Государственный комитет Российской Федерации
по высшему образованию

Самарский государственный аэрокосмический
университет имени академика С.П. Королева

М.А.Кораблин

ПРОГРАММИРОВАНИЕ, ОРИЕНТИРОВАННОЕ НА ОБЪЕКТЫ

Учебное пособие

Самара 1994

УДК 681.142.2

Программирование, ориентированное на объекты: Учебное пособие/ М.А.Кораблин. Самар. госуд. аэрокосм. ун-т; Самара, 1994. 94 с. ISBN 5-230-16-955-9

Пособие посвящено одному из основных направлений современного программирования, связанному с объектно-ориентированным подходом к разработке программ. Описываются основные концепции такого подхода, методы и средства его реализации, в совокупности составляющие особый стиль программирования.

В первую очередь ориентировано на студентов, изучающих информатику и связанных с задачами программирования прикладных информационных систем. Может быть рекомендовано при изучении дисциплин "Программирование", "Технология программирования", "Основы информационной технологии", "Моделирование на ЭВМ". Рекомедуется для использования в учебном процессе специальностей "Прикладная математика", "Автоматизированные системы обработки информации и управления", "Программное обеспечение вычислительных и автоматизированных систем". Выполнено на кафедре "Информационные системы и технологии".

Печатается по решению редакционно-издательского совета Самарского государственного аэрокосмического университета имени академика С.П.Королева

Рецензент С. В. Смирнов

ISBN 5-230-16-955-9

ПРЕДИСЛОВИЕ

Настоящее пособие не является руководством по какому-либо языку программирования. Более того, цель его заключается не в том, чтобы научить технике программирования. В него вошел материал, связанный с концепцией объектно-ориентированного подхода к разработке программ, в соответствии с которой окружающий нас реальный мир интерпретируется как совокупность взаимосвязанных и взаимодействующих объектов. Моделирование задач реального мира в рамках этой концепции связано с описанием (спецификаций) объектов реального мира в адекватных категориях языка программирования, что требует нового взгляда на уже сложившиеся методы программирования и связано в известном смысле с переосмыслением многих хорошо известных и устоявшихся понятий.

Основная цель данного пособия заключается в том, чтобы донести до читателя в сжатой лаконичной форме основные концепции объектно-ориентированного подхода, проиллюстрировать их и сформировать общее представление об этом направлении, которое позволит внимательному читателю легко перейти от уровня понимания подхода в целом к уровню умения его реализовать в разработках конкретных программ. Для этого в общем случае даже не обязательно использовать современные объектно-ориентированные языки (во многом "перегруженные" специальными понятиями). Многие аспекты объектно-ориентированного подхода могут быть реализованы и в известной технике модульного программирования с использованием абстрагирования типов, механизмов импорта-экспорта, процессов, сопрограмм и т.д.

Автор считал бы свою задачу выполненной, если бы у читателя на основе этого пособия сложился собственный критический взгляд

на объектно-ориентированное конструирование программных моделей. Такой взгляд особенно важен, поскольку программирование - быстро развивающаяся область знания. Многие понятия объектно-ориентированного подхода на сегодняшний день нельзя признать вполне сложившимися не только в методическом, конструктивном, но и в концептуальном отношении. Они не имеют строго определенной формальной математической основы и полностью базируются на интуиции и "здравом смысле". В этом плане использование объектно-ориентированного подхода в одних областях оказывается весьма плодотворным, в других - нет.

Фрагменты программ, приведенные в пособии, оформлены с использованием нотации, принятой в языке Модула-2. Выбор этого языка основан на двух обстоятельствах: традиция коллектива, в котором работает автор, и внутренняя стройность Модулы, позволяющая расширять программные разработки на строгой основе. Вместе с тем Модула-2 является представителем группы "паскалоидов", которая широко распространена.

Пособие рассчитано на читателя, который имеет некоторый опыт программирования на языке, имеющем средства абстрагирования типов, но вместе с тем не отягощен большим грузом старых проблем в технологии программирования, способен ощутить стройность математической интерпретации отдельных механизмов структуризации и готов сменить сложившиеся или только складывающиеся у него стереотипы. Все эти условия, по-видимому, необходимы для того восприятия материала, на которое рассчитывает автор.

Посмотрите на хорошо известный Вам мир программирования через объектно-ориентированные очки - может быть то, что Вы увидите, даст новый импульс к развитию Ваших способностей в этой области.

I. РАЗВИТИЕ КОНЦЕПЦИИ СТРУКТУРИЗАЦИИ В ЯЗЫКАХ ПРОГРАММИРОВАНИЯ

Понятие структуры всегда ассоциируется со сложным объектом, обладающим *свойством целостности*, и вместе с тем составленным из простых компонет (частей, элементов) путем использования определенной системы правил. Программирование можно интерпретировать как искусство разложения и классификации целого на части- декомпозиции решаемой задачи. В этом плане структуризацию в программировании можно трактовать как правила такой декомпозиции. Возможна, разумеется, декомпозиция и без правил, но в этом случае (как и в любой игре без правил) понять, как из частей образуется структура, трудно, а в общем случае, невозможно.

Исторически структуризация в программировании начиналась с введения в языки программирования управляющих структур - операторов условного перехода, выбора, циклов с различными правилами повторения и выхода и т.п. Цель такой структуризации заключалась в повышении читаемости и понимаемости разрабатываемых программ. Программирование с использованием оператора безусловного перехода (GO TO) в этом плане считалось нежелательным, не вписывающимся в систему правил структуризации. Из некоторых языков программирования этот оператор был вообще удален, чтобы не вводить программистов в искушение писать лаконичные, эффективные, хорошо работающие, но трудно понимаемые и неструктурные (!) программы. (Впрочем, в более поздних версиях этих же языков "неудобный" GOTO неожиданно "воскресал", несмотря на всю его "неструктурность").

Впоследствии сложилось мнение, что структуризация - это стиль программирования. Можно писать программы, следуя такому стилю (и используя GOTO), а можно писать вполне неструктурно и вместе с

тем, без GOTO.

Языки програмирования, в которые были введены управляющие структуры, оказались первым шагом на пути от ассемблера до современных языков (языки первого поколения, например, FORTRAN). Следующим этапом в развитии концепций структуризации явилось осознание необходимости структуризации данных. Появление таких структур, как записи, положило начало использованию в языках программирования механизмов абстрагирования типов (языки второго поколения, пример - PL1). Развитие этих механизмов, интерпретация типа как алгебры (множество объектов + множество операций над ними) и использование модуля как программного эквивалента абстрактного типа связано с появлением языков третьего поколения (Clu, Модула-2 и др.). Отличительной особенностью этих и им подобных языков является наличие развитых средств абстрагирования типов. В этом плане хорошо известная техника модульного программирования оказалась удачной основой, на которой концепция абстрагирования могла получить новые дополнительные качества. Среди них в первую очередь возможности инкапсуляции и механизмы импорта-экспорта. Инкапсуляция позволяет рассматривать модуль как набор программных объектов, помещенных в оболочку - капсулу. Такая оболочка может быть "непрозрачной", делающей невозможным использование объектов, определенных в модуле, вне его, "полупрозрачной", - в этом случае вне модуля известны только общие свойства объекта (например, заголовок процедуры), и полностью "прозрачной" (за пределами модуля можно использовать все свойства его объектов). Механизмы импорта-экспорта регулируют "степень прозрачности" капсулы модуля путем использования соответствующих деклараций определенных объектов.

Два отмеченных аспекта определяют языки, которые можно назвать языками, ориентированными на объекты. В таких языках программа определяется как набор модулей, каждый из которых содержит в себе определение абстрактного типа T, действий над объектами этого типа F_t и внутренних схем поведения объектов W_t. T и F_t экспортируются "полупрозрачным экспортом , W_t - "невидимы" вне модуля. Таким образом, любой модуль определяется триадой $M=<N,F_t,W_t>$, а механизмы импорта-экспорта определяют статические межмодульные связи.

В этой интерпретации модуль должен рассматриваться как программный эквивалент определенного класса объектов, содержащий в себе всю информацию об объектах этого класса. Например, модуль, реализующий класс объектов ТОЧКА, должен содержать описание абстрактного типа "точки" (T) и действия над объектами класса ТОЧКА (F_t), например, следующие:

```
PROCEDURE Create (X,Y:CARDINAL): ТОЧКА;
                        (Создать точку с координатами X,Y).
PROCEDURE Destroy (VAR T: ТОЧКА);  (Удалить точку T).

PROCEDURE Sm (T: ТОЧКА; New_X, New_Y: CARDINAL);
(Переместить точку T в новые координаты New_X, New_Y).
```

W_t в этом примере должны реализовать скрытые в модуле механизмы, связанные с реализацией F_t. В общем случае W_t могут быть связаны с созданием процессов "жизни" объектов класса. Например, описание класса "ТОЧКА, ДВИЖУЩАЯСЯ ПО ЭКРАНУ МОНИТОРА" должно инкапсулировать в себе процессы такого движения.

Подчеркнем, что модуль $<T,F_t,W_t>$ как программный эквивалент

класса содержит в себе описание только свойств этого класса. Объекты класса создаются вне модуля, а их число в общем случае непредсказуемо (в приведенном примере - это множество одновременно движущихся точек). Это обстоятельство приводит к тому, что переменные как программные эквиваленты объектов класса не определяются в модуле-классе и соответственно не экспортируются за его пределы. (В модуле-классе ТОЧКА не определена ни одна конкретная точка, определены лишь правила конструирования точек). В этом смысле экспорт переменных-объектов (часто разрешенный формально) должен рассматриваться как нарушение стиля объектно-ориентированного программирования.

Языки, ориентированные на объекты, являются предтечей объектно-ориентированных языков. Последние характеризуются наличием специфического механизма, реализующего отношения класс-подкласс (тип-подтип), связанного с использованием механизмов наследования свойств, основанных на таксономических моделях обобщения. Таксономия как наука сложилась в 19-м веке в результате систематизации наблюдений в биологии (в первую очередь). Такая систематизация заключалась в установлении отношений общего к частному, например:

"Млекопитающее" *> "Обезьяна" *> "Шимпанзе".

Класс (первоначально использовался термин "таксон") "Млекопитающее" характеризуется общими свойствами, подкласс "Обезьяна" в дополнение к этим свойствам обладает уточняющими (частными) свойствами, присущими только обезьянам, и т. д. Таким образом, использованный нами символ "*>" указывает направление расширения (дополнения) свойств класса его подклассами.

Механизм наследования свойств в объектно-ориентированных язы-

ках позволяет повысить лаконичность программ путем использования деклараций "класс-подкласс" и их надежность, поскольку любой подкласс может быть разработан на основе уже созданного (и отлаженного!) надкласса. Использование этого механизма непосредственно связано с возможностью расслоения свойств предметной области, для которой разрабатываются программы, и определения отношений класс-подкласс. Заметим, что во многих областях определение таких отношений проблематично.

Еще одна отличительная особенность объектно-ориентированных языков заключается в организации взаимодействий объектов на основе "посылки сообщений". Появление таких механизмов взаимодействий фактически разрушает концепцию организации вычислительных процессов на ЭВМ, основанной на традиционной архитектуре фон Неймана. Эта архитектура, связанная с принципом хранимой программы и ее последовательным выполнением на одном (!) процессоре, оказывается мало приспособленной для моделирования ситуаций, когда несколько активных объектов функционируют одновременно и меняют свои состояния в результате обмена сообщениями. Разработка новых архитектурных решений, адекватных концепции "обмена сообщениями", свойственной объектно-ориентированному подходу, связана с созданием многопроцессорных конфигураций ЭВМ. В то же время обмен сообщениями между объектами может быть смоделирован и в обычных однопроцессорных ЭВМ с помощью хорошо известных средств, обеспечивающих логический параллелизм выполнения одновременных активностей: сопрограмм, процессов, планируемых программ, событийных взаимодействий и использования методов дискретно-событийного управления.

В целом объектно-ориентированный подход к разработке программ

интегрирует в себе как методы структуризации управления, так и структуризацию данных. При этом понятие объекта (которое формально так и не определено), строго говоря, не содержит в себе каких-то принципиальных отличий в этих разновидностях структуризации. Объектом может быть и константа, и переменная, и процедура, и процесс. В этом плане противопоставление категорий статического и динамического на концептуальном уровне теряет смысл. Объекты в программах "рождаются" и "умирают", меняют свое состояние, запускают и останавливают процессы, "убивают" и "возрождают" другие объекты, т. е. воспроизводят все оттенки явлений реального мира. Под объектом можно подразумевать некоторое абстрактное понятие, например, "уравнение" или "график функции"; понятие, имитирующее реальную систему или процесс: "теплообменник", "станок", "автомобиль". В этом плане объект - это сущность процесса или явления, которую способны выделить наш опыт, знания и интуиция.

Объектно-ориентированное программирование как и программирование вообще остается искусством, где интуиция играет очень большую роль. Но в отличие от обычного программирования этот подход предлагает новую палитру методов и инструментов для реализации Ваших представлений о процессах реального мира.

II. СПЕЦИФИКАЦИЯ ОБЪЕКТОВ НА ОСНОВЕ АБСТРАГИРОВАНИЯ

Понятие класса объектов.- Имманентные свойства класса.- Элемент хранения.- Агрегирование свойств.- Сигнатуры.- Представление объектов значениями.- Константы типа.- Перечислимый тип.- Множественный тип.

В объектно-ориентированном подходе к разработке программ центральным является понятие класса объектов. *Класс* определяется как множество объектов, обладающих внутренними (*имманентными*) свойствами, присущими любому объекту класса. Причем спецификация (определение) класса проводится путем определения его имманентных свойств, которые в этом плане играют роль *классообразующих признаков*. Например, свойство "иметь успеваемость" присуще всем обучаемым (студентам, школьникам, курсантам и пр.) и является классообразующим признаком класса ОБУЧАЕМЫЙ. В качестве других признаков этого класса могут использоваться, например, "возраст", "уровень интеллекта", "способность к запоминанию материала" и т.п. Совокупность подобных свойств и определяет класс "обучаемых".

Понятие свойства является, таким образом, первичным в определении класса. Спецификация класса никак не связана с заданием значений свойств, более того, применительно к классу говорить о таких значениях не имеет смысла - обладание значениями является прерогативой объекта. Определяя класс ОБУЧАЕМЫЙ, мы задаем конечное множество его свойств (успеваемость, возраст и пр.). Определяя объект класса (например, с фамилией Петров), мы должны определить значения этих свойств:

Успеваемость (Петрова):= Отличник; Возраст(Петрова):= 20.

Этот аспект определяет класс как понятие *экстенсиональное*, а объект класса - как *интенсиональное* понятие.

С другой стороны любой класс является множеством, состав объектов которого может меняться в динамике работы программы (обучаемые приходят и уходят, а класс остается). Класс как множество в любой момент времени характеризуется набором принадлежащих ему объектов и может быть задан перечислением (списком обучаемых): Петров, Иванов, Сидоров, Штернберг.

Эти два способа задания класса существуют независимо один от другого. Состав имманентных свойств статичен и определяет содержательный семантический аспект спецификации класса. Состав объектов класса динамичен и определяет ассоциативный (групповой) аспект класса. Семантический аспект реализуется в программировании с использованием абстрактных типов, ассоциативный - на основе использования множественных типов.

Независимость двух аспектов описания класса заключается в том, что существование каждого из них никак не связано с существованием другого. Если множество классообразующих признаков пусто, класс тем не менее может существовать как ассоциация некоторых формальных объектов (символов, знаков). В приведенном примере фамилия - всего лишь идентификатор объекта, она не входит в состав имманентных свойств и потому не несет никакой семантической нагрузки - мы могли бы заменить фамилию "Петров" строкой "XXXX", а фамилию "Штернберг" строкой "Бергштерн". Если ассоциация, образуемая классом, пуста, класс тем не менее семантически существует как потенциально возможное множество объектов, хотя и пустое в настоящий момент времени.

Пусть A является множеством объектов a, обладающих свойствами

P: A={a/P(A)}. Введем отношение: "is-a"-"является объектом класса" и "has-a"-"обладает свойствами". Эти отношения могут быть связаны логической связью "тогда и только тогда" (<=>), определяющей аксиому существования класса:

_V_ a: a is-a A(P) <=> a has-a P(A).

(Здесь _V_ - квантор общности).

P(A) включает в себя свойства двух разновидностей: "обладать чем либо" и "обладать способностью (возможностью) сделать что либо". Например, "обладать цветом" ("иметь цвет" или в дальнейшем просто "цвет"). Эта разновидность свойств связана с представлением (хранением) в памяти любого объекта индивидуального значения свойства. Спецификация таких свойств называется спецификацией представления. Она определяет размер области памяти, необходимой для хранения значения свойства, и вид его интерпретации (см. далее). Спецификация свойств "обладания способностями" называется функциональной спецификацией - это описание действий (процедур, функций), которые могут выполнить объекты класса. Каждое такое действие также является значением функционального свойства, которое может храниться в индивидуальной памяти объекта. Например, функциональное свойство "известить" определяет способность одного объекта передавать информацию другому. Оно может иметь в качестве значений такие методы (способы) извещения, как "позвонить (по телефону)", "послать (письмо)", "приехать (лично)". Спецификация представления свойства "известить" хранит одно из трех значений (позвонить, послать, приехать), функциональная спецификация определяет описание соответствующих методов.

Ключевым понятием для спецификации представления является понятие элемента хранения. Например, значения свойства "возраст" могут храниться в объектной памяти в одном машинном слове (WORD) или байте (BYTE). Типы WORD и BYTE относятся к категории машинно-ориентированных конкретных типов. Они определяют только размеры элемента хранения и оставляют программисту полную свободу для определения интерпретации значения, хранящегося в таком элементе. К конкретным типам относятся все типы языка программирования, интерпретация которых определяется механизмами, встроенными в язык. Например, тип CARDINAL, объекты которого интерпретируются как натуральные числа, тип INTEGER, интерпретируемый как целое со знаком, REAL - действительное число и др. Встроенность механизма интерпретации конкретных типов задает и размеры элементов хранения объектов соответствующих типов. Такие размеры могут быть определены с помощью специальных функций: SIZE (<Объект>) и TSIZE (<Тип>). Например, TSIZE (CARDINAL) = 2 (байта); SIZE (V) = 2 (байта) / V is-a CARDINAL. (Здесь / выполняет роль префикса условия). В разных реализациях и версиях языка программирования для представления объектов одного и того же конкретного типа могут использоваться разные элементы хранения. Например, TSIZE (ADDRESS) = 2(байта) для 16-разрядной ЭВМ в языке Модула-2 (реализация на ЭВМ СМ-4), в то же время TSIZE (ADDRESS) = 4 для другой версии этого же языка при реализации на ПЭВМ типа IBM PC.

Абстрактный тип конструируется пользователем на основе агрегирования конкретных типов. Такое агрегирование связано с объединением нескольких свойств объекта в систему классообразующих признаков, определяющих новый класс. Агрегирование реализует от-

ношение "состоит из" (con-of). Например, отношение A con-of (B,C), где A,B,C - свойства, может быть реализовано в языке программирования декларацией, связанной с определением хорошо известного типа записи:

```
TYPE A=RECORD
        <Имя свойства>: B;
        <Имя свойства>: C
     END
```

Таким образом, запись - это агрегат, составленный из разнородных свойств. Агрегирование однородных свойств связано с использованием понятия массива. Например, декларация

```
TYPE A = ARRAY [1:3] OF B
```

определяет агрегат A con-of(B,B,B). Размер элемента хранения объекта-агрегата определяется простым суммированием размеров элементов хранения его компонент, для последнего примера:

TSIZE (A) = 6 / TSIZE(B)=2.

Спецификация имманентных свойств типа "обладать способностью" (спецификация методов, действий) связана с использованием особой разновидности абстрагирования - определением сигнатур, реализуемых обычно процедурными типами. Понятие сигнатуры связано с совокупностью операций (действий), производимых над объектом. Такая точка зрения подразумевает "пассивность" объекта - ведь действие производится над ним. Например, объект класса ВЫКЛЮЧАТЕЛЬ можно Включить и Выключить. Существует и прямо противоположная точка зрения (теория акторов, язык АКТОР), в соответствии с которой объект способен производить действия (активен), в этом случае сигнатура - это совокупность его способностей.

Для определения сигнатур используются процедурные типы. В об-

щем случае любой процедурный тип определяет:

- класс возможных действий;
- классы объектов, над которыми могут быть произведены эти действия.

Например, спецификация

TYPE DST = PROCEDURE (VAR ВЫКЛЮЧАТЕЛЬ)

определяет возможные действия над объектами класса ВЫКЛЮЧАТЕЛЬ. Любая процедура, описанная в программном модуле и имеющая заголовок формально совпадающий с декларацией DST, может рассматриваться как объект класса DST. Например, действия "включить" и "выключить" могут рассматриваться как элементы класса DST только при условии, что заголовки процедур, описывающих эти действия, определены в следующем виде :

PROCEDURE Включить (VAR S: ВЫКЛЮЧАТЕЛЬ);

PROCEDURE Выключить (VAR S: ВЫКЛЮЧАТЕЛЬ);.

Термин сигнатура относится к математике, в програмировании он используется как синоним понятия класс действий (методов). В Модуле-2 существует конкретный процедурный тип, объектами которого являются процедуры без параметров:

TYPE PROC = PROCEDURE (); .

Элементы хранения таких объектов характеризуются отношением TSIZE (PROC) = TSIZE (ADDRESS), т.е. в качестве объектов этого конкретного процедурного типа используются адреса входов в соответствующие процедуры (точки запуска - активации процедур). Это отношение справедливо для любого процедурного типа. В этом смысле спецификация представления методов ничем не. отличается от

спецификации представления любых других непроцедурных классов.

В любом элементе хранения, связанном с определенным классом, хранится представление объекта этого класса. Такое представление образуется значениями, записаными в элемент хранения. Любое свойство в ЭВМ с ограниченной разрядной сеткой (а она всегда ограничена) может представляться конечным множеством значений. Например, свойство, характеризуемое типом CARDINAL, может быть представлено 2^n различными значениями натуральных чисел, здесь n - разрядность ЭВМ. Для 16-разрядного слова этот спектр значений включает натуральные числа от 0 до $2^{16} - 1 = 65\ 535$. Свойство, характеризуемое типом CHAR (литера), может быть представлено $2^8 = 256$ различными символами (из набора ASCII и графических символов), поскольку элемент хранения такого свойства имеет размер в один байт: TSIZE (CHAR) = 1.

Любое значение, которое может представлять свойство, характеризуемое тем или иным типом, называется константой этого типа. Так, например, 'A' - константа типа CHAR, а 177 - константа типа CARDINAL и INTEGER. Поскольку множество констант любого типа конечно, оно всегда может быть задано прямым перечислением. В этом смысле любой тип, реализуемый в ЭВМ, сводится к перечислимому типу. Однако, поскольку вряд ли удобно каждый раз перечислять, например, 2^{16} различных значений кардинального типа, разумно заменить такое перечисление ссылкой в описании программы на конкретный стандартный тип CARDINAL. Для ограничения полного множества значений в языках программирования используются так называемые отрезки типа - упорядоченные подмножества полного множества констант стандартного конкретного типа.

В контексте нашего пособия важно отметить, что представление

объекта значениями может быть сконструировано путем *именования* констант типа. Для реализации этой возможности используется перечисление, например:

```
TYPE Нота=(До, Ре, Ми, Фа, Соль, Ля, Си); .
```

Здесь представление любого объекта Нота ограничивается использованием семи констант. Поскольку имена таких констант назначает программист, подобное именование содержит элементы абстрагирования типа.

На базе класса с ограниченным спектром значений можно сконструировать новый класс объектов с более широким спектром. Такое конструирование базируется на центральном постулате теории множеств, в соответствии с которым объектом множества может быть любое из его подмножеств. Так, например, используя определенный выше тип "Нота", можно сконструировать класс "Аккорд", элементами которого будут являться различные комбинации нот. Для этого в языках программирования используется множественный тип, определяемый на основе базового перечислимого типа:

```
TYPE Аккорд = SET OF Нота; .
```

Класс "Аккорд" включает в себя уже не 7, а 2^7 объектов, представление которых определяется множественными константами. Среди них:

{ До } -"чистая" нота "До";

{ До, Ми } -аккорд, составленный из двух нот;

{ До..Си } -аккорд, включающий в себя всю октаву;

{} - аккорд "молчания", не содержащий ни одной ноты.

Элемент хранения объекта "Аккорд" должен допускать размещение

в нем 2^7 различных значений, следовательно, минимальным адресуемым элементом, пригодным для хранения аккордов, является байт:

TSIZE(Аккорд) =1.

Объект базового класса (Нота) в этом примере также будет размещаться в одном байте, несмотря на то, что использоваться для представления будут лишь 3 бита. Множественный тип, построенный на основе отрезка типа [0..15], образует стандартный тип

BITSET = SET OF [0..15].

Нетрудно заметить, что TSIZE(BITSET)=2 (байта). Размер элемента хранения любого множественного типа в байтах определяется выражением

N DIV 8 +(N MOD 8) DIV (N MOD 8).

Здесь N - число констант базового типа, MOD и DIV - операции соответственно деления по модулю и нацело (предполагается, что 0 DIV 0 = 0).

Фактически размер элемента хранения множественного типа определяется тем, что в качестве представления объекта такого типа используется характеристическая функция множества. Например, представление аккорда {До,Ми,Си} в байте будет выглядеть следующим образом:

Си	Ля	Соль	Фа	Ми	Ре	До	
?	1	0	0	0	1	0	1
7	6	5	4	3	2	1	0

(7-й бит не используется)

Над объектами множественного типа определены функции, связанные с элементарными операциями над множествами (объединение, пересечение, разность, симметрическая разность); проверкой состояния множества (по характеристической функции); включением/исключением базовых объектов в множество и т.п. Подробнее об

этом можно прочитать в руководстве по языку программирования.

Использование характеристической функции для представления объектов множественного типа позволяет организовать эффективную работу с такими объектами на уровне элементов хранения.

III. ИДЕНТИФИКАЦИЯ ОБЪЕКТОВ

Идентификация именованием.- Квалидент.- Дистанция доступа.- Оператор присоединения.- Индексирование.- Идентификация указанием.- Свободный и ограниченный указатели.- Тип ADDRESS.- Квалидент с постфиксом "^".

Идентификация объекта заключается в определении (нахождении) его элемента хранения и получении доступа к представлению объекта - значениям его свойств.

Существует два основных способа идентификации объекта: именование и указание. Именование заключается в назначении объекту определенного имени. Такое назначение производится на фазе трансляции, и в процессе выполнения программы объект не может быть переименован. Например, декларация

VAR A,B: Объект

определяет наличие в программе двух объектов с именами A и B соответственно, каждый из которых имеет индивидуальный элемент хранения. Обратиться к объекту A по имени B в надежде, что "он Вас услышит" невозможно, невозможны операции вида "Назвать объект A новым именем ВОВА". Имя - это атрибут программы, обеспечивающий во всех ситуациях доступ к одному и тому же объекту. Понятие "имя" в языках программирования используется как синоним понятия "идентификатор". В этом смысле процесс программирования

и выполнения программы является процессом изменения только представления объектов, но не правил их идентификации.

Именоваться могут и отдельные свойства объектов-агрегатов. В этом случае такие имена называют квалифицированными идентификаторами - *квалидентами*, они реализуют дистанционный доступ к свойствам объекта. Например,

```
TYPE Объект = RECORD
                B : Дата_рождения; П : Вес
              END;
VAR  A,B : Объект; .
```

Квалидент A.B откроет доступ к дате рождения объекта A, B.B - к дате рождения объекта B и т.д. *Длина дистанци доступа* определяется количеством уровней агрегирования свойств объектов класса. В этом примере Длина=1. Если уточнить свойство Дата_Рождения:

```
TYPE Дата_рождения = RECORD
                       Г: Год; М: Месяц; Д: День
                     END;
```

то квалидент, открывающий доступ к году рождения объекта A, имеет длину дистанции, равную 2: A.B.Г. Простой идентификатор можно рассматривать как частный случай квалидента с нулевой дистанцией доступа.

Дистанционный доступ может существенно увеличить время идентификации атрибутов объекта, в которых хранятся значения его свойств. Сократить это время можно используя *оператор присоединения*

```
WITH < Квалидент > DO < Присоединяемый фрагмент > END.
```

Такой оператор сокращает длину дистанции доступа к атрибутам объекта, идентифицируемого через <Квалидент>. Если число таких атрибутов в присоединяемом фрагменте велико, то использование оператора присоединения может существенно сократить время выполнения этого фрагмента программы.

Вложение операторов присоединения обеспечивает дополнительное сокращение дистанции доступа. Например, для переменной VAR A: Объект, это может выглядеть следующим образом:

```
WITH A DO
   <Работа со атрибутами объекта A через имена B и П>;
   WITH B DO
         <Работа со атрибутами свойства B объекта A
          через имена Г,M,D>
   END
END.
```

Имена объектов и их свойств могут дублировать друг друга. Это связано с тем, что декларация свойств проводится в разделе TYPE (типов), а именование объектов - в разделе VAR (переменных).

Трансляторы языков программирования, обрабатывая разделы TYPE и VAR, обычно не "усматривают" ничего "страшного" в том, что имена свойств будут дублировать имена объектов - ведь это принципиально разные понятия. Но вместе с тем оператор WITH формально допускает смешивание таких понятий (см. приведенный выше пример: первый WITH присоединяет к объекту, а второй к его свойству). Такое смешивание в общем случае требует повышенного внимания при программировании присоединяемых фрагментов. Например,

```
VAR A,B: Объект; C: Год;

BEGIN . . .

(1)  WITH A DO
         WITH B DO C:=Г END; B.B.Г:=C
     END . . .

(2)  WITH A DO
         WITH B DO C:=Г; B.Г:=C END
     END . . .

     WITH A DO
         WITH B DO C:=Г END
     END;
(3)
     WITH B DO
         WITH B DO Г:=C END
     END.
```

Все три фрагмента преследуют одну цель : обменять информацию о годах рождения объектов A и B . Первый фрагмент достигает этой цели, второй - нет. Почему ? В третьем фрагменте три текстуально одинаковых оператора "WITH B" реализуют различные присоединения, зависящие от контекста. Какие? Для того, чтобы избежать возможных семантических ошибок, обусловленных такой контекстной зависимостью оператора присоединения, следует либо использовать полные квалиденты (и жертвовать эффективностью программы), либо избегать дублирования имен объектов и атрибутов (свойств). Последнее во всех отношениях предпочтительнее.

При работе с массивами объектов и (или) массивами однородных свойств идентификация осуществляется на основе индексирования (нумерации). Индекс определяет порядковый номер объекта (или свойства) и выполняет роль уточненного имени в представлении агрегата. Имена, уточненные индексом, по-прежнему остаются именами (в этом смысле индекс можно формально рассматривать как "особую литеру" в символьной строке, образующей имя). Замечания, сделанные выше относительно дублирования имен объектов и

свойств, приобретают еще большее значение применительно к именованию с индексированием.

Доступ к объекту, идентифицируемому именем, которое уточнено индексом, реализуется на основе вычисления адреса соответствующего элемента хранения. Арифметическое выражение, реализующее такое вычисление, использует индекс как натуральное число.

Указание - второй основной способ идентификации - связано с использованием особых объектов, в представлении которых хранится как бы "стрелка", указывающая на идентифицируемый объект. Такой особый объект называется указателем или ссылкой. Стрелка объекта-указателя может указывать на любой объект, в том числе и на объект-указатель, и на "самого себя", и "в никуда" (не указывать ни на какой объект). Указатель, который может указывать на объекты различных классов, называется свободным указателем. Указатель, который может указывать только на объекты определенного класса, называется ограниченным указателем.

Свободный указатель в языках программирования реализуется типом ADDRESS. Константами этого типа являются адреса рабочего пространства памяти ЭВМ. Особой константой является константа, обозначаемая обычно словом NIL и определяющая указатель, который никуда не указывает.

Ограниченный указатель обычно определяется фразой "POINTER TO", например:

```
TYPE Стрелка = POINTER TO Объект;.
```

Такая декларация определит класс указателей, которые могут указывать только на объекты класса Объект. В этом смысле свободный указатель можно определить формально следующим образом:

```
TYPE ADDRESS = POINTER TO WORD.
```

В ранних версиях языков программирования

TSIZE (ADDRESS) = TSIZE (WORD) = 2 (байта).

При этом размер рабочего пространства адресов, определяемый мощностью множества констант типа ADDRESS, составлял для 16-разрядных ЭВМ 2^{16} = 65536 = 64*1024 = 64K. Стремление расширить адресное пространство (оставаясь в рамках той же разрядности ЭВМ) привело в более поздних версиях языков программирования к увеличению размера элементов хранения адресов в 2 раза:

TSIZE (ADDRESS) = TSIZE (ARRAY[1..2] OF WORD) = 4 (байта).

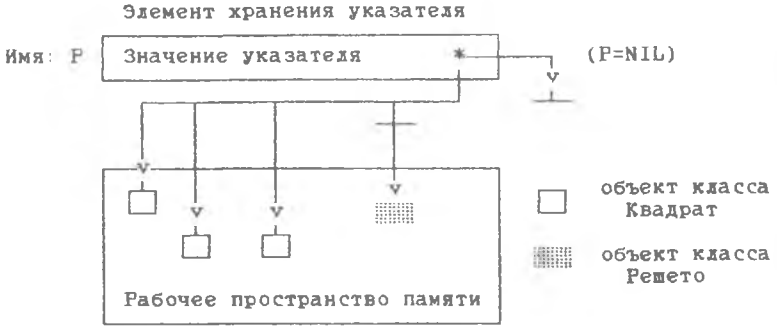
При этом ADDRESS стал интерпретироваться как структура:

```
TYPE  ADDRESS = RECORD
          SEGMENT, OFFSET: CARDINAL;
      END;
```

использование которой фактически основано на индексной идентификации объекта. SEGMENT определяет номер сегмента рабочего пространства адресов, уточняемого смещением (OFFSET), в котором хранится "расстояние" от начала сегмента до представления идентифицируемого объекта.

Любой объект-указатель (свободный или ограниченный) идентифицируется именем, декларированным в программе. Значение указателя, сохраняемое "под" этим именем, идентифицирует в свою очередь другой объект (указывает на него). Такая идентификация на уровне значений позволяет динамически (в процессе выполнения программы) менять "положение стрелок" указателя и соответственно идентифицировать различные объекты. "Чистое" именование не дает таких возможностей. Ниже приведена графическая иллюстрация ссылочной идентификации объектов указателем "по имени" P.

TYPE Квадрат: ... ; VAR P: POINTER TO Квадрат;

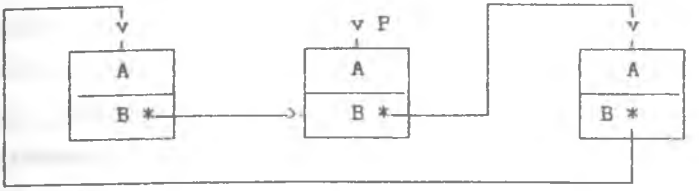


Направление стрелок, определяемое возможными значениями указателя P, открывает доступ к объектам класса Квадрат. Направление стрелки, указывающей на "решето", для P, декларированного как POINTER TO Квадрат, является недопустимым, стрелка P=NIL ни на что не указывает.

Идентификация объектов через ссылки открывает возможности организации динамически модифицируемых связанных структур. Объекты, из которых конструируются такие структуры, должны обладать свойством "Иметь связи с другими объектами", которое специфицируется как указатель. Например,

```
TYPE Элемент_Фигуры = RECORD
                        A : Квадрат;
                        B : POINTER TO Элемент_Фигуры
                      END.
```

Ниже приведена графическая иллюстрация одной из многих связанных структур - структуры Кольца, составленного из трех таких элементов.



VAR P: POINTER TO Элемент_Фигуры

На этой иллюстрации единственный указатель P последовательно (в направлении стрелок связей) открывает доступ ко всем элементам структуры Кольца. Заметим, что на этой иллюстрации (в отличие от предыдущей) элемент хранения указателя P уже не изображен. Просто рядом со стрелкой проставлено имя указателя - это обычный прием для графических иллюстраций представления связанных структур.

Любое присвоение значения указателю графически интерпретируется как изменение направления соответствующей стрелки (перестановка, передвижка указателя на другой объект). Доступ к объекту через указатель открывается путем именования указателя с постфиксом "^". Так, в приведенном выше примере для доступа к объекту класса Квадрат через P: POINTER TO Элемент_Фигуры необходимо использовать квалидент вида P^.A. В нем "зашифрована" следующая последовательность доступа:

P - доступ к указателю, идентифицирующему Элемент_Фигуры;

P^ - доступ к структуре Элемента, на которую указывает P;

P^. - доступ к атрибутам (компонентам) этой структуры;

P^.A - доступ к атрибуту Квадрат.

Каждый из подобных квалидентов открывает доступ к "своему"

уникальному объекту (или атрибуту). Нетрудно заметить, что для этого примера (и в общем случае)

SIZE (P) # SIZE (P^) # SIZE (P^.A).

Кстати, чему равно SIZE (P^) для этого примера?

Роль постфикса "^" (стрелки) заключается в "открытии" доступа к объекту через значение указывающей на него ссылки. Иногда эту операцию образно называют "раскрытием ссылки". Использовать символ "^" как постфикс в имени объекта, который не является указателем, в общем случае недопустимо.

Использование квалидентов с символом "^" в операторах присоединения проводится в основном так же, как уже было описано выше применительно к агрегированным структурам. Здесь следует помнить, что любое присоединение целесообразно с двух точек зрения:

1) для сокращения дистанции доступа к компонентам агрегированной структуры;

2) для повышения наглядности, выразительности и структурности программы.

Для случая P: POINTER TO Элемент_Фигуры использование оператора

WITH P^ DO < Присоединяемый фрагмент > END

реализует присоединение к Элементу_Фигуры, размещенному в памяти "под" P, а оператор

WITH P DO < Присоединяемый фрагмент > END

может реализовать присоединение только (!) к атрибутам самого указателя (т.е. полям SEGMENT и OFFSET) и не имеет никакого смысла в плане присоединения к Элементу_Фигуры. В этой связи также отметим, что любое присоединение, декларированное соответствующим оператором WITH, выполняется после того, как определено

значение присоединяющего квалидента, т.е. до "входа" в присоединяемый фрагмент. Поэтому любое изменение значения присоединяющего указателя внутри присоединяемого фрагмента не изменит уже созданного присоединения и неизбежно нарушит логику выполнения этого фрагмента. Приведем еще пример:

```
VAR P: POINTER TO Квадрат;
BEGIN ... P:= ...; (* Установка P на квадрат *)
 WITH P^ DO ...
  (* Работа с квадратом, на который указывает P *);
     P:= ...; (* Установка P на новый квадрат *)
          ... (* Работа с новым квадратом *)
END.
```

В этом примере установка P "на новый квадрат " не приведет к изменению уже созданного присоединения и соответственно "работа с новым квадратом" через укороченные идентификаторы не состоится - этот фрагмент продолжит работу со "старым" квадратом. Незнание этого обстоятельства может служить источником многих трудно идентифицируемых ошибок, возникающих только при идентификации объектов методом указания.

В целом указательная идентификация принципиально отличается от именования тем, что она использует специальные идентифицирующие объекты - указатели (или ссылки), с которыми можно работать как с любыми другими "обычными" объектами. Это существенно расширяет возможности "чистого" именования и позволяет реализовать динамическую идентификацию различных объектов через один и тот же указатель, идентифицируемый единственным присвоенным ему име-

нем.

IV. ИНТЕРПРЕТАЦИЯ ОБЪЕКТОВ

Полиморфизм. - Совместимость типов. - Функции преобразования и приведения типов. - Записи с вариантами. - Наследование свойств. - Определение " наложением ". - Самоинтерпретируемый объект.

Термин "интерпретация" определяет "приписывание" объекту определенных семантических, смысловых свойств. Например, символ "I", интерпретируемый как "Римская_Цифра", будет ассоциироваться с объектом определенной системы счисления, характеризуемой особыми свойствами этой системы.

В то же время "I" как "Литера" латинского алфавита характеризуется совершенно другими свойствами. "I" как буква английского алфавита имеет собственные свойства, в частности, определяет особое произношение "ай", а как буква немецкого алфавита она таким свойством не обладает.

Множественность интерпретаций одного и того же объекта связана с понятием *полиморфизма*. С проявлением полиморфных интерпретаций объектов мы сталкиваемся буквально на каждом шагу - это и многозначность многих оборотов речи (фразовых структур) и многоцелевое использование объекта (вспомните повесть М.Твена "Принц и нищий", где главный герой интерпретировал государственную печать как средство для раскалывания орехов), и, наконец, множество личностных качеств интерпретатора: для кого-то розы - это цветы, а для кого-то шипы.

В программировании объект как данность полностью определяется

понятием элемента хранения, уже использованным в предыдущих главах. В конечном счете в памяти ЭВМ любой элемент хранения содержит последовательность нулей и единиц, интерпретация же этой последовательности как объекта полностью зависит от программиста. Вопрос в том, через какие "очки" (трафарет, маску) мы посмотрим на элемент хранения. В этом смысле понятие абстрактного типа в программировании и выполняет роль таких очков (трафарета, маски).

Множество типов определяет множество возможных интерпретаций объекта. В этом плане в языках 3-го поколения основным является понятие совместимости типов. Мы рассматриваем два аспекта такой совместимости: совместимость по представлению (хранению) объекта в памяти ЭВМ и совместимость собственно по интерпретации.

Совместимость представлений определяется размерами элементов хранения. Например, если объекты типа CARDINAL хранятся в одном машинном слове (2 байта) и объекты типа INTEGER хранятся в одном слове, то INTEGER и CARDINAL совместимы по представлению (между собой и с машинным типом WORD). Аналогично совместимы по представлению CHAR и BYTE; WORD и ARRAY [1..2] OF BYTE и т.д.

Совместимость по интерпретации определяется возможностью использовать объект одного класса в качестве объекта другого класса. Например, ложку в качестве вилки. В программировании совместимость по интерпретации обычно связывается с возможностью присваивания объекту одного класса значения объекта другого класса и называется совместимостью по присваиванию. Пример такой совместимости:

```
VAR A: CARDINAL; B: INTEGER;      BEGIN ... A:=B .
```

Совместимость по присваиванию обычно подразумевает совмести-

мость представлений объектов.

Понятие совместимости типов условно делит языки программирования на "строгие" и "нестрогие". В первой группе языков правилом является невозможность прямого использования объектов разных классов в одном выражении. Такое выражение необходимо конструировать на основе специальных функций преобразования типов, приведения типов и специальных методов совмещения типов. Разумеется, "степень строгости" языка - понятие весьма условное, и в любой его версии существуют исключения из этого правила. "Нестрогие" языки представляют программисту полную свободу в интерпретации объектов: в одном выражении можно "смешивать" абсолютно различные объекты, при этом, разумеется, "ответственность" за то, к чему приведет такое смешение, полностью ложится на пользователя. Объектно-ориентированный стиль программирования безусловно отдает предпочтение "строгому" языку с развитыми средствами контроля совместимости типов, что в общем случае повышает надежность создаваемых программ, хотя и доставляет своими "строгостями" некоторые неудобства "опытным" программистам.

Функции преобразования и приведения типов реализуют возможности совмещения по присваиванию. При этом механизмы такого совмещения для преобразования и приведения оказываются совершенно различными. Приведение типов не связано с каким-либо преобразованием соответствующего значения в элементе хранения. Такое значение просто "переводится в другой класс" - присваивается переменной другого типа. Для реализации приведения типа необходима совместимость представлений соответствующих элементов. Например:

```
VAR A: INTEGER; B: CARDINAL;
BEGIN  A:=-3; B:= CARDINAL (A); ...
```

Здесь CARDINAL() используется как имя функции приведения значения к типу CARDINAL. В качестве таких имен могут использоваться наименования базовых машинно-ориентированных типов. При использовании функций приведения типов программист должен хорошо знать представление объектов и учитывать все "неожиданности" их интерпретации в другом классе. (Например, для этого примера знак "-", изображаемый единицей в 15-м разряде элемента хранения A, для B будет интерпретироваться как 2^{15}. Соответственно после приведения B = $2^{15} + 2^1 + 2^0 = 32771$). Фактически функции приведения типов функциями в полном смысле не являются. Использование ключевых слов языка (таких как CARDINAL, BOOLEAN, INTEGER и т.д.), определяющих имена базовых типов, в контексте BEGIN ... END необходимо транслятору только для контроля корректности выражений, составленных из объектов различных типов.

Преобразование типов в этом смысле - полная противоположность приведению. Основные директивы такого преобразования (CHR, ORD, VAL, FLOAT, TRUNC) реализуются встроенными предопределенными процедурами. Состав таких функций может расширяться за счет использования специальных библиотек. Три первые функции преобразования относятся к работе с перечислимыми типами и подробно описаны в соответствующей литературе. Здесь мы подчеркнем лишь один аспект использования функции VAL. Поскольку, как уже отмечалось, большинство базовых типов реализуются в ЭВМ на основе перечисления, VAL может работать с ними как с перечислимыми Общая синтаксическая структура вызова VAL при этом имеет следующий вид:

<Имя переменной типа B> :=

VAL (<Имя типа B>, <Объект класса CARDINAL>).

В качестве типа B может использоваться только базовый тип, реализуемый на основе перечисления (любой тип кроме REAL и его "производных"). Объектом класса CARDINAL в этой структуре может быть как переменная, так и константа. Например,

```
VAR c: CARDINAL; b: BYTE; i: INTEGER; ch: CHAR;
   BEGIN ch := 'A';  c := 32771;
         i := INTEGER ( c );                    (*1*)
         i := VAL ( INTEGER, c );               (*2*)
         b := BYTE ( ch );                      (*3*)
         b := VAL ( BYTE, ORD(ch) );            (*4*)
         b := VAL ( BYTE, c );                  (*5*)
```

К одинаковым ли результатам приведут операции (1) и (2)? (3) и (4)? К какому результату приведет операция (5)? Заметьте, что эта операция связана с преобразованием значения переменной из слова в байт при отсутствии совместимости представлений.

Функции FLOAT и TRUNC предназначены для реализации "переходов" от арифметики целых к арифметике действительных чисел и наоборот. Они подробно описаны в учебниках по программированию.

Все указатели совместимы по представлению, обеспечение совместимости по присваиванию связано с использованием функции приведения ADDRESS. Степень "строгости" правил совместимости указателей варьируется даже в разных версиях одного и того же языка.

Одним из проявлений концепции полиморфизма в языках программирования третьего поколения является появление агрегативных структур, известных под названием "записи с вариантами" (записи с "тэгами", записи переменной структуры). В такие структуры вво-

дятся специальные выделяющие (выбирающие) свойства, определяющие интерпретацию объекта. Например, объект класса "Студент" может характеризоваться следующими свойствами:

- успеваемостью,
- принадлежностью к группе,
- фамилией,
- размером получаемой стипендии.

Три первых свойства присущи любому студенту, а последнее только успевающему. Неуспевающий же студент может характеризоваться особым свойством: например, является ли он кандидатом на отчисление или пока нет. Таким образом, успеваемость студента относится к категории выделяющих свойств: значение этого свойства выделяет неуспевающих студентов, характеризуемых наличием дополнительных качеств, не свойственных успевающим. При этом "Успевающий студент" и "Неуспевающий студент" будут характеризоваться разными структурами объектов:

```
TYPE  Успеваемость = ( Отл, Хор, Уд, Неуд );
      Успевающий_Студент = RECORD
          FAM : Фамилия;
           GR : Номер_Группы;
           SB : Успеваемость;
           ST : REAL;  (* Размер стипендии *)
      END;


      Неуспевающий_Студент = RECORD
          FAM : Фамилия;
           GR : Номер_Группы;
```

```
        SB : Успеваемость;
        Кандидат_На_Отчисление : ( Да, Нет )
    END.
```

Нетрудно заметить, что в этих структурах есть общие части, а отличия связаны только с последним качеством (атрибутом, полем). Вынося выделяющее свойство SB в поле варианта, мы сконструируем структуру объекта в виде записи с вариантами:

```
TYPE  Студент = RECORD
        FAM : Фамилия;
        GR : Номер_Группы;
        CASE SB : Успеваемость OF
            Неуд : Кандидат_На_Отчисление : ( Да, Нет ) |
            Отл, Хор, Уд : ST : REAL
        END
    END.
```

Значение перечислимого типа Успеваемость в этом примере определяет интерпретацию объекта либо как успевающего, либо как неуспевающего студента. Таким образом полиморфизм структуры записи с вариантами заключается в возможности ее интерпретации на альтернативной основе.

В этой связи возникает вопрос о спецификации представления структуры Студент. Она содержит постоянную часть

TSIZE (Фамилия) + SIZE (GR) + TSIZE (Успеваемость)

и переменную (набор альтернатив), размер которой определяется значением SB. Либо это байт (в случае SB = Неуд)

SIZE (Кандидат_На_Отчисление) = 1; ,

либо двойное слово (в случае SB # Неуд) SIZE(ST)=4. Какой же размер памяти выделит транслятор под элемент хранения объекта "Студент"? Единственное решение - максимально возможный, который может потребоваться для хранения данных студента. Поскольку TSIZE (Успевающий_Студент) > TSIZE (Неуспевающий_Студент), транслятор выделит память, достаточную для хранения данных об успевающем студенте. Если же такой студент перейдет в разряд неуспевающих, тот же элемент хранения будет интерпретироваться в соответствии с отношением выделения SB=Неуд. При этом из четырех байт, выделенных транслятором под ST в расчете на успевающего студента, три последних просто не будут использоваться, а первый байт будет интерпретироваться как сохраняющий значение свойства Кандидат_На_Отчисление.

Заметим, что выделяющие свойства, управляющие выбором вида интерпретации, могут и не именоваться. В таких случаях вид альтернативной интерпретации определяется не выделяющим свойством, а фактическим использованием имени поля при обращении к объекту. Например:

```
TYPE  Студент = RECORD
          FAM : Фамилия; GR : Номер_Группы;
          CASE  : Успеваемость OF
              Неуд : Кандидат_На_Отчисление : ( Да, Нет ) ;
              Отл, Хор, Уд : ST : REAL
          END
      END.
```

Пусть VAR V: Студент. При этом в элементе хранения для V выделяющее поле вообще отсутствует. постоянная часть имеет размер TSIZE(Фамилия)+SIZE(GR), а альтернативная имеет размер

max {SIZE(Кандидат_На_Отчисление), SIZE(ST)}.

Обращение к объекту через квалидент V.Кандидат_На_Отчисление приведет к интерпретации альтернативной части в соответствии с перечислимым типом (Да, Нет), а обращение V.ST - к интерпретации той же части в соответствии с типом REAL. Заметим, что такая альтернативная интерпретация может оказаться весьма "неустойчивой", связанной с возможностями возникновения дополнительных ошибок. Наличие в структуре вариантной части последнего примера деклараций типа выделяющего свойства (Успеваемость), а также констант этого типа (Неуд,Отл,Хор,Уд), строго говоря, обусловлено только одним обстоятельством: стремлением сохранить общую синтаксическую структуру записи с вариантами. В смысле корректной интерпретации эти декларации не имеют никакого значения - ведь проверить значение несуществующего выделяющего свойства невозможно!

В общем случае независимо от того, именуется поле тэга или нет, записи с вариантами ограничивают набор возможных видов интерпретации объектов на альтернативной основе. В этом и состоит *регламентирующая роль* этих структур в полиморфной альтернативной интерпретации объектов.

Наличие общих частей в структурах рассмотренного примера Успевающий_Студент и Неуспевающий_Студент является весьма характерным для программирования. В этом смысле записи с вариантами можно рассматривать как форму *лаконичного описания типов*, позволяющую избавиться от повторов в описании свойств объектов. В объектно-ориентированных языках существует дополнительная возможность такой лаконизации, определяющая полиморфную интерпретацию объектов не на альтернативной основе, а на основе расширения

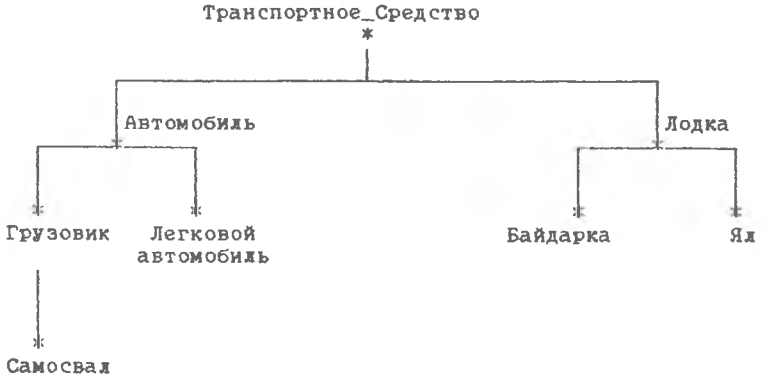
свойств. Эта возможность связана с механизмом наследования свойств.

Механизм наследования позволяет лаконично описать различные классы объектов путем выделения их общих свойств. Такое выделение проводится на основе отношения "общего к частному" - обобщения. Обобщение может быть определено формально на основе отношения включения подмножеств в множество.

Пусть A - класс объектов с имманентными свойствами P(A): A = {a/P(A)}, а B = {b/P(B)}. Если P(A) IN P(B) (P(A) является подмножеством P(B), IN - отношение включения), то A "обобщает" B (A*>B, "*>" - отношение обобщения). В отношении (A*>B) A является надклассом. B - подклассом. при этом любой объект класса B характеризуется наследуемыми свойствами P(A) и приобретенными P(B)-P(A). Например, любой автомобиль обладает свойствами транспортного средства и имеет некоторые особенные "автомобильные" свойства, которыми не обладает такое транспортное средство, как, например, лодка. В этом смысле

Транспортное_Средство *> Автомобиль, Лодка.

Причем P(Автомобиль)^P(Лодка) = P(Транспортное_Средство). (Здесь символ "^" используется как "пересечение множеств"). Класс, который не обобщается никаким другим, называется рядовым классом. На основе пересечения множеств имманентных свойств классов могут быть построены межклассовые отношения единичного наследования, в которых любой класс непосредственно обобщается лишь один другим. Например,



Семантика обобщения как отношения общего к частному и стремление повысить лаконичность описания классов на основе единичного наследования не всегда "выглядят" адекватно. Например,

```
TYPE Узел = RECORD
                A: Болт; B: Гайка;
            END;
```

Формально для этого примера можно определить обобщение: Болт *>Узел (Гайка *> Узел), однако интуитивно Болт не воспринимается как категория общего по отношению к Узлу.

Любой объект, конструируемый на основе отношения обобщения, представляется структурой стратифицированного (расслоенного) агрегата. Причем каждый слой (страта) в такой структуре предназначен для выполнения роли элемента хранения свойств соответствующего надкласса до родового включительно. Например, любой объект класса "Ял" (см. схему выше) будет определяться структурой:

```
TYPE Структура Яла = RECORD
                       A: Транспортное_Средство;
                       B: Лодка;
                       C: Ял;
                     END; .
```

Интерпретация Яла как транспортного средства связана только с использованием слоя A в элементе хранения. Интерпретация Яла как лодки - с использованием двух слоев: A и B, и, наконец, интерпретация Яла как особого вида лодки связана с использованием всех трех слоев: A,B,C. Декларация вида "Структура_Яла" в объектно-ориентированном языке заменяется отношением

Ял <* Лодка <* Транспортное_Средство.

Такая декларация определяет три возможные интерпретации объекта на разных уровнях обобщения (расширения свойств).

Еще раз подчеркнем, что между двумя рассмотренными видами полиморфной интерпретации объектов (записи с вариантами и наследование свойств) существует принципиальное различие: записи с вариантами реализуют полиморфную интерпретацию на альтернативной основе, а механизм наследованиния - на основе расширения свойств классов.

В практике использования методов программирования, ориентированного на объекты, широко распространен так называемый метод определения объектов "наложением" (соответствием). Этот метод может быть реализован разными способами, мы его рассмотрим на примерах, используя концепцию типа как "трафарета" (маски), определяющего вид интерпретации объекта "под маской". Конструируя средствами языка различные "маски", программист получает возможности полиморфной интерпретации объекта.

Пример1.

```
TYPE POINT = RECORD X,Y: INTEGER END;
     Point = RECORD Y,X: INTEGER END;
 VAR   A: ARRAY[1..2] OF WORD;
       P: POINTER TO POINT;
       p: POINTER TO Point;
       X,Y: INTEGER;
BEGIN X:=1; Y:=5;
      P:=ADR(A);                          (*1*)
        P^.X:=X; P^.Y:=Y;                 (*2*)
      p:=ADDRESS(P);                      (*3*)
        X:=p^.X; Y:=p^.Y                  (*4*)
```

Этот пример реализует "трансформацию" объекта-точки с декартовыми координататами (1,5) в объект-точку с координатами (5,1). В программе задан элемент хранения A размером в два слова, "маска" POINT, "привязанная" к указателю P, и "маска" Point, связанная с ограниченным указателем p. Операция (1) связана с "наложением" маски POINT на элемент хранения A и записью "через трафарет" значений координат точки в область памяти A. Операция (3) связана с наложением на ту же область памяти маски (трафарета) Point и чтением координат точки через новый трафарет. Таким образом, один и тот же объект, размещенный в A, интерпретируется в этом примере двояко: как точка с координатами (1,5) и симметричная ей точка с координатами (5,1). Заметим, что реально никакого преобразования координат не происходит, - все определяетсся структурой трафарета - маски, через которую мы смотрим на объект. (Рассматривая этот пример, ответьте на вопрос, почему для записи операторов (2) и (4) не используется

присоединение?)

Поскольку множественность интерпретаций объекта определяется множеством масок, которые могут накладываться на одну и ту же область памяти, использование метода наложения связано с контролем размеров таких масок, соответствия их размерам элементов хранения и т.д. "Выход" маски за пределы элемента хранения интерпретируемого объекта чреват непредсказуемыми ошибками (работа с "чужой" областью памяти). Наложение нескольких масок на один и тот же объект желательно выполнять по адресу элемента хранения объекта без дополнительных смещений "внутрь" структуры объекта. Если несколько разных масок частично совместны (имеют части с идентичными атрибутами, одинаково интерпретируемые части), желательно общие идентичные части располагать в начале маски (вверху), а не в середине или в конце (внизу). Эти простые рекомендации помогают избежать многих ошибок, связанных с полиморфной интерпретацией объекта. (Заметим, что такие ошибки имеют свойства скрытого "проявления", очень трудно обнаруживаются и идентифицируются).

Во многих прикладных задачах метод наложения связан с использованием масок, определяемых структурами различных массивов. Например, задан массив кардинальных чисел и требуется его "трансформировать" в массив символов. Наложение в этом случае является наиболее "естественным" методом такой трансформации:

```
VAR C: ARRAY [1..100] OF CARDINAL;
    P: POINTER TO ARRAY [1..200] OF CHAR;
    CH: ARRAY [1..200] OF CHAR;
BEGIN
  P := ADR(C); FOR I:=1 TO 200 DO CH[I]:=P^[I] END;...
```

Такие задачи связаны, как правило, с перекодировкой, преобразованием, трансформацией и т.п. больших массивов. Успех использования метода наложения здесь полностью определяется тем, удастся ли подобрать адекватную структуру маски-трафарета. Если удастся, то подобные преобразования могут быть выполнены очень просто, без использования специальных вычислений, связанных с различными форматами хранения данных, и неизменно сопутствующей им адресной арифметики. Попутно заметим, что использование метода наложения может помочь "обойти" многие ограничения, связанные с языком программирования. Например, используя наложение при интерпретации объектов, размещаемых в классе динамической памяти, можно "обойти" ограничения, связанные со статическими (константно - определяемыми) размерами массивов.

В заключение этой главы остановимся на самоинтерпретируемых объектах. Возможности самоинтерпретации связаны с использованием объектов процедурного типа, объектов-действий. Эта разновидность объектов сравнительно мало используется в технике "повседневного" программирования, в методологии же объектно-ориентированного подхода им отводится особая роль, роль активных объектов - *акторов*, определяющих динамику параллельно развивающихся процессов интерпретации.

Процедурный тип (или сигнатура, см. разд. II) определяет множество возможных действий, видов активности. Например,

```
TYPE Действие = PROCEDURE (Станок);
```

определяет сигнатуру для класса Станок. Пусть множество действий над Станком ограничивается двумя:

```
PROCEDURE Включить (С: Станок);
PROCEDURE Выключить (С: Станок); .
```

Декларация VAR D: Действие определяет объект класса Действие. Такой объект может хранить потенциально возможное действие над Станком (т.е. "помнить", что нужно сделать) и (в подходящее время) активизироваться (самоинтерпретироваться) по отношению к станку:

```
VAR D: Действие; C: Станок;
BEGIN...
    D:=Включить;...
    D(C);... D:= Выключить;... D(C); .
```

Операторы D(C) в этом фрагменте определяют самоинтерпретацию объекта D в отношении объекта C, а операторы присваивания - определение объекта D потенциально возможным действием. Образно говоря, операторы присваивания здесь "взводят курок" D, а когда D "выстрелит" и какой будет эффект от этого "выстрела" (включает он станок C или выключает) определяется в общем случае логикой программы. Использование в программе переменных класса Действие аналогично наличию множества взведенных курков, при этом отдельные "выстрелы" превращаются в треск автоматных очередей - самоинтерпретаций. Учитывая, что любое действие, связанное с такой самоинтерпретацией, может переопределить объекты-действия, логика выполнения подобных программ становится весьма запутанной. Основное применение этого механизма - моделирование сложных систем.

V. СОЗДАНИЕ / УНИЧТОЖЕНИЕ ОБЪЕКТОВ

"Время жизни" объекта. - Классы памяти. - Управление динамической памятью. - фрагментация. - Проблемы "висячих" ссылок и мусора. - Автоматическая память. - Локальная среда. - Активации

объекта.

Объекты, существующие в программе, делятся на две категории: *статические* и *динамические*. Эти категории определяются по-разному: на основе изменения состояния объектов модели и на основе "*времени жизни*" объектов. Первое определение предполагает, что любой объект, изменяющий свое состояние в процессе работы программы, является динамическим. В этом отношении, строго говоря, статическими объектами являются только константы, все объекты-переменные могут считаться динамическими. Второе определение предполагает возможность временного существования объектов, возможности создания и уничтожения объектов. В этом смысле объекты, время существования которых равно времени выполнения программы, расцениваются как постоянно существующие (статические), объекты же, время существования (жизни) которых меньше времени выполнения программы - как динамические. Второе определение касается объектов, которые идентифицируются только через указатели. Объекты, идентифицированные именем, в этом отношении всегда должны расцениваться как статические, поскольку их "создание" подготавливается транслятором и ассоциация между именем и элементом хранения объекта существует до окончания времени работы программы.

Создание объекта следует интерпретировать как выделение памяти под его элемент хранения. Такая интерпретация подразумевает разделение всего рабочего пространства памяти ЭВМ на две категории, два класса - статическую память и динамическую. Первый класс памяти, как следует из этого контекста, полностью находится под управлением транслятора и распределяется под статические

объекты, существующие в системе постоянно. Например, декларация

```
VAR A: POINTER TO CARDINAL;
    B: CARDINAL;
```

сообщает транслятору о необходимости "зарезервировать" в классе статической памяти два слова под элемент хранения объекта с именем A и одно слово под элемент хранения объекта с именем B.

Динамическая память предназначается для создания временно существующих объектов. Этот класс памяти имеет две разновидности: собственно динамическую и автоматическую. Собственно динамическая память (в отличие от статической) полностью находится в распоряжении программиста: по его директивам происходит выделение элементов хранения (создание объектов) и возврат ранее выделенных элементов в "зону" свободной памяти (пул "свободных" элементов), что в этом смысле равносильно "уничтожению" объекта.

Автоматическая память - особая разновидность динамической, которая также управляется директивами программиста, связанными с интерпретацией активных объектов (переменных процедурных типов). В этом смысле две разновидности динамической памяти делят этот класс памяти на два подкласса: память для интерпретации пассивных объектов (собственно динамическая) и память для интерпретации активных объектов (автоматическая). Несмотря на общность класса (динамическая память), распределение памяти в этих подклассах основано на разных принципах и реализуется совершенно разными алгоритмами.

Управление динамической памятью для пассивных объектов (в дальнейшем просто динамической памятью) реализуется на основе двух основных процедур (обычно импортируемых из системного модуля):

```
PROCEDURE ALLOCATE (VAR A: ADDRESS; N: CARDINAL);
PROCEDURE DEALLOCATE (VAR A: ADDRESS; N: CARDINAL); .
```

Здесь A - свободный указатель, который укажет на выделенную область памяти (элемент хранения размером N байт) при вызове ALLOCATE и получит значение NIL (т.е. никуда не будет указывать) при освобождении этой области "из-под" A путем вызова DEALLOCATE.

Использование ограниченных указателей делает во многих отношениях целесообразным использование специальных вызовов: NEW(P) и DISPOSE(P), где VAR P: POINTER TO <Объект>. (NEW и DISPOSE - псевдопроцедуры, их вызовы транслируются в вызовы ALLOCATE и DEALLOCATE соответственно). Использование NEW и DISPOSE позволяет избежать многих семантических ошибок, связанных с различными значениями N в последовательности вызовов ALLOCATE...DEALLOCATE, определяющей создание/уничтожение одного и того же объекта.

В целом последовательность вызовов NEW...DISPOSE (или соответственно ALLOCATE...DEALLOCATE), в общем случае полностью определяемая логикой программиста, порождает ряд проблем, связанных с организацией и распределением свободного пространства динамической памяти. Одной из таких проблем является проблема **фрагментации**. Эффект фрагментации заключается в том, что рабочая область динамической памяти "дробится" на части - фрагменты различной длины. Какие-то из них "заняты" - используются программистом под элементы хранения его объектов, какие-то "свободны", причем характер чередования свободных и занятых фрагментов в общем случае может быть совершенно произвольным. Любой запрос программиста на создание нового объекта приводит к тому, что система управления динамической памятью "подбирает" ему фраг-

мент, подходящий по размерам. Правила такого подбора могут быть различны, но общая закономерность одна: такой фрагмент должен иметь размер, не меньший, чем запрашиваемый программистом. Если подходящий фрагмент имеет больший размер, чем требуется, в прикладную программу будет отдана его часть, которая теперь будет рассматриваться системой как занятый фрагмент, а остаток останется в свободной зоне в качестве свободного фрагмента. При этом проблема фрагментации заключается в том, что эффект "дробления" может привести к тому, что в свободной зоне будет находиться множество "маленьких" разрозненных свободных фрагментов, в совокупности составляющих достаточный объем. Тем не менее, несмотря на такой объем, запрос программиста на новый элемент памяти может получить отказ по причине отсутствия целого подходящего элемента. Ниже приведен фрагмент программы и схема распределения динамической памяти, иллюстрирующие эффект фрагментации. При этом для простоты предполагается, что общий объем динамической памяти составляет 20 байт.

```
TYPE Треугольник = POINTER TO Фигура_1
     Фигура_1 = RECORD
                  Сторона_1, Сторона_2, Сторона_3:CARDINAL
     END;
     Четырехугольник = POINTER TO Фигура_2;
      Фигура_2 = RECORD
                  Сторона_1, Сторона_2, Сторона_3, Сторона_4:
                                                    CARDINAL
       END
VAR T1, T2: Треугольник; M1, M2: Четырехугольник;
```

```
BEGIN   NEW(T1);... NEW(M1);... NEW(T2);...
        DISPOSE(T1);... DISPOSE(T2); NEW(M2);...
```

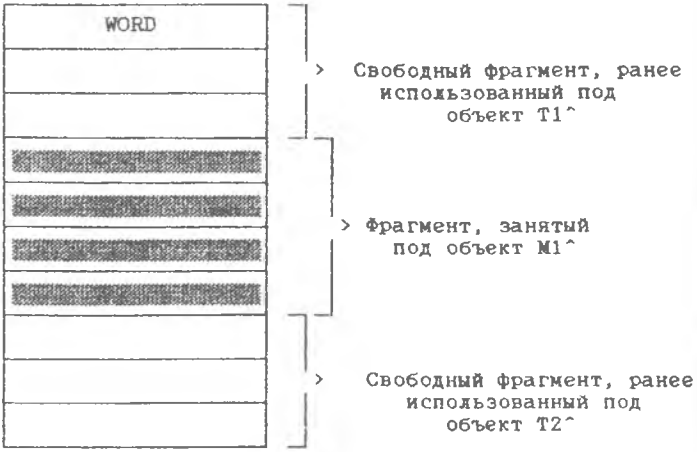


Иллюстрация построена для момента обработки запроса NEW(M2). В этот момент времени в динамической памяти имеются два свободных фрагмента общим объемом шесть слов, которых достаточно для выполнения запроса на выделение элемента хранения под объект M2^ (т.е. для объекта, на который будет указывать M2), однако фрагментация не позволяет системе выделить память под объект M2^.

Система управления динамической памятью ведет специальный список свободных фрагментов - пул памяти. При возвращении какого-либо элемента хранения, используемого в прикладной программе, в пул свободной памяти может быть реализовано "склеивание" соседних свободных фрагментов в один фрагмент большего объема. Например, если в предыдущей программе изменить последовательность обращений к динамической памяти на приведенную ниже,

то описанного выше отказа по памяти не произойдет:

```
BEGIN NEW(T1);...NEW(T2);...NEW(M1);...
      DISPOSE(T1);...DISPOSE(T2);... NEW(M2);...
```

Здесь при обработке запроса NEW(M2) в пуле динамической памяти будет находиться один свободный фрагмент объема шесть слов, образованный "склеиванием" элементов T1^ и T2^, выполненным при обработке запроса DISPOSE(T2). В общем случае вопросы эффективной реализации управления динамической памятью, обеспечивающей минимум отказов при ограниченном объеме, составляют отдельную проблему. Здесь мы только заметим, что с организацией выделения "первого подходящего" фрагмента памяти в программировании связывают такие термины как "хип" или "куча", относящиеся скорее к профессиональному жаргону, чем к научно-методической терминологии. Тем не менее эти термины довольно образно характеризуют принципы организации динамической памяти.

Организация корректной последовательности запросов связана, кроме того, как минимум еще с двумя проблемами. На том же жаргоне их называют проблемы "висячих ссылок" и "мусора", а определяют они две стороны одной и той же ошибки, заключающейся в некорректной работе с указателями. Следующий фрагмент программы иллюстрирует возникновение таких ошибок (тип "Треугольник" описан выше).

```
VAR T1, T2:Треугольник;
BEGIN NEW(T1);...T2:=T1;...
      DISPOSE(T1);          (* T2-"висячая ссылка" *)
      ............
```

```
NEW(T1);...NEW(T2);...
T1:=T2;                    (* Остался "мусор" *)
```

Из этого примера понятно, что "висячая ссылка" - это указатель прикладной программы, указывающий на свободный фрагмент динамической памяти. Поскольку этот фрагмент может быть выделен системой по какому-либо запросу другой прикладной программе, Т1 может открыть доступ к "чужим" данным и "разрешить" их интерпретацию как треугольника. Последствия такой интерпретации в общем случае непредсказуемы. Заметим, что "висячая" ссылка и "пустая" ссылка (имеющая значение NIL, см. разд.III) являются совершенно разными понятиями. "Мусор" - это занятый фрагмент динамической памяти, к которому в прикладной программе потерян доступ. В приведенном примере мусором оказался старый треугольник T1^, на который указывал T1 до передвижки (установки на T2). Этот мусор неустраним: программист не имеет к нему доступа, а система управления "считает" мусор занятым фрагментом памяти.

Объединяет эти два вида ошибок одно общее обстоятельство: они не обнаруживаются исполнительной средой. Идентифицировать подобные ошибки можно только путем тщательной проверки и отладки программы. И, наконец, по своим возможным влияниям на работу программы мусор гораздо "безобиднее" висячей ссылки. Он фактически приводит только к увеличенному расходу памяти, в то время как висячая ссылка способна при определенных условиях полностью парализовать процесс выполнения программы. В сложных системах "цена" висячей ссылки может оказаться очень высокой.

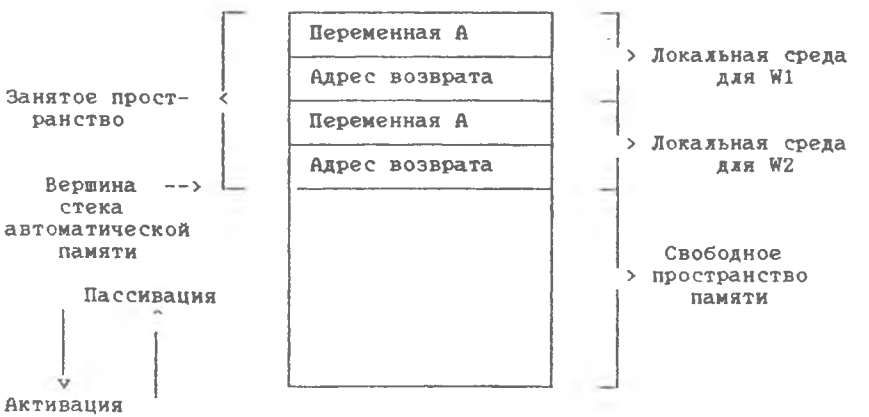
Использование автоматической памяти связано с созданием и уничтожением специальных элементов хранения, связанных с актив-

ными объектами - действиями или процедурами. Любая процедура требует для выполнения собственной индивидуальной локальной среды. Подобную среду образуют локальные переменные, объявленные в процедуре, формальные параметры, элемент хранения адреса возврата в процедуру, т.е. набор объектов, обеспечивающих выполнение действий, связанных с процедурой. Необходимость в локальной среде возникает только в момент вызова процедуры - момент интерпретации объекта процедурного типа. После завершения такой интерпретации необходимость в локальной среде исчезает. Таким образом, время жизни локальной среды ограничивается временем отработки программы, в которой она описана. Соответственно запрос на создание локальной среды связан с вызовом процедуры, а запрос на уничтожение - с окончанием фазы активности объекта (оператор RETURN или END в теле процедуры). Например:

```
VAR W1, W2: PROC;
PROCEDURE Работа_1;
     VAR A: INTEGER;... BEGIN... W2;...
END Работа_1;
PROCEDURE Работа_2;
     VAR A: INTEGER;... BEGIN... W2;...
END Работа_2;
BEGIN... W1:=Работа_1;... W2:=Работа_2;... W1;...
```

В этом фрагменте описаны два активных объекта процедурного типа PROC = PROCEDURE(): W1 и W2 и две процедуры без параметров: Работа_1 и Работа_2, которые могут использоваться как константы типа PROC. Интерпретация (активизация) W1 приведет к вызову Работы_1 и созданию локальной среды (содержащей переменную A). В процессе выполнения Работы_1 производится активизация объекта W2

и соответственно создание локальной среды для Работы_2. В любой текущий момент времени в системе могут быть активны несколько объектов. (В этом примере активизация W1 приводит к активизации W2, затем они оба остаются в активном состоянии и затем теряют свою активность в обратной последовательности: сначала пассивируется W2, затем W1). Последовательность активации и пассивации связана с вложенностью вызовов процедур, соответственно управление автоматической памятью основывается на использовании стека - структуры, поддерживающей такую вложенность. Ниже для этого фрагмента приведена иллюстрация распределения автоматической памяти, существующего в течение совместной активности объектов W1 и W2.



При активации каждого нового объекта вершина стека "опускается вниз" на величину, определяемую размерами локальной среды этого объекта,- при его пассивации вершина стека "поднимается вверх". С использованием автоматической памяти связаны две основные проблемы: рекурсии и множественности ассоциаций.

Рекурсия - механизм, позволяющий объекту совершать самоакти-

вацию. Например, по схеме:

W1-->W1 (прямая рекурсия)

или W1-->W2 ...-->W1 (косвенная рекурсия).

Прямая рекурсия связана с непосредственной повторной (вложенной) активацией, а косвенная - с опосредованной (причем число посредников в схеме W1-->...-->W1 может быть произвольным). Использование рекурсии напрямую связано с размерами рабочего пространства автоматической памяти. Использование рекурсивных активаций объектов, с одной стороны, позволяет иметь очень лаконичные и емкие по содержанию программы, с другой стороны, в рекурсивных схемах (особенно в косвенной рекурсии) возрастает вероятность появления трудно идентифицируемых ошибок.

Множественность ассоциаций заключается в том, что в классе автоматической памяти могут быть одновременно размещены несколько одноименных объектов, имеющих в общем случае различные значения и относящиеся к разным активностям одного и того же или опять-таки разных объектов. В приведенном примере существуют два одноименных объекта: переменная A, связанная (ассоциированная) с активностью W1, и переменная A, ассоциированная с активностью объекта W2. В соответствии с принципом стека система управления автоматической памятью всегда рассматривает в качестве активной последнюю созданную ассоциацию (самую "ближнюю" к вершине стека автоматической памяти). Возникновение множественности ассоциаций обусловлено только использованием в прикладных программах одноименных переменных с различной областью действия (областью видимости). Если их использование таких переменных и является необходимым (в чем всегда стоит усомниться), то при их интерпретации следует помнить о множественности ассоциаций.

VI. ДИНАМИЧЕСКИЕ СТРУКТУРЫ ОБЪЕКТОВ

Связанная организация памяти. - Ассоциативные структуры. - Списки. - Очереди. - Рекурсивные структуры. - Наборы. - Деревья.

Связанная организация памяти определяет множество структур, связи между элементами которых реализуются указателями. Каждый элемент такой структуры (объект) обладает, таким образом, свойством "иметь связи" с другими элементами, на которые указывает значение этого свойства. Связанная организация памяти может использоваться и для хранения статических структур данных, но поскольку реализация связей через ссылки дает возможность использовать динамические механизмы создания/уничтожения объектов, основным применением связанной организации является динамическое моделирование объектно-ориентированных систем.

Динамические структуры объектов, как уже отмечалось, характеризуются наличием особого свойства: "иметь переменный состав элементов структуры". Это свойство позволяет любую динамическую структуру рассматривать как ассоциацию связанных объектов переменного состава. (Заметим, что термин "ассоциация" используется в программировании очень часто и смысл, вкладываемый в это понятие, во многом зависит от контекста).

Свойство ассоциативности относится к общим групповым свойствам классов, оно присуще не объекту в отдельности, а совокупности объектов. Простейшим примером группового свойства является "Количество объектов в классе"- ни один из объектов класса в отдельности таким свойством не обладает. Реализация ассоциативных свойств классов требует использования специальных структур,

"связывающих" объекты-члены этих структур "узами" ассоциативности. В прикладных задачах это могут быть "узы" дружбы, общие качества, выделяющие свойства (например, свойство "Быть молодым") и т.п.. Ассоциация объектов, кроме того, как правило упорядочена по определенной системе правил - отношений порядка на множестве членов ассоциации. Такие правила устанавливают биективную (взаимно-однозначную) связь между множеством объектов-членов ассоциации и элементами натурального ряда чисел. В этом смысле ассоциация - более сложная структура, чем множество объектов.

Правила, регулирующие упорядоченность ассоциации, могут быть сконструированы как выделяющие свойства на множестве имманентных свойств объекта (например,упорядоченность по "Возрасту" объекта, "Приоритету" объекта). Они могут быть построены на основе времени модификации состава членов ассоциации (например,правило "первым пришел - первым вышел" хорошо известно всем, кто бывал в очередях: каждый вновь пришедший в очередь становится последним членом этой ассоциации очередников). Общее свойство ассоциации заключается в том, что возможность биекции ее структуры (состава) на натуральный ряд чисел фактически определяет наличие "линейного" порядка на множестве ее членов. Такой порядок определяется отношениями предшествования: "предок-потомок", "предыдущий-последующий" и т.п. Это свойство делает основной реализационной структурой ассоциации **линейный список**.

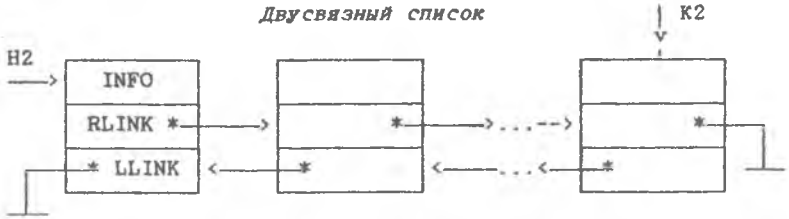
С помощью списков могут моделироваться разнообразные динамические структуры, поддерживающие различные отношения порядка. В программировании широко используются такие структуры, как стек, очередь, дек, пул - все они могут быть реализованы на списках. В зависимости от количества связей между соседними элементами раз-

личают односвязные и двусвязные списки с "встречным" направлением стрелок. Ниже приведены некоторые примеры организации списковых структур на связанной памяти.

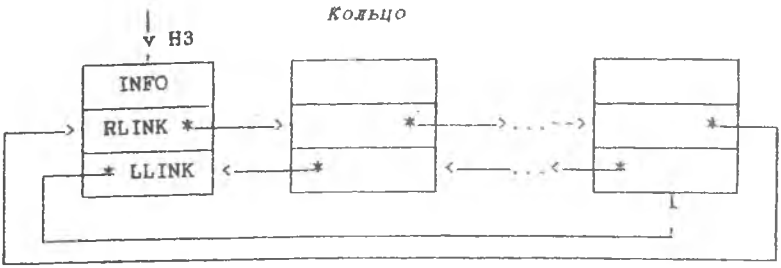
Односвязный список


```
TYPE PTR = POINTER TO Элемент;
     Элемент = RECORD INFO: Информационное_Поле;
                      LINK: PTR (* Поле связи *)
               END;
VAR H1: PTR; (* "Голова" списка *)
```

Двусвязный список




```
TYPE PTR = POINTER TO Элемент;
     Элемент = RECORD INFO: Информационное_Поле;
                      RLINK,LLINK: PTR (* Поля связи *)
               END;
VAR H2,K2: PTR; (* "Голова" и "Конец" списка *)
```

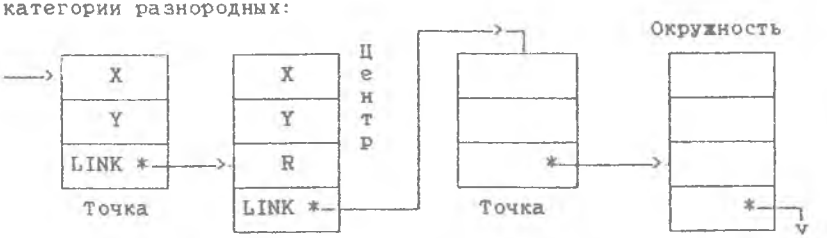


В общем случае любой элемент списка может содержать произвольное количество связей и являться членом многих списков. Это порождает большое многообразие списковых структур - фактически любая динамическая структура на связанной памяти конструируется из списков. По составу элементов списки разделяются на *однородные и разнородные*, в однородных используются объекты только одного класса, в разнородных - разных классов. Например, членами ассоциации "Элемент фигуры" могут быть объекты классов "Точка" и "Окружность":

```
TYPE Точка = RECORD
                X,Y: INTEGER (* Координаты точки *);
                LINK : ADDRESS
             END;
     Окружность = RECORD
                Центр: Точка; R: CARDINAL (* Радиус *)
                LINK : ADDRESS
              END; .
```

Как члены ассоциации, реализуемой односвязным списком, они характеризуются свойством "Иметь одну связь" (LINK) с "соседом" по ассоциации, в информационной же части эти объекты отличаются друг от друга как по представлению так и по интерпретации. Список, реализующий ассоциацию "Элемент фигуры", будет относиться к

категории разнородных:



Доступ к элементам списка реализуется через указатели. Указатель на первый элемент односвязного линейного списка (голову) открывает доступ ко всем его элементам: стрелки на рисунке указывают направление последовательного доступа. Для реализации такого доступа необходимо последовательно (в направлении, указываемом стрелками) осуществить "перестановку" (передвижку) указателя на нужный элемент списка. Естественно, что увеличение количества связей увеличивает возможности быстрого доступа к элементам списка. Например, любой элемент двусвязного списка открывает доступ к "левому" и "правому" соседу, а односвязного - только к "правому". Голова является особым элементом списка, вторым особым элементом является последний элемент - на него часто ставится указатель конца списка (К2 на схеме двусвязного списка). В структуре "кольца" понятие особого элемента становится чисто условным, поскольку никакие реальные внутренние "особенности" (как, например, К2^.LINK=NIL - условие "конца" в схеме линейного двусвязного списка) здесь не представлены.

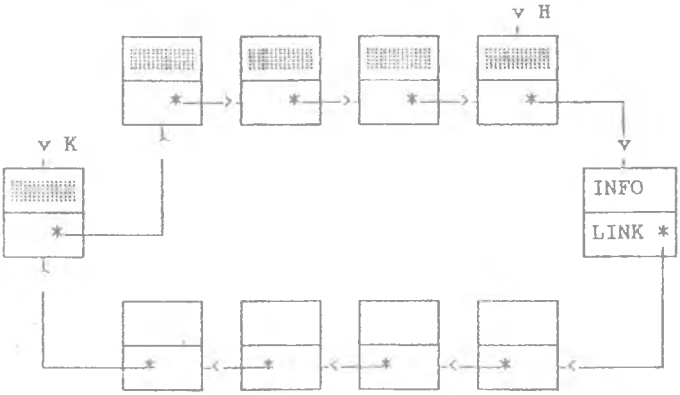
Интерпретация элементов разнородных списков связана с дополнительными трудностями- нужно не только получить доступ к соответствующему элементу структуры, но и определить, к какому классу он относится (в приведенном примере: "Точка" или "Окружность"). Для унификации процессов интерпретации таких структу

могут существенно помочь методы определения наложением (см. разд.IV). При этом совместимость представлений различных классов по полю связи становится существенным фактором обеспечения корректной работы с элементами списка. Обеспечена ли такая совместимость в определениях "Точки" и "Окружности" ?.

В задачах динамического моделирования сложных систем особый класс составляют системы с очередями. Очередь - ассоциация объектов, ожидающих доступа к системе обслуживания. Такая динамическая ассоциация характеризуется дисциплиной обслуживания (ожидания). Выше уже упоминалась дисциплина "первым пришел - первым вышел" (First In - First Out), обычно она обозначается аббревиатурой FIFO. Как разновидность очереди нередко рассматривают ассоциативную структуру стека, в этой интерпретации стек характеризуется дисциплиной "Last In - First Out" ("последним пришел - первым вышел") - LIFO. С точки зрения реализации на списках, эти структуры отличаются механизмами доступа: очередь доступна с двух концов - с "головы" (для выбора элемента из очереди) и с "хвоста" (для постановки в очередь), а стек - только с "головы" (и для включения в стек, и для вывода из стека). (В программировании используется также структура дека - линейного списка, доступ к которому возможен с любого из двух концов как для включения элемента в список, так и для удаления из списка).

Динамическое изменение состава объектов, находящихся в очереди, делает размер очереди (длину) величиной переменной. При этом моделирование очереди в статической структуре массива связано с резервированием избыточного объема памяти, достаточного для хранения очереди максимально возможного размера. На связанной динамической памяти возможно более эффективное решение, базирующееся

на использовании структуры "кольца", в которое включаются и из которого исключаются объекты-очередники. Для включения-исключения используются два указателя: на начало (голову) очереди - Н, и на ее конец - К. Такие указатели "передвигаются" по структуре кольца в направлении стрелок, связывающих элементы: К передвигается при включении нового элемента в очередь, Н - при выводе из очереди. В динамике К как бы "пытается догнать" Н, а Н - пытается "убежать" от К. Ситуация, когда К "догоняет" Н свидетельствует о том, что структура кольца полностью использована, - в этом случае необходимо дополнительное обращение к динамической памяти для выделения элемента хранения под новый объект, включаемый в очередь. Случай, когда Н "догоняет" К свидетельствует о том, что очередь пуста. Ниже приведена иллюстрация структуры кольца-очереди.



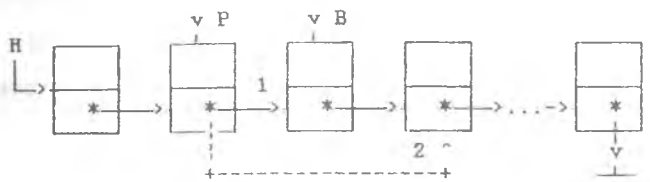
INFO - информационная часть объекта, LINK - связь с "сосе дом". Штриховка ▒▒▒ иллюстрирует "занятость" соответствующег элемента кольца (использование его для хранения объекта). В классе динамической связанной памяти возможны и другие решени организации очередей.

Основными операциями над списками являются операции вставки-удаления элементов. Такие операции всегда (независимо от техники реализации) должны выполняться корректно:

- сохранять общую структуру связанной организации списка;
- не приводить к образованию "мусора" и "висячих ссылок";
- сохранять отношение порядка элементов в списке.

Выполнение этих требований связано с корректным определением последовательности операций по модификации списков.

Например, ниже приведена иллюстрация к операции удаления элемента B из списка H.



Предполагается, что указатель B предварительно установлен на удаляемый элемент. Для удаления B необходимо:

1) Начав с головы списка H, "передвинуть" вспомогательный указатель P на элемент, предшествующий B в списке. (Как правило, это делается в циклах WHILE или REPEAT).

2) "Перебросить" связь P^.LINK (пунктир на рисунке). Это делается оператором: P^.LINK := B^.LINK (или оператором: P^.LINK := P^.LINK^.LINK).

При этом связь 1 на рисунке оказывается разорванной, а связь 2 установленной. Список H сохранил свою структуру, а элемент B не оказался "мусором".

При использовании сложных многосвязных списковых структур обеспечение корректности модификаций списков требует от програм-

миста особого внимания - любой "случайный" разрыв связи в списке превращает в "мусор" всю его часть, оставшуюся за этой связью.

Одной из самых распространенных ошибок при модификации списков является также ошибка, связанная с попыткой получить доступ к элементу через указатель H = NIL. Чтобы предотвратить такие ошибки, при конструировании булевских выражений, определяющих условия завершения (продолжения) циклов, связанных с поиском элемента и т.п., необходимо следовать простому правилу: вычисление условия (H#NIL), определяющего возможность доступа к элементу списка "под H", всегда должно предшествовать вычислению условия, содержащего квалидент с префиксом H^. В этом плане могут оказаться очень полезными правила последовательного вычисления логических условий:

A AND B = IF A THEN B ELSE FALSE;

A OR B = IF A THEN TRUE ELSE B.

Здесь A и B - логические условия.

Например, для вычисления (A AND B) вычисление B проводится только после проверки A с результатом TRUE, при A=FALSE операнд B вообще не вычисляется.

Список относится к особой группе структур - это так называемые рекурсивные структуры.

Приведем рекурсивное определение списка: Списком называется совокупность связанных элементов, из которых один является особым элементом (первым, "головой"), а все остальные образуют список. Рекурсивные структуры в программировании замечательны тем, что многие операции по их обработке можно эффективно реализовать с использованием рекурсивных процедур, которые отличаются боль-

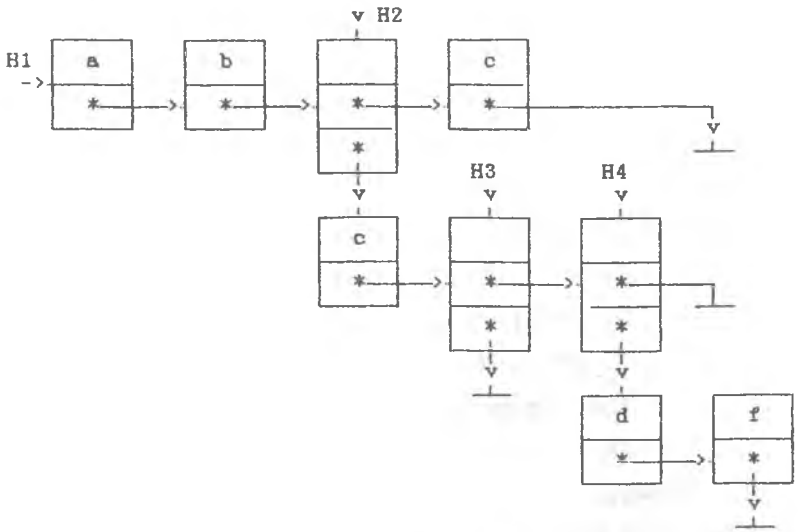
шой лаконичностью и наглядностью. В качестве примера приведем процедуру проверки, является ли H1 подсписком списка H:

```
TYPE   Указатель = POINTER TO Элемент;
       Элемент = RECORD
                    INFO : Информация;
                    LINK : Указатель;
                 END
PROCEDURE  Проверка (H,H1 : Указатель) : BOOLEAN;
       BEGIN
          IF H # NIL THEN
             RETURN (H = H1) OR Проверка (H^.LINK, H1)
          ELSE RETURN (H = H1)
          END
END  Проверка.
```

Проверка (H # NIL) в этом примере нужна только для того, чтобы предотвратить попытку интерпретировать пустую ссылку как элемент списка при вызове Проверка (H^.LINK, H1). Ситуация (H=H1=NIL) рассматривается как положительный результат проверки.

Другим примером рекурсивной структуры является структура набора, которая определяется следующим образом: Набором называется совокупность связанных элементов, каждый из которых может быть либо атомом, либо набором. Атом определяет "неделимый" элемент набора, предназначенный для хранения элементарной порции информации. Реализация наборов основана на использовании разнородных списков. Например, ниже приведена одна из возможных структур наборов. В этой структуре H1 - набор из четырех элементов (a,b,H2,c), из них H2 - набор, остальные - атомы. H2 состоит в свою очередь из трех элементов - атома c и двух наборов H3 и H4,

причем набор H3 - пуст (не содержит элементов), а H4 содержит два атома (d и f).



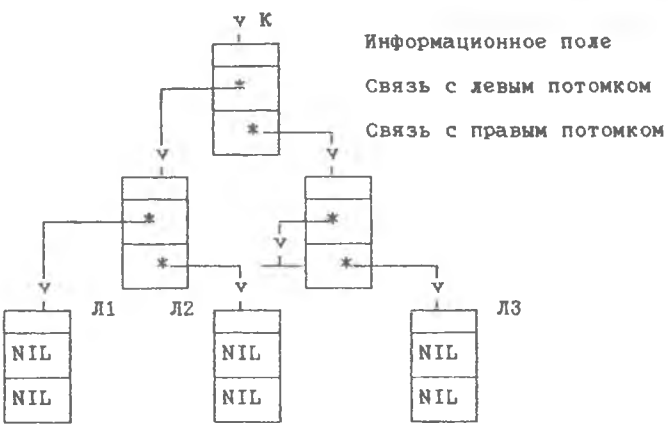
Элементы H2, H3, H4 определяют "головы" новых наборов и одновременно являются членами наборов "верхнего уровня" - при этом структура набора оказывается адекватной для реализации динамических "вложенных" понятий предметной области. Например, в ассоциацию H1-"Акционеры" могут входить как отдельные частные лица, так и коллективы - организации, которые являются ассоциациями собственных акционеров.

Элементы H2, H3, H4 на приведенной иллюстрации участвуют в двух связях - горизонтальной (связь между членами одного набора) и вертикальной (связь с членами "своего собственного" набора). Эта терминология часто используется для организации так называемых ортогональных списков, моделирующих структуры динамически изменяющегося плоского "игрового поля": "разреженных" матриц, "кроссвордов", цепей "домино" и т.д. Понятие "игрового поля",

разумеется, не означает, что ортогональные списки используются лишь в игровых задачах.

Еще один пример рекурсивной структуры, широко использующейся в программировании - структура дерева. Деревом называется совокупность связанных элементов - вершин дерева, включающая в себя один особый элемент - корень, при этом все остальные элементы образуют поддеревья. Наиболее широко используется структура бинарного дерева, все множество вершин которого делится (по отношению к корню) на два подмножества - два поддерева (левое и правое). Любая вершина бинарного дерева реализуется на связанной памяти элементом с двумя связями: с левым поддеревом и с правым.

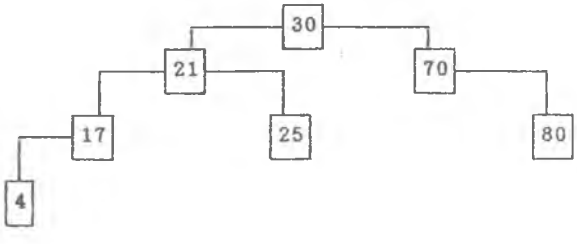
На иллюстрации изображена структура бинарного дерева на связанной памяти, здесь К - корень; Л1,Л2,Л3 - "листья" - вершины с "пустыми" связями ("не выросшими" поддеревьями).



Заметим, что в этой интерпретации дерево реализуется на однородных списках (в отличие от набора). Особое положение корня определяется единственным свойством - отсутствием вершин-предков,

любой лист дерева характеризуется полным отсутствием вершин-потомков. Поскольку любая вершина в связанной структуре дерева открывает доступ только "вниз" (только к своим поддеревьям), она может интерпретироваться как корень соответствующего поддерева. В этом смысле лист является корнем "не выросшего" дерева и определяет его своеобразную "точку роста". Включение в дерево новой вершины и удаление старой не должно нарушать общего отношения порядка на множестве вершин.

Примером такого отношения может служить отношение "больше-меньше", определяющее структуру бинарного дихотомического дерева. В таком дереве все вершины любого правого поддерева имеют значение информационного поля большее, чем значение такого же поля у корня, а вершины соответствующего левого поддерева - меньшее. Например, конструирование дихотомического дерева по последовательности целых чисел 30,70,80,21,25,17,4, начиная с 30, должно приводить к созданию следующей структуры:

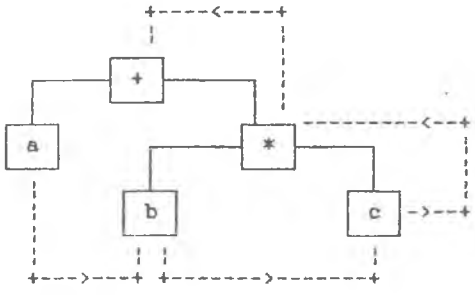


Нетрудно заметить, что процесс конструирования такого дерева происходит сверху-вниз, начиная с корня, путем последовательного сравнения числовых значений, размещаемых в вершинах, с целью определения места размещения соответствующей вершины в структуре дерева. Любая модификация дихотомического дерева (удаление вер-

шины, вставка новой вершины) не должна нарушать дихотомической структуры в целом.

В общем случае трансформация произвольной информационной строки (последовательности объектов) в структуру дерева и обратно основана на использовании глубоких структурных межобъектных отношений в исходной строке. Такая трансформация позволяет наглядно представить подобные отношения *в форме* дерева. В программировании дерево во многом рассматривается как *формальная* структура, наполняемая различным семантическим содержанием. Такой подход позволяет формально реализовать многие преобразования данных на основе унифицированных процедур обхода деревьев.

Например, в теории трансляции широко используется понятие польской инверсной записи (ПОЛИЗ) - особой системы представления математических выражений символьными последовательностями. Так, например, выражение " a + b * c " будет представлено в ПОЛИЗЕ строкой " a b c * + ". Если представить исходное выражение в естественной иерархической форме бинарного дерева :



то его восходящий обход (пунктир на рисунке) приведет к строке " a b c * + ", определяющей "польский" эквивалент исходной строки. Формула восходящего обхода "Левый - Правый - Корень" (ЛПК)

определяет правило обхода бинарного дерева: восходящий обход связан с обходом его левого поддерева, затем правого поддерева, затем корня. Поскольку каждая вершина дерева может интерпретироваться как корень "вырастающего из нее" поддерева, это правило применяется рекурсивно к каждой вершине обходимого дерева. Правило ЛПК (Левый - Корень - Правый) определяет так называемый смешанный обход, правило КЛП - нисходящий обход и т.д. Нетрудно заметить, что смешанный обход дерева дихотомии по правилу ЛКП приведет к формированию строки чисел (хранящихся в вершинах этого дерева), упорядоченной по возрастанию, а такой же обход по правилу ПКЛ - к формированию строки, упорядоченной по убыванию соответствующих чисел. Таким образом, между структурой дерева, отношением порядка на множестве информационных компонент его вершин и видом обхода существует глубокая связь, определяемая рекурсивной природой структуры дерева. Рекурсивные процедуры обхода бинарных деревьев пишутся прямо по формуле обхода с учетом спецификации представления вершин дерева. Например, ниже приведена процедура смешанного обхода бинарного дерева дихотомии, реализующего формулу ЛКП.

```
TYPE Вершина = POINTER TO Элемент ;
     Элемент = RECORD
                 Info : CARDINAL ;
                 LLink,RLink : Вершина
               END ;

PROCEDURE  Смеш_Обход (К : Вершина);
  BEGIN
```

```
    IF K # NIL THEN
        Смеш_Обход (K^.LLink); (* Обход левого поддерева *)
        WriteCard (K^.Info);  (* Обработка корня *)
        Смеш_Обход (K^.RLink); (* Обход правого поддерева *)
    END
END Смеш_Обход.
```

В традиционном программировании рекурсия часто рассматривается как некоторый заменитель итерации. Причем в качестве примеров рассматривается вычисление рекуррентных последовательностей, которые могут быть легко сформированы и обычными итераторами (циклами WHILE, REPEAT и т.п.). Природа рекурсии значительно глубже, чем механизм итерации, поэтому ее использование практически не имеет альтернатив в виде итераторов только тогда, когда решение задач проводится на рекурсивных структурах. Попробуйте написать процедуру Смеш-Обход без использования рекурсии, только на основе циклов и, если Вам это удастся, сравните ее с приведенным выше вариантом рекурсивной процедуры по наглядности, лаконичности, выразительности.

VII. ПРОЦЕССЫ В ОБЪЕКТАХ

Логический параллелизм. - Схема сопрограмм. - Понятие процесса. - Рабочая область процесса. - Создание/уничтожение процессов. - Реентерабельность. - Сигнальная синхронизация. - Основы мониторинга, ориентированного на процессы.

В любом программном объекте могут развиваться динамические процессы, определяющие изменение состояния объекта во времени.

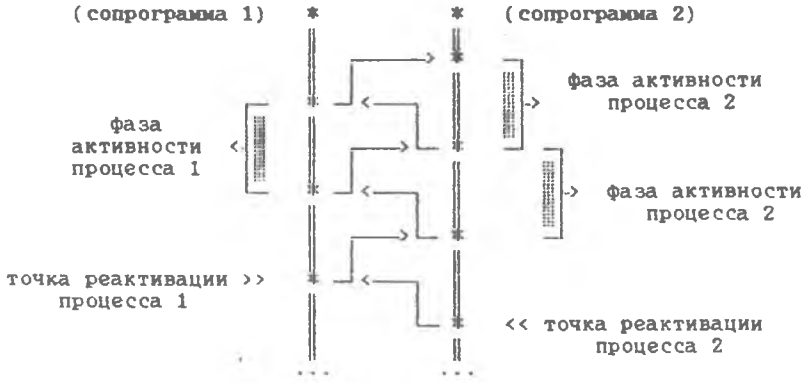
Такие процессы могут развиваться автономно (независимо один от другого) или во взаимодействии друг с другом. Концепция взаимодействия основывается на одновременном развитии нескольких процессов, при этом такая одновременность трактуется в программировании как логический параллелизм - одновременное выполнение нескольких действий (*активностей*), обусловленное логикой развития моделируемой системы. Реализация концепции логического параллелизма требует в общем случае наличия нескольких процессоров (устройств ЭВМ, обеспечивающих развитие параллельных процессов), что связано с использованием нового класса вычислительных систем - систем с *мультипроцессорной архитектурой*. Реализация параллельных процессов на обычной однопроцессорной ЭВМ связана с *имитацией* логического параллелизма последовательностью активаций разных процессов с сохранением общих логически обусловленных правил их взаимодействия. Такая имитация связана с понятием *квазипараллельности*. Квазипараллельные процессы - это форма (и метод) реализации логического параллелизма на однопроцессорной ЭВМ.

В свою очередь существует множество различных способов реализации квазипараллельности, отличающихся механизмами имитации одновременных действий последовательностями активностей. Не останавливаясь на подробном рассмотрении таких способов, мы отметим здесь общую закономерность: логическая параллельность (одновременность действий) в общем случае не приводима к последовательности активностей. Поэтому любой способ реализации квазипараллельности приводит к возникновению специфических проблем, известных в программировании как проблемы "тупиков", "критических секций", "семафоров" и т. п. Они подробно описаны в

специальной литературе, посвященной вопросам параллельного программирования и организации взаимодействующих процессов.

В основе общего механизма реализации квазипараллельности лежит *схема сопрограмм* - особая схема управления, отличающаяся от широко используемой схемы подпрограмм тем, что она строится не на основе принципа "главный - подчиненный" ("главная программа - подпрограмма"), а на основе "равноправия" всех взаимодействующих программ. В схеме сопрограмм нет главной процедуры - "хозяина", который будет манипулировать вызовом "подчиненных", - любая процедура в этой схеме трактуется как "равная среди равных". Ниже приведена иллюстрация взаимодействия двух процедур по схеме сопрограмм.

На этой иллюстрации двойной чертой изображаются *фазы активности* процесса, реализуемого сопрограммой, одинарной - передача управления от процесса процессу. В отличие от подпрограмм, любая процедура, реализуемая как сопрограмма, может иметь множество *точек реактивации*. Такие точки в тексте программы определяются расстановкой специальных операторов управления (операторы синхронизации, задержки, ожидания и т.п.).



Чередование во времени фаз активности одной и той же сопрограммы определяет логически обусловленную последовательность действий, которая и образует процесс. "Попадание" любого процесса в точку реактивации приводит к его пассивации. При этом управление передается другому процессу, причем выбор последнего производится на основе специального механизма, связанного с оператором управления, определяющим точку реактивации.

Каждый процесс, реализованный по схеме сопрограмм, имеет свою собственную *рабочую область* - индивидуальную область памяти, в которой сохраняется его локальная среда (включая в общем случае и адрес "возврата" - точку реактивации сопрограммы). Это обстоятельство и является основным фактором, разрушающим концепцию "хозяина". Если в схеме подпрограмм использование общего стека автоматически гарантирует возврат управления "хозяину" (вызывающей процедуре), то индивидуальное хранение локальных сред процессов в схеме сопрограмм в общем случае не может дать никаких гарантий по автоматической передаче управления между процессами. Более того, завершение любой сопрограммы (выход на END без передачи управления какому-либо процессу) приведет к остановке всей системы независимо от состояния других процессов. Объяснение этому очень простое: система "не знает", какому из процессов следует передать управление, поскольку общей информационной структуры, аналогичной общему стеку, в реализации "чистой" схемы сопрограмм просто нет!

Любая процедура может использоваться и как подпрограмма и как сопрограмма. Существование процедуры как сопрограммы связано с понятием процесса, при этом на основе одной сопрограммы может быть создано несколько процессов! Каждый их них может рассматри-

ваться как автономный динамический объект с собственной рабочей областью и индивидуальной локальной средой. Процедура, допускающая свое использование в качестве сопрограммы, на основе которой может быть создано несколько процессов, называется *реентерабельной*. (Ниже мы приведем примеры, связанные с реентерабельностью).

Любой процесс может реализовать обычное управление подпрограммами на основе общего стека и в то же время взаимодействовать с другими процессами на основе *трансферизации* (от слова TRANSFER) через точки реактивации. Заметьте, что в общем случае одна и та же процедура (одновременно) может использоваться и в роли подпрограммы, и как сопрограмма, определяющая развитие логически параллельных процессов!

Термин "сопрограмма" чаще всего используется для характеристики системного уровня программирования, связанного с использованием средств операционной системы или системных модулей языка программирования. При этом точки реактивации определяются операторами нижнего (системного) уровня. Использование для реактивации операторов более высокого уровня (сигнальная синхронизация, задержки на время и т. п.) в той же схеме сопрограмм как правило сопровождается уже терминологией программирования на основе взаимодействующих процессов. Понятие процесса интегрирует статические и динамические аспекты описания моделируемых систем. В некоторых языках программирования вводится даже специальный тип данных (PROCESS), объектами которого являются динамические процессы. Такие процессы могут к тому же динамически создаваться и уничтожаться (см. разд. V), что определяет многие нетривиальные возможности моделирования задач реального мира. Например, объект класса "Автомобиль" может быть в произвольный момент времени ди-

намически создан и так же уничтожен. В то же время в каждом таком объекте могут развиваться динамические процессы, например, класса "Движение" или "Торможение", которые также могут создаваться как на статической, так и на динамической основе. При этом вопрос о том, является ли движение атрибутом автомобиля или автомобиль атрибутом движения, перемещается в область философии - с позиций объектно-ориентированного подхода к программированию он может быть решен как угодно.

Создание процесса в Модуле-2 связано с использованием специальной процедуры (метода):

PROCEDURE NEWPROCESS (P: PROC; A: ADDRESS; N: CARDINAL;
VAR Pr: PROCESS).

Этот метод создает новый процесс Pr, развивающийся в соответствии с алгоритмом процедуры, определенной в P (по "телу" процедуры P), в рабочей области (A, N). Рабочая область выделяется по адресу A и имеет размер N байт. Она может быть создана как на статической, так и на динамической основе в классе динамической памяти. Разрушение рабочей области эквивалентно разрушению (уничтожению) процесса.

Метод NEWPROCESS содержит в качестве формальных параметров один объект процедурного типа (P: PROC) и один типа процесс (VAR Pr: PROCESS). Первый задает одну из множества процедур, которые могут использоваться как сопрограммы, определяющие развитие процесса. Второй предназначен для хранения текущего значения точек реактивации процесса. Выше (см. разд.II) уже отмечалось, что TSIZE (PROC) = TSIZE (ADDRESS), из этого контекста нетрудно понять, что TSIZE (PROCESS) = TSIZE (ADDRESS), т. е. формально и

тип PROC, и тип PROCESS хранят адреса и могут быть (опять-таки формально) просто заменены типом ADDRESS. Однако содержательно они определяют абсолютно разные классы объектов: процедуры, интерпретируемые в методе NEWPROCESS как сопрограммы, и динамические процессы, развивающиеся по телу этих процедур. В этом смысле абстрагирование типов здесь выступает в новой роли - как средство, позволяющее семантически разделить формально идентичные классы PROC и PROCESS.

Такое разделение становится совершенно необходимым для адекватного понимания тех ситуаций, в которых задача требует создания нескольких разных процессов, развивающихся по телу одной и той же процедуры. Например, в программе могут существовать несколько разных объектов класса "Автомобиль", каждый из которых обладает своим собственным процессом движения. В то же время алгоритм такого движения, описанный в процедуре "Движение_Авто", является общим для всех движущихся автомобилей. (Например, Движение_Авто может описывать порядок проезда определенного участка автомобильной дороги, регламентируемый правилами дорожного движения, скоростными ограничениями и т.п., одинаковыми для всех автомобилей).

```
VAR   Pr1, Pr2, Pr3 : PROCESS ;
      Ro1, Ro2, Ro3 : ARRAY [1..200] OF WORD;

PROCEDURE Движение_Авто ();
...
END Движение_Авто;
...
```

```
BEGIN
     NEWPROCESS (Движение_Авто, ADR(Ro1), 200, Pr1);
NEWPROCESS (Движение_Авто, ADR(Ro2), SIZE(Ro2), Pr2);
     NEWPROCESS (Движение_Авто, ADR(Ro3), 200, Pr3);
...
END; .
```

В этом примере три процесса Pr1, Pr2, Pr3 создаются по единственной (общей для всех них) процедуре Движение_Авто. Каждый из этих процессов будет развиваться по общим правилам (движения), но индивидуально и в индивидуальной рабочей области.

Программы, допускающие такое "одновременное" развитие нескольких процессов, как уже отмечалось, называются реентерабельными. В этом примере такой программой является Движение_Авто

Передача управления от одного процесса другому (трансферизация) на уровне сопрограмм осуществляется оператором "Передать управление от процесса P1 процессу P2". При этом в переменную P1 записывается точка реактивации этого процесса, а значение переменной P2 определяет точку активации процесса P2 (начало его очередной фазы активности). В Модуле-2 такую функцию реализует оператор TRANSFER :

```
PROCEDURE TRANSFER (VAR P1: PROCESS; P2: PROCESS).
```

NEWPROCESS и TRANSFER - два основных метода определения переменных типа PROCESS на уровне сопрограмм, определение таких переменных непосредственно присваиванием практически возможно, но надежность и корректность такого присваивания весьма сомнительна.

В общем случае арсенал методов управления развитием квазипараллельных процессов значительно шире и включает в себя не толь-

ко трансферизацию в чистом виде, но и *опосредованное управление*, реализуемое специальными объектами-посредниками, роль которых сводится к манипулированию активности процессов - мониторингу. Примером класса объектов-посредников является класс SIGNAL (сигнал). Реализация объектов этого класса может быть выполнена множеством самых различных способов, мы здесь кратко остановимся на одном из самых простых. В этой реализации SIGNAL - класс статических объектов, т.е. любой объект-сигнал создается на основе декларации соответствующих переменных вида: VAR S1,S2 : SIGNAL.

Над сигналом возможно только одно действие - подать сигнал SEND (VAR S: SIGNAL). Использование сигналов для синхронизации процессов предполагает, что она осуществляется на основе ожидания сигналов процессами. Переход процесса в состояние ожидания подачи сигнала (пассивация процесса) реализуется оператором "ждать подачи сигнала" WAIT (S: SIGNAL). Подача сигнала приводит к активации всех ожидающих его процессов (разумеется, в определенном порядке), таким образом, использование этой пары операторов позволяет манипулировать активностями процессов.

Механизм такой манипуляции основан на существовании специальной управляющей программы - монитора, которая реализует выбор активных процессов и передачу им управления. Такая программа использует специальные управляющие структуры управления, которые в общем случае можно назвать *расписанием активаций* процессов. Эта структура хранит информацию о состоянии всех процессов, протекающих в программной модели, при этом методы SEND и WAIT как директивы управления мониторингом реализуют адекватные изменения расписания активаций.

Расписание активаций является своеобразным динамически изме-

няемым планом активизации процессов и конструируется из особых объектов - *паспортов* (или дескрипторов) процессов. Каждый процесс, созданный в программе, снабжается паспортом, единственное назначение которого - представлять информацию о процессе в расписании активаций. Создание паспорта может быть реализовано и непосредственно при создании процесса, т.е. в методе NEWPROCESS. В рассматриваемом простейшем случае такой паспорт может быть описан, например, следующим образом :

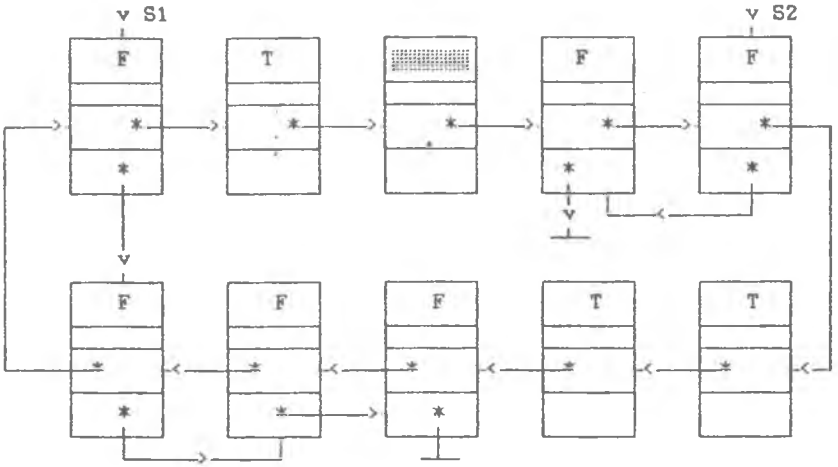
```
TYPE   PASPORT = POINTER TO PASP;
       PASP = RECORD
                STATUS : BOOLEAN;
                 (* Текущее состояние процесса *)
              Process : PROCESS;
                 LINK : PASPORT;
                QUEUE : PASPORT;
       END; .
```

При STATUS=TRUE процесс готов к активации (может быть активирован или находится в фазе активности), иначе пассивен (приостановлен в точке реактивации). Значение поля STATUS изменяется операторами опосредованного управления, так в нашем случае оператор WAIT переводит процесс (в программе которого он использован) в состояние пассивности. Оператор SEND может быть реализован по-разному: подача сигнала может пассивировать активный процесс (подающий сигнал), а может и не приводить к такой пассивации.

LINK в паспорте процесса определяет поле для связи с другими паспортами в расписании активаций, а QUEUE - поле для связей между паспортами процессов, ожидающих подачи одного и того же

сигнала, при этом TYPE SIGNAL = PASPORT.

Ниже для иллюстрации приведена одна из возможных структур расписания активаций, созданная для девяти процессов. Элемент с заштрихованным полем STATUS на этой иллюстрации является особым, он существует в системе всегда и предназначен выполнять роль головы кольцевого списка (кольца готовности процессов), который образуется связями LINK.



На приведенной иллюстрации расписания в кольцо готовности собраны девять паспортов, из них три паспорта свидетельствуют о готовности их процессов к выполнению (символ "T" - TRUE в поле STATUS). Это, конечно, не означает, что все три этих процесса активны в текущий момент времени,- активен лишь один из них, определенный правилом выбора готового процесса из кольца для активации. (В рассматриваемом случае такое правило может быть очень простым: при просмотре кольца в направлении LINK выбрать первый готовый процесс и сделать его активным).

Остальные шесть паспортов свидетельствуют о пассивности их процессов (символ "F") по причине ожидания сигналов. Из них четыре паспорта образуют линейный список по полю QUEUE, связанный с ожиданием сигнала S1, а два паспорта - такой же список, связанный с ожиданием сигнала S2. Если в процессе выполнения активного процесса будет произведена подача сигнала S1 (или S2) соответствующий список ожидания будет разрушен, а в поле STATUS всех составляющих его паспортов будет занесен символ готовности к выполнению: STATUS:=TRUE. При этом S1 (или S2) получит значение NIL, а для выполнения будет выбран новый процесс из кольца готовности.

Если в процессе выполнения активного процесса Pr будет выполнен, например, оператор WAIT(S1), то действия, связанные с пассивацией процесса Pr, заключаются в занесении в поле STATUS соответствующего паспорта значения FALSE и включении этого паспорта в список ожидания сигнала S2.

Таким образом, кольцо готовности - одна из компонент расписания активаций, которая не меняет свою структуру (если, конечно, не реализовать динамическое уничтожение процессов), а списки ожидания (другая компонента расписания) динамически создаются и уничтожаются в процессе сигнальной синхронизации. Механизмы интерпретации методов WAIT и SEND связаны с доступом к структуре расписания активаций через идентификатор текущего активного процесса (CurrentProcess). Это обычно указатель, установленный на паспорт такого процесса в расписании активаций. Смена активного процесса происходит только при его пассивации каким-либо оператором управления, при этом замена CurrentProcess новым активируемым процессом NewProcess связана с использованием следующего

механизма:

```
VAR CurrentProcess, NewProcess, HP: PASPORT;
      (* HP - вспомогательный указатель *)...

BEGIN...
 HP := CurrentProcess;
 CurrentProcess := NewProcess;
 TRANSFER (HP^.Process, CurrentProcess^.Process);
   ...
```

Таким образом, единственное назначение поля Process в структуре паспорта - обеспечить корректную передачу управления (трансферизацию) между квазипараллельными процессами .

Используемый здесь термин "трансферизация" призван подчеркнуть специфику взаимодействия процессов: это не обычная безусловная передача управления (реализуемая оператором GO TO) и не возврат управления (с помощью RETURN), это совершенно иной механизм управления.

В общем случае, мониторинг квазипараллельных процессов представляет собой отдельное, весьма сложное направление в программировании, ориентированном на объекты. Структура паспорта может при этом претерпевать существенные изменения и дополнения как реализационного, так и методологического плана. Например, использование приоритетов связано с введение дополнительного поля "Приоритет процесса". Кроме того, использование отношений хронологического порядка на множестве фаз активности требует использования в паспорте специальной "отметки времени": когда нужно активировать процесс и т.п. В целом структура расписания активаций может оказаться очень сложной, связанной с использованием

многих разновидностей списковых структур. Для понимания общей организации таких структур в задачах квазипараллельного программирования и "разложения" целого (динамики исследуемой системы) на части (процессы) объектно-ориентированный подход может оказаться весьма плодотворным.

VIII. ИНКАПСУЛЯЦИЯ

Модуль как програмный эквивалент класса объектов.- Концепция импорта/экспорта.- Закрытый и открытый экспорт.- Экспорт типов и переменных.- "Свои" и "чужие" объекты.- Расслоение свойств.

Инкапсуляция - одна из специфических особенностей программирования, ориентированного на объекты. Эта особенность предполагает не только возможности "разложения целого на части" (принципа, определяющего основы любого программирования), но и умения "скрывать" частности от общего (целого). Такой подход позволяет программисту не знать частных деталей реализации програмной системы, осуществлять конструирование из элементов, реализация которых скрыта от него "*под оболочкой*" модуля. Модуль в этом подходе приобретает роль основного конструктивного элемента, используемого для синтеза и разработки новых систем.

Специфические особенности модуля заключаются в следующем:

1) модуль - это автономно компилируемая програмная единица;

2) информационные и управляющие связи между модулями требуют использования в его описании деклараций, которые в совокупности определяют оболочку модуля, регламентирующую такие связи;

3) сборка програмной системы из модулей связана с отдельным технологическим этапом - *компоновкой* (линковкой) программы. Пра-

вила такой компоновки полностью определяются системой модульных оболочек.

Концепция оболочки реализуется декларациями *импорта/экспорта*, регламентирующими, какие объекты, определенные внутри модуля, можно использовать "за его пределами". Подобные декларации могут быть оформлены в разных видах. В Модуле-2, например, для этого используется специальный вид описания модуля - так называемая *спецификирующая оболочка* (оболочка определений, DEFINITION MODULE). В этой оболочке перечисляются объекты, экспортируемые из модуля, и специфицируются методы их использования (фактически, действия над объектами). Причем, спецификация процедурных методов проводится на уровне программиста, использующего модуль (потребитедя), которому представляются только заголовки процедур для работы с экспортируемыми объектами, но не программы их реализации. Например:

```
DEFINITION MODULE A;
  EXPORT QUALIFIED B,C,D;
  TYPE B;
  VAR C: B;
  PROCEDURE D(C:B);
END A.
```

В этом примере разрешено использование "за пределами" модуля A трех определенных в нем програмных объектов: типа B, переменной C и процедуры D.

Концепция модуля как програмного эквивалента класса объектов предполагает использование его как определителя собственной (индивидуальной) алгебры: множества возможных объектов и действий над ними. Такая концепция подразумевает, что в модуле определя-

ется *абстрактный тип* и *методы* - процедуры, манипулирующие с объектами этого типа. При этом стиль программирования, ориентированного на объекты, рекомендует экспортировать за пределы модуля только тип и процедуры - создание объектов этого типа должно производиться вне модуля - экспортера. Предыдущий пример в этом отношении нарушает такой стиль, разрешая экспорт переменной C.

Подобные стилевые особенности экспорта определяются следующими соображениями. Ведь переменная C в приведенном примере - собственная (внутренняя) переменная модуля A, размещенная в его статической памяти. Можно ли менять значение этой переменной за пределами модуля? Или это не соответствует общим "житейским" представлениям об экспорте? И вообще, что можно делать с переменной C за пределами модуля? Если что угодно, то какой смысл заводить C в модуле A? Если действия над C внутри A регламентированы процедурами A, то целесообразно экспортировать только такой регламент, т.е. процедуры. В любом случае переменная, определенная в одном модуле и используемая в другом, приобретает характер *разделяемой* переменной - с ней могут работать программы, определенные в различных модулях и, возможно, написанные разными людьми. Конечно, существуют ситуации, когда от такого экспорта невозможно или нецелесообразно отказываться, но, согласитесь, что в некоторых случаях он может быть похож на экспорт станков, которые используются как металлолом.

Для идентификации "своих" и "чужих" объектов (принадлежащих другому модулю) могут использоваться две формы импорта/экспорта: *квалифицированный* и *неквалифицированный*. Первая форма связана с использованием ключевого слова QUALIFIED в предложении экспорта и позволяет обращаться к экспортируемым объектам только по их

"внутреннему" имени, без префиксации именем модуля-экспортера. Вторая форма не требует использования этого ключевого слова, но корректная идентификация экспортируемых объектов в этом случае всегда связана с префиксацией. Например, для использования переменной C за пределами специфицирующей оболочки, определенной выше для модуля A, в случае квалифицированного экспорта достаточно простого именования C, а при неквалифицированном экспорте связано с использованием префиксированного имени A.C.

Кроме того, существуют еще две формы экспорта: *закрытый и открытый*. Открытый экспорт определяет передачу объектов, с которыми за пределами модуля-экспортера можно осуществлять любые операции, определенные в языке программирования. В этом отношении открытый экспорт снимает все ограничения, свойственные объектно-ориентированному стилю программирования и разрешает использовать станки не только как металлолом, но и как строительные конструкции, фундаментные блоки или парковые скульптуры.

Закрытый экспорт запрещает использование каких-либо операций над экспортируемыми объектами кроме тех, которые определены в модуле-экспортере. В этом смысле закрытый экспорт - это "экспорт сырья", "потребительских продуктов" и т.п.

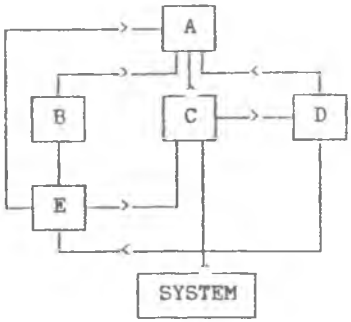
Закрытым экспортом обычно экспортируется тип данных, при этом в специфицирующей оболочке модуля отсутствует определение этого типа, он просто декларируется. В приведенном выше примере так экспортируется тип B. Модула-2 разрешает такой экспорт для ссылочных типов и некоторых отрезков типов. Вот,например, как может быть определен экспорт сигналов, используемых для синхронизации квазипараллельных процессов:

```
DEFINITION MODULE SINCHRON;
EXPORT QUALIFIED SIGNAL, SEND, WAIT;
TYPE SIGNAL;
PROCEDURE SEND (VAR S:SIGNAL);
PROCEDURE WAIT (VAR S:SIGNAL);
END SINCHRON.
```

Закрытость экспорта в этом модуле позволяет его рассматривать как полностью инкапсулированное определение абстрактного типа (алгебры) синхронизирущих сигналов. Все имманентные свойства объектов-сигналов скрыты от пользователя (в реализующей оболочке модуля - IMPLEMENTATION) и лишь два метода (SEND и WAIT) вынесены на экспорт. Закрытость экспорта разрешает над любыми сигналами, определенными вне SINCHRON, выполнение только двух действий: SEND и WAIT; использование для этого каких-либо других процедур и/или операторов языка невозможно.

Реализующие определения и имманентные свойства класса SIGNAL, определенные в модуле IMPLEMENTATION, уточняют определение сигнала SIGNAL = POINTER TO PASPORT (см. разд.VII) и определяют все детали работы с объектами этого типа.

Концепция инкапсуляции и взаимосвязей модулей через импорт-экспорт приводит к тому, что компоновка из модулей программных моделей, основанная на декларациях импорта-экспорта, оказывается связанной с образованием некоторых межмодульных структур, отображающих экспортные связи. Например, ниже приведена иллюстрация такой структуры:



Здесь главный модуль A использует модули B,C,D,E и системный модуль SYSTEM. Стрелки показывают направление экспорта программных объектов, инкапсулированных в соответствующих модулях. Структура связей на этой иллюстрации характеризуется наличием *базовых* модулей (из них стрелки только выходят), модулей *верхнего уровня* (он здесь один - A), в которые стрелки только входят, *путей* между базовыми и верхними модулями (SYSTEM-C-A), (SYSTEM-C-D-A), (SYSTEM-C-D-E-B-A и т.д.) и *петель* (C-D-E-C).

Несмотря на то, что наличие петель, вообще говоря, не является фатальным при компоновке модели A из модулей нижних уровней, тем не менее "развязка" таких петель связана с некоторыми проблемами. Реализационно и технологически они решаются корректным конструированием последовательности деклараций импорта в модуле A. Методологически же любая петля отражает некачественную декомпозицию задачи, непродуманную иерархию понятий и методов, связанных с ее решением. В этом плане лучшая схема импорта-экспорта должна основываться на выделении *программных слоев*, начиная с базового уровня и кончая верхним, предметно-ориентированным пакетом прикладных программ. При этом направление стрелок экспорта должно быть только снизу-вверх от базового слоя к верхним и, разумеется, петли должны быть исключены.

Подобное *расслоение свойств* на основе механизмов импорта-экспорта и инкапсуляции позволяет вести послойную разработку программ модулей, отладку на разных уровнях и в конечном счете позволяет повысить надежность и корректность разрабатываемого пакета программ.

ЗАКЛЮЧЕНИЕ

Объектно-ориентированный подход к разработке программ и связанный с ним стиль программирования, ориентированный на объекты, основаны на концепции абстрагирования типов. Модуль как программный эквивалент класса объектов, инкапсулирующий в себе определение такого абстрактного типа, является в этом отношении той конструктивной единицей в объектно-ориентированном подходе, которая позволяет совершить естественный переход от традиционного процедурного программирования к конструированию пакетов прикладных программ путем их послойной разработки и отладки.

Данное пособие, посвященное отдельным аспектам объектно-ориентированного подхода, преследует фактически одну цель - сформировать у читателя общее представление о парадигме абстрагирования, используя для этого представления и терминологию объектно-ориентированного подхода к разработке программ. Пособие, разумеется, не исчерпывает всех вопросов и не освещает всех тонкостей программирования, ориентированного на объекты. Более того, при написании этого пособия автор умышленно не ориентировался на конкретный объектно-ориентированный язык (например, Smalltalk). Такой подход определяется тем, что специфика реализации, терминологии и методологии использования конкретного языка всегда затушевывает интуитивные, абстрактные начала в процес-

се разработки программ, отрывает пользователя от привычных категорий программирования и тем самым порождает некоторый психологический барьер, а порою и неприятие нового подхода. В этом смысле автор считал для себя важным "сломать" такой барьер, показав читателю, что интуитивно легко ощущаемая категория объекта является абсолютно естественной для программирования, "впитывает" в себя все аспекты процесса структуризации и в этом плане логически развивает и дополняет обычное процедурное программирование новыми средствами абстрагирования.

Процесс абстрагирования является неотъемлемой частью логического мышления, и в этом отношении его развитие не имеет границ, как и развитие процесса познания вообще. Реализация такого развития на основе использования ЭВМ обеспечивает при этом не только (и не столько) новые возможности программирования, сколько новые возможности моделирования сложных объектов реального мира.

СОДЕРЖАНИЕ

К о р а б л и н Михаил Александрович

ПРОГРАММИРОВАНИЕ, ОРИЕНТИРОВАННОЕ НА ОБЪЕКТЫ

Редактор Л.Я.Чегодаева

Техн. редактор Г.А.Усачева

Лицензия ЛР N 020301 от 28.11.91.

Подписано в печать 29.11.94. Формат 60 x 84¹/16.

Бумага офсетная. Печать офсетная.

Усл.печ.л. 5,58. Усл.кр.-отт. 5,7. Уч.-изд.л. 5,5.

Тираж 200 экз. Заказ 496. . Арт.С - 104 / 94.

Самарский государственный аэрокосмический

университет имени академика С.П.Королева.

443086, Самара Московское шоссе, 34.

ИПО Самарского государственного аэрокосмического

университета. 443001 Самара, ул.Ульяновская, 18.