ББК 32.973
П27 УДК
681.3.06

КОРОЧКИН А.В.
П 27 Ада 95.Введение в программирование. - Киев : Свгг, 1998 -240с.
Рассматриваются концепции и основные средства нового стандарта известного языка программирования Ада, предназначенного для разработки программ широкого назначения, работа которых характеризуется высокой надежностью. Основное внимание уделяется новым конструкциям языка Ада 95 - обьектно-ориентированному программированию, системе иерархических библиотек, защищенным модулям.

Изложение сопровождается большим количеством примеров.

Для студентов, аспирантов и специалистов в области программного обеспечения компьютерных систем.

Корочкин Александр Владимирович

Ада 95. Введение в программирование

ББК 32.973

ISBN 966-952-82-0-8

К о р о ч к и н А. В.

Ада 95. Введение в программирование

Киев. "Свгг"
1998

КРАТКОЕ СОДЕРЖАНИЕ

СОДЕРЖАНИЕ

П Р Е Д И С Л О В И Е

Язык программирования Ада 95 - новый стандарат известного языка Ада (ISO/ IEC 8652 : 1995 (95-02-15), ANSI (95-04-10), FIPS (95-07-10)), разработанного в 1983 году по заказу Министерства Обороны США.

Новый стандарт языка отличает в первую очередь:

- наличие завершенных средств обьектно-ориентированного программирования;
- механизм иерархических библиотек;
- средства синхронизированного доступа к общим данным в виде защищенных модулей.

Ада 95 - первый язык обьектно-ориентированного программирования, прошедший полную международную сертификацию.

Ада 95 - современный язык программирования, сконцентрировавший в себе новейшие достижения в области программирования. Язык имеет универсальную направленность, позволяя решать задачи широкого назначения. Являясь естественным расширением Ады 83 (в языке сохранены все зарезервированные слова и добавлены только 6 новых), Ада 95 сохраняет и умножает свое основное назначение *-разработка больших программных систем, работа которых характеризуется высокой надежностью.*

Содержание книги базируется на курсе лекций, подготовленных автором для студентов старших курсов Национального Технического Университета Украины - "Киевский Политехнический Институт", специализирующихся в области компьютерных систем и сетей (7.091501).

Книга является введением в язык Ада 95 и может быть полезной всем, кто работает в области разработки программного обеспечения компьютерных систем.

Глава! Л Е К С И К А

1.1 Набор символов

В языке множество символов, используемых в тексте программы, основывается на стандарте ISO 10646 BMP и включает :

- строчные и прописные буквы латинского алфавита ;
- цифры;
- символ пробела ;
- специальные символы:

1.2 Лексемы, разделители, ограничители

Для построения минимальных составляющих языка, имеющих смысл (*лексем*), в Аде выделены следующие виды:

- ограничители
- идентификаторы
- числовые литералы
- символьные литералы
- строковые литералы
- примечания.

Ограничители бывают простые и составные. Простой ограничитель - один из специальных символов:

& ( ) * + , . / : ; < > I

Составной ограничитель состоит из пары специальных символов: =>

.. ** := /= >= <= « » o

Идентификаторы используются в качестве имён.

Литералы используются для задания значений определённого типа (числовых, символьных, строковых).

Примечание (комментарий) начинается с двух соседних дефисов (- -) и продолжается до конца строки.

Набор символов языка предназначен для формирования исходного текста программы. Он не влияет на выполнение программы, однако определяет вид программы, удобства при ее чтении и понимании, сопровождении, модификации и в итоге - правильность работы. Поэтому внешнему виду программы следует уделять определенное внимание.

Внешний вид программы задается с помощью :

- пробелов
- отступлений
- выравнивания
- задания длины строки.

Правильное использование пробелов улучшает читаемость программы и контроль за нею. Желательно использование пробелов до и после ограничителей, перед унарными операциями, до и после бинарных операций , после запятых и точек с запятыми.

П Например:

```
X    := Size + Frt_E15( 2.04- E**2);
Vol  := X * Y * Z ;
Sum: = Sum +  Sin (X=> 0.34) + 1.2324 ;
```

Отступление и выравнивание - достаточно известные средства формирования внешнего вида программы, оказывающие влияние на ее понимание через контроль ее структуры. Рекомендуются следующие количественные характеристики при использовании отступлений: два пробела для продолжении строки и три - при выделении группы строк. D Например

```
D14:
   for  i  in   1..10  loop
      S( i): = S (i) + 1 ; end   loop  D14;
```

```
if  Data < 10  then
Res : =  Data -77 end
if;
```

1.3 Идентификаторы

Идентификаторы используются в качестве имён переменных, подпрограмм, пакетов, задач и др. или для зарезервированных слов Ады.

Зарезервированные слова языка Ада 95:

abort	else	new	return
abs	elsif	not	reverse
abstract *	end	null	
accept	entry		select
access	exception		separate
aliased *	exit	of	

all		or.	
and	for	others	tagged *
array	function	out	task
at			terminate
	generic	package	then
begin	goto	pragma	type
body		private	
	if	procedure	
case	in	protected *	until *
constant	is		use
		raise	
declare		range	when
delay	limited	record	while
delta	loop	rem	with
digits		renames	
do	mod	requeue *	xor

(Отмечены слова, появившиеся в Аде-95).

Зарезервированные слова нельзя использовать в качестве идентификаторов, определяемых программистом. Они используются только в контексте, определяемом языком.

Написание идентификаторов определяется следующими правилами:

- идентификатор всегда начинается с буквы
- после первой буквы может следовать любая последовательность букв, цифр и символов подчеркивания
- нельзя использовать несколько стоящих подряд символов подчёркивания, а также символ подчёркивания в конце идентификатора. П

Например:

```
z66     Number     Vector Size          Pool_Index_N14
```

Буквы, используемые в идентификаторах,
могут быть как прописными, так и строчными. Если идентификаторы отличаются только размерами букв в одних и тех же позициях, то они считаются одинаковыми (SIZE, Size, siZe, size).

СОВЕТЫ:

* Выделяйте зарезервированные слова среди других элементов программы, например начинайте их всегда с прописной буквы, а остальные элементы - с заглавной.
* Выбор имен обьектов и типов должен обеспечивать понимание их назначения в программе.
* Используйте нижнее подчеркивание в сложных составных идентификаторах.
* Начинайте идентификаторы с большой буквы.
* Лучше короткие имена, чем аббревиатура.

1.4 Литералы

Литералы как виды лексем служат для явного обозначения некоторого типа и задаются цифрами, буквами и другими символами.

Литералы - это числовой, строковый, символьный литерал, литерал перечисления, литерал **null.**

Числовые литералы подразделяются на два класса: *вещественные* литералы и *целые* литералы. *Вещественный литерал* - это числовой литерал, который включает точку. *Целый литерал* - это числовой литерал без точки.

В языке разрешается задавать числовые литералы в различных системах счисления (двоичной, восьмеричной и др.). Кроме того, разрешается использовать символ подчёркивания при написании чисел.

Десятичный числовой литерал выражается в обычной десятичной системе счисления (по умолчанию основание равно десяти):

```
144   0      26E3          243_368_664     --  целые числа
11.2  0.0    0.4873        _415_926        --  вещественные литералы
11.23E-03    11.05E+10 -- вещественные литералы с порядком
```

Числовые литералы с основанием - это числовые литералы, в которых явно указано основание от двух до шестнадцати:

```
2#1011_1001#      8#472501#      16#1F3#   - -  целые литералы
8#704#E           - -  целый литерал
16#F.4C#E-3       2#1/0011_0001#E4  - - действительный литерал
```

Символьные литералы - это один из графических символов (включая пробел), заключенный между двумя символами апострофа:

```
< В '          4 * »          «>          ' 4 '
```

Строковые литералы образуются из последовательностей графических символов, заключённых между двумя символами кавычки:

"Next _Message"

"W"

Строковый литерал должен помещаться на одной строке, так как он является лексемой. Более длинные последовательности значений формируются операцией конкатенации (&) строковых литералов: "SYMBOLS" & "NEXT" & "LEVEL"

1.5 Операции

В языке определены шесть категорий операций :

- логические операции : and) or | xor
- операции отношения : = | /= | < | <= | > 1 >=
- бинарные операции : + | - | &
- унарные операции : + | -
- мультипликативные операции : * | / | mod | rem | ** | abs | not

1.6 Прагмы

Прагмы используются для задания в тексте программы информации для компилятора:

PRAGMA *Имя* (*Аргументы_Прагмы*) ;

Здесь *Аргумент_Прагмы* - имя или выражение.

Прагмы делятся на прагмы, определенные в языке, и дополнительные прагмы, определяемые реализацией.

Описание прагмы допустимо в определенных местах программы, там где разрешены операции, описания, спецификаторы, альтернативы, варианты, обработчики исключений.

П Примеры:

```
pragma  Priority (10);          pragma List (Off);
pragma  |n_l_ine ( Send);       pragma  Atomic ( Buffer);
```

Прагмы могут иметь различное назначение, например использоваться для управления печатью, указания приоритетов задач, работы с общими переменными, оптимизации и др. Всего в языке определены 39 прагм (Приложение 4) .

* Делайте текст программы ясным и понятным с помощью комментариев.
* Для каждого программного файла делайте заголовок.
* Делайте заголовок для спецификации каждого программного модуля, в котором указывайте назначения модуля, дату его создания (модификации), требуемый обьем памяти и другую необходимую информацию.

ИЗМЕНЕНИЯ:

О Добавлены еще шесть зарезервированных слов: **abstract, alliased, protected, requeue, tagged, until.**

О Расширен набор символов, используемых в тексте программы (8-и битовый на базе ISO-8859 и 16-битовый на базе ISO 10646). О

Изменены правила работы с прагмами.

1.7 Примеры

П Пример оформления заголовка процедуры :

```
Процедура для выполнения операции с векторами
A = ( B * C ) * D
Автор:  Королев С.В.
Кафедра вычислительной техники НТУ "КПИ"
Дата создания:  10.12.97
```

Глава2. Т И П Ы

Понятие *типа* наряду с понятием *объекта* являются фундаментальными в языке. Тип задает множество *значений* и множество *примитивных операций*, которые допускаются над этими значениями.

На основании типа создаётся *объект* (константа или переменная), который ассоциируется с типом. Для объекта тип задаёт:

- множество значений, которые он может принимать;
- множество примитивных операций, которые можно выполнять над ним.

Примитивная операция для типа - это совокупность предопределенных для типа операций и определенных пользователем *примитивных подпрограмм*, аргументы или результат которых имеют данный тип. Примером примитивной подпрограммы является подпрограмма, определяемая пользователем в спецификации пакета для типа, описанного в этой же спецификации.

Язык Ада является языком *строгой типизации*, то есть каждый объект должен быть явно описан и принадлежать определенному типу. Строгая типизация повышает надежность языка, так как позволяет компилятору выявлять существующие и потенциальные ошибки как на этапе компиляции, так и при выполнении программы за счет контроля, связаного с нарушением ограничений типа. Ада обладает развитым механизмом типов, предоставляя пользователю широкие возможности по созданию и использованию типов.

Все типы в языке делятся на *предопределённые* типы и типы, *создаваемые пользователем.*

Типы в языке группируются в *классы*. Специальный синтаксис обеспечивает описание типа соответствующего класса. Предопределенные типы объединяются в несколько классов : класс целых типов, класс дискретных типов, класс ссылочных типов и др. (Рис. *2.1)* .

Все типы делятся на простые и сложные (составные). Простые типы состоят из одной компоненты, составные - из нескольких. Простые типы представлены скалярными типами и *ссылочными* типами **(access types).**

К сложным типам относятся *массивы* **(array types),** *записи* **(record types),** *защищённые* типы **(protected types),** *заданные* типы **(task types).**

Дополнительно в языке используются *приватные (личные)* типы **(private types)** и *ограниченные (лимитированные)* типы **(limited types),**

тэговые типы **(tagged types),** *расширенные* типы и *абстрактные* типы **(abstract types).**

Простые типы включают ссылочные типы и скалярные типы. Скалярные типы включают дискретные типы и вещественные. Дискретные типы в свою очередь состоят из *перечисляемых* **(enumeration)** и *целых* **(integer).**

Вещественные типы состоят из *плавающего* типа **(float type)** и *фиксированного* типа **(fixed type).** Целые, фиксированные и плавающие типы образуют *числовой* тип.

Рис. 2.1

2.1 Атрибуты и дискриминанты

Для каждого типа, используемого в программе (предопределённого и создаваемого пользователем) формируются характеристики, которые можно получить с помощью действий, которые называются операциями над типами. Эти операции получили название *атрибутов* типа.

Атрибут определяет базовую операцию над понятием, задаваемым префиксом:

атрибут : = префикс 'обозначение_атрибу'та;

Определённые в языке атрибуты для каждого типа задают характеристики, которые можно использовать в программе. Например, границы диапазонов массивов, начальный и конечный индексы, адреса переменных в памяти, точность типа, минимальные и максимальные значения, состояние задачи и пр. Атрибуты типов будут детально рассмотрены при описании предопределённых типов и типов, создаваемых пользователем.

П Пример атрибутов:

```
Integer'Last        - -  наибольшее значение типа Integer
Float-Digits        - -  точность типа Float
Fixed'Delta         - -  величина ошибки в описании типа Fixed
Vector'First        - -  нижняя граница индекса массива Vector
PROC.Rez'Count      - -  число вызовов входа Rez в задаче PROC.
```

Полный перечень определённых в языке атрибутов приводится в Приложении.

Для сложных типов в языке введена параметризация типа с помощью специальной компоненты - *дискриминанта.* Дискриминант может быть либо дискретного типа либо ссылочного типа. Он используется для управления размером или структурой объекта. Для объектов защищённого и задачного типов дискриминант может быть использован для инициализации.

П Пример использования дискриминанта:

```
- - дискриминант в записи

type   Stack ( Size :    Stack_Size : = 99) is
   record
```



```
      Pos: Stack_Size; - - независимая от дискриминанта
                       - - компонента
                                  - - зависимая компонента
      Val: String (1..Size);
   end  record;
A :   Stack (400 );                - - значение дискриминанта 400
D :   Stack ;   - - значение дискриминанта задано по умолчанию

- - дискриминант в заданном типе
task   type   Rose ( Name :   Character) ;

TA :   Rose (A ); TB :
Rose ( B );
```

Ш СОВЕТЫ:

* Широко используйте атрибуты типов и обьектов в программе. Ф Получайте информацию о типах с помощью атрибутов типа .
* Используйте механизм дискриминантов для для параметризации типов, а также при инициализации обьектов задачного и защищенного типов;

2.2 Простые типы

2.2.1 Ссылочный тип

Ссылочные типы (**access type**) обеспечивают в языке непрямой доступ к обьекту или подпрограмме. ' Существуют два вида ссылочных типов :

- ссылочные на обьекты ;
- ссылочные на подпрограммы .

Ссылочный тип определяется следующим образом: Для обьектов :

TYPE *Имя_Ссылочного_Типа* **IS ACCESS [ALL | CONSTANT]** (*Указанш_Типа_Или_Подтипа [Ограничение]*) ;

Для подпрограмм:

TYPE *Имя Ссылочного_Типа* **IS ACCESS**

[PROTECTED] *Описание_Подпрограммы* ;

Здесь *Ограничение* - это дискриминант или ограничение индекса. П
Пример:

```
                is access integer; t
                is access Word;  y
pe Work type Byte

type Order is access all Store'Class; type Exed
is access constant integer; Send is access
procedure (Z : in integer);
```

Для каждого ссылочного типа среди множества его значений обязательно присутствует значение **null.** Значение **null** присваивается любому указателю после его объявления. Указатель со значением **null** не указывает ни на какой объект:

Для создания динамических объектов используется генератор new, который также формирует ссылку на созданный объект.

П Например:

```
X1 : Work : = new Integer; X2 :
Work : = new Integer'(10);
```

Помимо ссылки на объект возможно обращение к значениям объекта. Для этого используется постфикс " all ":

```
XLall : = 10;                XLAall : = XLAall + 4;
```

Для объектов ссылочного типа используются операции присваивания, проверки на равенство и неравенство.

П Пример:

```
type Matr is array (1 .. 10, 1.. 15) of integer;
type AC_Matr is access Matr;

- - объекты ссылочного типа
MA: AC_Matr new Matr;
MB: AC_Matr new Matr (1 .. 10 => (1 .. 15 => 0));

- - операции над объектами ссылочного типа
MA.all: = MB.all;
```



```
MB.all( 2 ,6 ): = 38;      MA( 2, 2 ): = 11;
```

Механизм работы с ссылочными типами в Аде 95 значительно расширен, обеспечивая более гибкий доступ к объектам. Новые атрибуты **'Access и 'Uncheked_Access** используются для создания значений ссылочных типов, определяющих обьект или подпрограмму:

```
X : = Object' Acess ;   Z : = Subprogram ' Access ;
```

Использование зарезервированных слов **all** и **constant** в описании ссылочных типов позволяет определять универсальный ссылочный тип , расширяющий возможности работы с объектами ссылочного типа:

```
type Ira is access all integer; type
Ото is access constant real;
```

Использование универсальных ссылочных типов основано на применении к ним генератора new, атрибутов **Access и 'Unchek-ed_Access** и , а также конструкции **aliased** (еще одно новое зарезервированное слово языка) :

```
Stone
Big_Stone Ira;
Wall      aliased integer;
          Big_Stone' access ;
```

Переменной типа Ira может быть присвоено значение адреса любой переменной типа **Integer** , если она помечена как all. Чтение и изменение переменной Big_Stone возможно с помощью ссылочной переменной Stone и атрибута **'access.**

Использование в описании универсального ссылочного типа слова **Constant** позволяет доступ только по чтению к переменным данного ссылочного типа (в нашем примере это касается типа Ото):

```
Water      : Ото;
Green_Water : aliased Water: = 486 ;
Lake       : Green_Water' Access ;
```

2.2.2 Скалярные типы

Скалярные типы (дискретные и вещественные) не имеют компонент, то есть являются типами с простыми значениями. *Дискретные* скаляр-

ные типы - это перечисляемые и целые типы. *Числовые* типы - это целые и вещественные типы. Для любого скалярного типа T имеются атрибуты:

T'FIRST; наименьшее значение типа T наибольшее
T'LAST; значение типа T

Перечисляемый тип

Множество значений перечисляемого типа задаётся явным перечне лением этих значений:

TYPE *Имя_Типа* IS (*a,,a:, ... ,a,,*) ;

Здесь литералы a; могут быть либо идентификаторами, либо символьными литералами. При этом a,.|< a_i <a_{i+1}. П Примеры:

```
type   Work_Week   is   (MON,  TUE,  WED,  THU, FRI);
type   Color       is   (Blue, Yellow);

type   Start       is   (On, Off);
type   TaskjState  is   (RUN, WAIT, READY);
```

Атрибуты перечисляемого типа (применимы также к целому типу):

T'Pos (X) Номер позиции X в её описании
T'Succ (X) Следующий за X элемент в типе T
T'Pred (X **)** Предшествующий элемент
T'Val (N) Элемент типа T, стоящий в позиции с номером N

Над объектами перечисляемых типов определены следующие операции:
Отношения: < < > >
Проверки принадлежности: In Not In

СОВЕТЫ :

* Используйте перечисляемые типы вместо числовых .

Числовые типы

Целый тип, а также плавающий и фиксированный типы образуют числовой тип. Над объектами числового типа выполняются арифметические операции.

Целый тип представляет множество значений целых чисел и операции над ними. Фиксированный тип представляет вещественные числа в формате с фиксированной запятой с указанием абсолютной точности представления. Плавающий тип представляет вещественные числа в формате с плавающей запятой с указанием относительной точности представления.

Числовые типы могут быть как предопределёнными типами (**Integer, Float, Duration**), так и типами, определёнными пользователем.

Более подробно числовые типы будут рассмотрены дальше в Главах 4 и 5.

2.3 Составные типы

2.3.1 Массивы

Массив в Аде рассматривается как составной объект, содержащий компоненты (элементы массива) одного и того же типа. Доступ к элементам массива выполняется с помощью *индексов.*

Различают *ограниченные* и *неограниченные* массивы. Для ограниченных массивов границы определяются во время описания объекта или во время описания индексируемого типа. Для неограниченных массивов границы не определяются во время описания. Это делается позже:

- при описании типов, использующих неограниченный индексируемый тип;
- при описании объектов;
- во время передачи параметров.

Неограниченные массивы позволяют передавать в подпрограммы фактические параметры в виде массивов различной длины.

Описание ограниченных массивов:

TYPE *Имя_Типа_Массива* **IS ARRAY**
(Дискретный_Диапазон) OF
Подтип_Компонентов_Массива;

Массив пуст, если хотя бы один из индексов определяется пустым диапазоном. П Пример:

```
type Stack  is array (1 .. 99)      of Element;
type Set    is array (Color )             of integer;

type Vector   is array (1 .. 100)          of float;
type Matrix   is array ( 1 .. 15, 1 .. 10 ) Of float;
type Matrix_New is array ( 1 .. 15)        of Vector;
```

Описание неограниченных массивов: **TYPE** *Имя_Типа_Массива* **IS ARRAY (T RANGE o) OF** *Подтип_Компонентов_Массива;*

```
type Vector_NN is array (integer range <>) of float;
type Vector_NN is array (integer range <>, integer range <> ) of float;
```

Здесь T - имя типа или подтипа; o - бокс, который обозначает, что границы неопределены и будут указаны позже.

Определение объектов, использующих неограниченный тип Vector_NN и Matrix _54, происходит с указанием границ диапазона:

```
type Matrix _54 is array
```

X, Z, S Vector_NN (1 .. 10); Vector_NN (-99 .. 99); Matrix _54 (1 .. 10, 1 .. N);

СОВЕТЫ:

* Избегайте анонимных типов при описании массивов;

Элементы массива, отрезки и агрегаты

Отрезок одномерного массива M задаётся в виде:

M *(Дискретный_Диапазон)*

Отрезки позволяют оперировать с группой элементов массива как с единым объектом, упрощая написание программы. Например, отрезок X(3 . . 7) указывает пять последовательных элементов массива X, при этом копирование этих элементов не производится:

При использовании механизма отрезков для двухмерных массивов их следует описывать в виде "вектор векторов", после чего для них можно создавать и использовать отрезки, как для типа Matrix_New, описанного выше. Обращение к элементам массива A типа Matrix_New выполняется в форме, отличной от обращения к обычному двумерному массиву:

```
A(i)(j)   := 3.14159;     X(3 .. 7) : = Z( 10
A( 1 .. 5) : = Teo;       Order : = Astra ( I * 4   14);
                                                    ( I + 1) * 4 ) ;
```

Агрегат массива - это значения массива, которые конструируются непосредственно из значений компонент. Используются для присваивания значений массивам.

Тип агрегата задаётся путем указания типа или подтипа: T *'агрегат*

Если тип не задан явно, то он определяется из контекста.

П Пример:

```
(1 .. 25 => 4.557);  Vector ( 1 | 23 = 5,  others => 0 );
Z: = (1 |7.. 10=> 4.2,    9=>3.14);
```

Ш СОВЕТЫ:

- Используйте агрегаты только для случаев работы с обычным порядком аргументов. > Используйте только простые агрегаты.
- Отдавайте предпочтение отрезкам при работе с частями массивов вместо циклов.

Атрибуты массивов

Для объектов индексируемого типа или ограниченного индексируемого подтипа M (массива) определены следующие атрибуты:

- **M'FIRST** Нижняя граница первого индекса; тоже самое, что и **M'FIRST(N)**
- **M'FIRST(N)** Нижняя граница N-го индекса
- **M'LAST** Верхняя граница первого индекса; тоже самое,

что **и M'LAST(N)**

- **M'LAST(N)** Верхняя граница N-го индекса
- **M'LENGTH** Число элементов первого измерения
- **M'LENGTH(N)** Число элементов N-го измерения
- **M'RANGE** Подтип **M'FIRST .. M'LAST,** соответствующий правильным значениями для первого индекса
- **M'RANGE(N)** Подтип **M'FIRST.M'LAST,** соответствующий правильным значениям N-го индекса

2.3.2 Записи

Запись - составной объект, компоненты которого имеют имена и могут иметь различный тип. Описание типа записи должно иметь следующую форму:

```
TYPE Имя_Зашси                    IS
     RECORD
          Имя_Компоненты_1: Тип_Компоненты_1;
          Имя_Компоненты_1: Тип_Компоненты_1;

          Имя_Компоненты_1: Тип_Компоненты_1;
END  RECORD;
```

Тип записи может содержать любое число компонентов. П
Пример описания типа записи:

```
type  Complex_Numbers is record X:
   float; Y:  float; end  record;
```

```
type  Data  is
    record
      V:  Vector; M
      : Matrix; end
      record;
```

- - дискриминант в записи

```
                                  is
type  Massiv (N, M : Positive)
    record
       V  :  array (1 .. N ) of    integer;
       W  :  array (1 .. N, 1 .. M ) of  integer; end
    record ;
```

```
Object:  Massiv (5, 10 );
```


К объектам типа запись можно обращаться как к единому целому, а также можно обращаться к каждой компоненте записи отдельно:

```
  Z2, Sqr  :  Complex_Numbers;          A, B          Data;

Z2.X :=   1.23; Sqr   Z2.Y  : =      - 0.23 ; B . V ;
:=   Z2;              A . V   : =
```

---

iUJ **СОВЕТЫ:**

* Используйте записи с дискриминантом вместо массивов с уточняемыми границами;
* Используйте записи для обьединения разнородных (гетерогенных) связанных данных;
* Для больших и сложных записей используйте разбиение их на более простые подзаписи;

2.4 Дополнительные типы

Дополнительные типы предназначены для реализации в языке средств объектно-ориентированного программирования; абстракции данных и процедур, программирования параллельных процессов и решения задачи взаимного исключения.

Защищённый тип (**protected type**) используется для организации синхронизированного доступа к общим переменным с помощью *защищённых операций.* При этом автоматически обеспечивается взаимное исключение.

Задачный тип **(task type**) предназначен для описания объектов, являющихся задачами (**task**), выполнение которых может быть организовано параллельно.

Тэговый тип (**tagged type**) используется в типе запись. Запись, помеченная тэгом, может быть расширена, реализуя динамический полиморфизм.

Абстрактный тип (**abstract type)** - расширение тэгового типа для случая, когда объекты типа не описываются. С абстрактными типами в языке связано понятие *абстрактной подпрограммы.*

Тэговые типы, расширенные типы и абстрактные типы обеспечивают в Аде реализацию парадигм *объектно-ориентированного программирования.*

Приватные (личные) типы (**private type)** и ограниченные (лимитированные) типы (**limited type)** используются в пакетах для абстрагирования путём введения ограничений на множество допустимых для типа операций. В Аде некоторые типы изначально являются приватными, например, задачный тип рассматривается как ограниченный приватный.

Ш СОВЕТЫ:

* Для организации больших структур данных используйте расширение типов;
* Используйте все имеющиеся в языке средства обьектно-ориентированного программирования.

ИЗМЕНЕНИЯ:

О Записи, защищенные и задачные типы могут быть параметризированы с помощью дискриминантов. О Новые типы: **protected, integer modular, tagged, abstract.**

2.5 Примеры

П Примеры типов :.

Перечисляемый тип:

```
type Week is ( Mon, Tue, Wed, Thu, Fri, Sat, Sun );
```

Целый тип:

```
type Int_My is integer range 1 .. 25;
```

Фиксированный тип (тип с фиксированной запятой):

```
type Size is delta 0.001 range 0.0.. 155.0;
```


Плавающий тип (тип с плавающей запятой):

```
type Weight is digits 8 range 0.5.. 25.0;
```

Ссылочный тип:

```
type TW is access Weight;
```

Массивы:

```
type Vector is array (1..50) of integer;
```

```
type Coordinate is
  record
      X: integer;
      Y: fixed; end
  record;
```

Приватный тип:

```
package Deck is
   type Elem is private;
   procedure Sum (X, Y : in Elem; Z : out Elem);
private
   type Elem is integer range -10 .. 10; end
Deck;
```

Задачный тип:

```
task type AVT is
   entry Dv(X: in Float); end
AVT;
```

Защищённый тип:

```
protected type Buffer is
  entry Write (Z: in Elem);
```

```
    procedure Read (X: out Elem);
private
    Pool: Vector;
end Buffer;
```

Тэговый тип:

```
type Dogs is tagged
   record
   Name : Dog_Name; Weight
   : Dog_Weight; end record;
```

Расширенный тип:

```
type Xdogs is new Dogs with
   record
      Age : Dog_Age; end
   record;
```

Абстрактный тип:

```
type Lot_X is abstract tagged private;
```


Глава3 ПРЕДОПРЕДЕЛЕННЫЕ ТИПЫ

В Аде имеется набор *предопределённых* типов, готовых для непосредственного использования или для построения на их основе новых *(производных)* типов *и подтипов.*

Предопределенные типы **Boolean, Integer, Character, String, Float** и операции для них описаны в пакете **Standard** (Приложение 1).

3.1 Предопределённый логический тип

Предопределённый логический тип **Boolean** содержит только два литерала **False и True.** Для создания объекта логического типа используется описание вида:

Имя_переменной: **BOOLEAN;**

Объект предопределённого логического типа может принимать только значения **False и True.** П Например:

```
Flag : Boolean;
T    : constant Boolean : = False;   - - логическая константа
Full : Boolean : = True;             - - начальное значение
```

Над объектами предопределённого логического типа определены следующие операции:

операции отношения

логические операции

AND OR XOR NOT

Результаты всех перечисленных операций имеют тип **Boolean.**

3.2 Предопределённый символьный тип

Предопределённый тип **Character** является символьным типом, множество значений которого задают символы кода ASCII. Для

создания объектов предопределённого символьного типа используется описание вида:

Имя_переменной: **CHARACTER;**

Символьные литералы - это любой из *95* графических символов, заключённых между двумя символами апострофа, включая пробел.

Над объектами типа Character допускаются операции, определённые для типа **Boolean.**

П Пример:

```
X, Y      :  character;
West      :  constant  Character: =  'WEST'; - - символьная
                                              - - константа
Zet       :  Character: =  'S';        - - задание начального
                                         - - значения
```

Строки символов

В языке для описания строк символов имеется предопределённый тип **String,** задающий одномерный массив, компоненты которого имеют предопределённый тип **Character:**

```
type  STRING  is array  ( POSITIVE range <>)   of Character;
```

Предопределённый тип **Positive** описывается как **subtype**

```
POSITIVE  is   integer    range   1 ..  integer'last;
```

Так как при описании типа **String** диапазон границ не определяется (неограниченный массив), то при создании объектов этого типа или подтипов необходимо эти границы указывать:

```
A                 :  String ( 1 ..10);
Message           :  String ( 1  .. 6 ) : =  'ABCD_EF';
subtype   Buffer  is  String (1 .. 99);
```

Операции над строками символов включают следующие операции:

ф < > < >

а также операцию конкатенации &. Кроме этих операций в языке для работы со строками символов имеются дополнительно подпрограммы **Pos, Substr, Delete.**

3.3 Предопределённый целый тип

В Аде предопределён целый тип **Integer.** Наибольшее и наименьшее значение типа определяются конкретной реализацией и задаются значениями констант **Max_lnt и Min_lnt** из пакета **System.** Создание объекта целого типа выполняется с помощью конструкции:

Имя_переменной : **INTEGER;**

Например:

```
Size    :  Integer;
A, B, C :  Integer;
Vol     :  Constant  Integer : = 25',    - - целая константа
X       :  Integer := 10;    -- задание начального значения
```

Для целого типа разрешены следующие операции (в порядке убывания их приоритетов):

операции высшего приоритета (результат типа **Integer) ABS** (абсолютная величина), ** (возведение в степень)

мультипликативные операции (результата типа **Integer)** *, /, MOD (вычет по модулю), REM (остаток)

унарные и бинарные аддитивные операции (результат типа **Integer)**

операции принадлежности (результат типа **Boolean) In Not In**

операции отношения (результат типа **Boolean)**

Ф < < >

Ограничение значений, которые могут принимать объекты целого типа, выполняется с помощью конструкции **RANGE L .. R,** называемой *ограничением диапазона:*

```
Number  : Integer  range 1  . 10;
Cent    : Integer  range 1  , 99 : = 35;
```

Для целых десятичных литералов разрешается использовать различные системы счисления:

```
десятичную          - - 3     564     5E+3     456_77 356
двоичную            --2#11001022#
восьмеричную        --8#417#
шестнадцатеричную   --16#6AOF#
```

Для целого типа также предопределены подтипы: **Short_lnteger** (короткий целый **)** и **Long_lnteger** (длинный целый), которые определяют более узкие и более широкий диапазоны целых чисел, чем тип **Integer.** Кроме того, можно использовать подтипы **Natural и Positive,** описанные в пакете **Standard:**

```
subtype NATURAL is integer   range 0.. integer'last;
subtype POSITIVE is integer  range 1 .. integer'last;
```

Атрибуты целого типа и объектов целого типа:

- **T'LAST** - - наибольшее значение типа Т,
- **T'FIRST** - - наименьшее значение типа Т.

3.4 Предопределённый плавающий тип

Предопределённый плавающий тип в языке задаётся описанием вида:

Имя_переменной : **FLOAT;**

Объекты типа **Float** могут принимать значения из диапазона, границы которого определяются конкретной реализацией. Изменение диапазона точности плавающего типа используется в предопределённых подтипах **Short_Float и Long_Float.**

П Примеры:

```
Rez   Float;
X     constant  Float. =  1.289;
Temp  Float :=  1.02567E-02;
```


Операции над объектами предопределённого типа **Float:**

```
ABS          **
+   ,   -    (унарные операции)
+   ,   -    (бинарные операции)
     Ф   <   >   <   >
In           Not In
```

Вещественные литералы, в отличие от целых, должны содержать десятичную точку:

0.0 0.1 12346.89 2.00E-4 2.234E+03 10000000.00 Кроме того, вещественные литералы разрешается представлять в других системах счисления:

```
2#1011.1101#      2#1010.001#E-2
8#1074.22#        8#6517720.563#E+12
16#0.12765A#      16#F01B.4576#E-03
```

П Пример:

Использование в процедуре предопределенных типов :

```
          STORE
procedure is
   A, B X Boolean;
   Numb   Integer: = 25 ;
   F      Shortjnteger;
TOP       Character;
Z1.Z2     Constant Float:          =  1.45E-01;
begin     Float;
          - -  разрешено использование всех операций над
          объектами
        - - предопределённых типов и подтипов,
        - - использованных в описательной части процедуры

Z  =  Z 2 + TOP; =  True;
1  =  X  + 140 * 2; = ' R ' ;
A
X  end   STORE;
```

Глава4 ТИПЫ, ОПРЕДЕЛЯЕМЫЕ ПОЛЬЗОВАТЕЛЕМ

В Аде, наряду с предопределёнными типами, пользователю предоставляется широкие возможности по созданию собственных типов с различными свойствами, необходимых для решения конкретной задачи. .При создании собственных типов пользователь может либо использовать уже существующие типы (в том числе и предопределённые), создавая на их основе *подтипы* и *производные* типы, либо создавать оригинальные новые типы.

Общая форма определения типа:

TYPE *Имя_типа* IS *Определение_типа;*

4.1 Подтипы

Подтип (subtype) не создаёт новый тип, а вводит *ограничение* на существующий (базовый) тип. При этом подтип сохраняет все свойства, присущие базовому типу, в том числе множества допустимых операций, и совместим с базовым типом, то есть не требует явных преобразований между базовым типом и подтипом.

Описание подтипа имеет следующий вид:

SUBTYPE *Имя_Подтипа* IS
Имя_Базового_Типа [Ограничение];

Ограничения, как правило, связаны с элементами типа *(диапазона, индекса, точности* и *дискриминанта)* и зависят от вида базового типа.

При работе с подтипом происходит постоянный контроль ограничений, введённых при описании подтипа, и возбуждение исключений в случае их нарушений, что повышает надёжность программы.

П Примеры подтипов:

```
subtype Opel is Cars;
subtype Smalljnteger is integer range -100 .. 100 ;
subtype Vector_ 10 is Vector (1 .. 10);

subtype Amper is float digits 7 range 0.0 .. 200.0;
```



```
subtype Volt is fixed Power delta 0.001 ;
```

ffl **СОВЕТЫ:**

» Ограничивайте диапазоны подтипов настолько, насколько это возможно; * Используйте подтипы для улучшения понимания (чтения) программы;

4.2 Производные типы и производные классы.

Производный тип является новым типом, создаваемым на основе *родительского* типа. Он сохраняет множество значений и операций родительского типа. Объявление производного типа осуществляется следующим образом:

TYPE *Имя_Производного_типа* IS NEW
Имя_родительского_типа [Ограничения];

Ограничения, как и в случае подтипа, могут быть четырёх видов рассмотренных выше. П Примеры производных типов:

```
type Wigth is new Integer range 1 .. 77;
type Sum   is new Float;
type Vec_16 is new Vector100(1 .. 16);
```

Производный тип наследует множество значений родительского типа. Это множество может быть ограничено (если имеется *Ограничение* в описании производного типа) или расширено (если родительский тип описан как тэговый).

Производный тип наследует предопределенные операции и примитивные подпрограммы от родительского типа.

Для производных типов в языке введено понятие *производных классов*. Производный класс для типа T (класс типа T) - это множество типов, содержащее сам тип T (корневой тип класса) и все типы, производные прямо или косвенно (непрямо) от типа T, а также соответствующие *универсальные типы* и типы *широкого класса.*

Каждый тип в языке может быть одним из следующих : • специфическим типом (specific type) _• типом широкого класса (class-wide type)

• универсальным типом (universal type).

Специфический тип определяется при описании типа, описании формального типа или при описании типа в обьявлении обьекта. *Тип широкого класса* определяется для каждого производного класса, корнем которого является тэговый тип. *Универсальный тип* определяется для числовых классов **(integer, real, fixed).**

Типы широкого класса и универсальный тип включают все типы в этом классе, специфический тип - только себя самого.

Типы широкого класса более подробно будут рассмотрены в главе 16.

Ш СОВЕТЫ :

* Используйте производные типы и подтипы совместно.

* Используйте производные типы вместо расширяемых, если не добавляете новые компоненты к типу.

4.3 Эквивалентность типов

В Аде используется именная эквивалентность типов. То есть объекты являются *эквивалентными* тогда и только тогда, когда они описаны с помощью одного и того же идентификатора типа.

П Например:

```
A, B  : Vec16;
C     : Vec16;    -- объекты A, B и C имеют одинаковый тип

X, Y  : array (1 .. 20) of Float; - - все три объекта имеют разный Z
:  array (1 .. 20) of Float; - - (анонимный) тип
```

Преобразование типов осуществляется как преобразование выражений по следующему правилу:

Имя_типа (Выражение);

Преобразования разрешены только для числовых производных и индексируемых типов. При этом должны соблюдаться определённые требования, такие как наличие общего родительского типа для производных типов и одинаковая размерность и тип компонент для индексируемых типов.

П Например:

```
X: integer; Z :
float;

X : ч  integer ( Z + 2.33 );
Z : =  float ( X ) + Z / 4.235E-03;
```

Кроме того, возможно и обратное преобразование типов. В Аде не допускаются неявные преобразования типов, так как они делают программы трудными для понимания.

Переходы от подтипов к базовому типу и наоборот не являются преобразованием типа, так как подтип не вводит новый тип.

4.4 Целые типы

Описание целого типа, создаваемого пользователем, задаётся ограничением диапазона:

TYPE *Имя_типа* **IS RANGE L .. R;**

В Аде 95 наряду с обычными целыми типами (signed integer) появились *модульные* целые (modular integer). Для модульных целых используется циклическая арифметика:

TYPE *Имя_Типа* **IS MOD** *Основание;*

Диапазон целого типа определяется границами величин L и R или основанием для модульных типов. При этом создаётся *новый* целый тип, который характеризуется именем типа и диапазоном значений целых чисел.

Производный целый тип описывается как

TYPE *Имя_тип* **IS** NEW *Целый_тип;*

Подтип целого типа:

SUBTYPE *Имя_типа* **IS** *Целый_тип* **RANGE L .. R;**

П Например:

```
type  Dec type      is  range   1 ..10;
Degree type         is  range - 100 .. 100;
Rose type  Band     is  range   1 .. N; is   mod   256 ;   - -
subtype  N23       mod  2 * * 8
type Resolt          is   Dec  range  2.. 6;              - - подтип
                    is  new  Integer  range  -33.. 56;

A, B, C   Dec;
X22       Degree: = 25;
DS        constant  Rose: = 345;
```

4.5 Вещественные типы

Вещественные типы, создаваемые пользователем, определяются задаваемой точностью и спецификатором ограничения точности.

4.5.1 Плавающий тип

Для типов с плавающей точкой (плавающих типов) объявление типа имеет вид:

TYPE *Имя_типа* **IS DIGITS** *Статическое_Целое_Выражение* **[RANGE L..RJ;**

Здесь **DIGITS** *Статическое_Целое_Выражение* задаёт погрешность представления, **RANGE L..R** - уточнение диапазона значений типа, величина L определяет нижнюю границу диапазона, R - верхнюю границу.

```
type  Higth  is  digits      9;
type  Wigth  is  digits     11   range   0.0 .. 1500.0;
type  'Coef  is  digits      8   range  -1.0E-10 .. 1.0E10;
```

Максимальное число цифр, которое может быть задано в погрешности представления (определении точности), определяется именованным числом **System.MaxJDigits.**

Для плавающих типов и подтипов определены следующие атрибуты:

- **T'BASE** - - базовый тип
- **T'FIRST** - - значения нижней границы Т

- **T'LAST** - - значения верхней границы Т
- **T'SIZE** - - атрибут представления (число битов, отводимых в - - памяти для размещения объектов типа Т).

Группа атрибутов **T'DIGITS, T'MANTISSA, T'EPSILON, TEMAX, T'SMALL, T'LARGE** вырабатывает различные дополнительные характеристики, такие как, например, число десятичных цифр мантиссы, наибольшее значение порядка и др.

Атрибуты **T'SAFEJTMAX, T'SAFE_SMALL, T'SAFE_LARGE** вырабатывают характеристики хранимых чисел.

Для объекта А плавающего типа определены атрибуты **A'ADRESS**- адрес первого кванта памяти, отводимого под А, **A'SIZE** - число битов, отводимых в памяти для размещения объекта.

Для каждого плавающего типа также определены машинно-зависимые атрибуты.

4.5.2 Фиксированный тип

Для типов с фиксированной точкой (фиксированных типов) объявления типа имеют вид:

TYPE *Имя_тша* **IS DELTA** *Статическое_Целое_Выражение* **RANGE L .. R;**

Здесь **DELTA** *Статическое_Целое_Выражение* задаёт точность фиксированного типа , а **RANGE L .. R** - уточняет диапазон. П Пример:

```
type  Precision  is  delta   0.0001; 0.005
type  Amper      is  delta   range   0.0.. 150.0
```

Кроме атрибутов **T'BASE, T'FIRST, T'LAST** (см. 5.5.1) имеются специальные атрибуты, связанные с характеристиками чисел с фиксированной точкой: **T'DELTA, T'MANTISSA, T'SMALL, T'LARGE, T'FORE, T'AFT,** а также **T'SAFE_ SMALL, T'SAFE_LARGE.**

JQ СОВЕТЫ:

* Ограничивайте диапазоны скалярных типов настолько, насколько это возможно.

* Используйте подтипы,так как они улучшают чтение и понимание

программы.

* Совмещайте использование подтипов и производных типов.
* Используйте перечисляемые типы вместо числовых кодов.

4 .6 Примеры

Пусть для работы в программе необходимы четыре типа, задающих одномерные массивы (вектора), элементы которого имеют тип Elem. Возможны различные варианты определения таких типов:

1. Через определение новых типов.

```
Type  VectorlO    is  array  (1 .. 10)    of  Elem;
Type  VectortOO   is  array  (1 .. 100)   of  Elem;
Type  VectorN     is  array  (1 .. N)     of  Elem;
Type  VectorH     is  array  (1 .. H )    of  Elem;
```

При совместном использовании обьектов этих типов необходимо выполнить явное преобразование типов, так как они не являются эквивалентными.

2. Через использование анонимных типов.

```
VA :  array   (1..  10 )   of   Elem ;
VB:   array   (1..  100)   of   Elem ;
VC:   array   (1 -  N)     of   Elem ;
VD:   array   (1 -  H)     of   Elem;
```

Здесь неявно при объявлении обьектов созданы типы (анонимные). Вследствие этого совместное использование объявленных обьектов невозможно, так как для анонимных типов нельзя выполнить явное преобразование типов.

3. Через производные типы.

В этом случае задается родительский тип

```
Type  Vector  is array (1.. 1000) of Elem;
```


на основании которого путем введения ограничений индекса создаются производные типы:

```
Type  VectorlO    is  new   Vector( 1 .. 10);
Type  Vector 100  is  new   Vector(1 ..100);
Type  VectorN     is  new   Vector( 1 .. N );
Type  VectorH     is  new   Vector( 1 .. H );
```

Для обьектов этих типов сохраняются все свойства родительского типа (множество значений с учетом введенных ограничений и множество операций). При совместном использовании обьектов этих типов необходимо выполнять явные допустимые преобразования,

4. Через использование подтипов. Определяется базовый тип (создается или берется уже готовый):

```
Type  Vectors  is  array (integer  range< > )  of  Elem;
```

Создаются подтипы на основе базового типа

```
Subype  VectoMO    is   Vectors( 1 .. 10);
Subype  VectoMOO   is   Vectors( 1 .. 100);
Subype  Vec_N      is   Vectors( 1 .. N );
Subype  VecJH      is   Vectors( 1 .. H );
```

При совместном использовании обьектов этих подтипов преобразования типов не требуется, компилятор лишь осуществляет контроль за соблюдением введенных в подтип ограничений,

Глава5. О П Е Р А Т О Р Ы

В языке Ада имеются два вида операторов: *простые* и *составные*. Простой оператор не содержит в себе других операторов; составной оператор содержит в своём составе другие операторы.

Любой оператор определяет некоторое действие, осуществление которого называется *выполнением* оператора. Последовательность операторов задаёт действия, которые необходимо выполнить над данными.

Простые операторы:

- пустой оператор — Null
- оператор присваивания
- оператор выхода — **Exit**
- оператор перехода — **Goto**
- оператор задержки — **Delay**
- оператор возбуждения — **Raise**
- оператор возврата — **Return**
- оператор вызова подпрограммы
- оператор вызова входа задачи
- оператор прекращения — **Abort**

Составные операторы: условный оператор — **If .. end if**
оператор выбора оператор цикла оператор — **Case Loop**
принятия оператор блока оператор отбора. — **Accept**

Некоторые операторы разрешается помечать с помощью размещаемый перед ним (*именованные* *имени,* операторы). D Например: — **Select**

- - именованный цикл

UFO:

```
for i in 1.. 10 loop
    Res: = Res + Matr( i);
end loop UFO;
```


Некоторые операторы Ады подробно рассмотрены в главах, с которыми они связаны:

- операторы вызова процедуры и выхода — в главе "Подпрограммы";
- операторы вызова входа, **Requeue -** операторы, **Delay, Accept, Select и Abort —** в главе "Задачи";

Ш СОВЕТЫ:

* Начинайте каждый оператор или описание с новой строки.
* Не размещайте больше одного оператора в строке.
* Используйте размещение операторов с отступлениями. »
Не размещайте сложные операторы в одной строке.
* Помечайте end оператор везде, где это возможно.

5.1 Пустой оператор

Пустой оператор является простейшим оператором языка:

Пустой_оператор : = NULL;

Выполнение пустого оператора заключается в переходе к выполнению следующего оператора. Используется в случаях, когда по синтаксису языка Ада требуется присутствие хотя бы одного оператора, а в программе не требуется выполнения действий. Например, в процедуре, осуществляющей запуск описанных в ней задач.

5.2 Операторы присваивания

Служат для замены значения переменной новым значением, которое определяется выражением. Переменная в левой части оператора присваивания и выражение в его правой части должны быть одного типа, при этом тип не должен быть лимитируемым личным **(limited private).**

Общий вид оператора присваивания:

Имя_переменной : = Выражение;

При выполнении оператора присваивания сначала вычисляется имя переменной и выражение, а затем значение выражения становится новым значением переменной. Значение выражения должно удовлетворять всем ограничениям, наложенным на тип переменной.

Если существующие ограничения не выполняются, то возбуждается исключение CONSTRAINT_ERROR и значение переменной не изменяется. D Примеры:

```
X             =          10;
A(Г)          =          B(I) +C(I);
Self          =          Sum(X.Y);
Discriminant  =          B * B - 4.0 * A * C
```

П Пример проверки ограничений:

```
I, J     : Integer range   10          50;
К, L     : Integer range   10          100;
I    : =  J;  - - одинаковые диапазоны
I    : =  L;  - - при L > 50 возбуждается исключение
```

Если тип выражения и тип переменной не совпадают, то необходимо производить *явное* преобразование типов:

```
Xin   integer; real;
Zero
      Xin  : =   integer( Zero + 3.14 );
```

При преобразовании массивов в операторе присваивания разрешается присваивание покомпонентно, отрезками и целиком:

```
A, B, C Vector;


A(5)      :=  B(6) + C
C(2..7)   :=  B(3..8);
B         :=  C;
```

Щ СОВЕТЫ:

* Минимизируйте глубину выражениий.
* Используйте механизм отрезков при работе с частями массивов вместо операторов цикла.

5.3 Условные операторы

Условный оператор выбирает для выполнения одну или ни одной из входящих в него последовательностей операторов в зависимости от значения истинности одного или нескольких условий.

Общий вид условного оператора:

```
IF      Условие    THEN
                   Последовательность_Операторов
{ELSIF Условие THEN
                   Последовательность _Операторов}
{ELSIF
                   Последовательность _Операторов} END
IF;
```

Выражение, задающее *Условие*, должно быть логического типа. *Последовательность_Операторов* может содержать любое число операторов.

D Примеры:

```
If  A > B then
  C: = 100;
  Z: = Z-H; end
if;

if A > B then
    MIN : = B;
else
    MIN : = A;
end if;

If A = 0  and   B = 0  then

   Vec_Z: = SuM (Vec_A, Vec_B); C:
   = 23.356;
   get ( Number_Of_Array); put
   ("End of task");
end if;
```

5.4 Операторы выбора

Оператор выбора отбирает для выполнения одну из нескольких альтернативных последовательностей операторов. Выбор альтернативы осуществляется в зависимости от значения *Выражения.*

Общий вид оператора выбора:

CASE *Выражение* IS

WHEN *Вариант_1 =>*
Последовательность _Операторов;
WHEN *Вариант_2 =>*
Последовательность _Операторов;

WHEN *Вариант_M =>*
Последовательность „Операторов;
WHEN OTHERS =>
Последовательность _Операторов;
END CASE;

Выражение в операторе выбора должно быть дискретного типа.

D Например:

```
case MODE_OPR is
   when '+' => ADD;
   when '-' => SUBSTR;
   when '*' => MULTI;
   when '/' => DIVIDE;
   when others => X : = Y;
end case;
```

СОВЕТЫ:

* Используйте оператор **Case** вместо if .. end **if** везде, где это возможно.
* Минимизируйте использование части **Others** в операторе **Case.**
* Используйте расширения типов и диспетчеризацию типов вместо оператора **Case.**

5.5 Операторы цикла

Операторы цикла в языке Ада имеют три формы: **while, for и loop** и могут быть именованными. Простейший цикл имеет вид:

LOOP
- - *Последовательностъ_Операторов;*
END LOOP;

Он создаёт бесконечный цикл, использовать оператор *выхода* Для выхода из него можно

EXIT [WHEN *Условие]* П [*Имя_Цикла];*

Например :

```
MM: loop
      get (x);
      if  x = 0  then
      exit  MM; end
    if; end  loop
    MM;
```

Праметрический цикл задаётся с помощью цикла **FOR**

FOR *Параметр_Цикла* **[REVERSE]**
Дискретный_Диапазон **LOOP**
- - *Последовательность_Операторов;* **END LOOP;**

Наличие ключевого слова **Reverse** присваивает параметру цикла значение из дискретного диапазона в обратном порядке.

Параметр цикла явно не описывается, так как он определяется типом дискретного диапазона. Поэтому он локален по отношению к циклу и вне его не существует. Кроме того, параметр цикла нельзя изменять и передавать как параметр в процедуру, которая сможет это сделать.

П Примеры:

```
fori in 1 .. 10 loop
   X:= X+ Sin(i);
```

```
end loop;

for j in reverse M'first . .    M'last loop
   Rez : = Rez + M(j);          - - диапазон цикла задаётся
end loop;                       - - через атрибуты массива
```

Оператор цикла со схемой итерации **WHILE** имеет вид:

```
WHILE Условие LOOP
            - - Последовательность_Операторов;
END LOOP;
```

Перед началом выполнения последовательности операторов проверяется условие; если его значение **True,** то последовательность выполняется, если **False,** то выполнение оператора цикла заканчивается.

П Пример:

```
DET:  while  TAB(k)<100  loop S:  = S +
      TAB(k); k:  = k+1; end  loop  DET;  - -
      именованный цикл
```

Именованный оператор цикла целесообразно использовать в операторе **exit,** а также для вложенных циклов.

С целью повышения скорости обработки двумерных массивов рекомендуется их обработку выполнять, начиная со второго индекса:

```
type TT is array ( 1 .. N, 1 .. M ) of integer ;

Massiv :  TT ;

for j  in 1 .. M  loop for  i
   in  1 .. N  loop
      -; -  обработка  Massiv ( i , j ) ;
  - - типа  TT end  loop
; end  loop ;
```


СОВЕТЫ:

* Внимательно задавайте границы циклов; используйте для этого по возможности атрибуты типов и обьектов;
* Используйте в циклах конструкцию for;
* Используйте отрезки вместо циклов при копировании частей массивов;
* Избегайте оператора **exit** в циклах **for** и **while;** Ф Помечайте вложенные операторы цикла, особенно если они содержат оператор выхода

5.5 Оператор перехода

Определяет явную передачу управления на помеченный меткой оператор:

GOTO *Имя_Метки;*

Метка записывается в форме

«Идентификатор»

Использование оператора перехода затрудняет понимание программы и его следует избегать. Кроме того, его нельзя использовать для передачи управления из подпрограммы, пакета, тела задачи и оператора **accept.**

СОВЕТЫ:

* Не используйте оператор Goto !!!

5.7 Операторы блока

Оператор блока содержит последовательность операторов, которой может предшествовать раздел локальных описаний, а завершать - обработчик исключений:

```
[DECLARE
                - - Описательная_Часть;]
BEGIN
                - - Последовательностъ_0ператоров;
[ EXCEPTION
                - - Обработчики_Исключений]
END;
```

Блок используется для ограничения области действия описаний, так как все локальные описания существуют только в блоке и не существуют перед блоком и после него. Память для объектов, описанных в блоке, распределяется при входе в него и освобождается при выходе из него. Блок может быть помеченным.

D Пример:

```
-- обмен значениями векторов
SWAP: declare
        Temp begin Vector; -- локальная переменная в блоке
        Temp VA VB
      end SWAP = VA;
      --       = VB;
      именованны = Temp;
      й блок
```

ffl **СОВЕТЫ:**

* Используйте оператор блока для локальных описаний и локальных обработчиков исключений.
* Помечайте вложенные блоки.

П Пример использования операторов блока, цикла и выхода :

```
declare
   Deep : Matrix ;   -- локальная переменная
begin
  A : for i in 1 .. N loop          -- вложенные именованные
      B : for J in 1 .. N loop      -- циклы
      Deep(j,i):= MA(i,j) - MB(j,i);
      if Deep(j.i)= 0.0 then
   exit B ;                         -- выход из внутреннего цикла
end if;
end loop B ;
end loop A ;
Res : = Deep ;
end ;                    -- конец блока
```


Главаб. П Р О Г Р А М М Н Ы Е М О Д У Л И

6.1 Абстракции

Общие концепции языка Ада являются прекрасной основой для разработки больших программных систем. Примером применения языка для разработки таких систем является его успешное использование в целом ряде проектов:

- системе управления скоростной магистралью TGV (Франция);
- системах контроля метрополитеном (Париж, Каир, Калькутта);
- банковских системах (Швейцария);
- управление атомномными электростанциями (Чехия) ;
- авиастроении (Боинг-777, США) и др.

Разработка больших программных систем представляет собой сложный процесс, включающий несколько этапов: анализ и описание требований, проектирование, реализация, модификация, сопровождение [5].

Одним из основных среди них является этап проектирования, на котором определяется структура будущей программы, как совокупность подзадач, из которых далее "собирается" программа.

В основе этапа проектирования сложных программных систем лежит, как правило, *декомпозиция*. Декомпозиция программы заключатся в определении оптимального набора модулей, взаимодействующих по хорошо определённым и простым правилам, которые сообща выполняют заданную функцию.

При эффективном проведении декомпозиции исходная задача разбивается на подзадачи таким образом, чтобы каждая подзадача имела один и тот же уровень рассмотрения, могла решаться одновременно с остальными (что особенно важно для параллельных вычислительных систем) и взаимодействовать с другими подзадачами. Объединение подзадач позволяет решить исходную задачу.

Эффективность декомпозиции как основного механизма этапа проектирования в значительной мере зависит от способов её реализации. Существует множество методик проектирования программ. Наиболее современные из них основываются на абстракциях различного вида.

В программировании уже давно используются различные виды абстракций (например, процедурные абстракции) и новейшие языки программирования с этой точки зрения характеризуются все более высоким уровнем реализуемых видов абстракций. Абстрагирование предполагает игнорирование на этапе проектирования ряда подробностей, что позволяет свести задачу к более простой.

Декомпозиция, базирующаяся на абстракции, при разбиении программы на компоненты предполагает продуманный выбор компонент и осуществляется путем изменения списка детализации.

В простейшем случае все модули могут представлять собой только процедурную абстракцию, то есть программа будет состоять из процедур и функций.

Реализация абстракций основывается на двух подходах - абстракции на основе параметризации и абстракции на основе спецификации.

Абстракция через параметризацию есть абстракция неограниченного набора различных вычислений, которые выполняются в программе. Такой вид абстракции повышает универсальность программ, позволяя относительно просто описывать вычисления, легко и эффективно реализуется в языках программирования.

Более высокий уровень обобщения достигается в *абстракциях через спецификацию*. Абстракция в этом случае имеет спецификацию и тело. Назначение модуля, реализованного на основе такой абстракции, теперь становится ясным через его спецификацию, описывающую цель его работы, а не через тело. То есть происходит абстрагирование от процесса вычислений в теле до уровня знания того, что данный модуль в итоге должен реализовать.

Преимущество абстракции заключается в том, что при использовании модуля нет необходимости знать тело, то есть можно абстрагироваться от него и не обращать внимание на несущественную информацию. При этом несущественность способа реализации тела позволяет легко переходить от одной реализации к другой без внесения изменений в структуру программы, а также выполнять эту реализацию на разных языках программирования.

В общем случае абстракция на основе спецификации наделяет программу двумя свойствами: *локальностью* и *модифицируемостью*.

Локальность означает, что реализация одной абстракции может быть выполнена без анализа реализации другой абстракции. Для написания программы, использующей абстракции, достаточно только понимать её поведение, а не подробности её реализации. Принцип локальности облегчает составление программы и анализ уже созданной абстракции. Он позволяет составлять программу из абстракций, создаваемых независимо друг от друга.

Абстракция на основе спецификации позволяет упростить модификацию программы. Если реализация абстракции меняется (через изменение тела абстракции), но её спецификация при этом остаётся прежней, то эти изменения не повлияют на оставшуюся часть программы. Объём исправлений может быть значительно сокращён путем выделения потенциальных модификаций уже на начальном этапе разработки программ и последующим ограничением их небольшим числом абстракций. Модифицируемость существенно повышает эффективность программы.

Выделим следующие виды абстракций:

- процедурная абстракция
- абстракция данных
- абстракция процесса.

Процедурная абстракция позволяет расширить множество заданных языком программирования операций и является мощным и широко используемым средством повышения эффективности языка программирования.

Абстракция данных позволяет расширить множество заданных языком типов данных и операций над ними, реализуя возможность добавлять к базовому уровню операции и новые типы данных. Абстракция достигается представлением операции как части типа:

абстракция_данных : = < объекты, операции >

Абстракции данных играют важную роль в проектировании программ, так как выбор правильных структур данных играет решающую роль в создании эффективной программы.

Абстракции данных позволяют отложить окончательный выбор структуры данных до момента, когда эти структуры станут вполне ясными для проектировщиков. Вместо непосредственного определения структуры данных вводится абстрактный тип со своими объектами и операциями, в терминах которого осуществляется реализация модулей. Модификация программы теперь сводится к изменениям в реализации типа без изменения модулей.

Абстракция процесса возникла в связи с развитием и усложнением операционных систем, где понятие процесса было введено для лучшего восприятия работы вычислительной системы и построение механизмов операционных систем на едином концептуальном базисе.

Абстракция процесса в языках программирования для параллельных вычислительных систем связана с описанием параллельно выполняемых модулей. Для процесса определяется состав допустимых состояний и переходов из одного состояния в другое. Характерными состояниями процесса являются:

- порождение;
- выполнение (активное);

- готовность;
- блокирование;
- окончание.

Процесс рассматривается как динамический объект, в отношении которого требуется обеспечить реализацию каждого из допустимых состояний, а также допустимые переходы из состояния в состояние в ответ на события, которые могут явиться причиной таких переходов.

Главной особенностью процессов в параллельных системах, определяющей организацию вычислительного процесса в ней, является их параллельность. То есть в системе одновременно существует и развивается множество процессов.

Другое важное свойство процессов, реализующих параллельную программу, заключается в том, что эти процессы, как правило, взаимосвязаны, то есть между ними поддерживаются различного вида связи: функциональные, пространственно-временные, управляющие, информационные и т.д. Управляющие связи устанавливают между процессами отношения типа "порождающий-порождаемый", информационные связи приводят к взаимодействующим процессам.

6.2 Реализация абстракций

Язык Ада с самого начала его создания был ориентирован на использование абстракций различного вида с реализацией как через спецификацию, так и через параметризацию.

С этой целью в Аде определён общий (унифицированный) вид абстрактного модуля:

НАСТРОЙКА

СПЕЦИФИКАЦИЯ

ТЕЛО

Абстракция через спецификацию реализуется наличием в модуле двух частей: спецификации модуля и тела модуля. Абстракция через параметризацию реализуется с помощью описания настройки модуля.

Унифицированное представление каждого вида абстракции в Аде, а также возможность разделения спецификации и тела предоставляют дополнительные возможности для эффективной реализации преимуществ абстракций при проектировании программ с помощью Ады.

Абстракция процедур в Аде реализуется через развитый механизм процедур и функций и поддерживается механизмом раздельной компиляции подпрограмм, в основе которого - отделение спецификации от тела и их раздельная компиляция. Это обеспечивает эффективную поддержку (реализацию) важного свойства абстракции модифицируемости, так как компиляция и перекомпиляция тела подпрограммы выполняется без перекомпиляции программы, содержащей спецификацию подпрограммы. Кроме того, процедурная абстракция в Аде (как и все остальные виды абстракций) поддерживается таким мощным средством, как механизм исключений, направленным на выявление и обработку ошибочных (исключительных) ситуаций во время выполнения программы.

Абстракция данных в Аде реализуется через развитые средства инкапсуляции с помощью модулей типа пакет и приватных типов. Пакет позволяет связать любой набор спецификаций для работы со структурой данных (классом структур данных) с некоторыми подробностями реализации. Пакеты используются для выполнения следующих функций:

- указание набора разрешённых операций над типом данных;
- реализаций тела, состоящих из операций, определённых ранее в спецификации;
- сокрытие всех объектов, спецификаций и подробностей реализации.

Реализация абстракций через параметризацию для абстрактных процедур и данных выполнена с помощью так называемых настраиваемых **(generic)** модулей. Параметризация в таких модулях позволяет использовать в качестве параметров не только традиционные виды параметров, но также типы, подпрограммы и пакеты, что значительно расширяет возможности организации различного вида вычислений.

Абстракция процесса в Аде реализована с помощью модулей типа задача **(task**). Такие модули обеспечивают параллельное выполнение частей программы, взаимодействие этих частей через *механизм рандеву*

- средство высокого уровня, основанное на передаче сообщений. Дополнительные возможности для описания множества процессов и их взаимодействия обеспечиваются в языке задачным типом (**task type**) и защищенным типом (**protected type).**

Таким образом, программа на языке Ада составляется из одного или нескольких видов программных модулей (программных сегментов) следующего вида:

- подпрограммы;
- пакеты;

- задачи;
- настраиваемые (родовые) модули;
- защищённые модули (Ада-95).

Все виды модулей имеют единую структуру, состоящую из спецификации модуля и тела модуля. Каждый программный модуль определяет интерфейс между модулем ("сервером") и его пользователем ("клиентом").

Спецификация модуля и *тело* модуля - различные составляющие модуля. Синтаксически они являются законченными программными объектами и при желании могут компилироваться *раздельно*.

Тело модуля скрыто от пользователя, который использует ресурсы модуля только так, как это определено в спецификации. В противоположность телу спецификация есть видимая часть модуля, описывающая ресурсы, предоставляемые при использовании данного модуля.

Каждый вид модуля определяет свои требования к спецификации и телу модуля. Эти особенности будут рассмотрены при обсуждении каждого из перечисленных видов программных модулей.

6.3 Примеры

П Пример подпрограммы:

```
procedure Sum_Complex (A, B : in  Complex ;
                       C : out  Complex); - - спецификация

procedure Sum_Complex (A, B : in  Complex ;
                       C : out  Complex)  is - - тело begin
      C . x : = A . x + B . x ;
      C . y : = A . y + B . y ;
end Sum _Complex;
```

П Пример функции :

```
function Mult_Complex (A, B : Complex)
                       return Complex; - - спецификация

function Mult_Complex (A, B : Complex )
                       return Complex  is Z:      - - тело
   Complex;
```



```
begin
Z . x : = A . x * B . x -
A . y * B . y ;  Z . y:= A . x * B .y
+A.y * B . x; return Z; end
Mult_Complex;
```

D Пример пакета :

```
package Complex_Numbers is - - спецификация пакета type
   Compex  is  record x:  float; y:  float; end  record;

   function "+" (A , B : Complex) return Complex ;
   function "/" (A , B : Complex) return Complex ;
   function "-" (A, B : Complex) return Complex ;
   function "*" (A, B : Complex) return Complex ;

end  Complex_Numbers;

package body Complex_Numbers is              -- тело пакета
  function "+" (A, B : Complex) return Complex  is

  end "+";
  function 7" (A , B : Complex) return Complex  is

  end "/";
  function "-" (A , B : Complex)  return Complex  is

  end"-";

  function "*" (A , B : Complex) return Complex  is

  end "*"; end

Complex_Numbers;
```

D Пример настраиваемого пакета :

```
generic
    type Elem is digits <> ;    - - параметр настройки
package ComplexJM umbers is  - -  спецификация настройки type
    Complex is private;    - -  приватный тип
    function "+" (A , B : Complex) return Complex ; function
    7" (A , B : Complex) return Complex ;; function "-" (A , B :
    Complex)  return Complex ; function "*" (A , B : Complex)
    return Complex;

private                  - -  приватная часть спецификации
    type Complex is
    record
x:  Elem; y:  Elem; end
record ; end
Complex_Numbers;

package body ComplexJMumbers is
          . . .                  - -  тело настраиваемого пакета

end  Complex_N umbers; D
```

Пример задачи :

```
task Stack is
    entry  Pop ( X :  out  Elem ); - - спецификация задачи
    entry  Push (X : in  Elem ); end
Stack;

task  body Stack is                - - тело задачи
Pool:   Elem; begin
loop
       accept  Pop (X : out Elem )  do
           Pool: = X ;
       end Pop;
```



```
       accept Push ( X : in Elem )  do
     X : = Pool; end
 Push; end loop; end
 Stack;
```

D Пример защищенного модуля :

```
protected  Data  is                    - - спецификация
   function  Read (X : out integer);
   entry  Write  (X  :  in  integer);
private
   Share_Elem  : integer;
   Tag          : boolean : = false ;
end Data;

                                                  - - тело
protected  body  Data is begin
   function  Read ( X : out integer)  is begin
   Tag : = True ; return
   Share_Elem; end  Read;
 entry Write (X: in integer) when Tag = True is Tag : =
 fals ; Share_Elem: = X; end  Write; end  Data;
```

Глава ^ П О Д П Р О Г Р А М М Ы

Подпрограммы в языке Ада определяют программный модуль для описания действий, выполнение которых инициируется *вызовом* подпрограммы. Существует два вида подпрограмм: *процедуры и функции.*

Подпрограмма в Аде реализует процедурную абстракцию через спецификацию и через параметризацию.

Подпрограммы в Аде могут быть *рекурсивными*, то есть вызывать сами себя. Ещё одно важное свойство подпрограмм заключается в том, что все они *реентерабельные.*

По -сравнению с Адой 83 , изменения, касающиеся подпрограмм, в Аде 95 направлены , в основном, на улучшение механизма работы с параметрами.

7.1 Спецификация подпрограммы

Спецификация подпрограммы задаёт имя подпрограммы и всю необходимую информацию о параметрах и результатах подпрограммы . Спецификация определяет соглашение о её вызове.

Информация о формальных параметрах процедуры задаёт *вид* и тип параметров. Вид параметра определяется ключевыми словами ***in, out, in out:***

in - параметр подпрограммы рассматривается как *входной,* то есть является константой, и разрешается только его чтение;

out - параметр подпрограммы рассматривается как *выходной,* то есть является в процедуре переменной, и разрешается изменять его значения;

In out - параметр является одновременно и *входным* и *выходным*, то есть рассматривается как переменная, значение которой можно читать и изменять.

Если вид параметра явно не задан, то предполагается вид ***In.***

Передача параметра в подпрограмму выполняется с использованием двух механизмов передачи : путем *копирования* (by copy) или путем *вызовы ссылки* (by reference).

Формальные параметры при копировании должны иметь элементарный тип. При этом передача параметров осуществляется только перед и после выполнения подпрограммы.

Формальные параметры при использовании вызова ссылки должен быть производным от одного из следующих типов: • тэгового типа;

- задачного типа;
- защищенного типа;
- лимитированного типа (но не личного);
- составного типа с компонентами передаваемых через ссылку;
- личного типа.

При этом передача параметра выполняется непосредственно при вызове ссылки.

Для параметров других типов механизм передачи не определен.

Описание параметра в спецификации подпрограммы может заканчиваться выражением. В этом случае говорят, что имеет место выражение *по умолчанию* (default_expression) для формального параметра (только для вида **in).** При вызове подпрограммы с такими параметрами их можно опускать, при этом значение фактического параметра берется из выражения по умолчанию.

П Пример спецификации подпрограмм:

```
procedure  Model_Train ;
procedure  Cat( Number: in out  Float);
procedure  Zt_lsk( X : in  Data; Y : out  Elem );
procedure  Sound (A, D, C :  in  Note);

procedure  Crown ( Dase :  out  Size );
procedure  Print_FF( Arg :  in Positive; Num :  in Integer: = 200);
procedure  SUM (A, B  :  in  Vector; C :  out  Vector);

function  Lotto  return  Number; function
Demo_Run_Model( X : Vector;
                                  Y : integer)  return  Vector;
function  " -" ( MA, MB :  Matrix )  return  Matrix ;
function  Root (X :  float)  return  float;
```

В Аде 95 подпрограмма может быть описана как *абстрактная :*

Спецификация_Подпрограммы **IS ABSTRACT ;**

Абстрактная подпрограмма - это подпрограмма, которая не имеет тела. Понятие абстрактной подпрограммы базируется на использова-

нии *тэговых* типов, связано с объектно-ориентированным программированием и более подробно будет рассмотрено в Главе 16. О Пример описания абстрактных подпрограмм:

```
package  Abstract_Objects  is

   type  Snow  is abstract  tagged  limited  private; function
   Action (Arg : Snow)  return  Snow  is abstract; procedure
   Power (X : in integer; Y : in out Snow)
                                        is  abstract;

end  Abstract_Objects;
```

В Аде подпрограммы могут иметь одинаковые имена, то есть допускается *совмещение* подпрограмм:

```
procedure  ADD  (A, D      in  integer;  C  :  out integer);
procedure  ADD  (X, Y      in  float;    Z  :  out real);
procedure  ADD  (V1.V2     in  Vec;      A  :  out Vec );
```

При совмещении выбор конкретной подпрограммы на выполнение при вызове выполняется автоматически на основании свойств параметров или типа результата:

```
ADD( 345,  Nu);      ADD( 234.876,  Reck );

ADD(Vec_1, Vec_2, Vec_Sum );
```

Спецификация подпрограммы может быть опущена, если подпрограмма описана и вызывается в одном программном модуле; В этом случае её тело выступает в качестве спецификации.

Ш СОВЕТЫ:

ф Выбирайте имена подпрограмм и параметров так, чтобы можно было понять их назначение.

* Подпрограмма - эффективный и понятный вид абстракции; используйте подпрограммы для увеличения абстракции.

* Ограничивайте каждую подпрограмму выполнением только одного действия.

* Используйте подпрограммы для инкапсуляции и сокрытия деталей реализации, которые могут быть изменены.

* Список параметров определяет интерфейс подпрограммы; тщательно выбирайте имена и порядок формальных параметров - это улучшает понимание и использование подпрограммы.

* Используйте именованное связывание фактических параметров при вызове, если число параметров велико; это уменьшает количество ошибок при работе с подпрограммами.

* Обязательно указывайте направление передачи параметров во всех процедурах через описание вида параметров **(in, out, in out).**

** Используйте вид параметов для ограничения доступа (более ограничены параметры вида in , out , меньше - вида **in out).**

* Используйте параметры по умолчанию для добавления новых параметров к существующим; размещайте их в конце списка параметров.

7.2 Тело подпрограмм

Тело подпрограммы определяет её выполнение. Тело подпрограммы начинается со спецификации, после чего размещаются локальные описания и операторная (основная) часть подпрограммы.

Локальные описания могут включать описания типов, переменных, исключений, процедур, пакетов и задач, а так же спецификаторы представления.

Основная часть подпрограммы содержит последовательности операторов и, возможно, обработчики[1] исключений. П Пример

Процедуры и функции для сложения векторов. Процедура SUM предназначена для сложения векторов фиксированной размерности, функция "+" - для сложения векторов любой размерности:

```
procedure  Sum (A, B :  in Vector;  C  :  out Vector)  is begin
   for  i  in  1 .. N  loop C(
      i): = A( I) + B( i);
   end loop;
end  Sum; 2
```

```
function  "+"(Vx, Vy  :  Vector)  return  Vector is
```

```
   Vres:   Vector;
begin
    if    Vx'first /= Vy'first    or   Vx'last /= Vy'last  then
      raise   Error;                    - - возбуждение исключения
    end if;
    for  i   in    Vx'Range    loop Vres
        (i): =    Vx( i) + Vy( i);
    end   loop;
    return   Vres ;      - -  возвращаемое значение функции end
" + ";
```

Для повышения эффективности работы подпрограмм в языке предусмотрена прагма **Inline:**

Pragma INLINE(*Имя подпрограммы, (Имя подпрограммы }*);

Прагма используется для того, чтобы в тексте при каждом вызове процедуры , указанной в прагме **Inline,** выполнялась подстановка тела соответствующей подпрограммы.

D Например:

```
procedure Happer( X:        in   Integer;   Z:     out   String);
pragma Inline (   Happer);
```

Здесь компилятор должен все вызовы подпрограммы Happer заменить телами этой подпрограммы. Это повысит производительность программы, в которой используется данная подпрограмма, поскольку сокращается время выполнения процедуры Happer.

7.3 Вызов подпрограммы и согласование параметров.

Вызов подпрограммы вызывает выполнение тела соответствующей подпрограммы. Вызов подпрограммы - это либо оператор вызова процедуры, либо вызов функции. При вызове подпрограммы происходит *связывание* фактических и формальных параметров. В Аде такое связывание осуществляется двумя, способами:

- неявно, когда соответствие устанавливается порядком написания фактических параметров (*позиционное связывание)*;

- явно, когда фактические параметры приводятся вместе с именами формальных параметров в произвольном порядке (*именованное связывание):*

```
SUM (VA, VB,   RES);                          - - позиционное
SUM (B => VB, C=> RES,   A => VA);            - - именованное
```

D Примеры вызова подпрограмм из 7.1:

```
Model_Train;                  Cat( X);        Zt_lsk( 62.8, Rez);
Sound( G, E );              Sound( Y = > E, X = > G );
Crown (Stack_Size);

Print_FF( Coutn ); - - второй параметр используется по умолчанию
Print_FF( ATF, 12 );              Res: = Lotto ;
MA : = MB - MC ;                Z : = D( Vec_A, alfa ) + Vec_B;
```

Независимо от способа связывания тип каждого фактического параметра должен совпадать с типом соответствующего формального параметра.

Согласование параметров при вызове подпрограммы - это согласование формальных и фактических параметров. Согласование параметров включает:

- согласование типов ;
- согласование вида;
- согласование подтипа;
- полное согласование.

Некоторые правила согласования параметров :

- тип каждого фактического параметра должен совпадать с типом соответствующего формального параметра;
- фактический параметр, сопоставляемый с формальным, имеющим вид in out или out должен быть именем переменной:
- скалярный формальный параметр должен удовлетворять всем уточнениям диапазонов типов, причем для вида in, in out - перед вызовом подпрограммы, а для вида out, in out - в момент окончания выполнения подпрограммы;если спецификация параметров

включает выражение по умолчанию и имеет вид in , то при вызове подпрограммы не обязательно сопоставление для такого параметра;

7.4 Процедуры

Спецификация процедуры имеет следующий вид:

```
PROCEDURE   Имя_Процедуры
                    [(Формальные ^Параметры)];
```

Тело процедуры имеет следующий вид:

```
PROCEDURE       Имя_Процедуры
                [(Формальные_Параметры)]   IS
    - - Локальные_Описания
BEGIN
    - - Последовательность „Операторов
[EXCEPTION
    - - Обработчики_Исключений]
END   Имя_Процедуры;
```

Выполнение процедуры характеризуется результатом выполнения, связанным с

- изменением её параметров (in out);
- формированием значений параметров (out);
- изменением глобальных переменных.

Процедура вызывается посредством вызова подпрограммы, который рассматривается как оператор. Выполнение процедуры завершается либо при достижении конца тела, либо при выполнении оператора возврата (**return).** D Пример: Спецификация и тело процедуры

• - спецификация

```
procedure SWAP (A,B : in integer; C : out integer);
```

• - тело

```
procedure SWAP (A, B: in integer; C: out integer) is
   Temp : integer
begin
   Temp  : = A ;
```



```
   A  :  = B ;
   B  :  = Temp;
end  SWAP;
```

СОВЕТЫ:

- Избегайте использования вида **in out** при описании формальных параметров.
- Используйте глаголы действия при выборе имен процедур.

7.5 Функции

В отличие от процедуры *функция* вырабатывает некоторое значение, называемое результатом вызова процедуры. Кроме того, если вызов процедуры является самостоятельным оператором, то вызов функции выполняется обязательно в выражении. Если сравнивать процедуры и функции, то основные различия заключаются в следующем:

• функция всегда при вызове возвращает только одно значение; процедура - любое количество значений, в том числе и ни одного;

• обязательно присутствие слова **return** в спецификации функции;

• наличие в теле функции одного или нескольких операторов **return,** которые с помощью последующего за ним выражения определяют результат, возвращаемый функцией;

• спецификация функции начинается со слова **function.**

Спецификация функции:

```
FUNCTION  Имя_Функции   RETURN   Тип_Результата ;
```

Тело функции:

```
FUNCTION            Имя_Функци    RETURN
                                  Тш_Результата   IS
     - - Локальные_Описания
BEGIN
     - - Последовательность_Операторов
     -- RETURN  Выражение [
EXCEPTION
     - -  Обработчики _Исключений \
```

END *Имя_Функции*

В качестве имени функции можно использовать обозначение операции, например, "+" или "<". D Пример: Спецификация и тело функции.

```
- - спецификация функции
function Max (A, B : Data) return Data ;

- - тело функции
function Max (A, B : Data) return Data is
    C : Data : = 0.0;
begin
    if A > B then
        C : = A ;
    else
        C : = B ;
    end if;
    return C;
end Max;
```

Ш СОВЕТЫ:

Используйте функции без параметров. Минимизируйте побочные эффекты работы функций. Используйте функции для подпрограмм, которые формируют результат в виде одного значения.

Минимизируйте количество операторов **return** в теле функции. Используйте неограниченные массивы в формальных параметрах , имеющих тип массивов.

Делайте размеры локальных переменных в теле подпрограммы зависящимим от фактиче ских параметров.

ИЗМЕНЕНИЯ:

О Тело подпрограммы может быть получено с помощью операции переименования.

О Подпрограмма может быть описана как абстрактная. О Улучшен механизм работы с параметрами и результатами подпрограмм.

О Изменены правила работы с параметрами вида out. О Изменены некоторые аспекты согласования формальных и фактических параметров при вызове подпрограмм.

7.6 Примеры

D Пример 1.

Процедура содержит вложенные функции.

```
procedure Art (A, B. C : in Elem ;
                                    Rez_Max, Rez_Min : out Elem ) is
   function Max (X : in Elem ;  Y : out Elem ) return Elem  is begin
       if X < Y then
       return  Y; else
      return  X; end  if
   ; end Max;

   function    Min (X: in Elem ; Y : out Elem ) return Elem  is begin
       if X > Y then
       return  X; else
  return  Y; end  if ;
end Max; begin  - -
Art

    Rez_Max : = Max (A, Max ( B , C ));
    Rez_Min : = Min (A, Min ( B, C ));

end Art;
```

D Пример 2. - -
неограниченный тип

```
type   Vector is   array(integer  range <>) of  Item ; procedure

Swap_Vector (VA, VB :   in   out  Vector) is

   Temp : Vector (VA'first . . VA'last); - - локальное описание begin

    if   VA'first  /=  VB'first  or   VA'last   /=   VB'last    then
       raise Vector_Error;   - -   возбуждение исключения end   if;

    Temp:=    VA;
    VA    :=    VB;
    VB    : =    Temp;

end   Swap_Vector;
```

В процедуре Swap_Vector используется неограниченный тип Vector для описания формальных параметров. Этим достигается возможность использования данной процедуры для работы с векторами любой размерности. В теле процедуры осуществляется проверка размерности обрабатывемых векторов и в случае ошибки в задании их границ возбуждается исключения Vector_Error. D Пример вызова процедуры Swap_Vector:

```
Swap_Vector ( X (1 .. 10 ), Y (1 .. 10 ));
```


Глава 8. П А К Е Т Ы

Пакеты - это один из наиболее важных видов программных модулей, из которых составляется программа на языке Ада. Пакеты реализуют *абстракцию данных* через спецификацию.

Пакеты объединяют совокупность логически связанных понятий (ресурсов). Простейшие пакеты в качестве ресурсов задают описания типов и объекты. Более сложные пакеты включают в качестве ресурсов подпрограммы и задачи, которые могут вызываться из-вне пакета.

Как программный модуль пакет состоит из *спецификации* и *тела*. Спецификация пакета *описывает* (перечисляет) ресурсы пакета. Тело пакета реализует ресурсы, предоставляемые пакетом.

8.1 Спецификация пакета

Спецификация пакета имеет следующий вид:

PACKAGE *Имя_Пакета* **IS**
- - Видимый_Раздел_Описания_Ресурсов_Пакета

[**PRIVATE**
-- Личный_Раздел_Описаний]

END *Имя_Пакета;*

Спецификация пакета разделяется на *видимую* и *приватную* (личную) части. Приватная часть спецификации начинается после слова **private.**

Описания из видимой части спецификации могут быть использованы вне пакета, прямая видимость при этом обеспечивается спецификатором **Use.**

Видимый раздел описания пакета может содержать описания следующих ресурсов:

- констант;
- типов;
- объектов;
- подпрограмм;
- исключений;
- задач;
- пакетов.

П Пример спецификации пакета:

```
package DATA is

   n : integer constant: = 100 ; type
   Vector is array ( 1 .. n) type of integer; .. n
   Matrix is array ( 1 .. n, 1 procedure ) of integer; C : out
   Sum_Vec (A, B : in Vector; TIME: Vector);
   exception;

end DATA;
```

При работе с пакетом DATA в пользовательской программе разрешается использование всех ресурсов пакета, перечисленных в его спецификации.

Пакет может быть размещен внутри пользовательской программы.

Для подключения пакета, компилируемого *раздельно* (в этом случае он является библиотечным модулем), применяются спецификаторы **with и use.**

```
Wit DATA; DATA;
h
```

Описания из приватной части спецификации видны только внутри области действия этого описания, в теле пакета и в дочерних модулях.

Ш СОВЕТЫ:

* Описывайте в спецификации пакеты только то, что необходимо для использования вне пакета.
* Минимизируйте число обьявлений в спецификации пакета.
* Минимизируйте использование спецификатора **with** в спецификации пакета.
* Не используйте глобальные данные в пакете.
* Избегайте ненужной видимости; прячьте детали реализации от пользователя.

8.2 Приватные типы и приватные расширения

Прваатные (*личные)* типы (**private types**) и приватные расширения (**private extensions**) обьявляются в видимой части спецификации пакта и позволяют отделить характеристики типа, которые можно использовать вне программного модуля, от характеристик, которые непосредственно можно использовать в пакете. То есть для приватных типов вводятся два типа характеристик: *'внутренние* - для использования только в пакете и *внешние* - разрешаемые вне пакета. Описание *приватного типа:*

TYPE *Имя_Типа [Дискриминант* J IS **[[ABSTRACT]**

TAGGED] [LIMITED] PRIVATE;

Описание *приватного расширения:*

TYPE *Имя_Типа [Дискриминант]* IS
[ABSTRACT] NEW WITH PRIVATE;

П Пример объявления приватного типа:

```
type Elem is private;
type File_X ( N : integer ) is limited private ;
type Rev is tagged private;
type Pipe_Control is abstract tagged private;
```

П Пример приватного расширения типа:

```
type Stone is new Wood with private;
type Dust is abstract new Park with private;
type Stars is new Moon with private;
```

Для типа могут существовать *предопределенные примитивные* операции. Они либо наследуются, либо определяются непосредственно при создании типа. Например, типа **Integer** имеет предопределенные операции "+", "-", "*", "/" и др. Если тип описан как приватный, то для него примитивные операции *не наследуются.* Для приватного типа вне пакета разрешены *только* следующие операции:

- присваивания (: =);
- создания обьектов данного типа ;

- сравнения на равенство и неравенство (= *Ф*);
- проверка на принадлежность (in , not in);
- передача обьектов типа в качестве параметров,

а так же операции, описанные в *видимой* части пакета для этого типа в виде подпрограм (приватные операции).

Описание личного типа выполняется как в видимой, так и в личной частях спецификации.

В видимой части личные типы задаются здесь только на уровне их идентификаторов.

В личной части выполняются полные описания личного типа. К пользователю пакета эти описания отношения не имеют (для него они невидимы) и предназначены для компилятора при реализации видимой части и использования в теле пакета. П Пример:

```
package   SMO  is

    type Matrix is private ;                    - - объявление личного типа
    procedure Transp_Matr ( MX :   in   out   Matrix);

private                                        - - приватная  часть  спецификации
    type   Matrix   is   array (1 .. 100,1 .. 100 )   of   real;- -детали

                                                   ~ реализации личного типа

    Venta :   constant   Matrix;               - - субконстанта

end   SMO;
```

Константы личного типа задаются в пакете в виде *субконстант.*

Ш СОВЕТЫ:

* Используйте личные типы вместо обычных.
* Отдавайте предпочтение ограниченным личным типам вместо личный.
* Используйте личные дочерние пакеты для локальных описаний, используемых только в реализации спецификации пакета.

8.3 Лимитированные типы

Лимитированный (ограниченный) тип (**limited type**) - тип, для которого не допустимы операции присваивания.

Тип является лимитированным, если он унаследован от одного из следующих типов:

- типа, имеющего в описании зарезервированное слово **limited;**
- задачного типа;
- защищенного типа;
- составного типа с лиимтированными компонентами.

Для лимитированного типа не существуют предопределенные операции сравнения на равенство (equality operators).

Для лимитированного типа определены следующие правила:

- не разрешается инициализация при обьявлении объекта лимитированного типа;
- выражение по умолчанию не допускается в описании компонент, если тип записи - лимтированный;
- не разрешается инициализирующий указатель (генератор), если тип является лимитированным;
- формальный параметр настройки вида **in** не может быть лимитируемым типом.

Для лимитируемого типа не используются агрегаты (в сложных типах) и конкатенация (в массивах) . П Рримеры:

```
type  Focus  is limited  privated;
type  OMP    is  abstract tagged limited private ;
```

Производный тип от лимитируемого типа является лимитируемым ; составной тип - лимитируемый , если тип хотя бы одной из его компонент является лимитированным. Лимитируемый тип может быть помечен как тэговый. В этом случае возможно личное расширение, результатом которого является также личный тип.

D Например:

```
package  ASTRA  is

   type   T   is   tagged;                      - тэговый тип
   type   TN   is   new   T with  private; -    - расширение типа  T
```

```
private                                  - - личная часть спецификации
   type TN is new T with  - - детализация производного типа TN
   record
         Z :  integer; - - добавление новой компоненты
      end record ;

end ASTRA;
```

8.4 Тело пакета

Тело пакета обеспечивает реализацию ресурсов пакета, перечисленных в его спецификации:

```
PACKAGE BODY  Имя_Пакета  IS
    - - Локальные_Описания [
BEGIN
        - - Последовательность_Операторов [
EXCEPTION
        - - Оработчики_Исключений ] ]
END Имя_Пакета;
```

Локальные_Ошсания в теле пакета предназначены для реализации ресурсов, перечисленных в спецификации пакета, а также для описания локальных ресурсов, необходимых для их реализации. Локальные описания в теле пакета являются невидимыми и недоступными вне пакета.

Последовательъность_Операторов в теле выполняется только при *предвыполнении* пакета. Как правило, предвыполнения тела пакета связаны с инициализацией объектов, описанных в спецификации.

Обработчики_Исключений предназначены для обработки исключений, возникающих при выполнении операций в теле пакета. Если обработчик отсутствует в теле пакета, то исключение распространяется на часть программы, где находится пакет.

П Пример тела пакета Data , спецификация которого описана выше


```
package  body  DATA  is

   VA, VB :  Vector; - -  локальные  переменные и тип sybtype
   Short_Vector is Vector ( 1 . .  10 ) ;

   procedure  Sum_Vec (A, B : in  Vector;
                                          C : out Vector) is
   begin
      for i in 1 .. n  loop
     C ( i ) : = A ( i ) +  B ( i ) ;
   end  loop; end  Sum_Vec;

   procedure Clean_Vec ( Z:  in  out  Vector)  is - - локаль-
   begin                                          - - ная процедура
   for j  in 1 .. N  loop
     Z ( j ) :  = 0; end
  loop; end
  Clean_Vector ;

begin  - - операторная  часть тела пакета

   Clean (VA);  - - инициализация  локальных  переменных
   Clean (VB );

end  DATA;
```

ffl СОВЕТЫ:

* Используйте пакеты для обьединения логически связанных типов и объектов.
* Используйте пакеты для сокрытия информации.
* Используйте пакеты с тэговыми и приватными типами для абстрактных типов данных.
* Используйте спецификатор **renames** вместо спецификатора **use** для пакетов.
* Для обеспечения видимости операторов используйте спецификатор **use type** вместо **renames.**
* Минимизируйте использование спецификатора use .

8.5 Контролируемые типы

Контролируемые типы (**controlled types**), позволяют пользователю определить, что происходит с обьектами в начале и в конце их цикла существования.Для таких типов можно определить операцию инициализации, автоматически вызываемую, когда обьект предвыполняется, и операцию *финализации* - когда обьект становится недоступным . Контролируемые типы обеспечивают средства программирования динамических структур данных, эффективного использования памяти и других ресурсов.

В языке предопределены контролируемые типы в предопределенном в пакете **Ada** . **Finalization** . При манипуляции с обьектами три действия являются фундаментальными:

- *инициализация* (initialization)
- *инализация* (finalization)
- *присваивание* (assignment).

Каждый обьект инициализируется после создания (например, при обьявления в описании).

Каждый обьект финализируется перед уничтожении (например, при выходе из тела подпрограммы, содержащей описание обьекта).

Операция присваивания является частью оператора присваивания, передачи параметов, инициализации и др.

Эти три фундаментальных операции автоматически обеспечиваются языком. Использование дополнительно типа **Controlled** предоставляет пользователю воможность самостоятельно управлять некоторыми из этих операций. В частности, контролируемые типы могут быть производными от двух тэговых типов, определенных в пакете **Ada** . **Finalization :**

```
type Controlled is abstract tagged private;
type Limited_Controlled is abstract
                              tagged limited private;
```

Для этих типов в пакете определены процедуры Initialize, Finalize Adjust. П Пример обьявления контролируемого типа с помощью пакета Ada
. Finalization :

```
type Strong is new Controlled with .. - ;
```


Спецификация пакета Ada . Finalization :

```
package Ada . Finalization is
pragma Preelaborate ( Finaiization );
   type Controlled is abstract tagged private ;
   procedure Initialize (Object    in out Controlled );
   procedure Adjust     (Object    in out Controlled);
   procedure Finalize   (Odject    in out Controlled );

   type Limited_Controlled is
   abstract tagged limited private;

   procedure Initialize { Object : in out Limited_Controlled );
   procedure Finalize (Object : in out Limited_Controlled) ;
private
        . . .  -  не определено в языке
end Ada . Finalization ;
```

8.6 Примеры

П Пример 1 .

Пакет с пассивными ресурсами в виде объектов, не требующий тела.

```
package OBJECTS is

   N          : constant : = 50 ;
   X . Y . Z  : fixed;
   V  : array (1 .. N ) of fixed ;   - - обьекты анонимного типа
   W : array (1 .. N , 1 .. 25 ) of float;
   Flag  : boolean : = True ;

end OBJECTS;
```

D Пример 2.

Пакет с пассивными ресурсами в виде совокупности переменных и типов **package** WORK **is**

```
   type Units is delta 0.01 range 0.0. . 16.0;
```

```
    type    Elem  is integer  range  - 40 . .   77;
    VA, VB   :   Elem;
    Z_U      :   constant  Units : = 2 .44 ;
end  WORK;
```

D Пример 3

Пакет с пассивными ресурсами и приватной частью.

```
package  POLICE  is
    type Kent  is  private;
    type Art   is  limited  private;
    Max_Kent :      constant  Kent; constant
    Size     :      integer: = 99 ;
    type  Place  is   array (2 .. Size, 10 .. Size )   of  Kent;
    type  Pipe   is   array (- Size .. Size )  of      Art
private
    type  Kent  is    integer  range   -8 .. 32
    type  Art   is    new  float;
    Max_Kent  :                    constant
Kent : = 16 ; end   POLICE;
```

В пакете Police в качестве ресурсов описаны два личных типа Kent и Art, а также субконстанта Max_Kent. В спецификации пакета присутствует личный (невидимый из-вне пакета) раздел описания ресурсов после слова **private** , в котором для компилятора задаются подробности реализации приватных типов и субконстант и {в соответствии с которыми они использовались в теле пакета, если бы оно тре-бовались).

D Пример 4.

Пакет с активными ресурсами.

```
package  PAGE_WORK  is
    type Elem  is private
    record
        X   Positive  ;
        Y   Positive;
        D   Character
    end  record ;
    procedure Clean      (Object :   in   Elem)
```



```
procedure In_Page  ( Object :   in  Elem ); procedure Out_Page (
Object:   out  Elem ); private Error_Page :   exception ; - - исключение
для обработки ошибок
                                  - - в процедурах
end  PAGE_WORK ;                    - -  Clean , In_Page ,  Out_Page

package body  PAGE_WORK  is :=
    Nx : constant  60; := 45; (1 ..Nx, 1
    Ny: constant  .. Ny)
    Page : array  function          of  Elem ;  View_Page  return
                                    Boolean  is
                  end  View_Page;
    procedure Clean    ( Object:  in   Elem )  is

    end  Clean;
    procedure In_Page  ( Object:   in   Elem )  is

    end  In_Page;
    procedure Out_Page (Object; out  Elem )  is

    end Out_Page; end
 PAGE_WORK;
```

Пакет Page_Work обеспечивает пользователю ресурсы для работы со страницами символов с помощью процедур :

- очистка страницы (Clean)
- вставка вимвола на страницу (In_Page)
- выборка символа из страницы (Out_Page)

В теле пакета для реализации этих ресурсов добавлена процедуры-просмотра страницы View_Page, а так же две константы NX, Ny, дающие размер страницы, и вспомогательная переменная Page.

Глава9. З А Д А Ч И

Созданный более 15-и лет назад язык Ада уже тогда включал средства *параллельной обработки*, что выделяло его среди других универсальных языков программирования. Сегодня, когда можно говорить о начале широкого практического применения параллельных вычислительных систем, наличие в Аде средств программирования параллельных процессов придает языку еще большую ценность.

Язык предоставляет развитые средства для программирования параллельных процессов. Программа представляется в виде модулей, которые могут выполняться параллельно на множестве процессоров или на одном процессоре в режиме разделения времени. При этом модули могут взаимодействовать между собой, синхронизироваться или обмениваться информацией. Такие модули в Аде называются *задачами.* Они предназначены для реализации *абстракции процесса.* Понятие задачи *(процесса)* является фундаментальным в языке, так как выполнение Ада программы - это выполнение одной или нескольких задач. При этом каждая задача имеет собственную нить управления, выполняется независимо и параллельно.

Задачи, как и остальные модули, имеют две части: *спецификацию* и *тело* задачи. Однако задача не является компилируемым модулем. Поэтому задачи размещаются в описательной части пакета или подпрограммы.

В Аде 95 средства параллельной обработки получили дальнейшее развитие в двух направлениях. Это во-первых, появление в языпсе еще одного механизма работы с процессами - *защищенных* модулей, а во-вторых - специальное приложение к языку "Распределенные системы". Защищенные модули будут детально рассмотрены в Главе 13.

В настоящем разделе рассматриваются средства работы с *заданными модулями* **(task units).**

9.1 Спецификация задачи

Спецификация задачи (**task**) задаёт имя задачи и содержит описания *входов* задачи и *спецификаторов представления:*

TASK [TYPE] *Идентификатор_задачи*
[Дискретный диапазон] IS

- - *Описание Входов*

- - *Спецификатор_Представления*
[PRIVATE
- - *Описание_Входов*
- - *Спецификатор представления]*

END *Идентификатор _задачи;*

О Примеры: **task** Metro ; - - задача с вырожденной спецификацией

```
task Stack is
   entry Read (X: out Elem ); - - входы для приёма
                                - - и передачи
   entry Write (Y: in Elem); - - информации от
                                - - других задач
private
   entry Get (Z: in integer);
end Stack;
```

```
task Semaphore is
   entry P;
   entry V;
   for P use at Spec_Address; end
Semaphore;
```

задача имеет входы только для синхронизации - - спецификатор адреса

Наряду с описанием единичной задачи в Аде разрешается описание множества задач через использование *задачного типа.* О Например:

```
task type Silk; - - заданный тип

A, B : Silk ;      - - создание двух одинаковых задач A и B типа Silk Top :

array (1 ..5) of Silk; - - создание массива задач Top(1 . .5)
```

Использование сылочного типа для создания задач типа Silk

```
type Cotton is access Silk ; Event :
Cotton : = new Silk ; Anchor :
Cotton ;
```

Задачный тип является *лимитированным приватным* типом, для которого запрещены все операции.

В задачном типе в языке допускается использование *дискримина-тов*, что предоставляет дополнительные удобства при работе с задач-ным типом. Механизм дискриминантов позволяет параметризировать задачи при их описании. Например, с помощью дискриминанта можно идентифицировать (внутренне) задачи, создаваемые на основе задачно-го типа:

```
task type Node( Name :   Character ) ;    - - задачный тип
                                                  - -  с дискриминантом
Z1 Node( A); Node( B); Node( C);   создание задач
Z2
Z3 task  body  Node  is    - -                тело
   задачного типа begin
   put( Name); case
   Name do A= >    ...
        B = >    ...    - - обработка дискриминанта в теле задачи C
        = >    ...

end  Node;
```

Здесь созданы три одинаковые задачи Zl, Z2, Z3 , однако их поведение можно сделать разным за счет анализа в теле задачного типа дискриминанта Name , который у всех трех задач разный.

С помощью дискриминанта задачи можно связывать с данными, встраивая задачи в данные, или данные, логически связанные задачами, размещать вне задач, используя ссылочный дискриминант.

```
type  Box  is
   record
      Z:    integer;
      X:    integer;
   end  record;

task  type   Monterey ( Mr : access Box );
```

Теперь тип Box можно использовать в теле задачного типа Monterey.

Ш СОВЕТЫ:

* Используйте дискриминант для инициализации задач.
* Используйте дискриминант для установления приоритета задачи, размера семейства входов, размера памяти, неодходимого для задачи.
» Используйте дискрминант для указания данных, ассоциируемых с задачей.
* Минимизируйте динамическое порождение задач.

9.2 Тело задачи

Тело задачи **(task body)** определяет действия задачи при её выполнении. Тело задачи имеет следующую форму:

```
TASK  BODY   Идентификатор_Задачи   IS
        - - Описания
BEGIN
        - - Последовательность_0ператоров [
EXCEPTION
        - - Обработчики_Исключений ]
END  Идентификатор_Задачи;
```

Тело задачи аналогично телу подпрограммы. Его отличительная особенность - наличие оператора приема **accept** в случае , если в спецификации задачи описан соответствующий вход. Локальные описания в теле задачи могут содержать описания вложенных задач :

```
task  body  Metro   is task
   Station  is
      entry Way_A (T : in Train ); end
   Station ; task  body   Station   is
        —— TiTrvTwjvrm  A.  B.
```

```
    Port :   Data ;
  begin

    accept  Way_A (T: in Train ) do
       Port: = Train ;
    end  Way_A;

  end  Station;
begin
    . . .
end  Metro;
```

9.3 Взаимодействие задач .Механизм рандеву

Задачи при выполнении могут *взаимодействовать*. Взаимодействие задач в языке включает :

- синхронизацию задач;
- передачу информации между задачами.

Механизм взаимодействия задач построен на концепции последовательных взаимодействующих процессов (Communication Sequential Processes - CSP), предложенной Хоаром Н. [15, 16]. Программа в рамках CSP - набор последовательных процессов (задач), выполняющихся параллельно. Взаимодействие процессов осуществляется с помощью операций ввода-вывода. Во время взаимодействия процессы могут обмениваться информацией, при этом взаимодействие - синхронизованное, то есть обмен выполняется только тогда, когда каждый процесс достигает соотвествующего оператора ввода-вывода, в противном случае процессы дожидаются друг друга (блокируются).

В Аде механизм CSP получил дальнейшее развитие и реализацию в виде *механизма рандеву* (rendezvous mechanism).

Взаимодействие задач в языке с использованием механизма рандеву выполняется следующим образом.

1. Две задачи могут взаимодействовать тогда и только тогда, когда хотя бы в одной из них имеется описание *входа* (**entry).**
2. Взаимодействие задач заключается в том, что одна задача *вызывает вход* другой задачи, указывая имя задачи и входа вместе с фактическими параметрами.
3. Задача, имеющая описание некоторого входа, может *принимать и обрабатывать* вызов этого входа из другой задачи.

4. Если обе задачи (вызывающая и принимающая вызов) одновременно выполняют действия, связанные с взаимодействием, то между ними устанавливается связь посредством *рандеву*.
5. Если при выполнении взаимодействия, одна из задач начнет выполнять свои действия по уставновлению связи раньше другой (вызывать или принимать вызов), то эта задача простанавливается (блокируется) до тех пор, пока другая задача не начнет выполнять свои действия, необходимые для взаимодействия, таким образом задачи синхронизируют свои действия при взаимодействии.
6. Во время рандеву задачи выполняют действия, связанные с передачей (приемом) данных, причем передача информации может выполняться в обеих направления (обмен).
7. После завершения рандеву задачи продолжают свое выполнение.
8. Во время рандеву задачи могут не передавать информации, в этом случае рандеву используется только для синхронизации задач, хотя в общем случае рандеву объединяет и синхронизацию и передачу информации между задачами.
9. Механизм рандеву в Аде является *ассиметричным* , то есть задача, вызывающая вход другой задачи, должна знать и явно указывать имя вызваемой задачи. Задача, принимающая вызов своего входа, не должна знать (и указывать) имя обращающейся к ней задачи. Такая разновидность рандеву эффективна для организации взаимодействия одной задачи с несколькими, например в модели Клиент-Сервер, когда задача Сервер может обслуживать любые задачи - Клиенты, не зная их имен.
10. Если несколько задач вызвали один и тот же вход некоторой задачи, то эти задачи приостанавливаются , становятся в очередь к данному входу и будут обслуживаться вызванной задачей либо в порядке поступления вызовов, либо по приоритетам. После обслуживания вызова задача удаляется из очереди. Для каждого входа формируется своя очередь. Задаче разрешается находиться только в одной очереди по вызову входа.

Реализация механизма рандеву в языке выполнена с использованием понятий *описание входа* (**entry**), *оператор вызова входа* и *оператора приема (принятия)* **accept** Дополнительные возможности программирования механизма рандеву обеспечивает использование оператора *выбора* (**select,),** *задержки* **(delay**), *прекращения* (abort) и *упорядочивания очереди* (**requeue).**

n*

9.4 Входы задач и операторы принятия

Понятие *входа* в задаче предназначено для организации взаимодействия задач. Внешне описание входа аналогично описанию подпрограммы :

ENTRY *Имя_Входа [(Индекс_Входа)]*
[(Раздел_Формальных_Параметров)] ;

D Пример описания входов :

```
entry Wait;  - - вход без параметров только для синхронизации
entry Bus ( 1 .. 4 );                  - - семейство входов
entry Data (X : in integer); - - вход для приема информации
entry Resoult (X , Y : in Vector; Z : out Matrix );
entry Buffer_Read ( Number) ( Element : out Item ); - - семейство
                                                        - - входов
```

Описание входов задачи выполняется в ее спецификации. П
Пример спецификации задачи с описанием ее входов :

```
task Station is
   entry Way_A(X: in Train )
   entry Way_B(Y: out Train);
end Station;
```

Входы задач могут быть описаны как *семейство входов.* Семейство входов образуют несколько входов задачи, имеющиех одно имя и различающихся индексом, который указывается при вызове необходимого входа мз семейства. D Например:

```
task Airoport is                                  - - 10 входов
   entry Plan_Place (1 10)(P: in out Plain );
end Airoport ;
```

Раздел формальных параметров в описании входа может отстутствовать, если вход используется только для синхронизации задач без передачи инфоормации. Описание раздела формальных параметров входа аналогично описанию процедур, то есть необходимо указать имя параметра, его вид (in , out, in out) и тип. Вид параметра здесь определяет направление движения значения этого параметра при взаимодействии задач:

- из вызывающей задачи в принимающую (in);
- из принимающей задачи в вызывающую (out).

Взаимодействи задач основано на вызове входов задач. *Оператор вызова входа* подобен оператору вызова процедуры. В вызывающей задаче указывается *составное имя* , включающее имя вызываемой задачи и имя входа, а также индекс для семейства входов. Далее размещается список фактических параметров, если они имеются. D Например :

```
Station . Way_B ( Berlin );
Airoport. Plain_Place (k ) ( New_Raice);
```

Согласование формальных и фактических параметров в операторе вызова входа осуществляеттся также как и для подпрограмм.

Оператор принятия (accept) задает действия, которые выполняются вызываемой задачей при вызове данного входа. Оператор принятия неразрывно связан с соответствующим описанием входа, то есть если в спецификации задачи описаны входы, то в теле этой задачи могут быть использованы операторы принятия для этих входов. Оператор принятия определяет действия принимающей задачи во время рандеву.

Общий вид *оператора принятия:*

```
ACCEPT Имя_Входа [ ( Индекс_Входа ) 1
                                  [ ( Раздел _Параметров ) ] [ DO ]
   . . .  - - Последовательностъ_Операторов
[ EXCEPTION)                              - - тело оператора приема
   . . .  - -Исключения
END [ Имя_Входа };
```

Индекс_Входа связан с семейством входов. *Раздел_Параметров* должен совпадать с разделом параметров в описании соответствующего входа.

Использование оператора принятия допускается только в теле соответствующей задачи. D Например:

```
accept   Receiver;    - - тело отсутствует ( вырожденное )

accept  In_Box(X:  in Integer; Y : in   Float  ) do
Data   : =   X ; Temp : =
Y ; end   In_Box ;

accept  Fast ;

 accept  Port  ( Id) ( From : in Data ) do
     X : =  From ;
 end   Por t;

 accept  Gent   do
     put ( "Task is called"); end
 Gent;

accept   Base ( Z : inout  integer ) do

exception
     when Constrain_Error   = >
          2: = Integer' Last; end
Base;
```

Тело задачи может содержать несколько операторов принятия одного и того же входа. Тело оператора приема может быть пустым, либо содержать несколько операторов, даже если вход не имеет параметров . В теле разрешается использование операторов **return** и **requeue,** вызывать подпрограммы, внутри которых имеются операторы вызова входов других задач.

Выполнение оператора приема начинется с вычисления индекса входа (если рассматривается семейство входов). Этот индекс определяет, какой из входов семейства должен выполняться. Затем выполнение тела *блокируется* до тех пор, пока не произойдет вызов соответствующего входа в вызывающей задаче. После этого через фактические параметры принимаются значения из вызывающей задачи, выполняется последовательность операторов из тела оператора **accept** и передаются значения из принимающей задачи в вызывающую.

Вход задачи считается *открытым*, если задача блокирована на операторе приема, соответствующему данному входу (или на операторе отбора **Select** с открытой альтернативой). Иначе вход считается *закрытым.*

При вызове открытого входа в принимающей задаче выполняется тело соответствующего оператора приема. Если вызываемый вход закрыт, то вызов данного входа добавляется в очередь к этому входу, а задача - блокируется до тех пор, пока вызов не будет исключен .

Вызов входа *завершенной* задачи вызывает исключение **Tasking_Error** в точке вызова.

Ш СОВЕТЫ:

•. Минимизируйте время выполнения тела оператора приема **Accept**
± Минимизируйте количество операторов **Accept и Select** в теле задачи.
•. Минимизируйте количество опреаторов **Accept**, связанных с одним входом. » Чаще используйте обработку исключения **Tasking_Error.**
• Используйте механизм рандеву, если создаваемые приложения требуют взаимодействия задач без буферизации.

9.5 Оператор перенаправления очереди

Оператор *перенаправления очереди* **Requeue** используется для завершения оператора **accept** *или тела входа* при необходимости перенаправления соответствующего вызова входа в новую (или ту же самую) очередь . Такое перенаправление может быть выполнено с (или без) исключением вызова с помощью оператора **abort:**

REQUEUE *Имя_Входа*
(WITH ABORT] ;

Оператор **requeue** должен размещаться в вызываемой конструкции (теле входа, операторе приема). При выполнении оператора **requeue** сначала вычисляется имя входа, а так же префикс, идентифицирующий исходную задачу или защищенный модуль, и выражение, идентифицирующее вход в семействе входов.

D Примеры:

```
requeue   Write  with abort;
requeue   Read ( k);
```

9.6 Операторы задержки

Оператор задержки (**delay**) используется для блокирования дальнейшего выполнения задачи на указанную длительность времени. Время может задаваться двумя способами: в секундах от текущего времени в операторе **delay** и в виде конкретного времени в операторе задержки **delay until.** D Например:

```
delay 2*X +4;
                    - вычисление выражения, определяющего
delay 4.7  -- время задежки
delay :  9.15 -- задержка на 4,6 секунды
                - задержка до 9 часов и 15 минут
```

9.7 Оператор прекращения

Оператор прекращения задачи **abort** предназначен для аварийного завершения одной или нескольких задач. При этом любые дальнейшие действия с такими задачами запрещены.

Общий вид оператора прекращения :

ABORT *Имя_Задачи [, Имя_Задачи]*;

Прекращение нескольких задач, перечисленных в операторе **abort,** выполняется в произвольном порядке. При прекращении задачи, любая вложенная в нее задача тоже прекращается, исключая операции, для которых операция прекращения отсрочена (**abort** - отсроченные операции) . К таким операциям относятся :

- защищенная операция ;
- ожидание окончания вызова входа ;
- ожидание завершения зависимых задач ;
- выполнение присваивания и процедур **Initialize и Finalize** для типа **Controlled;**

При прекращении задачи все зависящие от нее задачи прекращаются.

СОВЕТЫ:

Используйте оператор **abort** только в крайних случаях требующих безусловного завершения задачи

9.8 Операторы отбора

Существует четыре формы *оператора отбора* (**select**) . Первая обеспечивает *селективное ожидание* (отбор с ожиданием) одной или нескольких альтернатив. Две других обеспечивают *временной (таймированный)* и *условный вызовы входа.* Четвертая - обеспечивает *асинхронную передачу управления.* Назначение оператора отбора - предоставить пользователю развитые средства программирования взаимодействия задач.

9.8.1 Селективный отбор

Эта форма оператора **Select** позволяет обьединить ожидание и выбор одной или нескольких альтернатив. Отбор может зависеть от условий, связанных с каждой альтернативой приема с отбором :

```
SELECT
      [ Защита]
         Альтернатива_Отбора
(OR
      [ Защита ]
         Альтернатива_Отбора }
ELSE
        Последовательность_Операторов
END SELECT;
```

Здесь *Защита* - выражение вида **WHEN** < *Условие* > = >.

Альтернатива_Отбора - это:

- *Альтернатива_Принятия* (**ACCEPT)** *[Последовательность^Операторов]*
- *Альтернатива_Задержки* (**DELAY)** *[Последовательность_Операторов]*
- *Альтернатива_Завершения* (**TERMINATE) .**

Эта форма оператора Select должна содержать по крайней мере одну *Альтернативу_Принятия.* В дополнение он может содержать одну *Альтернативу_Завершения* или одну (несколько) *Альтернтив_Задержки*, либо раздел **else.**

Альтернатива_Отбора считается *открытой,* если она не имеет защиты, или значение *Условия* - **True.** В противном случае альтернатива называется *закрытой.*

Порядок выполнения оператора **Select с** отбором :

- в произвольном порядке вычисляются *Условия*, заданные в защите, то есть определяются *открытые альтернативы;*
- для *Открытых_Альтернатив _Задержки* вычисляются длительности задержки;
- для *Открытых _Альтернатив_Пришятия* входа семейства вычисляется индекс входа;
- выполняется одна из *Открытых_Альтернатив,* либо раздел **else.**'

Первыми рассматриваются открытые *Алътернативы_Принятия.* Отбор одной из них производится немедленно, если возможно соответствующее рандеву (то есть вход вызван другой задачей). Для нескольких *Открытых_Алыпернатив* отбор производится в соответствии с тактикой обработки очередей во входах.

Если никакое рандеву не возможно немедленно и отсутствует раздел **else,** то задача блокируется до тех пор, пока можно будет выбрать открытую альтернативу.

Отбор других форм альтернатив или раздела **else** осуществляется следующим образом:

- отбирается раздел **else** и выполняется последовательность операторов, если нельзя немедленно отобрать *Альернативу_Принятия*, в частности, если все альтернативы закрыты;
- отбирается открытая *Альтернатива_Завершения.*

9.8.2 Таймированный вызов входа

Таймированный вызов входа предназначен для вызова входа, который отменяется, если вызов входа не может быть обработан (принят) прежде, чем истечет заданное временное ограничение:

```
SELECT
        - - Оператор<_Вызова_Входа
        - - [Последовательностъ_Операторов ]
OR
```



```
        - - Оператор задержки
END SELECT;
```

Временной (Таймированный) вызов входа производит вызов входа, который отменяется, если рандеву не началось на протяжении указанного в операторе delay интервала времени. При выполнении данного вида оператора Select сначала вычисляется имя входа и фактические параметры, а затем вычисляется величина задержки в операторе **delay** и , наконец, производится вызов входа.

D Пример:

```
select
    Buffer. Read ( Element); or
    delay 3.44 ;  - - время ожидания принятия входа
end  select;
```

9.8.3 Условный вызов входа

Условный вызов входа выполняет вызов входа, который отменяется, если рандеву нельзя произвести немедленно:

```
SELECT
        - - Оператор'_Вызова
        - - [ Последовательность_Операторов ]
ELSE
        - - Последовательность_Операторов END
SELECT;
```

При выполнении оператора Select данного вида сначала вычисляется имя входа и его фактические параметры. Вызов входа отменяется, если вызванная задача не достигла соотвествующего оператора **accept** , и выполняются операторы в разделе **else** . Если рандеву происходит, то выполняется также и последовательность операторов, указанная после вызова входа. **Select**

```
    Buffer. Write ( Data ); Pool: =
    Data + 100.0 ; else
        Pool: = - 97.88 ;
```

```
end select ;
```

Ш СОВЕТЫ:

* Условный вызов входа может изменить атрибут **Count** для входа, даже если условный вызов входа не был выбран.
* Используйте оператор отбора **Select** с альтернативой **terminate.**

9.8.4 Асинхронная передача управления

Асинхронный оператор Select обеспечивает асинхронную передачу управления при завершении вызова входа или истечении времени задержки:

```
SELECT
    - - Защелкшающая_Альтернатива
THEN ABORT
    - - [ Последовательность_Операторов ( Завершающая часть) ]
END SELECT;

SELECT
    - - Вызов входа или оператор задержки
      [ Последовательность операторов ]
THEN ABORT
    - - Последовательность операторов ( Завершающая часть ) END
SELECT; D Пример:

select
   Monitor. Read ;
   X : = X +
1 ;        then
abort
   Z : = S (i);
end select;

select
   delay 1.5;
   put (" Тазк
                  is delayed")
then abort;
   Res : = FUNC( A,B );
end select;
```


- это вычисление должно закончится
- - за 1,5 сек

Если используется вызов входа и вызов откладывается в очередь, то выполняется последовательность операторов после **then abort**. Если вызов входа отобран сразу, то последовательность операторов после **then abort** не начинается.

Ш СОВЕТЫ:

ф Используйте асинхронный оператор **Select** вместо оператора **abort.** * Минимизируйте использование асинхронного оператора отбора **Select.**

9.9 Выполнение и зависимость задач

Выполнение программы на языке Ада - это выполнение одной или нескольких задач. Каждая задача имеет собственную нить управления, выполняется независимо и параллельно, взаимодействуя (явно или неявно) с другими задачами.

Выполнение задачи - это выполнение тела задачи. Инициация этого выполнения называется *активизацией* задачи и включает предвыполнение описания тела.

Задача в Аде - аналог понятия процесса и может находиться в следующих состояниях:

- гототвности (ready)
- выполнения или активности (run)
- блокированиея (blocked)
- завершения (terminate)
- пассивном (inactive).

Создание задачи (задачного обьекта) может быть выполнено :

- как часть предвыполнения описания обьекта, размещенного непосредственно внутри некоторой области описания;
- как часть выполнения генератора для ссылочных типов.

Все задачи независимо от вида их создания активизируются *одновременно.* При этом задача, порождающая и активизирующая новые задачи, *блокируется* до тех пор, пока не закончится активизация всех созданных ею задач.

D Например :

```
procedure Control is
  task type Recources;  - - заданный тип TA, TB : Resources ;
  - - создание двух задач TX : array ( 1'.. 3 ) of Resources; - -
  создание массива
                                                    - - задач
type Normal is access Resources; - - ссылочный тип F :
Normal : = new Resources ; - - активизация F . all begin
  - - в этом месте процедура Control блокируется пока не активизи-
  - - руются (одновременно) все задачи из описания процедуры :
  - - TA, TB , F, TX ( 1 ) .. TX(3).
  - - начало выполнения тела процедуры
end Control;     - - ожидание завершения всех активизированных - -
                   задач
```

Различают *нормальное* и *ненормальное* завершение задачи. Нормальное завершение - при достижении end или при передаче управления операторами **exit, return, goto, requeue** или при выборе альтернативы завершения. Ненормальное завершение - когда управление передается из конструкции через оператор **abort** или при возбуждении исключения.

Взаимодействие задач включает:

- активизацию и завершение задач
- вызов защищенной процедуры из защищенного модуля
- вызов входа другой задачи.

В языке различают понятия *задача заканчивается* и *задача завершается.* Задача закончилась (**completed**) (закончила свое выполнени) - если выполнено ее тело; в теле возбуждено исключение, но не имеется соответствующий обработчик исключения, а при наличии обработчика - по окончанию его выполнения. Задача завершена (terminated) - если она закончила свое выполнение и нет зависимых от нее задач.

Ш СОВЕТЫ:

- Не используйте незавершающиеся задачи.

9.10 Атрибуты задач

Если T - некоторый задачный тип, то для него определены следующие атрибуты:

- **T'Callable** - Вырабатывает значение True если задача вызвана, иначе вырабатывает значение **False.** Задача считается в состоянии вызова, если она не завершена или аврийная . Значение атрибута имеет тип **Boolean.**
- **TTerminated** - Вырабатывается значение **True** , если задача завершена, иначе - **False.** Значение атрибута имеет тип **Boolean.** Если E - имя входа (семейства входов) задачи или задачного типа, то для него определен атрибут E'Count. Этот атрибут допускается только внутри тела задачи или защищенного модуля; исключение -вход задачи в другом программном модуле, который находится в свою очередь в теле задачи.
- **E'Count** - Вырабатывает количество вызовов данного входа, присутствующих в очереди к входу E. Значение атрибута имее тип универсальный целый.

Ш СОВЕТЫ:

- Избегайте в задачах зависимость от значения атрибута входа **Count**

9.11 Разделяемые переменые

В языке в задачах могут использоваться разделяемые (общие) переменные.

Разделяемые (shared) переменные - это переменные (обьекты) в общей памяти, которые используются для взаимодействия задач и доступ к которым для чтения или изменения возможен из нескольких задач в программе.

Так как при выполнении программы возможна ситуация, при которой задачи одновременно обращаются к разделяемым переменным, то во избежание конфликтных ситуаций и корректности работы программы необходимо обеспечить *синхронизированный доступ* задач к разделяемым переменным. Синхронизированный доступ к разделяемым переменным - фундаментальная задача параллельного программирования, называемая *задачей взаимного исключения.* Ее решение основывается на том, что в любой момент времени доступ к разделяемой переменной должен предоставляться только одной задаче. В языках программирования для организации синхронизированного доступа ис-

пользуются специальные программно-аппаратные средства: семафоры, критические участки, мониторы и др.

Ада 83 обеспечивала управляемый доступ к разделяемым переменным с помощью прагмы **Shared,** которая гарантировала неделимость операций чтения и изменения над ними. К сожалению, область применения прагмы **Shared** была ограничена. В частности, прагму нельзя было использовать для компонентов массива, а сам массив требовал работы с ним только через ссылочный тип.

По этой и другим причинам, в Аде 95 прагма Shared заменена на две прагмы **: Atomic и Volatile.**

Pragma ATOMIC (*Локальное_Имя*) ;
Pragma ATOMIC_COMPONENTS (*Локальное_Имя_Массива*) ;
Pragma VOLATILE (*Локальное_Имя*) ;
Pragma VOLATILE_COMPONENTS (*Локалъное_Имя_Массива*) :

Здесь:

- *Локалъное_Имя* - описание обьекта или описание типа.
- *Локальное_Имя_Массива* - описание типа массива или обьекта, имеющего анонимный тип массива.

Прагма **Atomic** для всех обьектов, указанных в ней (atomic - обьектов), обеспечивает неделимость операций их чтения и изменения. Операции над atomic-обьектами выполняются только последовательно.

Прагма **Volatile** для обьектов, указанных в ней (volatile - обьектов), обеспечивает выполнение операций чтения и изменения над ними непосредственно в памяти.

П Примеры:

```
Size : integer; pragma
Atomic (Size);

Flag : boolean; pragma
Volatile ( Flag );
```

```
type Elem is new fixed ;
pragma Atomic ( Elem ) ;
```



```
type Vector is array (1 .. N) of integer;
pragma Atomic_Components ( Vector);
```

```
Matrix : array (1 .. 10 , 1 .. 20 ) of fixed ;
pragma Volatile _Components ( Matrix);
```

Разделяемые переменные и прагмы для них можно использовать в языке для:

- взаимодействия задач;
- взаимодействия Ада программ с другими процессами, не являющимися Ада программами;
- организации управления устройствами в Ада программах.

Щ СОВЕТЫ:

- Не используйте не защищенные прагмами разделяемые переменные для синхронизации задач.
- Используйте для разделяемых переменных прагмы **Atomic и Volatile.**

9.12 Приоритеты задач

Каждая задача может иметь приоритет, который задается с помощью прагмы **PRIORITY:**

pragma PRIORITY(*Выражение);*

Использование данной прагмы разрешено непосредственно в спецификации задачи, защищенного модуля или описательной части подпрограммы. *Выражение* должно иметь подтип **Priority** целого типа, определенного в пакете **System.**

Приоритет задачи определяет ее поведение при состязании с другими задачами за системные ресурсы, например центральный процессор. Задача, имеющая больший приоритет, получает преимущества. Если две задачи с разными приоритетами готовы к выполнению, то первой выполняется задача с более высоким проиоритетом. Если задачи имеют одинаковый приоритет (или он не задан), то порядок выполнения таких задач не определен.

В Аде 83 приоритет задачи фиксировался при ее описании и не мог меняться, то есть был статическим. В Аде 95 приоритет задачи может быть изменен в процессе ее существования (кроме приоритета, определенного для подпрограммы), то есть является динамическим.

Р Например :

```
task  Police is
   pragma Priority (10); end
Police;

task Save is
   entry  Receive (X : in Elem );
   pragma   Priority ( N + 2 );
end  Save;
```

Развитый механизм планирования выполнения задач, основанный на использовании статических и динамических приоритетов, в языке обеспечивается средствами, определенными в Приложении "Системы реального времени".

УПРАЖНЕНИЯ:

- Используя прагму **Priority** , исследуйте порядок запуска задач.

СОВЕТЫ:

* Используйте задачи для описания параллельных алгоритмов.

* Не полагайтесь на прагму **Priority** ; отдавайте предпочтение средствам языка для разработки систем реального времени

ИЗМЕНЕНИЯ:

О Спецификация задачного типа может иметь приватную часть.

О В описании задачного типа можно использовать дискриминант.

О Новый оператор **Requeue.**

О Новая форма оператора отбора **Select.**

О Новая форма оператора завершения **Abort.**

CJ Новая форма оператора задержки **Delay until.**

О Новые прагмы для разделяемых переменных : **Atomic и Volatile.**

О Изменены праила работы с приоритетами задач.

9.13 Примеры

Программа моделирует работу системы, включающей завод, склад и магазин. Завод производит продукцию и отправляет ее на склад, откуда ее забирает магазин для реализации. Указанные обьекты реализованы в программе в виде соответствующих задач. Задача , моделирующая работу склада, контролирует количество товара на складе, не допуская его переполнения. Кроме того, она не допускает отпуск товара для магазина, если склад пуст.

```
procedure  Factore_Shop_Warenhouse is task
   Factory;

   task Shop;

   task   Warenhouse is entry in   Data);
      In_Store   (   X   entry out Data);
      Out_Store ( X ;
   end   Warenhouse;

   task  body  Warenhouse  is
   N :      constant integer: =  50 ; Pool  :
   array ( 1 .. N  )  of  Data; Count:  integer: =
   0 ; Injndex, Outjndex  :  integer: = 1 ;
   begin loop select
            when  Count <  N  =>
                  accept  In_Store( X:   in   Data ) do
               Pool (Injndex ): = X : end
         In_Store; Injndex : = Injndex + 1 ;
         Count: = Count + 1 ; or
            when Count  >  o  = >
                  accept  Out_Store ( X : out Data) do
                     X : = Pool ( Outjndex );
               end   Out Store:
```

```
   Outjndex : = Outjndex + 1 ; Count: =
   Count + 1 ; end select; end   loop; end
   Warenhouse;

   task body  Shop  is
   TV :  Data;
   begin loop
         Warenhouse (TV)

      end loop; end

   Shop;                                    продажа товара

end Factory; begin null;


   task  body Factory  is
     Item   :  Data;
   begin
     loop
       . . .                          - -   производство  товара
          Warenhouse (Item );   - - передача товара на склад end
     loop;

end   Factory_Shop_ Warenhouse;

                                    главная  процедура
```


Глава 10. И С К Л Ю Ч Е Н И Я

Надёжность - главная отличительная особенность языка, которая характеризовала Аду 83 и получила дальнейшее развитие в Аде 95. *Исключения* - одно из основных средств языка , обеспечивающих его высокую надежность.

Все ошибки в программах можно разделить на те, которые возникают на этапе компиляции программы, и те, которые возникают на этапе её выполнения.

На повышение эффективности выявление ошибок на этапе компиляции направлены многие механизмы языка, например, строгая типизация. Однако, отсутствие ошибок на этапе компиляции не гарантирует, что ошибки не возникнут, при выполнении программы.

В Аде имеются средства, позволяющие обрабатывать ошибки, возникающие во время выполнения программы, а также реагировать на специально выбранные программные события. Они основаны на механизме, который получил название *механизм исключений* (*исключительных ситуаций).*

Концепция исключений в языке базируется на следующих понятиях и действиях^

1 . При_выполнении программы в случае возникновения *исключительной ситуации* (ошибки или заранее определенные события) нормальное выполнение программы прекращается и *возбуждается исключение.*

2. Устанавливается связь между возникшим исключением и заранее определенным (или предопределенным) *обработчиком* данного исключения.

3. Управление передается *обработчику исключения.*

4. По завершению обработки исключения обработчиком управление не возвращается в точку возникновения исключения.

Обработчик исключения размещается в конце блока, подпрограммы, пакеты, задачи, защищенного модуля, оператора **accept.** Если в программном элементе, где произошла исключительная ситуация, отсутствует обработчик исключения, то возможны два варианта:

- работа этого программного элемента завершается аварийно;
- обработка исключения осуществляется в другой части программы.

Во втором случае выполняется поиск соответствующего обработчика исключения в других частях программы и передача ему управления. Такой процесс получил *название распространение исключения.*

Наличие в языке средств работы с исключениями повышает эффективность языка при разработке систем реального времени, в которых ошибки, возникающие при выполнении программы, не должны приводить к остановке в работе программы, управляющей, например, некоторым объектом.

10.1 Описание исключений

Для описания исключений используется следующая форма:

Список_Имён_Исключений : **EXCEPTION;**

Каждое исключение обозначается соответствующим идентификатором:

```
End_Of_File ALARM 1.ALARM2 exception;
Overflow High_Temperature  exception;
                           exception;
                           exception ;
```

Описания исключений размещаются в видимых разделах спецификаций, но не могут быть параметрами процедур.

Наряду с определяемыми пользователем исключениями, язык содержит некоторые предопределённые исключения, описанные в пакетах **STANDARD, IO_EXCEPTION** и др. Такие исключения не требуют описания и возбуждаются автоматически, но могут также явно возбуждаться пользователем.

Список предопределённых исключений из пакета **STANDARD:**

- **PROGRAMMERROR** (Ошибка программы). Исключение возбуждается при попытке вызова подпрограммы активизации задачи или конкретизации настраиваемого модуля, если тело соответствующего модуля ещё не обработано; если выполнение функции завершено через оператор **end** при выполнении оператора отбора с ожиданием, когда все альтернативы закрыты и отсутствует раздел **else.**
- **CONSTRAINT_ERROR** (Ошибка ограничения). Исключение возбуждается при нарушении ограничений диапазона; ограничения индекса или дискриминанта; при попытке использовать несуществующие компоненты записи и др.
- **STORAGE_ERROR** (Ошибка памяти). Исключение возбуждается в одной из следующих ситуаций: при выполнении генератора, когда исчерпана динамическая память под задачу; при вызове подпрограммы, если обнаружилась нехватка памяти.
- **TASKING_ERROR** (Ошибка задачи). Исключение возбуждается при взаимодействии задач.

Существовавшее в Аде 83 исключение **NUMERIC_ERROR** считается устаревшим, так как оно в Аде 95 вошло в исключение **CONSTRAINT_ERROR.** Исключение **NUMERIC_ERROR** (Ошибка числа) возбуждалось при выполнении предопределённой числовой операции, когда невозможно получить математически верный результат. Для корректности программ, написанных на Аде 83, при их обработке компиляторами для Ады 95, необходимо выполнить переименование исключения **NUMERUC_ERROR в CONSTRAINT_ERROR.**

10.2 Возбуждение исключений

В языке *возбуждение исключений* производится *неявно* для предопределённых исключений и *явно* пользователем с помощью оператора:

RAISE *[Имя_Исключени]*;

При выполнении оператора **Raise** работа программы прекращается и возбуждается исключение с указанным именем и управление предается соответствующему обработчику. Оператор исключения без имени допустим только внутри обработчика исключения. При этом он снова возбуждает исключение, которое вызвало переход на обработчик исключения с данным оператором возбуждения.

П Например:

```
Raise  ALARM2;
Raise  STORAGE_ERROR;        предопределённое исключение
Raise;
```

10.3 Обработчик исключений

Обработчик исключений определяет действия программы в ответ на возбуждение исключения и имеет следующий вид:

WHEN *Имя_Исключения => Последовательность_Операторов*[n],

WHEN *Имя_Исключения 1 \ Имя_Исключения* 2 I... I

WHEN OTHERS => *Последовательность_Операторов;* =>
Последовательъностъ_Операторов;

Ключевое слово **others** означает все исключения, которые могут возбуждаться в окружении обработчика, но обработчик исключения для них явно не указан.

Обработчики исключений размещаются в операторах блока, в телах подпрограмм, пакетов, оператора приема, задач или настраиваемых модулей в заключительной части перед словом **end.** Каждая конструкция, в которой разрешается размещать обработчики исключений, называется *окружением.*

Общий вид обработчика исключений должен быть следующим:

BEGIN
- - *Последовательность_Операторов*
EXCEPTION
- - *Обработчик_Исключений_1*

- - *Обработчик_Исключений_М* **END;**

При возбуждении исключения все остальные операторы в окружении не выполняются. Вместо этого начинается выполнение соответствующего обработчика исключений, если он имеется. Если такой обработчик не имеется в окружении, то выполнение блока, процедуры, пакета или задачи прекращается и происходит распространение исключения.

П Пример обработчика исключения:

```
when  CONSRAINT_ERROR when      =>   return    Integer'Last;
ALARM 1                         =>   Fire(x);
```

Выполнение обработчика исключения определяется не местом его расположения, а зависит от конкретного выполнения программы, то есть связь исключения и обработчика - *динамическая.* На эту связь также оказывает влияние момент появления исключения - во время предвыполнения описания или во время.выполнения оператора.

Если исключение появилось при предвыполнении описательной части подпрограммы, пакета, задачи, то предвыполение прекращается. Распространение в этом случае исключения зависит от того, где оно возбуждается :

• в описании подпрограммы, пакета, задачи, защищенного модуля - тогда исключение распространяется на ту часть программы, которая содержит это описание: если описание - библиотечный элемент, то выполнение главной процедуры завершается;
• в описании блока - тогда исключение распространяется на охваты^ вающий блок программу;
• в описании тела подпрограммы - тогда исключение распространяется на участок программы, содержащий вызов подпрограммы;
• в описании тела пакета - тогда исключение распространяется на ту часть программы, которая содержит пакет; если пакет - библиотечный, то выполнение главной процедуры завершается ;
• в описательной части задачи - то исключение распространяется на ту часть программы, которая активизировала данную задачу и задача завершается.

Если исключение возникло при выполнении оператора в блоке, подпрограмме, пакете, задаче, защищенном модуле, то его выполнение заменяется выполнением локального обработчика исключения, если он имеется. В противном случае происходит распространение исключения на другие части программы.

Если исключение появилось:

- в блоке, то исключение возбуждается в охватывающей блок программе ;
- в теле пакета, то обработка тела завершается, а для библиотечного модуля - прекращается; если пакет не является библиотечным модулем, то исключение возбуждается в той части программы, которая содержит тело пакета или его след;
- в задаче - выполнение задачи прекращается;
- в последовательности операторов обработчика исключение (но не в блоке), то выполнение обработчика завершается и распространяется новое исключение.

При выполнении задач причиной возбуждения исключения является часто появление исключений при взаимодействии задач во время рандеву. При этом возможны следующие ситуации :

• вызванная задача завершается или уже завершена перед вызовом входа: исключение **TASKING_ERROR** возбуждается в вызывающей задаче в точке вызова ;
• вызванная задача становится аварийной во время рандеву ; исключение **TASKING_ERROR** возбуждается в точке вызова ;

• Исключение возбуждается в операторе **accept**; в Аде 95 оператор **accept** может иметь обработчик исключений; если его нет, то исключение распространяется на вызвающую задачу и часть программы, содержащую оператор принятия **accept;**

10.4 Подавление проверок

Для отмены возможности возбуждения выбранных исключений можно воспользоваться прагмой **SUPPRES.** Такое указание компилятору называется подавление проверок и его можно сделать как для всей программы, так и в пределах ее части. Описание прагмы **SUPPRES:**

Для всей программы:

pragma SUPPRES (*Имя_Проверки) ;* Для

конкретного обьекта, типа, подпрограммы и др.:

pragma SUPPRES (*Имя_Проверки*, ON = > *Имя*);

Если используется вторая форма прагмы **Suppres** с именем, то прагма распространяется на все операции с обьектами, имеющими это имя :

- все обьекты базового типа для указанного в прагме типа (подтипа);
- вызов подпрограмм с указанным именем ;
- активизацию задач с указанным именем (задачного типа);
- конкретизацию указанного настраиваемого модуля.

Прагма **Suppres** должна размещаться непосредственно в разделе описаний или спецификации пакета. Ее действие распространяется от прагмы до конца зоны описания данного блока или модуля.

Для каждой исключительной ситуации в языке определен перечень проверок. Например, для исключительной ситуации **Constraint_Error** имеются проверки ссылок (**Access_Check**), дискриминантов (**Discriminant_Check**), индексов (**Index_Check**), длины (**Length_Check**), диапазона (**Range_Check).**

П Например :

```
pragma Suppres (Index_Check );
```


Здесь прагма отменяет (подавляет) проверку границ массивов и индексов компонент массива в исключении **Constraint_Error.**

```
pragma Suppres ( Discriminant_Check, ON => Elem );
```

подавляет проверку дискриминанта для всех обьектов типа Elem.

Если в некоторых случаях подавление проверки невозможно, то компилятор игнорирует соответствующую прагму **Suppres.**

Внимание ! Подавление проверок может сократить обьем программы после компилирования и повысить скорость ее выполнения. Однако при этом следует помнить, что этот выигрыш достигается за счет снижения ее надежности.

10.5 Пакет ADA.EXCEPTIONS

В языке имеется ряд предопределенных библиотечных пакетов, связанных с исключениями. Это , например, пакет **IO_EXCEPTIONS** связанный в исключениями при вводе-выводе.

Рассмотрим предопределенный пакет **ADA.EXCEPTIONS** . Он предоставляет пользователю дополнительные ресурсы в виде типов, констант и процедур для работы с исключениями.

Спецификация пакета **ADA.EXCEPTIONS :**

```
package ADA.EXCEPTIONS is

   type Exceptionjd is private; Nulljd : constant Exceptionjd ;
   function Exception_Name (Id : Exceptionjd) return String ;

   type Exception_Occrrence                    is limited private;
   type Exception_Occurrence_Access  is access
                                        all ExceptionjDccurrence;
   NullJDccurrence : constant Exception_Occurrence;

   procedure RaiseJException ( E :  in  Exceptionjd ;
                                        Message : in String );
   function ExceptionJVIessage (X : ExceptionjDccurrence )
                                                  return String;
   procedure Reraise _Occurrence( X : in ExceptionJDccurrence );
```

```
   function    Exceptionjdentity ( X : Exception_Occurence)
                                                  return   Exception;
   function    Exception_Name    ( X : ExceptionjDccurence )
                                                 return   String;
   function    Exception_Information( X : ExceptionjDccurence )
                                                 return String ; procedure
   Save_Occurence( Target: out Exception_Occuremce ;
                                    Source: in   ExceptionjDccurence);
   function   SavejDccurence ( Source :    ExceptionjDccurence )
                                return   ExceptionjDccurence_Access;
private
   . . .                  - -   не определены в языке

end   ADA . EXCEPTIONS ;
```

Пользователь может использовать ресурсы этого пакета при описании новых видов исключений и их обработчиков.

Ш СОВЕТЫ:

* Экспортируйте (делайте видимыми для пользователя) имена всех исключений, которые могут быть возбуждены.
* В пакете описывайте каждое исключение, которое может быть возбуждено каждой подпрограммой или входом задачи,описанных в пакете.
* Если возможно, избегайте изменения состояния информации в модуле перед возбуждением исключения.
* Не возбуждайте предопределенные исключения или предопределенные реализацией исключения.
* Используйте исключения при определении абстракций.
* Не используйте оператор **goto** в исключениях и обработчиках.
* Используйте в обработчиках информацию, полученную с помощью процедур пакета **Ada . Exceptions.**

4 Обрабатывайте все исключения (пользовательские и предопределенные)

* Используйте оператор блока для локализации объединения частей программы с обработчиками.

ИЗМЕНЕНИЯ:

О Исключение **NumericJError** переименовано в **ConstraintJError.** О Обработчик исключения можно размещать непосредственно в операторе **Accept.** О Добавлен пакет **Ada . Exception** для работы с исключениями.

10.6 Примеры

П Пример 1.

Процедура для вычисления факториала.

```
procedure Factorial ( N : natural ) return integer   is
begin
    if  N = 0   then
       return 1 ;
    else
        return N * Factorial ( N -1 ); - - возбуждение исключения
    end   if;
excepion                                    - - обработчик исключения
   when  ConsraintJError  = >      return   Integer'Last;
end   Factorial;
```

При выполнении функции вычисления факториала возможны исключительные ситуации, связанные с тем , что :

- аргмент функции может быть задан отрицательным числом ;
- аргумент функции имеет большое значение и при вычислении функции произойдет переполнение.

В первом случае предопределенное исключение Constaint_Error по несоответствию формального и фактического параметров возбуждается в программе, где вызывается функция Factorial, и соответствующий обработчик исключения должен быть предусмотрен в ней.

Во втором случае исключение ConstrainJError по переполнению возбуждается уже при выполнении функции и обработчик исключения размещается в самой функции. Действия обработчика приведут к тому, что выполнение подпрограммы не завершится аварийно и в качестве результата при переполнении функция Factorial возвращает максимально допустимое в реализации значение целого типа.

D Пример 2.

Настраиваемый пакет для работы с буфером.

```
generic
    N :    positive ;
    type   Elem   is   private;
package   BUFFEFM/VORK;
    procedure   In_Buffer (X :    in    Elem);
    procedure   Out_Elem (Y       out   Elem );
    procedure   Cfean_Buffer;

    Over : exception;   - - исключения
    Full  : exception ;

end   BUFFER_WORK;

package body   BUFFER_WORK  is

    Index :   integer : = 0;
    Buffer:   array  (1 .. N ) of Elem ;

    procedure   Clean_Buffer  is
    begin
        for i   in    1 . . N    loop Buffer
            (i): = 0 . 0 ;
        end   loop; end
    Clean Buffer;

    procedure   In_Buffer (X : in   Elem ) is begin
       if   Index  > N then
            raise   Over;            - - исключение по переполнению
       else
       Buffer (i): = X ; Index : =
       Index + 1 ; end   if;
```



```
    end    In Buffer ;


    procedure Out_Buffer ( Y :   out    Elem )   is begin
        if   Index = 1    then
            raise   Full;            - - возбуждение исключения при
                                     - -  отсутствии данных в буфере
        else
    Index   : =   Index -1 ; Y       : =   Buffer
    (Index) ; end  if; end Out_Buffer;

end   BUFFER_WORK;
```

В данном примере настраиваемый пакет Buffer_Work предос тавляет пользователю пакета ресурсы для работы с некоторым буфе ром размерности N и типом элементов Elem. Ресурсами являются три процедуры : очистки буфера, записи в буфер по текущему индексу (указатель Index) и считывания из буфера верхнего элемена (указатель Index - 1). В пакете предусмотрены два исключения :

- Full - для возбуждения исключения при попытке считывании информации из пустого буфера;
- Over - - для возбуждения исключения при попытке записи в заполненный буфер.

Так как обработчики исключений Full и Over отсутствуют в пакете, то при возбуждении этих исключений их обработка должна выполняться в тех частях пользовательской программы, где осуществляется вызов процедур In_Buffer и Out_Buffer.

Глава!!. НАСТРАИВАЕМЫЕ МОДУЛИ

Настраиваемые модули в языке реализуют абстракцию данных через *параметризацию,* предоставляя пользователю возможность практически неограниченного формирования различных вычислений.

Параметризация модулей в языке обеспечивается путем расширения спецификации, куда добавляется *настраиваемая часть,* в которой описываются *формальные параметры настройки.* Далее размещается спецификация подпрограммы или пакета. То есть, обычный модуль (пакет или подпрограмму) можно превратить в настраиваемый, если дополнить его спецификацию описанием настройки.

Настраиваемый модуль реализует все преимущества абстракций через параметризацию :

1. Сокращение затрат на создание программы, так как один настраиваемый модуль может заменить несколько модулей, отличающихся используемыми в них типами, переменными, подпрограммами, пакетами.
2. Программы становятся короче, занимают меньший объём памяти, и в итоге их проще писать и отлаживать.
3. Типы становятся *переносимыми,* то есть модификация используемых в программе типов не требует изменений в настраиваемых модулях.

Строгая типизация языка обеспечивает его надёжность, но не позволяет создавать универсальные программы, работающие с множеством различных типов. Предусмотреные в языке *настраиваемые средства* (generic facilities), основанные на механизме настраиваемых модулей, позволяют разрабатывать универсальные программные модули (пакеты и подпрограммы) в виде шаблонов (templates).

Настраиваемые модули разрабатываются с помощью специальных настраиваемых средств и не используются непосредственно для выполнения. Они служат основой для создания других программных модулей с заданными свойствами.

11.1 Спецификация настройки и тела

Спецификация настройки настраиваемого модуля имеет следующую форму:

GENERIC

Формальные_Параметры_Настройки

Спецификация_Пакета_Или_Подпрограммы.

Параметры настройки во многом аналогичны формальным параметрам подпрограмм. В качестве *Формального_Парамет-ра_Настройки* можно использовать:

- формальный обьект;
- формальный тип ;
- формальный модуль.

Формальный модуль настройки - это *формальная подпрограмма* или *формальный пакет*

Часть спецификации настройки, влючающая ключевое слово **generic** и список формальных параметров настройки называется *формальной частью* настройки.

Настраиваемые модули могут не иметь параметров настройки. В этом случае они используются для создания идентичных модулей. П

Примеры формальной части настройки :

```
generic                          -- без параметров

generic
   N : integer;                  -- формальный обьект

generic                          -- формальный обьект, со значением
   Size : positive := 66 ;       -- по умолчанию

generic
   type Post is (<>);            -- формальные типы type
   Port is private;

generic                          -- формальная функция
   with function Read ( Elem : float) return float;

generic                          -- формальная процедура
   with procedure Cooler ( Power: in Data ; Volt: out fixed )


generic                  -- формальный пакет
   with package Nord is new West(.<>);
```

Формальные параметры видны вне настраиваемого модуля и используются при его настройке. Кроме того, они используются при описании обычной спецификации настраиваемого модуля и в его теле.

П Пример спецификации настраиваемой процедуры Sort :

```
generic
    N :      integer  ;  - -  формальная  часть  настраиваемой
    type     Data     is     private;   -   -   процедуры   Sort
    with function Max (A, B: Data) return Data;
procedure  Sort (X  : in  out  Data);                  - - спецификация
```

Настраиваемая процедура Sort имеет три параметра настройки: объект N, тип Oata и функцию Max.

П Пример спецификации настраиваемого пакета REZ:

```
generic
    type   Elem  is  range  <>;  - - формальная часть
    with   procedure  Sum (A, B : in   Elem ;   C :   out  Elem );
package  REZ  is                          - - спецификация, пакета  REZ
    procedure  Beta (X :  in  Elem);
    REZ_Error:   exception; end
REZ;
```

Тело настраиваемого модуля является телом либо пакета, либо подпрограммы и реализует ресурсы, описанные в спецификации. При этом в теле используются формальные параметры настройки данного модуля. Тело настраиваемого модуля является шаблоном для настройки. Синтаксически настраиваемое тело идентично телу ненастраиваемого модуля.

П Пример тела настраиваемой процедуры Sort , спецификация настройки которой описана выше:

```
procedure Sort (X :   in   out Data) is
    Temp:    Data;
begin
    for  i   in   1   .. N    loop for j
       in   1 .. N    loop
```



```
    Temp(i):=    Max ( X( i), X ( j )); end   loop;
end loop; X: =   Temp ; end    Sort;
```

Здесь при реализации тела процедуры используются формальные параметры настройки : тип Data, обьект N, функция Max .

11.2 Конкретизация настраиваемых модулей

Конкретизация или *настройка (* instantiation) настраиваемых модулей выполняется во время компиляции. При этом создаётся новый модуль с заданными свойствами, которые определяют подставленные фактические параметры. Созданный таким образом модуль аналогичен обычному модулю языка (пакету или подпрограмме) и может использоваться в программе. Количество создаваемых при конкретизации модулей неограничено. Результат конкретизации настраиваемой процедуры есть процедура, настраиваемого пакета - пакет. Конкретизация - копирование текста шаблона. При этом каждый формальный параметр получает фактическое значение.

Общий вид конкретизации:

| PACKAGE | FUNCTION | PROCEDURE] ***Имя*** **IS**

NEW *Имя_Настраиваемого_Модуля*

[Список _Фактических_Параметров_Настройки];

Здесь *Имя* - название нового создаваемого при конкретизации модуля.

Конкретизация должна выполняться в описательной части программы, подпрограммы, блока и приводит к новому множеству описаний.

Фактические параметры при конкретизации задаются как в позиционной форме, так и в именованной . При этом они могут быть:

- выражением;
- именем переменной;
- именем подпрограммы;
- именем входа;
- именем типа;
- именем подтипа;

- именем настройки пакета. П Пример конкретизации:

```
package HORSE is new REZ (integer, Vecsum);

procedure Sort_Mod is new
                Sort (N => 20 ; Data => Float; Max => Mf);

function Alfa is new Machine; 11.3
```

Формальные параметры настройки

11.3.1 Формальные объекты

Формальные объекты настройки - обычные параметры, аналогичные параметрам подпрограмм. Могут быть константами, выражениями, переменными. Используются для передачи значений или переменных в настраиваемые модули.

Для формальных обьектов настройки их *вид* ограничен значениями in или **in out,** при умолчании подразумевается in. Вид in определяет формальный параметр как константу, **in** out - переименовывает соответствующий настраиваемый фактический параметр.

П Например:

```
generic
   N :   in  out  integer_40;

generic
   X float;
     float  : =  0.245 ; - - значения по умолчанию
   Y Vector;
```

Ш СОВЕТЫ:

- Используйте подтипы при обьявлении формальных обьектов вида **in out.**

11.3.2 Формальные типы

Формальные типы используются для передачи в настраиваемый модуль подтипов некоторых классов типов.

Формальный тип может быть одним из следующих:

- индексируемым типом;
- ссылочным типом;
- определением личного (приватного типа);
- определением производного типа ;
- склалярным типом

Допускается использование дискриминантов в формальных типах , где это разрешено.

Формальные индексируемые типы основываются на классе типов массива.

D Например:

```
generic
   type  Elem     is  range  < > ;
   type  Vector   is  array (integer range <> ) of  Elem ;
   type  Matrix   is  array (integer range < > )  of  Vector;

generic
   type   Sec    is  delta < >;
   type   Index  is  (<>);
   type   Tab!   is  array  ( Index) of  Sec; - - с дискриминантом

generic
   type  Page  is  private;
   Na :  integer  : я  50 ;          - - значение по умолчанию
   type  Lines  is  array  (1.. Na )  of  Page ;
```

Формальные ссылочные типы основываются на классе всех ссылочных типов.

П Например:

```
generic
   type  Nord;
   type  Ac_Nord  is  access  Nord;
```

```
generic type    is          private;
   Send type    is          access  Send;
   Post type    is          access  all  Send;
   Office
```

Формальные скалярные типы - это класс, определенный для дискретных, целых, плавающих, фиксированных типов:

- *формальный дискретный тип* (обозначается (o));
- *формальны целый тип* (обозначается **range** (<>));
- *формальный тип с плавающей запятой* (обозначается **digits o) ;**
- *формальный тип с фиксированной запятой* (обозначается **delta** o);
- *формальный десятичный фиксированный* (обозначается **delta** < > **digits** < >);

Формальные личные и производные типы основываются на классах личных и производных типов.

Класс типов для формального личного типа - лимитированные и тэговые. Класс формальных производных типов - производный класс.

Форма записи для *формальных личных типов :*

TYPE *Имя_Типа* *IS* *[* [ABSTRACT]

TAGGED] [LIMITED] PRIVATE ;

Форма записи для *формальных производных типов :*

TYPE *Имя_Типа* IS [ABSTRACT]
NEW *Имя_Подтипа* *[* WITH PRIVATE];

Если формальный тип имеет дискриминант, то он не должен включать параметры по умолчанию.

П Примеры классов определения для формальных личных типов :

```
type    Square     is    private;
type    Root       is    limited    private;
type    Street     is    tagged     private;
type    Node ( N : natural)  is  limited  private ;
                                  - - используется дискриминант
```



```
type    Border  is  tagged    limited    private ;
```

П Примеры классов для определения формальных производных типов

```
type  Ferro  is  new  Square;

type  Corsa  is  abstract  new  Street
                 with  null record ;  - - для тэгового типа
```

Ш СОВЕТЫ:

* Используйте формальные приватные типы , если требуется обычное присваивание обьектов в теле настраиваемого модуля, и формальные лимитированные приватные типы - когда не требуется присваивание.
» Используйте формальные тэговые типа, производные от типа **Ada.Finalization.Controlled ,** если необходимо специальное присваивание в теле настраиваемого модуля.
* Используйте формальные ограниченные абстрактные типы при расширении формальных личных тэговых типов.

11.3.3 Формальные подпрограммы

Формальные подпрограммы как параметр настройки указываются с помощью спецификации через контекст **with.**

Возможны три формы описания подпрограммы, соответствующей формальной подпрограмме настройки:

```
WITH  Спецификация_Подпрограммы;
WITH  Спецификация_Подпрограммы  IS  Имя;
WITH  Спецификация_Подпрограммы  IS  <>;
```

Последние два описания позволяют опустить при конкретизации соответствующий фактический параметр, то есть использовать его по умолчанию . *Имя* означает имя подпрограммы, которая видна (прямо видна) в месте описания спецификации настройки ; *бокс* < > используется при умолчании и является эквивалентным имени подпрограммы, видимой (прямо видимой) в месте конкретизации.

П Например :

```
with procedure Doom (Z in out float); out fixed) is
with procedure Palm (X, D Coconaut; in string ; C : out boolean )
with procedure Parcel( A, B is <> ;
```

При конкретизации формальные подпрограммы Palm и Parcel задаются по умолчанию. Для Palm по умолчанию используется подпрограмма Coconaut, а для Parcel - ищется подпрограмма с тем же именем в том же тексте, где конкретизируется формальная подпрограмма.

11.3.4 Формальные пакеты

Формальные пакеты как параметр настройки используются при передаче пакетов в настраиваемые модули. Формальный пакет должен быть конкретизацией некоторого настраиваемого пакета. При конкретизации настраиваемого модуля должна быть предварительно выполне-нена конкретизация фактического пакета, являющегося фактическим параметром.

Описание фактического параметра обьявляет, что фактический пакет является конкретизацией данного настраиваемого пакета.При конкретизации фактический пакет должен быть, в свою очередь, конкретизацией этого настраиваемого пакета.

Общая форма *формального пакета:*

WITH PACKAGE *Имя_Формального_Пакета* **IS**

NEW *Имя_Настраиваемого_Пакета* (<>) j

[*Фактические_Параметры_Настройки];*

Имя_Формалъного_Пакета - настраиваемый пакет, являющийся шаблоном для формального пакета. *Имя_Настраиваемого _Пакета* - настраиваемый пакет, являющийся настраиваемым шаблоном для формального пакета.

Если вместо фактических параметров формального пакета используется конструкция (<>) , то формальные параметры могут быть любой конкретизацией шаблона,

П Например :

```
with Way, Wax;          - - библиотечные настраиваемые пакеты
```



```
generic                          - - настраиваемый пакет Soul
   with package Nord    is new Way ( < > ) ; - - формальные
                                                  - - пакеты
   with package Nepal   is new Wax (integer, Size );

package SOUL   is use
   Nord, Nepal ;
    . . .                  - - спецификация пакета Soul
end SOUL;
```

Конкретизация пакета Soul требует сначала настройки формальных пакетов:

```
package N_Way is new Way (Float); package
N_Wax is new Wax (integer, 60);
```

а затем настройки самого пакета **package** N_Soul **is new**

```
SOUL ( N_Way, N_Wax );
```

Различают два назначения формальных пакетов :

- для определения дополнительных операций в новой абстракции в терминах некоторой существующей абстракции, уже определенной через настраиваемый пакет;
- для реализации некоторой абстракции несколькими различными способами.

Формальные пакеты значительно расширяют возможности настраиваемых модулей. В первом случае формальные пакеты позволяют импортировать все типы и операции, полученные при конкретизации, то есть происходит расширение существующей абстракции без перечисления всех операций существующей абстракции в новой. Во втором случае реализация новой абстракции может быть выполнена несколькими способами, например путем включения в список параметров настройки настраиваемой функции формального пакета. Для этого абстракция реализуется через параметризацию и в качестве параметров используется формальная подпрограмма. П Например :

Имеется настраиваемый пакет MM, формальные параметры настройки которого - типы и подпрограммы. Можно определить новую абстракцию в виде, например, настраиваемой процедуры PP, где ее параметр настройки - формальный пакет MM. Тогда при реализации тела PP можно использовать необходимые формальные параметры пакета MM и при конкретизации как PP, так и MM появляется возможность различных реализаций абстракции PP:

```
generic                                          - - Пакет MM
    type    Data   is   private;
    Mum:   Data ;
    with   procedure   Elf (X:   in    Data;   Y:   out   Data); package
MM is

end   MM;

generic                          - -   настраиваемая процедура PP
   with   package West_MM    is    new    MM ( <> );
   use    West_MM;
procedure   PP (A :   in    Data ; B :   out   Integer);

  - -              тело процедуры   PP
procedure PP (A :   in    Data ; B :   out  Integer)   is
   Temp :    Data : = Num + 1 ;
begin
    Temp : = A - Temp ; Elf
    (Temp, B) ;

end PP;

- - конкретизация пакета   MM
package   Old_MM    is   new    MM ( Data = > integer,
                                  Num = > 100,    Elf = > Work _Data);

- - конкретизация процедуры   PP procedure
Green   is   new    PP ( Old_MM );
```


11.4 Правила сопоставления параметров настройки

Для формальных и фактических параметров настройки пакетов при конкретизации действуют следующие правила:

1. *Формальные объекты.* Типы фактических параметров настройки должны соответствовать типу формального параметра настройки и всем наложенным ограничениям.
2. *Формальные личные (приватные) типы.* Формальный личный тип сопоставляется с любым фактическим типом. При этом, если формальный личный тип:

- является лимитируемым личным **(limited private),** то ему может быть сопоставлен любой тип, включая заданный тип;
- не является лимитируемым личным типом, то ему может быть сопос тавлен любой фактический тип, над которым определены опера ции равенства, неравенства и присваивания;
- имеет дискриминант, то он должен согласовываться с дискрими нантом фактического типа.

3. *Формальные скалярные типы.* Формальному скалярному типу, специфицированному как:

- (o),сопоставляется любой дискретный тип;
- **range o,** сопоставляется любой целый тип;
- **digits o,** сопоставляется любой тип с плавающей запятой;
- **delta o,** сопоставляется любой тип с фиксированной точкой.

4. Формальные индексируемые типы. Тип фактического параметра должен иметь столько индексов, сколько и настраиваемый формальный тип. Если либо формальный индексируемый тип, либо соответствующий фактический параметр является неограниченным, то и другой тип должен быть неограниченным.

5 .Формальные ссылочные типы. Если тип объекта в формальном ссылочном типе является настраиваемым, то он замещается соответствующим фактическим параметром. После выполнения подстановки формальный ссылочный тип и фактический ссылочный тип должны иметь одинаковые типы объектов.

6 .*Формальные подпрограммы.* Если существует несколько типов настройки в формальной подпрограмме настройки, то они прежде всего замещаются соответствующими фактическими параметрами. После замещения подпрограммы должны быть такими, чтобы их соответствующие параметры имели одинаковые тип, вид и ограничение. Для функции дополнительно типы результата и ограничения на них должны быть одинаковыми.

7. *Формальные пакеты* . Фактический пакет должен быть конкретизацией формального пакета. При конкретизации настраиваемого модуля с формальными пакетами они должны быть предварительно конкретизированы.

Операция передаётся в качестве параметра настройки для формального параметра - функции. Аналогично можно передать функцию как фактический параметр настройки для формального параметра - операции.

Ш СОВЕТЫ:

* Используйте настраиваемые модули с формальными параметрами для достижения максимальной адаптируемости пакета
* Используйте настраиваемые модули для инкапсуляции алгоритмов, независимых от типа данных.
* Используйте настраиваемые модули во избежания дублирования кода.

ИЗМЕНЕНИЯ:

О Формальным параметром может быть формальный пакет.

*О Фактические параметры могут быть конкритизированы в самом настраиваемом пакете.

О Разрешены новые формы настраиваемых параметров. CJ Изменилась формальная нотация для формальных параметров пакета.

11.5 Примеры

D Пример 1:

Настраиваемый пакет, ресурсами которого являются константы и типы. Такой пакет не требует тела.

```
generic
      N :  integer;
      M:   integer;
      K:   positive ;
package  RUIN   is
   Cord :   constant: = N * M ;
type  Baltic  is ( 1 .. Cord )  of Character;
type  Belle  is  record
   X  array (1 .. N ) of  Data ;
   Y  array (1 .. N , 1 .. M ) of float;
   Z  string (1 .. K );
end;
end   RUIN;
```

Конкретизация пакета RUIN :

```
package WAR   is  new  RUIN  (8, 10,14);
package WORD  is   new  RUIN  ( K => Size, M => 10, N = > 99 );
```

П Пример 2:

Настраиваемый пакет VEC для работы с векторами имеет два формальных параметра настройки - размерность вектора и тип элементов вектора.

```
Generic
    N:   positive;
   type   Elem   is  digits  <>;
package   VEC   is

   type  Vector  is  array  (1. . N)  of  Elem;
   procedure  Sum_Vec  (VA, VB:  in  Vector;

   C :  out  Vector ); procedure
   Mulm_Vec (VA, VB :  in  Vector;        C :   out   Elem ); -
end  VEC;

package   body   VEC  is                  - тело пакета VEC

   procedure    Sum_Vec (VA, VB        in   Vector; VC:  out
                                        Vector)   is

   begin
```

```
        for  i   in   1  ..  N   loop
       VC( i): = VA( i) + VB( i); end
    loop; end   Sum_Vec;

    procedure   Mulm_Vec (VA,  VB:    in    Vector;
                                          C:   out   Elem)   is Temp :
    Elem : =  0.0 ; begin
       for  j   in   1  ..  N  loop
      Temp : =  Temp  +  VA( j) * VB( j); end
   loop; C: =  Temp; end    Mulm_Vec;


end  VEC;
```

Настраиваемый пакет VEC при конкретизации может настраиваться на любую размерность векторов, задаваемую целым положительным числом и на любой тип (подтип), описывающий тип с плавающей точкой.

Пример использования и настройки пакета VEC в процедуре:

```
with  VEC; procedure  Vec_Work   is

type   Floats   is  digits  8; package   DATA  is  new
VEC (80,  Floats); use    DATA; A ,B, C :   Vector; Rez    :
Floats; begin

    Sum_Vec( A, B, C);
   . . .

   Mult_Vec (A, B, Rez);

end  Vec_Work;
```


П Пример 3. [12]

```
generic                 - - формальная часть настраиваемого пакета
    Size : Positive ;
    type Item is private ;       - - любой  тип при конкретизации
package STACK is

    procedure Push ( E : in    Item );   - - спецификация пакета
    procedure Pop   ( E : out   Item ); Overflow,   Underflow:
    exception ;

end  STACK;

package body STACK is                - - тело  пакета

   type   Table   is array ( Positive range<>) of  Item ; Space :
   Table (1 .. Size); Index : Natural: = 0 ;

   procedure Push ( E : in   Item )   is   - - тело   процедуры begin
       if   Index > = Size  then raise
          Overflow;
       end if;

      Index : = Index + 1 ;
      Space (Index): = E ;

  end   Push;

  procedure Pop ( E : out   Item ) is   - - тело   процедуры begin
      if Index = 0  then raise
      Underflow; end   if;
   E : = Space( Index); Index :
   = Index -1 ; end   Pop;
```

end STACK; Конкретизации

пакета Stack :

```
package Stack200 is new STACK (Size = > 200,
                                Item = > integer);
package Vector_Stack is new STACK ( 55 , Vector) ;
```

Использование ресурсов из настроенных пакетов Vector_Stack и Stack200 :

```
Vector_Stack . Pop( X );
Stack200 . Push (16 );
```

Другая версия пакета для работы со стеком использует приватный тип Stack с дискриминантом :

```
generic
   type Item is private;
package ON_STACK is

   type Stack (Size : Positive ) is limited private ;
                                  - - используется дискриминант
procedure Push ( S : in out Stack ; E : in Item ); procedure Pop (S :
in out Stack ; E : out Item ); Overflow, Underflow: exception ; private
   type Table is array (Positive range <>) of Item; type
   Stack (Size : Positive ) is record
      Space : Table (1 .. Size ); Index :
      Natural: = 0 ; end record;

end On_Stack;
```

Настройка пакета

```
package Best_Stack is new On_Stacke (float);
```


Pool : Stack (50); Использование ресурсов :

Best_Stack. Push (Pool, 3.1426);

D Пример 4.

```
package LA is              - - ненастраиваемый пакет

   type Gent is private
   procedure FA ( . . . ) I
   procedure FB (... ) ;

end LA;

with LA; use LA;     - - настраиваемый пакет TOX
generic
   type Tender is new Gent;
package TOX is

   procedure ËA (X : in out Tender);
   function ES (Z, Y : Tender ) return Tender;

end TOX;
```

Настраиваемый пакет TOX с формальным производным типом Tender, который может быть только типа Gent или производным о него. Для реализации в теле пакета TOX подпрограмм EA и ES можно использовать все операции, допустимые для типа Gent , определенные в пакете LA (процедуры FA и FB).

Глава 12. З А Щ И Щ Е Н Н Ы Е МОДУЛ И

Ада 95 характеризуется появлением в ней еще одного вида программных модулей - *защищенных модулей (* **protected units**). Их назначение - расширение возможностей языка при программировании параллельных процессов, в частности - для решения проблемы доступа к общим ресурсам. Кроме того, защищенные модули обеспечивают поддержку различных парадигм *систем реального времени.*

В Аде 83 механизм рандеву был единственным средством, использовавшимся для организации как межзадачного взаимодействия, так и для *синхронизированого доступа* к *общим данным* (shared data). Опыт использования языка показал , что это не всегда удобно, и потребовал пересмотра методов синхронизации и введения в язык дополнительных средств, обеспечивающих более эффективные средства работы с общими данными.

В Аде 95 эта проблема решена с помощью добавления в язык ориентированного на данные механизма синхронизации, основанного на концепции *защищенных объектов.* Операции над защищенными обьектами позволяют нескольким задачам синхронизировать свои действия при работе с общими данными.

Концепция защищенных модулей основана на механизмах *критических условных областей (* critical region condition) и *условных мониторов (* condition monitors) [14,15]. Идея монитора, предложенная Б.Хансеном и развитая С.Хоаром, основывается на обьединении переменных, описывающих общий ресурс, и действий , определяющих способы доступа к ресурсу.

Монитор - программный модуль, содержащий переменные и процедуры для работы с ними, причем доступ к переменным возможен только через процедуры монитора. Такой концепции в определенной мере соответствует в Аде реализация пакетов. Однако назначение монитора не только в инкапсуляции. Монитор - средство распределения ресурсов и взаимодействия процессов. Это назначение монитора реализуется с помощью свойств, которыми наделены процедуры монитора. Характерная особенность процедур монитора - *взаимное исключение* ими друг друга. В любой момент времени может выполняться только одна процедура монитора. Если некоторый процесс вызвал и выполняет процедуру монитора, то ни один другой процесс не может выполнять процедуры монитора. При попытке вызова другим процессором этой же или другой процедуры монитора

он блокируется и помещается в очередь до тех пор, пока активный процесс не закончит выполнение мониторной процедуры. То есть в мониторе не может "находиться" больше одного процесса. Такое свойство процедур монитора обеспечивает взаимное исключение и синхронизацию процессов, работающих с монитором.

В условных мониторах с помощью условных переменных (типа Condition) и операций над ними Signal и Wait обеспечивается более гибкая возможность работы с монитором. Здесь в зависимости от значения условных переменных можно блокировать или разблокировать процессы, находящиеся в мониторе, используя операции Signal и Wait и разрешать выполнение других процедур монитора.

Таким образом, монитор подобен пакету. Он не является активным подобно задаче, то есть не требует процессорных ресурсов, а сам предоставляет свои ресурсы в виде общих переменных и процедур и контролирует их использование.

Нетрудно заметить, что "характерное" свойство процедур монитора присуще входам задач в Аде, которые также в случае необходимости блокируют вызывающие процессы и формируют очереди к входам . Следовательно, возможный подход к реализации идеи монитора С.Хоара в Аде - обьединение свойств пакетных и задачных модулей, результатом которого и стало появление защищенных модулей.

Общие данные и операции над ними (*защищенные операции)* объединяются в защищенном модуле, аналогично тому, как это делается в пакетах. Доступ к общим ресурсам возможен только через защищенные операции, которые обладают свойствами, позволяющими решить *задачу взаимного исключения* при работе с общими ресурсами.

Понятие защищенного модуля , базируется на таких понятиях как *защищенный тип (* protecred type) , *защищенный объект (* protected object), *защищенная операция* (protected operation), *защищенная подпрограма* (protected subprogram) , *защищенный вход* (protected entry), *барьер* (barrier).

12.1 Спецификация и тело защищенного модуля

Являясь еще одним видом программного модуля в Аде, защищенный модуль имеет унифицированную структуру, принятую в языке . то есть состоит из спецификации и тела.

Спецификация защищенного модуля:

PROTECTED [TYPE] *Имя_Защищенного_Модуля*
[Дискриминант] IS

- - *Описание_Защищенных_Операций [*
PRIVATE 1
- - *Описание _Защищенных_Элементов*

END *Имя_Защищенного_Модуля*;

Здесь *Описание Защищенных _Операции* - описание защищенных подпрограмм, защищенных входов, контекстов представления ; *Описание_Защищенных_Элементов* - описание защищенных операций и описание компонент.

Защищенный тип является лимитированным типом. Он может иметь дискриминант, аналогично задачному типу, что позволяет минимизировать число операций при инициализации защищенных обьектов.

Защищенные операции - это функции. процедуры и входы, описанные в спецификации защищенного модуля. Приватная часть спецификации ограничивает видимость *защищенных элементов :* операций и обьектов, описанных в ней. Общие данные, доступ к которым координируется защищенным модулем, описываются в приватной части его спецификации.

Защищенные функции обеспечивают доступ только по чтению компонентов защищенного модуля.

Защищенные процедуры обеспечивают эксклюзивное чтение и запись компонент защищенного модуля.

Защищенные входы обеспечивают те же фунции, что и входы задач, дополнительно реализуя с помощью барьеров эксклюзивный доступ к телу защищенного входа.

D Примеры спецификации защищенного модуля :

```
protected Circle is

   procedure Control;
                                  - - защищенные подпрограммы
   function Best (X : Float); end
Circle;

protected type Monitor is
   entry Wait;                    - - защищенные входы
   entry Signal;

private
   Condition : Boolean := False ; - - защищенная компонента

end Monitor;

protected Systems (Id : in Positive ) is

   entry Read ( X : out integer);          - - защищенный вход
   procedure Write (Y : in integer);       - - защищенная процедура

private
   Data : array (1.. Id ) of integer;      - - защищенная компонента end
Systems;
```


Защищенные подпрограммы (процедуры и функции) не являются блокирующими защищенными операциями. С блокированием вызывающих задач связаны действия только над защищенными входами.

Тело защищенного модуля реализует защищенные операции, объявленные в его спецификации , используя для этого локальные ресурсы, объявленные непосредственно в теле.

PROTECTED BODY *Имя_Защищенного_Модуля* IS
Локальные_Описания
BEGIN

- - *Реализация защищенных операций и защищенных элементов* **END**

Имя_Защищенного_Модуля; В теле защищенного модуля видны описания

из его приватной части.

Защищенные процедуры и функции реализуются в теле защищенного модуля точно так же, как это делается в теле пакета.

В отличие от задачного типа, реализация защищенного входа в теле защищенного модуля не связана с оператором принятия **accept,** а выполняется с помощью *тела входа*, в котором обязательно используется *барьер*. Тело защищенного входа обеспечивает эксклюзивные операции чтения и записи.

Описание тела защищенного входа :

ENTRY *Имя_Защищенного_Входа* **WHEN** *Условие* **IS**

BEGIN

- - Последовательность _Операторов

END *Имя_Защищенного_Входа* ;

Здесь конструкция **When** *Условие* является барьером, а само *Условие* - логическим выражением, которое определяет : *открыт* или *закрыт* вход. Проверка условия во входе выполняется при вызове защищенного входа. Если значение *Условия* - **True,** то вход *открыт* и выполняется тело защищенного входа, иначе - вход *закрыт* и выполнение тела блокируется до тех пор, пока *Условие* в барьере не будет изменено другой задачей при помощи вызова защищенной процедуры или другого защищенного входа.

D ЛТример тела защищенного модуля:

```
protected body Monitor is

   procedure Block (X : in Process ) is - - локальная процедура
      * * .
   end Block;

   procedure Deblock (X : in Process ) is - - локальная процедура

   end Deblock ;

   entry Wait when Condition is - - тело
   begin
                защищенного входа с барьером
   Conditio Signal with not Condition is - - тело
   ( Proces _ ,. - - защищенного входа с барьером
   Wait;
   entry
   begin
Condition : = True ; Deblock (
Process_Name ); end Signal; end
Monitor;
```


12.2 Применение защищенных модулей.

Применение защищенных модулей в Аде 95 ориентировано на модель межзадачного взаимодействия, где имеет место *непрямое* взаимодействие задач. Это, например, работа с общими переменными в вычислительных системах с общей памятью или взаимодействие задач через буфер.

В Аде 83 непрямое взаимодействие задач А и В реализуется с помощью вспомогательной задачи С (Рис. 12.1). Задача С имеет входы Read и Write для записи и чтения данных из буфера -локальной переменной. Задача А через вызов входа Write записывает информацию в буфер, а задача В , вызывая вход Read считывает данные из буфера. Механизм рандеву обеспечивает взаимное исключение при работе с буфером. Существенный недостаток данной схемы межзадачного взаимодействия - необходимость наличия активной задачи С, для которой требуются процессорные ресурсы. Кроме того, средства механизма рандеву не всегда просто позволяют запрограммировать все "тонкости" взаимодействия задач А и В с задачей С (например, анализ буфера, действия задач при записи в заполненный буфер или при считывании из пустого буфера и т.д.).

В Аде 95 для реализации подобного взаимодействия задач предполагается использование механизма защищенных модулей. Вместо задачи С применяется защищенный модуль СР (Рис 12.2) . Такая замена имеет несколько положительных сторон : не являясь в отличие от задачи активным, защищенный модуль не требует постоянных процессорных ресурсов, а также кроме работы со входами • (где добавлены барьеры) позволяет работать с защищенными процедурами и функциями .

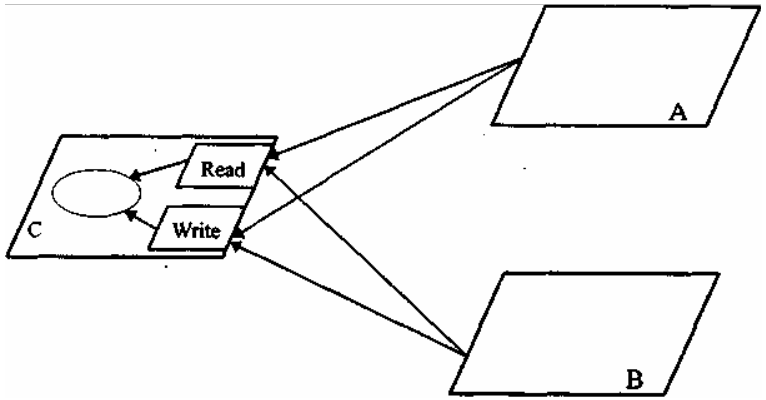
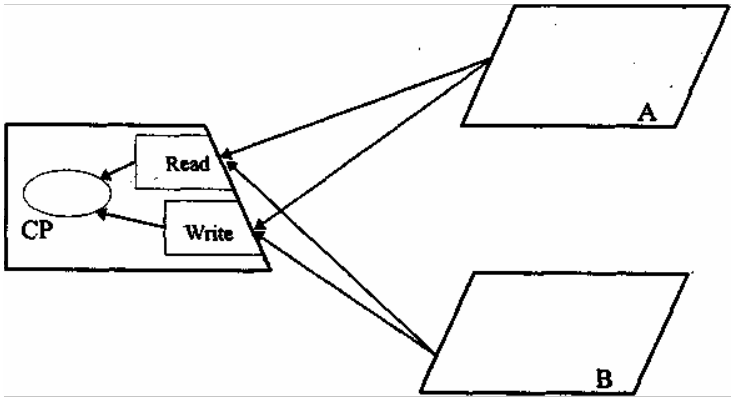


Рис. 12.1

Рис. 12.2



Таким образом, задачи могут взаимодействовать с защищенным модулем, вызывая входы и подпрограммы защищенного модуля.

Вызов защищенной функции позволяет вызывающей задаче считывать данные из защищенного модуля. Несколько задач могут выполнять такое считывание одновременно, вызывая соответству ющую защищенную функцию. При выполнении считывания в теле защищенной функции запрещено изменение данных. Тело защи-щеной функции может содержать вызов другой защищенной функции, но не вызов защищенной процедуры.

Вызов защищенной процедуры позволяет вызывающей задаче как считывать, так и записывать информацию в защищенный модуль. В отличие от защищенной функции при выполнении тела защищенной процедуры *разрешается* изменение информации. Если несколько задач выполняют вызов защищенных процедур, то только одна задача получает возможность работы с вызванной процедурой. В теле защищенной процедуры разрешен вызов и защищенной функции и защищенной процедуры.

Вызов защищенного входа приводит к блокированию вызывающей задачи до тех пор, пока вызванный вход будет *открыт* , то есть условие, указанное в барьере примет значение **True.** Если вход открыт, то выполняется тело вызванного входа и данные могут передаваться или приниматься из вызывающей задачи. Барьер не должен зависеть от параметров входа, но он может зависеть от индекса семейства входов или других переменных , видимых в теле входа. Это позволяет вызывающей задаче "видеть" условие в барьере. Кроме того, барьер может зависеть от глобальных переменных в защищенном модуле. В этом случае появляется возможность управления барьером из других программных структур с помощью вызова защищенной процедуру или входа путем изменения ими глобальных переменных.

В теле защищенного входа разрешается изменение информации, а также использование операторов **Return и Requeue.** С каждым защищенным входом связана очередь. Задачи блокируются в них, пока вход закрыт. Оператор **Requeue** возвращает вызывающую задачу назад в очередь или в другую совместимую с ней очередь.

Для защищенного модуля вводится понятие *защищенного действия* (protected action). Защищенное действие начинается при вызове защищенной операции и завершается после выполнения ее тела.

Новое защищенное действие *не начинается* в защищенном модуле до тех пор, пока выполняется другое защищенное действие. Исключение здесь - если оба защищенных действия являются выполнением защищенной функции. Для подпрограмм различают внутренний и внешний вызов защищенной подпрограммы. Внутренний вызов не отличается от вызова обычной подпрограммы. Внешний вызов приводит к тому, что тело защищенной подпрограммы выполняется как часть защищенного действия.

Если защищенный вход открыт, то соответствующее тело выполняется как часть защищенного действия. Если вход закрыт, то вызывающая задача помещается в очередь.

Общее правило для защищенного действия :

* начало защищенного действия соответствует достижению выполняемого ресурса в защищенном модуле ; чтение обеспечивается защищенной функцией, эсклюзивное чтение и запись - остальными защищенными операциями;
* завершение защищенного действия - завершение соответствующего выполняемого ресурса.

Завершающий шаг защищенного действия всегда связан с проверкой очередей к защищенным входам, прежде, чем будет разрешено новое защищенное действие. То есть, наряду с вызовом защищенной подпрограммы и выполнения тела защищенного входа, защищенное действие включает работу с очередями - добавление и удаление задач из очередей. Обработка очередей выполняется при любом изменении условия в барьере, так как очереди связаны не с задачами, а с барьерами.

Ш СОВЕТЫ:

* Используйте защищенные обьекты для обеспечения взаимного исключения при работе с общими ресурсами.
* Используйте защищенные обьекты для синхронизации задач.
* Избегайте глобальных переменных в барьерах входов.
* Используйте защищенные процедуры для реализации обработчиков прерываний.
* Не используйте незащищенные переменные.
* Отдавайте при организации взаимного исключения предпочтение защищенным типам как альтернативе механизма рандеву с точки зрения скорости работы программы.

12.3 Примеры

П Пример 1.

Пример организации вычислений, где несколько задач используют общую переменную, являющуюся счетчиком .

```
protected Next is    - - спецификация защищенного модуля
   procedure Add ( Value      out Positive ); - - защищенные
   procedure Sub ( Value      out Positive ); - - процедуры
private
   Counter : Integer: = 0 ;        - - общая переменная
end Next;

protected body Next is          - - тело защищенного модуля
   procedure Add (Value :       out Positive ) is
   begin
   Counter: = Counter + Value  1
   : = Counter; end Add;

   procedure Sub (Value :  out Positive ) is begin
Counter: = Counter -1 ; Value : =
Counter; end Sub; end Next;
```

Задачи увеличивают или уменьшают значение переменной Counter , вызывая процедуры Add и Sub из защищенного модуля Next :

```
Next. Add ( z);        Next. Sub ( v);
```

Наличие защищенного модуля гарантирует синхронизированный доступ задач к защищенной переменной Counter. Очереди при работе с защищенным модулем здесь не создются, так как используются только защищенные процедуры, а не защищенные входы.

П Пример 2.

Защищенный модуль С21 выполняет роль буфера, куда задачи А и В записывают и считывают данные. Защищенный модуль С21 должен

обеспечить синхронизированный доступ задач к общему ресурсу, которым является переменная Bank.

```
protected   C21   is
   entry   In_Bank   (X :  in   Elem );
   entry   Out_ Bank (Y : out  Elem );
private
   Flag  :   Boolean : =  False ;
   Bank  :   Elem;          - -  общая  переменная
end   C21 ;

protected body  C21   is
   entry  In_Bank (X :  in   Elem )    when   Flag = False   is
      Bank: = Elem ;
      Flag : = True ; end
   In_Bank;

   entry  Out_Bank (Y : out   Elem )   when Flag = True   is
      Elem : =    Bank;
      Flag  : =    False ;
   end  Out_Bank;

end  C21;

task A;
task B;

task   body   A   is

   C 21. In_Bank ( Data);      - - вызов входа end   A;

task  body B is

   C21.Out_Bank(Res);

end B ;
```


П Пример 3. Настраиваемый пакет для работы с буфером [12].

```
generic
   type  Item   is  private;              - - параметры  настройки
   Maximum_Buffer_Size   :  in Positive;

package  BOUNDED_BUFFER_PACKAGE    is

   subtype Bufferjndex  is  Positive  range
                                  1 ..   Maximum_Buffer_Size ;
   subtype Buffer_Count  is  Natural  range
                                  0.. Maximum_Buffer_Size ; type
   Buffer_Array  is  array  ( Bufferjndex)  of  Item;

   protected       type  Bounded_Buffer  is  -  -  защищенный  тип

      entry Get ( X : out Item );          -  -  защищенные  входы

      entry Put ( X : in Item );

   private

      Getjndex  :  Bufferjndex : =  1 ;      - - приватная часть
      Putjndex  :  Bufferjndex : =  1 ;

      Count      :  Buffer_Count : = 0 ;
      Data        :  Buffer_Array;

   end   BoundedJBuffer; end

BOUNDED_BUFFER_PACKAGE; package  body

BOUNDED_BUFFERJ=>ACKAGE  is

   protected  body   BoundedJ3uffer  is  - - тело защищенного
                                                - -  типа
```

```
        entry  Get(  X :    out    Item)
            when   Count => 0 is  - - тело - - защищенного     входа с
        барьером begin

            X : =  Data (Getjndex);

        Getjndex : = (Getjndex mod Maximum_Buffer_Size) + 1; Count:

        =  Count -1 ; end Get;

        entry  Put( X :   in Item)
                              when Counr < MaximumJ3uffer_Size  is
        begin                                  - - тело защищенного входа

        Data ( Getjndex) : = X ;                                  - -          c барьером

        Put Jndex : = ( Putjndex mod MaximumJ3uffer_Size) + 1 ;

        Count: =   Count + 1 ;

        end    Put;

    end   Bounded_Buffer; end

BOUNDEDJSUFFEFLPACKAGE;
```


Глава13.СТРУКТУРА ПРОГРАММЫ И РАЗДЕЛЬНАЯ КОМПИЛЯЦИЯ

Основное назначение языка Ада - программирование больших программных систем различного назначения, обладающих высокой надежностью. Существует достаточно много методик разработки таких программ, основанных на различных подходах :

- нисходящее проектирование
- восходящее проектирование
- модульное проектирование
- обьектно-ориентированное проектирование
- функциональное проектирование
- структурное проектирование и др.

Ада 83 - прекрасный язык для структурного проектирования, проектирования "сверху-вниз" и "снизу-вверх", что доказывает его успешное применение в ряде больших проектов [13].

Появление в Аде 95 средств обьектно-ориентированного программирования делает язык дополнительно мощным инструментом обьектно-ориентированного проектирования.

В языке имеются различные механизмы, обеспечивающие поддержку перечисленных выше подходов к проектированию. Они основываются на имеющихся в языке:

- средствах раздельной компиляции
- средствах обьектно - ориентированного программирования
- средствах программирование для распределенных систем
- средствах программирования для параллельных систем.

На поддержку эффективных методов проектирования программ направлены также средства создания гибких многовариантных структур программы, основанных на модульности, развитой системе библиотек, а также возможности *раздельной компиляции.*

Концепция *дочерних пакетов* обеспечивает возможность выделения подсистем иерархических библиотечных модулей, за счет чего большая система может быть структурирована на ряд подсистем. Подсистема используется для объединения логически связанных библиотечных модулей, которые вместе реализуют абстракцию высокого уровня.

Абстракция и инкапсуляция поддерживаются концепцией пакетов и приватных типов. Связанные данные и пояггоогпяммм мп

группироваться вместе и рассматриваться как один обьект. Информационное сокрытие реализуется через строгую типизацию и раздельную компиляцию пакетов и подпрограмм. Исключения и задачи - дополнительные средства языка, влияющие на структуру программы.

13.1 Библиотечные модули и раздельная компиляция

При формировании структуры программы , состоящей из используемых в языке видов программных модулей, она может быть выполнена :

- в виде единой программы, содержащей все используемые модули;
- в виде программы, где все (или часть) используемых модулей находятся вне программы в *библиотеке*.

Понятие *библиотеки* (library), *библиотечного элемента (* library items) и *библиотечного модуля* (library unit): основополагающие в языке при рассмотрении вопросов структуры и компиляции программы.

Библиотечный элемент - это :

- описание подпрограммы или пакета;
- тело подпрограммы или пакета;
- описание настройки или конкретизации;
- описание переименования подпрограммы, пакета или настройки.

При этом описания и переименования могут быть приватными, а библиотечные элементы могут иметь иерархическую структуру, то есть иметь родительские и дочерние библиотечные элементы.

Библиотечный модуль - это либо библиотечный элемент, либо *субмодуль* (subunit/

Субмодуль - *раздельно* компилируемое тело подпрограммы, пакета, задачи, защищенного модуля, спецификации которых находятся в других модулях вместе со *следом* тела.

В соответствии с принятой в языке концепцией построения структуры программы - программа на языке Ада 95 - набор *Сегментов* (set of partitions), каждый из которых может выполняться :

- в отдельном адресном пространстве
- на отдельном компьютере.

В общем случае программа формируется из программных модулей, определенных в языке: подпрограмм, пакетов, задач, защищенных модулей, настраиваемых модулей. Модули в программе могут быть *вложенными*, библиотечными, либо *компилируемыми раздельно*.

В Аде при реализации вопросов компилирования программы принята концепция *раздельной компиляции (* separate compilation). При раздельной компиляции каждого компилируемого модуля выполняется контроль согласования данного модуля с уже откомпилированными. Этим раздельная компиляция отличается от независимой компиляции, где такая проверка не выполняется. То есть связь между раздельно компилируемыми модулями контролируется точно так же, как и между частями одного компилируемого модуля. Для обеспечения такого контроля компилятор хранит всю необходи мую дополнительную информацию о каждом компилируемом модуле, что облегчает проверку согласования интерфейсов между разными модулями.

Текст Ада программы обрабатывется за одну или несколько компиляций. Каждая компиляция - обработка последовательности *компилируемых модулей (* compiled units).

Компилируемые модули - это компоненты программы , которые можно компилировать раздельно.

Компилируемый модуль - это :

- библиотечный модуль;
- вторичный модуль.

Вторичный модуль - раздельно компилируемое тело соответствующего библиотечного модуля или субмодуль другого компилируемого модуля.

Библиотечный модуль можно использовать в любом контексте, а субмодуль - только в родительском модуле, хранящем его след.

В процессе компиляции на вход компилятора подаются компилируемые модули в виде библиотечных модулей или вторичных модулей.

Результат компиляции библиотечного модуля заключается в том, чтобы определить его как новый библиотечный элемент, а вторичного модуля - определить либо тело библиотечного модуля, либо тело программного модуля, описанного в другом компилируемом модуле.

Простая программа может состоять из простого компилируемого модуля. Компиляция может не требовать компилируемого модуля. Например , состоять только из прагмы.

D Примеры библиотечных модулей:

```
procedure A is
```

```
                                    - - процедура
end ...A;

package ZZZ is

    procedure ln_Buf ( . . . ) ;          - - пакет
    procedure Out_Buf (... );

end ZZZ;

with RR;
package body MM is                        - - тело пакета

end MM;

with TT;
package SS renames TT;                    - - переименование

private procedure BB.R () ; - - приватная дочерняя процедура

generic
    type Volt is private ;                - - настраиваемая процедура
procedure Test (X : in Volt);
```

13.1.1 Спецификаторы контекста

Для указания библиотечных модулей, используемых в компилируемом модуле, применяется *спецификатор контекста.* Спецификаторы контекста (Context clauses) имеют две формы:

- *спецификатор совместимости*

WITH *Список_Библиотечных_Модулей;*

- *спецификатор использования*

USE *Список _Библиотечных_Пакетов;*

В *Списке_ Библиотечных_Модулей (Пакетов)* указываются простые имена библиотечных модулей или пакетов. В спецификаторе **use**

допустимы только имена тех библиотечных пакетов, которые предварительно были указаны в спецификаторе **with.**

Спецификаторы контекста могут применяться только либо в спецификаторе контекста библиотечного модуля (собственно к модулю или к его вторичному модулю), либо к компилируемому моду-лю (самому модулю или его субмодулю).

Библиотечный модуль, указанный в спецификаторе **with,** видим непосредственно внутри компилируемого модуля (исключая случаи сокрытия).

D Например:

```
with Bridge; use Bridge ;       - - используемый в программе
                                  - - библиотечный пакет
procedure Methods is

    - - видны и доступны все ресурсы билиотечного модуля Bridge end

Methods;
```

Спецификаторы **with** задают связь (зависимость) между компи-лируемыми модулями и библиотечными модулями, упомянутыми в спецификаторе. Эта зависимость определяет порядок компиляции и перекомпиляции модулей, а также порядок предвыполнения компи-лируемых модулей. D Пример :

```
with A, B, C; use
C; procedure Dust is

    use A;

    procedure Water is      - - вложенная процедура
    use B;

    end Water;

end Dust;
```

in*

Здесь используются три библиотечных пакета A, B и C. Ресурсы пакета B используются только во вложенной процедуре, что указывается соответствующим спецификатором use.

13.1.2 Субмодули

Для обеспечения возможности иерархического построения программы в языке введено понятие *субмодуля,* которое используется для реализации раздельной компиляции, при которой тело соответствующего программного модуля может описываться и компилироваться раздельно.

Субмодули подобны дочерним модулям, с той разницей, что субмодули дополнительно поддерживают раздельную компиляцию; родитель содержит *след тела,* указывающий существование и мес-то размещения каждого из субмодулей; описания , появившиеся в теле родительского модуля, могут быть видимыми внутри субмодуля.

Виды следа:

Спецификация_Подпрограммы **IS SEPARATE ;**
PACKAGE BODY *Имя_Пакета* **IS SEPARATE ;**

TASK BODY *Имя_задачи* **IS SEPARATE ;**
PROTECTED BODY *Имя_Защищенного_Модуля* **IS SEPARATE;**

След (stub) тела должен размещаться непосредственно в теле библиотечного пакета или разделе описаний компилируемого модуля. Если тело модуля задается следом этого тела, то субмодуль с этим телом компилируется раздельно.

Субмодуль может содержать тела процедур, пакетов, задач и защищенных модулей.

Описание субмодуля:

SEPARATE *(Имя_Родительского_Модуля)*

Тело_Соответствующего_Модуля
END;

П Например:

```
separate ( Ajax) package  - -  субмодуль с телом  пакета
body  Shep  is
```



```
end  Shep;
```

Для каждого субмодуля определяется *родительский* модуль, то есть компилируемый модуль, содержащий *след* данного субмодуля^. Родительский модуль является *предком,* если он - библиотечный модуль.

П Пример:

```
- - родительский модуль
procedure Bank  is A, B,
C:  integer;
    procedure ZZ (V:  in out integer) is separate; - - след тела
                                                    - -процедуры
    package  Data  is

        procedure Count (X, Y: in integer; S : out integer);

        function Deposite (W : Positive); end

    Data;

    package  body Data is separate ; - - след тела  пакета

begin

    Data.Count (A, B, C);
      …
    Data. Deposite (B);

    ZZ(C);
     …
end  Bank;

separate (Bank)          --  субмодуль
procedure ZZ(V: in out integer) is

end  ZZ;
```

```
separate ( Bank);              - - субмодуль
package body  Data   is

    procedure   Macler( T : in out integer) is separate; - - тел след end

Data;

separate ( Bank. Data)  - - субмодуль
procedure   Macler( T: in out integer)   is

      end Macler;
```

СОВЕТЫ:

Ш * Размещайте спецификацию каждого библиотечного пакета в отдельном файле.

* Минимизируйте использование субмодулей.

» Испльзуйте приватные дочерние модули и спецификатор **With** вместо размещения в теле.

* Вместо вложенного размещения в теле пакета , используйте личные дочерние модули и спецификатор **With** в родительском модуле.

13.2. Порядок компиляции

Правила и порядок компиляции модулей является непосредственным следствием правил видимости.

Порядок компиляции, как уже обсуждалось выше, зависит от структуры программы, то есть он определяется наличием в программе библиотечных модулей, субмодулей, вторичных модулей.

Общие правила порядка компиляции:

1. Компилируемый модуль должен компилироваться после компиляции всех библиотечных модулей, указанных в его спецификаторе контекста **with.**
2. Вторичный модуль (тело подпрограммы или пакета) должен компилироваться после соответствующего библиотечного модуля. то есть' тела пакетов или подпрограмм должны компилироваться после соответствующей спецификации.
3. Субмодуль компилируется после компиляции своего родительского модуля.

Перекомпиляция связана с изменением отдельных модулей и ,как правило, не требует полной перекомпиляции всей программы.

Общие правила порядка перекомпиляции:

1. Изменение библиотечного модуля требует перекомпиляции компилируемого модуля, который его использует.
2. Изменение компилируемого модуля не требует перекомпиляции используемых им библиотечных модулей.
3. Изменение библиотечного модуля требует перекомпиляции вторичного модуля.
4. Изменение родительского модуля требует перекомпиляции субмодуля.
5. Изменение субмодуля не требует перекомпиляции родительского модуля. Изменение вторичного
6. модуля не влияет на другие компилируемые модули, кроме субмоделей этого тела.

В общем случае для библиотечных пакетов при рассмотрении перекомпиляции следует считать устаревшим тело пакета после перекомпиляции соответствующей спецификации.

На порядок компиляции модет оказать влияние использование прагмы **Inline** , связанной с открытыми подстановками; также оптимизация, осуществляемая компилятором, и др.

Если прагма **Inline** применяется к описанию подпрограммы, то выполнение открытой подстановки требует того, чтобы тело пакета было откомпилировано раньше, чем модули, использующие эту подпрограмму. То есть открытая подстановка создает зависимость вызывающего модуля от тела пакета.

При выполнении оптимизации компилятор может компилировать несколько модулей, тем самым создавая дальнейшую зависимость между этими модулями. Это может повлиять на перекомпиляцию.

Пример.

```
П Версия 1.
- - файл  A.ada
procedure  A is package
B is
                        - - вложенный пакет

   end B;
   package   body B is
```

```
        end B;
begin

end A;
```

На вход компилятора поступает один компилируемый модуль (файл A.ada). При любых изменениях в процедуре A , в спецификации или теле пакета B требуется перекомпиляция всей программы.

D Версия 2.

```
- - файл B.ada
package B is          - - библиотечный пакет

end B;
package body B is

end B;

- -    файл  A.ada with
B;  use B; procedure
A is begin            - - основная  программа

end A;
```

Программа представлена в виде двух компилируемых модулей. Первым компилируется пакет B (файла B. ada), а затем - процедура A (файл A.ada). Перекомпиляция : изменения в пакете требуют перекомпиляции в основной программе; изменения в процедуре A не требуют перекомпиляции в пакете B.

П Версия 3.

```
- - файл  B.ads
package  B is   - - библиотечный элемент (описание B )
  « . .
end   B;
- - файл  B.adb
package body   B is   - - библиотечный  элемент (тело B )

end  B ; - - файл
A.ada with B; use
B; procedure  A is

end  A;
```


Программа представлена в виде компилируемы модулей A (файл A.ada), B (файл B.ads) и вторичного модуля B (файл B.adb). Первым компилируется файл A.ada , а затем - в любой последовательности остальные модули. Перекомпиляция : изменения в B. ads требуют перекомпиляции A.

П Версия 4.

```
- - файл
A.ada procedure  A is
package  B  is          - - родительский модуль

   end B;
   package  body B is separate ; - - след тела пакета B
begin

end A;

- -    файл    B.adb
separate   (A)
package   body B is     - - субмодуль с телом

end   B;
```

Программа представлена в виде родительского модуля и субмодуля. Порядок компиляции - сначала родительский модуль (файл A.ada , а затем - субмодуль (файл B.adb) . Перекомпиляция:

- изменения в родительском модуле требуют перекомпиляции субмодуля;
- изменения в субмодуле не требуют перекомпиляции родительского модуля.

13.3 Выполнение программы

В Аде 83 выполнение программы было связано с понятием *главной программы*, которое определялось конкретной реализацией. Однако для любой реализации роль главной программы могла выполнять процедура без параметров.

В Аде 95 вместо понятия главная программа введено понятие *главная подпрограмма*. Это связано с измененим концепции Ада программы в языке. Ада программа теперь - это *набор* Сегментов *(* set of partitions) , каждый из которых может выполняться параллельно, возможно в отдельном адресном пространстве и возможно на отдельном компьютере.

Сегмент программы - это программа или часть программы, которая может быть вызвана из-вне реализации Ады. Пользователь теперь может назначить (связать) для сегмента библиотечный модуль, используя для этого соответствующие средства языка.

Пользователь может запланировать одну подпрограмму как главную для сегмента. То есть главная подпрограмма обязательно должна быть подпрограммой.

Каждый сегмент имеет *анонимную окружающую задачу* **(Environvent_Task**), под которой подразумевается задача, которая осуществляет предвыполнение библиотечных элементов окружа-ющей описательной части, и затем выполняет вызов главной подпрограммы, если она имеется. Выполнение сегмента - это выполнение окружающей задачи.

Структура окружающей задачи:

```
task Environment_TasK ;

task body Environment_Task is
      . . .  (1)  - -     описание  окружения
                  - -     (библиотечные элементы )
begin
      …"  (2 )   - -     вызов главной процедуры, если она есть
end   Environment_Task;
```


Описание окружения (1) - это последовательность описательных элементов, содержащих копии библиотечных элементов, входящих в сегмент.

Последовательность операторов (2) - вызов главной подпрограммы , если она есть, и пустой оператор, если ее нет. Если главная подпрограмма имеет параметры, они передаются. Механизм формирования параметров и результата определяется реализацией.

Выполнение программы - выполнение набора сегментов. Выполнение сегмента начинается с выполнения ее окружающей задачи, заканчивается - когда окружающая задача завершается и включает выполнение всех задач сегмента.

Реализация должна обеспечить механизм взаимодействия сегментов с помощью специальных пакетов и прагм. Стандартные прагмы для такого взаимодействия определены в приложении языпса "Распре-деленные системы " (Annex E).

Сегмент может не иметь главной подпрограммы. В этом случае выполнение сегмента - предвыполнение различных библиотечных элементов и задач, созданных при этом предвыполнении.

Перед выполнением главных подпрограмм все библиотечные модули вместе с телами , необходимыми для нее, *предвыполняются* (elaboration).

К предвыполняемым библиотечным модулям относятся модули, перечисленные в спецификаторе контекста **with** главной подпро-граммы, ее тела и субмодулей , а также модули , упомянутые в спецификаторах контекста этих библиотечных модулей, их тел , субмодулей и т.д.

Библиотечный модуль из спецификатора контекста субмодуля должен быть предвыполнен до тела библиотечного модуля - родителя этого субмодуля.

Для предвыполнения должно быть обеспечено следующее требование : тело любого библиотечного модуля предвыполняется прежде любого компилируемого, при предвыполнении которого необходимо предвыполнение тела этого библиотечного модуля.

Для управления порядком предвыполнения библиотечных моду-лей используются следующие прагмы : **Preelaborate, Pure, Elaborate , Elaborate_AII, Elaborate_Body.**

Прагмы для библиотечных модулей:

pragma Preelaborate [(*Имя_Библиотечного_Модуля*
pragma Pure [(*Имя_Библиотечного _Модуля]*
pragma Elaborate_Body [(*Имя_Библиотечного _Модуля*

Прагмы, допускаемые только внутри спецификатора контекста

pragma Elaborate [(*Имя_Библиотечного_Модуля*
pragma Elaborate_All *[(Имя_Библиотечного_Модуля*

Прагма **Elaborate** предписывает, что тело указанного библиотечного модуля предвыполняется перед текущим библиотечным элементом.

Прагма **Elaborate_AII** означает, что каждый библиотечный элемент, необходимый для указанного описания библиотечного модуля предвыполняется прежде текущего библиотечного элемента.

Прагма **Elaborate_Body** означает, что тело библиотечного модуля выполняется немедленно после его определения.

Прагма **Preelaborate** связана с довыполнением библиотечных модулей в некоторых стуациях.

В языке введено понятие *чистого библиотечного элемента*. Это - предвыполняемый библиотечный элемент, который не содержит объявления ни одной переменной или является ссылочным типом.

Прагма **Pure** используется для обьявления того, что данный бибилиотечный модуль является чистым. Если прагма применяется к бибилиотечному модулю, то его компилируемый модуль должен быть чистым и он будет зависеть семантически только от компилируемых модулей других библиотечных модулей, которые объявлены как чистые.

Если библиотечный модуль описан как чистый, то при реализации разрешается опускать вызов подпрограммы библиотечного уровня из библиотечного модуля, если результаты не нужны после вызова.

13.4 Иерархические библиотеки

В Дде 95 концепция использования библиотечных модулей для построения больших и сложных программных систем получила дальнейшее развитие в виде механизма *иерархических библиотек*.

Иерархические библиотеки основываются на ведении понятия *родительского* (parent) и *дочернего* (child) модулей. Дочерние модули в свою очередь могут быть либо *публичными* ((public child), либо *приватными* (private childe).

Имя дочернего библиотечного модуля является составным и определяет его позицию в иерархии библиотеки. Например, P.C - имя дочернего модуля, для которого модуль P является родительским. На рис . 13.1 представлено дерево иерархии библиотечных модулей.

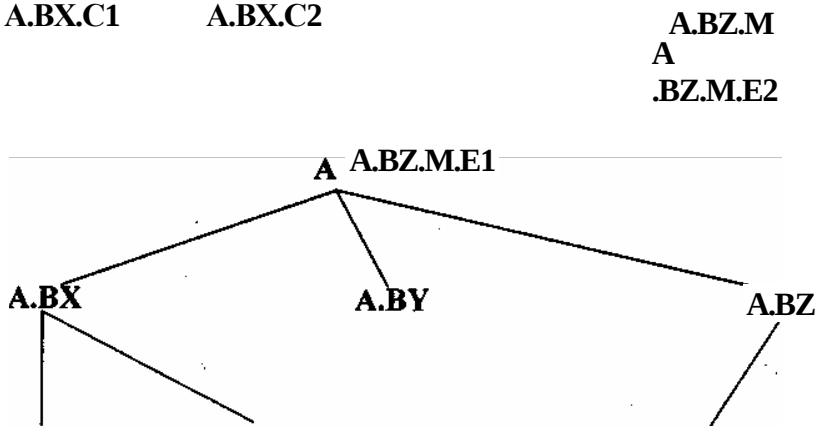


Рис. 13.1

Если дочерний библиотечный модуль P.C описан в контексте with, то он рассматривается как модуль, находящийся в его родительском модуле P, и для него сохраняются правила именования и види-

мости как для вложенных модулей, несмотря на то, что модуль P.C является самостоятельным библиотечным модулем

Добавление нового дочернего модуля P.X не требует немедленной перекомпиляции существующих библиотечных модулей.

Использование иерархии библиотек позволяет решить многие проблемы, существовавшие в Аде 83. Например, создание пакетов, работающих с общим личным типом, или расширение ресурсов существующего пакета. Такие проблемы либо не решаются в Аде 83, либо требуют соответствующих изменений в программе и перекомпиляции модулей.

П Пример 1.

Имеется пакет VEC для работы с векторами. Ресурсы пакета - приватный тип Vector и две функции Sum и Sub.

```
package VEC is

    type Vector is private;
    function Sum (VA, VB : Vector ) return Vector ;
    function Sub (VA, VB : Vector) return Vector;

private
    type Vector is array (1.. 20 ) of float;

end VEC;
```

Необходимо добавить к пакету еще подпрограмму Mult для умножения векторов. Это можно сделать:

- путем изменения спецификации пакета VEC, добавив в него спецификацию процедуры Mult;
- путем создания дочернего пакета VEC.MLT.

Первый подход потребует перекомпиляции не только самого модуля , но и всех модулей, зависящих от него. Второй , основанный на механизме иерархических библиотек, гораздо эффективнее.

```
package VEC.MLT is
    function Mult (VA, VB : Vector) return Float; end
VEC.MLT.
```


Так как пакет VEC.MLT является дочерним для пакета VEC , то в нем приватный тип Vector доступен и может использоваться в подпрограмме Mult.

Правила видимости для дочерних библиотечных пакетов:

- дочерний библиотечный пакет декларируется внутри области видимости определения его родительского модуля после его спецификации;
- приватная часть спецификации и тело дочернего пакета "видят" приватную часть родительского модуля.

П Пример 2.

```
package BOX is

    type Data is private ;

private end

BOX;

- - дочерний пакет
package BOX.ZIP is
    type Result is private;

private

end BOX.ZIP;
```

В силу того, что пакет BOX.ZIP является дочерним пакетом для пакета BOX, при определении в нем приватного типа Result можно использовать тип Data . Клиенты пакета BOX (модули использу-ющиее этот пакет) не требуют перекомпиляции, если дочерний пакет изменяется; новые дочерние модули могут быть добавлены без изменения существующих клиентов.

13.5 Приватные дочерние модули

Приватные дочерние модули (private child units) предназначены для ограничения видимости в иерархических библиотеках. Такие

проблемы возникают при разработке больших программных систем, в которых необходимо ограничить видимость для клиента.

В Аде 83 в этом плане невидимым было тело субмодуля, которое компилировалось отдельно. Однако субмодуль подлежал перекомпиляции при изменениях в телах модулей более высокого уровня иерархии.

В Аде 95 такая проблема решается путем использования дочерних модулей, которые являются полностью приватными для своих родителей.

Пример:

```
package A is

   type FFA is private;

private
   type FFA is new Base;
end A;

package A.X is                    -- дочерний пакет от A
   X1.X2.X3: exception;
end A.X ;

                                  -- дочерний пакет от A
package A.Y is
   type FFY is ... ;
   function Zzy ( Y1 : FFY ) return Ffy ;
   procedure Tty ( Y2 , Y3 : in FFA );

end A.Y ;

procedure A.T ( T1, T2 : out Float );  -- дочерняя процедура

private package A.U is  -- приватный дочерний пакет

end A.U ;

private package A.V is  -- приватный дочерниий пакет

end A.V ;
```


В данном примере родительский пакет A содержит тип FFA , который используется во всей программе. Система содержит три общих (публичных) дочерних модуля : пакеты A . X , A . Y и процедуру A . T . Кроме того, в подсистеме имеются два приватных дочерних модуля - пакеты A . U и A . V .

Приватный дочерний модуль может быть определен в любой точке дочерней иерархии. Правила видимости при этом такие же как и для общих дочерних модулей, за исключением следующих особенностей :

- приватный дочерний модуль видим только только внутри поддерева иерархии, корнем которой является его родительский модуль;
- видимая часть приватного дочернего модуля может иметь доступ к приватной части его родителя. (Про этом невозможен прямой экспорт информации о приватном типе к пользователю, потому что это не в его видимости. Также невозможен косвенный экспорт через общие модули).

В нашем примере, так как приватный дочерний модуль есть прямой дочерний модуль пакета A , пакет A . U видим в телах A, A . Y и A . T (модуль A . X тела не имеет) , а также видим в обоих телах и спецификации пакета A . V . Но он невидим вне A и пользовательский пакет не может иметь доступ к A . U вообще.

При построении иерархических библиотек разрешается использовать настраиваемые модули. Любой родительский модуль может иметь настраиваемые дочерние модули. При этом настраиваемый родительский модуль может иметь только настраиваемые дочерние модули. Для настраиваемого родительского модуля его настраивамый дочерний модуль настраивается в любой точке его видимости в обычном порядке. Настраиваемый родительский модуль должен конкретизироваться раньше своих дочерних модулей. П Пример настраиваемого пакета A :

```
generic
   type Base is delta < >; N :
   integer;

package A is

end A;
```

```
generic
package  A . X  is

end A . X ;
```

Конкретизация пакетов A и A . X :

```
With  A;
package N_A  is new A(100, Float);

With A . X ;
package  N_A. N_X  is  new  N_A . X ;
```

Очевидно, что иерархические системы библиотек в Аде 95 являются мощным средством построения больших программных систем из компонентов подсистем.

Ш СОВЕТЫ:

* Используйте дочерние библиотечные модули, если новый библиотечный модуль является логическим расширением имеющейся абстракции.
* Если новый библиотечный модуль есть независимый, то есть вводит новую абстракцию, которая зависит только частично от существующей, то инкапсулируйте новую абстракцию в раздельный библиотечный модуль.
* Используйте дочерний библиотечный пакет для реализации подкомпонент системы.
* Используйте публичные дочерние модули для всех частей подсистемы, которые должны быть видимыми при использовании подсистемы.
* Используйте дочерние библиотечные пакеты для управления видимостью частей подсистемы.
* Используйте приватные дочерние пакеты для всех описаний, которые не должны быть использованы вне подсистемы.
* Используйте дочерние модули для представления различных видов явления для разных пользователей (клиентов).
* Используйте дочерние пакеты вместе обычных пакетов для представления различных видов абстракций.

13.6 Предопределенная библиотека

В языке имеется развитая предопределенная стандартная библио тека, содержащая предопределенные пакеты. В соответствии с приня той в языке иерархией библиотек стандартная библиотека упорядочена определенным образом. Все библиотечные пакеты рассматриваются как дочерние от пакета **Standard.** На первом уровне иерархии дочерних пакетов располагаются три дочерних пакета : **System, Interface, Ada.** Дочерние пакеты от этих трех образуют последующие уровни иерархии.

Пакет **System** имеет дочерние модули **System . Storage_Elements, System . Storage_Pools** для работы с памятью.

Пакет **Interface** имеет дочерние модули **Interface .C , Interface .COBOL, Interface .Fortran,** которые поддерживает средства совместимости с программами на других языках.

Пакет **Ada** является родительским для остальных основных предопределенных пакетов языка. Для совместимости программ, написанных на Аде 83, следует выполнить следующее переименование

```
with Ada.TextJO;
package   TextJO   renames   Ada.TextJO;
```

Спецификация пакета Ada :

```
package Ada  is
    pragma Pure (Ada );
end  Ada;
```

Пакет Ada является пустым (исключая прагму его **Pure**). Среди дочерних пакетов присутствуют пакеты :

- **Ada .Text_lO** - для реализации ввода-вывода
- **Ada . Excecption** - для реализации исключений
- **Ada . Numeric** - для численных вычислений.

В свою очередь пакет **Numerics** является родительским для нескольких дочерних пакетов, которые обеспечивают дополнитель ные средства для машинных вычислений. Это такие пакеты как

- **Generic_Elementary_Function**
- **Float_Random**
- **Discrete Random.**

ffl СОВЕТЫ:

* Используйте константы Pi и e из пакета **Ada . Numerics .**

ИЗМЕНЕНИЯ:

О Введено понятие дочернего модуля.

(З Введено понятие иерархии библиотек и механизм работы с ними. О В структуре Ада программы используется понятие сегмента. О Пересмотрено понятие главной программы.

Глава14. П Р А В И Л А В И Д И М О С Т И

Правила видимости задают область действия описаний и определяют, какие идентификаторы, символьные литералы и знаки операций видимы в (из) различных местах текста программы.

Изменения в правилах видимости в Аде 95 направлены в первую очередь на то, чтобы сделать их более ясными и содержательными по-сравнению с Адой 83. Кроме того, в новых правилах видимости не-обходимо было учесть появление в языке таких средств, как механизм иерархических библиотек, средств обьектно-ориентированного программирования и др.

К наиболе важным изменениям в правилах видимости следует отнести возможность применения спецификатора использования **Use** для типов (операций), а также возможность *переименования* тел подпрограмм, настраиваемых модулей и библиотечных модулей.

14.1 Зона и область действия описания

Правила видимости (visibility rules) основываются в языке на понятиях *зоны описания* и *области действия описания.*

Зона описания (region declarative) - часть текста программы, внутри которой имеется :

- любое описание, отличное от описания перечисления;
- оператор блока;
- оператор цикла;
- оператор принятия (**accept);**
- обработчик исключений.

Зона описания может включать также дополнительно следую-щее:

- если в зоне описания имеется след тела, то - соответствующее тело;
- если в зоне описания имеется описание типа записи, то - соот-ветствующий контекст представления записи;
- если в зоне описания имеется описание библиотечного модуля, то - описания всех его дочерних модулей ;

Каждая зона описания рассматривается как логически непрерывная часть текста программы.

Зоны описания могут быть вложены в другие зоны описания, на-пример при вложенности пакетов, подпрограмм, задач, блоков и др., а также если в них имеются описания перечисляемых типов, оператор цикла или оператор принятия **accept.**

Описание находится *непосредственно* в описательной части, если его зона описания есть внутренняя зона описания, то есть является вложенной.

Локальное описание находится непосредственно внутри зоны описания. *Глобальное описание* находится внутри другого описания, являющимся внешним по отношению к рассматриваемому. Например, зона описания библиотечного пакета **Standard** содержит глобальные описания, так как этот пакет является внешним для всех библиотечных модулей.

Дочерние модули от библиотечного считаются находящимися внутри зоны описания родительского модуля, несмотря на то , что они располагаются вне спецификации или тела родительского модуля. Все библиотечные модули являются *потомками (дочерними* модулями) пакета **Standard** и находятся в его зоне описания . Они не находятся внутри спецификации (тела) пакета **Standard** , но находятся внутри его зоны описания.

Область действия описания (scope of declarations) - часть текста программы, где и только где имеют силу описания : обьявленные описания идентификатора, символьного литерала, знака операции.

Область действия описания, находящегося непосредственно в зоне описания, распространяется от начала описания до конца зоны. Этот раздел называется *непосредственной* областью действия описания. Для перечисленных ниже описаний область действия распространяет-ся за пределы непосредственной области действия:

- описание в видимом разделе описания пакета;
- описание входа ;
- описание компоненты ;
- спецификация дискриминанта;
- спецификация параметра;
- описание параметра настройки.

Если отсутствует описание подпрограммы, то спецификация подпрограммы, заданная в теле подпрограммы или следе тела, действует как описание и распространяется как спецификация параметра в указанном выше списке описаний.

Видимая часть описания - часть текста описания, содержащего описания, видимые из-вне. *Приватная часть описания* - не видима извне. Видимая и невидимая части описания определяются только для программных модулей и сложных типов.

Область действия всегда содержит непосредственную область действия. В дополнение для данного описания, которое имеет место непосредственно в видимой части внешнего описания или есть публичным дочерним модулем для внешнего описания, область действия данного описания распространяется до конца зоны внешнего описания , исключая область действия библиотечных элементов.

E

В языке существует нотация для введения видимых описаний, которые не являются прямо видимыми. Например, спецификация параметров находится в видимой части описания подпрограммы и они могут использоваться при помощи именованной нотации при вызове подпрограммы. В то же время, описание ресурсов из видимой части пакета может обозначаться через расширенное имя при вызове из-вне пакета или может быть сделано непосредственно видимым с помощью спецификатора **Use.**

14.2 Видимость

Правила видимости определяют какие описания видны (или непосредственно видны) в каждом месте текста программы. Они применимы к *явным* и *неявным* описаниям.

Описание видимо только в определенной части своей области действия; оно начинается в конце описания , а в спецификации пакета - после зарезервированного слова is , следующего после идентификатора пакета.

Видимость может быть *прямой (непосредственной)* или *видимостью по имени.*

Описание видимо *по имени* в точках программы для :

- описания, находящегося в видимом разделе описания пакета ;
- описания входа задачного типа;
- описания компоненты описания типа ;
- спецификации дискриминанта описания типа ;
- спецификации параметра подпрограммы или описания входа ;
- описания параметра настройки модуля.

Описание видимо прямо (*непосредственно*) там, где нет видимости по имени. В этом случае описание видимо непосредственно в определенном разделе его области действия; этот раздел распространяется до конца непосредственной области действия описания, за исключением ситуации, когда описание скрыто.

Описание, находящееся в видимом разделе пакета можно сделать непосредственно видимым с помощью спецификатора использования Use. (См. Раздел 14.3).

Описание *скрыто* во внутренней зоне описания, если в нем имеется *омоним* этого описания. В этом случае внешнее описание является *скрытым* в непосредственной области действия внутреннего омонима.

Описание является омонимом другого описания, если они имеют один и тот же идентификатор.

D Пример.

```
procedure A is
   X, Y, Z : integer ; - - глобальные переменные для процедуры B

   procedure B is - - вложенная процедура
            - - локальные переменные, невидимые в процедуре A X
      : integer ; - - омоним, так как в процедуре A имеется
                        - - переменная с именем X W, S:
   integer; begin
            - - можно использовать Y, Z, A . X - глобальные ;
    - -.  S , W, X - локальные W : = X ;     - -
означает   W : = B . X S : = Y ;     - - означает   S
: = B . Y X : = A . X ;  - - означает   B . X : = A. X S :
= W ;     - - означает   B . S : = B . W end  B;
begin - -  тело A
   . . .    - - разрешено использование только переменных  X . Y, Z
end A;
```

В данном примере имеются описания в процедуре A и процедуре B. Так как процедура B вложена в процедуру A, описания в процедуре A являются глобальными и видимыми в B. Описания в процедуре B - локальные и невидимы в процедуре A. Если один и тот же идентификатор находится в разных описаниях (является омонимом), то он соответствует разным понятиям. В нашем примере таким является идентификатор X. При его использовании в теле процедуры B идентификатор X рассматривается как локальная переменная и относится к B . Доступ в теле процедуры B к глобальной переменной X должен выполняться с помощью *составного* имени A . X , которое делает глобальную переменную X из процедуры A непосредственно видимой в теле B.

В спецификации подпрограммы скрыто каждое описание, совпадающее с описанием подпрограммы. Аналогичное правило действует для конкретизации настройки с описанием подпрограммы, .в описании входа, в разделе формальных параметров оператора **accept.** В этих случаях описания не являются видимыми ни по имени, ни непосредственно.

14.3 Спецификаторы использования

Спецификаторы использования обеспечивают прямую видимость определенных ресурсов в заданных частях программы. Различают два вида спецификатора использования:

- спецификатор использования пакета;
- спецификатор использования типа.

Спецификатор использования пакета:

USE *Имя_Пакета {, Имя_пакета } ;*

Обеспечивает прямую видимость всех ресурсов из видимой части спецификации всех перечисленных пакетов. Спецификатор использования типа:

USE TYPE *Имя_Подтипа {, Имя_Подтипа } ;*

Обеспечивает прямую видимость примитивных операций для перечисленных в нем типов.

Для каждого спецификатора использования, имеющегося в некоторой части текста, определяется *область действия описаний спецификатора.*

Для спецификатора использования внутри спецификатора контекста **With** область действия описания или переименования библиотечного модуля - полная область определения данного описания. Для спецификатора использования внутри спецификатора контекста тела библиотечного модуля область действия - все тело и любые соответствующие субмодули.

Для спецификатора использования, находящегося непосредственно внутри зоны описания, область действия - часть зоны описания, начинающейся после спецификатора использования, заканчивающейся концом зоны описания. Отсюда - область действия спецификатора

использования в приватной части не включает видимую часть любого потомка этого библиотечного модуля.

Для каждого типа T или **T'Class,** определенных в спецификаторе Use , каждая примитивная операция типа T потенциально видна в этом месте, если ее определение видно в этом месте.

D Пример.

```
with T; use  T;   - -   библиотечный пакет   T;
                        - -   его ресурсы видны    в пакете A
package A  is
   A1 , A2, A3 : integer;

end A;

procedure  Buster  is
    package  B  is
        B1, B2 : integer;
        A3      : integer;
    end   B;

procedure   ZZ   is
    A2, Z1.Z2:    integer;
    use   A;
    use   B;
begin
     - - Al   означает A . Al
     - -A2  означает ZZ.A2
     - - A3   означает A.A3
     - - Bl   означает B . Bl
     - - B2  означает  B . B2
     - - Z1   означает   ZZ . Zl
     - - Z2   означает   ZZ . Z2
  end     ZZ;

end Buster;
```

Процедура ZZ является вложенной в процедуру Buster, которая использует библиотечный пакет T и содержит вложенный пакет B.

Ш СОВЕТЫ

* Используйте приватные дочерние пакеты вместо вложенных пакетов.
» Ограничивайте видимость программных модулей с помощью размещения их внутри тел пакетов, если нельзя для этого использовать личные дочерние пакеты.
* Минимизируйте области , к которым применяется спецификатор **With.**
* Указывайте в спецификаторе" **With** только те модули, которые действительно необходимы.

14.4 Описания переименования

Описания переименования (renaming declarations) определяют новое имя для следующих понятий : обьект, подпрограмма, исключение, пакет, вход, настраиваемый модуль.

Переименование может быть использовано для разрешения конфликтов по имени и для введения сокращений. Переименование не скрывает старое имя (идентификатора или операции). Новое имя и старое имя не обязательно видимы в одном и том же месте программы.

Описание переименования обьекта:

Новое_Имя_Обьекта : **TYPE RENAMES**

Старое_Имя_Обьекта;

Описание переименования подпрограммы:

Спецификаци_Новой_Подпрограммы : **RENAMES**

Старое_Имя_Подпрограммы;

Описание переименования исключения:

Новое_Имя_Исключения : **EXCEPTION RENAMES**

Старое_Имя_Исключения;

Описание переименования пакета:

PACKAGE *Новое Имя* **RENAMES**

Старое_Имя_Пакета;

Описание переименования настройки:

Пакета :

GENERIC PACKAGE *Старое_Имя Пакета* **RENAMES;**

Процедуры : **GENERIC PROCEDURE**

Старое_Имя Процедуры **RENAMES ;**

Функции:

GENERIC FUNCTION *Старое_Имя_Функции*

RENAMES;

Описание переименования объекта определяет новый взгляд на переименованный объект, свойства которого остаются идентичными свойствам объекта, существовавшими до переименования.

При переименовании объектов проблемы могут возникать с объектами, зависящими от дискриминантов. Кроме того, переименование невозможно, если переменная имеет неограниченный тип или тип **aliased.** Отрезки массивов не могут быть переименованы, если запрещено переименование самого массива.

Задача или защищенный модуль переименовываются как объект. Единичная задача (защищенный модуль) не может быть переименована, так как имеет анонимный тип. Объекты анонимного типа массив или ссылочного типа тоже не могут быть переименованы.

При переименовании подпрограмм или входов их спецификации должны иметь одинаковый профиль параметров и результат, а также одинаковые виды параметров. D Примеры:

```
Name : Dog renames Cat;
ER434 : exception renames Fire;
package Tank renames Vector;
generic package DECCA renames Alt;
```


Различают переименование описания подпрограммы (входа) и переименования теля подпрограммы. Если подпрограмма описана в спецификации пакета, то ее определение может быть выполнено через переименование в теле пакета. Переименование требует согласования вида параметров.

```
function T(A, B : Fixed ) return Fixed renames " + ";
procedure Stub(X: integer) renames Grant;
function Wax return Matrix renames Queen;
entry Data_In( E: out Element) renames Base;
```

Процедура может быть переименована только как процедура, функция (операция) - только как функция (операция).

Вход может быть переименован как процедура, новое имя определяется в контексте, допускающем имя процедуры. Вход из семей-ства входов может быть переименован, но семейство входов не может быть переименовано целиком.

Ш СОВЕТЫ:

* Минимизируйте использование контекста **With** в спецификациях.
* Не оперируйте глобальными параметрами в подпрограммах и пакетах.
* Избегайте ненужной видимости; прячьте детали реализации программы от пользователяю
» Используйте дочерние библиотечные модули для контроля видимости как части подсистемного интерфейса.
* Используйте личные библиотечные модули для представления разных видов понятий . » Размещайте в спецификации пакета только то , что неоходимо при использовании вне пакета.
* Минимизируйте число описаний в спецификации пакета.

14.5 Примеры

П Пример 1 . Инкапсуляция задач и процедур.

```
package DOMEN is
   procedure Assa ( X: in integer; W: in float);
```

```
                                        - - видимая процедура
end    DOMEN ;

package body  DOMEN  is
   procedure  Road is              - - скрытая процедура

   end Road;
   task  Way   is                  - - скрытая задача
        entry Lion (Z : fixed ); end
   Way;

   procedure  Assa ( X:  in integer;  W : in float) is

   begin            - тело видимой процедуры

 Road;                             - - вызов скрытой процедуры

 Way. Lion (X );      - - вызов входа скрытой задачи

 end Assa;

 end    DOMEN;
```

В данном примере в спецификации пакета Domen описана процедура Assa, которая видна и доступна вне пакета пользователю пакета Domen. В теле пакета при реализации процедуры Assa используются процедура Road и задача Way . Они описаны только в теле, поэтому они не видны и недоступны непосредственно пользователю вне пакета. То есть выполнен сокрытие процедуры Road и задачи Way. Их выполнение осуществляется при вызове подпрограммы Assa , так как в теле подпрограммы Assa есть вызов процедуры Road и обращение к входу Lion задачи Way.

D Пример 2. Сокрытие деталей реализации типа

```
package   NAME _ARRAY    is
    type  Page   is   limited   private;
    procedure   Viev ( Name : in out  Page);
private          - - невидимая  часть  описания пакета
```



```
type   Data  ; type  Page  is
access end     NAME ARRAY;      Data;

package body  NAME_ARRAY          is
   type  Data    is      - - скрытая реализация типа
      record                                          Data

         X:    Page;
      end    Data;
   Book_1 ,   Book_2 :   Page;
   procedure  Viev (Name: in  out  Page)  is  separate; end
NAME_ARRAY;
```

В данном примере реализация типа Page выполненена через ссылку на тип Data , описание которого выполнено в теле пакета Name_Array . Тем самым осуществлено сокрытие деталей реализации типа Page вне пакета.

D Пример 3. Использование дочернего пакета.

```
package  PARK  is

   type  Dino    is private ;
   function  Alex   (X. Y       Dino)    return   Dino ;
   function  Bob   (X,          Dino)    return   Dino;
   function  Suzen ( X, Y       Dino)    return   Integer ;
private

end   PARK;

package     body  PARK   is

                    function   Alex(X, Y :   Dino)   return      Dino
  end  Alex;
 function  Bob   (X  is
                    : Dino)   return      Dino    is
```

```
    end  Bob;
    function  Suzen ( X, Y : Dino )  return  Integer  is

    end  Suzen;
    procedure  Port (A : in Dino : B : out Dino )      is - -
                внутрення ( скрытая )  подпрограмма

    end  Port;

end  PARK;

package  PARK. JURASSIC   is              - -  дочерний  пакет

     procedure Adventure (X: in out   Dino); end

PARK. JURASSIC ; package body  PARK.

JURASSIC  is

    procedure  Adventure(X : in out Dino)  is
       ...   - -  видна приватная часть пакета PARK end
    Adventure;

end  PARK . JURASSIC ;
```


Глава15. В В О Д - В Ы В О Д

В языке Ада отсутствуют операторы ввода-вывода. Для организации ввода-вывода различной информации пользователю предоставляются развитые средства в виде ресурсов предопределенных пакетов, являющихся дочерними корневого пакета **Ada.** Настраиваемые пакеты **DirectJO и SeqentialJO** обеспечивают операции ввода-вывода, которые применительны к файлам , содержащих элементы данного типа. Настраиваемый пакет **Storage_IO** поддерживает чтение и запись в буфер памяти. Пакеты **Text_IO и Wide_Text_IO** поддерживают дополнительные операции для ввода-вывода текста. Гетерогенный ввод-вывод обеспечивается пакетами **Streams.Stream_IO и Text_lO.Text_Stream.** В пакете **IO_Exceptions** определены исключения, связанные с вводом-выводом.

Такой подход поддерживает независимость программ и возможность их переносимости. Кроме того, обеспечивается возможность реализации дополнительных средств ввода-вывода, которые пользователь сам может реализовать через пакеты.

Указанные предопределенные пакеты обеспечивают разнообразные средства работы с файлами в виде типов, процедур и функций:

Типы для работы с файлами: **File_Type** - - лимитированный личный ти **File_Mode** - - перечисляемый тип

Процедуры

Open - -	открытие файла	**Close** - -	закрытие файла
Create - •	создание файла	**Delete** - -	уничтожение файла
Read Put	чтение файла	**Write** - -	запись в файл
Reset Set	запись в файл	**Get**	- чтение файла
Index	восстановление файла для чтения записи - - установка текущего индекса		

Функции работы с файлами :

Name - - определение имени внешнего файл
End_Of_File -- проверка конца файла
ls_Open -- проверка состояния файла
Mode - - проверка текущего вида файла
Form -- форма внешнего файла

Index	текущий индекс
Size	размер внешнего файла

Существуют два вида доступа к внешним файлам - *последовательный* доступ и *прямой* доступ. Соответствующие типы файлов и связанные с ними операции описаны в настраиваемых пакетах SeqentialJO и DirectJO. Файловый объект, используемый для последовательного доступа, называется *последовательным файлом* (sequential file) или *файлом прямого доступа.* Файловый объект для прямого доступа называется *прямым файлом* (direct file) или файлом прямого доступа. В языке также используются потоковые файлы (stream file).

15.1 Пакет SeqentialJO

Предопределенный настраиваемый пакет **SeqentialJO** обеспечивает файловые типы и операции ввода-вывода последовательных файлов. При последовательном доступе файл рассматривается как последовательность значений, предаваемых в порядке их поступления. При открытии файла передача начинается с начала файла.

Описание настройки и спецификации пакета **SeqentialJO :**

```
with Ada.lOJExceptions;
generic
   type Elemet_Type (<>) is   private;

   package  Ada. SeqentialJO  is

   type File_Type  is  limited  private; type  FileJVIode is

   (ln_File,  Oirt_File,  Append_File); --  управление файлами
   procedure Create (File Mode  in  out   File_Type;
                     Name Form  in  in    FileJVIode := OutJ=ile;
                                in  String := ing
   procedure  Open ( File Mode  in  out   File_Jype;
                     Name       in  out   FileJVIode;
                     Form       in  in    String; String
                                          :="");
```



```
   procedure  Close  ( File :  in   out   File_Type);   out
   procedure  Delete ( File :  in   File_Type);         out
   procedure  Reset  ( File :  in   File_TypeJVIode;
                       File :  in   FileJVIode); out
   procedure  Reset (  File :  in   File_Type);

   function  Mode ( File  :  in   File_Type)  return  FileJVIode;
   function  Name (File   :  in   File_Type)  return  String;
   function  Form ( File  :  in   File_Type) return   String;

   function IsJDpen ( File  :  in   File_Type)  return  Boolean;
```

Операции ввода-вывода procedure Read (File : in File_Type ; Item :out Element_Type); procedure Write(File : in File_Type; Item: in Element_Type);

```
   function EndJDfJ=ile( File : in  FileJType)  return  Boolean;

-.-  Исключения
   StatusJError : exception renames   IO_Exceptions.Status_Error;
   ModeJError   : exception renames   IOJExceptions.ModeJError;
   NameJError : exception renames     IOJ5xceptions.Name_Error;
   UseJError    : exception renames   IO_Exceptions.UseJ=rror;
   DeviceJError: exception renames    IO_Exceptions.DeviceJ=rror;
   EndJError    : exception  renames  IOJ=xceptions.EndJError;
   DataJError   : exception  renames  IOJ=xceptions.DataJ=rror;

   private
      ... - -  Не определены в языке end
   Ada. SeqentialJO;
```

Операции над последовательными файлами выполняются с помощью подпрограмм **Read, Write и EndJDf_File.**

Procedure Read (File : in File_Type; Item : out Element_Type); Читает элемент файла типа InJFile и вызывает его значение через параметр Item.

Procedure Write (File: in File_Type; Item : in ElementJType);
Записывает в файл типа Out_File значение параметра Item.
Function End_Of_File (File : in File_Type);
Если достигнут конец файла, то возвращает значение **True** , иначе - **False.**

15.2 Пакет DirecMO

Пакет обеспечивает ввод-вывод для файлов прямого доступа. При прямом доступе файл рассматривается как набор элементов, занимающих последовательные позиции в линейном порядке. Позиция элемента задается индксом (тип **Count).** Открытый прямой файл имеет *текущий* индекс.

Описание настройки пакета **DirectJO :**

```
with Ada.lO_Exceptions;
generic
   type Element_Type is private;
package Ada . DirectJO is

   type File_Type is limited private;

   type File_Mode is (In_File, Inout_File, Out_File);
   type Count is range 0. . - -определяется реализацией
   subtype Positive_Count is Count range 1.. Count'Last;

- - Управление файлами

   procedure Create (File : in out File_Type;
                     Mode: in File_Mode : = Inout_File;
                     Name: in String : = "";
                     Form : in String : = ""
      procedure Open (File ) ;
                      Mode in
                      Name in
                      Form in
                           in
      procedure Close (File
                           in out File_Type );
```



```
procedure Delete ( File : in out File_Type ) ;

procedure Resest ( File : in out File_Type ;
Mode                    : in File Mode ) :
procedure Resest ( File : in out FileJType);

function Mode    ( File : in File_Type )
                          return File_Mode;
function Name    ( File : in File_Type )
                          return String;
function Form    ( File : in File_Type )

                    in    File_Type)
                          return Boolean;
                          return String;

function Is_Open ( File     : - -

   Операции ввода - вывода

procedure Read ( File in                          File_T
                 Item out                         Eleme
                 From in                          Positi
procedure Read ( File in                          File_T
                  Item out                        Elem

procedure Write (File in                          File_Typ
                 Item in                          Elemen
                 To in                            Postive_
procedure Write (File in                          File_Typ
                  Item in                         E!emen

   procedure Set_Index( File : in FileJType;
                        To : in Positive_Count ) ;

   function Index ( File : in File_Type)
                          return Positive_Count;
   function Size ( File : in File_Type) return Count;

   function End_Of_File (File : in File_Type)
                          return Boolean;
```

```
- - Исключения

    Status_Error:  exception   renames
                                IO_Exceptions.Status_Error;
    ModeJError:    exception   renames
                                IO_Exceptions. ModeJError;
    Name_Error:    exception   renames
                                IO_Exceptions.Name_Error;
    Use_Error :    exception   renames
                                IO_Exceptions.Use_Error;
    Device_Error: exception    renames
                                IO_Exceptions. Device_Error;
    End_Error :    exception   renames
                                lOJExceptions.EndJError;
    Data_Error:    exception   renames
                                IO_Exceptions. Data_Error;

    private
      ...     - - Не определены в языке end
    Ada . DirectJO ;
```

Операции над прямымим файлами выполняются с помощью подпрограмм **Read, Write, Setjndex, Index, Size, End_Of_File.**

Рассмотрим подробно представленные подпрограммы.

```
procedure  Read( File : in    File_Type; Item : out   ElementJType;
                                          From : in Positive_Count);
procedure Read ( File : in    FileJType ;Item : out    ElementJType);
```

Первая процедура сначала устанавливает текущий индекс (параметр From), а затем через параметр Item возвращает значение текущего элемента и увеличивает текущий индекс на единицу.

```
procedure. Write ( File : in FileJType; Item : in ElememtJType;
                                          To: in Positive_Count);
procedure  Write ( File : in Fi!e_Type;   Item : in ElementJType);
```


Первая процедура вначала устанавливает индекс файла (параметр To), а затем текущему элементу присваивается значение Item и текущий индекс увеличивается на единицу.

```
procedure Set_Index( File : in   FileJType;
                                          To: out  Positive_Count);
```

Устанавливает текущий индекс данного файла через параметр To.

```
function Index ( File : in   File_type)  return Positive_Count;
```

Возвращает текущий индекс файла. **function Size** (File :

```
in   FileJType)  return Count;
```

Возвращает текущий размер внешнего файла, связанного с данным файлом.

```
function End_Of_File ( File:   in   FileJType) return Boolean;
```

Если значение текущего индекса больше размера внешнего файла, то возвращает значение **True** , иначе **False.**

15.3 Пакет StorageJO

Настраиваемый пакет **StorageJO** обеспечивает средства для чтения и записи в буфер памяти. Пакет поддерживает конструкции, определяемые пользователем в пользовательских пакетах ввода-вывода.

Спецификация пакета **Storage_IO:**

```
with Ada.lO_Exception;
with System.Storage_Elements;
generic

    type   ElementJType   is    private;

package    Ada. StorageJO   is

    pragma   Preelaborate (StorageJO ); Buffer_Size   : constant

System.StorageJElements. Storage
```

```
                                    _Count : =  - - определяется реализацией
   subtype    BuffeMType   is
              System.Storage_Elements.Storage_Array (1 .. Buffer_Size);

          Операции  ввода-вывода procedure
   Read ( Buffer   :   in Buffer_Type;
                                          Item :   out Element_Type);

   procedure Write( Buffer : out   Buffer_Type;

                                          Item : in      Element_Type);

    - -  Исключения

   Data_Error  : exception   renames
                                          IO_Exceptions. Data_Error;

end   Ada. StorageJO;
```

В каждом экземпляре этого пакета, получаемом после конкретизации, константа BufFer_Size имеет значение, которое определяет размер (в элементах памяти) буфера, необходимого для представления объекта подтипа

Процедуры **Read и Write** из пакета **StorageJO** соответствуют процедурам **Read и Write** из пакета **DirectJO,** однако содержимое параметра Item читается (считывается) из указанного буфера вместо внешнего файла.

15. 4 Пакет Text_IO

Пакет **Text_IO** используется для ввода-вывода текстовых файлов в форме, удобной для пользователя. Спецификация пакета приведена в Приложении 3.

Для использования ресурсов пакета в программном модуле пользователя ему должен предшествовать *спецификатор контекста:*

```
with   Ada . TextJO;
```

Ввод-вывод значений соответствующих типов текстовых файлов осуществляется через процедуры Put и Get. Пакет **TextJO** содержит в свою очередь настраиваемые пакеты **IntegeMO,**

Float_IO и Fixed_IO для ввода-вывода соответственно целых, плавающих и фиксированных типов.

15.4.1 Ввод-вывод целых типов

Процедуры ввода-вывода для целых типов определены в настраиваемом пакете **IntegeMO,** который находится в пакете **Text_IO.** Пакет **IntegeMO** имеет один параметр настройки:

```
type   NUM   is   range <>,
```

который задаётся через соответствующий целый тип при конкретизации. Например:

```
package  Io_Int_Short   is   new   IntegeMO ( shortjnteger);
package  Pos            is    new  IntegeMO ( positive );
package  INPUT66        is    new  IntegeMO ( integer);
```

Значения целых типов выводятся в виде десятичных литералов или литералов по основанию без подчёркивания и порядка. Значение основания принадлежит целому подтипу:

```
subtype   Number_Base  is integer range  2 .. 16;
```

При выводе в процедурах могут использоваться параметры ширины поля и основания, которые по умолчанию задаются в пакете **IntegeMO** переменными:

```
Default_Width                Field Number   :=  Num'Width; :=
Default Base                 Base               10;
```

Для ввода определены следующие процедуры:

```
procedure   Get (
                                File     | in    FlileJType;
                                Item     | out   Num;
procedure   Get (               Width    | in    Field: = 0);
                                File     | in    Field_Type;
Item    | out   Num;
Width   | in    Field : = 0 ) ;
```

ja определены следующие процедуры:

```
procedure   Put (   File      in   File_Type;
                    Item      in   Num;
                    Width      in  Field := Default_Width ;
```

```
                    Base    :   in  Number_Base : =
                                                Default_Base);
procedure   Put (   Item    :   in  Num;
                    Width   :   in  Field := DefaultJ/Vidth ; Base  :
                    in  Number_Base ; =
                                                Default_Base);
```

Значение параметра Item выводится в виде целого литерала без подчёркивания, порядка и ведущих нулей и со значением знака минус, если значение отрицательное. Если в выводимой последовательности число символов меньше значения параметра Width , она дополняется слева пробелами.

Если параметр Base равен десяти (по умолчанию он также равен десяти), то число выводится в соответствии с синтаксисом десятичного литерала, в противном случае - в соответствии с синтаксисом литерала по основанию.

П Пример:

```
package  IOJNT99  is  new  TextJO.IntegerJO (integer); use
  IOJNT99;
          по умолчанию при выводе используются параметры
     --   Default_Width  =4
     --   Default Base  = 1 0

  X  :=  121 ;
  put ( X );                    выводится   "ЬЫ21"
  put(-X,  6);                  выводится    "bb-121" 2 ); - -
  put(X,  Width  => 12,  Base   выводится
                                             "bb2#1111001#"
```

15.4.2 Ввод-вывод вещественных типов

Процедуры ввода-вывода вещественных типов определены в настраиваемых пакетах **Float_IO и FixedJO,** которые находятся в пакете **Text_IO.** Пакет **Float_IO** имеет параметр настройки:

```
type  NUM  is  digits <> ;
```

который задаётся через соответствующий тип с плавающей запятой при конкретизации.

Пакет **Fixed_IO** имеет параметр настройки:

```
type  NUM  is  delta <>,
```

который задаётся через соответствующий тип с фиксированной запятой при конкретизации. П Например:

```
type  FLT  is  digits  8;    - - описания типов type
Data  is  delta  0.001  range  -15.0  .                  20.0;

package  FLTJO  is  new  FloatJO ( FLT );  - -
конкретизация package  INPUT_DATA  is  new  FixedJO ( Data);
```

Вывод значений вещественных типов осуществляется как вывод десятичных литералов без подчёркивания. Формат вывода определяется полями Fore (целая часть), Aft (дробная часть), Exp (порядок) с добавлением десятичной точки и буквы E в случае необходимости: Fore. Aft
Fore . Aft E Exp

Значения этих полей по умолчанию задаются в спецификациях соответствующих пакетов.

Get:

```
procedure  Get(   File    :  in     File_Type;
                  Item    :  out    Num ;
                  Width   :  in     Field : =0);
procedure  Get(   Item    :  out    Num;
                  Width   :  in     Field : = 0 ) ;
```

Для вывода вещественных типов определены процедуры Put:

```
procedure  Put(   File    |  in     File_Type ;
                  Item    |  in     Num;
                  Fore    |  in     Field   |  =  Default  Fore ;
                  Aft     |  in     Field   |  =  Default_Aft;
                  Exp     |  in     Field   |  =  Default_Exp )
procedure  Put(   Item    |  in     Num ;
                  Fore    |  in     Field   |  =  Default Fore ;
                  Aft     |  in     Field   |  =  Default  AFft:
                  Exp     |  in     Field   |  =  Default_Exp )
```

Значения параметра Item выводится в виде десятичного литерала в формате, определяемом параметрами Fore , Aft , Exp .

Если целая часть с учётом знака минус содержит менее Fore символов, то в начале выводятся пробелы.

D Пример:

```
package ZZE is new Float IO ( real); - - конкретизация
use ZZE;
X : = -121.4287;
put(X); put(X, - - выводится "b
5,3, Ob-put (X, 1.214287E+02"
FORE --выводится "b-121.428"
AFT => 3, EXP 5,
2 ) ; --выводится
"bbbl.214E+2"
```

15.4.3 Ввод-вывод перечисляемых типов

Процедуры ввода-вывода для перечисляемых типов определены в настраиваемом пакете **EnumerationJO,** который при конкретизации настраивается на перечисляемый тип (Enum).

D Например :

```
type Color is ( blue, green, gray); package COLJN
is new EnumerationJO ( Color);
```

При выводе значений используются либо строчные, либо прописные буквы. Формат вывода (вместе с заключительными пробелами) задаётся необязательными параметрами ширины поля. По умолчанию эти параметры:

```
Default_Width : Field := 0;
Default_Setting : Type_Setting := Upper_Case;
```

Для ввода перечисляемых типов определены процедуры **Get:**

```
procedure Get( File : in File_Type; Item : out Enum);
procedure . Get ( Item : out Enum);
```

Для вывода перечисляемых типов определены процедуры **Put:**

```
procedure Put( File in File_Type;
Item in Enum;
Width in Field : = Default Width;
Set : in Type_Set : = Default_Setting );
procedure Put ( Item : in Enum ;
Width : in Field : = Default_Widrth ;
Set : in Type_Set: = Default_Setting );
```


Значение параметра Item выводится как литерал перечисления (либо идентификатор, либо символьный литерал). Если число выводимых символов меньше, чем значение параметра Width , то в конце выводятся пробелы, дополняющие число символов до Width .

Так как тип **Boolean** является перечисляемым типом, то настраиваемый пакет **EnumerationJO** может быть использован после конкретизации для ввода-вывода элементов типа **Boolean.**

15.5 Исключения при вводе-выводе

При выполнении операций ввода-вывода могут быть возбуждены предопределённые исключения. Они описаны в пакете **lOJExceptions.** Спецификация **пакета lOJExceptions:**

```
package lOJExceptions is
STATUSJERROR : exception;
MODEJERROR exception;
NAMEJERROR exception;
USEJERROR exception;
DEVICEJERROR exception;
ENDJERROR exception;
DATAJERROR exception;
LAYOUT ERROR exception;
end
lOJExceptions ;
```

Исключение Status Error возбуждается при попытке выполнить действие над еще не открытым файлом или при попытке открыть уже открытый файл.

Исключение Mode_Error возбуждается при попытке чтения или проверки конца файла вида Out_File , а также при чтении записи в файл вида In_File.

Исключение Name_Error возбуждается при попытке вызова процедур Create и Open , если строка, заданная параметром Name , не позволяет идентифицировать внешний файл.

Исключение Use_Error возбуждается при попытке выполнить операцию, не разрешенную по причинам, зависящими от характеристик внешнего файла.

Исключение Device_Error возбуждается при невозможности завершения операции вво-вывода из-за неисправности устройств.

Исключение End_Error возбуждается при попытке пропустить признак конца файла.

Исключение Data_Error возбуждается в процедурах Read ,если вводимая последовательность не соответствует требуемому типу, а в процедурах Get - не соответствует синтаксису или не принадлежит диапазону типа.

Исключение Layout_Error возбуждается в текстовом вводу-выводе при вызове функций Col, Line, Page , если возвращаемое зна чение превышает Count'Last; в процедуре Put - при попытке вывести большое количество символов в строку.

Ш СОВЕТЫ:

* Всегда закрывайте все открытые файлы после окончания их использования.
* Используйте константы и переменные в качестве формальных параметров Name и Form в предопределенных пакетах ввода-вывода.
* Избегайте ввод-вывода ссылочных типов.
* Используйте пакеты **SequentialJO и DirectJO** вместо **StrearrMO** при организации гетерогенного ввода-вывода низкого уровня.

15.6 Примеры

П Пример 1. Создание файла прямого доступа.


```
With Ada.DirectJO ;
procedure   EXC15_1  is

   type  Post  is
      record
         X   : string ( 1 .. 64 ); end
      record ;

   - - Конкретизация пакета
   package   WorkJO  is  new
                              Ada.DirectJO ( Element_Type = > Post);
   use   WorkJO;

   ZZ    : File_Type ; - - используется тип из предопределенного
                                 - - пакета
   Ident: string (1 .. 12) : =  "File_15_3"; - - идентификатор
                                                          - - файла
begin

    Create ( File = > ZZ;    Mode = > Inout_File ;
                                                 Name = > Ident;   Form = >"");

    Close (ZZ); end

EXC15 1;
```

Глава 16. ОБЪЕКТНО - ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Язык Ада поддерживает современную практику разработки программного обеспечениия, основанную на использовании обьектно-ориентированых технологий.

Ада 95 - первый язык обьектно-орентированного программирования, прошедший полную международную сертификацию. Наличие законченных средств обьектно-ориентированного программирования является отличительной чертой языка. Ада 83 поддерживала методику обьектно-ориентированного проектирования, частично ре-ализуя парадигмы обьектно-ориентированного программирования с помощью абстракции данных через приватные типы в пакетах.

Ада 95 уже полностью реализует парадигмы обьектно-ориентированного программирования:

- *инкапсуляцию* (encapsulation)
- *наследование* (inheritance)
- *полиморфизм (* polymorphism);

Обьектно-ориентированное программирование в языке базируется на следующих основных понятиях:

- *Тип* - задает множество значений и множество операций;
- *Объект* - константа или переменная, ассоциируемая с типом и имеющая структуру и состояние;
- *Класс* - множество типов, закрытое лри порождении, то есть если данный тип является членом класса, то все типы, производные от данного типа, тоже являются членом класса ; множество типов одного класса имеют общие свойства (например, общие примитивные операции);
- *Инкапсуляция* - средство, обеспечивающее группирование объектов и их операций, сокрытие деталей их реализации, а также наличие абстрактного интерфейса к ним;
- *Наследование* - возможность определения новых абстракций на основе уже существующих с наследованием их свойств и добавлением новых;

- *Полиморфизм* - возможность умножения различий между абстракциями, при которых программа может быть написана в терминах их общих свойств:

Наряду с полной поддержкой указанных понятий, в Аде 95 реализованы дополнительно средства, обеспечивающих более эффективное использование преимуществ обьектно-ориентированных технологий. Это:

- *Смешанное наследование (* Mixin Inheritance) , использование которого позволяет вводить разновидности родительских абстракций, используемых только для обеспечения свойств производным от них абстракциям.
- *Множественное наследование (* Multiple Inheritance), при использовании которого возможно наследование компонент и операций от нескольких родительских типов.
- *Диспетчеризация (* Dispatching) - средство, обеспечивающее динамический выбор реализации соответствующей абстракции. Реализация этих понятий и средств в языке основана на введении

дополнительных типов : *тэговых, расширяемых и абстрактных.*

16.1 Тэговые типы

Тэговые типы **(tagged types)** - это запись или приватный тип, помеченный словом **tagged.**

TYPE *Имя* **IS [ABSTRACT] TAGGED**
(*Запись* | **PRIVATE);**

```
type Plan is tagged
   record
      X : fixed:
      Y : fixed:
   end record;
```

```
type Green is tagged private; type

Data is tagged null record;
```

```
package RESOURCE is

   type Control is tagged private; private
   type Control is tagged
      record
      Item : integer; Numbe
      : natural; end record;

end RESOURSE;
```

С каждым объектом тэгового типа связан *тэг,* указывающий на особенности типа. Для создаваемого объекта тэгового типа его тэг никогда не изменяется. Тип, помеченный как тэговый, может быть описан в спецификации пакета и для него могут быть объявлены новые примитивные операции.

Особенностью тэгового типа является возможность добавления к нему новых компонент при построении на его основе производных типов. Этот процесс называется *расширением типа.*

Тип, производный от тэгового типы, наследует все его свойства - множество значений и множество примитивных операций. В этот тип можно добавлять новые компоненты. Кроме того, можно объявлять дополнительные операции с помощью подпрограмм, а также выполнять *замещение* наследованных от родительского типа операций. В языке для работы с тэговыми типами определен специальный пакет

```
package Ada . Tags is

   type Tag is private ;
   function Expanded_Name (T : Tag ) return String ;

   function External_Tag (T: Tag) return String ; function
   Internal_Tag ( External: String ) return Tag ; Tag_Error:
   exception ;

private
   . . . - - не определено в языке end Ada.
Tags ;
```


Функция Expanded_Name возвращает имя первого подтипа специфического типа, идентифицированного с помощью тэга Т.

Функция External_Tag возвращает символьную строку, являющуюся внешним представленим данного тэга.

Функция lnternal_Tag возвращает тэг, который соответствует данному внешнему представлению тэга, или возбуждает исключение Tag_Error в случае, если аргумент функции не является внешним представлением тэга.

СОВЕТЫ :

Определяйте один тэговый тип в пакете.

Используйте тэговый тип для сохранения общего интерфейса для различных реализаций абстракции.

16.2 Расширение типа

Основная идея программирования, основанного на расширении - возможность обьявления нового типа, который, используя существующий родительский тип, наследует, модифицирует и добавляет к его компонентам и операциям новые компоненты и операции. Такой подход сокращает время разработки программы, так как не требует перекомпиляции существующей системы.

Расширения типа (type extensions) в Аде 95 основывается на существующей в Аде 83 концепции производных типов: новый тип может быть получен из существующего (родительского) как производный тип. Производный тип наследует операции родительского типа и мы могли добавлять новые операции . Однако в Аде 83 не разрешалось добавлять новые компоненты к производному типу, то есть механизм был статическим.

В Аде 95 производный тип может быть расширен новыми компонентами, если он помечен как тэговый. Общий вид расширения типа:

TYPE *Имя* **IS [ABSTRACT]** NEW
Имя_Родителъского_Типа **WITH (*Запись* \ PRIVATE) ;**

Здесь *Имя* - расширенный тип, *Имя_Родительского_Типа* - тэговый тип.

Производный от тэгового тип называется *раширенным типом (* type extended). Каждое расширение типа порождает в свою очередь тэговый тип и может служить основой для последующего расширения.

П Пример расширения типов:

```
type  Place  is  new  Plan  with  null  record;
                                    - - без добавления компонент

type  Res_Data  is  access  all  Data'Class;

type  Ex_Plan  is  new  Plan  with
   record
       Z:  fixed:
   end  record;
```

```
type  Elements  is  new  Data  with record
    V :  Vector; M :
    Matrix; N :
    integer; end
    record;

 package  RAM  is
     type  Tend  is  new  Control  with  private

 private
     type  Tend is  new  Control  with record
    Z :  Matrix ;  - -  новые компоненты end
record; end  RAM;
```

СОВЕТЫ:

* Определяйте расширение тэговых типов в дочерних пакетах.

16.3 Типы широкого класса.

Объединение тэговых типов и его потомков (типов, производных от тэгового типа) формирует производные классы тэговых типов.

Особенности тэговых типов наделяют эти классы уникальными свойствами, которые отличают их от других производных классов. Для таких классов можно создавать операции, применимые ко всем типам класса; использовать различные реализации для одной и той же примитивной операции для разных типов в классе. Наконец, класс может быть использован как основа для реализации множественного наследования.

Для каждого тэгового типа T в языке автоматически определяется соответствующий тип T 'Class, называемый *типом широкого класса (* class-wide type). Тип T 'Class объединяет все типы, производные от типа T, и является неограниченным, так как всегда возможно его расширение.

Тип широкого класса определяется для каждого класса производных типов, корнем которого является тэговый тип. Множество значений типа T 'Class - это имеющее дискриминант объединение множества значений каждого специфического типа в производном классе, корнем которого является тип T. В качестве дискриминанта здесь выступает тэг каждого тэгового типа из класса. Каждый объект типа T 'Class имеет:

- тэг, который отличает его от всех других типов класса;
- значение, тип которого определяется соответствующим тэгом.

Идентификация с использование тэга может выполняться динамически, во время выполнения программы.

Для каждого подтипа S тэгового типа T определены следующие атрибуты:

- S ' Class - подтип класса T'Class .
- S ' Tag - означает тэг подтипа S.
- T ' Tag - тэг корневого типа класса.

D Например:

```
package AVTO  is

    type  Cars  is tagged  - -  корневой тип  класса Cars record

        end  record ; procedure  Motor (X : in
    out  Cars);
```

```
type Ford is new Cars with  - - расширение типа  Cars
   record

    end record ;
procedure Motor (X : in  out  Ford );
procedure Cord (X :      out  Ford );

type Opel  is new Cars with  - - расширение типа  Cars
   record

   end record;
procedure  Motor  ( X : in  out  Opel);
procedure Glass  ( Y :      out  Opel)

- - расширения типа Opel

type Cadet  is  new  Opel  with  . . .  end  record; type Vectra  is

new  Opel  with  . . .  end record; end  AVTO ;
```

Дерево иерархии производных классов, основанных на типе Cars в пакете AVTO:

```
                    Cars ' Class
                                  Opel' Class
                                              Vectra' Class

Ford' Class                 Cadet' Class
```


Тип Cadet не является подтипом Opel, это разные типы и при совместном использовании они требуют явного преобразования. Класс Cars ' Class объединяет все типы, основанные на тэговом типе Cars : Ford, Opel, Cadet, Vectra. Подкласс Opel 'Class имеет в качестве корневого типа Opel и объединяет типы Cadet и Vectra.

ffl СОВЕТЫ:

* Используйте операции типа широкого класса в случаях, когда не известны все возможные потомки для данного тэгового типа.
* Используйте операции типа широкого типа, когда желательно избежать наследования или совмещения. » Используйте тип широкого класса для реализации динамического полиморфизма.
* Используйте тип широкого класса вместо переменных записей.
* Избегайте использование расширенных типов для параметризации абстракций.
* Используйте тип широкого класса в качестве интефейса для набора тэговых типов, образующих класс.

16.4 Операции над тэговыми типами

После расширения типа возможно добавление к нему новых примитивных операций с помощью новых подпрограмм или совмещения с уже существующими.

В пакете AVTO для расширенных типов Ford и Opel определены новые операции,

```
procedure Glass (Y : out  Opel);
procedure Cord (X : out  Ford );
```

Кроме того, в пакете определена операция Motor , которая является совмещенной для трех типов.

В языке допускаются определения операций с одинаковыми именами (совмещение операций), при которых операции имеют одинаковые имена, но разную реализацию (разные тела). Например, операции Get определенные в предопределенном пакете Text_IO являются совмещенными:

```
procedure  Get (Item :  out  Character);
```

```
procedure Get (Item : out Enum );
```

При вызове совмещенной операции ее конкретная реализация выбирается на этапе компиляции (статически) на основании типа аргумента или результата. Правда это возможно только в. том случае, когда эти типы известны заранее.

Если типы, используемые в совмещенных операциях, формируются в процессе выполнения программы, то выбор реализации совмещенной операции должен выполняться динамически , то есть при выполнении программы; Такой выбор в языке осуществляется при помощи *диспетчеризации* (dispatching).

В языке для работы с тэговыми типами используются примитивные подпрограммы, называемые *диспетчерскими операциями (* dispatching operations). В общем случае диспетчерская операция обеспечивает при выполнении программы выбор соответствующей реализации операции, если тип аргументов этой операции не может быть определен до выполнения программы, то есть тип может быть разным при разных вызовах этой операции. Такая ситуация возникает при совмещении операций.

Таким образом, язык обеспечивает две возможности реализации операции посылки:

- статическую при компиляции
- динамическую при выполнении программы.

Вызов диспетчерской операции - это вызов Примитивной подпрограммы, аргументы которой имеют тэговый тип. При вызове такой подпрограммы используется фактический параметр тэгового типа (или типа широкого класса). Выбор тела для примитивной подпрограммы будет осуществлен на основе тэга фактического параметра. Это происходит во время выполнения программы, то есть имеет место динамический полиморфизм. П Пример :

```
procedure Bond ( X in integer);
procedure Bond (X in Data);
procedure Bond {X tn Color' Class ) ;
```

Операция Bond является совмещенной для формальных паря мет ров трех типов. Вызов в программе процедуры Bond связан с выбором одной реализации ее тела, которая выполняется на основании типа ее параметра X. Для первых двух версий выбор осуществляется

статически - при компиляции, а для третьей - динамически при выполнении программы.

Диспетчерский вызов может использовать тело, описание которого невидимо в месте вызова.

Тип широкого класса T'Class не имеет собственных примитивных операций и для него они определяются пользователем. Операции широкого класса применимы к обьектам любого типа внутри класса T.

П Например :

```
                    in out Cars ' Class ) is
procedure Control (X begin
   Motor ( X );
end Cars;
```

Процедура Control имеет аргумент типа широкого класса. В теле процедуры используется подпрограмма Motor, которая в пакете AVTO определена для трех типов, входящих в класс Cars'Class : Ford , Opel , Cars. Вызов процедуры Control выполняется с подстановкой фактического параметра, имющего тип Cars'Class. Выбор (одной из трех) процедур Motor для выполнения осуществляется динамически на основании тэга фактического параметра. Это пример диспетчерской операции (операции посылки) , характеризующей особенности выполнения операций над обьектами широкого типа.

СОВЕТЫ:

* Избегайте исключений в совмещенных операциях, которые неизвестны пользователю.

16.5 Абстрактные типы и подпрограммы

Механизм абстрактных типов и абстрактных подпрограмм обеспечивает в языке реализацию парадигм обьектно-ориентированного программирования, связанных со смешанным и множественным наследованием.

Абстрактный тип задает основу для создания типов с общими свойствами. Абстрактные процедуры - для определения примитивных операций для абстрактных типов.

Абстрактный тип - это тэговый тип, используемый как родительский при расширениях типа, однако он не может иметь обьектов (нельзя обьявить обьект абстрактного типа).

Абстрактная подпрограмма - подпрограмма без тела, однако подразумевается, что такая подпрограмма является совмещенной в том месте, где она наследуется. Вызов с помощью диспетчерской операции абстрактной подпрограммы всегда связан с некоторым телом совмещенной подпрограммы, поскольку обьект абстрактного типа не может быть создан.

Абстрактная подпрограмма предназначена для абстрактных тэ-говых типов. Она не может быть вызвана прямо или косвенно.

Абстрактный тип описывается как тэговый тип, абстрактная подпрограмма описывается как обычная подпрограмма , при этом добавляется ключевое слова **abstract.**

```
type  Monstr is abstract  tagged  null  record;

function  Demon  return  Monstr  is  abstract;
procedure  Unit (X , Y : In integer;  Z ;
                                  in  out  Monstr)  is  abstract;
```

Для типов, производных от абстрактного типа, следует обеспечить фактические подпрограммы, необходимые для реализации абстрактных подпрограмм родительского типа.

П Например:

```
package  SET  is

    type Oil is abstract  tagged  null  record;
    procedure Work (X:  in  out  Oil)  is  abstract;

end  SET;
```

Процедура Work определена для абстрактного типа Oil ; она не имее тела.

Пакет SET с его абстрактными ресурсами можно использовать при разработке нового пакета :


```
with  SET;
package  LIST  is

    type Ointment  is  new  SET. Oil  with record
        P:  real; end  record ; procedure  Work (O :  in  out  Ointment);  -
    -  совмещенная
                                                          - -  процедура
    procedure  Olive (O :  in  Ointment);

end  LIST ;
```

В пакете LIST определен тип Ointment как расширение абстрактного типа Oil с добавлением компоненты P. Для типа Ointment определены две операции - подпрограмма Work и Olive. Процедура Olive не является абстрактной , процедура Work является совмещенной и определяет фактическую подпрограмму для абстрактной подпрограммы Work из пакета SET.

В теле пакета LIST выполнена реализация процедуры Work, то есть находится ее тело.:

```
package body  LIST  is

    procedure  Work (O :  in  out  Ointment)  is begin

    O: = Olive ( O ) ;

end Work ; end  LIST;
```

Таким образом, для тэговых типов и его потомков определены три вида операций:

- примитивные неабстрактные;
- примитивные абстрактные;
- операции широкого класса.

Неабстрактные операции должны переопределяться для каждого подкласса.

Абстрактные операции должны быть совмещенными с операциями для неабстрактных производных типов.

Операции широкого класса не могут совмещенными при определении подкласса. Они могут быть переопределены для производного класса, являющимся корневым в производном типе. В этом случае создаваемые абстракции сохраняют свойства широкого класса.

Если новый тип произведен от тэгового, то он наследует его примитивные операции. Если необходимо не наследовать некоторые операции, то их следует рассматривать либо как операции широкого класса, либо описывать их в отдельном дочернем пакете.

Эффект придания операциям абстрактности гарантирует, что каждый потомок должен опеределять свою версию примитивной операции. Эти операции описываются в пакете вместе с тэговым типом перед определением производных типов. Новый тип в этом случае наследует примитивные операции родительского типа.

СОВЕТЫ:

* Используйте абстрактные типы в качестве формальных типов в настраиваемых модулях.
* Используйте абстрактные типы для различных реализаций одной абстракции.
* Определяйте корневой тип класса как абстрактный.

16.6 Множественная реализация и множественное наследование.

Множественная реализация абстракции - важная составляющая обьектно-ориентированного программирования. В Аде 83 она обеспечивалась тем, что пакет мог иметь несколько альтернативных тел. Однако при этом только одна реализация могла использоваться в программе.

В Аде 95 множественная реализация основывается на использовании тэговых типов. При множественной реализации абстракции тэг, связанный с объектом, позволяет динамически выбирать соответствующую реализацию.


```
package ABSTRACT_DATA is

   type Data is abstract tagged private;
   procedure Zond ( X : in Data ; Y : out Data ) is abstract;
   function Fast (X: Data) return Data is abstract;

private
   type Data is abstract tagged null record;

end ABSTRACTJDATA ;
```

Пакет ABSTRACT_DATA обеспечивает абстрактную спецификацию набора данных. Тип Data описан как абстрактный тэговый приватный тип. Абстрактные подпрограммы Zond и Fast определяют примитивные операции для типа Data. При этом они не имеют тел и не могут быть непосредственно вызваны.

На основании типа Data может быть выполнено расширение типа с добавлением компонент и совмещенными операциями для абстрактных процедур Zond и Fast.

Реализация абстракций, определенных в пакете ABSTRACT_ DATA , выполняется в пакете RR_DATA.

```
with ABSTRACT_DATA; use ABSTRACT DATA;
package RR_DATA is

   type RR is new Data with private ;
   procedure Zond ( X : in RR ; Y : out RR) is abstract;
   function Fast ( X : RR) return RR is abstract;

private
   type RR is new Data with
      record
         N : integer;
      end record ;

end RR DATA;
```

Тело пакета RR Data :

```
package body RR_DATA is
```

```
   procedure Zond (X : in    RR ; Y :   out RR )   is begin

   end Zond;
   function    Fast (X :   RR )   return RR   is begin

   end Fast; end

RR_DATA;
```

Для реализации абстракций из пакета ABSTRACT_DATA можно воспользоваться также типом широкого класса, базирующимся на типе Data. Например, содав процедуру, аргументы которой имеют тип Data'Class.

```
procedure  Action (X : in   Data'Class;   Y: out   Data'Class)  is
    Demo:   Data'Class ;
begin

   Demo : = Fast (X);
   Zond ( Demo , Y);

end   Action ;
```

Вызов процедуры Action является диспетчерским вызовом, то есть ее выполнение зависит от тэга фактического параметра.

Множественное наследование позволяет производному типу (или классу) иметь несколько родителей. Реализация множественного наследования в языке обеспечивается механизмом библиотечных модулей, использованием приватных расширений, а также с помощью ссылочных дискриминантов.

П Пример:

```
package  ABSTRACT_SET   is

   type Set  is  abstract  tagged   limited  private ;

   procedure Rembo ( X :   in   out Set)              is  abstract;
   function    Rolex  ( Y : Set)  return    integer is  abstract;

private
   type Set is abstract tagged limited  with   null record ;

end   ABSTRACT_SET ;
```

В спецификации пакета ABSTRACT_SET представлен абстрактный тип Set и абстрактные процедуры Rembo и Rolex.

В рассматриваемом ниже пакете GOLD определен тип Elem - как производный от Set, при этом интерфейс для работы с ним наследуется из пакета ABSTRACT_SET , а реализация выполнена с использованием некоторого уже существующего типа Data, который может быть описан в другом пакете.

```
With  ABSTRACT_SET ;  use  ABSTRACT_SET;
package  GOLD is

   type  Elem ( D : Positive) is  new  Set     with   private;
   procedure Rembo ( X :   in   out Elem )              is  abstract;
   function    Rolex  (Y :  Elem)  return   integer  is  abstract;

private
   type   Elem   is   new Data  with
     record
        Z:   Data( 1 ..  D ) ;  end
     record ;

end  GOLD;
```

Для реализации примитивных операций Rembo и Rolex в теле пакета GOLD можно использовать операции, допустимые для типа Data.

П Пример *2:*

1

В примере рассмотрена реализация смешанного наследования. Здесь один из родителей обеспечивает только свойства для производного класса. Для этого используется комбинация расширения тэгового типа и настраиваемого модуля.

```
generic
    type Word   is abstract tagged private ;
package MIX is

    type    Page  abstract  new   Word with private;
    procedure List (X :   in out Page )   is   abstract;

private
    type   Page is abstract new Word with record
                - - дополнительные компоненты
      end record ;

end   MIX;
```

При конкретизации пакета MIX можно добавлять операции типа Page *к* существующему тэговому типу. Получаемый при этом тип будет принадлежать к классу типа , задаваемого фактическим типом настройки. Типы Word и Page являются абстрактными, то есть соответствующий формальный тип должен быть тоже абстрактным.

D Пример 3:

```
package  AMD  is

type Zoo is limited tagged private; procedure
Bear (X : in Zoo); private
    type  Garden ( D :   access Zoo) is limited ... type
    Zoo is limited record
   E :   Garden (Zoo' access ); end
record ; end   AMD;
```


Здесь скрытый в приватной части пакета тип Garden имеет ссылочный дискриминант, причем со ссылкой на тип Zoo . Определение обьекта типа Zoo сязано с созданием динамической структуры (ссылающейся на себя) для этого обьекта, которая будет включать компоненты типа Garden.

Типы Zoo и Garden могут в свою очередь являться расширением некоторого приватного типа Domen .

D Например :

```
type  Garden  ( D :   access   Zoo  ' Class ) is  Domen    with ...
                                                  new
```

Ш СОВЕТЫ:

* Используйте ссылочные дискриминанты для обеспечения множественного наследования.

ИЗМЕНЕНИЯ:

О Тип запись (или приватный тип) может быть помечен как тэговый и может быть расширен при создании производных типов.

О Введено понятие атрибута Class и типа широкого класса (Class-Wide type).

О Введено понятие диспетчерской операции и диспетчерского вызова подпрограммы (dispatching call).

О Введено понятие абстрактного типа и абстрактной подпрограммы.

16.7 Примеры

П Пример 1 . Пакет с тэ! овами типами и их расширениями

```
package  ORBIT_WORK  is

    type   Gum    is integer   range    - 10C .. 10C ;

    type   Orbit   is tagged
        record
        X:   fixed; Y:
        boolean ; end record ;
```

```
    procedure Fiat  ( O      in Orbit;    F : out  Gum ) ;
    procedure Ford  ( O      in  out      Orbit) ;
    procedure Volvo ( O      out          Orbit);

    type Orbit_A is new Orbit  with      --расширение типа  Orbit
        record
           A:   float;
        end record;

    procedure Ford (A:   in out  Orbit);

    type Orbit_B is new Orbit_A with -- расширение типа Orbit_A
        record
           B:  character;
        end record;

    procedure Ford(B: in out  Orbit_B);
    procedure Sun ( B : in            Orbit_B);

end ORBIT_WORK;
```

Пакет Orbit_Work содержит в спецификации описания трех типов: Orbit и два расширенных типа Orbit_A и Orbit_B , являющимися расширениями типов Orbit и Orbit_A соответственно.

Для типа Orbit определены три процедуры Fiat, Ford, Volvo . Для типа Orbit_A определена процедура Ford. Для типа Orbit_B определены процедуры Ford и Sun.

Для всех трех типов определена процедура Ford, то есть она является совмещенной.

Типы Orbit_A и Orbit_B наследуют операции Fiat и Volvo от типа Orbit.

Тело пакета Orbit_Work : **package**

```
body  ORBIT_WORK  is

   procedure Fiat   (O :   in Orbit; F : out Gum )   is separate ;

   procedure Volvo  (O :   out  Orbit)  is  separate ;

   procedure Ford   (O:   in   out  Orbit)   is

   end   Ford;

   procedure Ford   (A:   in   out  Orbit_A)   is

   end Ford;
   procedure Ford   ( B :   in   out  Orbit_B) is
    . . .

   end   Ford; end

ORBIT_WORK;
```

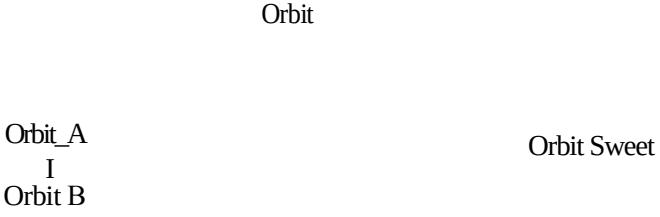

П Пример 2 Использование пакета Orbit_Work

```
with ORBIT_WORK;
package   SWEET   is

   type   Sweet_Orbit  is  new   Orbit_Work. Orbit  with
      record
         Z : Orbit_Work. Gum ; end
      record;
   procedure Fiat   (SO      in     SweetJDrbit;
         FF:  out                  Orbit_Work. Gum ); out
   procedure Ford  ( SO:   in      SweetJDrbit); out
   procedure Volvo  ( SO :         SweetJDrbit);
   procedure Reno  ( SO:   in      SweetJDrbit);

end   SWEET;
```

Для пакета SWEET определен тип SweetJDrbit, производный от Orbit с добавлением компоненты Z. Процедуры Fiat, Ford, Volvo

являются совмещенными. Дополнительно определена процедура Reno.

Совокупность типов Orbit, Orbit_A, Orbit_B и Sweet образуют класс, его корень - тип Orbit. Иерархия классов Orbit:



Типы Orbit_A и Orbit_B образуют подкласс.

G Пример 3. Пакет с абстрактными типами и абстрактными процедурами.

```
package ABSJRACTJDRBIT is

   type Orbit is abstract tagged null record;
   procedure Ford (AO : in out Orbit) is abstract;

end ABSTRACTJDRBIT ;
```

Использование пакета :

```
with ABSTRACTJDRBIT ; package
ORBIT_WORK_2 is

   type Gum is integer range -100.. 100 ;

   type Orbit_A is new Abstract_Orbit. Orbit with
                                  - - расширение типа Orbit

      record X:
         fixed;
```



```
      Y : boolean ; A:
      float; end record ;

   procedure Fiat ( OA
   procedure Ford ( OA        in Orbit_A ; F : out Gum);
   procedure Volvo ( OA       in out Orbit_A ); out
                              Orbit_A);

type Orbit_B is new Orbit_A with --
   record                расширение типа   Orbit B
      B : character;
   end record ;

   procedure Ford ( B : in out Orbit_B ); end

ORBIT_WORK_2;
```

Пакет ABSTRACT_ORBIT определяет абстрактный тип Orbit как запись без компонент и абстрактную процедуру Ford для него. Процедура Ford не имеет тела. В пакете ORBIT_WORK_2 на основании типа Orbit содается тип Orbit_2 с добавлением компонент и новых прмитивных операций. Тип Orbit_B является расширением типа Orbit_A. Операция Ford является совмещенной дли типов Orbit A и Orbit B.

Приложение!. ПАКЕТ STANDARD

Пакет STANDARD содержит все предопределенные идентификаторы языка, предопределенные типы (Boolean, **Integer, Character, Float, Duration)** и операции для них, а также исключения. Пакет автоматически доступен каждому компилируемому модулю и не требует спецификатора **with.**

Спецификация библиотечного пакета STANDARD :

```
package Standard is
   pragma Pure ( Standard);

   type Boolean is (False, Treu);

   - - Предопределенные операции для типа Boolean

   - function "=" (Left,Rigth : Boolean ) return Boolean;
   - function "/=" (Left, Rigth : Boolean ) return : Boolean ;
   - function "<" (Left, Rigth Boolean) return : Boolean;
   - function "<="( Left, Rigth Boolean) return Boolean;
   -function"> "(Left, Rigth: Boolean) return : Boolean;
   - function ">="( Left, Rigth Boolean ) return Boolean;

 - - Предопределенные
 логические операции для этого типа

 - - function "and" (Left, Rigth : Boolean) return Boolean ;
 - - function "or"  (Left, Rigth : Boolean ) return Boolean ;

 - -function "xor"  (Left, Rigth : Boolean ) return Boolean ;
 - - function "not" (Left, Rigth : Boolean) return Boolean;

 - - Целый тип

 type Integer is   определен -реализацией;

 - - Подтипы целого типа

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



```
 subtype Positive is Integer range 1 .. Integer'Last; для типа Integer:

 - - Предопределены операции

 - - function "="  ( Left, Rigth : Integer'Base) return Boolean;.
 -- function7="  (Left,Rigth : Integer'Base) return Boolean;
 - - function "<"  (Left, Rigth : Integer'Base) return Boolean;
 -- function"<="  (Left,Rigth : Integer'Base) return Boolean;
 - - function ">"  (Left, Rigth : Integer'Base) return Boolean;
 - - function ">=" (Left, Rigth :

 - - function"+"  ( R igth : Integer'Base) return Integer'Base;
 - - function "abs" ( Rigth : Integer'Base) return Integer'Base;
 - - function"-"  ( Rigth : Integer'Base) return Integer'Base;

 - - function "+"  ( Left, Rigth : Integer'Base) return Integer'Base;
 - - function"-"  ( Left, Rigth: Integer'Base) return Integer'Base;
 - - function "*"  (Left, Rigth: Integer'Base) return Integer'Base;

 - - function"/"  ( Left, Rigth: Integer'Base) return Integer'Base;
 - - function "rem" ( Left, Rigth: Integer'Base) return Integer'Base;
 - - function "mod"(Left, Rigth: Integer'Base) return Integer'Base;

 - - function "**"  ( Left, Rigth: Integer'Base) return Integer'Base;

 - - Вещественный тип type Float is

   определен_реализацией;

 - Для этого типа предопределены следующие операции:

 --function "="  (Left, Rigth: Float)  return Boolean;
 --function "/=" (Left, Rigth: Float)  return Boolean;
 - - function "<"  ( Left, Rigth: Float)  return Boolean;

 - - function "< =" ( Left, Rigth : Float)  return Boolean;
```

```
   -- function ">"     (Left,Rigth: Float)   return   Boolean;
   -- function ">=" ( Left, Rigth : Float)  return   Boolean;

   -- function "+"    (Left, Rigth: Float)    return    Float;
   - - function "-"    ( Left, Rigth : Float )    return    Float;
  - - function "abs" ( Left, Rigth : Float )    return    Float;

    --function "+"     (Left, Rigth: Float)     return   Float;
    --function"-"      (Left, Rigth: Float)     return   Float;
    --function"*"     (Left, Rigth: Float)     return   Float;
   - - function "/"      (Left, Rigth : Float )    return   Float;

   - - function "**" (Left, Rigth : Float'Base )  return    Float ;
```

~ Кроме того, для универсального вещественного типа и корневого числового (rootjnumeric) предопределены операции "*" и

```
- - символьный тип

type   Character  is ( null ...    ) ;
```

- - Предопределенные операции для типа Character - те же самые, что и для любого перечисляемого типа.

```
   Type Wide_Character is ( nuil   . . . FFFE, FFFF } ;

- - стандартный   набор  символов  ASCII
   package  ASCII   is . .   end ASCII :
```

-Предопределенный строковый т:;п:

```
   type String is am[1]   ?T<- -^t?ve r ,«;ge < > ) c.f Character;
   pragma Pack (K f л:

   --Следующие о!.и-..'иши предопределены для типа String:

   --function "="    (Left, Rigth: String) return Boolean;
   --function '/=" (Left, Rigth: String) return Boolean;
```



```
   --function "<"   (Left, Rigth: String) return Boolean;

   --function "<=" (Left, Rigth: String) return       Boolean;
   --function ">"   (Left, Rigth: String) return       Boolean;
   --function ">="(  Left, Rigth. String) return       Boolean;

   --function "&"    ( Left. String;     Rigth: String)      return String;
   - - function "&"   ( Left: Character; Rigth: String)    return String;
    - -function "&"   ( Left: String; Rigth: Character)      return String;
    - -function "&"   ( Left: Character ;Rigth: Character) return String;

   type Wide_String is array ( Positive range <>) of Wide_Character;
   pragma Pack ( Wide_String);

f  type   Duration   is   delta   определяет реализацией  range
                                  определено реализацией;

    - - Для типа Duration предопределены все те операции, что и для

    - - любого   другого   фиксированного типа.

-  Предопределены   следующие   исключения:

CONSTRAINT_ERROR          :          exception;

PROGRAM_ERROR   :   exception;   STORAGE

_ERROR   :   exception;   TASKING_ERROR   :

exception; end Standard;
```

Приложение 2. ПАКЕТ SYSTEM

В пакете описаны основные характеристики , зависящие от конфигурации системы.

```
Package System is
   pragma Preelborate ( System );

   type Name is определенный_реализацией_перечисляемый_тип System
   _Name : contstant Name : = определено^реализацией;

- - Системно-зависимые именованные числа

  Min_mt : constant: = root_integer'First;
  Max_Int : constant: = root_integer'Last;

  Max_Binary_Modulus : constant: = определено реализацией
  Max_Nonbinary_Modulus : constant: = определено „реализацией

  Max_Base_Digits
  Max_Digits : constant: = определено реализацией
             \ constant: = определено реализацией

  Max_Mantissa
  Fine Delta : constant: = определено реализацией
             : constant: = определено реализацией

  Tick : constant: = определено ^реализацией

-' описания связянные с памятью

  type Address is определено реализацией
  Null_Address : constant Address ;

  Storage_Unit : constant: = определено реализацией
  Word_Size : constant: = определено реализацией * Storage_Unit;
  Memory_Size : constant: = определено реализацией


  function "<" (Left, Right: Address) return Boolean ;
```



```
  function "<=" ( Left, Right: Address) return Boolean ;
  function ">" ( Left, Right: Address ) return Boolean ;
  function ">=" ( Left, Right: Address ) return Boolean ;
  function "<" ( Left, Right: Address ) return Boolean ;
  function "=" ( Left, Right: Address ) return Boolean ;
  function "/=" (Left, Right: Address) return Boolean ;

  pragma Convention (Intrinsic, "<"); ... - - далее для все
подпрограмм, определенных в этом пакете

- - другие системно-зависимые определения : type Bit_Order is

  ( High_Order_First, Low_Order_First) ; Default_Bit_Order:

  constant Bit_Order;

- - Описания, связанные с приоритетами subtype Any_Priority is

  Integer range определено .реализацией;

  sybtype Priority is Any_Priority range
                      Any'Priority'First .. определенореализацией;

  subtype Interrupt_Priority is Any_Priority range

                                        Priority'Last + 1 .. );

  Default_Priority : constant Priority ( Priority'First + Priority'Last);

private
  ... - - не определено в языке
end System;
```

f.

Приложение3. П А К Е Т Т Е Х Т _ Ю

Пакет **Ada.Text_IO** обеспечивает текстовый ввод-вывод.

```
with Ada..IO_ Exceptions;
package Ada.Text_ IO is

  type File_Type is limited private;

  type File_Mode is ( In _ File, Out_File, Append_File );

  type Count is range 0.. определяется „реализацией; subtype

  Positive_Count is Count range 1 .. Count'Last; Unbounded :

  constant Count: = 0 ;

  - - длина страницы и строчки

  subtype Field is Integer range
                              0 .. определяется_реализацией;

  sybtype Number_Base is Integer range 2.. 1 6;

  type Type_Set is (Lower_Case, Upper_Case);

  -Управление файлами

  procedure Create ( File : in out File_Type;
                     Mode : in File_Mode := Out_File;
                     Name : in String := "";
                     Form : in String := "" );

  procedure Open ( File : in out File_Type;
                   Mode : in File_Mode;
                   Name : in String;
                   Form : in String := "");
```

```
   procedure Close  (File    :    in  out   File_Type);

   procedure Delete (File    :    in  out   File_Type );

   procedure Reset   (File   :    in  out   File_Type;
                                            Mode:    in File_ Mode
   ); procedure Reset   (File   :    in out File_Type);

   function  Mode  (File     :    in     File_Type )    return   File_Mode;

   function  Name  (File     :    in     File_Type)     return   String;

   function  Form   (File    :    in     File_ Type)    return   String;

   function  Is_Open(File    :    in     File_Type)     return   Boolean;
```

- - Управление файлами ввода-вывода, определяемыми по умолчанию

```
   procedure   Set_ Input   (File :  in  File_Type);

   procedure   Set_ Output (File :  in  File_Type );

   procedure   Set_ Error   (File :  in  File_Type);

   function     Standard_Input      return   File_Type;

   function     Standard „Output  return  File_Type;

   function     Standard_Error      return  File_Type;

   function     Current_Input        return   File_Type;

   function     Current_Output       return  File_Type;

   function     Current_Error        return   File_Type; type

   File_Access  is  access  constant  File_Type;
```

```
function Standard_Input return File_Access;
function Standard _Output return File_Access;
function Standard_Error return File_Access;
function Current_Input return File_Access;
function Current_Output return File_Access;
function Current_Error return File_Access;
```

- - Управление буфером **procedure** Flush (File :

in **out** File_Type,) ; **procedure** Flush;

- Определение длины страниц и строчек

procedure Set_Line_Length (File: in FiIe_Type ; To: in Count);

procedure Set_Line_Length (To : in Count); **procedure** Set_

Page_ Length (File: in File_Type; To : in Count); **procedure** Set_

Page_ Length (To : in Count); **function** Line_ Length (File : in

FileJType) **return** Count; **function** Line_ Length **return** Count;

function Page_ Length (File: in File_Type) **return** Count;

function Page_ Length **return** Count;

-Управление страницами, строчками и столбцами

```
procedure New_Line ( File : in File_Type;
                     Spacing : in Positive_ Count: = 1);
procedure New_Line ( Spacing : in Positive_Count: = 1 );
procedure SkipJLine ( File : in File_Type;
                      Spacing : in Positive_Count := 1 );
procedure Skip_Line ( Spacing : in Positive_Count: = 1);
function End_Of_Line (File : in File_Type) return Boolean;
function End_Of_Line return Boolean;
procedure New_Page ( File : in File_Type);
procedure New_Page;
procedure Skip_Page (File : in File_Type);
procedure Skip_Page;
function End_Of_Page (File : in File_Type) return Boolean;
function End_Of_ Page return Boolean;
function End_Of_File (File : in File_Type) return Boolean;
function End_Of_File return Boolean;
procedure Set_Col (File : in File_Type; To: in Positive_Count;);
procedure Set_Col (To : in Positive_Count);
procedure Set_Line (File : in File_Type; To: in Positive_Count);
procedure Set_Line (To : in Positive_Count);
function Col (File : in File_Type) return Positive_Count;
```

```
function Col return Positive_Count;

function Line (File : in File_Type) return Positive_Count;

function Line return Positive_Count;

function Page (File : in File_Type) return Positive_Count;

function Page return Positive_Count;

— Символьный ввод-вывод

procedure Get (File : in File_Type; Item: out Character);

procedureGet (Item : out Character);

procedure Put (File : in File_Type; Item: in Character);

procedure Put (Item : in Character ;

procedure Look_Ahead (File in FileJType ;
                      Item out Character;
                      End_of_Line out Boolean);

procedure Look_Ahead (Item out Character;
                      End_of_Line out Boolean );

procedure Getjmmediate (File in FileJype;
                        Item out Character);

procedure Getjmmediate (Item out Character );

procedure Getjmmediate ( File in File_Type;
                         Item out Character;
                         Available out Boolean );

procedure Getjmmediate (Item : out Character;
```



```
                        Available : out Boolean);

- Строковый ввод-вывод

procedure Get (File : in File_Type;
               Item : out String );

procedure Get fltem : out String );

procedure Put (File : in File_Type; Item
               : in String ) ;

procedure Put (Item : in String );

procedure Get_ Line (File : in File_Type;
                     Item : out String;
                     Last : out Natural);

procedure GetJJne (Item : out String;
                   Last : out Natural);

procedure PutJLine (File : in FileJType;
                    Item : in String );

procedure Put_Line (Item : in String ); -

Настраиваемый пакет для ввода-вывода целых

generic
  type Num is range < >;
package IntegerJO is

  Default_Width : Field : = Num'Width;
  Default JJase : Number JJase : = 10;

  procedure Get (File : in File_Type;
                 Item : out Num;
```

```
                    Width  : in   Field: =0);
   procedure  Get (Item   : out  Num ;
                   Width  : in   Field : = 0 );

   procedure  Put ( File  : in  File_Type;
                    Item  : in  Num;
                    Width : in  Field:= Default _ Width;
                    Base  : in  Number_ Base : = Default _ Base ) ;

   procedure  Put (Item   in  Num;
                   Width  in  Field: = Default _ Width;
                   Base   in  NumberJJase := Default _Base);

   procedure  Get (From   in String; out
                   Item   Num; out
                   Last   Positive );

   procedure  Put (To     out String;
                   Item   in  Num;
                   Base   in  Number_ Base: =
                                          Default _Base );
end  Integer_ IO;


generic
   type  Num  is  range < >;
package  Modular_IO  is

   Default_Width : Field          : = Num'Width;
   Default_Base  : Number_Base    : = 10;

   procedure  Get (File   in   File_Type;
                   Item   out  Num;
                   Width  in   Field: =0);
   procedure  Get (Item   out  Num ;
                   Width  in   Field: = 0);

   procedure  Put (File   in  File_Type;
                   Item   in  Num;
```



```
                   Width  : in   Field:= Default _ Width;
   Base   : in                   Number_ Base : = Default _ Base ) ;

   procedure  Put (Item   : in   Num;
                   Width:  in   Field: = Default_Width;
                   Base   :  in  Number_Base := Default _Base);

   procedure  Get (From Item  in String; out
                   Last       Num;       out
                              Positive);

   procedure  Put  (To    : out String;
                    Item  : in  Num;
                    Base  : in  Number_ Base: =
                                          Default _Base);
end  Modular. IO;
```

~ Настраиваемые пакеты для ввода-вывода вещественных

```
generic
   type  Num  is  digits<>;
package  Float_IO  is

   Defaul_Fore  Field          : = 2;
   Defaul_Aft   Field          := Num'Digits -1;
   Defaul_ Exp  Field          : = 3;

              procedure  Get (File   :  in
                     File_Type; Item   :  out
                     Num; Width :  in   Field
                     = 0);

   procedure  Get (Item   :  out  Num;
                   Width:  in   Field: = 0);

   procedure  Put  (File  :   in   File_Type;
                    Item  :  in  Num;
    Fore :                    in   Field: =  Defaul_Fore;
                    Aft   :  in  Field: = DefauL Aft;
```

2(1*

```
                Exp  :  in  Field: = Defaul_ Exp);

    procedure Put (Item   :  in Num; in Field: =
              Fore  : Aft    Defaul_Fore ; in Field: =
              : Exp  :       Defaul_Aft ; in Field: =
                             Defaul_ Exp );

    procedure Get (From :  in  String ; out Num ; out
              Item  : Last     Positive );
              :

    procedure
                  Put (To   : out String; Item :  in
                       Num; Aft   : in Field : = Default.
                       Aft; Exp :  in Field : = Default.
                       Exp );
end Float_IO;

generic
   type  Num  is  delta <>;
package  Fixed_IO is

   Default_Fore: Field:  =  Num'Fore-
   Default _Aft : Field := Num 'Aft;'
   Default _Exp : Field: = 0 •
                          in  File_Type; out
   procedure    Get  (File   Num; in   Field := 0 );
                      Item   out  Num; in   Field:
                      Width  = 0 );
   procedure    Get  (Item
                      Width  in                    Fil
   procedure  Put (File   ;  e_Type;
                      Item   in
                      Fore
                      Aft    m;                    Nu
                      Exp    ; in
                                                   Fie
                          ld: = Default _Fore ;
                          in    Field: = Default _Aft;
                          in    Field : = Default _Exp);
                    Num; Field : = Default _Fore;
procedure    Put (Item
                  Fore
```



```
                  in
                  Aft    : in   in   Field : = Default  _Aft;
                  Exp    :      in   Field : = Default  Exp );

    procedure     Get  (From in   String;
                         Item  in   out  Num;
                         Last  out  Positive);

    procedure     Put (To      out  String;
Item Aft Exp end  Fixed_IO;    in   Num;
                               in   Field := Default
                                                    .Aft;
generic                        in   Field: = Default Exp);
   type  Num  is  delta <>
digits <>; package  Decimal_ Ю  is

   Default_Fore: Field : = Num'Fore;
   Default _Aft  : Field: = Num'Aft;
   Default _Exp : Field: = 0 ;

   procedure  Get (File Item '  in   FileJType;
                        Width   out  Num;
                                in   Field := 0);
   procedure  Get (Item   :
   out   Num ;
                        Width :  in   Field: = 0);

   procedure  Put (File Item    in   File_Type;
                        Fore    in   Num;
                        Aft Exp in   Field : = Default
                                in   Field : = Default.
   procedure  Put (Item   :     in   Field : = Default
                                i                    Fore;
   n                            Num;                .Aft;
                        Fore    in   Field: = Default JExp);
                        Aft     in   Field: = Default
   Exp                             in   Field : = Default.
                                                     _Fore;
                                                     _Aft; -
                                                     Exp );
```

```
   procedure     Get (From in                                  Strin
                      Item
                      Last g;
                           in                                  out
   procedure  Put (To      Num;
Item Aft Exp end  DecimaL out
IO;                                                            Positi
                           ve);

                           out
                                String;
                              in   Num;
                              in   Field := Default _ Aft;
                              in   Field: = Default_ Exp );
```

```
-- Настраиваемый пакет для ввода-вывода перечислимых
generic
   type  Enum  is  (< >); package
Enumeration _ IO  is

   Default_Width :  Field ;
   Default_Setting: Type _Set: = Upper_Case;

procedure  Get (File  :      in  File_Type ; out  Enum );
           Item :

procedure  Get (Item  :      out  Enum );

procedure  Put (File  : Item in  File_ Type ; out Enum; in
           : Width : Set  :  Field : = Default_Width; in
                             Type_Set : = Default_Setting):



procedure  Put( Item  :      in  Enum ; in  Field: =
           Width : Set       : Default_Width; in Type _Set :
                             = Default_Setting);


   procedure  Get( From   :  in String;
                      Item   :  out  Enum;
 Last                        :  out  Positive);
```



```
   procedure    Put (To     :  out String ;
                      Item   :  out  Enum; Set
                      :  in  Type _Set: =
                                     Default_Setting);

end  Enumerations _ IO; - -

Исключения

   Status_ Error : exception renames   IO_ Exceptions. Status_ Error;
   Mode_ Error  : exception renames    IO_ Exceptions. Mode_ Error;

   Name_ Error : exception renames     IO_ Exceptions. Name_ Error;
   Use_ Error   : exception renames    IO_ Exceptions. Use_ Error;

   Device_Error : exception renames    IO_ Exceptions. Device_Error;
   End_ Error   : exception renames    IO_ Exceptions. End_ Error;

   Data_Error   : exception renames    IO_ Exceptions. Data_Error;
   Layout_Error : exception renames    IO_ Exceptions. Layout_Error;

private

   ... - - зависит от реализации

end Ada .TextJO;
```

Приложение 4. ПРАГМЫ, ОПРЕДЕЛЕННЫЕ В ЯЗЫКЕ

Pragma All_CalIs_Remote
[(Имя_библиотечного_Модуля **)];** **Pragma Asynchronous** (
Локальное_Имя);

Pragma Atomic (*Локальное_Имя*) ;
Pragma Atomic_Components (*Локальное_Имя_Массива)* ;
Pragma AttachJBandler (*Имя_Обработчика, Выражение*);

Pragma Controlled (*Локальное_Имя_Первого_Подтипа*);

Pragma Convention ([Convention => *Идентификатор]*
I Entity =s>-*Локальное_Имя*]);

Pragma Discard_Names [([**On** =>] *Локальное_Имя*)];
Elaborate (*Имя_библиотечного_Модуля {,*
Имя_библиотечного_Модуля });
Pragma Elaborate_All (*Имя_библиотечного_Модуля {,*
Имя_библиотечного_Модуля });
Elaborate_Body[(*Имя_библиотечного_Модуля*)];

Pragma

Pragma

Pragma

Pragma
Export ([Convention => ***Идентификатор]***
[**Entity** => *Локальное_Ймя]*);

Pragma Import ([Convention => *Идентификатор]*
I **Entity** =>*Локальное_Имя]*);

Pragma Inline (*Имя,* (*Имя*)) ;
Pragma InspectionJPoint [(*Имя_Обьекта, Имя_Обьекта)]*;*

Pragma Interrupt_Handler (*Имя_Обработчика*)**; Pragma Interrupt_Priority** { (*Выражение)];* **Pragma Linker_Options** (*Строковое_Выражение*) ; **Pragma List** (*Идентификатор*);

Pragma Locking_Policy (*Идентификатор_Дисциплины*);

Pragma NormaIize_Scalars ;

Pragma Optimize (*Идентификатор*);

Pragma Pack (*Локальное_Имя_Первого_Подтипа*
) ; **Pragma Page ;**

Pragma Preelaborate [(*Имя_библиотечного_Модуля* **)**

]; Pragma Priority (*Выражение*); **Pragma Pure [** (

Имя_Библиотечного_Модуля **)]**; **Pragma**

Queuing_Policy (*Идентификатор ,Дисциплины)* ;

Pragma Remote_Call_Interface [(*Имя_Библиотечного_Модуля)*
]; **Pragma RemoteJTypes [(***Имя_Библиотечного_Модуля*)];

Pragma Restrictions ([*Ограничение, Ограничение]*);
Pragma Reviewable;

Pragma Shared_Passive [(*Имя_Библиотечного_Модуля)]*;
Pragma Storage_Size (*Выражение*);

Pragma Suppress (*Идентификатор* [, [On => J *Имя])*;

Pragma Task_Dispathcing_Policy
(*Идентификатор•_Дисциплины)* ;

Pragma Volatile (*Локальное_Имя)* ;
Pragma VolatiIe_Components (*Имя_Локального^Массива)* ;

СЛОВАРЬ ТЕРМИНОВ

В данном приложении описаны некоторые термины, используемые в языке. При этом основное внимание уделено тем терминам, которые появились в новом стандарте языка, или их трактовка изменилась по сравнению с Адой 83.

Абстрактная подпрограмма (abstract subroutine). Подпрограмма, не имеющая тела. Требует совмещения в месте наследования. Используется при диспетчерском вызове.

Абстрактный тип (abstract type). Разновидность тэгового типа. Предполагает использование в качестве родительского типа при расширении типа. Не имеет обьектов. Назначение - создание общего интерфейса для нескольких абстракций.

Библиотечный модуль (library unit). Отдельно компилируемый модуль (пакет, настраиваемый модуль, подпрограмма). Может иметь дочерние и родительские библиотечные модули, образуя подсистему. Дискриминант (discriminant). Специальная компонента обьекта или значение именованного типа. В Аде 95 расширена для задачного и защищенного типов.

Диспетчерский вызов(dispatching call). Вызов совмещенной примитивной подпрограммы с аргументом тэгового типа, при котором выбор ее соответствующей реализации выполняется динамически на основании тэга фактического параметра.

Задачный тип (task type). Составной тип, элементами которого являются задачи. Может иметь дискриминант.

Защищенный тип (protected type). Составной тип, компоненты которого защищены от одновременного доступа.

Класс (class). Обьединение типов. Типы класса имеют общий набор примитивных операций.Класс всегда замкнут при порождении, то есть если тип принадлежит классу, то все типы, производные от него, тоже принадлежат этому классу.

Лимитированный тип (limited type). Тип, для которого не допускаются операции присваивания.

Настраиваемый модуль (generic unit). Шаблон программного модуля. используемый для создания модуля с заданными свойствами при конкретизации . Может быть параметризован через обьекты, типы, подпрограммы, пакеты.В Аде 95 разрешено использование формальных пакетов.

Приватное расширение (limited extension) . Расширение записи, при котором добавленная часть расширения является невидимой для пользователя.

Примитивные операции (primitive operations). Операции , которые определяются вместе с объявлением типа. Они наследуются в классе. Для тэговых типов примитивные операции - диспетчерские подпрограммы, обеспечивающие динамический полиморфизм. Программа (programm). Множество сегментов, каждый из которых может выполняться в отдельном адресном пространстве , возможно в отдельном компьютере.

Программный модуль (programm unit). Пакет, задача, защищенный модуль, настраиваемый модуль, подпрограмма. Средства построения программ в языке.

Производный тип (derived type). Тип, определяемый с помощью другого типа(родительского). Наследует свойства родительского типа (компоненты и примитивные операции). Класс родителского типа всегда содержит все его производные типы, образуя производный класс.

Расширение записи (record extension). Тип, получаемый из типа записи, помеченной тэгом, путем добавления новых компонентов. Сегмент (partition). Элемент распределенной программы. Содержит набор библиотечных модулей. Может выполняться в отдельном адресном пространстве.

Тип широкого класса (wide-class type). Тип T'Class, определяемый для любого тэгового типа Т. Объединяет все типы, производные от Т. Является неограниченным.

Тэговый тип (tagged type). Тип , помеченный словом tagged. Предназначен для реализации в языке парадигм обьектно-ориентированного программирования. Тэговый тип может быть расширен с помощью добавления новых компонент. Целый тип (integer type). Простой целый и модульный целый типы.

СПИСОК

1. Вегнер П. Программирование на языке Ада : Пер. с англ.- М.: Мир 1983.-239с.
2. Бар Р. Язык Ада в проектировании систем : Пер. С англ.- М.: Мир 1988.-320 с.
3. Василеску Ю. Прикладное программирование на языке Ада : Пер. с англ.- М.: Мир, 1990. - 348 с.
4. Джехани Н. Язык Ада : Пер. с англ.- М.: Мир, 1988. - 549 с.
5. Дисков Б., Гатэг Дж. Использование абстракций и спецификаций при разработке программ : Пер. с англ.- М.: Мир, 1989. - 424 с.
6. Органик Э. Организация системы ИНТЕЛ 432 : Пер. С англ - М • Мир, 1987.-446с.
7. Пайл Я . Ада - язык встроенных систем : Пер. с англ.- М.: Мир, 1984. - 238 с.
8. Перминов О.Н. Введение в язык программирования Ада. - М.: Радио и связь, 1991. - 228 с.
9. Язык программирования Ада. ГОСТ 27831 - 88.
10. Ada programming language. American National Standards Institute, Inc. ANSI/MIL - STD-1815A-1983.
11. Ada 95 Rationale. January 1995.
12. Ada 95 Reference Manual. ISO/EEC 8652 -1995, 1995.
13. Ada 95 Quality and Style : Guidelines for Professional Programmers. SPC-94093-CMC. Ver. 01.00.10, October 1995, DOD AJPO.
14. Feldman M.B. Software Construction and Data Structures with Ada 95. - Addison- Wesley, -1996 .
15. Hansen Per Brinch. The Architecture of Concurrent Programming, Engelwood Clifs, NJ, Printice-Hall, Inc., 1977 .
16. Hoare C.A.R. Monitors : An operating System Structuring Concept. Comm. Of ACM., 1974, V/ 17, No 10, pp. 549 - 557 .
17. International Standard programming language - Ada. ISO 8652 : 1987 .

Шдп. додруку 12.03.98

ПашрДРУк № 2 Умовн. друк-арк. 13,95

Тираж 1000.

м. Київ, вул. Солом'янська, 3

Фірма 252151, Ки'ш, вул. _____ осі6 Волинська 60. ти. Умовн. Фарбо-

ТОВ вщб. 14,61 Зам.№ 8738