), которые будут использоваться для ввода и вывода:

```
with System.Machine_Code;  use System.Machine_Code;
with Interfaces;           use Interfaces;

function Get_Processor_Cycles return Unsigned_64 is
      Low, High : Unsigned_32;
      Counter : Unsigned_64;
begin
      Asm ("rdtsc",
            Outputs =>
                  (Unsigned_32'Asm_Output ("=a", High),
                  Unsigned_32'Asm_Output ("=d", Low)),
            Volatile => True);

      Counter :=
      Unsigned_64 (High) * 2 ** 32 +
      Unsigned_64 (Low);

      return Counter;
end Get_Processor_Cycles;
```

Пункты *Unsigned_32'Asm_Output* из примера предоставляют ассоциациям между регистрами машины и переменными исходного уровня, которые будут обновлены. “=a” и “=d” относятся к eax и edx регистрам машины, соответственно. Использование типов *Unsigned_32* и *Unsigned_64* от Интерфейсов пакета гарантирует корректное представление данных. Мы собираем два 32–разрядных значения, чтобы сформировать единственные 64 битовых значения.

Мы устанавливаем параметр *Volatile* в *True*, чтобы сообщить компилятору, что вызов этой инструкции многократно с теми же исходными вводами может привести к различным выводам. Это устраняет возможность, что компилятор оптимизирует многократные вызовы в единственный вызов.

С включенной оптимизацией компилятор GNAT достаточно умен, чтобы использовать eax и регистры edx, чтобы реализовать *High* и *Low* переменные, приводящие к нулю в результате переполнения для интерфейса блока.

Интерфейс вставки машинного кода обеспечивает много функций вне того, что было показано здесь. Больше информации может быть найдено в Руководстве пользователя GNAT и Справочнике GNAT.

3. Взаимодействие через интерфейс с C

Много усилия было потрачено, делая Ada простой, чтобы взаимодействовать через интерфейс с другими языками. Иерархия пакета *Interfaces* и прагм *Convention*, *Import* и *Export* позволяют Вам выполнять внешне языковые вызовы при соблюдении надлежащего представления данных для каждого языка.

Давайте запустимся со следующего кода C:

```
struct my_struct {
      int A, B;
};
void call (my_struct * p) {
      printf ("%d", p->A);
}
```

Чтобы вызвать эту функцию из программы написанной на языке Ada, компилятор Ada требует, чтобы описание структуры данных передалось, а также описание самой функции. Чтобы получить **struct** *my_struct*, как она представлена в языке C, мы можем использовать следующую запись вместе с **pragma** *Convention*. Прагма (pragma) указывает компилятору, что данные необходимо интерпретировать в памяти как это делает компилятор C.

```
type my_struct is record
      A : Interfaces.C.int;
      B : Interfaces.C.int;
end record;
pragma Convention (C, my_struct);
```

Описание внешнего вызова подпрограммы к коду Ada называют “binding”, и это выполняется на двух этапах. Во–первых, спецификация подпрограммы Ada, эквивалентная функции C, кодирована. Функция C возвращает аргументы к функции Ada и пустая функция отображается в виде прототипа процедуры Ada. Затем вместо того, чтобы реализовывать подпрограмму, используя код Ada, мы используем **pragma** *Import*:

```
procedure Call (V : my_struct);
pragma Import (C, Call, "call"); -- Third argument optional
```

Прагма *Import* определяет, что каждый раз, когда *Call* вызван кодом Ada, это должно вызвать функцию *Call* с соглашением о вызовах языка C.

И это – все, что для этого необходимо. Вот пример вызова *Call*:

```
declare
      V : my_struct := (A => 1, B => 2);
begin
      Call (V);
end;
```

Вы можете также сделать подпрограммы Ada доступными для кода C, и примеры этого могут быть найдены в Руководстве пользователя GNAT.

Взаимодействие через интерфейс с C++ и Java использует зависящие от реализации функции, которые также доступны с GNAT.

Глава тринадцатая. ЗАКЛЮЧЕНИЕ

Все обычные парадигмы императивного программирования могут быть найдены на всех трех языках, которые мы рассмотрели в этом документе. Однако Ada отличается от остальных, в котором это более явно при выражении свойств и ожиданий. Это – хорошая вещь: быть более формальным, предоставляет лучшую передачу среди программистов в команде и между программистами и машинами. Вы также получаете больше обеспечения когерентности программы на многих уровнях. Ada может помочь уменьшать стоимость поддержки программного обеспечения, сместив усилие к созданию системы согласно озвученной спецификации в первый раз, вместо того, чтобы работать тяжелее, чаще, и за больший счет, исправлять ошибки, найденные позже в системах уже в производстве. Приложения, у которых есть потребности надежности, долгосрочные требования по техобслуживанию или безопасность/проблемы безопасности, являются теми, для которых у языка Ada есть доказанный послужной список.

Считается все более и более распространены системы, реализованные на нескольких языках, и у Ada есть стандартные средства взаимодействия через интерфейс, что позволяет коду Ada вызывать подпрограммы и/или структуры справочных данных от других языковых сред, или наоборот. Использование языка Ada, таким образом, позволяет простое взаимодействие через интерфейс между различными технологиями, используя каждого для того, в чем это является лучшим.

Мы надеемся, что это руководство обеспечило некоторое понимание мира разработчика программного обеспечения о языке Ada и уже сделало язык Ada более доступной для программистов знакомый с программированием на других языках.

Глава четырнадцатая. ССЫЛКИ

http://www.adaic.org/learn/materials/ веб–сайта Ada Information Clearinghouse, сохраняемый Ассоциацией Ada Resource, содержит ссылки ко множеству учебных материалов (книги, статьи, и т.д.), который может помочь в изучении Ada. Страница Development Center http://www.adacore.com/knowledge на веб–сайте AdaCore также содержит ссылки, к полезной информации включая видео и печатные учебные руководства для языка Ada.

Самый всесторонний учебник – Programming in Ada 2005 автор John Barnes, который ориентирован к профессиональным разработчикам программного обеспечения.

1. Дополнение от переводчика

Корпорация AdaCore создала и наполняла блог GEM.

В статье Gem#161, завершаемой семилетний цикл статей AdaCore, автор Jamie Ayre отметил:

Идея серии Gems состояла в том, чтобы предоставить информацию вокруг областей по технологии AdaCore и языку программирования Ады. Это было в частности успешно в выдвижении на первый план менее известных инструментов и обеспечении справки и подсказок в оптимизации Вашего использования GNAT и Ada, включая программные продукты: GNAT Pro, SPARK Pro, GNATstack, GNAT Component Collection, PolyORB, AWS, and GtkAda.

Из 160 статей больше всего откликов получили следующие десять:

1. Gem #119: GDB Scripting - Part 1
2. Gem #128: Iterators in Ada 2012 - Part 2
3. Gem #140: Bridging the Endianness Gap
4. Gem #123: Implicit Dereferencing in Ada 2012
5. Gem #84: The Distributed Systems Annex 1 - Simple client/server
6. Gem #142: Exception-ally
7. Gem #138: Master the Command Line - Part 1
8. Gem #59: Generating Ada bindings for C headers
9. Gem #127: Iterators in Ada 2012 - Part 1
10. Gem #117: Design Pattern: Overridable Class Attributes in Ada

Доступ к статье: Gem #161 : So long and thanks for all the memories!

Архив Gem доступен по ссылке: http://www.adacore.com/adaanswers/gems/archive

Ниже в разделе приводится аннотация статей блога GEM.

Gem #1: Лимитированные типы в Ада 2005 – лимитированные агрегаты

Краткое содержание: Gem Ада #1 – При помощи Ада 2005 можно создавать лимитированные объекты с помощью агрегатов, что позволяет применять правила полного покрытия; ранее подобное было доступно только для нелимитированных типов.

Gem #2: Лимитированные типы в Ада 2005 – Нотация типа <> в агрегатах

Краткое содержание: Gem Ада #2 – Нотация типа <> может быть использована в агрегатах для запроса начального значения определенных компонентов.

Gem #3: Лимитированные типы в Ада 2005 – Конструкторные функции

Краткое содержание: Gem Ада #3 – Можно использовать функции конструктора для создания объектов лимитированных типов. Результаты вызовы подобных функций реализуется «на месте» в самих создаваемых объектах.

Gem #4: Контроль позиции ШИМ для радиоуправляемых сервосистем

Краткое содержание: Gem Ада #4 – На данном примере показано практическое применение событий с контролем по времени в Ада 2005

Gem #5: Поиск по ключу в сетах

Краткое содержание: Gem Ада #5 – Сеты – это контейнеры элементов. Во всех контейнерах, в том числе – сетах, можно осуществлять поиск элементов по их значению с учетом того, что значение элемента известно. В некоторых приложениях эквивалентность элементов определяется только по некоторой их части («ключевой» части); часто приходится осуществлять поиск элемента, когда известно только значение такого ключа. Здесь представлен способ осуществления ключевого поиска элемента в контейнере.

Gem #6: Сравнение идиом различного представления в Ада 95 и интерфейсов Ада 05

Краткое содержание: Gem Ада #6 – идиома различного представления позволяет осуществлять деривацию из множества помеченных типов одновременно. Этот механизм весьма удобен для составления абстракций, дополняющих типы интерфейсов.

Gem #7: Красота числовых литералов в языке Ада.

Краткое содержание: Gem Ада #7 – Известно ли вам, что волшебство языка Ада распространяется на целые и действительные числа (на самом деле, на числовые литералы). Читайте дальше, чтобы узнать больше о данном Gem Ада.

Gem #8: Фабричные функции

Краткое содержание: Gem Ада #8 – Фабричные функции позволяют создавать объекты какого–либо класса, когда известен только объект другого класса. С их помощью можно реализовывать присваивание объектам надклассовых типов, чтобы избежать исключений, вызванных несоответствием тегов.

Gem #9: Надклассовые операции, итераторы и типовые алгоритмы

Краткое содержание: Gem Ада #9 – типовые алгоритмы можно использовать для работы с любыми контейнерами (включая массивы), в

которых возможна итерация элементов. В данном примере показано, как можно систематически изменять операцию копирования контейнеров одного класса, превратив ее в типовой алгоритм, подходящий для любого контейнера.

Gem #10: Лимитированные типы в Ада 2005 – Расширенные операторы возврата

Краткое содержание: Gem Ада #10 – Для присвоения создаваемому в функцией объекту имени можно использовать функцию extended_return_statement.

Gem #11: Лимитированные типы в Ада 2005 – Конструкторные функции, часть 2

Краткое содержание: Gem Ада #11 – Данный пример демонстрирует, как конструкторные функции можно применять в разных контекстах для создания лимитированных объектов на месте.

Gem #12: Лимитированные типы в Ада 2005 – Нотация типа <> в агрегатах, часть 2

Краткое содержание: Gem Ада #12 – Данный пример демонстрирует, как с помощью нотации типа <> в агрегатах можно лучшим образом применять начальные значения компонента записи, чтобы избежать повторения кода.

Gem #13: Идиомы для работы с прерываниями (Часть 1)

Краткое содержание: Gem Ада #13 – Для работы с прерываниями в языке Ада обычно используются две идиомы. Характеристики одной из них более подходят под определение качественной разработки программного обеспечения, в то время как у другой лучшие рабочие показатели. В данном gem рассматриваются эти две идиомы.

Gem #14: Идиомы для работы с прерываниями (Часть 2)

Краткое содержание: Gem Ада #14 – Для работы с прерываниями в языке Ада обычно используются две идиомы. Характеристики одной из них более подходят под определение качественной разработки программного обеспечения, в то время как у другой лучшие рабочие показатели. В данном gem рассматриваются эти две идиомы.

Gem #15: Таймеры

Краткое содержание: Gem Ада #15 – Таймеры – весьма важные программные элементы встроенных систем и систем, работающих в реальном времени. Простой и понятный способ создания таймеров облегчает задачу разработки программного обеспечения.

Gem #16: Прагма No_Return

Краткое содержание: Gem Ада #16 – Прагму No_Return можно использовать для пометки процедур, в которых невозможно обычным способом задать возвращение значения.

Gem #17: Прагма No_Return, Часть 2 (функции)

Краткое содержание: Gem Ада #17 – GNAT выдает предупреждение при обнаружении функций, которые могут завершить выполнение, и иногда

подобные предупреждения о неисправности ложные. Прагма No_Return позволяет избежать подобных ложных предупреждений.

Gem #18: Предупреждения GNAT

Краткое содержание: Gem Ада #18 – Данный «gem» содержит информацию не о языке Ада как таковом, а о наборе элементов GNAT, связанных с предупреждениями. Здесь показано, как наилучшим образом обращаться с предупреждениями, генерируемыми GNAT.

Gem #19: Передача данных объектов Ада в потоке XML

Краткое содержание: Gem Ада #19 – Передача данных объектов Ада в потоке XML

Gem #20: Использование прагмы Shared_Passive для обеспечения сохранности данных

Краткое содержание: Gem Ада #20 – Использование прагмы Shared_Passive для обеспечения сохранности данных

Gem #21: Анализ XML текста

Краткое содержание: Gem Ада #21 – Консорциум World Wide Web (W3C) разрабатывает различные спецификации на формат файлов XML. В частности, определены различные API для загрузки, обработки и записи файлов XML. Несмотря на то, что для языка Ада API не определены, XML/Ада соответствует им настолько, насколько возможно. В данном gem описывается, как использовать XML/Ада для анализа XML файлов.

Gem #22: Ада знает много языков

Краткое содержание: Gem Ада #22 – Ада знает много языков. Сколько именно? Все языки, которые можно написать на клавиатуре. Читайте дальше, чтобы узнать больше о данном Gem Ада.

Gem #23: Нежелательно значение null

Краткое содержание: Gem Ада #23 – синтаксис «not null» позволяет программам, написанным на языке Ада 2005, предотвращать «null» значение доступа в случаях, когда подобное нежелательно. Этот новый вид синтаксиса позволяет создавать полезную документацию.

Gem #24: Нежелательно значение null (Часть 2 – Эффективность)

Краткое содержание: Gem Ада #24 – С помощью синтаксиса «not null» можно повысить эффективность программ за счет отсутствия необходимости в неявно определенных динамических проверках.

Gem #25: Как осуществлять поиск текста

Краткое содержание: Gem Ада #25 – Стандартом Ада определяется несколько подпрограмм, связанных с поиском текста в строке. Кроме того, в GNAT доступны дополнительные пакеты поиска. В данном «gem» описываются эти функции.

Gem #26: Атрибут Mod

Краткое содержание: Gem Ада #26 – С помощью T'Mod можно преобразовывать целочисленные типы со знаком в модульные целочисленные типы, применяя модульную (циклическую) арифметику.

Gem #27: Изменение представления данных (Часть 1)

Краткое содержание: Gem Ада #27 – Часть 1, автоматическое изменение представления

Gem #28: Изменение представления данных (Часть 2)

Краткое содержание: Gem Ада #28 – Часть 2, эффективность

Gem #29: Введение в Веб–Серверы Ада (AWS)

Краткое содержание: Gem Ада #29 – Введение в Веб–Серверы Ада (AWS)

Gem #30: Безопасное и надежное программное обеспечение: Введение

Краткое содержание: На этой неделе в «gem» представляется введение в новую брошюру Джона Барнса: Безопасное и надежное программное обеспечение: Введение в Ада 2005. В течение нескольких месяцев будут опубликованы все тринадцать глав брошюры.

Gem #31: предусловия/постусловия

Краткое содержание: Gem Ада #31 – Понятие предусловий и постусловий определено уже давно. Предусловие – это такое условие, которое должно быть соблюдено (возвращать значение true) перед тем, как часть программы начнет свое исполнение. Постусловие, в свою очередь, – условие, которое должно быть соблюдено (возвращать значение true) после того, как часть программы закончила свое исполнение.

Gem #32: Безопасное и надежное программное обеспечение: Глава 1, Безопасный синтаксис

Автор: Джон Барнс

Краткое содержание: На этой неделе в «gem» представляется первая глава новой брошюры Джона Барнса:

Безопасное и надежное программное обеспечение: Введение в Ада 2005.

В течение нескольких месяцев будут опубликованы все тринадцать глав брошюры. В приложении, приведенном в конце Gem #30, можно просмотреть содержание и библиографию всего буклета. Надеемся, что вам понравится!

Gem #33: Проверка доступности (Часть I: Ада95)

Краткое содержание: Gem Ада #33 – Наличие недействительных указателей (указателей на несуществующие объекты) в программе может привести к катастрофическим последствиям. В языке Ада применяется набор «правил доступности», которые позволяют разработчике избежать подобных проблем, таким образом повышая отлаженность программного обеспечения.

Gem #34: Безопасное и надежное программное обеспечение: Глава 2, Безопасная проверка соответствия типов.

Автор: Джон Барнс

Краткое содержание: На этой неделе в «gem» представляется вторая глава новой брошюры Джона Барнса: Безопасное и надежное программное обеспечение: Введение в Ада 2005.

Gem #35: пакет кольцевых буферов в иерархии GNAT (Часть 1)

Краткое содержание: Gem Ада #35 – В Ада95 были введены так называемые «защищенные типы», представляющие собой фундаментальные элементы, необходимые для эффективного многопоточного программирования и обработки прерываний. В данном Gem описывается использование защищенных типов в ходе реализации классических асинхронных буферных абстракций, предоставляемых иерархией элементов GNAT. Предполагается, что читатель по крайней мере в какой-то степени знаком с защищенными типами, поэтому их семантика объясняется лишь частично.

Gem #36: Безопасное и надежное программное обеспечение: Глава 3, Безопасные указатели

Автор: Джон Барнс

Краткое содержание: На этой неделе в «gem» представляется третья глава новой брошюры Джона Барнса: Безопасное и надежное программное обеспечение: Введение в Ада 2005.

Gem #37: Пакет кольцевых буферов в иерархии GNAT (Часть 2)

Краткое содержание: Gem Ада #37 – В части 1 настоящего Gem были кратко представлены кольцевые буферы, защищенные типы, и объявление обобщенного пакета GNAT. Bounded_Buffers, экспортирующего защищенный тип Bounded_Buffer. В части 2 исследуется скрытая часть Bounded_Buffer и реализация защищенных записей и функций.

Gem #38: Безопасное и надежное программное обеспечение: Глава 4, Безопасная архитектура

Автор: Джон Барнс

Краткое содержание: На этой неделе в «gem» представляется четвертая глава новой брошюры Джона Барнса: Безопасное и надежное программное обеспечение: Введение в Ада 2005.

Gem #39: Эффективный поток ввода–вывода для регулярных типов.

Краткое содержание: Gem Ада #39 – Запись значений в потоки и их считывание не составляет особого труда в языке Ада благодаря «потоковым атрибутам», но начальную реализацию атрибутов некоторых регулярных типов можно осуществлять более эффективными способом. В данном Gem описывается, каким образом пользователь может определять подобные эффективные реализации.

Gem #40: Безопасное и надежное программное обеспечение: Глава 5, Безопасное объектно-ориентированное программирование

Автор: Джон Барнс

Краткое содержание: На этой неделе в «gem» представляется пятая глава новой брошюры Джона Барнса: Безопасное и надежное программное обеспечение: Введение в Ада 2005.

Gem #41: Проверка доступности (Часть II: Ada2005)

Краткое содержание: Gem Ада #41 – В Ада 2005 представлены несколько улучшений в отношении указательных типов, которые упрощают задачу разработчика и делают язык более гибким. Но большая сила означает большую ответственность, поэтому для того, чтобы новые функции не

создавали недействительные указатели, возможности проверок доступности также были расширены.

Gem #42: Безопасное и надежное программное обеспечение: Глава 6, Безопасное конструирование объектов

Автор: Джон Барнс

Краткое содержание: Данный gem с шестой главой новой брошюры Джона Барнса будет последним: мы сделаем небольшой перерыв на лето. Безопасное и надежное программное обеспечение: Введение в Ада 2005.

Gem #43: Безопасное и надежное программное обеспечение: Глава 7, Безопасное управление памятью

Автор: Джон Барнс

Краткое содержание: Серия Gem Ада вернулась! Надеемся, вы отлично провели лето.

Gem #43 – это седьмая глава новой брошюры Джона Барнса: Безопасное и надежное программное обеспечение: Введение в Ада 2005.

Gem #44: Проверка доступности (Часть III)

Краткое содержание: Gem Ада #44 – С помощью ‘Unchecked_Access можно обходить правила доступа, а контролируемые типа помогают укротить эту опасную функцию.

Gem #45: Безопасное и надежное программное обеспечение: Глава 8, Безопасный запуск

Автор: Джон Барнс

Gem #45 – это восьмая глава новой брошюры Джона Барнса: Безопасное и надежное программное обеспечение: Введение в Ада 2005.

Gem #46: Несовместимость Ада 83 и Ада 95

Краткое содержание: Gem Ада #46 – Часть 1, Неограниченные массивы в обобщенных типах

Gem #47: Безопасное и надежное программное обеспечение: Глава 9, Безопасная коммуникация

Автор: Джон Барнс

Gem #47 – это девятая глава новой брошюры Джона Барнса:

Безопасное и надежное программное обеспечение: Введение в Ада 2005.

Gem #48: Расширенные интерфейсы в Ада 2005

Краткое содержание: Gem Ада #48 – В Ада 2005 представлена концепция интерфейсов, предназначенных для разработки классов объектов. Интерфейсы – удобный инструмент для реализации новых иерархии, но после введения их в действия их довольно трудно расширить. Добавление новых примитивов может привести к ошибкам механизма создания производных типов, так как типу необходимо реализовать все наследованные абстрактные примитивы. В данном Gem демонстрируются два способа преодоления данной проблемы, один из них – общий для ООП как такового, другой – специфичный для Ада 2005.

Gem #49: Безопасное и надежное программное обеспечение: Глава 10, Безопасный параллелизм

Автор: Джон Барнс

Краткое содержание:

Gem #49 – это десятая глава новой брошюры Джона Барнса:

Безопасное и надежное программное обеспечение: Введение в Ада 2005.

Gem #50: Разрешение совмещения

Краткое содержание: Gem Ада #50 – В данном Gem обсуждаются некоторые аспекты разработки языка, связанные с разрешением совмещения.

Gem #51: Безопасное и надежное программное обеспечение: Глава 11, Обеспечение безопасности при помощи SPARK

Автор: Джон Барнс

Краткое содержание:

Gem #51 – это одиннадцатая глава новой брошюры Джона Барнса:

Безопасное и надежное программное обеспечение: Введение в Ада 2005.

Gem #52: Возможности создания скриптов в GNAT (Часть 1)

Краткое содержание:

Gem Ада #52 – По мнению большинства программистов, все языки программирования можно разделить на две категории: языки описания сценариев (скриптов), и все остальные. Трудно назвать одно определяющее различие между этими двумя категориями. Язык относят к той или иной категории чаще всего на основе того, каких масштабов приложения на нем пишутся, является ли он интерпретируемым или компилируемым, и насколько легко производить такие операции как манипуляция внешними процессами.

Однако, с уверенностью можно заявлять, что Ада не относится к скриптовым языкам. Несмотря на это, в данном Gem описываются несколько пакетов сборки GNAT, с помощью которых можно выполнять задачи, связанные со скриптованием, которые считаются возможными только в языках, специализирующихся на скриптах. Конечно же, переносимость программ – одно из важнейших преимуществ языка Ада!

Gem #53: Безопасное и надежное программное обеспечение: Глава 12: Заключение

Автор: Джон Барнс

Краткое содержание:

Gem #53 – это последняя глава новой брошюры Джона Барнса:

Безопасное и надежное программное обеспечение: Введение в Ада 2005.

Надеемся, вам понравилась эта серия публикаций. В приложении, приведенном в конце Gem #30, можно просмотреть содержание и библиографию всего буклета.

Gem #54: Возможности создания скриптов в GNAT (Часть 2)

Краткое содержание: Gem Ада #54 – В Gem # 52 мы показали несколько возможностей GNAT, которые превращают язык Ада в язык скриптования (в какой-то степени). Мы продемонстрировали функции

GNAT.Command_Line, GNATCOLL.Mmap, GNAT.Regpat и GNAT.AWK. В данном Gem продолжается обсуждение возможностей описания сценариев, на этот раз – в контексте работы с внешними процессами.

Gem #55: Введение в сопряжение Ада/Java

Краткое содержание: Gem Ада #55 – Данный Gem будет последним в 2008 г. (время летит!). Новые Gem начнут публиковаться с 12 января 2009 г. А пока что наилучшие пожелания и веселых праздников все читателям!

Сопряжение Ада и Java – весьма сложная задача. В отличие от C, C++ или Fortran, Ада и Java работают в двух разных средах: Java – на JVM, Ада – напрямую на ОС. Поэтому невозможно напрямую связать функции Java и скомпилированный код на языке Ада с помощью прагмы Import. Разработчикам предлагаются два возможных решения данной проблемы: компилировать код напрямую в байт–код Java, используя GNAT с JVM, либо использовать Java Native Interface (JNI), с помощью которого можно осуществлять взаимодействие среды машинного кода и JVM. В данном Gem будет обсуждается второй подход.

Вручную использовать слой JNI довольно сложно и может привести к множеству ошибок. К счастью, в AdaCore доступен набор инструментов для автоматизации генерации интерфейсов с помощью GNAT–AJIS. Данный Gem – первый в серии публикаций, в которых будет продемонстрировано, как использовать этот набор инструментов чтобы создавать смешанные приложения языков Ада и Java.

Gem #56: Создание вызовов Java в Ада с помощью GNAT–AJIS

Краткое содержание: Gem Ада #56 – В предыдущем Gem была представлена функция ada2java, связывающее специфицирования Ада и Java для поддержки вызовов Ада в Java. Несмотря на то, что ada2java не поддерживает пакеты сопряжений Ада с спецификациями Java, его можно использовать для поддержки вызовов Java в Ада. В данном Gem будет рассмотрена такая возможность на примере обратных вызовов (в контексте Ада – вызовов доступа к подпрограммам).

Gem #57: Диспетчеризация вызовов между Ада и Java.

Краткое содержание: Gem Ада #57 – в предыдущем Gem было показано, как создавать вызовы Java в Ада с помощью обратных вызовов и функции ada2java. Следующий шаг – это механизм межъязыковой диспетчеризации для поддержки расширения тегированного типа Ада в Java, что позволяет осуществлять взаимные вызовы диспетчеризации.

Gem #58: Обработка исключений Ада/Java

Краткое содержание: Gem Ада #58 – Ада и Java сильно зависимы от исключений. Модель данных Ада основывается на том, что данные проверяются в процессе исполнения, после чего она отображает различные исключения, например, Constraint_Error в случае ошибки, связанной с ограничениями. В Java также производятся проверки, в ходе которых может быть обнаружено множество исключений, чаще всего – проверки преобразований типов и ответ null при запросе значения переменной объекта по указателю. Поэтому очень важны исключения, развитые в обоих языках,

которые даже потенциально можно обнаружить/обработать без знания того, в каком именно языке они образовались.

Gem #59: Генерация связей с заголовочными файлами C в Ада.

Краткое содержание: Gem Ада #59 – Одна из наиболее сложных областей Ада, с которой у разработчиков часто возникают проблемы, – взаимодействие Ада с другими языками. Несмотря на то, что в Ада доступны удобные и полноценные возможности создания интерфейсов для работы с другими компилируемыми языками, такими как C, C++ и Fortran, написание связующего кода для использования масштабных и комплексных API может вызывать трудности и приводить к ошибкам.

В данном Gem будет продемонстрирован новый инструмент AdaCore, автоматизирующий генерацию интерфейсов для заголовочных файлов C с помощью компилятора.

Gem #60: Генерация связей с заголовочными файлами C++ в Ада.

Краткое содержание: Gem Ада #60 – В Gem #59 было показано как можно легко автоматически генерировать связи с заголовочными файлами C в Ада. В данном Gem будет продемонстрировано как с не меньшей легкостью можно генерировать связи с заголовочными файлами C++.

Gem #61: Сопряжение с конструкторами C++

Краткое содержание: Gem Ада #61 – В предыдущем Gem, демонстрирующем генерацию связей с заголовочными файлами C++, было кратко упомянуто сопряжение Ада с конструкторами C++ с помощью прагмы CPP_Constructor.

В данном Gem представляются типичные примеры использования конструкторов C++ в GNAT в программах, написанных на смешанных языках, а в следующем Gem будет показано применение нескольких весьма полезных взаимодействий функций Ада 2005 и конструкторами C++.

Gem #62: Конструкторы C++ и Ада 2005.

Краткое содержание: Gem Ада #62 – В предыдущем Gem было продемонстрировано, как можно просто осуществлять взаимодействие Ада и конструкторов C++.

В данном Gem будет показано, как создавать связи Ада и конструкторов C++ на более продвинутом уровне.

Gem #63: Действие прагмы Suppress.

Краткое содержание: Gem Ада #63 – Функции Ада в основном нацелены на предотвращение нарушений свойств типов данных, что достигается либо ограничениями на стадии компилирования, либо, в случае динамических свойств, – проверками в ходе выполнения. Тем не менее, проверки в ходе выполнения в Ада можно сдержать, но разработчику не стоит пользоваться этой функцией с целью обхода системы контроля типов данных.

Gem #64: Обработка многоэлементных файлов исходного кода.

Краткое содержание: Gem Ада #64 – В данном Gem демонстрируется, как можно компилировать приложения в GNAT, если в файле исходного кода содержится множество элементов. Лучше всего разбить файл исходного кода

на несколько частей, и в данном Gem будет показано, как именно можно этого добиться, но, помимо этого, в GNAT доступен обходной вариант, который позволяет сохранить файл исходного кода.

Gem #65: gprbuild

Краткое содержание: Gem Ада #65 – gprbuild – это новая функция построения программ, заменившая gnatmake. Она работает со множеством языков, автоматически контролирует зависимость по исходному коду и уменьшает необходимость повторной компиляции. В данном Gem на высоком уровен описывается работа gprbuild.

Gem #66: Редактор клавиш быстрого доступа GPS.

Краткое содержание: Gem Ада #66 – Доступ к большинству функций GPS можно получить из меню и контекстных меню. Но большинство пользователей предпочитают пользоваться клавишами быстрого доступа, так как они более быстры и эффективны. Хотя некоторые клавиши быстрого доступа к GPS предопределены программой, остальные можно переназначить на свое усмотрение, что будет особенно полезно при переходе на новый редактор.

Gem #67: Управление рабочим меню GPS.

Краткое содержание: Gem Ада #67 – Для работы с GPS есть множество редакторов и меню. Несколько таких меню могут быть отображены на экране одновременно. Гибкое меню рабочего стола позволяет организовывать необходимые окна наиболее удобным разработчику способом. В данном Gem описываются некоторые менее известные аспекты рабочего меню GPS.

Gem #68: Работа со SPARK. – Часть 1

Краткое содержание: Gem Ада #68 – После публикации данного Gem мы планируем сделать перерыв на лето. Новые Gem начнут публиковаться с 7 сентября 2009 г.

В этом и следующем Gem будет представлено простое руководство по функциям SPARK и его интеграции с GPS. В данном Gem будет показано, как начать проект SPARK и доказать, что в вашей программе SPARK отсутствуют доступы к неинициализированным переменным, и что она выполняется без ошибок.

Gem #69: Работа со SPARK. – Часть 2

Краткое содержание: Gem Ада #69 – С возвращением! Надеемся, вы отлично провели лето. Пора вернуться к тому, на чем мы остановились.

В этом и предыдущем Gem представлено простое руководство по функциям SPARK и его интеграции с GPS. В предыдущем Gem было показано, как начать проект SPARK и доказать, что в вашей программе SPARK отсутствуют доступы к неинициализированным переменным, и что она выполняется без ошибок. В данном Gem будет продемонстрировано как доказать, что ваши программы SPARK выполняют условия своих контрактов.

Gem #70: Идиома Scope Locks

Краткое содержание: Gem Ада #70 – В данном Gem мы немного передохнем от SPARK. Мы вернемся к нему через две недели и начнем серию из шести Gem по данной теме.

Инкапсуляция разделяемых переменных в защищенных операциях не всегда возможна. В данном Gem будет показано как добавлять взаимные исключения в существующий последовательный код с помощью комбинации контролируемых и защищенных типов так, чтобы получившийся в результате код был надежен и максимально неизменен.

Gem #71: Работа с Tokeneer – Урок 1

Краткое содержание: Gem Ада #71 – В предыдущем Gem было продемонстрировано, как создавать проект SPARK для GPS, работать с инструментами Examiner и Simplifier для верификации различных свойств функции простого линейного поиска. В данной серии публикаций будут более досконально исследованы возможности SPARK с помощью исходного кода Tokeneer, биометрической программной системы с высоким уровнем безопасности, разработанной по заказу Агентства Национальной Безопасности.

В данном Gem будет показано, как исправить ненадлежащую инициализацию.

Gem #72: Работа с Tokeneer – Урок 2

Краткое содержание: Gem Ада #72 – В предыдущем Gem было показано, как исправлять ненадлежащую инициализацию в SPARK на основе исходного кода Tokeneer. В данном Gem будет показано, как определять недействительные утверждения.

Gem #73: Работа с Tokeneer – Урок 3

Краткое содержание: Gem Ада #73 – В предыдущем Gem было показано, как определять недействительные утверждения в SPARK на основе исходного кода Tokeneer. В данном Gem будет показано, как определять правильность входных данных.

Gem #74: Работа с Tokeneer – Урок 4

Краткое содержание: Gem Ада #74 – В предыдущем Gem было показано, как определять правильность входных данных в SPARK на основе исходного кода Tokeneer. В данном Gem будет показано, как с помощью набора инструментов SPARK Toolset верифицировать свойства безопасности и надежности для приложений.

Gem #75: Работа с Tokeneer – Урок 5

Краткое содержание: Gem Ада #75 – В предыдущем Gem было показано, как с помощью набора инструментов SPARK Toolset верифицировать свойства безопасности и надежности приложений на основе исходного кода Tokeneer. В данном Gem будет показано, как обрабатывать ошибки переполнения.

Gem #76: Работа с Tokeneer – Урок 6

Краткое содержание: Gem Ада #76 – В предыдущем Gem было показано, как обрабатывать ошибки переполнения на основе исходного кода Tokeneer. В данном Gem будет показано, как обеспечить надежность потока информации.

Данный Gem является последним в серии публикаций о Tokeneer. Мы хотели бы поблагодарить команду разработчиков SPARK в Praxis HIS за предоставленные ресурсы.

Мы сделаем перерыв на праздники и вернемся с новыми публикациями 11 января 2010 г. С праздниками вас, уважаемые читатели, где бы вы ни были!

Gem #77: Куда делась моя память? (Часть 1)

Краткое содержание: Gem Ада #77 – Для контроля за использованием памяти, обнаружением утечек памяти и разрешения проблем общего характера, связанных с памятью, существует множество инструментов и библиотек. В данной и нескольких последующих публикациях будет предложен обзор этой тематики и описание работы с подобными инструментами.

Gem #78: Куда делась моя память? (Часть 2)

Краткое содержание: Gem Ада #78 – Для контроля за использованием памяти, обнаружением утечек памяти и разрешения проблем общего характера, связанных с памятью, существует множество инструментов и библиотек. Эта серия из трех публикаций содержит обзор этой тематики и описание работы с подобными инструментами.

Gem #79: Куда делась моя память? (Часть 3)

Краткое содержание: Gem Ада #79 – Для контроля за использованием памяти, обнаружением утечек памяти и разрешения проблем общего характера, связанных с памятью, существует множество инструментов и библиотек. Эта серия из трех публикаций содержит обзор этой тематики и описание работы с подобными инструментами.

Gem #80: Функции shift и rotate в SPARK

Краткое содержание: Gem Ада #80 – В данном Gem будет обсуждаться проблема, с которой я сам недавно столкнулся, когда работал с крипто-алгоритмами в SPARK: как применять предопределенные функции Ада shift и rotate с беззнаковыми типами из программы SPARK.

Gem #81: Семафоры GNAT

Краткое содержание: Gem Ада #81 – В ранее опубликованном Gem (#70, Идиома Scope Lock) обсуждалась иногда возникающая необходимость применять механизм низкоуровневой синхронизации для защищенной объектной конструкции высшего уровня. Приведенный код ссылался на возможности пакета семафоров в иерархии GNAT. В данном Gem будут исследованы абстрактные представления, предоставляемые этим пакетов, в частности – проектные решения.

Gem #82: Надежность на основе типов 1: Обработка данных, значение которых может быть потенциально изменено пользователей (данных типа «tainted»).

Краткое содержание: Gem Ада #82 – Благодаря надежной системе типов в Ада довольно удобно проверять в процессе компиляции верификацию таких параметров безопасности, как, например, использование значений, которые не могут быть изменены пользователем там, где это

необходимо, или надлежащая проверка данных перед их использованием в уязвимом контексте (например, при возможной попытке взлома путем внедрения SQL кода).

В этой и последующей публикации будут представлены два кратких примера данной практики. В данном Gem обсуждается обработка данных, которые потенциально могут быть изменены пользователем.

Gem #83: Надежность на основе типов 2: Проверка входных данных

Краткое содержание: Gem Ада #83 – Благодаря надежной системе типов в Ада довольно удобно проверять в процессе компиляции верификацию таких параметров безопасности, как, например, использование значений, которые не могут быть изменены пользователем там, где это необходимо, или надлежащая проверка данных перед их использованием в уязвимом контексте (например, при возможной попытке взлома путем внедрения SQL кода).

В предыдущем Gem обсуждалась обработка данных, которые потенциально могут быть изменены пользователем. В данном Gem будет разъяснено, как осуществлять проверку входных данных, поступающих команде SQL. (Пройдя по данной ссылке, можно прочитать веселый комикс, описывающий взлом вводом SQL–кода: http://xkcd.com/327/)

Gem #84: Приложение «Распределенные системы» 1 – Простой клиент/сервер

Краткое содержание: Gem Ада #84 – Этот Gem – первый в серии публикаций, показывающих возможности вспомогательного приложения по распределенным системам, описанного в Справочном Руководстве Ада (Приложение E). В данном Gem будет показано, как архитектура простого клиент/сервера может быть реализована с помощью приложения «Распределенные системы» (ПРС).

Gem Ада #85: Приложение «Распределенные системы» 2 – Распределенные объекты

Краткое содержание: Gem Ада #85: Этот Gem – второй в серии публикаций, показывающих возможности вспомогательного приложения по распределенным системам, описанного в Справочном Руководстве Ада (Приложение E). В первой публикации было показано, как архитектура простого клиент/сервера может быть реализована с помощью приложения «Распределенные системы» (ПРС). В данном Gem представляются распределенные объекты, с помощью которых можно устанавливать динамические отношения между компонентами распределенного приложения.

Gem #86: Тест по языку Ада 1 – Базовые типы

Краткое содержание: Данный Gem – один из периодически публикуемых тестов по языку Ада, состоящих из вопросов по свойствам и функциям языка, взятых из курсов AdaCore. (Больше о наших курсах можно прочитать по ссылке: http://www.adacore.com/home/products/gnatpro/professional_services/training/.)

Gem #87: Приложение «Распределенные системы», Часть 3 – Почтовые ящики

Краткое содержание: Gem Ада #87 – Этот Gem – третий в серии публикаций, показывающих возможности вспомогательного приложения по распределенным системам, описанного в Справочном Руководстве Ада (Приложение E). В двух предыдущих публикациях серии было представлено приложение «Распределенные системы» (ПРС). Было показано, как реализовывать архитектуру клиент/сервер, и описали распределенные объекты. В данном Gem будет показано, как на основе этого можно реализовать асинхронную передачу сообщений.

Gem #88: GPS – автоматическое дополнение ввода (Часть 1 из 2)

Краткое содержание: В данном Gem в виде легко воспроизводимого на практике руководства будет показано, как пользоваться автоматическим дополнением ввода. Здесь будут описаны уже доступные в GPS 4.4.0 возможности. В дальнейшем будут представлены новые возможности, реализованные в следующей версии GPS.

Gem #89: Архетипы кода программирования в реальном времени – Часть 1

Краткое содержание: Gem Ада #89 – В ходе программирования систем, работающих в реальном времени, код, обеспечивающий параллелизм и собственно работу в реальном времени часто следует явно или неявно определенным образцам. Такие образцы можно собрать в типовые архетипы и применять в случае необходимости. В этой серии публикаций будет описан набор таких архетипов, которые значительно облегчают разработку систем реального времени.

Gem #90: Приложение «Распределенные системы», Часть 4 – ПСР и C

Краткое содержание: Этот Gem – четвертый в серии публикаций, показывающих возможности вспомогательного приложения по распределенным системам, описанного в Справочном Руководстве Ада (Приложение E). В предыдущих публикациях серии было представлено приложение «Распределенные системы» (ПРС), и пояснена реализация различных парадигм взаимодействий. В данном Gem будет описано как применять данные инструменты в программе на языке C.

Gem #91: Автоматическое дополнение ввода (Часть 2 из 2)

Краткое содержание: В данном Gem продолжается тема автоматического дополнения ввода GPS, начатая в первая части этой серии публикаций. В данном Gem можно ознакомиться с новшествами, которые станут доступны в следующей версии GPS.

Gem #92: Архетипы кода программирования в реальном времени – Часть 2

Краткое содержание: В ходе программирования систем, работающих в реальном времени, код, обеспечивающий параллелизм и собственно работу в реальном времени часто следует явно или неявно определенным образцам. Такие образцы можно собрать в типовые архетипы и применять в случае необходимости. В этой серии публикаций будет описан набор таких

архетипов, которые значительно облегчают разработку систем реального времени.

Gem #93: Программирование высокоэффективных многоядерных приложения – Часть 1

Краткое содержание: В данном Gem описывается реализация в Ада программы оценки эффективности «Chameneos–Redux», которая сравнивает показатели многопотокового приложения на многоядерной платформе. Эта серия публикаций будет нацелена на описание разработки и реализации способов создания высокоэффективных версий программы в Ада.

Gem #94: Архетипы кода программирования в реальном времени – Часть 3

Краткое содержание: В ходе программирования систем, работающих в реальном времени, код, обеспечивающий параллелизм и собственно работу в реальном времени часто следует явно или неявно определенным образцам. Такие образцы можно собрать в типовые архетипы и применять в случае необходимости. В этой серии публикаций будет описан набор таких архетипов, которые значительно облегчают разработку систем реального времени.

Gem #95: Динамический анализ стека в GNAT

Краткое содержание: В данном Gem описывается возможность GNAT динамически рассчитывать объем используемого задачей стека в процессе выполнения.

Gem #96: Архетипы кода программирования в реальном времени – Часть 4

Краткое содержание: В ходе программирования систем, работающих в реальном времени, код, обеспечивающий параллелизм и собственно работу в реальном времени часто следует явно или неявно определенным образцам. Такие образцы можно собрать в типовые архетипы и применять в случае необходимости. В этой серии публикаций будет описан набор таких архетипов, которые значительно облегчают разработку систем реального времени.

Gem #97: Подсчет ссылок в Ада – Часть 1

Краткое содержание: В данной серии из трех публикаций будет описано возможная реализация автоматического управления памятью с помощью подсчета ссылок. В части 1 объясняется, как применять контролируемый тип для реализации автоматического подсчета ссылок, а также описываются некоторые аспекты правильной обработки подсчета. В части 2 будут проанализированы некоторые проблемы, связанные с многозадачностью. В части 3 будет описана реализация подсчета слабых ссылок.

Gem #98: Программирование высокоэффективных многоядерных приложений – Часть 2

Краткое содержание: Данный Gem – второй в серии публикаций, в которых описывается реализация в Ада программы оценки эффективности «Chameneos–Redux», которая сравнивает показатели многопотокового

приложения на многоядерной платформе. Эта серия публикаций будет нацелена на описание разработки и реализации способов создания высокоэффективных версий программы в Ада.

Gem #99: Подсчет ссылок в Ада – Часть 2 Безопасность операций

Краткое содержание: В данной серии из трех публикаций описывается возможная реализация автоматического управления памятью с помощью подсчета ссылок. В части 1 объясняется, как применять контролируемый тип для реализации автоматического подсчета ссылок, а также описываются некоторые аспекты правильной обработки подсчета. В части 2 будут проанализированы некоторые проблемы, связанные с многозадачностью. В части 3 будет описана реализация подсчета слабых ссылок.

Gem #100: Подсчет ссылок в Ада – Часть 3 Слабые ссылки

Краткое содержание: В данной серии из трех публикаций описывается возможная реализация автоматического управления памятью с помощью подсчета ссылок. В части 1 объясняется, как применять контролируемый тип для реализации автоматического подсчета ссылок, а также описываются некоторые аспекты правильной обработки подсчета. В части 2 будут проанализированы некоторые проблемы, связанные с многозадачностью. В части 3 будет описана реализация подсчета слабых ссылок.

Gem #101: Сервера SOAP/WSDL

Краткое содержание: В данном Gem описывается построение сервера для обеспечения веб–услуг в сети.

Gem #102: Клиент SOAP/WSDL

Краткое содержание: В данном Gem будет продемонстрирована использование веб-услуг в соответствии с описанием, представленным в документе WSDL.

Gem #103: Архетипы кода программирования в реальном времени – Часть 5

Краткое содержание: В ходе программирования систем, работающих в реальном времени, код, обеспечивающий параллелизм и собственно работу в реальном времени часто следует явно или неявно определенным образцам. Такие образцы можно собрать в типовые архетипы и применять в случае необходимости. В этой серии публикаций будет описан набор таких архетипов, которые значительно облегчают разработку систем реального времени.

Gem #104: Функция gprbuild и конфигурационные файлы – Часть 1

Краткое содержание: Данный Gem – первый в серии из трех публикаций, нацеленных на описание конфигурационных возможностей gprbuild. В этой публикации будет пояснено, как настроить gprbuild для работы с пользовательским компилятором с помощью конфигурационных файлов.

Gem #105: Леди Ада целует питона – Часть 1

Краткое содержание: В данной серии из двух публикаций будет пояснено как применять коллекцию компонентов GNAT для сопряжения кода на языке Ада и кода на языке Python. В первом Gem описываются

преимущества такой практики и трудности, с которыми можно столкнуться при прямом сопряжении с библиотеками Python. Во втором Gem будет показано, как начать работать с GNATCOLL для того, чтобы значительно упростить процесс сопряжения.

Gem #106: Леди Ада целует питона – Часть 2

Краткое содержание: В данной серии из двух публикаций будет пояснено как применять коллекцию компонентов GNAT для сопряжения кода на языке Ада и кода на языке Python. В первом Gem описывались преимущества такой практики и трудности, с которыми можно столкнуться при прямом сопряжении с библиотеками Python. Во втором Gem будет показано, как начать работать с GNATCOLL для того, чтобы значительно упростить процесс сопряжения.

Gem #107: Предотвращение освобождения памяти в типах с подсчитанными ссылками.

Краткое содержание: Автор данного Gem – Ada Magica (К.К.В. Грайн). Он продолжает тему, обсуждение которой было начато в ранее опубликованном Gem. Кристоф представит несколько способов того, как можно обезопасить API типов с подсчитанными ссылками с помощью некоторых возможностей Ада 2005.

Gem #108: Функция gprbuild и конфигурационные файлы – Часть 2

Краткое содержание: Данный Gem – второй в серии из трех публикаций, нацеленных на описание gprbuild. В этой публикации будет пояснен процесс конфигурации данного инструмент с помощью конфигурационных файлов.

Gem #109: Программные расширения и библиотеки общего пользования в Ада – Часть 1

Краткое содержание: В данном Gem мы обсудим поддержку библиотек общего пользования в GNAT, чтобы проще было понять, как создавать программные расширения в Ада.

Gem #110: Программные расширения и библиотеки общего пользования в Ада – Часть 2

Краткое содержание: В данном Gem мы продолжим обсуждение поддержки библиотек общего пользования в GNAT, и поясним, как создавать программные расширения в Ада.

Gem #111: Приложение «Распределенные системы», Часть 5 – Встроенный блок преобразования имен

Краткое содержание: Этот Gem – пятый в серии публикаций, показывающих возможности вспомогательного приложения по распределенным системам, описанного в Справочном Руководстве Ада (Приложение E). В предыдущем Gem было продемонстрировано, как осуществлять сопряжение кода ПРС с приложением C/C++ в виде автономной библиотеки Ада.

В данном Gem мы покажем, как блок преобразования имен ПРС можно встроить в главную подобласть определения вместо того, чтобы запускать его как автономный процесс.

Gem #112: Среда языка Ада для работы с Lego Mindstorms

Краткое содержание: В данной серии публикаций будет описана среда программирования языка Ада GNAT в контексте работы с набором робототехнических инструментов Lego Mindstorms. В серии будут исследованы высоко- и низкоуровневые интерфейсы для работы с аппаратным обеспечением, подмножество языка, поддерживаемое библиотекой программ этапа выполнения, и наиболее эффективные способы работы со средой. Также будут рассматриваться другие вопросы. В первом Gem серии будут представлены аспекты доступного разработчикам языкового подмножества Ада общего характера и показано, как интерфейсы Ада взаимодействую с одним таким аспектом.

Gem #113: Шаблон «Посетитель» в Ада

Краткое содержание: Шаблон «Посетитель» – это шаблон проектирования, с помощью которого можно выполнять определенные методы по отношению к объекту («посетителю»), основываясь на типе другого объекта. Этот шаблон также называют двойной диспетчеризацией, так как вид вызываемой подпрограммы зависит от типов обоих объектов.

Gem #114: Протоколирование с помощью функции GNATCOLL.Traces

Краткое содержание: Набор компонентов GNAT содержит пакет программ для протоколирования информации в различных текстовых файлах. Дополнительная информация может быть зарегистрирована для каждого протокола, что позволяет лучше понимать поведение приложения в условиях отсутствия программы отладки.

Gem #115: Среда языка Ада для работы с Lego Mindstorms – Часть 2

Краткое содержание: В данной серии публикаций будет описана среда программирования языка Ада GNAT в контексте работы с набором робототехнических инструментов Lego Mindstorms. В серии будут исследованы высоко- и низкоуровневые интерфейсы для работы с аппаратным обеспечением, подмножество языка, поддерживаемое библиотекой программ этапа выполнения, и наиболее эффективные способы работы со средой. Также будут рассматриваться другие вопросы. В данном Gem представляются базовые шаги инициализации и отключения аппаратного обеспечения Mindstorms.

Gem #116: Ада и исключения C++

Краткое содержание: Одна из главных проблем, с которой можно столкнуться в ходе сопряжения Ада и C++, – это действие исключений одного компилятора в другом. В данном Gem будет продемонстрировано, как новый механизм работы с исключениями, реализованный в GNAT, способствует работе с исключениями одного языка в другом. Следует принять во внимание, что приведенный код будет работать только в версиях GNAT начиная с Pro 7.

Gem #117: Проектировочный шаблон: Атрибуты класса с вручную корректируемыми параметрами в Ада 2012

Краткое содержание: В данном Gem будет обсуждаться реализация «атрибутов класса» (так же, как в языке Python) в Ада.

Gem #118: Аспекты переносимости направления файл-система и GNATCOLL.VFS

Краткое содержание: В данном Gem обсуждаются некоторые аспекты переносимости, связанные с файловыми системами, в частности, вопросы имен файлов, работа с чувствительными и нечувствительными к регистру файловыми системами, наборами символов и символическими ссылками. Кроме того, представляется рабочий пакет GNATCOLL.VFS.

Gem #119: Создание скриптов GDB – Часть 1

Краткое содержание: GDB, или GNU Project Debugger, – весьма удобный инструмент. В общих случаях его применение включает базовые команды CLI – break, run, print, и т.д. Но, кроме этого, он предоставляет доступ ко множеству возможностей. Одна из них – создание скриптов. По аналогии с .rc для конфигурации оболочки можно добавить .gdbinit для отладчика. В данном Gem описываются некоторые доступные возможности скриптования GDB и общая настройка с помощью .gdbinit.

Gem #120: Создание скриптов GDB– Часть 2

Краткое содержание: В предыдущем Gem было показано, какие возможности скриптования присутствуют в GDB на основе его языка написания макро-кода, с помощью которого можно настраивать отладчик через файл gdbinit. В данном Gem будут обсуждаться более продвинутые возможности скриптования GDB.

Gem # 121 – Команды для точек прерывания – Часть 1

Краткое содержание: В данном Gem предоставляется простая демострация применения команд для точек прерывания в GDB, используемых с целью проверки некоторых параметров в ходе выполнения отлаженной программы.

Gem #122: Команды для точек прерывания – Часть 2

Краткое содержание: В предыдущем Gem описывались простые примеры применения команд для точек прерывания с целью проверки некоторых параметров в ходе выполнения. В данном Gem продолжается обсуждение этой темы с примерами, демонстрирующими преимущества API Python.

Gem #123: Неявное разыменование в Ада 2012

Краткое содержание: В Gem #107 был описан метод чтения для создания безопасных ссылок на объекты контейнеров. В примере описывался указатель с подсчитанными ссылками, но подобные методы можно определять и для других контейнеров.

Gem #124: Скрипты GPS для статического анализа

Ну что ж, приступим...

Возможность осуществления быстрого запроса простых параметров из базы кода ценна как в ходе разработки, так и на этапе отладки. В большинстве случаев можно обойтись простыми решениями, наподобие grep, или обычным скриптом, основанным на запросах синтаксиса. Для более сложных запросов, требующих знания семантики, необходимо применять доступные инструменты, которые не всегда идеально подходят.

В данном Gem описывается способ осуществления запросов с помощью возможностей скриптования GPS на Python – IDE GNAT Programming studio.

Gem #125: Обнаружение бесконечной рекурсии с помощью API Python в GDB.

Ну что ж, приступим...

Предположим, вам нужно вывести таблицу факториалов от 9 до 0.

Gem #126: Проекты библиотек агрегатов

Ну что ж, приступим...

Общепринятая практика создания масштабных приложений на Ада – создание множества модулей. Каждый модуль (или подсистема) имеет свой проектный файл, который необходим не только для поддержки построения модуля, но и помощи в редактировании кожа через GPS. Это позволяет разработчикам концентрировать внимание на необходимом им модуле. В целом этот подход упрощает приложения, разбивая его на модули.

Gem #127: Итераторы в Ада 2012 – Часть 1

Итераторы в Ада 2012 облегчают восприятие текста структур данных. В данном Gem описывается новый синтакс и его взаимодействие с языком. В части 2 будет пояснено, как можно определять пользовательские итераторы в ходе формулирования новых структурных данных.

Gem #128: Итераторы в Ада 2012 – Часть 2

В части 1 мы описали базовые формы итераторов в Ада 2012 и привели несколько примеров. В этой части мы обсудим данную тему более детально и покажем, как создавать итераторы пользовательских структур данных. Начнем с изучения двух функций поддержки, введенных в Ада 2012.

Gem #129: Базы данных API, безопасные по отношению к типам – Часть 1

Для создания запросов к системам управления базами данных (СУБД) традиционно используется язык SQL. Этот язык в основном стандартизирован, несмотря на то, что каждый поставщик предлагает свои расширения и ограничения. Довольно удобно организовывать данные в виде таблицы и полей по реляционной модели. В последнее время выросла популярность так называемых баз данных «noSQL», использующих совершенно другую парадигму для лучшей производительности за счет большего объема ограничений. В данном Gem такие базы данных обсуждаться не будут.

Gem #130: Базы данных API, безопасные по отношению к типам – Часть 2

В первом Gem этой серии обсуждалось написание правильных с синтаксической точки зрения и безопасных по отношению к типам запросов SQL. Теперь нам нужно запустить эти запросы на выбранной нами СУБД и получить результаты. В данном Gem поясняется, как использовать с этой целью независимый от типа СУБД API в GNATColl.

Gem #131: Базы данных API, безопасные по отношению к типам – Часть 3

В первых двух публикациях данной серии обсуждалось, как запускать безопасные по отношению к типам запросы SQL на различных системах баз данных. В обеих публикациях было отчетливо видно применение SQL. В третьей части поясняется применение Системы управления объектами и отношениями между ними (СУОО), которая запускает запросы SQL в неявном пользователю виде.

Gem #132: Ошибочное выполнение – Часть 1

Многие разработчики, работающие с языком Ада, приходят в замешательство при виде термина «ошибочный», так как в контексте Ада данный термин означает не совсем то же, что в нормальном языке. «Ошибочный» в обыденности означает «неправильный». Но по отношению к Ада данный термин обозначает определенный тип «неправильности». В данном Gem будет прояснено значение данного термина в контексте работы с Ада.

Gem #133: Ошибочное выполнение – Часть 2

В предыдущем Gem было пояснено, что «ошибочное выполнение», согласно Справочному руководству Ада, означает, что может произойти что угодно, в частности, программа может работать правильно. В данном Gem продолжается обсуждение данной тематики.

Gem #134: Ошибочное выполнение – Часть 3

В данном Gem продолжается обсуждение примера ошибочного выполнения, начатого в части 2.

Gem #135: Ошибочное выполнение – Часть 4

Данный Gem – последний в серии публикаций на тему ошибочного выполнения. В нем обсуждается проектирования языка. Зачем в Ада вообще нужно ошибочное выполнение?

Gem #136: Сколько метров в килограмме?

В компилятор GNAT была добавлена возможность проверки измерений. Пользователь может определить физическую величину для объекта, и компилятор будет проверять совместимость объектов с их измерениями способом, схожим с таковым в инженерной практике. Измерения алгебраических выражений (включая мощности со статическими экспонентами) рассчитывается по их составляющим. В компиляторе GNAT доступен пакет поддержки системы МКС (метр-килограмм-секунда), и при необходимости пользователь может добавлять дополнительные пакеты (сантиметр-грамм-секунда, имперские единицы, и т.п.)

Gem #137: Тест по Ада 2 – Два наследника: один прямой, другой – резервный?

Тема данного Gem из серии периодических тестов по языку Ада, составленных на основе обучающих курсов AdaCore, – наследование.

Gem #138: Освойте командную строку – Часть 1

Приложения можно настраивать несколькими способами. Чаще всего применяется командная строка, конфигурационные файлы и графические пользовательские интерфейсы. Технология GNAT предоставляет различные средства для сопряжения с ними: Ada.Command_Line и

GNAT.Command_Line, GNATCOLL.Config и GtkAda. Последние два из них будут обсуждаться в последующих публикациях; тема данной серии – управление командной строкой.

Gem #139: Освойте командную строку – Часть 2

В первой части данной серии было описано, как использовать команду GNAT.Command_Line для получения параметров и аргументов, передаваемых приложению. Для этого все еще необходимо написание большого объема кода. В данной части будет пояснен высокоуровневый API команды GNAT.Command_Line и показано, как значительно упростить процесс работы с командной строкой.

Gem #140: Как преодолеть Большой Байтовый Раздел?

Два важнейших вопроса мучают всех разработчиков всю жизнь:

- Разбивать яйца всмятку с твердого или мягкого конца?
- Сохраняя многобайтовое значение, начинать с наиболее важного байта, или с наименее важного байта?

Более тридцати лет назад Дэнни Коен заключил мир между большими эндианами и маленькими эндианами, но сфера программного обеспечения до сих пор кишит кодом, обменивающим байты только для того, чтобы обработать данные, производимые системами различного поведения. Могут ли языки прийти на помощь?

Gem #141: Разберитесь с конфигурацией

В серии Gem, посвященной GNAT.Command_Line (#138 и #139), мы упомянули, что есть несколько способов осуществления контроля над поведением приложения со стороны пользователя – Операции с командной строкой (описанные в данных Gem), графические пользовательские приложения (например, GtkAda) и файлы конфигурации. Именно о файлах конфигурации и пойдет речь в данной публикации.

Gem #142: Исключительный друг исключений

Компилятор GNAT известен благодаря качеству генерируемых им сообщений об ошибках. Это относится и к сообщениям, связанным с исключениями. В данном Gem будет продемонстрировано, как можно извлечь из них пользу.

Gem #143: Возвращение к источникам

Понятие проектов было введено в технологию GNAT начиная с версии 3.15 2002 года. В дальнейших версиях его реализация была улучшена, и теперь с проектами могут взаимодействовать все инструменты GNAT. С их помощью удобно описывать организацию источников приложений и то, как с ними должны взаимодействовать различные инструменты. С ними также могут взаимодействовать пользовательские инструменты с помощью удобных API из набора компонентов GNAT, которые и описываются в данном Gem.

Gem #144: Немного о байтах. Символы и схемы кодирования

В данном Gem описываются некоторые понятия, связанные с кодированием символов и Юникодом. В нем поясняется, почему существуют различные наборы символов, а также то, как работать с ними в случае

необходимости обработки входных и исходящих данных на разных языках в приложении.

Gem #145: Тест по языку Ада 3 – утверждения

Краткое содержание: Данный Gem продолжает серию периодически публикуемых тестов, основанных на обучающих курсах AdaCore. Тема данного теста – утверждения.

Gem #146: Подтипы в Ада 2012 – Часть 1

Новая версия языка Ада полна инструментов для определения параметров типов. В данной серии из трех публикаций мы опишем три аспекта, которые могут быть использованы для определения неизменяемых параметров типов. Тема первого Gem – аспект Static_Predicate.

Gem #147: Подтипы в Ада 2012 – Часть 2

В предыдущем Gem серии мы показали, как использовать Static_Predicate для определения неизменяемых параметров скалярных объектов. Тема данного Gem – аспект Dynamic_Predicate.

Gem #148: Подтипы в Ада 2012 – Часть 3

В предыдущих Gem серии мы показали, как использовать Static_Predicate и Dynamic_Predicate для определения неизменяемых параметров объектов. Тема данного Gem – аспект Type_Invariant.

Gem #149: Говорить правду, но (возможно) не всю.

В Ада 2012 утверждение необходимых параметров программы не ограничены прагмой Assert. В данном Gem описывается использование прагмы Assertion_Policy, с помощью которой можно определять, какие утверждение должны выполняться в определенные отрезки времени выполнения.

Gem #150: Вышедшие и неинициализированные

В данном Gem описываются некоторые, возможно, неожиданные случаи, когда переменные не всегда корректирует свое значение после присваивания, что нельзя сразу понять из кода.

Gem #151: Определение математических параметров программы

Добавление множества новых утверждений в Ада 2012 может вызвать желание утвердить параметры данных, которые не учитывают возможность возникновения переполнений. В GNAT определены функция переключения и прагма, с помощью которых можно добиться именно такого эффекта.

Gem #152: Определение нового языка в файле проекта

Возможно использовать язык программирования, изначально неизвестный gprbuild. Этого можно добиться, например, определив все характеристике необходимого языка в файле проекта.

Gem #153: Многоядерное прохождение лабиринта, Часть 1

В данном Gem представляется проект «amazing», включенный в примеры компилятора GNAT Pro. Название проекта связано с тем, что его задача – нахождение выхода из лабиринтов (англ. «maze» – лабиринт). Но это – не обычные лабиринты, в которых есть только один выход. Решений может быть множество, например, десятки тысяч. Суть в том, чтобы найти все решения как можно быстрее. Поэтому эти задачи решаются параллельно, с

применением нескольких ЦП и проектирование методом «разделяй и властвуй». В первой публикации данной серии мы представим программу и разъясним подход к разрешению данной проблемы.

Gem #154: Многоядерное прохождение лабиринта, Часть 2

В данной серии публикаций описывается проект параллельного разрешения лабиринтов («amazing»), включенный в примеры GNAT Pro. В первом Gem серии был представлен сам проект и проектировочный подход к параллельному программированию. Второй Gem серии содержит разъяснение основных изменений, которые необходимы для обеспечения оптимальной производительности в многоядерных архитектурах. Данные изменения касаются критической проблемы производительности, которая не была известна во время первого выпуска программы в 1980 г. Они демонстрируют фундаментальные отличия традиционного многопроцессового и современного многоядерного программирования.

Gem #155: Настройка базы данных GPRBuild для работы с новым языком.

Базу данных GPRBuild можно дополнить файлом XML, в котором описаны характеристики нового языка. Это позволит файлу проекта, в котором объявлен данный язык автоматически вызывать компилятор для получения доступа к источникам данного языка и всем необходимым параметрам.

Gem #156: Контроль отображения в GNAT

В компиляторе существует много вариантов генерации и контроля выходных данных. Они часто малоизвестны, так как информация о них спрятана в кипах сложной документации. В данном Gem будут продемонстрированы данные варианты и то, как их можно использовать для контроля исходящих данных из компилятора.

Gem #157: Генерация кода и Gprbuild

Данная серия публикация посвящена тому, как настраивать gprbuild для вызова генераторов кода перед компиляцией самого кода.

Gem #158: GPRinstall– Часть 1

GPRinstall – инструмент, относительно недавно разработанный AdaCore, с помощью которого можно устанавливать проект вне зависимости от того, является он стандартным проектом, проектом библиотек, проектом библиотек агрегатов или стандартным проектом агрегатов. С ним можно забыть как о ручном копировании артефактов, так и о необходимости обеспечения надежной процедуры установки для всех платформ.

Gem #159: GPRinstall– Часть 2

GPRinstall – инструмент, относительно недавно разработанный AdaCore, с помощью которого можно устанавливать проект вне зависимости от того, является он стандартным проектом, проектом библиотек, проектом библиотек агрегатов или стандартным проектом агрегатов. С ним можно забыть как о ручном копировании артефактов, так и о необходимости обеспечения надежной процедуры установки для всех платформ.

Gem #160: Разработка блочных тестов с помощью GNATtest.

Никому не хочется тратить время на тестирование. Это скучно и утомительно: нужно писать средство тестирования, сами тесты, а также подстраивать их под разработку. К счастью, для этого есть GNATtest – небольшой инструмент, входящий в состав набора инструментов GNAT, с помощью которого можно автоматически создавать и поддерживать средства тестирования, что значительно упрощает этот процесс.

Gem #161: До свиданья, и спасибо за ваше внимание и участие!

В последнем Gem мы рассмотрим некоторые ключевые моменты всех публикаций, включая освещенные темы и ваши любимые Gem. В будущем планируется создать новый блог.