Gem #39: Эффективный поток ввода-вывода для регулярных типов (Array Types).

Автор: Pat Rogers, AdaCore

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

Давайте начнем…

Ada имеет понятие “потоков”, которые во многом похожи на другие языки: последовательности элементов, содержащие значения произвольных, возможно, различных типов. Размещение значения в потоке легко с помощью языковых атрибутов потока. Программист просто вызывает подпрограмму атрибута типа и задает поток и значение. Например, чтобы поместить целое значение V в поток s, можно написать следующее:

```
Integer’Write (S, V);
```

Строго говоря, S - это не поток, а скорее значение доступа, обозначающее поток. Подпрограмма Integer'Write преобразует значение V в массив «элементов потока» - по существу массив элементов хранения - и затем помещает их в поток, обозначенный S. Фактически, размещение байтов в потоке выполняется динамически отправка на процедуру, специфичную для представления потока.

Хотя это обсуждение сформулировано с точки зрения размещения значений в потоках, вы должны понимать, что чтение значений из потоков очень похоже на их запись и что применяются те же проблемы эффективности и решения.

Для составных типов, таких как типы массивов или записей, каждое значение компонента индивидуально записывается в поток, используя описанный выше подход. Рассмотрим тип массива «A», определяющий Integer как тип компонента. Версия A'Write по умолчанию вызовет Integer'Write для каждого компонента. Таким образом, каждое значение Integer преобразуется в массив элементов хранения и записывается в поток. Такое поведение, основанное на компонентах, необходимо, потому что программисты могут определять свои собственные версии атрибутов потока и, естественно, ожидают, что они будут вызваны даже тогда, когда рассматриваемые типы используются как типы компонентов в окружающем массиве или типах записей.

Но предположим, что тип массива является структурно просто последовательностью смежных байтов, а тип компонента не имеет определяемого пользователем атрибута потока. В этом случае вызов компонента-специфического атрибута для каждого компонента массива является ненужным и неэффективным.

Например, предположим, что вы работаете с Military-Standard 1553B для передачи значений приложений между удаленными устройствами. В конечном счете, Mil-Std-1553B отправляет и получает 32-словные буферы, где каждое слово представляет собой 16-битное значение без знака. Предположим также, что вы хотите писать и читать эти буферы в потоки и из них. Мы можем переопределить атрибуты потока так, чтобы значение всего буфера было записано непосредственно в поток, вместо того, чтобы писать один буферный компонент за раз.

Тип буфера может быть объявлен следующим образом:

```
type Buffer is array (1..32) of Interfaces.Unsigned_16;
```

Затем можно переопределить атрибуты потока для типа Buffer.

Сначала мы объявим подпрограммы:

```
   procedure Read_Buffer
      (Stream : not null access Ada.Streams.Root_Stream_Type'Class;
       Item   : out Buffer);

   procedure Write_Buffer
      (Stream : not null access Ada.Streams.Root_Stream_Type'Class;
       Item   : in Buffer);
```

Все такие атрибуты потока имеют одинаковые формальные типы параметров, то есть параметр доступа, обозначающий тип корневого потока класса, определенный языком, и тип, который должен быть записан в этот поток или считываться с него.

Затем мы «привязываем» подпрограммы к атрибутам потока для буфера типа, тем самым переопределяя версии по умолчанию:

```
   for Buffer'Read  use Read_Buffer;
   for Buffer'Write use Write_Buffer;
```

Определенный языком тип потока и тип элемента массива объявляются в пакете Ada.Streams:

```
package Ada.Streams is

   type Root_Stream_Type is abstract tagged limited private;

   type Stream_Element is mod 2 ** Standard'Storage_Unit;

   type Stream_Element_Offset is range
     -(2 ** (Standard'Address_Size - 1)) ..
     +(2 ** (Standard'Address_Size - 1)) - 1;

   ...

   type Stream_Element_Array is
      array (Stream_Element_Offset range <>) of aliased Stream_Element;

   procedure Read
     (Stream : in out Root_Stream_Type;
      Item   : out Stream_Element_Array;
      Last   : out Stream_Element_Offset)
   is abstract;

   procedure Write
     (Stream : in out Root_Stream_Type;
      Item   : Stream_Element_Array)
   is abstract;

   ...
end Ada.Streams;
```

Пользовательские программы Read_Buffer и Write_Buffer будут вызывать эти поточно-ориентированные процедуры чтения и записи (через динамическую диспетчеризацию) один раз для всего значения массива Buffer, вместо того, чтобы называть их один раз для компонента массива. Обе процедуры очень похожи, поэтому мы будем опускать тело Read_Buffer ради краткости и показать только реализацию Write_Buffer:

```
   procedure Write_Buffer
      (Stream : not null access Ada.Streams.Root_Stream_Type'Class;
       Item   : in Buffer)
   is
      Item_Size : constant Stream_Element_Offset :=
                     Buffer'Object_Size / Stream_Element'Size;
```

```
      type SEA_Pointer is
         access all Stream_Element_Array (1 .. Item_Size);

      function As_SEA_Pointer is
         new Ada.Unchecked_Conversion (System.Address, SEA_Pointer);
   begin
      Ada.Streams.Write (Stream.all, As_SEA_Pointer (Item'Address).all);
   end Write_Buffer;
```

В приведенном выше примере мы не можем просто преобразовать значение Item, типа Buffer массива, в значение типа Stream_Element_Array, поэтому вместо этого мы работаем с указателями. Мы определяем тип доступа, обозначающий Stream_Element_Array, который является точным размером, с точки зрения Stream_Elements, входящего значения Buffer. Обратите внимание на использование атрибута Buffer'Object_Size в этом вычислении. Этот атрибут дает нам размер объектов типа Buffer, мудрый подход, поскольку, как правило, размер типа может не равняться размеру объектов этого типа. Затем мы можем использовать непроверенное преобразование для преобразования адреса элемента формального параметра в этот тип доступа. Выделение, которое преобразует доступное значение (через .all), дает нам значение типа Stream_Element_Array, которое мы можем передать на вызов Ada.Streams.Write.

Таким образом, мы избегаем обработки каждого компонента буфера типа, вместо этого записывая сразу все значение Buffer. Это гораздо более эффективный подход. Как мы говорили ранее, чтение значений из потоков аналогично написанию значений для них и только отличается очевидными, второстепенными способами. Это верно для использования атрибутов потока по умолчанию, а также для реализации Read_Buffer.

Связанный исходный код отсутствует

Attached Files нет

Об авторе

Пэт Роджерс был профессионалом в области вычислительной техники с 1975 года, в основном работая над приложениями на основе микропроцессора в режиме реального времени на языках Ada, C, C ++ и других языках, включая высокопроизводительные имитаторы полета и системы контроля и сбора данных (SCADA), контролирующие опасные материалы. Впервые узнав язык программирования Ada в 1980 году, он был директором лаборатории Ada9X для совместной программы Advanced Strike Technology для ВВС США, исследователя принципов в распределённых системах и исследовательских проектов отказоустойчивости с использованием Ada для ВВС США и армии, а также помощника директора по исследованиям в NASA Software Engineering Research Center. У него есть B.S. и M.S. степени в области проектирования компьютерных систем и компьютерных наук Университета Хьюстона и доктора философии в информатике из Университета Йорка, Англия. Являясь членом старшего технического персонала AdaCore, он специализируется на поддержке разработчиков программ, которые работают в режиме реального времени и для встроенных систем, создаёт и предоставляет учебные курсы, а также является руководителем проекта и разработчиком плагина GNATbench Eclipse для Ada. Он также имеет черный пояс 3-го Дан в Тэ Квон До и является основателем клуба AdaCore «The Wicked Uncles».

Last Updated: 10/13/2017
Posted on: 6/9/2008

Обсуждение…

2 responses to “Gem #39: Efficient Stream I/O for Array Types”

1. 10 июня 2008 года в 7:17

Кристоф Грин сказал:

Два комментария к этому прекрасному Gem Ada #39:

1. Я понимаю, почему вы используете GNAT-специфический объект `Object_Size`. Однако это не переносится, и мне интересно, почему вы не используете стандартизируемый объект

```
Item_Size: константа Stream_Element_Offset: =

Item'Size / Stream_Element'Size;
```

Я не вижу, как `Item'Size` и `Buffer'Object_Size` могут когда-либо отличаться.

2. Сначала я задавался вопросом, почему вы не использовали напрямую `Item'Access` или `Item'Unchecked_Access`, но, на первый взгляд, я понял, что дал бы неправильный тип указателя. Но тогда - почему бы вам не использовать RM 13.7.2 `System.Address_to_Access_Conversions` вместо `Unchecked_Conversion As_SEA_Pointer`? (Я понимаю, последнее делает то же самое за кулисами, но должна быть какая-то причина, почему этот пакет был предоставлен ARG.)

BTW: Не могли бы вы позаботиться о будущем, чтобы апостроф больше не показывался как «′». Это чрезвычайно угадывает ваши примеры.

2. 10 июня 2008 года в 13:08

Мартин Доуи сказал:

Имеет ли декларация:

```
type Buffer is array (1..32) of Interfaces.Unsigned_16;
```

не требуется условие `'Size`? Или компилятор не мог бы разместить каждый 16-разрядный элемент массива, например, 4-байтную границу?