МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

Юго-Западный государственный университет

(ЮЗГУ)

Кафедра вычислительной техники

УТВЕРЖДАЮ
Проректор по учебной работе
О.Г. Локтионова
«__» _____ 2016 г.

ДИНАМИЧЕСКИ РАСПРЕДЕЛЯЕМАЯ ПАМЯТЬ

Методические указания по выполнению самостоятельной работы

для студентов направления подготовки 09.03.01

Курск 2016

УДК 621.3
Составитель: Э.И. Ватутин

Рецензент
Кандидат технических наук, доцент *В.С. Панищев*

Динамически распределяемая память: методические указания по выполнению самостоятельной работы по дисциплине «Программирование» / Юго-Зап. гос. ун-т; сост.: Э.И. Ватутин; Курск, 2016. 31 с.: ил. 2.

Методические рекомендации содержат сведения по проектированию и разработке оконных приложений для решения поставленных переборных задач на современных языках программирования высокого уровня.

Предназначены для студентов направления подготовки 09.03.01 «Информатика и вычислительная техника».

Текст печатается в авторской редакции

Подписано в печать _____. Формат 60x84 1/16.
Усл. печ. л. Уч. – изд.л. Тираж 30 экз. Заказ . Бесплатно.
Юго-Западный государственный университет
305040, Курск, ул. 50 лет Октября, 94.

ВВЕДЕНИЕ

Большинство использованных до этого переменных являлись статическими: память под них выделялась в сегменте данных в момент запуска программы на исполнение, а их размер был известен на этапе компиляции. Исключение составляют лишь динамические массивы, поддерживаемые компилятором, и длинные строки, однако при работе с ними программист непосредственно не занимается работой с динамической памятью: всю черновую работу осуществляет компилятор. Данная особенность является неоспоримым преимуществом языка Delphi и способствует разработке более стабильно работающих программ. Однако иногда возникает необходимость в ручном управлении памятью и размещении в ней данных, чей размер может меняться во время работы программы. Информация по-прежнему хранится в оперативной памяти, однако данная память логически устроена по-другому и называется динамической. При работе с ней возможен ряд трудноуловимых ошибок, поэтому использовать ее следует в случае крайней необходимости. В некоторых случаях возможен вариант с расположением с памяти статического буфера (например, в виде одномерного массива), который используется лишь частично. Если размер подобного буфера достаточно большой, лучше использовать динамическую память. Часто динамическая память применятся как один из способов организации работы с такими структурами данных как деревья, списки, очереди или кольца.

Типизированные и нетипизированные указатели. Операция получения адреса

Работа с динамической памятью реализуется через указатели – специальный тип данных, который хранит адрес другой переменной в памяти (в общем случае – адрес произвольной ячейки оперативной памяти). При работе в операционной системе Windows в 32-разрядном режиме программе доступно адресное пространство размером почти 2 ГБ, однако обращения к произвольным ячейкам в его пределах недопустимы и могут привести к ошибке нарушения контроля доступа (англ. Access Violation). Обращаться можно лишь к областям памяти, соответствующим статическим переменным, либо участкам памяти, выделенным динамически. Каждый процесс в Windows работает в своем адресном пространстве, что означает невозможность напрямую обратиться к памяти другой программы. Такой подход реализуется аппаратно с использованием защищенного режима работы процессора и обеспечивает стабильность работы программ и операционной системы: одна сбойная программа не может нарушить работоспособность других (однако это могут сделать драйверы, работающие на нулевом кольце защиты). В адресное пространство процесса проецируются соответствующие сегменты данных динамически

подключаемых библиотек DLL, к данным которых при необходимости можно обращаться по указателю.

В Delphi существует понятие типизированных и нетипизированных указателей. Нетипизированный указатель задается с использованием типа `Pointer` и представляет собой просто адрес ячейки памяти. При работе с типизированными указателями кроме адреса компилятор имеет в своем распоряжении информацию о типе хранимого значения. Подобные указатели описываются с использованием символа «^», указываемого перед идентификатором типа.

```
var
  Ptr1: Pointer; // Нетипизированный указатель
  Ptr2: ^Тип;    // Типизированный указатель
```

При объявлении типизированных указателей разрешается использовать типы данных, которые еще не объявлены в программе. Например:

```
type
  PArray = ^TArray;
  TArray = array [1..10] of Integer;
```

Значение указателю может быть присвоено путем использования операции взятия адреса «@». Указатель, который указывает в никуда, обозначается ключевым словом `nil`. Обращение по такому указателю гарантированно приводит к ошибке нарушения доступа с возбуждением соответствующей исключительной ситуации.

Для обращения к области памяти, на которую указывает переменная типа указатель, используется операция разыменования указателя, обозначаемая путем указания символа «^» после имени переменной в коде программы. Рассмотрим несколько примеров.

```
var
  A: Integer = 10;
  Ptr1: Pointer;
  Ptr2: ^Integer;

begin
  Writeln('A = ', A);     // Обращение к переменной A
напрямую по имени. A=10

  Ptr1 := @A;             // Получение адреса переменной
A, сохранение в нетипизированном указателе
  Integer(Ptr1^) := 20;  // Обращение к переменной A с
использованием нетипизированного указателя (требуется
указание типа)
```

```
  Writeln('A = ', A);     // A=20

  Ptr2 := @A;             // Получение адреса переменной
A, сохранение в типизированном указателе
  Ptr2^ := 30;            // Обращение к переменной A с
использованием типизированного указателя (тип
определяется компилятором автоматически)
  Writeln('A = ', A);     // A=30

  Readln;
end.
```

Кроме иллюстрации простейших приемов работы с указателями пример показывает, какой потенциальной опасностью обладают указатели. Запись значения по неправильно инициализированному указателю способна изменить значение переменной даже в том случае, когда в коде с ней в явном виде никаких действий не производится (ее имя не фигурирует). Отыскание подобных ошибок при работе с т.н. «блуждающими указателями» представляет собой нетривиальную задачу.

Над указателями не разрешено выполнение каких-либо операций кроме разыменования, копирования и сравнения, однако возможна организация адресной арифметики путем приведения указателя к типу `Integer` и выполнения необходимых арифметических действий. Например:

```
Integer(Ptr) := Integer(Ptr) + 4;
```

Управление выделением и освобождением динамической памяти

Для управления динамической памятью в Delphi существует две пары подпрограмм. Первая пара из них представлена подпрограммами `New` и `Dispose`. Процедура `New` предназначена для выделения динамической памяти под переменную целиком и принимает в качества параметра типизированный указатель. После работы данная динамическая память должна быть освобождена, для чего используется процедура `Dispose`, также принимающая в качестве параметра значение указателя. Пример программы с их использованием приведен ниже.

```
var
  P1: ^Integer;

begin
  New(P1);        // Выделение динамической памяти

  P1^ := 2;       // Работа с элементом
```

```
  Writeln(P1^);  // 2

  Dispose(P1);   // Освобождение памяти

  Readln;
end.
```

Замечание. Довольно грубыми, но, к сожалению, часто встречающимися ошибками являются отсутствие выделение памяти, отсутствие освобождения памяти или повторение указанных действий. При работе с динамической памятью нужно быть предельно внимательным и стараться не допускать указанных ошибок.

Если в программе требуется выделение блока динамической памяти с заранее известным размером, применяется процедура `GetMem`, принимающая в качестве параметров указатель и требуемый объем памяти в байтах. Освобождение памяти, выделенной подобным образом, осуществляется с использованием процедуры `FreeMem`.

При работе с динамической памятью освобождение памяти всегда производится вручную, о чем не следует забывать. Если, например, выделение динамической памяти производится в подпрограмме и адрес хранится в локальной переменной-указателе, то при выходе из подпрограммы без освобождения выделенной памяти данный адрес будет потерян и память не сможет быть корректно освобождена вплоть до завершения работы программы. Подобные ситуации называется утечками памяти (англ. memory leaks), они должны быть устранены в ходе отладки. Память, выделенная с использованием процедуры `New` должна быть освобождена процедурой `Dispose`, а выделенная с использованием `GetMem` – процедурой `FreeMem`, причем нарушение данного правила недопустимо.

Пример использования подпрограмм `GetMem` и `FreeMem` приведен ниже.

```
type
  PByteArr = ^TByteArr;
  TByteArr = array [1..MaxInt] of Byte;

var
  F: File;
  Buf: PByteArr;
  S, Size, I: Integer;

begin
  AssignFile(F, 'File.txt');
  Reset(F, 1);
```

```
  Size := FileSize(F);
  GetMem(Buf, Size);

  BlockRead(F, Buf^, Size);
  CloseFile(F);

  S := 0;
  for I := 1 to Size do
    S := S xor Buf[I];

  FreeMem(Buf);

  Writeln(S);
  Readln;
end.
```

В данном примере находится контрольная сумма всех байт файла с использованием функции XOR. Т.к. размер файла заранее неизвестен, память выделяется динамически. Альтернативным подходом к решению задачи является использованием динамических массивов, поддерживаемых компилятором, или чтение файла в буфер фиксированного размера порциями, что не потребует ручного управления динамической памятью.

С использованием данной пары функций можно осуществить выравнивание блока памяти на произвольный адрес (обычно для этого он выбирается кратным степени двойки). Приведенный ниже код демонстрирует подобный подход.

```
var
  P, AlignedP: Pointer;
  Size: Integer;

begin
  Size := 1000;

  GetMem(P, Size + 16);  // Выделение памяти с запасом, адрес не выровнен

  Integer(AlignedP) := (Integer(P) + 16) and $FFFFFFF0; // Выравнивание на 16 путем сброса младших бит адреса

  // Работа с выровненным указателем AlignedP

  FreeMem(P);  // Освобождение памяти производится по исходному указателю
```

```
  Readln;
end.
```

Замечание. Для обработки запросов на выделение динамической памяти существует специальная область данных, называемая кучей (англ. heap). В программе на Delphi существует локальная куча, управляемая при помощи менеджера памяти (англ. memory manager), объявленного в модуле `System` и представляющего собой запись типа `TMemoryManager` со ссылками на три подпрограммы: для выделение (`GetMem`), освобождения (`FreeMem`) и переразмещения (`ReallocMem`) фрагмента динамической памяти. При необходимости менеджер памяти может быть подменен своим с использованием подпрограмм `GetMemoryManager` и `SetMemoryManager` (например, путем подобной подмены можно следить за динамикой выделения и освобождения динамической памяти, определять максимальный размер или частоту запросов к менеджеру памяти, контролировать утечки памяти). В операционной системе Windows существует отдельный ряд подпрограмм для работы с глобальной кучей (например, `VirtualAlloc` и `VirtualFree`). С их помощью, например, можно выделять области памяти, которые операционной системе запрещено сбрасывать в файл подкачки.

Управление динамическими структурами данных

Без использования ручного управления динамической памятью сложно обойтись при реализации работы с динамическими структурами данных, которые мы рассмотрим на примере односвязного линейного списка (рис. 15.1).

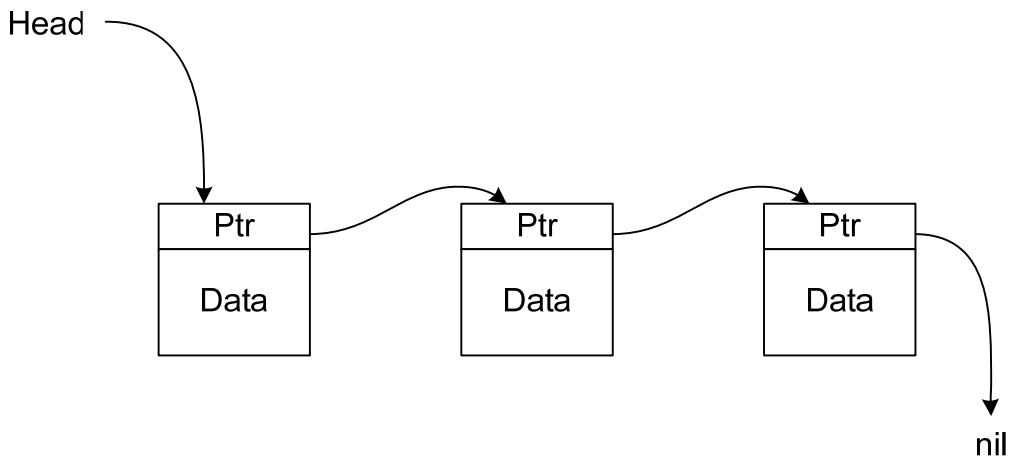

Рис. 1. Односвязный список

Элемент данного списка состоит из записи с двумя полями: указателя на следующий элемент `Ptr` и поля, хранящего данные элемента списка

Data. Все элементы располагаются в динамической памяти и ссылаются друг на друга по цепочке. Указатель на первый элемент списка хранится в переменной Head и обычно называется указателем на *голову списка* (при реализации двусвязных списков иногда вводится также понятие указателя последний элемент списка, называемой *хвостом*). Последний элемент списка хранить указатель в никуда, равный значению nil. Данная структура данных обладает интересной особенностью: при добавлении или удалении элементов списка с сохранением порядка остальных не требуется осуществлять сдвигов других элементов (в отличие, например, от реализации той же операции применительно к массивам), достаточно лишь перестроить соответствующие значения указателей. Оборотной стороной медали является большее время обработки данных и невозможность сразу узнать, где в памяти находится *i*-й элемент, для чего приходится организовывать линейный обход элементов.

Замечание. В Delphi существует стандартный класс TList, который внешне похож на динамической список, однако на самом деле представляет собой обертку над динамическим массивом.

Пример программы, демонстрирующей основные действия со списком, приведен ниже.

```
uses
  SysUtils;

type
  PListItem = ^TListItem;
  TListItem = record
    NextPtr: PListItem;
    Data: Integer;
  end;

var
  Head: PListItem = nil;

{ Добавление элемента в начало списка }
procedure AddItem(Value: Integer);
var
  Item: PListItem;
begin
  New(Item);

  with Item^ do begin
    NextPtr := Head;
    Data := Value;
  end;
```

```
  Head := Item;
end;


{ Вывод списка на экран }
procedure PrintList();
var
  Curr: PListItem;
  I: Integer;
begin
  Writeln('Head = $', IntToHex(Integer(Head), 8));

  Curr := Head;
  I := 0;

  while Curr <> nil do begin
    Inc(I);
    Writeln(I, ': Addr = $', IntToHex(Integer(Curr),
8), ' NextPtr = $',
        IntToHex(Integer(Curr^.NextPtr), 8), ' Data =
', Curr^.Data);
    Curr := Curr^.NextPtr;
  end;

  if I = 0 then
    Writeln('List is empty');

  Writeln;
end;


{ Удаление первого совпавшего элемента }
procedure DeleteFirst(Value: Integer);
var
  Curr, Prev, Next: PListItem;
begin
  Curr := Head;
  Prev := nil;

  while Curr <> nil do begin
    if Curr^.Data = Value then begin
      Next := Curr^.NextPtr;

      if Prev <> nil then
```

```
        Prev^.NextPtr := Next;

      if Curr = Head then
        Head := Next;

      Dispose(Curr);

      exit;
    end;

    Prev := Curr;
    Curr := Curr^.NextPtr;
  end;
end;


{ Подсчет суммы элементов списка }
function GetSum(): Integer;
var
  Curr: PListItem;
begin
  Result := 0;
  Curr := Head;
  while Curr <> nil do begin
    Result := Result + Curr^.Data;
    Curr := Curr^.NextPtr;
  end;
end;


begin
  AddItem(10);
  AddItem(20);
  AddItem(-10);
  PrintList();

  DeleteFirst(-10);
  PrintList();

  Writeln('Sum = ', GetSum());

  Readln;
end.
```

Указатель на следующий элемент для последнего элемента списка можно настроить на голову списка с получением структуры данных, называемой односвязным кольцом. При необходимости в состав списка или кольца могут быть добавлены указатели, обеспечивающие перемещение в обратную сторону – в таком случае соответствующая структура данных будет называться двухсвязной. Также при необходимости с использованием схожих подходов могут быть организованы и другие динамические структуры (матрицы, торы, гиперкубы и пр.).

Библиографический список

1. Емельянов С.Г., Ватутин Э.И., Панищев В.С., Титов В.С. Процедурно-модульное программирование на Delphi: учебное пособие. М.: Аргамак-Медиа, 2014. 352 с.
2. Зотов И.В., Ватутин Э.И., Борзов Д.Б. Процедурно-ориентированное программирование на C++: учебное пособие. Курск: КурскГТУ, 2008. 211 с.