ГЛАВА 7

Модульное программирование

Внутри каждой большой задачи сидит маленькая, пытающаяся пробиться наружу.

Закон больших задач Хоара

Представьте себе такую ситуацию: вы руководитель отдела в программистской фирме. К вам пришел заказчик со следующим предложением: «Мне нужна программа для нахождения равновесной цены на рынке».

Вы: «Отлично. Вы пришли в нужное место. У нас лучшие специалисты по нахождению цен на рынке, и именно равновесных».

Что? Что-то не так? У вас возникло чувство дежа-вю? Или, может, авторы что-то перепутали? Уверяем вас, что нет. Просто на этой ситуации, с которой мы действительно начали самую первую главу книги, мы хотим обсудить еще одну из весьма важных концепций, входящих в обязательный багаж знаний любого квалифицированного программиста.

Итак, разовьем ситуацию дальше. Представьте, что с учетом приобретенного опыта вы провели анализ предметной области, построили информационную модель задачи, разработали алгоритм решения. Пора приступать к программированию. И тут… вы вспоминаете, что вы *руководитель отдела*, то есть у вас в подчинении *несколько* человек. И каждый горит желанием поработать. Ваша прямая обязанность – разделить задачу между всеми так, чтобы, в конечном счете, и подчиненные не «били баклуши», теряя квалификацию, и работа была сделана по возможности быстро и качественно. Как это сделать?

Конечно, вы можете возразить, что данная задача достаточно проста, а значит, ее вполне можно «отдать на откуп» одному человеку. Все верно, однако в реальной ситуации задача будет посложнее описанной и вопрос, нами заданный, неизбежно возникнет. Итак…

В принципе, достаточно несложно прийти к мысли, что в указанной нами ситуации имеющуюся задачу надо попытаться разбить на подзадачи, выполнение каждой из которых поручить отдельному специалисту. Этот принцип давно и с успехом работает в других областях человеческой деятельности. И все бы хорошо. Вот только то, что мы с вами успели изучить к настоящему моменту, не дает ответа на вопрос: а можно ли точно

так же разбить и программу на части? Представьте себе, как ваши программисты договариваются: коллеги, участок программы с 1250-й по 1740-ю строку пишу я, не трогайте его, будьте так добры.

Подводя итог, хотим отметить: коллективная деятельность является основополагающей формой выполнения работ в человеческом обществе. К тому же возможность разбиения работы на составляющие облегчает достижение результата, даже если коллектив состоит всего из одного исполнителя. И разработка программного обеспечения не могла стать исключением. А значит, сообщество программистов неизбежно должно было выработать средства поддержки такой формы деятельности. Обсуждению этих средств с общих позиций и конкретного их воплощения в языке Object Pascal и посвящена настоящая глава.

1. Концепция модульного программирования

Технология модульного программирования – оформившаяся в начале 70-х годов XX века идея разработки больших программных систем [45]. Это фундаментальная концепция, являющаяся основой всех современных подходов к проектированию и реализации. В то же время суть ее проста и отражает широко известные научные и технические методы, заключающиеся в поиске и реализации некоторого базового набора элементов, комбинации которых дают решение всех задач из определенного круга.

Если концепция структурного программирования, рассмотренная нами в главе 5, предлагает некоторый универсальный *алгоритмический* базис, то модульное программирование состоит в разработке под конкретную задачу или круг задач (предметную область) *собственного* базиса в виде набора *модулей*, позволяющего наиболее эффективно по целому ряду критериев построить программный комплекс. Модули, входящие в базис, – это целые программы (в отличие от примитивов структурного программирования), решающие некоторые подзадачи основных задач.

Средства поддержки технологии модульного программирования в целом следует разделить на два уровня. Первый, которому мы уделим основное внимание, – это *аппарат подпрограмм*, имеющийся в языках программирования, и в том числе, причем в весьма развитом виде, в языке Pascal. Эти языковые средства направлены на оформление алгоритма в виде отдельного модуля и определение его *интерфейса* с другими частями программного комплекса. Второй уровень – это надъязыковые средства в виде различных инструментальных систем разработки пакетов

прикладных программ. Целью этих систем является создание универсальных средств сборки программы из модулей по заданию, составленному пользователем-непрограммистом на так называемом профессиональном языке, то есть языке, отражающем понятия предметной области.

1.1. Необходимость модульного разбиения программной системы

Применение технологии модульного программирования начинается уже с первых этапов разработки – четкой формулировки задачи и построения математических моделей. Целью так называемого *модульного анализа* предметной области является выделение подзадач, алгоритмы решения которых будут оформлены в виде модулей. Рассмотрим мотивы, которыми руководствуются при выделении той или иной подзадачи-модуля.

Прежде всего, пытаются избежать дублирования кода, основанного на применении похожих или даже совпадающих методов или их фрагментов, необходимых для решения разных задач предметной области. Такие универсальные или, как еще говорят, инвариантные методы оформляются в виде самостоятельных модулей и составляют модульный базис.

Однако выделение некоторого метода в отдельный модуль часто производится, даже если он используется при решении всего лишь одной задачи. Необходимость такого разбиения может быть вызвана большим объемом программы. Как и в других областях человеческой деятельности, со сложной программистской задачей значительно легче справиться по частям (при некотором уровне сложности другого способа просто не существует), программируя и отлаживая части отдельно. Нетрудно понять, что держать в голове логику работы программы размером в сотню-другую тысяч строк, заниматься ее совершенствованием и отладкой, просто невозможно.

Развивая эту мысль, укажем, что еще одним побуждающим мотивом использования модульного программирования является разбиение задачи на части с целью ее коллективного решения. В этом случае каждому исполнителю необходимо выделить собственный участок работы, что наиболее естественно сделать, если разрабатываемые им фрагменты общей программной системы будут оформлены в виде отдельных модулей.

Следующим важным моментом является «объем» больших программных комплексов. В настоящее время многие программные системы настолько велики по размерам, что их размещение в оперативной памяти как единого целого либо чрезвычайно неэффективно, либо в

принципе невозможно. Следовательно, такую программу приходится разбивать на части – модули, которые загружаются в память по мере возникновения в них необходимости.

Еще один фактор связан с уже упоминавшейся нами необходимостью модификации, которой подвергается любая промышленная программа в течение своей «жизни». Правильно выполненный модульный анализ предусматривает выделение частей программы, которые в перспективе могут быть изменены, с тем чтобы это не привело к необходимости переделки всей системы в целом. Характерным примером такой модифицируемой части является пользовательский интерфейс. Допустим, при первоначальной разработке в качестве интерфейса программы был выбран консольный вариант. В дальнейшем по мере увеличения функциональности возникла потребность перейти на графический интерфейс пользователя. Очевидно, разработчик существенно облегчит себе жизнь, если при проектировании уже первого варианта системы предусмотрит возможность такой ситуации и выделит блок «визуализации» в отдельный модуль, изменение которого не будет затрагивать остальных частей программы.

Наконец, модульная структура программы может облегчить отладку. Правда, в данном случае применение модульного подхода – палка о двух концах. С одной стороны, ясно, что отдельный модуль легче отлаживать, чем программу в целом. Кроме того, модульное разбиение помогает быстрее локализовать ошибку при тестировании программы. Однако, с другой стороны, модульность порождает и новые проблемы при отладке. Во-первых, возникает вопрос: как отлаживать отдельный модуль, если он во время работы взаимодействует с другими частями программы, которые еще не отлажены или просто не реализованы? Во-вторых, программа, собранная из правильно работающих частей, может в целом работать неправильно, поскольку, например, имеются рассогласования во взаимосвязи модулей при обмене данными. Разумеется, существуют подходы к решению этих проблем. Первая из них решается с помощью «имитаторов» или «заглушек», то есть созданием фиктивного окружения отлаживаемого модуля, реализующего передачу ему стандартной отладочной информации. Вторая проблема может быть решена лишь путем тщательного проектирования и проверки интерфейса модулей. В целом, надо понимать, что отладка многомодульной программы – это специфичная проблема, требующая применения особой техники.

1.2. Средства поддержки модульной технологии в языках программирования

Надеемся, вышесказанное убедило читателя в безусловной необходимости технологии модульного программирования. На самом деле можно сказать, что в настоящее время все программы, исключая, быть может, простейшие чисто школьные задачи, разрабатываются с использованием данной технологии. Рассмотрим теперь в общем виде, какие возможности обычно предоставляют языки программирования для поддержки модульной технологии.

1.2.1. Подпрограммы. Практически все распространенные языки программирования предлагают средства оформления модулей в виде *подпрограмм*, разделяя их на два вида: *процедуры* и *функции*. Между собой они отличаются синтаксическим оформлением, а также способом вызова и передачи результирующего значения. Отметим, что содержательно вызов подпрограммы любого вида состоит в одном и том же: работа вызвавшего модуля приостанавливается, управление передается на первый оператор вызванного модуля, модуль выполняет действия до момента своего завершения, управление возвращается вызвавшему модулю.

Подпрограммы-функции используются в случаях, когда необходимо вернуть некоторый, чаще всего числовой, результат (если не учитывать так называемого побочного эффекта, о котором мы поговорим ниже). Обычно функции участвуют в выражениях в качестве операндов.

Подпрограммы-процедуры вызываются с помощью отдельного оператора и возвращают результат произвольного типа в заранее оговоренные переменные (или вообще не выдают результирующих значений, производя лишь какие-нибудь действия).

Использование подпрограмм предполагает решение вопроса о способе обмена информацией между ними. Существует два подхода к организации такого обмена: с помощью списка параметров и с помощью общих разделяемых разными модулями областей памяти.

Список параметров – набор переменных, задаваемых при описании подпрограммы и используемых в ее теле в качестве средства получения исходных данных (входные параметры) и передачи результатов работы (выходные параметры). При написании подпрограммы параметрам задаются некоторые содержательные имена. Поскольку эти имена лишь формально обозначают переменные для обмена информацией, то и называются они *формальными параметрами*. При вызове подпрограммы в списке параметров указываются имена переменных, в которых должны находиться входные для модуля данные или в которые

надо поместить выходные результаты. Эти переменные называются *фактическими параметрами*.

Передача параметров в подпрограмму и из нее обычно осуществляется двумя принципиально отличными способами: *по значению* и *по ссылке* (адресу). В первом случае в подпрограмме для параметра отводится собственная память, куда копируется значение фактического параметра из вызвавшего модуля. Во втором подпрограмма получает адрес фактического параметра вызвавшего модуля и работает с ним напрямую.

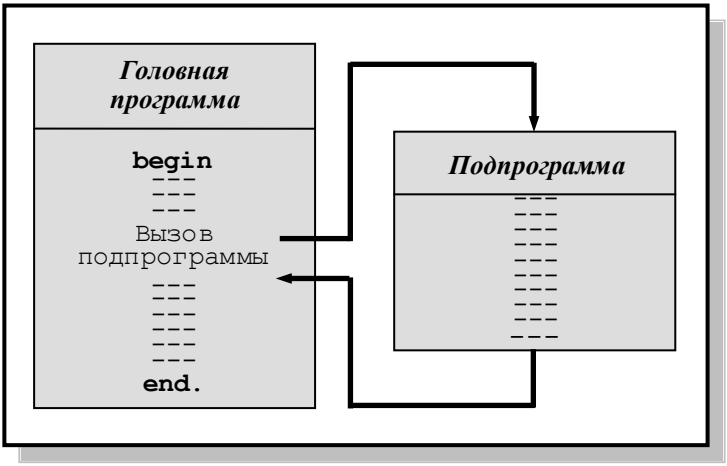

Рис. 7.1. Вызов подпрограммы

Альтернативным способом обмена информацией между модулями является выделение некоторых общедоступных глобальных переменных. В общем случае этот вариант ускоряет работу подпрограмм, но может привести к существенным трудностям в логике программы в целом. Кроме того, общедоступность глобальных переменных ведет к возможности их несанкционированного изменения, а отловить подобные ошибки – дело весьма непростое.

1.2.2. Сборка. Программа, состоящая из модулей, должна быть собрана в единое целое. Технически сборка может осуществляться несколькими способами.

Первый вариант – сборка на уровне исходных текстов. Она выполняется программистом самостоятельно с помощью редактора текстов. При этом используется соответствующая технология оформления модулей в виде *внутренних подпрограмм*. В результате компиляции собранного текста получается единый объектный модуль.

Второй вариант предполагает отдельную трансляцию каждого модуля. Исполняемый модуль из полученных объектных модулей строится специальной программой – редактором связей. Организация такого способа сборки требует наличия в языке программирования аппарата *внешних подпрограмм*.

Наконец, возможен вариант, когда окончательное объединение модулей в оперативной памяти вообще не происходит, а они загружаются с диска по мере необходимости. Как мы уже упоминали, этот вариант характерен для больших программных систем.

2. Подпрограммы в языке Object Pascal

2.1. Описание и вызов процедур и функций

Правила обращения к функциям и процедурам нам уже фактически известны. Совсем обойтись без подпрограмм в языке Pascal практически невозможно, и мы уже использовали довольно большое их количество, просто не заостряя на этом внимания.

Итак, формально *функция* вызывается в выражениях указанием ее имени со следующим за ним в скобках списком параметров. Вызов *процедуры* оформляется записью в виде отдельного оператора, состоящего из имени процедуры со списком параметров. Заметим, что и у процедур, и у функций список параметров может отсутствовать.

Синтаксические правила оформления процедур и функций, а также правила обращения к ним одинаковы для внутренних и внешних подпрограмм языка Pascal.

Текст подпрограммы состоит из заголовка и следующего за ним блока. Напомним, что блок – это совокупность объявлений и исполняемых операторов, причем весь набор операторов заключается в скобки **begin** и **end**.

Заголовки функции и процедуры имеют вид:

```
function <имя функции>(<список формальных параметров>):
  <тип возвращаемого результата>;
procedure <имя процедуры>(<список формальных параметров>);
```

Здесь **function** и **procedure** – зарезервированные слова. Имена функций и процедур – идентификаторы.

Рассмотрим примеры описания функции и процедуры:

```
{ ================================================= }
```

```
{ Пример 7.1 }
{ Подпрограммы }

{ Показательная функция }
function Deg(a, x: Real): Real;


begin
   Deg := Exp(x * Ln(a));
                        { Оператор присваивания, указывающий }
                        { результирующее значение }
   // Result := Exp(x * Ln(a));
   { Это второй способ вернуть результат }
end; { Deg }

{ Частное и остаток от деления целых чисел }
procedure DivMod(x, y: Integer; var d, m: Integer);
begin
   d := x div y;
   m := x mod y;
end; { DivMod }
{ ================================================= }
```

Список параметров представляет собой перечисление формальных параметров через точку с запятой. Каждый параметр задается идентификатором со следующим за ним через двоеточие описателем типа. Если несколько параметров принадлежат одному типу, все их можно перечислить перед описателем типа через запятую.

Способ передачи параметра (по значению или по ссылке) задается с помощью ключевого слова **var**. Если **var** указано перед параметром, то параметр передается по ссылке, в противном случае – по значению. Параметры, передаваемые по ссылке, в терминологии языка Pascal называются *параметрами-переменными*. Параметры, передаваемые по значению, называются *параметрами-значениями*.

Параметры-значения в языке Pascal передаются только в одну сторону – из вызывающей подпрограммы в вызываемую. Другими словами, параметр-значение может быть только входным и не может использоваться для возвращения вычисленных подпрограммой значений. Таким образом, входные параметры, оформленные как параметры-значения, защищены от непредусмотренного изменения в процессе работы подпрограммы, что, конечно, является плюсом. Достигается это за счет того, что реально подпрограмма оперирует с копией переданного параметра. Создание копии (выделение памяти и копирование значения) осуществляется автоматически. Попутно это позволяет внутри подпрограммы использовать

параметры-значения как обычные переменные, при этом изменение их значений никак не скажется на фактических параметрах.

Однако в таком подходе есть и свои минусы. Если в подпрограмму требуется передать структуру, требующую для размещения сотню килобайт, создание копии может стать весьма накладным, как по памяти, так и по времени, которое будет потрачено на собственно копирование значений. Корректное решение этой проблемы мы обсудим чуть ниже.

При передаче в подпрограмму параметра-переменной копии не создается, вместо этого передается адрес фактического параметра, что, естественно, экономит время, память и дает возможность изменять значение параметра внутри подпрограммы.

Соответствие между формальными и фактическими параметрами при вызове устанавливается по порядку их следования в заголовке подпрограммы и в операторе вызова (или в указателе функции). Разумеется, количество фактических и формальных параметров должно совпадать.

Важное требование, о котором следует постоянно помнить, состоит в обязательной идентичности типов соответствующего фактического и формального параметра. Таким образом, анонимное объявление типа формального параметра не допускается, то есть нельзя создать подпрограмму вида

```
procedure Sort(Arr: array[1..10] of Integer);
```

Для функций помимо параметров нужно указать еще тип возвращаемого значения. Чтобы передать результат функции вызывающей подпрограмме (программе), используется оператор присваивания, в левой части которого стоит имя функции или зарезервированный идентификатор `Result`.

2.2. Виды параметров

Вернемся к проблеме передачи в подпрограмму параметра большого размера. Мы выяснили, что передача его по значению приводит к существенным накладным расходам, а передача по ссылке может привести к порче данных внутри подпрограммы, и, что самое неприятное, компилятор не сможет помочь нам обнаружить этот факт. Что же делать? На помощь приходит возможность языка Object Pascal объявлять параметры как константы:

```
type
  Numbers = array[1..10000] of Integer;
  Complex = record
```

```
    Re: Real;
    Im: Real;
  end;

function Min(const a, b: Integer): Integer;
  { передача по значению }
procedure Print(const a: Numbers);
  { передача по ссылке }
function Mult(const c1, c2: Complex): Complex;
  { передача по ссылке }
```

Обсудим, что происходит с *константными параметрами* при вызове подпрограммы.

Прежде всего, отметим, что внутри подпрограммы значение параметра-константы изменено быть не может вне зависимости от его типа. Компилятор «зорко» следит за этим.

Что касается способа передачи, то для простых типов данных, таких, как в функции `Min`, параметры-константы передаются по значению, то есть для них создается копия. Для сложных типов данных (массивов, записей) передача параметров-констант осуществляется по ссылке. Таким образом, модификатор **const** на пару с компилятором решают проблему эффективной и безопасной передачи в подпрограммы параметров большого размера.

Рассмотрим еще одну задачу. Пусть нам требуется функция, заполняющая массив типа `Numbers` начальными значениями. Пишем функцию:

```
procedure Fill(var a: Numbers; Value: Integer);
var
  i: Integer;
begin
  for i := 1 to 10000 do
    a[i] := Value;
end;
```

А теперь представим себе, что в большинстве случаев инициализирующее значение будет равно нулю и лишь иногда другому значению. Хотелось бы иметь возможность сократить вызов процедуры `Fill`, не указывая второй параметр, если его значение должно быть равно нулю.

Такая возможность обеспечивается *параметрами по умолчанию*. Чтобы задать значение параметру, которое будет использоваться по умолчанию при вызове, при описании этого параметра ставится знак «=» и указывается конкретное значение:

```
procedure Fill(var a: Numbers; Value: Integer = 0);
```

В результате следующие два вызова процедуры `Fill` будут эквивалентными:

```
Fill(Arr, 0);
Fill(Arr);
```

Иметь значения по умолчанию могут только параметры-значения и параметры-константы. У подпрограммы может быть несколько аргументов со значениями по умолчанию. При этом действует следующее ограничение: все такие аргументы должны находиться в конце списка параметров.

Последний из специальных видов параметров, который мы обсудим, – *открытые массивы*.

Допустим, мы хотим написать подпрограмму сортировки массива целых чисел. При ее объявлении мы должны будем указать некоторый заранее созданный тип-массив. В соответствии с требованием идентичности формальных и фактических параметров при вызове этой подпрограммы мы можем использовать лишь созданный нами тип. Очень неудобно! К тому же при описании типа мы должны указать конкретный размер массива, что тоже сужает возможности по использованию данной подпрограммы. В то же время очевидно, что алгоритм сортировки не зависит ни от типа массива, ни от числа элементов в нем. Открытые массивы языка Pascal позволяют разрешить эту ситуацию.

Синтаксически объявление параметра как открытого массива производится конструкцией вида:

```
array of <тип элементов>
```

Таким образом, процедуру сортировки мы можем объявить как:

```
procedure Sort(var a: array of Integer);
```

Нумерация элементов открытого массива начинается с нуля. Номер последнего элемента можно узнать с помощью функции

```
High(<имя открытого массива>)
```

При вызове подпрограммы с параметром – открытым массивом в качестве фактического параметра может быть подставлен любой массив с соответствующим типом элементов, а также простая переменная того же типа.

Открытый массив может являться параметром-значением, параметром-переменной и параметром-константой.

В окончании раздела приведем пример реализации процедуры сортировки.

```
{ ================================================= }
{ Пример 7.2 }
{ Процедура сортировки с параметром - открытым массивом }
```

```
procedure Sort(var a: array of Integer);
var
   i, j: Integer;
   temp: Integer;
begin
   for i := 1 to High(a) do
      for j := High(a) downto i do
        if a[j] < a[j - 1] then
        begin
           temp := a[j];
           a[j] := a[j - 1];
           a[j - 1] := temp;
        end;
end;
{ ================================================= }
```

3. Внутренние подпрограммы. Область действия имен

Внутренние процедуры и функции в языке Pascal описываются в разделе объявлений блока. Причем здесь речь идет как о блоке основной программы (начинающейся со слова **Program**), так и о блоке любой процедуры или функции. Другими словами, подпрограммы могут быть описаны внутри других подпрограмм.

Рассмотрим пример описания внутренних подпрограмм.

```
{ ================================================= }
{ Пример 7.3 }
{ Внутренние подпрограммы }
Program P1;

{$APPTYPE CONSOLE}

var
   a, b: Real;
   x: Boolean;
   i: Integer;

   procedure P2;
   var
      a: Integer;
      x: Boolean;

      procedure P4;
      type
         Complex = record
```

```
          Re, Im: Real;
        end;
      var
        a: Real;
        f: Complex;
      begin
        { Тело процедуры P4 }
        // a - 4, b - 1, x - 2, i - 1, f - 4
      end; { P4 }

   begin
      { Тело процедуры P2 }
      // a - 2, b - 1, x - 2, i - 1
   end; { P1 }


   procedure P3;
   var
      b: Integer;
      a: Real;
   begin
      { Тело процедуры P3 }
      // a - 3, b - 3, x - 1, i - 1
   end; { P3 }

begin
   { Тело программы P1 }
   // a - 1, b - 1, x - 1, i - 1
end.
{ ================================================== }
```

Главная программа `P1` содержит в своей декларативной части описание двух процедур `P2` и `P3`. Процедура `P2`, в свою очередь, содержит описание процедуры `P4`.

Написание текста программы и процедур может выполняться независимо, то есть в разные моменты времени, разными людьми. Но если принято решение, что все подпрограммы являются внутренними, то перед компиляцией всей созданной программной системы должна быть выполнена сборка текста (с помощью редактора текста) в единый исходный модуль в соответствии с синтаксисом языка и выбранными правилами вложенности подпрограмм.

Поскольку в рассматриваемом случае все модули подаются на вход транслятора одновременно, то он «видит» все используемые имена объектов программы. Поэтому центральным вопросом, связанным с использованием внутренних подпрограмм, является вопрос разделения

имен между подпрограммами. Например: могут ли две подпрограммы использовать один и тот же идентификатор для обозначения разных объектов? Или, напротив: доступен ли некоторый объект (переменная, подпрограмма и т.п.), описанный в одной подпрограмме, для использования в другой подпрограмме?

Для однозначного разрешения этих и других подобных вопросов вводится система правил, определяющая *области действия* (доступности, «видимости») имен. Правила состоят в следующем. Объект, объявленный в некотором модуле, является *локальным* для этого модуля. Он «не виден» во всех других модулях (как «соседних», так и «объемлющих» данный), за исключением тех, которые описаны как внутренние подпрограммы данного модуля. Для этих внутренних подпрограмм объект является *глобальным*, то есть доступным для использования в том смысле, как он описан.

Однако если имя глобального объекта используется для описания другого объекта во внутренней подпрограмме, то вновь вступает в действие первая часть правила. Это значит, что имя вновь локализуется и бывший глобальный объект становится недоступен по «переобъявленному» имени внутри этой внутренней подпрограммы.

Проиллюстрируем правила на приведенном выше примере 7.3. Главная программа `P1` может использовать в своей исполняемой части обращения к процедурам `P2` и `P3`, но ни `P1`, ни процедура `P3` не могут вызвать процедуру `P4`, так как она локализована в процедуре `P2`. Процедура `P3` может обратиться к процедуре `P2`, но не наоборот, так как `P2` объявлена ранее `P3`.

Далее: поскольку переменные `a`, `b`, `x`, `i` описаны в главной программе, они являются глобальными для всех внутренних модулей. С другой стороны, во внутренних подпрограммах имеются переобъявления имен, что локализует их. В приведенной схеме около имен стоит номер модуля, в смысле объявления которого следует понимать переменную. Например, одно и то же имя `a` объявлено во всех модулях. Это означает, что имеется четыре разных переменных с именем `a`, причем компилятор однозначно определяет, где о какой переменной идет речь, на основании вышеизложенных правил. Дополнительно заметим, что объявленный в `P4` тип `Complex`, а также переменная `f` недоступны для использования нигде, кроме процедуры `P4`.

На основе вышесказанного может сложиться впечатление, что в результирующем загрузочном модуле под все локальные и глобальные переменные будут отведены свои области памяти, и например в памяти, занимаемой программой `P1`, будет выделено 8 байт под вещественное `a` главной программы, 4 байта под целое `a` процедуры `P2` и еще два раза по 8

байт под вещественные a, принадлежащие процедурам P3 и P4. Это не так. Язык Pascal обладает механизмом динамического размещения переменных. Суть его состоит в том, что переменная существует (то есть реально размещается в оперативной памяти) только в то время, когда является активным блок, где она объявлена. Блок считается активным после начала его выполнения и до завершения его работы. Другими словами, память под переменные отводится только после входа в подпрограмму, где они описаны, и отбирается у них после завершения работы подпрограммы. Заметим, что переменные, объявленные в главной программе, существуют все время работы программы (для них отводится специальная область исполняемого модуля).

Понимание данного метода управления памятью очень важно, так как в противном случае могут возникнуть необоснованные надежды на сохранение записанных в локальные переменные значений после завершения работы соответствующей подпрограммы. Например, требуется разработать модуль, который, в частности, должен подсчитывать число обращений к нему. Если программист объявит счетчик вызовов как локальную переменную и будет надеяться, что при каждом новом входе в подпрограмму в счетчике сохранится прежнее значение, к которому достаточно лишь прибавить 1, то в общем случае это приведет к ошибке. Мы пишем «в общем случае», поскольку случайно может оказаться, что система управления памятью каждый раз отводит под счетчик одно и то же место в памяти и оно в промежутках между работой модуля не отдается под другие переменные.

Главным достоинством динамического распределения памяти под переменные является снижение затрат памяти. Однако имеется и существенный недостаток – дополнительное время на работу системных процедур управления памятью, а также дополнительные затраты памяти для размещения этих процедур.

Использование глобальных переменных позволяет организовать обмен информацией между модулями помимо аппарата параметров. С одной стороны, эта возможность облегчает запись текста программы, так как отпадает необходимость постоянно переписывать длинные списки фактических параметров сложных модулей. С другой стороны, этот способ «размывает» межмодульный интерфейс, делает его неявным, что может породить ошибки.

В общем случае можно сформулировать рекомендацию, состоящую в следующем: используйте глобальные переменные только тогда, когда без них нельзя обойтись или когда их использование имеет серьезные основания.

4. Побочный эффект

В языке Pascal при работе с подпрограммами возможно возникновение так называемого *побочного эффекта*.

Первое из его проявлений связано с тем, что функция может возвращать значения не только стандартным для нее способом, но и через параметры, помеченные ключевым словом **var**. Например, процедура одновременного вычисления частного и остатка от деления целых чисел, приведенная выше, может быть переписана в виде функции:

```
{ ================================================= }
{ Пример 7.4 }
{ Функция с побочным эффектом }
function DivMod(x, y: Integer; var m: Integer): Integer;
begin
   DivMod := x div y;
   m := x mod y;
end;
{ ================================================= }
```

Функция `DivMod` в качестве «основного» результата возвращает частное, а «побочно» еще и остаток от деления.

Другой возможный побочный эффект связан с ошибками при работе с глобальными и локальными переменными и возникает при пропуске описания локальной переменной, имя которой совпадает с именем однотипной глобальной переменной. Подобную ошибку компилятор обнаружить не в состоянии. Рассмотрим пример программы сортировки двух чисел, которая использует процедуру перестановки значений двух переменных:

```
{ ================================================= }
{ Пример 7.5 }
{ Еще раз побочный эффект }
program Sort;

{$APPTYPE CONSOLE}

var
   x, y :Real;
{ Перестановка значений двух переменных }
procedure Swap(var a, b: Real);
begin
   y := a;
   a := b;
   b := y;
```

```
end;

begin
   Writeln('Упорядочение двух чисел по возрастанию');
   Writeln('Введите два вещественных числа');
   Readln(x, y);

   if x > y then
      Swap(x, y);

   Writeln(x, y);
end.
{ ============================================== }
```

В этом примере грамотно написанная процедура перестановки значений двух переменных `Swap` содержит рабочую переменную `y`, имя которой «случайно» совпадает с именем глобальной переменной, обозначающей одно из сортируемых чисел. Как мы знаем, в этом нет ничего страшного, если рабочая переменная локализуется в подпрограмме соответствующим объявлением. Однако в нашем примере это не сделано. Разумеется, компилятор не выдаст сообщение об ошибке, так как имя `y` описано до его использования. При работе программы в случае, когда `x > y`, значение введенной переменной `y` будет заменено значением `x`, то есть произойдет *побочный эффект* при работе процедуры `Swap`.

5. Передача ссылки на модуль (процедурный тип данных)

Допустим, требуется составить подпрограмму печати значений произвольной функции для некоторого набора аргументов. Например, эта подпрограмма должна уметь печатать «таблицу Брадиса» для выбранной элементарной функции: `Sin(x)`, `Cos(x)` и других. Подумаем над реализацией этой подпрограммы. Для начала вопрос: должна ли подпрограмма печати сама вычислять значения функций? Очевидно, что нет. Задачи расчета значений функции и печати этих значений, вообще говоря, совершенно независимы, так что разумно их разделить. Тогда возникает вопрос номер два: как передать значения функции в подпрограмму печати?

Возможно решение – предварительно вычислить все требуемые значения, записать их в массив и передать его как фактический параметр. Надеемся, что читателю очевиден существенный недостаток этого подхода. Для размещения массива требуется память, в то время как на самом деле

целиком он нам не нужен (по крайней мере, об этом в условии задачи ничего не было сказано) – для печати достаточно иметь только текущее значение функции. Итак, от идеи рассчитать все значения заранее отказываемся – цикл печати будет вызывать расчетную подпрограмму с конкретным значением аргумента. Остается последний вопрос: как сообщить этой расчетной подпрограмме, для какой функции производить вычисления?

Ответ на этот вопрос подводит нас к важному понятию межмодульного интерфейса – к ситуации, когда в модуль нужно передать не значение некоторого параметра, а правило (алгоритм) его вычисления. Другими словами, нам требуется одному модулю сообщить имя другого, которым следует воспользоваться для получения необходимых данных.

Для решения этой проблемы язык Pascal включает возможность конструирования специального типа данных – *процедурного типа*. Значениями этого типа являются процедуры и функции, описанные в некотором блоке. Процедурный тип, так же как и массивы или записи, – это некоторый класс типов. Конкретный процедурный тип характеризуется видом заголовка подпрограммы, включающего описание параметров. Общий вид объявления процедурного типа представлен ниже:

```
type
  <имя типа> = <заголовок функции или процедуры без имени>;
```

Например, следующее объявление процедурного типа

```
type
  FuncType = function(x :Real): Real;
```

задает класс всех вещественных функций с одним вещественным аргументом. «Константой» этого типа может быть функция с любым именем (а также любым другим именем формального параметра), но с такой же структурой заголовка и типами аргумента и возвращаемого значения. Например, функция

```
function Pol3(z: Real): Real;
begin
  Pol3 := Sqr(z) * z - 3.5 * Sqr(z) + 6.7 * z - 70.4;
end;
```

принадлежит типу `FuncType`.

Синтаксически параметр процедурного типа в списке формальных параметров подпрограммы описывается так:

```
<идентификатор>: <имя процедурного типа>;
```

Например:

```
procedure PrintTable(Func: FuncType);
```

В теле подпрограммы идентификатор рассматривается как имя процедуры или функции, список формальных параметров которой указан в объявлении процедурного типа.

Существует важное правило оформления подпрограмм, передаваемых в качестве параметров, касающееся стандартных подпрограмм, содержащихся в библиотеках системы Pascal, таких, например, как `Sin(x)`, `Cos(x)`. Их имена нельзя непосредственно передавать в качестве параметров. Если возникает такая необходимость, то следует оформить собственную подпрограмму, в которую, как в капсулу, заключить обращение к стандартной подпрограмме. Например, для `Sin(x)` можно записать подпрограмму-оболочку вида:

```
function SinX(x: Real): Real;
begin
   SinX := Sin(x);
end;
```

и в качестве фактического параметра использовать имя `SinX`.

Ниже приведен пример программы, содержащей процедуру печати таблицы значений произвольной функции (принадлежащей к ранее рассмотренному типу), задаваемой как параметр. Значения функции вычисляются в `N + 1` равноотстоящих точках отрезка `[a, b]`.

```
{ ================================================ }
{ Пример 7.6 }
{ Табулирование функций }
{ Передача имени функции в качестве параметра }
program Table;

{$APPTYPE CONSOLE}

type
   FuncType = function(x: Real): Real;

function Pol3(z: Real): Real;
begin
   Pol3 := Sqr(z) * z - 3.5 * Sqr(z) + 6.7 * z - 70.4;
end;

function SinX(x: Real): Real;
begin
   SinX := Sin(x);
end;

{ Процедура печати таблицы значений функции }
procedure PrintTable(a, b: Real; N: Word; Func: FuncType);
var
```

```
   i: Word;
   Step, x, y: Real;

begin
   Step := (b - a) / N;
   x := a;
   Writeln(' X Y ');
   for i := 1 to N do
   begin
      y := Func(x);
      Writeln(x:5:2, ' ', y:9:2);
      x := x + Step;
   end;
   y := Func(b);
   Writeln(b:5:2, ' ', y:9:2)
end;

begin
   PrintTable(2.4, 10.6, 10, Pol3);
   Readln;
   PrintTable(2.4, 10.6, 10, SinX);
   Readln;
end.
{ ================================================ }
```

В этом примере для печати таблицы необходимо в теле программы записать оператор вызова процедуры `PrintTable`, где требуется явно указать имя подпрограммы, реализующей функцию, значения которой нас интересуют. При этом все такие функции должны быть заранее написаны в виде подпрограмм и включены в общую программу.

Приведенная программа, конечно, является чисто тестовой. В общем случае программа печати таблиц значений функций должна предоставлять средства ввода функций в привычном всем формульном виде. Достижение этого идеала – задача непростая. Язык Pascal, как, впрочем, и все другие широко распространенные языки программирования, не имеет встроенных средств представления и обработки формул, то есть средств аналитических преобразований формул. Для этого существуют специализированные пакеты программ (Maple, Mathcad, Mathematica). Конечно, средствами языка Pascal можно реализовать подобную систему, но это отдельная сложная задача, требующая специальных знаний. Рассмотрение соответствующей методологии выходит за рамки нашей книги.

Однако кое-что для частичного решения рассматриваемой проблемы мы все-таки можем сделать. Мы не можем справиться с вводом формулы функции, то есть все они должны быть запасены в виде подпрограмм. Но

мы можем предоставить пользователю возможность выбирать требуемую функцию, вводя либо ее имя, либо номер.

Язык Pascal позволяет объявлять переменные процедурного типа. Мы можем сделать объявление вида:

```
var
  f: FuncType;
```

Далее этой переменной можно присваивать имена функций, принадлежащих типу `FuncType`, то есть допустимы операторы вида:

```
f := SinX;
f := Pol3;
```

Имя такой переменной может быть использовано при вызове соответствующей подпрограммы. Если, например, переменной `f` было присвоено одно из вышеприведенных значений, то можно написать оператор вида:

```
x := f(4.5);
```

Значения процедурного типа могут использоваться и при конструировании сложных типов. Например, можно объявить массив с элементами процедурного типа:

```
type
  ArrayOfFunc = array[1..2] of FuncType;
```

Мы можем запасти соответствующую типизированную константу:

```
const
  F: ArrayOfFunc = (SinX, Pol3);
```

Обращение к функции теперь может выглядеть как

```
x := F[1](4.5);
```

Усовершенствуем теперь программу из примера 7.6, позволив пользователю вводить номер функции, таблицу значений которой требуется напечатать. Номер функции будет выбираться из меню.

```
{ ================================================= }
{ Пример 7.7 }
{ Табулирование функций - вариант 2 }
{ Передача имени функции в качестве параметра }
program Table;

{$APPTYPE CONSOLE}

type
```

```
   FuncType = function(x: Real): Real;

var
   Key: Char;
   K: Byte;

function Pol3(z: Real): Real;
begin
   Pol3 := Sqr(z) * z - 3.5 * Sqr(z) + 6.7 * z - 70.4;
end;

function SinX(x: Real): Real;
begin
   SinX := Sin(x);
end;

const
   F: array[1..2] of FuncType = (SinX, Pol3);

{ Процедура печати таблицы значений функции }
procedure PrintTable(a, b: Real; N: Word; Func: FuncType);
var
   i: Word;
   Step, x, y: Real;


begin
   Step := (b - a) / N;
   x := a;
   Writeln(' X Y ');
   for i := 1 to N do
   begin
      y := Func(x);
      Writeln(x:5:2, ' ', y:9:2);
      x := x + Step;
   end;
   y := Func(b);
   Writeln(b:5:2, ' ', y:9:2)
end;

begin
   repeat
      Writeln(' Печать значений функции');
      Writeln('Функция                    Клавиша');
      Writeln('Sin(x) 1');
      Writeln('Sqr(x) * x - 3.5 * Sqr(x) + 6.7 * x - 70.4 2');
      Writeln('Конец работы               Любая другая');
      Writeln;
```

```
      Readln(Key);
      if (Key < '1') or (Key > '2') then
         break;

      case Key of
      '1': K := 1;
      '2': K := 2;
      end;
      PrintTable(2.4, 10.6, 10, F[K]);
   until False;
end.
{ ================================================== }
```

6. Бестиповые параметры

Считая это как бы само собой разумеющимся, мы часто на протяжении книги пользовались очень удобным свойством некоторых операций и стандартных подпрограмм, состоящим в том, что один и тот же знак операции или одно и то же имя подпрограммы можно записывать в совокупности с операндами (аргументами) разных типов. Например, вещественная и целая операция сложения, а также операция объединения строк имеют одно и то же обозначение « + »; функция возведения в квадрат `Sqr(X)` может применяться как к целому, так и вещественному аргументу, причем выдает результат того же, что и аргумент, типа и т.д. Такое свойство операции называется *полиморфизмом*.

Оставляя в стороне вопрос о реализации полиморфных операций над предопределенными типами данных (это проблема разработчиков компилятора), задумаемся над тем, как можно самому реализовывать полиморфные подпрограммы. Выгоды такого подхода к созданию подпрограмм очевидны: если требуется реализовать некий алгоритм общего характера, было бы очень удобно иметь одну универсальную подпрограмму вне зависимости от типа параметров.

Однако на пути реализации полиморфных подпрограмм стоят серьезные трудности. Пусть мы хотим реализовать подпрограмму сложения последовательности чисел. Очевидно, что исполняемый код модуля, складывающего вещественные числа, должен содержать команду вещественного сложения, а целые числа – команду сложения целочисленного. Значит, мы должны иметь два экземпляра кода модуля (или, по крайней мере, его части). Можно, конечно, предложить способ автоматического распознавания, какой вариант кода модуля нужен в данный момент. Правда, в этом случае возникает дополнительная

проблема: когда выяснится тип складываемой последовательности – во время компиляции или во время исполнения программы? В первом случае связывание нужного кода может выполнить сборщик (редактор связей), во втором нужны дополнительные средства динамического связывания.

Язык Object Pascal предоставляет две возможности для реализации полиморфных подпрограмм. Одну из них мы разберем сейчас, о второй поговорим в разделе «Перегрузка подпрограмм».

Итак, рассмотрим задачу сравнения двух переменных. Если мы полагаем, что переменные считаются равными при условии их побитового совпадения (что справедливо во многих случаях), то процедура сравнения может содержать одни и те же машинные команды для любого типа аргументов. Другими словами, вне зависимости от типа переменных-аргументов мы можем рассматривать их как последовательности переменных типа `Byte` и производить их побайтовое сравнение.

Обобщая вышесказанное, можно заметить, что, если удается предложить алгоритм, состоящий из одних и тех же машинных команд для разных типов аргументов, тогда тяжесть реализации полиморфизма переносится в область синтаксического оформления того факта, что формальные параметры подпрограммы не связаны с определенным типом, а также того, что фактические параметры интерпретируются внутри подпрограммы не так, как они объявлены в вызывающем модуле.

Язык Pascal имеет соответствующие средства.

6.1. Смена типа

Вначале разберемся в том, как можно указать, что двоичный код некоторой переменной будет интерпретироваться другим способом, нежели это предписано объявлением переменной. В языке имеется возможность *смены типа* переменной путем указания перед ней имени нового целевого типа (сама переменная при этом заключается в круглые скобки).

Пусть, например, переменная `c` объявлена как символ, то есть `Char`. Мы можем интерпретировать двоичный код символа, записанного в эту переменную, как целое число типа `Byte`, записав смену типа `Byte(c)`.

Важно понять, что смена типа не означает приведения типа. При смене типа не производится перекодировка значения. Другими словами, если мы записываем смену типа `A := Real(S)`, где `S` принадлежит типу `String[7]`, а `A` типу `Real`, то это не означает, что строка `S`, равная, например, `'-3.1416'`, будет преобразована к виду с плавающей точкой и в `A` будет записано вещественное число `-3.1416`. Смена типа позволяет лишь интерпретировать двоичный код, представляющий строку `'-3.1416'`, как вещественное число. Заметим, что операция смены типа требует совпадения длин переменных исходного и целевого типа, поэтому в нашем

примере строка имеет длину 7, что означает общую длину переменной 8 (за счет дополнительного байта, задающего текущую длину строковой переменной).

В качестве примера смены типа можно привести прием, позволяющий прочитать длину строковой переменной без использования функции `Length`. Длина строки, находящейся в данный момент в строковой переменной, содержится в самом первом байте переменной, который имеет номер ноль[1]. Доступ к этому байту можно получить обычным способом выделения символа строки, то есть с помощью селектора `S[0]`, где `S` – имя некоторой строковой переменной. Однако надо понимать, что количество символов строки закодировано в этом байте как целое число типа `Byte`, но имеет «родительский» тип `String`. Таким образом, если мы запишем оператор `L := S[0]`, где `L` – целая переменная типа `Byte`, то компилятор выдаст ошибку вида «Type mismatch» – «несовпадение типов». Заметим, что мы не можем воспользоваться и процедурой перевода строки в число `Val`, так как в этом байте находится число, а не символ. Выход состоит в смене типа. Оператор `L := Byte(S[0])` обеспечивает решение задачи.

6.2. Реализация полиморфной подпрограммы

Рассмотрим теперь, как возможность смены типа переменной используется при реализации полиморфных подпрограмм. Стандартные подпрограммы обработки строк типа `Pos`, `Val`, `Copy` и другие[2] в качестве параметров допускают строковые переменные любой длины. Эта интуитивно естественная возможность не согласуется с требованиями к соответствию формальных и фактических параметров, которые должны быть строго однотипными. Таким образом, если формальный параметр описан как `String`, а подставленный фактический параметр – как `String[20]`, то компилятор выдаст сообщение об ошибке (несовпадение типов). Мы рассмотрим, как можно обойти это препятствие, используя возможность смены типа и задания так называемых *бестиповых параметров* подпрограммы.

Бестиповые параметры могут быть *только* параметрами-переменными. Кстати, ответьте на вопрос – почему? Их запись в заголовке

[1] В Delphi это правило справедливо только для типа ShortString.

[2] Более детально о функциях работы со строковыми типами мы поговорим в главе 9.

процедуры выглядит очень просто: после зарезервированного слова **var** следует имя формального параметра без указания его типа.

Важно то, что в теле подпрограммы бестиповые параметры не могут использоваться сами по себе. Каждое их вхождение в выражение или другое применение должно сопровождаться сменой типа, то есть указанием, как в данной конкретной ситуации интерпретировать двоичный код значения параметра.

Пусть требуется реализовать целую функцию, подсчитывающую количество вхождений некоторого задаваемого как параметр подпрограммы символа в строку, также являющуюся параметром подпрограммы. Потребуем, чтобы эта функция была полиморфной в том смысле, что ее фактическим параметром может быть строка любого типа, то есть объявленная с любой длиной. Ниже приведена программа, решающая эту задачу.

```
{ ================================================== }
{ Пример 7.8 }
{ Использование бестиповых параметров и смены типа }
{ для реализации полиморфной процедуры }
Program UntypedPar;

{$APPTYPE CONSOLE}

var
   S: String[20];
   S1: String[62];
{ Подсчет количества вхождений символа в строку }
function NumOfChar(var S; C: Char): Byte;
var
   Count, i, Len: Byte;

begin
   { Длина строки - фактического параметра }
   Len := Byte(ShortString(S)[0]);
   Count := 0;
   i := 1;
   while i <= Len do
   begin
      if ShortString(S)[i] = C then
        Count := Count + 1;
      Inc(i);
   end;
   NumOfChar := Count;
end;

begin
```

```
   S := 'Полиморфизм';
   Writeln(NumOfChar(S, 'о'));
   S1 := 'Borland Pascal';
   Writeln(NumOfChar(S1, 'a'));
   Readln;
end.
{ ================================================ }
```

Сделаем замечания по примеру. Бестиповым параметром функции `NumOfChar` является параметр `S` (исходная строка). В теле подпрограммы ему приписывается тип `ShortString(S)`. Синтаксис языка допускает после операции смены типа запись селектора элементов массивов или полей записей. Поэтому запись `ShortString(S)[i]`, используемая в функции для выделения символа из строки, является правильной. Ну и, наконец, обратим внимание, что мы воспользовались вышеизложенным приемом получения длины строки из ее первого байта: `Len := Byte(ShortString(S)[0])`.

7. Перегрузка подпрограмм

Вернемся к задаче о сложении последовательности чисел. Очевидно, что алгоритм решения этой задачи никак не связан с типом, с помощью которого мы будем представлять элементы последовательности. Однако реализовать лишь одну подпрограмму для нахождения суммы элементов последовательности невозможно, хотя бы потому, что, как мы указывали ранее, команды, осуществляющие сложение целых и вещественных чисел, в процессоре различны. Как же быть?

В данном случае существует, можно сказать, соломоново решение. Мы все-таки вынуждены реализовать две подпрограммы сложения, однако язык Object Pascal позволяет нам назвать эти подпрограммы одним и тем же именем и тем самым создать иллюзию выполнению операции одной и той же командой. Эта возможность и называется *перегрузка подпрограмм*. Распознавание, какую из подпрограмм вызвать, мужественно берет на себя компилятор, используя для этого информацию о типах фактических параметров. Синтаксически перегрузка оформляется с помощью ключевого слова **overload.**

```
<описание подпрограммы>; overload;
```

Следующий пример демонстрирует создание и использование перегруженных подпрограмм.

```
{ ================================================ }
{ Пример 7.9 }
{ Перегрузка - сложение последовательности чисел }
Program Summa;

{$APPTYPE CONSOLE}

const
   NMax = 1000;
type
   IntArray = array[1..NMax] of Integer;
   RealArray = array[1..NMax] of Real;
var
   IntNum: IntArray;
   RealNum: RealArray;
   i: Integer;
   Sum: Real;

function SumOfSequence(Arr: array of Integer): Integer;
  overload;
var
i: Integer;
begin
   Result := 0;
   for i := 0 to High(Arr) do
      Result := Result + Arr[i];
end;
function SumOfSequence(Arr: array of Real): Real; overload;
var
   i: Integer;
begin
   Result := 0;
   for i := 0 to High(Arr) do
      Result := Result + Arr[i];
end;

begin
   { Формирование последовательностей }
   Randomize;
   for i := 1 to NMax do
   begin
      IntNum[i] := random(1000);
      RealNum[i] := random(1000);
   end;

   { Расчет и вывод результатов }
   Sum := SumOfSequence(IntNum);
```

```
   Writeln('Сумма последовательности целых чисел: ', Sum:6:1);
   Sum := SumOfSequence(RealNum);
   Writeln('Сумма последовательности вещественных чисел: ',
     Sum:6:1);
end.
{ ================================================ }
```

Отметим, что мы использовали открытые массивы в качестве параметра функций `SumOfSequence`, чтобы сделать их максимально общими.

Ключевое слово **overload**, как видно из представленного кода, должно указываться для каждой подпрограммы, использующей один и тот же идентификатор в качестве имени.

Используя механизм перегрузки, необходимо иметь в виду, что компилятор распознает подпрограммы только по списку типов параметров и никак не учитывает наличие и тип возвращаемого результата подпрограммы. Таким образом, нельзя перегрузить процедуру и функцию с одним и тем же списком типов параметров или две функции, отличающиеся лишь возвращаемым типом.

Следующие подпрограммы компилироваться не будут.

```
function Cap(S: ShortString): ShortString; overload;
begin
   ...
end;

procedure Cap(var Str: ShortString); overload;
begin
   ...
end;
```

Напротив, подпрограммы

```
function Func(X: Real; Y: Integer): Real; overload;
begin
   ...
end;

function Func(X: Integer; Y: Real): Real; overload;
begin
   ...
end;
```

будут успешно отранслированы, поскольку списки типов их параметров различны, – как видим, порядок типов в списке также имеет значение.

Подводя итог, отметим, что механизм перегрузки является весьма удобным средством, особенно активно использующимся в объектно-ориентированном программировании, речь о котором пойдет в главе 10.

8. Рекурсивные подпрограммы

Вспомним задачу о вычислении значения факториала из главы 5. Решение, которое мы тогда предложили, было основано на формуле $N! = (N - 1)! * N$. Если для факториала ввести обозначение $F_n = N!$, то формулу можно будет переписать так: $F_n = F_{n-1} * N$. Подобного вида формулы, где значение на очередном шаге определяется через значение на предыдущем, известны нам еще со школы. Вспомним, например, арифметическую и геометрическую прогрессии. Такая запись зависимостей называется в математике *рекурсивной*. В общем случае можно говорить не только о рекурсивных формулах расчетов, но и о рекурсивных алгоритмах, когда результат на очередном шаге алгоритма получается на основе результата предыдущего шага, причем выполнение каждого шага происходит по одной и той же схеме. Не правда ли, сказанное весьма похоже на определение цикла? Так и есть. Одним из способов программирования рекурсивных алгоритмов является использование циклов. Это не всегда так просто, как в примере с факториалом, но всегда возможно. Использование подпрограмм позволяет запрограммировать рекурсивные алгоритмы еще одним способом.

В общем смысле идея выглядит довольно просто. Отдельный шаг рекурсивного алгоритма оформляется в виде подпрограммы. В тот момент, когда необходимо получить результат предыдущего шага, подпрограмма вызывает сама себя с соответствующими значениями параметров. Единственное, о чем необходимо позаботиться, – в нужный момент прервать эту цепочку вызовов. Подпрограмма, устроенная по изложенной схеме, и называется *рекурсивной*.

Прежде чем обсуждать технические детали организации таких подпрограмм в языке Pascal, посмотрим пример:

```
{ ================================================ }
{ Пример 7.10 }
{ Рекурсия – вычисление факториала }
Program FactorialNew;
```

```
{$APPTYPE CONSOLE}

var
   N: Word;

function Factorial(N: Word): Longword;
begin
   if N = 0 then
   begin
      Result := 1;
      Exit;
   end
   else
      Result := N * Factorial(N - 1);
end;

begin
   Write('Аргумент: ');
   Readln(N);
   WriteLn(N, '! = ', Factorial(N));
   ReadLn;
end.
{ ================================================== }
```

Как видим, рекурсивная функция вычисления факториала не отличается большой сложностью. Строка

```
   Result := N * Factorial(N - 1);
```

реализует формулу расчета, а условие

```
   if N = 0 then
```

не дает рекурсии стать бесконечной и устанавливает начальное значение. Несмотря на простоту, данная функция дает точное представление об общей схеме любой рекурсивной подпрограммы.

А теперь обещанные технические подробности. Вызов любой подпрограммы сопровождается целым рядом обслуживающих действий, код для которых автоматически генерирует компилятор. Во-первых, в месте вызова подпрограммы компилятор вставляет команду передачи управления на адрес, с которого начинается код подпрограммы. Во-вторых, необходимо где-то запомнить *адрес возврата* – адрес инструкции,

которая в коде вызывающего модуля расположена следующей за вызываемым. В-третьих, параметры подпрограммы, переданные по значению, нужно, как мы помним, копировать, то есть выделять под них место. Область памяти, в которой выделяется это «место» (в ней же размещаются и локальные переменные подпрограммы), называется *стеком*. Адреса возврата запоминаются в нем же.

Учитывая сказанное, нетрудно догадаться, что для вычисления факториала использовать рекурсивную подпрограмму не нужно. Накладные расходы слишком велики в сравнении с реализацией с помощью цикла. На самом деле такая ситуация имеет место практически в любом случае. Единственным преимуществом рекурсии является более простой и естественно выглядящий код. Однако в некоторых случаях это преимущество может иметь решающее значение.

В качестве примера рассмотрим один из известных алгоритмов сортировки – *сортировку слиянием*.

Алгоритм основан на операции слияния двух отсортированных массивов в третий. Используются три индекса, по одному на каждый массив. Индекс в результирующем массиве на каждом шаге увеличивается на единицу. В исходных массивах элементы с текущими индексами сравниваются. Меньший элемент при сортировке по возрастанию (или больший при сортировке по убыванию) копируется в результирующий массив. Индекс в массиве, из которого выполнялось копирование, перемещается на следующий элемент.

В алгоритме сортировки слияние используется следующим образом. Массив условно делится пополам. Предполагая, что каждая половина уже упорядочена, выполняется их слияние. То же самое повторятся для каждой из половин, затем для четвертей и так далее, пока размер каждой из частей не станет равен единице. Исходя из описания алгоритма, очевидно, напрашивается рекурсивная реализация.

```
{ ================================================= }
{ Пример 7.11 }
{ Рекурсия – сортировка слиянием }
{ Основная подпрограмма }
procedure QSort(iStart, iEnd: Integer);
var
   iMiddle: Integer;
begin
   if (iStart = iEnd) then
      Exit;
   iMiddle := (iStart + iEnd) div 2;
   QSort(iStart, iMiddle);
```

```
   QSort(iMiddle + 1, iEnd);
   Combine(iStart, iMiddle, iMiddle + 1, iEnd);
end;
{ ================================================ }
```

Сортировка слиянием относится к числу «быстрых» сортировок, поэтому в названии процедуры использована буква Q – первая буква английского quick.

Для упрощения в реализации процедуры предполагается, что сортируемый массив является глобальной переменной.

```
{ ================================================ }
{ Пример 7.11 }
{ Рекурсия - сортировка слиянием }
{ Объявления }

{$APPTYPE CONSOLE}

const
   NMax = 5000;
type
   TArray = array [1..Nmax] of Integer;
var
   B, C: TArray;
{ ================================================ }
```

Осталось лишь написать процедуру слияния.

```
{ ================================================ }
{ Пример 7.11 }
{ Рекурсия - сортировка слиянием }
{ Слияние частей }
procedure Combine(iStart1, iEnd1, iStart2, iEnd2: Integer);
var
   i, j, k: Integer;
begin
   i := iStart1;
   j := iStart2;
   k := iStart1;
   while ((i <= iEnd1) and (j <= iEnd2)) do
   begin
      if B[i] <= B[j] then
      begin
         C[k] := B[i];
         Inc(i); Inc(k);
      end
      else begin
         C[k] := B[j];
```

```
        Inc(j); Inc(k);
      end
   end;
   if (i > iEnd1) then
      while (j <= iEnd2) do
      begin
         C[k] := B[j];
         Inc(j); Inc(k);
      end;
   if (j > iEnd2) then
      while (i <= iEnd1) do
      begin
         C[k] := B[i];
         Inc(i); Inc(k);
      end;
   for i := iStart1 to iEnd2 do
      B[i] := C[i];
end;
{ ================================================= }
```

Надеемся, что в особых комментариях представленный код не нуждается. Отметим лишь, что по исчерпании элементов одного из массивов элементы оставшегося просто копируются в результирующий. Также заметим, что для сокращения места мы несколько отступили от принципа: «На каждой строке по одному оператору».

В конце раздела приведем пример возможного использования написанной сортировки.

```
{ ================================================= }
{ Пример 7.11 }
{ Рекурсия - сортировка слиянием }
{ Тело программы }
begin
   Randomize;
   for i := 1 to Nmax do
      B[i] := random(1000);
   for i := 1 to Nmax do
      write(B[i], ' ');
   Readln;
   QSort(1, NMax);
   for i := 1 to Nmax do
      write(B[i], ' ');
   Readln;
end.
{ ================================================= }
```

9. Внешние подпрограммы

В наиболее полном объеме технологию модульного программирования поддерживает аппарат *внешних* подпрограмм. Только раздельная компиляция модулей обеспечивает возможность организовать достаточно «распараллеленную» коллективную разработку проекта, скрытие деталей реализации и в целом реализовать в полной мере все преимущества модульного подхода.

Язык Pascal обладает замечательным высокотехнологичным аппаратом реализации принципа раздельной компиляции. Отдельно компилируемой частью программы на языке Pascal может быть не только отдельный функциональный модуль, а целая коллекция подпрограмм и, что очень существенно, данных или их описаний. Этот набор подпрограмм и данных оформляется в виде особой программной единицы – **`Unit`**. Это английское слово имеет много значений и довольно трудно переводится в данном контексте. Мы будем придерживаться уже использованного термина – *библиотека*, наиболее полно отражающего содержание данной программной конструкции.

9.1. Оформление библиотеки

Описание библиотеки состоит из четырех основных частей: *интерфейсной* секции, секции *реализации*, секции *инициализации* и секции *финализации*.

Интерфейсная часть библиотеки (помечаемая в тексте зарезервированным словом **`interface`**), по сути, представляет собой каталог того, что в ней находится: здесь перечисляются заголовки подпрограмм и записываются объявления типов, переменных, констант, которыми можно пользоваться в программе, присоединившей к себе библиотеку.

Секция реализации (**`implementation`**) содержит сами тексты подпрограмм. Однако она может содержать не только те подпрограммы, которые были вынесены в каталог – интерфейс библиотеки, но и вспомогательные, «невидимые» пользователю подпрограммы, необходимые для реализации основных модулей. Кроме того, в этой секции могут быть сделаны всевозможные объявления данных. Эти данные также не будут «видны» в программе, присоединившей библиотеку, но могут использоваться при реализации подпрограмм библиотеки.

Третья часть библиотеки – секция инициализации (**initialization**) – не является обязательной. Ее записывают, если перед началом работы всей программы необходимо присвоить начальные значения некоторым переменным, объявленным в библиотеке. Выполнение всего программного комплекса начинается не с передачи управления на первый оператор главной программы, как это можно было ожидать, а с выполнения секций инициализации всех присоединенных к этой программе библиотек.

Четвертая часть библиотеки – секция финализации (**finalization**) – также не обязательна. Более того, она может присутствовать, только если в библиотеке использовалась секция инициализации.

В целом синтаксис записи библиотеки выглядит следующим образом:

```
Unit <Имя библиотеки>;
interface
[uses ;]
...
implementation
[uses ;]
...
[initialization]
...
[finalization]
...
end.
```

В заголовке библиотеки указывается ее имя. В принципе это произвольный идентификатор, однако при его выборе необходимо учитывать следующие соображения. Как уже отмечалось, требование на присоединение библиотеки выражается с помощью записи зарезервированного слова **uses**, в котором записывается имя вызываемой библиотеки. Сборщик (редактор связей) интерпретирует это имя как имя файла, в котором расположена библиотека. Это имя имеет стандартное расширение .dcu (Delphi compiled unit) и образуется после трансляции дублированием имени исходного текстового файла библиотеки и заменой расширения .pas на .dcu. Имя же исходного файла задается средствами редактора текстов. Таким образом, желательно, чтобы имя библиотеки, задаваемое в заголовке, совпадало с именем текстового файла (разумеется, с его первой частью), в котором размещается библиотека. В противном случае необходимо дополнительно указывать, где находится библиотека.

После зарезервированного слова **interface**, начинающего интерфейсную часть библиотеки, могут быть указаны вспомогательные библиотеки, которые необходимо присоединить к данной. Это делается,

если, например, в интерфейсной части определяются данные, при задании которых требуются понятия, определенные в других библиотеках.

Далее в интерфейсной части следуют описания констант, типов и переменных, которые будут доступны программе, присоединившей эту библиотеку.

И, наконец, в интерфейсной части приводятся заголовки подпрограмм. Еще раз подчеркнем, что сами алгоритмы, по которым работают подпрограммы, здесь не записываются.

Следующая часть библиотеки, начинающаяся с зарезервированного слова **implementation**, синтаксически похожа на обычный блок. В ней допускаются все объявления данных и записываются тексты подпрограмм, собственно и составляющих библиотеку, а также вспомогательные подпрограммы. Как видно из описания, секция реализации также может включать запрос на присоединение библиотек, которые необходимы для реализации подпрограмм.

Реализация подпрограмм, заголовки которых вынесены в интерфейсную секцию, имеет одну особенность. Допускается не записывать списки параметров в их заголовках. Они как бы копируются из интерфейсной секции. Однако пользоваться этой возможностью мы не рекомендуем, дополнительный контроль в данном случае лишним не будет: в случае несоответствия описаний заголовков компилятор сможет сообщить об ошибке.

Секция инициализации начинается с зарезервированного слова **initialization**. В этой секции можно разместить операторы, которые должны быть выполнены в программе, подключившей данный модуль. Например, это может быть открытие файлов, необходимых для работы подпрограмм библиотеки.

Секция финализации начинается с зарезервированного слова **finalization**. В ней можно поместить операторы, которые будут выполнены при завершении работы программы. Например, это может быть закрытие используемых в библиотеке файлов.

Завершается текст библиотеки словом **end**, после которого ставится точка.

9.2. Квалифицированные идентификаторы

Принцип действия имен имеет свои особенности при использовании библиотек. Все идентификаторы, объявленные в интерфейсной секции библиотеки, становятся глобальными для присоединившей ее программы,

то есть они видны в главной программе и во всех внутренних подпрограммах. Вместе с тем эти идентификаторы могут быть переобъявлены. В этом случае библиотечный идентификатор становится недоступным, однако к нему все же можно обратиться, использовав так называемый *квалифицированный идентификатор*. Квалифицированное имя состоит из двух частей, разделенных точкой. Первая часть – имя библиотеки, в которой объявлен идентификатор, вторая – собственно идентификатор.

9.3. Пример – библиотека матричных операций

Рассмотрим пример создания и использования библиотеки. Ниже приведен текст библиотеки, содержащей некоторые операции обработки матриц: ввод, вывод матрицы, сложение матриц и поиск максимального элемента в матрице. Пользователь библиотеки получает в свое распоряжение новый тип данных `Matrix`. Кроме того, он может пользоваться переменными `M` и `N`, задающими текущую размерность матрицы. Эти переменные получают начальные значения при инициализации библиотеки. В случае необходимости размерность может быть переопределена либо с использованием процедуры ввода `InDim`, либо, например, просто операторами присваивания.

```
{ ================================================ }
{ Пример 7.12 }
{ Библиотека матричных операций }
Unit Matr;
interface
const
   NRow = 10; { Максимально допустимое число строк }
   NCol = 10; { и столбцов                          }

type
   RowInd = 1..NRow; { Тип индекса строк }
   ColInd = 1..NCol; { Тип индекса столбцов }
   Matrix = array[RowInd, ColInd] of Real; { Тип матриц }

var
   M: RowInd; { Текущие значения числа строк }
   N: ColInd; { и столбцов }
```

```
procedure InDim;
procedure InMatrix(var A: Matrix);
procedure OutMatrix(A: Matrix);
function MaxElement(A: Matrix) :Real;
procedure PlusMatrix(A, B: Matrix; var C: Matrix);

implementation

var
   i: RowInd;
   j: ColInd;

{ Ввод размерностей матрицы (числа строк и столбцов) }
procedure InDim;
begin
   {$R+ автоматическая проверка диапазона допустимых значений }
   Readln(M, N);
   {$R-}
end;

{ Ввод матрицы }
procedure InMatrix(var A: Matrix);
begin
   for i := 1 to M do
      for j := 1 to N do
         Read(A[i, j]);
   Readln;
end;

{ Вывод матрицы }
procedure OutMatrix(A: Matrix);
begin
   for i := 1 to M do
   begin
      for j := 1 to N do
         Write(A[i, j]:5:2,' ');
      Writeln;
   end;
end;

{ Нахождение максимального элемента матрицы }
function MaxElement(A: Matrix): Real;
var
Max: Real;
begin
   Max := A[1,1];
   for i := 1 to M do
```

```
      for j := 1 to N do
         if A[i, j] > Max then
            Max := A[i, j];
   MaxElement := Max;
end;

{ Сложение матриц }
procedure PlusMatrix(A, B: Matrix; var C: Matrix);
begin
   for i := 1 to M do
      for j := 1 to N do
         C[i, j] := A[i, j] + B[i, j];
end;

initialization
   { Инициализация: задание размерностей матрицы «по умолчанию» }
   M := 2;
   N := 3;
end.
{ ================================================= }
```

В следующей программе, использующей вышеприведенную библиотеку, следует обратить внимание на квалифицированное имя `Matr.M`. Необходимость в нем возникает, поскольку в программе определена переменная `M` для обозначения максимального элемента матрицы, имя которой совпадает с именем переменной, обозначающей число строк матрицы.

```
{ ================================================= }
{ Пример 7.12 }
{ Использование библиотеки Mtr }
program MatrixProc;
uses Matr;
var
   X, Y, Z: Matrix;
   M: Real; { Максимальный элемент матрицы }

begin
   Write('Размерности матрицы по умолчанию: ');
   Writeln(Matr.M, ' на ', N);

   Writeln('Введите матрицу ...');
   InMatrix(X);
```

```
   M := MaxElement(X);
   Writeln('Максимальный элемент матрицы: ', M:5:2);

   Writeln('Введите матрицу ...');
   InMatrix(Y);

   PlusMatrix(X, Y, Z);

   Writeln('Сумма первой и второй матриц:');
   OutMatrix(Z);

   Readln;
end.
{ ============================================== }
```

10. Общие принципы сборки многомодульной программы

Итак, мы провели анализ предметной области, выделили набор процедур и функций для обработки данных, рассортировали подпрограммы по библиотекам и, наконец, реализовали их. Как теперь получить из всего этого богатства работающую программу? Говоря другим словами, как ее «собрать»? Обсудим.

Прежде всего, если оба возможных предложения **uses** библиотеки пусты, то она может быть оттранслирована как самостоятельная единица. В результате получится объектный файл с расширением .dcu в Delphi или tpu в версии Borland Pascal 7.0.

Далее, если библиотека подключает другие модули, для ее трансляции предварительно должны быть собраны они. Точнее, это требование является жестким в Borland Pascal 7.0, а в Delphi компилятору достаточно наличия файла с исходным кодом подключаемой библиотеки. Конечно, если вы предоставите ему файл объектный, он тоже не обидится.

В реальной практике прикладного программиста необходимость в трансляции библиотек по отдельности возникает не так уж часто. Обычно они транслируются непосредственно вместе с основной программой, для которой, собственно, и были написаны. Поскольку, как мы уже отмечали, модульный принцип разработки программ используется повсеместно, все интегрированные среды предоставляют специальные механизмы для управления всем тем, из чего, в конечном счете, складывается готовая программа. Чаще всего основным элементом этого механизма является понятие *проекта*. В общем смысле проект – это совокупность файлов, необходимых для сборки исполняемого модуля (а также динамической или

статической библиотеки), плюс набор файлов, в которых сохраняется информация о настройках интегрированной среды для данного проекта, порядке сборки, указаний компилятору и редактору связей. Обычно в наборе дополнительных файлов выделяется один главный, который именуется *файлом проекта*. Сам проект, по сути, можно считать некоторым логическим контейнером, содержимое которого составляет исходный материал для получения готовой программы.

Наличие проекта позволяет в большинстве случаев выполнять сборку программы нажатием пары кнопок. Более подробную информацию о содержимом проекта в Delphi и порядке его настройки можно найти в справочной системе.

Принципы модульной технологии в полном объеме используют и разработчики сред программирования. Как мы обсуждали еще в главе 2, любая интегрированная среда является весьма сложной системой, включающей в себя большое количество различных инструментов. Однако кроме такого функционального разбиения модульный подход используется и при реализации средств языка программирования. Существенная часть возможностей языка Object Pascal оформлена в виде подпрограмм, разбитых на некоторое количество библиотек, обычно именуемых *системными*, в том смысле, что их создателями являются разработчики среды программирования.

Использование системных библиотек, каждая из которых содержит довольно большое количество подпрограмм, предполагает решение одного важного вопроса: все ли подпрограммы библиотеки включаются в результирующий исполняемый модуль или только те, вызовы которых содержатся в программе явно?

Сборщик, входящий в систему Pascal, к счастью, обладает возможностью осуществлять «умное связывание», то есть после привязки всей библиотеки в целом в исполняемый модуль не включается код неиспользуемых подпрограмм. С другой стороны, все информационные объекты (константы и переменные), объявленные в интерфейсной секции, остаются в исполняемом модуле вне зависимости от того, использовались они или нет.

Последний момент, который нам осталось здесь обсудить, касается ситуации, когда библиотека `Lib1` нуждается в использовании библиотеки `Lib2`, а та, в свою очередь, «желает» использовать `Lib1`.

Как мы помним, библиотека содержит два предложения **`uses`**, одно в интерфейсной секции и одно в секции реализации. Правила, касающиеся циклического подключения библиотек, в этих секциях разные.

В интерфейсной секции циклы в использовании библиотеками друг друга не допускаются, то есть следующий код вызовет ошибку компиляции.

```
Unit Lib1;
interface
uses Lib2;  { Компиляция Lib1 невозможна,  }
            { Lib2 содержит ссылку на Lib1 }
...
end.

Unit Lib2;
interface
uses Lib1;  { Компиляция Lib2 невозможна,  }
            { Lib1 содержит ссылку на Lib2 }
...
end.
```

Вызвано это требование тем фактом, что компилятор должен оттранслировать интерфейсную часть библиотеки прежде, чем другая библиотека сможет воспользоваться имеющимися в ней объявлениями. Например, в интерфейсной части могут быть объявлены глобальные переменные, под которые должна быть выделена память, – использовать эти переменные до момента получения ими действительного адреса, конечно же, невозможно.

Правило «отсутствия цикла» в последовательности подключения библиотеками друг друга распространяется на любую длину этого цикла. То есть ошибочной будет и схема: `Lib1` использует `Lib2`, `Lib2` использует `Lib3`, `Lib3` использует `Lib1`. Общий принцип подключения библиотек в интерфейсной секции может быть сформулирован следующим образом: *последовательность подключений библиотеками друг друга должна быть такова, чтобы компилятор смог найти порядок трансляции, при котором интерфейсная секция любого модуля транслируется прежде, чем используется*. Фактически, если построить граф подключений, где библиотеки являются узлами, а использование – ребрами, то этот граф должен представлять собой дерево.

В секции реализации, напротив, циклическое подключение вполне допустимо, что придает большую гибкость разработке модульной структуры программы.

11. Концепция нисходящего проектирования программы

Проектирование программной системы, как и любой другой вид деятельности по созданию нового объекта, является творческим процессом, который невозможно полностью формализовать, и тем самым выдать проектировщику точную инструкцию. Вместе с тем существуют общие

фундаментальные подходы к организации этого творческого процесса, предлагающие как бы некоторую «дисциплину мышления», которая оказывается полезной в самых разных областях приложений, в том числе и в программировании.

Один из таких подходов – формирование базисного набора конструктивных элементов – уже рассматривался нами, причем в разных его применениях. На этом подходе основаны методологии модульного и структурного программирования. Своеобразной идейной надстройкой над ними является метод решения задачи «сверху вниз», широко применяемый в настоящее время при проектировании программных систем.

Напомним, что центральная идея модульного программирования состоит в предложении разрабатывать программу по частям, оформляя их в виде операций более высокого (чем имеющийся базовый) уровня – модулей, с помощью которых можно выстроить алгоритмы решения для некоторого набора задач предметной области. В свою очередь, структурное программирование дает рекомендацию, как стандартным образом формулировать эти алгоритмы, предлагая три основных алгоритмических примитива: последовательность действий, выбор из нескольких вариантов действий и повторение действий (цикл). «Действиями» являются, например, обращения к модулям. В общем же случае действие – один из упомянутых алгоритмических примитивов, который подставляется в другой примитив или сам в себя.

До настоящего момента мы оставляли в стороне вопрос о порядке подстановки. Более точно его можно сформулировать так: в какой последовательности производить сборку программы из действий? Здесь существует два крайних подхода.

Первый состоит в так называемом проектировании «снизу вверх». Представьте себе, что перед вами рассыпанный на полу детский конструктор и вы пытаетесь собрать из его элементов одну из игрушек, изображенных в приложенной к конструктору инструкции. Беря детали, вы начинаете комбинировать их, постепенно наращивая заложенный поначалу фундамент. В некоторый момент может выясниться, что ранее было принято неверное решение и продолжение сборки не приведет к получению желаемого результата.

В этой ситуации необходимо вернуться на несколько шагов назад, демонтируя детали, и попытаться выбрать верный путь. Такой метод построения и называется разработкой «снизу вверх».

В приложении к проектированию программной системы его можно сформулировать следующим образом. Опираясь на базовый язык программирования или на уже разработанный модульный базис, проектировщик комбинирует его элементы одним из трех определенных техникой структурного программирования способов, в попытке поэтапно

построить алгоритм работы системы и выделить конструктивные элементы (модули) промежуточных уровней иерархии.

Другой подход, называемый проектированием «сверху вниз», мы проиллюстрируем на примере разработки чертежа некоторой конструкции. Вначале рисуется общий внешний облик объекта. Далее начинают прорисовывать его элементы, все более и более детально. Проектировщик стремится на нижнем уровне детализации получить как можно более «крупные» типовые подконструкции, которые соответствуют стандарту, а значит, могут быть закуплены на каком-нибудь предприятии. При некотором выбранном конструкторском решении может оказаться, что для построения объекта требуется уникальная деталь. Тогда следует либо принять решение о самостоятельном производстве детали, либо вернуться «наверх» по иерархии детализации и попробовать выбрать другой способ уточнения способа создания подконструкции.

Переходя к проблеме проектирования системы, мы скажем, что метод «нисходящего проектирования» (так часто называют метод проектирования «сверху вниз») представляет собой последовательность решений по декомпозиции задачи с применением на каждом этапе одной из базовых алгоритмических структур.

Обратим внимание на важный аспект проектирования, который до сих пор оставался вне нашего рассмотрения. Проектирование программы состоит не только в разработке ее процедурной части, но и в формировании информационной модели задачи, то есть в проектировании типов данных. В процессе нисходящей разработки осуществляется уточнение структуры данных с использованием заложенных в язык стандартных приемов конструирования новых типов. Таким образом, на каждом этапе уточняется вид конкретного обрабатываемого значения, например оно определяется как запись, как массив и т.п.

Метод нисходящего проектирования, по-видимому, более адекватно отражает характер мыслительного процесса проектировщика, идущего в своих размышлениях от постановки общей задачи к деталям ее решения. Вместе с тем, разумеется, невозможно строго выдержать один из рассмотренных подходов и в реальности происходит их смешение, когда человек, разрабатывающий программную систему, многократно возвращается вверх и вниз по иерархии решений, то производя декомпозицию задачи на части, то пытаясь скомбинировать решение подзадачи из элементарных заготовок.

Говоря о нисходящем проектировании, мы как бы отделяли во времени этот этап от остальных этапов создания программы, то есть записи ее на алгоритмическом языке и отладки. На самом же деле распространенным подходом является совмещение процессов проектирования, реализации и

отладки, так что правильнее говорить о нисходящей (возможно, коллективной) разработке программной системы.

Современные системы программирования направлены в том числе и на автоматизацию труда проектировщика программного комплекса. В целом, такие системы обеспечивают возможность одновременного выполнения на компьютере всех «программистских» этапов (то есть речь, конечно, не идет о разработке математических методов) создания системы. Другими словами, разработчику, например, предоставляется возможность (конечно, при соблюдении им правил технологии) начать отлаживать программу еще до того, как она не только записана полностью на языке программирования, но даже и спроектирована в целом.

Сразу, конечно, возникает вопрос: как можно отлаживать программу, если не все ее части реализованы? Основной прием, позволяющий решить эту проблему, состоит в следующем. Вместо модулей, которые еще не написаны, но к которым производится обращение из отлаживаемой части программы, подставляются так называемые *заглушки* или *имитаторы*, то есть «пустые» модули, имеющие соответствующий интерфейс, но не решающие в полном объеме свою подзадачу, а лишь каким-то образом имитирующие ее решение, например возвращающие стандартные значения, заданные либо константами, либо с помощью датчика случайных чисел. При этом имитатор может выводить на экран дисплея какое-то сообщение, подтверждающее, что он успешно отработал.

Общие приемы нисходящей разработки программы и возможности интегрированной среды языка Pascal по поддержке этой методологии мы продемонстрируем на примере, рассмотрению которого посвящен следующий раздел.

11.1. Пример разработки программы «сверху вниз»

Допустим, требуется составить программу, которая по заданной площади и периметру прямоугольного треугольника определяет его стороны. Такая постановка задачи может быть формализацией вопроса типа: если садоводу разрешено выбрать участок определенной площади и оговорено, что в силу особенностей ландшафта он должен иметь форму прямоугольного треугольника, то может ли садовод обойтись имеющимся у него материалом для огораживания участка?

Математическая модель задачи записывается в виде приведенной ниже системы уравнений, где a, b – катеты треугольника, c – его гипотенуза, а S и P – площадь и периметр соответственно:

$$c^2 = a^2 + b^2,\ P = a + b + c,\ S = \frac{a \cdot b}{2}.$$

Цель решения системы – найти a, b и c. После преобразований мы получим квадратное уравнение вида:

$$(2P) \cdot x^2 - (P^2 + 4S) \cdot x + 4P \cdot S = 0,$$

вещественные решения которого дают значения катетов a и b.

Если дискриминант уравнения меньше нуля, построить треугольник при данных значениях площади и периметра невозможно. Принимая во внимание постановку содержательной задачи, можно потребовать, чтобы катеты были не меньше некоторой заранее оговоренной величины, например одного метра.

Перейдем теперь к проектированию, реализации и отладке программы. Однако прежде сделаем замечание методического характера. Рассматриваемый пример, безусловно, представляет собой очень простую задачу, для решения которой в реальной практике, конечно, не потребовалось бы использовать весь арсенал методов нисходящей разработки. Точно так же на примере конструирования прогулочной лодки трудно продемонстрировать, как создаются авианосцы. В то же время разбор сколь-нибудь серьезного по своим масштабам примера представляется практически невозможным – за неизбежным обилием деталей потеряются принципиальные идеи. Таким образом, нам придется довольствоваться небольшой программой и не обращать внимания на явную избыточность иерархии проектных решений и, соответственно, многомодульность результирующей программы.

Итак, приступим.

На первом шаге проектирования алгоритм решения задачи формулируется как простая последовательность действий: ввести данные об объекте обработки – участке, выполнить расчет и вывести результат, который является частью информационной модели участка. Будем считать, что *техническое задание* на программу требует реализовать возможность повторных расчетов в течение одного сеанса работы с программой. Тогда только что рассмотренный алгоритм должен быть телом цикла, повторяющего расчеты до выдачи команды завершения работы. Эти проектные решения уже могут быть зафиксированы в виде программы на языке Pascal:

```
{ ================================================ }
{ Пример 7.13 }
{ Основная программа }
Program Sizes;
```

```
{$APPTYPE CONSOLE}

uses DataLib1, Lib1;

var
  Lot: LotType;            { информационная модель участка }
begin
  repeat { повторять }
    Input(Lot);          { ввод модели участка              }
    Solve(Lot);          { решение                          }
    Output(Lot);         { вывод результата                 }
  until Finish;          { до получения команды завершения  }
end.
{ ================================================= }
```

В упомянутых в этой программе библиотеках DataLib1 и Lib1 разместим имитаторы подпрограмм и определений типов данных. В первой библиотеке будут строиться информационные модели. На этом шаге мы лишь сделаем предположение, что модель участка наиболее адекватно выражается типом данных *запись*:

```
{ ================================================= }
{ Пример 7.13 }
{ Библиотека информационных моделей - шаг 1 }
Unit DataLib1;
interface
type
   LotType = record
   end;

implementation
   end.
{ ================================================= }
```

В библиотеке Lib1 реализованы имитаторы всех модулей, упомянутых в главной программе, причем функция выхода имитируется с помощью датчика случайных чисел:

```
{ ================================================= }
{ Пример 7.13 }
{ Библиотека подпрограмм - шаг 1 }
Unit Lib1;
```

```
interface
uses DataLib1;
procedure Input(var Lot: LotType);
function Finish: Boolean;
procedure Solve(var Lot: LotType);
procedure Output(Lot: LotType);

implementation

function Finish: Boolean;
begin
   Finish := (Random < 0.5);
end;

procedure Input(var Lot: LotType);
begin
   WriteLn('Данные об участке введены...');
end;

procedure Solve(var Lot: LotType);
begin
   WriteLn('Расчет выполнен... ');
end;

procedure Output(Lot: LotType);
begin
   WriteLn('Результаты выведены...');
end;

begin
   Randomize;
end.
{ ================================================ }
```

Хотя по сути решения задачи сделано довольно мало, мы, тем не менее, уже реализовали заметную часть программы и даже можем отлаживать не только ее синтаксис, но и семантику, по крайней мере в части правильности реализации цикла.

Следующие шаги разработки состоят в уточнении понятий, определенных в библиотеках. Для того чтобы отладка стала управляемой, нам, прежде всего, следует реализовать функцию `Finish`.

Ее алгоритм может состоять в следующем: запрашиваем, следует ли продолжать работу, до тех пор, пока пользователь не даст синтаксически правильный ответ:

```
{ ================================================= }
```

```
{ Пример 7.13 }
{ Функция завершения работы программы }
function Finish: Boolean;
var
   Key: Char;
begin
   Write('Еще? (Y-Да/N-Нет)');
   repeat
      Readln(Key);
   until (UpCase(Key) = 'Y') or (UpCase(Key) = 'N');
   Finish := UpCase(Key) = 'N';
end;
{ ================================================ }
```

Уточним алгоритмы других процедур. Для записи процедуры ввода модели участка следует, прежде всего, уточнить способ представления участка в программе. Возьмем в качестве модели набор величин, характеризующих размеры участка: длины сторон, площадь и периметр. Кроме того, имеется еще одна важная характеристика, указывающая на корректность заданных величин. Эту характеристику мы отразим с помощью булевского поля `Exists` (существует?). Таким образом, уточненная библиотека `DataLib1` имеет вид:

```
{ ================================================ }
{ Пример 7.13 }
{ Библиотека информационных моделей - шаг 2 }
Unit DataLib1;
interface
type
   LotType = record
      S,              { площадь    }
      P,              { периметр   }
      a, b,           { катеты     }
      c :Real;        { гипотенуза }
      Exists: Boolean;   { признак корректности соотношения }
                         { геометрических характеристик     }
   end;

implementation

end.
{ ================================================ }
```

Процедура ввода модели участка должна запрашивать значения площади и периметра до тех пор, пока не будут введены синтаксически правильные значения (то есть числа, а не другие наборы символов). Кроме того, можно частично проверить семантику значений: они должны быть, по крайней мере, положительными.

Процедура расчета распадается на два последовательных действия – решение квадратного уравнения и вычисление размеров сторон.

Вывод состоит в печати значений сторон, если треугольник существует при исходных соотношениях площади и периметра, а также если стороны удовлетворяют заданным пороговым значениям (выполнение всех этих условий отражается в поле `Exists`).

После этих уточнений библиотека `Lib1` принимает следующий вид:

```
{ ================================================ }
{ Пример 7.13 }
{ Библиотека подпрограмм – финальный вариант }
Unit Lib1;
interface
uses DataLib1;

procedure Input(var Lot: LotType);
function Finish: Boolean;
procedure Solve(var Lot: LotType);
procedure Output(Lot :LotType);

implementation
uses Lib2;

{ Запрос на завершение работы }
function Finish: Boolean;
var
   Key: Char;

begin
   Write('Еще? (Y-Да/N-Нет)');
   repeat
      Readln(Key);
   until (UpCase(Key) = 'Y') or (UpCase(Key) = 'N');
   Finish := UpCase(Key) = 'N';
end;

{ Ввод площади и периметра }
procedure Input(var Lot: LotType);
   { Проверка корректности данных }
   function IsDataRight :Boolean;
   var
      IOR: Word;
```

```
   begin
      IOR := IOResult;
      with Lot do
      begin
         if IOR <> 0 then
         begin
           Write('Ошибка 1: Неправильный формат');
           Writeln('вещественного числа');
         end
         else
         if (P <= 0) and (S > 0) then
         begin
            Write('Ошибка 2: Недопустимые характеристики');
            Writeln('интервала');
         end;
         IsDataRight := (IOR = 0) and (P > 0) and (S > 0);
      end;
   end;

begin
   Writeln('Введите площадь и периметр участка');
   repeat
      {$I-}
      with Lot do
         Readln(S, P);
      {$I+}
   until IsDataRight;
end;

{ Решение }
procedure Solve(var Lot: LotType);
var
   Root: RootsType;   { корни уравнения }
begin
   Equation(Lot, Root);   { решение квадратного уравнения }
   Sides(Lot, Root);      { вычисление сторон треугольника }
end;

{ Вывод }
procedure Output(Lot: LotType);
begin
   with Lot do
      if not Exists then
         Writeln('При данных соотношениях площади и периметра',
           ' треугольник не существует')
      else
         Writeln('Катеты ', a:6:2, ' ',b:6:2,
```

```
          'Гипотенуза ', c:6:2);
end;

end.
{ ================================================ }
```

Для того чтобы программу можно было отлаживать, необходимо реализовать имитаторы определения типа RootsType, а также процедур Equation и Sides. Определение типа мы добавим в библиотеку DataLib1, а имитаторы процедур поместим в новую библиотеку Lib2:

```
{ ================================================ }
{ Пример 7.13 }
{ Библиотека информационных моделей - шаг 3 }
Unit DataLib1;
interface
type
   LotType = record
      S,                { площадь    }
      P,                { периметр   }
      a, b,             { катеты     }
      c: Real;          { гипотенуза }
      Exists: Boolean; { признак корректности соотношения }
                        { геометрических характеристик     }
   end;
   RootsType = record
   end;

implementation

end.
{ ================================================ }
```

```
{ ================================================ }
{ Пример 7.13 }
{ Библиотека вспомогательных подпрограмм – шаг 1 }
Unit Lib2;
interface
uses DataLib1;

procedure Equation(Lot: LotType; var Root :RootsType);
procedure Sides(var Lot: LotType; Root :RootsType);

implementation

{ Имитатор решения уравнения }
procedure Equation(Lot: LotType; var Root: RootsType);
begin
end;

{ Имитатор вычисления сторон }
procedure Sides(var Lot: LotType; Root: RootsType);
begin
   with Lot do
   begin
      Exists := (Random < 0.5);
      a := 3;
      b := 4;
      c := 5;
   end;
end;

begin
   Randomize;
end.
{ ================================================ }
```

Написанная часть программы практически полностью отражает структуру программного комплекса. Определен интерфейс всех подпрограмм. Эту часть можно отлаживать и модифицировать, например совершенствовать процедуры ввода и вывода. Но при этом мы фактически еще не приступали к реализации математического метода решения! Проделанная нами работа демонстрирует содержание труда проектировщика программной системы.

Следующий шаг – уточнение процедур из библиотеки `Lib2`. Для этого следует разобраться со способом представления значений типа `RootsType`. Квадратное уравнение имеет два корня, которые мы обозначим как `X1` и `X2`. Нас интересуют только вещественные значения

корней, поэтому мы введем логический признак Complex, истинное значение которого будет означать, что в нашем случае решение уравнения отсутствует. Кроме того, как мы договаривались ранее, следует определить пороговые значения для сторон треугольника. Мы введем их в виде поименованных констант. Окончательный вариант библиотеки DataLib1 выглядит следующим образом:

```
{ ================================================== }
{ Пример 7.13 }
{ Библиотека информационных моделей - финальный вариант }
Unit DataLib1;
interface
const
   Min_a = 1;
   Min_b = 1;
   Min_c = 1;
type
   LotType = record
      S,           { площадь     }
      P,           { периметр    }
      a, b,        { катеты      }
      c: Real;     { гипотенуза }
      Exists: Boolean;   { признак корректности соотношения }
                         { геометрических характеристик     }
   end;
   RootsType = record
      Complex: Boolean;
      X1, X2: Real;
   end;

implementation

end.
{ ================================================== }
```

Поскольку способ реализации процедур Equation и Sides очевиден, мы без дополнительных комментариев приведем окончательный вид библиотеки Lib2:

```
{ ================================================== }
{ Пример 7.13 }
{ Библиотека вспомогательных подпрограмм - финальный вариант }
Unit Lib2;
interface
uses DataLib1;
procedure Equation(Lot: LotType; var Root :RootsType);
```

```
procedure Sides(var Lot: LotType; Root :RootsType);

implementation

procedure Equation(Lot: LotType; var Root: RootsType);
var
   A, B, C, D: Real;
begin
   with Root do
   begin
      A := Lot.P + Lot.P;
      B := -(Sqr(Lot.P) + 4 * Lot.S);
      C := 4 * Lot.P * Lot.S;
      D := Sqr(B) - 4 * A * C;
      Complex := D < 0;
      if not Complex then
      begin
         X1 := (-B + Sqrt(D)) / (A + A);
         X2 := (-B - Sqrt(D)) / (A + A);
      end;
   end;
end;

procedure Sides(var Lot: LotType; Root: RootsType);
begin
   with Lot, Root do
   begin
      Exists := not Complex;
      if Exists then
      begin
         a := X1;
         b := X2;
         c := P - a - b;
         Exists := (c >= Min_c) and (a >= Min_a) and (b >= Min_b);
      end;
   end;
end;

end.
{ ============================================== }
```

Итак, сделаем некоторые выводы из рассмотренного нами примера нисходящей разработки программы и сформулируем ряд общих положений этой технологии.

Часто можно услышать мнение, что основная трудность решения задачи с использованием компьютера заключается в разработке и реализации

математического метода. Не отрицая ни в коем случае основополагающую роль решения этой проблемы, следует отметить, что, как видно из примера, процесс проектирования и реализации частей, формально не относящихся к собственно расчетным процедурам, а также проектирования системы в целом занимает существенный, а часто и значительно больший объем работ.

Полученная в результате пошагового проектирования «сверху вниз» структура программы может быть интерпретирована как иерархия инструментальных слоев или, как еще говорят, *виртуальных машин*. В нашем примере программа `Sizes`, можно сказать, реализована на виртуальной машине `Lib1` плюс `DataLib1`, а `Lib1`, в свою очередь, на виртуальной машине `Lib2`. Модификация программы может производиться путем замены виртуальной машины нижнего уровня без изменений в верхней части иерархии. Например, если мы решили усовершенствовать интерфейс с пользователем в нашей программе, нам достаточно переработать «машину» `Lib1`, в то время как головная программа остается без изменений.

Рассмотренная технология обеспечивает возможность параллельной коллективной разработки в целом. Хотя это, конечно, простой пример, но, тем не менее, отметим, что программу `Sizes` можно было начинать реализовывать и отлаживать даже до того, как придуман математический метод решения задачи.

12. Выводы

Настоящая глава посвящена рассмотрению одной из ключевых концепций в проектировании и разработке программ – модульному программированию. Основная идея модульного программирования – разбиение задачи на более простые, максимально независимые друг от друга подзадачи – модули – явилась мощным катализатором в развитии процесса коллективной разработки программ. Возможность создавать отдельные модули, впоследствии объединяя их в одну программу, привела к появлению технологий разделения труда между членами программистского коллектива и, наконец-то, позволила перевести количество рабочих рук (или, что вернее, голов) в качество и быстроту разработки программного продукта.

В главе рассмотрены основные элементы технологии модульного программирования и их реализация в языке программирования Object Pascal. Изучены как базовые (понятие подпрограммы, модуля), так и

сложные вопросы (процедурный тип данных, полиморфизм, нисходящее проектирование).

Литература

1. Богатырев Р. Летопись языков Паскаль // Мир ПК. –№ 4.–2001.

2. Богатырев Р. От Паскаля к языку Zonnon: реализация новых идей на платформе .NET // Мир ПК.–2003.–№ 9.

3. Большая советская энциклопедия. Издание третье. – М.: Советская энциклопедия, 1970.

4. Брэдли Д. Программирование на языке ассемблера для персональной ЭВМ фирмы IBM.–М.: Радио и связь, 1988.

5. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++. Второе издание. – Бином, 1998.

6. Касперски К. Техника оптимизации программ. Эффективное использование памяти.– СПб.: БХВ-Петербург, 2003.

7. Кетков А., Кетков Ю. Практика программирования: Бейсик, Си, Паскаль. Самоучитель.–СПб.: БХВ-Петербург, 2001.

8. Кетков А., Кетков Ю. Практика программирования: Visual Basic, C++ Builder, Delphi. Самоучитель.–СПб.: БХВ-Петербург, 2002.

9. Кнут Д.Э. Искусство программирования. Т. 1. Основные алгоритмы.–М.:Вильямс, 2000.

10. Кнут Д.Э. Искусство программирования. Т. 3. Сортировка и поиск.–М.:Вильямс, 2004.

11. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы. Построение и анализ.–М.:МЦНМО, 1999.

12. Кэнту М. Delphi 7 для профессионалов.–Спб.:Питер, 2004.

13. Лингер Р., Миллс Х., Уитт Б. Теория и практика структурного программирования.–М.: Мир, 1982.

14. Модернизация и ремонт персонального компьютера.– [http://www.allcompinfo.com]

15. Новичков А. Система генерации проектной документации Rational SoDA.– [http://www.citforum.ru/programming/digest/soda.shtml]

16. Олифер В.Г., Олифер Н.А. Сетевые операционные системы.–СПб.: Питер, 2002.

17. Рихтер Дж. Windows для профессионалов: создание эффективных Win32 приложений с учетом специфики 64-разрядной версии Windows / Пер. с англ.– 4-е изд.–СПб.: Питер; М.: Издательско-торговый дом «Русская Редакция», 2001.

18. Служба тематических толковых словарей. – [http://www.glossary.ru/]

19. Стивенс Р. Delphi. Готовые алгоритмы.– Спб.:Питер, 2004.

20. Тейксейра С., Пачеко К. Borland Delphi 6. Руководство разработчика. –Вильямс, 2002.

21. Шень А. Программирование: теоремы и задачи.–М.:МЦНМО, 2004.

22. Юров В. Assembler.–СПб.: Питер, 2002.

23. Cantu M. Essential Delphi.– [www.marcocantu.com/edelphi]

24. Cantu M. Essential Pascal.– [www.marcocantu.com/epascal]

25. Cantu M. Mastering Borland Delphi 2005.–Sybex, 2005.

26. Cantu M. Mastering Delphi 7.–Sybex, 2003.

27. Clark R., Koehler S. The UCSD Pascal Handbook.–Prentice-Hall, 1982.

28. Dahl O., Dijkstra E., Hoare C.A.R. Structured Programming.–London, England: Academic Press, 1972.

29. Gutknecht J., Zueff E. Zonnon Language Report. Draft.–ETH Zurich, June 2003.

30. Gutknecht J., Zueff E. Zonnon for .NET, A Language and Compiler Experiment. // Computer Systems Institute, ETH Zürich, Switzerland. JMLC Conference.–August 2003.

31. Gutknecht J. Zonnon: A .NET Language Beyond C# // Moscow: Microsoft Conference, June 15-17, 2003.

32. Gutknecht J., Wirth N. The Oberon System // Software – Practice and Experience.–Vol.19, № 9.–1989.–p.857-893.

33. Jacobson I., Christerson M., Jonsson P., Overgaard G. Object-oriented Software Engineering.–Workingham, England: Addison-Wesley Publishing Company, 1992.

34. Jensen K., Wirth N. PASCAL – User Manual and Report, ISO Pascal Standard.–1974.

35. Intel Architecture Software Developer's Manual. Volume 1: Basic Architecture.– Intel Corporation: 1997.

36. Meyer B. Object-oriented Software Construction.–Prentice Hall, 1988.

37. Rumbaugh H., Blaha M., Premarlani W., Eddy F., Lorensen W. Object-Oriented Modeling and Design.–Prentice-Hall, 1995.

38. Sedgewick R. Algorithms.–Reading, MA: Addison-Wesley, 1983.

39. Tesler L. Object Pascal Report. // Structured Language World, Vol.9, No.3.– 1985.–p. 10-14.

40. Thiriez H. Modelling of an interactive scheduling system in a complex environment // European Jornal of Operational Research. –1991.– №50.–p.37-47.

41. Wirth N. Program Development by Stepwise Refinement // Communications of the ACM vol.26(1).– January 1983.

42. Wirth N. The Programming Language Pascal // Acta Informatica, 1.–1971.–p. 35-63.

43. Wirth N. Programming in Modula-2.–Springer-Verlag, 1982.

44. Wirth N. The programming language Oberon // Software – Practice and Experience, Vol.18, № 7.–1988.–p.671-690.

45. Wirth N. Programming in Modula-2.–Springer, 1974.

46. Wirth N. Pascal and its Successors // Conference on Computer Pioneers.–Bonn, 2001.