ГЛАВА 9

Динамическое управление памятью

Все жалуются на свою память, но никто не жалуется на свой разум.

Франсуа де Ларошфуко

Умелая *работа с памятью* – один из наиболее важных элементов мастерства программиста. От того, как будет написан соответствующий код, обычно зависит не только производительность программы, но и ее работоспособность. Ошибки, допущенные при работе с памятью, обходятся очень дорого на этапе отладки программы, требуя привлечения больших временных ресурсов для их обнаружения.

Наш опыт подсказывает, что для грамотной работы с памятью прежде всего необходимо понимать, как эта работа «устроена изнутри», какие действия происходят, когда компилятор преобразует наши скупые синтаксические конструкции в *машинный код*. Для достижения этого понимания нужно немного потрудиться, чем мы сейчас и займемся. В начале мы рассмотрим общие принципы, не зависящие от языка программирования и реализации компилятора, и лишь потом конкретную реализацию изученных механизмов в языке Object Pascal и системе Borland Delphi.

Приступим!

1. Проблемы работы с памятью в многозадачной операционной системе

Для начала, слегка коснемся некоторых важных системных вопросов. Конечно, заявленная в заголовке раздела тема значительно лучше подходит для курса «Операционные системы» и, вообще говоря, заслуживает длительного разговора [16, 17]. Мы постараемся ограничиться в данной книге минимумом, необходимым для понимания материала.

Итак, будем считать, что мы работаем в *многозадачной операционной системе*, в частности Windows.

Принцип многозадачности подразумевает возможность одновременной работы нескольких программ. Вспомним, как происходит работа в

однозадачной системе типа MS DOS. Вот вы запустили какую-то программу, например игрушку, и не успели погрузиться в таинственный мир лабиринта и нацелить любимую базуку на вражеский корабль, как к вам подскочил приятель и сообщил «радостную» новость: необходимо срочно отформатировать дискету. Что делать? Увы, вы вынуждены записываться, выходить из программы, форматировать дискету, запускать программу снова, загружаться... Неудобно, не правда ли? А все потому, что в однозадачной операционной системе не может выполняться более одной программы единовременно.

Попробуем задуматься над вопросом – как работает многозадачность? Компьютер, на котором исполняется множество программ, один, процессор в нем (по крайней мере на момент написания этой книги) чаще всего тоже, и память, и видеоадаптер и т.д. Не вдаваясь в подробности, можно заключить, что несколько одновременно запущенных программ должны каким-то образом делить между собой системные ресурсы, в частности процессорное время и оперативную память. Подумаем – какие проблемы могут возникнуть при совместной работе программ с памятью?

Прежде всего, кто-то должен следить за тем, чтобы программе была своевременно предоставлена необходимая память. Затем необходимо решение проблемы несанкционированного доступа одной программы к памяти, занятой другой. Далее, нужно решать вопросы, связанные с надежностью, производительностью и, наконец, просто возможностью нехватки оперативной памяти, вытекающей из ее конечного объема. Все это берет на себя операционная система, фактически устраняя программиста от работы с аппаратурой напрямую. Так, все попытки запросить или освободить память, равно как и операции доступа, проходят под чутким контролем операционной системы. Более того, функции по работе с памятью (выделение, удаление и другие), предоставляемые языками программирования, и в частности языком Object Pascal (соответствующие возможности будут рассмотрены далее в этой главе), на самом деле реализованы на функциях операционной системы из состава так называемого Windows API – Application Programming Interface.

Что касается конечного объема оперативной памяти, установленной в компьютере, частично эта проблема решается с помощью механизма операционной системы под названием *виртуальная память*. Коротко: этот механизм позволяет интерпретировать свободное место на жестком диске как продолжение оперативной памяти, а его «физическим» проявлением является создание на диске *файла подкачки*. Конечно, эта виртуальная память работает существенно медленнее оперативной (жесткий диск, как известно, – устройство с элементами механики), что заставляет операционную систему применять различные оптимизационные алгоритмы, чтобы повысить производительность работы программ. Не

вдаваясь в детали этих алгоритмов, укажем, что они основаны на страничной организации оперативной памяти и стратегиях замещения страниц.

Все случаи разделения ресурсов между несколькими потребителями порождают проблему одновременного доступа потребителей к одному ресурсу. В случае с памятью Windows решает эту проблему с помощью механизма *виртуального адресного пространства* – адреса оперативной памяти, с которыми работают одновременно запущенные программы, на самом деле являются логическими, трансляцией их в реальные занимается операционная система, исключая в результате конфликты использования программами одних и тех же участков памяти.

2. Адресное пространство программы

Понимание внутреннего устройства адресного пространства программы – важный шаг в изучении данной главы. Поговорим об этом подробнее.

Пусть у нас есть некоторая программа на Object Pascal. В силу определений стандарта языка она выглядит примерно так:

```
Program <Имя программы>;
uses <Список подключаемых модулей>;
<Блок объявлений глобальных переменных, констант и типов данных (var, const, type)>
<Блок подпрограмм (procedure, function)>
begin
   <Тело программы>
end.
```

Рассмотрим пример программы, которая реализует функцию, вычисляющую среднее арифметическое двух чисел.

```
{ ================================================== }
{ Пример 9.1 }
{ Вычисление среднего арифметического двух чисел }
Program Simple;

{$APPTYPE CONSOLE}

var
   a, b: Integer;
   c: Real;

function Average(first, second: Integer): Real;
var
   res: Real;
```

```
begin
   res := (first + second) / 2;
   Result := res;
end;
begin
   Write('Введите числа: ');
   ReadLn(a, b);
   c := Average(a, b);
   WriteLn('Average(a, b) = ', c);
   ReadLn;
end.
{ ========================================== }
```

Попробуем разобраться, что собой представляет эта программа, будучи запущенной на исполнение.

Не вдаваясь в подробности, связанные с устройством и хранением служебной информации, и еще в некоторые системные вещи, заметим, что ситуация выглядит так:

1. При запуске программы ей выделяется некоторый *диапазон адресов*. В 32-разрядных системах Microsoft Windows адрес представлял собой целое неотрицательное число, содержащееся в 32 двоичных разрядах. Соответственно, мы имеем 2^{32} разных адресов, что позволяет адресовать 4 гигабайта памяти. Надо понимать, что реальный объем ОЗУ меньше 4 гигабайт и начиная с некоторого момента адресуемые ячейки памяти на самом деле будут находиться в так называемой виртуальной памяти, т.е. в файле подкачки на жестком диске. Кроме того, существенная часть этих 4 гигабайт занята служебной информацией и не может быть использована нами. Тем не менее, того, что остается, обычно более чем достаточно. Если же нет? Это тема отдельной книги[1].

2. Выделенный программе диапазон адресов – адреса в диапазоне $0 – 2^{32}–1$. Такой диапазон имеет каждая программа. Как мы уже говорили, адреса, которыми оперируют программы, являются виртуальными, а Windows производит их трансляцию в реальные, обеспечивая то, что адресные пространства разных программ не пересекаются. Более того, тем самым обеспечивается защита данных и кода, так как одна программа не может «добраться» до адресного пространства другой. Так, для рассмотренного примера *Сегмент кода* содержит одну подпрограмму и головную программу.

---

[1] Интересующимся прежде всего рекомендуем прочитать книгу: 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. – 752 с.

3. При запуске программы в адресном пространстве выделяются несколько областей. Во-первых, это *Сегмент кода* – область памяти, в которой располагаются машинные инструкции, которые, собственно, и со-

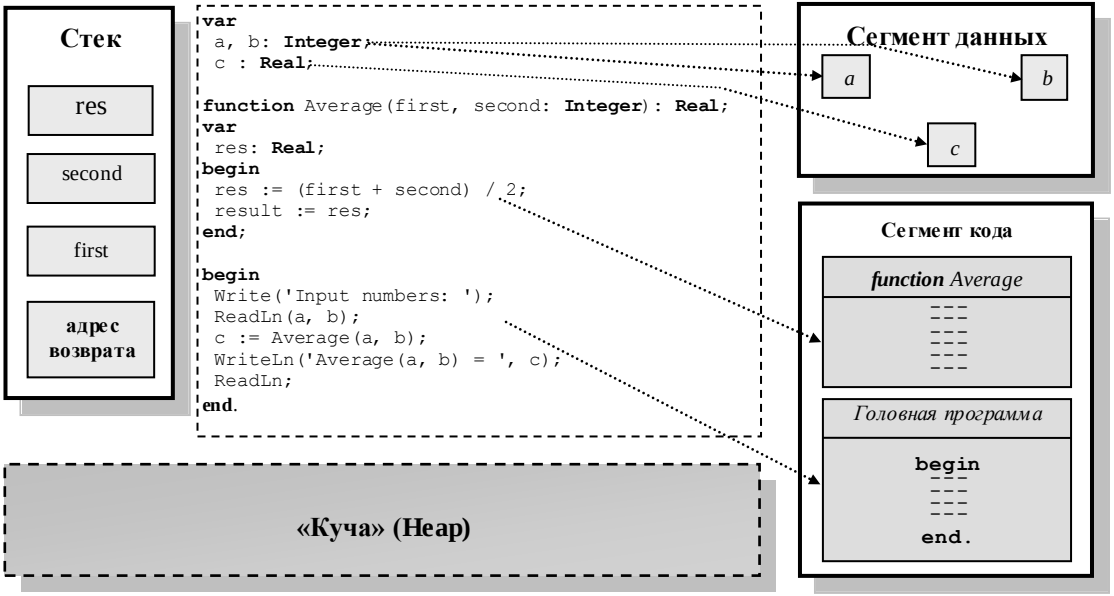


Рис. 9.1. Адресное пространство программы

ставляют программу. Можно выделить инструкции, составляющие головную программу, и инструкции, составляющие тела подпрограмм. Во-вторых, это *Сегмент данных* – область памяти, в которой располагаются глобальные переменные программы. Так, для рассмотренного примера это переменные *a, b* и *c*. В-третьих, это *Сегмент стека*. *Стек* – специальная структура, работающая по следующему принципу: пришедший первым элемент уходит последним (представьте себе, что вы складываете в стакан монеты одну за другой, а потом начинаете их доставать).
В стеке размещаются локальные переменные подпрограмм, а также вся информация, которая требуется для их корректной работы. В частности, это, например, адрес возврата. Заметим, что в отличие от Сегмента кода и Сегмента данных, информация в которых существует с момента старта программы до ее окончания, данные в стеке находятся лишь в момент работы подпрограммы. После завершения работы подпрограммы вся связанная с ней информация из стека удаляется. Для рассмотренного примера в момент работы подпрограммы `Average` стек будет содержать адрес возврата и локальные переменные `first`, `second` и `res`. По окончании работы функции будут удалены из стека локальные переменные и выбран адрес возврата для перехода по нему в точку, из которой был осуществлен запуск функции.

4. Заметим на рисунке таинственную *Кучу*. *Куча* (англ. *Heap*) – специальная область памяти, которая будет изучена в следующем разделе.

3. Динамическое управление памятью в языке Object Pascal

Рассмотрев устройство адресного пространства программы, самое время перейти к проблеме его практического использования. Как мы уже знаем, память для глобальных и локальных переменных, а также параметров подпрограмм выделяется компилятором автоматически. Более того, работа с адресами памяти, по которым размещаются переменные, скрыта от нас за их именами. А что если мы хотим поработать с адресами самостоятельно? Для этого существует специальная конструкция языка программирования Object Pascal – *указатель*.

Указатели бывают двух видов – *типизированные* и *бестиповые*. Начнем изучение с наиболее часто используемых типизированных указателей.

3.1. Работа с адресами. Типизированные указатели

Типизированный указатель – адрес области оперативной памяти, содержимое которой интерпретируется в соответствии с заявленным при объявлении указателя типом данных. Синтаксис объявления типа данных – указателя выглядит так.

```
type
  <Имя типа данных-указателя> = ^<Тип данных>;
```

Так, например, в следующем примере мы объявляем новый тип данных (указатель) `PInteger`. Переменные типа `PInteger` будут хранить адреса ячеек оперативной памяти, причем содержимое этих ячеек будет интерпретировано как значение типа `Integer`.

```
type
  PInteger = ^Integer;
```

В дальнейшем мы можем использовать объявленный тип данных так:

```
var
  p: PInteger;
```

Заметим, что можно объявить указатель не только на стандартный тип данных, но и на тип, созданный программистом, как в следующем примере:

```
type
  TPerson = record
    FIO: String[50];
    BirthdayY: Word;
    BirthdayM: Byte;
    BirthdayD: Byte;
    Phone: String[15];
  end;
  PPerson = ^Person;
```

Кроме того, допустим сокращенный синтаксис объявления указателя:

```
var
  p1: ^Integer;
```

Охарактеризуем указатели с точки зрения определения типа данных. Множество значений типа «указатель» напрямую зависит от размера виртуального адресного пространства, в котором работает программа. В 32-разрядных операционных системах этот размер, как мы уже обсуждали, равен 4 гигабайтам. Поскольку минимально адресуемой ячейкой памяти является один байт, мы получаем 2^{32} адресов, то есть

множество значений типа «указатель» составляют целые числа от 0 до 2^{32}-1. Заметим, что в 64-разрядных операционных системах, работающих на 64-разрядных процессорах, и указатели имеют диапазон значений от 0 до 2^{64}-1. Размер памяти под переменные типа «указатель» в точности определяется диапазоном значений и составляет 4 байта в 32-разрядном случае и 8 байт в 64-разрядном.

Разговор об операциях, применимых к указателям, начнем с инициализации. Наиболее часто используемый способ установки начального значения указателя состоит во взятии адреса существующей переменной (или подпрограммы). Для этого можно применять практически эквивалентные: операцию взятия адреса `@` и функцию `Addr` библиотеки `System`, автоматически подключаемой к программе. Рассмотрим первый вариант, связанный с использованием операции взятия адреса (о втором варианте ниже при обсуждении бестиповых указателей).

```
{ ================================================ }
{ Пример 9.2 }
{ Операция взятия адреса }
Program Pointers;

{$APPTYPE CONSOLE}

type
   PInteger = ^Integer;
var
   p1, p2, p3: PInteger;
   i1, i2, i3: Integer;
begin
   i1 := 1;
   i2 := 2;
   i3 := 3;
   p1 := @i1;
   p3 := @i3;
   p2 := p1;
end.
{ ================================================ }
```

Не правда ли, все довольно просто? Тем не менее опыт показывает, что работа с указателями относится к числу весьма тяжело усваиваемых тем, поэтому прокомментируем приведенный код:

1. Вначале мы присваиваем переменным `i1`, `i2` и `i3` значения `1`, `2` и `3` соответственно.

2. Далее сохраняем в переменной p1 адрес переменной i1.

3. После этого сохраняем в переменной p3 адрес переменной i3.

4. И, наконец, в переменную p2 записываем тот же адрес, который хранится в переменной p1, т.е. адрес i1.

5. В результате мы настроили указатели следующим образом: p1 и p2 указывают на i1, а p3 указывает на i3.

Итак, инициализировать указатели мы научились. Неплохо было бы теперь иметь возможность добраться до значения, лежащего в ячейках памяти, на которые он указывает. Конечно же, в языке Object Pascal такая операция имеется. Эта операция в некотором смысле осуществляет действия, обратные операции взятия адреса, и называется *операцией разыменования*.

Операция разыменования для типизированного указателя p записывается как p^. Запись p^ означает: в оперативной памяти рассмотреть значения в ячейках начиная с адреса p и интерпретировать в соответствии с типом указателя.

Усовершенствуем предыдущий пример:

```
{ ================================================= }
{ Пример 9.3 }
{ Операция разыменования }
Program Pointers;

{$APPTYPE CONSOLE}

type
   PInteger = ^Integer;
var
   p1, p2, p3: PInteger;
   i1, i2, i3: Integer;
begin
   i1 := 1;
   i2 := 2;
   i3 := 3;
   p1 := @i1;
   p3 := @i3;
   p2 := p1;
   p2^ := 5;
   p3^ := p1^ + p2^;
   WriteLn(i1, ' ', i2, ' ', i3);
   ReadLn;
end.
{ ================================================= }
```

Приведем комментарии к добавленным строкам кода.

1. По адресу, который хранится в `p2`, записываем число `5` (т.е. теперь `i1 = 5`).

2. По адресу, который хранится в `p3`, записываем сумму значений, хранящихся по адресам `p1` и `p2` (т.е. теперь `i3 = i1 + i1 = 10`).

3. Запускаем программу, результат выглядит так: `5 2 10`.

В заключение обращаем ваше внимание на следующие важные моменты:

1. Операция взятия адреса не может быть применена к числу или константе! Адрес есть лишь у тех объектов, которые хранятся в памяти.

2. Память, выделенная под переменную-указатель при ее объявлении, инициализируется по умолчанию нулевым значением. В Object Pascal существует возможность явно присвоить указателю нулевой адрес, для этого введена специальная константа **`nil`** (пишут `p1 := nil;`). Попытка доступа к значению, хранящемуся по нулевому адресу, вызывает ошибку времени исполнения программы!

3. К сожалению, Object Pascal не обладает большим набором операций по работе с указателями. Так, мы не можем прибавить к адресу число (т.е. не можем напрямую при помощи встроенных средств, однако мы можем создать эти средства самостоятельно). Тем не менее в дополнение к рассмотренным операциям к типизированным указателям могут быть применены операции сравнения = и <>, результат которых зависит от того, хранят указатели один и тот же адрес или нет.

Из рассмотренного выше типизированный указатель предстает перед нами как средство создания синонимов для имен переменных. На самом деле его применение значительно шире, что и будет показано далее в этой главе.

3.2. Работа с адресами. Бестиповые указатели

Наряду с типизированными указателями, хранящими адрес вместе с информацией о типе тех данных, что располагаются по этому адресу, существуют так называемые *бестиповые указатели*, у которых отсутствует какая-либо привязка к конкретному типу данных. Для поддержки таких указателей существует тип `Pointer`.

Переменные типа `Pointer` практически во всем подобны типизированным указателям, кроме того факта, что к ним неприменима операция разыменования, поскольку неизвестно, как интерпретировать обнаруженную по адресу информацию. Тем не менее добраться до значения, хранящегося по адресу в указателе типа `Pointer`, возможно,

выполнив предварительно приведение типа от `Pointer` к какому-либо типизированному указателю. Рассмотрим соответствующий пример.

```
{ ================================================= }
{ Пример 9.4 }
{ Бестиповые указатели }
Program Untype_pointers;

{$APPTYPE CONSOLE}

type
   PWord = ^Word;
var
   pi: PWord;
   p: Pointer;
   i: Word;
begin
   i := 32008;
   p := @i;
   pi := p;
   WriteLn(i, ' ', pi^);
   ReadLn;
end.
{ ================================================= }
```

Выполнив присваивание `pi := p;` мы скопировали адрес в переменную `pi` и получили возможность использовать операцию разыменования для доступа к значению, хранящемуся по этому адресу. В результате имеем: 32008 32008.

Изменим пример, объявив `pi` как указатель на `Byte`.

```
{ ================================================= }
{ Пример 9.5 }
{ Бестиповые указатели. Особенности интерпретации }
Program Untype_pointers;

{$APPTYPE CONSOLE}

type
   PByte = ^Byte;
var
   pi: PByte;
   p: Pointer;
   i: Word;
begin
   i := 32008;
   p := @i;
   pi := p;
```

```
   WriteLn(i, ' ', pi^);
   ReadLn;
end.
{ ================================================= }
```

Запускаем программу и видим следующий результат: 32008 8.

Почему же в результате мы получаем 8? Ответ прост: по адресу, на который указывает `pi`, располагается переменная `i`. Эта переменная принадлежит к типу `Word` и представляет собой двухбайтовое неотрицательное целое число. Попробуем понять, как хранится это число. Для этого рассмотрим следующее разложение: 32008 = 125 * 256 + 8. Таким образом, число 32008 хранится в виде совокупности старшего байта (равного 125) и младшего байта (равного 8). Архитектура IBM PC предполагает, что числа хранятся в памяти в перевернутом виде, сначала младший байт, потом старший. Соответственно, преобразовав `pi` к типу «Указатель на байт», мы получаем при его разыменовании первый байт, относящийся к числу 32008, т.е. младший байт, равный 8.

Бестиповые указатели применяются в тех случаях, когда способ интерпретации значений, хранящихся в оперативной памяти, неизвестен. Впоследствии, как правило, происходит преобразование к типизированному указателю. Заметим, что тип `Pointer` удобен тем, что он совместим по присваиванию с указателем на любой тип данных.

В дополнение к существующим для типизированных указателей операциям язык Object Pascal предоставляет функции `Addr` и `Ptr`, которые могут применяться к бестиповым указателям:

```
function Addr(X): Pointer;
```

Функция `Addr` является аналогом операции `@` и выполняет сходные действия, возвращая адрес объекта `X`, хранящегося в памяти. Результат имеет тип `Pointer`.

```
function Ptr(Address: Integer): Pointer;
```

Функция `Ptr` восполняет некоторый недостаток адресной арифметики Object Pascal, преобразуя целое число в бестиповый указатель. Рассмотрим пример.

```
{ ================================================= }
{ Пример 9.6 }
{ Бестиповые указатели. Функции Addr и Ptr }
Program Untype_pointers2;

{$APPTYPE CONSOLE}

type
   PByte = ^Byte;
```

```
var
   pi1, pi2: PByte;
   p: Pointer;
   i: Word;
begin
   i := 32008;
   p := Addr(i);
   pi1 := p;
   pi2 := Ptr(Integer(p) + 1);
   WriteLn(i, ': старший байт: ', pi2^, '; младший байт: ', pi1^);
   ReadLn;
end.
{ ================================================= }
```

Разберем переменную `i` на составляющие ее байты. Для этого известным нам способом получим доступ к младшему байту числа через указатель `pi1`. Для того чтобы получить доступ к старшему байту, необходимо прибавить к `p` единицу. Сделать это можно преобразовав `p` к целому типу, прибавив 1, а затем интерпретируя результат как адрес. В результате имеем на экране:

```
32008: старший байт: 125; младший байт: 8.
```

В следующем примере применим другую технику для разбора числа на составные части, используя массивы:

```
{ ================================================= }
{ Пример 9.7 }
{ Указатели и массивы }
Program Pointers_and_arrays;

{$APPTYPE CONSOLE}

type
   PByte = ^Byte;
   TByteArray = array [0..1] of Byte;
var
   i: Word;
   pBuffer: ^TByteArray;
begin
   i := 32008;
   pBuffer := @i;
   WriteLn(i, ': старший байт: ', pBuffer[1],
     '; младший байт: ',pBuffer[0]);
   ReadLn;
end.
{ ========================================= }
```

Надеемся, что представленный код в дополнительных комментариях не нуждается.

3.3. Статическое и динамическое распределение памяти

Поговорим об управлении памятью. Принципиальный вопрос: когда выделяется необходимая программе память и когда эта память освобождается, т.е. возвращается системе? Интуитивно понятны два возможных варианта.

Первый подразумевает, что вся необходимая программе (или подпрограмме) память выделяется в начале ее работы и освобождается по окончании. С этим вариантом мы работали на протяжении всех предыдущих глав. Соответствующие инструкции автоматически подставляются в код компилятором. В этом случае говорят о *статическом распределении памяти*. К достоинствам данного подхода следует отнести простоту, производительность (возможность оптимизации времени работы) и сравнительно небольшое количество «узких мест», приводящих к ошибкам. Недостаток же всего один, но зато очень серьезный. Он состоит в том, что в рамках статического распределения памяти мы обязаны заранее точно знать, сколько памяти нам понадобится, что бывает далеко не всегда. Вспомним ситуацию с объявлением массива. Ввиду того, что мы не знаем количества элементов на этапе написания программы, мы вынуждены заводить массив заведомо большего размера, в результате чего существенная часть памяти выделяется просто так и вообще не используется. Заметим также, что мы не имеем возможности вернуть неиспользуемую память системе. Справляется с этими недостатками второй вариант – *динамическое распределение памяти*.

Динамическое распределение памяти предполагает выделение и освобождение памяти в те моменты, когда это действительно необходимо во время работы программы. Язык Object Pascal предоставляет богатые возможности для работы с динамической памятью: кроме операций выделения/освобождения памяти, в язык встроены готовые динамические структуры, такие, как динамические массивы и длинные строки, скрывающие от пользователя особенности внутренней реализации и существенно упрощающие работу. В предыдущих главах мы изучили практически все, что касается статической организации данных, перейдем теперь к работе с данными динамическим способом.

3.4. Динамическое распределение памяти

в языке программирования Object Pascal

Вспомним рисунок 9.1 и комментарии к нему в начале главы. Один из элементов рисунка – *Куча (Heap)* – остался неизученным. Настала пора с ним разобраться. Итак, куча – это специальная область в адресном пространстве программы, в которой выделяется память, запрашиваемая динамически, в момент работы программы.

Object Pascal содержит две группы подпрограмм для осуществления операций выделения/освобождения памяти в куче. Рассмотрим их.

3.4.1. Работа с памятью в стиле `GetMem`, `FreeMem`. Первая группа предназначена для выделения (освобождения, изменения размеров) памяти с заданием размера обрабатываемого блока ячеек в байтах и работы с этим блоком при помощи механизма бестиповых указателей:

```
procedure GetMem(var P: Pointer; Size: Integer);
```

Процедура `GetMem` выделяет блок памяти размером `Size` (в байтах) и возвращает адрес выделенного блока в переменную-указатель `P`. В случае если выделить память не удалось, возникает ошибка времени исполнения.

```
procedure FreeMem(var P: Pointer);
```

Процедура `FreeMem` применяется при необходимости освободить память, выделенную ранее при помощи `GetMem`. Переменная `P` содержит указатель на эту память. Освобождение памяти означает, что память становится свободной и может быть выделена при следующем обращении к `GetMem`. Заметим, что повторное освобождение одного и того же участка памяти приводит к ошибке. Значение переменной `P` по окончании `FreeMem` не определено.

```
procedure ReallocMem(var P: Pointer; Size: Integer);
```

Процедура `ReallocMem` производит перераспределение ранее выделенного участка памяти, адрес начала которого содержится в переменной `P`, а новый размер – в переменной `Size`. При изменении размера выделенного блока данные, находившиеся в блоке, сохраняются.

Кроме рассмотренных средств, Delphi содержит еще несколько подпрограмм, например `AllocMem`, которая отличается от `GetMem` тем, что инициализирует память нулевыми значениями. С полным списком подпрограмм и их назначением желающие могут ознакомиться в документации или справочной системе Delphi.

Рассмотрим в качестве демонстрационного примера математическую задачу: найти сумму квадратов первых n натуральных чисел.

Издавна известна следующая формула:

$$1^2 + 2^2 + \ldots + n^2 = \frac{n(n+1)(2n+1)}{6},\ n \in N. \qquad (9.1)$$

Приведем ее доказательство, воспользовавшись *методом математической индукции*, который состоит в том, что доказываемое утверждение проверяется для начального n (обычно для $n = 1$) и показывается, что из справедливости при $n = k$ следует справедливость при $n = k + 1$.

Доказательство:

1. При $n = 1$ имеем: $1^2 = 1$,

$$\frac{1(1+1)(2+1)}{6} = 1, \text{ что и требовалось.}$$

2. Предположим справедливость утверждения при $n = k$, т.е.

$$1^2 + 2^2 + \ldots + k^2 = \frac{k(k+1)(2k+1)}{6}, \text{ где } k \in N.$$

Покажем справедливость утверждения при $n = k + 1$, т.е.

$$1^2 + 2^2 + \ldots + (k+1)^2 = \frac{(k+1)(k+2)(2k+3)}{6}, \text{ где } k \in N. \qquad (*)$$

Для доказательства рассмотрим левую часть (*).

$$1^2 + 2^2 + \ldots + (k+1)^2 = 1^2 + 2^2 + \ldots + k^2 + (k+1)^2 = (**).$$

Воспользуемся предположением индукции:

$$(**) = \frac{k(k+1)(2k+1)}{6} + (k+1)^2 = \frac{(k+1)(2k^2+7k+6)}{6} = \frac{(k+1)(k+2)(2k+3)}{6},$$

что и требовалось доказать.

Напишем программу, которая запрашивает n, выделяет память для хранения квадратов первых n натуральных чисел, вычисляет эти квадраты и записывает их в массив, после чего производит суммирование и сравнивает результат с теоретическим, полученным по формуле (9.1).

```
{ ================================================ }
{ Пример 9.8 }
{ Указатели и массивы }
Program GetMemEx;

{$APPTYPE CONSOLE}

type
   PByte = ^Word;
```

```
var
   n: Word;     { Количество чисел                                  }
   P: Pointer; { Указатель на буфер для хранения квадратов чисел }
   Pt: PWord;  { Указатель, используется при расчетах             }
   i: Integer; { Счетчик цикла                                    }
   S: Integer; { Сумма квадратов }
begin
   Write('Введите количество чисел:');
   ReadLn(n);
   { Выделяем память }
   GetMem(P, n * SizeOf(Word));
   { Вычисляем квадраты и записываем их в буфер                   }
   { Используем вспомогательный типизированный указатель Pt }
   { для перемещения по буферу.                                   }
   { Компенсируем отсутствие операций + для указателей            }
   Pt := P;
   for i := 1 to n do
   begin
      Pt^ := i * i;
      Pt := Ptr(Integer(Pt) + SizeOf(Word));
   end;
   { Считаем сумму }
   Pt := P;
   S := 0;
   for i := 1 to n do
   begin
      S := S + Pt^;
      Pt := Ptr(Integer(Pt) + SizeOf(Word));
   end;
   WriteLn('По формуле S = ', n*(n + 1)*(2* n + 1) div 6);
   WriteLn('В результате расчета S = ', S);
   { Освобождаем память }
   FreeMem(P);
   ReadLn;
end.
{ ================================================= }
```

Разумеется, результаты прямого суммирования и расчета по формуле совпадут. Обратите внимание на способ перемещения по буферу, состоящий в инициализации вспомогательного указателя значением `P`, преобразовании его к целому типу, прибавлении размера одного числа в байтах, преобразовании к типу-указателю при помощи функции `Ptr` и, наконец, разыменовании.

Отметим, что вовсе необязательно считать сумму квадратов чисел, используя буфер для их хранения. В данном случае без этого можно

обойтись, но чаще всего обрабатываемые данные бывают нужны неоднократно в ходе работы программы, что приводит к необходимости их хранения.

3.4.2. Работа с памятью в стиле New, Dispose. Теперь поговорим о второй группе подпрограмм, позволяющих выделять и освобождать память динамически. Этот способ ориентирован на работу с типизированными указателями и состоит в использовании процедур `New` и `Dispose`.

```
procedure New(var P: Pointer);
```

Процедура `New` предназначена для выделения памяти с использованием типизированных указателей.

```
procedure Dispose(var P: Pointer);
```

Процедура `Dispose` предназначена для освобождения памяти с использованием типизированных указателей.

Рассмотрим пример.

```
{ ================================================= }
{ Пример 9.9 }
{ New и Dispose }
Program NewEx;

{$APPTYPE CONSOLE}

type
   TPerson = record
      FIO: String[50];
      BirthdayY: Word;
      BirthdayM: Byte;
      BirthdayD: Byte;
      Phone: String[15];
   end;
   PPerson = ^TPerson;

var
   P: PPerson;
   F: String[50];
   Y: Word;
   M: Byte;
   D: Byte;
   Ph: String[15];
begin
   Write('Введите ФИО:');
   ReadLn(F);
```

```
   Write('Введите год рождения:');
   ReadLn(Y);
   Write('Введите месяц рождения:');
   ReadLn(M);
   Write('Введите день рождения:');
   ReadLn(D);
   { Выделяем память }
   New(P);
   { Сохраняем данные }
   P^.FIO := F;
   P^.BirthdayY := Y;
   P^.BirthdayM := M;
   P^.BirthdayD := D;
   P^.Phone := Ph;

   // Обработка

   { Освобождаем память }
   Dispose(P);
   ReadLn;
end.
{ ================================================= }
```

Как видите, все достаточно просто. Дополнительное удобство состоит в том, что более не нужно указывать размер выделяемой памяти, так как он известен по типу указателя.

4. Динамические структуры языка Object Pascal

4.1. Динамические массивы

Рассмотрим одну из часто используемых возможностей языка Object Pascal – *динамические массивы*. Разработчикам компилятора удалось создать замечательный по своей простоте и удобству механизм, что бывает совсем не часто. Всем нам хорошо известен обычный (статический) массив, многократно использованный в этой книге к настоящему моменту. В этой главе мы узнали, что можно устранить недостаток, связанный с необходимостью указания размера массива на этапе компиляции программы, при помощи реализации динамических массивов посредством бестиповых указателей, процедур `GetMem`, `FreeMem`, `ReallocMem` и «самодельных» операций перемещения

указателя по массиву (см. пример 9.8). Оказывается, Object Pascal реализует подобную структуру на уровне языка. Посмотрим, как ей можно воспользоваться.

4.1.1. Введение в динамические массивы. Для объявления динамического массива используется стандартная конструкция **array** без указания диапазона индексов.

```
type
  <Имя типа данных> = array of <Тип элемента>;
```

Объявим массив элементов типа Word.

```
type
  TWordArray = array of Word;
var
  M: TWordArray;
```

Интересно, что собой представляет переменная M – массив неизвестного размера? Все очень просто, на самом деле M – это просто указатель. Изначально, после объявления, массив M состоит из 0 элементов, соответственно, M по смыслу близка к нулевому указателю.

Размер массива устанавливается при помощи процедуры

```
procedure SetLength(var S; NewLength: Integer);
```

Это процедура осуществляет выделение необходимой памяти для хранения NewLength элементов типа, указанного при объявлении массива S.

Доступ к элементам динамического массива осуществляется обычным образом при помощи операции индексации «[ ]», с учетом того, что диапазон индексов 0..NewLength-1. Заметим, что компилятор сам заботится не только о выделении памяти, но и о запоминании параметров массива, в частности количества его элементов.

Вызов SetLength с параметром 0 осуществляет освобождение памяти, занимаемой элементами массива, вызов SetLength со значением, не равным нулю и отличным от текущего значения, приводит к перераспределению памяти с сохранением значений элементов.

Перепишем рассмотренный ранее пример с вычислением суммы квадратов с использованием динамических массивов.

```
{ ================================================ }
{ Пример 9.10 }
{ Указатели и массивы }
Program DynArrEx;
```

```
{$APPTYPE CONSOLE}

type
   TWordArray = array of Word;
var
   n: Word;              { Количество чисел }
   i: Integer;           { Счетчик цикла    }
   S: Integer;           { Сумма квадратов  }
   A: TWordArray;        { Массив           }
begin
   Write('Введите количество чисел:');
   ReadLn(n);
   { Выделяем память }
   SetLength(A, n);
   { Вычисляем квадраты и записываем их в массив }
   for i := 1 to n do
      A[i - 1] := i * i;
   { Считаем сумму }
   S := 0;
   for i := 1 to n do
      S := S + A[i - 1];
   WriteLn('По формуле S = ', n*(n + 1)* 2* n + 1) div 6);
   WriteLn('В результате расчета S = ', S);
   { Освобождаем память }
   SetLength(A, 0);
   ReadLn;
end.
{ ================================================= }
```

Отметим, что текст программы стал намного короче и понятнее. Также обращаем внимание на конструкцию `A[i - 1]` – напоминаем, что динамические массивы всегда индексируются с нуля. Для того чтобы застраховать себя от неприятностей с диапазонами индексов, часто пишут так:

```
for i := Low(A) to High(A) do
   ...;
```

Функции `Low` и `High` возвращают индексы первого и последнего элементов массива соответственно (первый всегда имеет индекс `0`), пользуясь тем, что компилятор хранит информацию о количестве элементов массива[1].

[1] Эстеты могут открыть справочную систему Delphi и обнаружить, что по смещению -4 от начала массива компилятор сохраняет его длину (4 байта), а по смещению -8 – количество ссылок, т.е. информацию о том, сколько раз

4.1.2. Присваивание и копирование. Сравнение. Известно следующее изречение, неплохо характеризующее то, что многие пользователи думают о компьютерах и программах: «*Компьютеры бесподобны: за несколько минут они могут совершить такую грандиозную ошибку, какую не в состоянии сделать множество людей за многие месяцы*» (М. Мичэм).

Надо понимать, что динамическое распределение памяти одновременно является и мощным инструментом программирования, и потенциальным источником труднонаходимых ошибок. Одно из таких «узких мест» – присваивание и копирование динамических массивов. Рассмотрим этот вопрос подробно.

Пусть A и B есть динамические массивы одинакового типа и размерности. Зададимся вопросом: что будет, если мы напишем в программе `A := B`? Это тем более интересно, если предварительно для A не выделялась память. Рассмотрим пример:

```
var
   A, B: array of Integer;
begin
   SetLength(B, 10);
   B[5] := 5;
   A := B;
   A[5] := 7;
   ...
end.
```

На самом деле, подобное присваивание разрешено, и для него не требуется выделения памяти под массив A. Однако присваивание в данном случае приводит к созданию второго имени, синонима для доступа к массиву B. Теперь он доступен и как A. Таким образом, в отличие от обычных массивов, в которых в результате присваивания между однотипными массивами происходило копирование данных, в динамических массивах такого не происходит. Значит, `A[5]:= 7` приводит к тому, что `B[5]` также равно 7.

Как осуществить копирование? Простейший способ выглядит так:

```
var
   A, B: array of Integer;
begin
   SetLength(B, 10);
   B[5] := 5;
```

используется данный массив. Так, например, после присваивания A := B количество ссылок равно 2. Заметим, что от версии к версии компилятора эта структура может меняться, т.е. на это не рекомендуется ориентироваться в программах.

```
    SetLength(A, 10); { Выделяем память! }
    for i := Low(A) to High(A) do
    A[i] := B[i];
    ...
end.
```

Существует и встроенная функция `Copy`, которая позволяет осуществить подобные действия.

```
var
   A, B: array of Integer;
begin
   SetLength(B, 10);
   B[5] := 5;
   A := Copy(B, 0, 10);
   ...
end.
```

В данном примере, независимо от того, была ли выделена память под массив `A`, создается новый массив из 10 элементов, в который копируются 10 элементов массива `B`, начиная с 0-го.

В заключение заметим, что применение к динамическим массивам одного типа операции сравнения приводит к тому, что сравнивается не содержимое массивов, а ссылки, т.е. проверяется, не есть ли это синонимы.

4.1.3. Многомерные динамические массивы. Динамические массивы могут быть многомерными. Рассмотрим ситуацию на примере двумерного массива.

Для его создания необходимо использовать конструкцию **`array of`** два раза. Создадим тип данных «матрица» для хранения таблицы целых чисел и объявим переменную этого типа:

```
type
   TMatrix = array of array of Integer;
var
   M: TMatrix;
```

Для установки размеров матрицы необходимо использовать `SetLength` с двумя параметрами – количеством строк и количеством столбцов. Так,

```
SetLength(M, 10, 20);
```

создает матрицу размером `10×20`. Доступ к элементам реализуется обычным образом, так, мы можем написать `M[5, 7]:= 9`.

Рассмотрим еще одну интересную возможность, связанную с многомерными динамическими массивами. Оказывается, мы можем написать так:

```
SetLength(M, 10);
```

Что будет в результате? Десять одномерных динамических массивов нулевой длины. Впоследствии мы можем указать для каждого из них свое количество элементов, получив непрямоугольную таблицу. Рассмотрим применение этого механизма на примере практической задачи.

4.1.4. Многомерные динамические массивы в задаче о поиске оптимального пути. Рассмотрим задачу.

В Тридевятом королевстве жил да был король. И решил он назначить первым министром самого мудрого из всех возможных кандидатов. Для этого король устроил следующее состязание. Лучшие строители королевства построили лабиринт, устроив его так: в начале лабиринта находилась одна-единственная дверь. Пройдя через нее, мудрец оказывался перед двумя дверьми, на следующем уровне дверей было уже три, затем четыре и т.д. По правилам состязаний кандидат на почетную должность в каждый момент времени мог идти только *через одну из двух ближайших дверей*. За проход через каждую дверь мудрец платил фиксированный штраф от 1 до 50 тугриков. Победителем считался мудрец, заплативший наименьший штраф. Схема лабиринта из 500 уровней с указанием штрафов была предоставлена кандидатам за день до соревнований.

Вооружившись компьютером, помогите мудрецу одержать победу.

Анализ задачи

Попробуем разобраться в постановке задачи.

Что нам дано?

1. Дана схема лабиринта, в котором на первом уровне 1 дверь, на каждом следующем уровне количество дверей увеличивается на единицу.
2. Дан штраф за проход через каждую из дверей.
3. Установлена схема перемещения, в соответствии с которой в каждый момент времени мы можем идти вперед только через одну из двух ближайших дверей.

Что нам надо найти?

Необходимо найти такой порядок прохождения через двери, чтобы заплатить наименьший штраф к моменту выхода из лабиринта (всего нужно пройти 500 уровней).

Некоторые дополнительные соображения:

1. По-видимому, 500 уровней – достаточно много, а времени на раздумья всего один день. Придется учитывать это при решении задачи.
2. Будем далее считать, что уровней у нас n и решать задачу в общем случае, понимая, что n достаточно большое.

Математическая модель

Предлагаем следующую математическую модель для данной задачи: будем представлять лабиринт в виде равностороннего треугольника из чисел, в котором в первой строке – 1 число, во второй строке – 2 числа, ..., в *n*-ой строке – *n* чисел. Число соответствует штрафу.

На рисунке 9.2 приведен пример того, как может выглядеть соответствующий треугольник, где круги – двери, числа – штрафы, стрелками отмечены разрешенные направления движения. Любая из дверей на последнем уровне – выход из лабиринта[1].

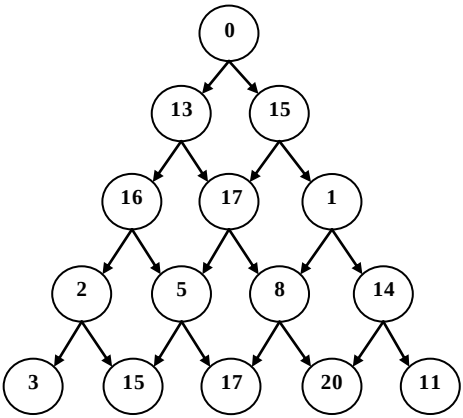


Рис. 9.2. Модель лабиринта (пример)

Попробуем разработать алгоритм для решения задачи.

Алгоритм 1. Полный перебор

Первое, что приходит в голову, – построить переборный алгоритм, который проанализирует все возможные варианты перемещения от начальной двери до последнего уровня, сосчитает сумму штрафа для каждого пути и найдет путь с минимальным штрафом.

Для того чтобы запрограммировать подобный алгоритм, необходимо прежде всего понять, как построить перебор путей. Нетрудно видеть, что

[1] В таком виде (начиная с треугольника из чисел) с точностью до замены минимизации максимизацией задача предлагалась на международной олимпиаде по информатике в 1994 г. в Швеции. Смотри, например, http://olympiads.win.tue.nl/ioi/ioi94/

для этого надо знать, что такое путь. Вернее, необходимо построить *модель пути*. Попробуем это сделать.

Заметим, что независимо от выбора двери в каждый момент времени длина всех путей одинакова и составляет *n*-1. Доказательство этого факта очевидно: за один шаг (под шагом будем понимать осуществление выбора из двух дверей) мы независимо от результатов выбора продвигаемся вперед (или вниз, если смотреть на рисунок 9.2) на один уровень. Таким образом, за 1 шаг мы попадаем на 2-й уровень, за 2 шага – на 3-й, а за *n*-1 шагов – на *n*-й.

Теперь разберемся с тем, как закодировать факт выбора. Каждый выбор двери представляет собой переход «влево вперед» или «вправо вперед». Будем обозначать 0 движение влево, а 1 – вправо, тогда в итоге получаем, что путь есть последовательность нулей и единиц длины *n*-1.

Формально путь $P = \{p_1, p_2, ..., p_{n-1}\}$, где $p_i \in \{0; 1\}$, по $\overline{i = 1, n-1}$.

Теперь мы можем перебрать все такие последовательности, для каждой из них сосчитать сумму элементов треугольника и найти путь, на котором сумма достигает минимума, решая поставленную задачу.

Попробуем оценить, сколько времени потребуется мудрецу на получение решения. Для этого оценим количество путей. Количество путей есть количество разных последовательностей длины *n*-1, элементы которой могут иметь 2 разных значения. Известно, что это количество равно 2^{n-1}.

Берем хороший инженерный калькулятор и вычисляем для $n = 500$

$$2^{499} = 1{,}636695303948070935006594848413 8 * 10^{150}.$$

Впечатляет? Похоже, мудрец не доживет до того времени, когда сгенерируются все возможные пути, не говоря уже о подсчете сумм.

Для тех, кого не убедило приведенное число астрономических размеров, представим текст программы для экспериментов. Некоторые пояснения к программе:

1. Программа не решает исходную задачу, она просто генерирует все возможные последовательности 0 и 1 длины *n*.

2. Идея алгоритма получения последовательностей – взятие {0, 0, …, 0} в качестве исходной последовательности и двоичное прибавление единицы для получения каждой следующей цепочки.

В результате, например, цепочки длины 3 будут получены в следующем порядке:

```
0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
```

```
1 1 0
1 1 1
```

3. Двоичное прибавление единицы осуществляется так: ведется анализ текущей последовательности справа налево. Если последний символ нуль, то он заменяется на единицу, после чего получена новая цепочка. Если последний символ единица, то он сам и все стоящие перед ним единицы заменяются нулями, а первый встретившийся нуль заменяется на единицу, что опять же дает новую цепочку. При этом устанавливается ограничение, чтобы алгоритм закончил работу, если последовательность уже состоит из одних единиц.

4. Программа содержит вывод на экран для того, чтобы можно было оценить ее работоспособность при малых *n*. Для проведения экспериментов для больших *n* вывод на экран нужно закомментировать.

5. Реализация алгоритма не претендует на оптимальность. Целью являлось сделать ее по возможности максимально понятной. Желающим ознакомиться с этим и другими сходными алгоритмами рекомендуем прекрасную книгу [21], в которой этому посвящен целый раздел. Однако даже если реализацию можно ускорить в 10 раз (что выглядит сомнительным), ситуация принципиально не изменится. Уже при $n = 50$ дождаться генерации всех цепочек будет проблематично. Учтите, что данный алгоритм лишь перебирает все пути, не вычисляя суммы.

Итак, вот обещанный текст программы:

```
{ ================================================== }
{ Пример 9.11 }
{ Полный перебор - получение всех путей }
Program Enumeration_of_possibilities;

{$APPTYPE CONSOLE}

var
   np: Word;
   a: array of Byte;

procedure Print;
var
   i: Word;
begin
   WriteLn;
   for i := Low(a) to High(a) do
      Write(a[i], ' ');
```

```
end;

procedure Enum(n: Word);
var
   i: Integer;
begin
   SetLength(a, n - 1);
   { Начальная последовательность }

   for i := Low(a) to High(a) do
     a[i] := 0;
   { Получена новая цепочка. Печатаем }
   Print;
   { Прибавление единицы }
   while True do
   begin
      i := High(a);
      if a[i] = 0 then
      begin
         a[i] := 1;
         { Получена новая цепочка. Печатаем }
         Print;
      end
      else begin
         { второе условие гарантирует то, }
         {  что алгоритм закончит работу  }
         while (a[i] = 1) and (i >= Low(a)) do
         begin
            a[i] := 0;
            i := i - 1;
         end;
         if i < Low(a) then
           break; // Уже получена последняя цепочка. Выход
         a[i] := 1;
         { Получена новая цепочка. Печатаем }
         Print;
      end;
   end;
   SetLength(a, 0);
end;

begin
   Write('Введите n: ');
   ReadLn(np);
```

```
   Enum(np);
   WriteLn('Все последовательности получены');
   ReadLn;
end.
{ ================================================ }
```

Алгоритм 2. Динамическое программирование (метод Беллмана)

Итак, перебор показал свою бесперспективность. Как помочь мудрецам? Подумаем над нашей задачей еще раз.

Рассмотрим двери A и B на рисунке 9.3. Нетрудно видеть, что при переборе мы вынуждены обсчитать выделенную пунктиром область 2 раза, один раз для двери A и один раз для двери B. Очевидно, подобные повторы встречаются почти для каждой двери. Это и есть тот ресурс, за счет которого мы должны попытаться ускорить алгоритм – исключить многократные расчеты одного и того же. Как это сделать?

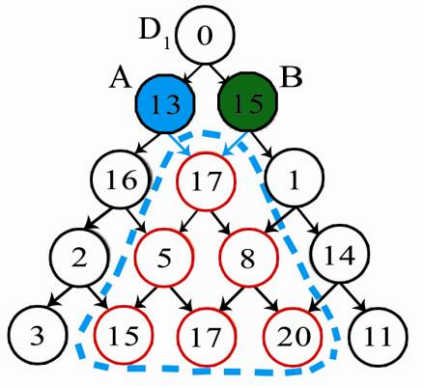


Рис. 9.3. Модель лабиринта. Повторные вычисления

Ответ на этот вопрос дает известный метод Беллмана (часто в литературе подход носит название «динамическое программирование» [11]) – любой участок оптимального пути должен быть оптимальным. Что это значит?

Это значит, что если последовательность дверей $\{D_1, \ldots, D_n\}$ – оптимальный (лучший, в нашей задаче с минимальной суммой штрафа) путь с уровня 1 (дверь D_1 фиксирована) до уровня n, то любая ее часть $\{D_i,\ldots,D_j\}$ – оптимальный путь из двери D_i к двери D_j. Действительно, ведь если это не так и из D_i в D_j есть лучший путь, то можно взять его и вставить в наш «оптимальный вариант пути», улучшив его, что

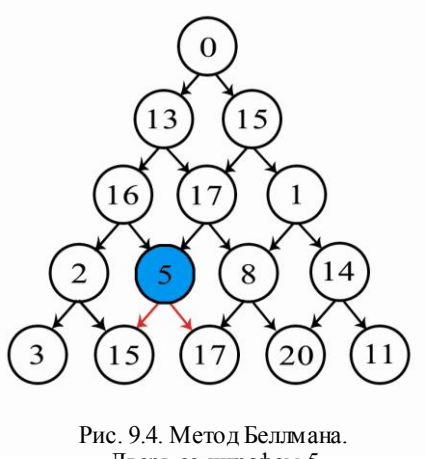


Рис. 9.4. Метод Беллмана. Дверь со штрафом 5

невозможно по предположению об оптимальности.

Применим эти соображения. Рассмотрим любую дверь предпоследнего уровня (пусть это дверь со штрафом 5 на рис. 9.4). Пройдя через эту дверь, можно выйти из лабиринта ровно двумя способами – через дверь со штрафом 15 и через дверь со штрафом 17. Очевидно, что в целях минимизации расходов нужно идти в дверь со штрафом 15, что даст в результате суммарный штраф от прохода двух уровней 5 + 15 = 20.

Это значит, что, попав в отмеченную на рисунке дверь, мы в лучшем случае выйдем из лабиринта, заплатив штраф, равный 20. Начав с предпоследнего ряда, мы можем последовательно дверь за дверью рассчитать минимальные штрафы указанным выше способом. В результате, добравшись до входной двери, мы получим минимальную сумму штрафа, а вспомнив, в какой из двух вершин достигался минимум, – оптимальный путь из этой вершины в конец. Таким образом мы построим оптимальный путь.

Приведем на рисунке 9.5 значения штрафов, рассчитанных «с конца» указанным способом.

Синим выделены стрелки, по которым нужно осуществлять переход от верхней вершины к нижним. Заметим, что оптимальный путь может получиться не один (см. две стрелки, выходящие из вершины с оптимальным штрафом 26).

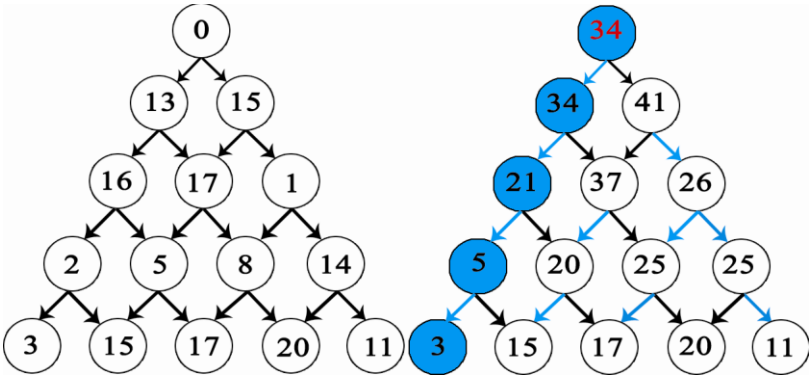


Рис. 9.5. Метод Беллмана. Вспомогательные расчеты

Оценим трудоемкость алгоритма. В соответствии с написанным выше нам необходимо будет рассчитать $1+2+...+n=\frac{n(n+1)}{2}$ значений, что намного лучше предыдущего варианта.

Проектирование

Рассмотрим вопрос проектирования программы, решающей поставленную задачу по второму алгоритму.

Первый вопрос – как представить данные. Язык программирования не содержит средств реализации треугольных структур данных. Будем представлять исходные данные (штрафы) в виде двумерного динамического массива, в первой строке которого 1 элемент, во 2-й строке – 2 элемента, в *n*-й строке – *n* элементов. Обозначим эту структуру A.

Дополнительно заведем точно такой же массив для вспомогательных расчетов по определению минимального штрафа. Обозначим его S.

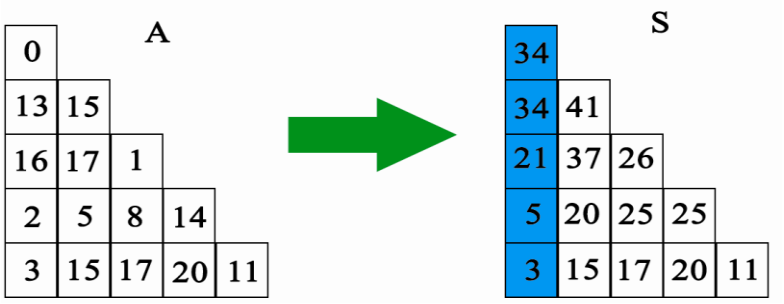


Рис. 9.6. Проектирование данных. Переход к двумерному массиву

Заметим, что рассматриваемые структуры существенно лучше обычного двумерного массива для этой задачи из-за большой (почти двукратной) экономии памяти.

Опишем математически алгоритм получения массива S, рассмотренный выше в словесной форме.

$$S_{n,j}=A_{n,j},\ \text{при}\ \ j=\overline{1,n}$$

$$S_{k,j}=A_{k,j}+\min\left\{S_{k+1,j},S_{k+1,j+1}\right\},\ \text{при}\ \ k=\overline{1,n-1},\ \ j=\overline{1,k}\ .$$

Реализация

Не забывая о том, что динамические массивы индексируются с нуля, а матрица S должна пересчитываться с конца, выполним следующую программную реализацию. Матрицу A будем генерировать случайным

образом, каждый из вас легко может организовать здесь ввод данных из файла.

```
{ ================================================== }
{ Пример 9.12 }
{ Метод Беллмана - получение оптимального пути }
Program Bellman;

{$APPTYPE CONSOLE}

uses Math;

const
   { Максимальный штраф }
   MaxPenalty = 10;
type
   { Штраф }
   TPenalty = 1..MaxPenalty;
   { Исходный массив штрафов }
   TPenaltyArr = array of array of TPenalty;
   { Вычисляемый массив минимальных штрафов }
   TSumArr = array of array of Cardinal;
var
   { Исходный массив штрафов }
   A: TPenaltyArr;
   { Вычисляемый массив минимальных штрафов }
   S: TSumArr;
   { Количество уровней }
   n: Word;

   { Инициализация массивов, ввод данных }
procedure InitArrays;
var
   i, j: Integer;
begin
   Randomize;
   Write('Введите n: ');
   ReadLn(n);
   { Установка размеров }
   SetLength(A, n);
   SetLength(S, n);
   for i := 0 to n - 1 do
   begin
      SetLength(A[i], i + 1);
      SetLength(S[i], i + 1);
   end;
   { Инициализация и ввод данных }
```

```
   for i := 0 to n - 1 do
      for j := 0 to i do
      begin
         S[i, j] := 0;
         A[i, j] := Random(MaxPenalty - 1) + 1;
      end;
   end;

{ Освобождение памяти }
procedure DoneArrays;
var
   i: Integer;
begin
   for i := 0 to n - 1 do
   begin
      SetLength(A[i], 0);
      SetLength(S[i], 0);
   end;
   SetLength(A, 0);
   SetLength(S, 0);
end;
{ Вывод результатов на экран }
procedure OutputResults;
var
   i, j: Integer;
begin
   WriteLn('Matrix A: ');
   for i := 0 to n - 1 do
   begin
      for j := 0 to i do
        Write(A[i, j], ' ');
      WriteLn;
   end;
   WriteLn('Matrix S: ');
   for i := 0 to n - 1 do
   begin
      for j := 0 to i do
         Write(S[i, j], ' ');
      WriteLn;
   end;
   WriteLn('Минимальный штраф ', S[0, 0]);
end;

{ Вычисление S }
procedure Calculate;
var
   i, j: Integer;
begin
   for j := 0 to n - 1 do
```

```
      S[n - 1, j] := A[n - 1, j];
   for i := n - 2 downto 0 do
      for j := 0 to i do
      S[i, j] := A[i, j] + Min(S[i + 1, j], S[i + 1, j + 1]);
end;

begin
   InitArrays;
   Calculate;
   OutputResults;
   DoneArrays;
   ReadLn;
end.
{ ================================================= }
```

Запускаем... Программа работает моментально даже при n = 1000. Теперь мы победим, даже если на подготовку к состязанию дадут 5 минут. Естественно, при больших *n* нужно отключить вывод массивов A и S на экран.

Вы были внимательны? Тогда вы должны были заметить, что мы нашли только сам штраф, но не нашли путь. Задача нахождения пути не приведет к большим изменениям в программе. Один из вариантов – завести еще один массив той же конфигурации, в котором запоминать, какую из дверей мы выбираем на каждом шаге. После заполнения S необходимо будет пройти от первой двери к последнему уровню, руководствуясь теми данными, которые будут в третьем массиве. Предлагаем вам произвести изменения в программе самостоятельно.

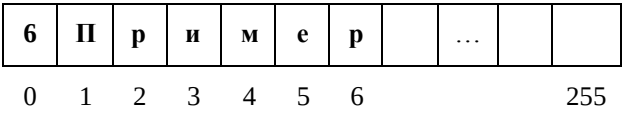
4.2. Обработка строковой информации. Короткие и длинные строки

В главе 4 мы сказали несколько слов о строках в языке Object Pascal. Теперь мы знаем достаточно для того, чтобы поговорить о строковых типах более подробно.

4.2.1. Строки в стиле Pascal 7.0. Как говорилось в главе 4, обычные строки в языке Pascal представляли собой массив символов длиной не более 255. Нулевой байт содержал фактическую длину строки, остальные 255 – символы. Сам строковый тип данных при этом назывался `String`, а элемент строки принадлежал типу `Char`. Работа со строкой велась как с обычным массивом, доступ к нулевому символу был несколько затруднен

(требовал преобразования типа), вместо этого существовал способ получения длины строки при помощи функции `Length`.

```
var
   S: String;
begin
   S := 'Пример';
end.
```



Разумеется, в ячейках содержатся не сами буквы, а их коды согласно таблице кодов ASCII.

При написании программ мы могли пользоваться следующими возможностями (индексация символов везде ведется с единицы):

```
function Length(S): Integer;
{ Возвращает длину строки }
procedure Insert(Source: String; var S: String;
   Index: Integer);
{ Вставляет подстроку Source в строку S с позиции Index }
procedure Delete(var S: String; Index, Count: Integer);
{ Удаляет из строки S Count символов с позиции Index }
function Copy(S; Index, Count: Integer): String;
{ Копирует из строки S Count символов с позиции Index }
function Pos(Substr: String; S: String): Integer;
{ Ищет первое вхождение слева подстроки Substr в строке S }
{ Возвращает номер символа, с которого начинается подстрока, }
{ или 0, если она не обнаружена}
```

В дополнение можно отметить следующее:

1. `S := ''` – создание пустой строки;
2. `c := s[i]` – взятие `i`-го символа строки;
3. `s[i] := c` – запись символа в `i`-ю позицию в строке;
4. `s := s1 + s2` – строка `s2` пристыковывается к строке `s1` и результат записывается в `s`.
5. `s1 := s2` – копирование содержимого строки `s2` в строку `s1`.
6. `s1 < s2` (и другие операции сравнения) – лексикографическое сравнение (по алфавиту).

Кроме того, мы могли ограничить количество символов в строке, написав, например, `String[20]` в качестве типа данных.

В принципе, перечисленного набора операций достаточно для того, чтобы выполнить любые действия над строковыми данными.

Рассмотрим задачу.

Дана строка текста. Удалить все цифровые символы.

Решение 1

Рассмотрим простой вариант реализации с использованием дополнительной памяти. Будем действовать так: создадим новую строку. Будем проверять все ее символы по очереди. Символы, не являющиеся цифровыми, будем переписывать в новую строку.

```
{ ================================================ }
{ Пример 9.13 }
{ Удаление цифр из строки с использованием }
{ дополнительной памяти                    }
Program SimpleDelete;
{$APPTYPE CONSOLE}


var
   i: Integer;
   s1, s2: String;
begin
   Write('Введите строку: ');
   ReadLn(s1);
   s2 := '';
   for i := 1 to Length(s1) do
      { Пользуемся упорядоченностью кодов символов }
      { в таблице кодов ASCII                      }
      if (s1[i] < '0') or (s1[i] > '9') then
         s2 := s2 + s1[i];
   WriteLn(s2);
end.
{ ================================================ }
```

Решение 2

Теперь попробуем устранить недостаток, связанный с использованием дополнительной памяти. Кажется, что самый простой способ состоит в том, чтобы удалять из строки `s1` каждый найденный цифровой символ. Вносим изменения в предыдущую реализацию:

```
{ ================================================ }
{ Удаление цифр из строки. Пример НЕ РАБОТАЕТ! }
Program MistakeInDelete;
```

```
{$APPTYPE CONSOLE}

var
   i: Integer;
   s: String;

begin
   Write('Введите строку: ');
   ReadLn(s);
   for i := 1 to Length(s) do
      { Пользуемся упорядоченностью кодов символов }
      { в таблице кодов ASCII                       }
      if (s[i] < '0') or (s[i] > '9') then
         Delete(s, i, 1);
   WriteLn(s);
end.
{ ================================================= }
```

Данная реализация представляет собой пример типичной ошибки, которую допускают многие начинающие программисты. Вспомним, как функционирует цикл **for**. Во-первых, значения пределов изменения переменной цикла вычисляются до начала работы цикла. А это значит, что удаление символов никак не влияет на количество итераций, оно подсчитано в начале. Во-вторых, после удаления символа строка сдвигается влево и на `i`-е место встает новый символ, также подлежащий проверке. А цикл **for** заботливо увеличит `i` на единицу, в результате чего мы этот символ вообще выпустим из рассмотрения.

Напрашивается вывод: в случаях, когда строка изменяется в размерах в ходе цикла, оператор **for** обычно не годится для решения задачи. Используем **while**.

```
{ ================================================= }
{ Пример 9.14 }
{ Удаление цифр из строки без использования }
{ дополнительной памяти                      }
Program RightDelete;

{$APPTYPE CONSOLE}

var
   i: Integer;
   s: String;
begin
   Write('Введите строку: ');
```

```
   ReadLn(s);
   i := 1;
   while (i <= Length(s)) do
      { Пользуемся упорядоченностью кодов символов }
      { в таблице кодов ASCII                       }
      if (s[i] < '0') or (s[i] > '9') then
         Delete(s, i, 1)
      else
         i := i + 1;
   WriteLn(s);
end.
{ ================================================= }
```

4.2.2. Строки в стиле Object Pascal. В языке Object Pascal ситуация кардинально изменилась. Рассмотренный только что тип данных был переименован в `ShortString` (отдельный символ по-прежнему имеет тип `Char`), а в дополнение к этому введен тип данных `AnsiString` (отдельный символ имеет тип `AnsiChar`), который представляет динамически распределяемые строки. Длина этих строк лимитирована ... 4 гигабайтами!

`ShortString`, `AnsiString` – какая-то путаница, может воскликнуть наш читатель. А что же делать со старыми программами, где был просто `String`? Ничего страшного, с большими шансами старые программы будут работать и дальше. Директива компилятора `{$H}` отвечает за то, какие именно строки используются в программе при упоминании типа `String`. По умолчанию установлено `{$H+}`, что означает `String = = AnsiString`. То есть по умолчанию все строки – длинные, а если надо иначе, можно использовать тип `ShortString`. Если вы хотите изменить это соглашение, установите директиву `{$H-}`, тогда `String = = ShortString`, а при необходимости использовать длинные строки вы можете написать `AnsiString`. Относительно символов все еще проще – `AnsiChar` и `Char` – это одно и то же.

Итак, будем в дальнейшем считать, что `String` – длинная строка. Прежде всего, хотелось бы отметить, что к таким строкам применимы все те операции и функции, которые были рассмотрены в предыдущем разделе. Дополнительно, в отличие от Pascal 7.0, система программирования Delphi содержит огромное количество подпрограмм обработки строк, содержащихся в модуле `StrUtils`. Чего там только нет: интеллектуальные поиски, удобные операции взятия подстрок, преобразование к другому регистру, перекодировка из одной кодировки символов в другую, выделение слов и многое другое. Так, например, функция `PosEx` является расширением функции `Pos`, позволяя проводить

поиск не с начала строки, а с любого символа. Другой модуль, `SysUtils` содержит функции форматирования строк, основная из которых, `Format`, позволяет производить подстановку параметров разных типов в строку, управлять форматом вывода числа и т.д. Обязательно познакомьтесь с содержимым этих модулей в справочной системе (формат книги, к сожалению, не позволяет вместить еще и эту информацию).

По своей организации длинные строки в существенной степени похожи на изученные нами динамические массивы. Но есть и серьезные отличия. Рассмотрим их.

Итак, длинные строки объявляются так же как обычные. Длина строки может быть установлена при помощи подпрограммы `SetLength`. Кроме того, строка автоматически перераспределяется при попытке присвоить ей значение (у динамических массивов это не так). Заметим, что, так же как и для массивов, язык Object Pascal использует механизм подсчета ссылок, чтобы удалить строку из памяти, когда она более никому не нужна. Фундаментальное отличие состоит в использовании для строк так называемой «copy-on-write» техники. Для понимания этого вопроса рассмотрим пример:

```
var
  A, B: String;
begin
  A := 'Привет!';
  B := A;
  ...
end.
```

В данном примере после присваивания `B := A` строки `A` и `B` указывают на одну и ту же структуру в оперативной памяти. При этом механизм подсчета ссылок запоминает, что на участок памяти ссылаются сразу 2 объекта.

Но! При попытке изменить содержимое строки `B` компилятор создаст в памяти новую строку и мы получим 2 разные строки.

```
var
  A, B: String;
begin
  A := 'Привет!';
  B := A;
  B[3] := 'И';
  ...
end.
```

В результате `A = 'Привет!'`, но `B = 'ПрИвет!'`. Заметьте существенную разницу с динамическими массивами.

4.2.3. И еще о строках (осталось за кадром). Мы считаем, что изложенной информации более чем достаточно для написания программ, работающих со строками. Однако язык Object Pascal содержит еще два строковых типа – `WideString` и `PChar`.

Тип `WideString` рассчитан на то, что символы могут принадлежать не таблице кодов ASCII (однобайтовый код), а таблице кодов Unicode (двухбайтовый код). Несмотря на то, что кодировка Unicode активно продвигается в последние годы, подавляющее большинство программ в Европе, Австралии и Америке по-прежнему работают с однобайтовыми символами. При необходимости с типом `WideString` можно ознакомиться в документации или справочной системе, его устройство почти не отличается от уже изученного материала.

Тип `PChar` предназначен для написания программных систем, которые работают с функциями Windows API или другими программами/библиотеками, написанными на C/C++, где строка представляет собой последовательность символов, заканчивающуюся нулем. Потребность в работе с типом `PChar` возникает нечасто, при необходимости можно ознакомиться с данным материалом в справочной системе.

5. Выводы

В главе 9 мы познакомились с одним из наиболее сложных вопросов практического программирования – динамическим распределением памяти. Проникнув в дебри адресного пространства программы, мы бодро прошагали по всем его сегментам, забрались в кучу, изучили, как работать с указателями – специальным средством, позволяющим нам общаться «на ты» с этим непростым механизмом. При изучении материала мы рассмотрели большое количество примеров, иллюстрирующих как надо и как не надо работать с адресами и динамической памятью. Наряду с этим мы изучили встроенные в язык Object Pascal динамические массивы и длинные строки, два необычайно полезных инструмента для минимизации усилий по написанию мощных программ, способных решать сложные задачи.

Теперь мы по праву можем сказать, что умеем многое. Многое, но не все, ведь нет предела совершенству. В следующей главе мы изучим последнюю тему, затрагиваемую в этой книге, последнюю по порядку, но отнюдь не по ее важности, – объектно-ориентированное программирование.

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.