ПРОЦЕДУРЫ И ФУНКЦИИ. ПЕРЕДАЧА ПАРАМЕТРОВ

Содержание

1. Цель работы

1. Получение навыков разработки приложений в среде Delphi.
2. Изучение способов обмена информацией между подпрограммами с помощью параметров вызова.

2. Теоретическая часть

2.1. Подпрограммы. Процедуры и функции

Часто некоторую последовательность инструкций требуется повторить в нескольких местах программы. Чтобы программисту не приходилось тратить время и усилия на копирование этих инструкций, в большинстве языков программирования предусматриваются средства для организации подпрограмм. Таким образом, программист получает возможность присвоить последовательности инструкций произвольное имя и использовать это имя в качестве сокращенной записи в тех местах, где встречается соответствующая последовательность инструкций.

Подпрограмма – некоторая последовательность инструкций, которая может повторяться в нескольких местах программы.

Процедурой называется особым образом оформленный фрагмент программы, имеющий собственное имя (идентификатор), который выполняет некоторую четко определенную операцию над данными, определяемыми параметрами.

Упоминание имени процедуры в тексте программы приводит к ее активизации и называется *вызовом процедуры*. Вызов может быть осуществлен из любой точки программы. При каждом таком вызове могут пересылаться различные параметры. Сразу после активизации процедуры начинают выполняться входящие в нее операторы, после выполнения последнего из них управление возвращается обратно в основную программу, и выполняются операторы, стоящие непосредственно за оператором вызова процедуры.

Описание процедуры состоит из заголовка и тела. Заголовок процедуры имеет вид:

```
procedure <имя> [(<сп.ф.п.>)];
```

Здесь `<имя>` – имя процедуры (правильный идентификатор); `<сп.ф.п.>` – список формальных параметров.

Список формальных параметров необязателен и может отсутствовать. Если же он есть, то в нем должны быть перечислены имена формальных параметров и их типы, например

```
procedure MyProc (a: Real; b: Integer; c: Char);
```

Функция отличается от процедуры тем, что результат ее работы возвращается в виде значения этой функции, и, следовательно, идентификатор функции может использоваться наряду с другими операндами в выражениях.

Описание функции состоит из заголовка и тела.

Заголовок функции имеет следующий вид:

```
function <имя> [(<сп.ф.п.>)]: <тип>;
```

Здесь <тип> – тип возвращаемого функцией результата.

Список формальных параметров здесь также необязателен. Заголовок функции может иметь следующий вид:

```
function MyFunc (a, b: Real): Real;
```

Операторы тела подпрограммы рассматривают список формальных параметров как своеобразное расширение раздела описаний: все переменные из этого списка могут использоваться в любых выражениях внутри подпрограммы. Таким способом осуществляется настройка алгоритма подпрограммы на конкретную задачу

Объявление и работа с процедурами и функциями отличаются в следующем:

1. в заголовке функции помимо описания формальных параметров обязательно указывается тип возвращаемого ею результата;
2. для возврата функцией значения в точку вызова среди ее операторов должен быть хотя бы один, в котором имени функции или переменной Result присваивается значение результата;
3. вызов процедуры выполняется отдельным оператором;
4. вызов функции может выполняться там, где допускается ставить выражение, в частности, в правой части оператора присваивания.

2.2. Передача параметров

Для обмена информацией между основной программой и процедурой используется один или несколько параметров вызова. Они перечисляются за именем процедуры и вместе с ним образуют оператор ее вызова.

Механизм замены формальных параметров на фактические позволяет нужным образом настроить алгоритм, реализованный в подпрограмме. Компилятор обычно следит за тем, чтобы количество и тип формальных параметров строго соответствовали количеству и типам фактических параметров в момент обращения к подпрограмме. Смысл используемых фактических параметров зависит от того, в каком порядке они перечислены при вызове подпрограммы. Поэтому программист должен сам следить за правильным порядком перечисления фактических параметров при обращении к подпрограмме.

Формальные параметры подпрограммы могут быть трех видов:

1. Параметры-значения;
2. Параметры-переменные;
3. Параметры-константы.

Например,

```
procedure MyProc(A: Real; var B: Real; const C: String);
```

Здесь *A* – параметр- значение, *B* – параметр-переменная, *C* – параметр-константа.

Определение формального параметра тем или иным способом существенно только для вызывающей программы. Если формальный параметр объявлен как параметр-значение или параметр-константа, то при вызове ему может соответствовать произвольное выражение. Если формальный параметр объявлен как параметр-переменная, то при вызове подпрограммы ему должен соответствовать фактический параметр в виде переменной нужного типа.

Контроль за неукоснительным соблюдением этого правила осуществляется компилятором. Если бы для вызова процедуры `MyProc` была бы написана строка

```
MyProc(X, -Y);
```

и для нее был бы использован такой заголовок

```
procedure MyProc (A: Real; var B: Real); ,
```

то при обращении к процедуре компилятор указал бы на несоответствие типа фактических и формальных параметров. Параметр обращения – *Y* есть выражение, в то время как соответствующий ему формальный параметр *B* описан как параметр-переменная.

Чтобы понять, в каких случаях использовать тот или иной тип параметров, нужно знать, как осуществляется замена формальных параметров на фактические в момент обращения к подпрограмме.

Если параметр определен как параметр-значение, то перед вызовом подпрограммы это значение вычисляется, полученный результат копируется во временную память и передается подпрограмме. Даже если в качестве фактического параметра указано простейшее выражение в виде переменной или константы, то все равно подпрограмме будет передана лишь копия переменной (константы). Любые возможные изменения в подпрограмме параметра-значения никак не воспринимаются вызывающей программой, так как в этом случае изменяется копия фактического параметра.

Если параметр определен как параметр-переменная, то при вызове подпрограммы передается сама переменная, а не ее копия (фактически в этом случае подпрограмме передается *адрес* переменной). Изменение параметра-переменной приводит к изменению самого фактического параметра в вызывающей программе.

В случае параметра-константы в подпрограмму также передается адрес области памяти, в которой располагается переменная или вычисленное значение. Однако компилятор блокирует любые присваивания параметру-константе нового значения в теле подпрограммы.

Параметры-переменные используются как средство связи алгоритма, реализованного в подпрограмме, с внешним миром. С помощью этих параметров подпрограмма может передавать результаты своей работы вызывающей программе. Разумеется, в распоряжении программиста всегда есть и другой способ передачи результатов – через глобальные переменные. Однако злоупотребление глобальными связями делает программу, как правило, запутанной, трудной в понимании и сложной в отладке. В соответствии с требованиями хорошего стиля программирования рекомендуется там, где это возможно, использовать передачу результатов через фактические параметры-переменные.

С другой стороны, описание всех формальных параметров как параметров-переменных нежелательно по двум причинам. Во-первых, это исключает возможность вызова подпрограммы с фактическими параметрами в виде выражений, что делает программу менее компактной. Во-вторых, в подпрограмме возможно случайное использование формального параметра, например, для временного хранения промежуточного результата, т. е. всегда существует опасность непреднамеренно испортить фактическую переменную.

Поэтому параметрами-переменными следует объявлять только те, через которые подпрограмма в действительности передает результаты вызывающей программе. Чем меньше параметров объявлено параметрами-переменными и чем меньше в подпрограмме используется глобальных переменных, тем меньше опасность получения непредусмотренных программистом побочных эффектов, связанных с вызовом подпрограммы, тем проще программа в понимании и отладке.

По той же причине не рекомендуется использовать параметры-переменные в заголовке функции. Если результатом работы функции не может быть единственное значение, то ло-

гичнее использовать процедуру или нужным образом декомпозировать алгоритм на несколько подпрограмм.

Существует еще одно обстоятельство, которое следует учитывать при выборе вида формальных параметров. Как уже говорилось, при объявлении параметра-значения осуществляется копирование фактического параметра во временную память. Если этим параметром будет массив большой размерности, то существенные затраты времени и памяти на копирование при многократных обращениях к подпрограмме можно минимизировать, объявив этот параметр параметром-константой. Параметр-константа не копируется во временную область памяти, что сокращает затраты времени на вызов подпрограммы, однако любые его изменения в теле подпрограммы невозможны – за этим строго следит компилятор.

Нетипизированные параметры.

Одним из свойств языка *Object Pascal* является возможность использования *нетипизированных параметров.* Параметр считается нетипизированным, если тип формального параметра-переменной в заголовке подпрограммы не указан, при этом соответствующий ему фактический параметр может быть переменной любого типа. Нетипизированными могут быть только параметры-переменные:

```
procedure MyProc(var aParametr);
```

Нетипизированные параметры обычно используются в случае, когда тип данных несущественен. Такие ситуации чаще всего возникают при разного рода копированиях одной области памяти в другую, например, с помощью процедур *BlockRead, BlockWrite, Move-Memory* и т. п.

Параметры-массивы.

Объявление переменных в списке формальных параметров подпрограммы на первый взгляд ничем не отличается от объявления их в разделе описания переменных. В обоих случаях, действительно много общего, но есть одно существенное различие: типом любого параметра в списке формальных параметров может быть только стандартный или ранее объявленный тип. Поэтому нельзя, например, объявить следующую процедуру:

```
procedure S(a: array (1..10] of real);  ,
```

так как в списке формальных параметров фактически объявляется тип-диапазон, указывающий границы индексов массива.

Если необходимо передать в подпрограмму массив, то следует первоначально описать его тип. Например:

```
type
    aType = array [1..10] of Real;
procedure S(var a: aType);
```

Поскольку короткая строка является фактически своеобразным массивом, ее передача в подпрограмму осуществляется аналогичным образом:

```
type
    InType = String [15];
    OuType = String [30];
function St(S: InType): OuType;
```

Компилятор *Object Pascal* поддерживает и открытые массивы, при использовании которых легко решается проблема передачи подпрограмме одномерных массивов переменной длины.

Открытый массив представляет собой формальный параметр подпрограммы, описывающий базовый тип элементов массива, но не определяющий его размерности и границы:

```
procedure MyProc(OpenArray: array of Integer);
```

Внутри подпрограммы такой параметр трактуется как одномерный массив с нулевой нижней границей. Верхняя граница открытого массива возвращается стандартной функцией `High`. Используя `0` как минимальный индекс и значение, возвращаемое функцией `High`, как максимальный индекс, подпрограмма может обрабатывать одномерные массивы произвольной длины.

Процедурные типы

Основное назначение процедурных типов – дать программисту гибкие средства передачи функций и процедур в качестве фактических параметров обращения к другим процедурам и функциям.

Для объявления процедурного типа используется заголовок процедуры (функции), в котором опускается ее имя, например:

```
type
    Proc1 = procedure (a, b, c: Real; var d: Real);
    Proc2 = procedure (var a, b);
    Proc3 = procedure;
    Func1 = function: String;
    Func2 = function (var s: String): Real;
```

Как видно из приведенных примеров, существует два процедурных типа: тип-процедура и тип-функция.

В программе могут быть объявлены переменные процедурных типов, например,так:

```
 var
pl     : Proc1;
f1, f2 : Func2;
ap     : array [1..N] of Proc1;
```

Переменным процедурных типов допускается присваивать в качестве значений имена соответствующих подпрограмм. После такого присваивания имя переменной становится синонимом имени подпрограммы.

3. Практическая часть

Описанная ниже последовательность разработки учебных программ предусматривает наличие в репозитории Delphi формы *fmExample*. Чтобы убедиться в ее наличии откройте репозиторий объектов (*Tools/Repositopy*). В случае отсутствия необходимо выполнить п. 3.1 практической части лабораторной работы «Основы программирования на Object Pascal в среде Delphi», где приводится последовательность создания и регистрации в репозитории заготовки для окна учебной формы.

3.1. Программа возведения в степень

С помощью опции *File/New Application* главного меню *Delphi* откройте новый проект. Сохраните проект под именем *Power.pas* в папке *C:\Projects\Power*, а модуль под именем *Main.pas* в той же папке.

В *Object Pascal* не предусмотрена операция возведения вещественного числа в произвольную степень. Эту задачу можно решить с использованием стандартных математических функций *Exp* и *Ln* по следующему алгоритму:

$$X^Y = e^{(Y*Ln(X))}.$$

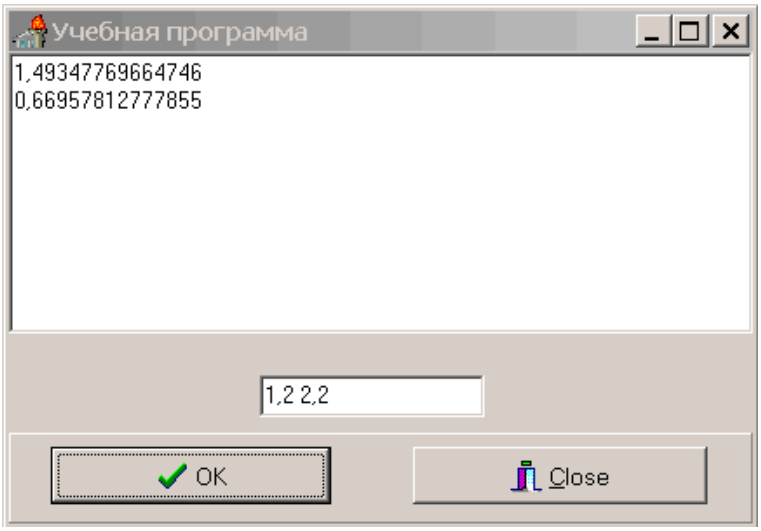
Создайте функцию с именем *Power* и двумя вещественными параметрами *A* и *B,* которая будет возвращать результат возведения *A* в степень *B*. Обработчик события *bbRunClick* учебной формы *fmExample* читает из компонента *edInput* текст и пытается выделить из него два числа, разделенных хотя бы одним пробелом. Если это удалось сделать, он обращается к функции *Power* дважды: сначала возводит первое число *X* в степень второго числа *Y,* затем *X* возводится в степень -*Y*.

```
procedure TfmExample.bbRunClick(Sender: TObject);
function Power(A, B: Real): Real;
{Функция возводит число A в степень B. Поскольку логарифм отрица-
тельного числа не существует, реализуется проверка значения A: от-
рицательное значение заменяется на положительное, для нулевого ре-
зультат равен нулю. Кроме того, любое число в нулевой степени дает
единицу.}
begin
     if A > 0 then
          Result := Exp(B * Ln(A))
     else if A <0 then
          Result := Exp(B * Ln(Abs(A)))
     else if B = 0 then
          Result := 1
     else
         Result := 0;
end;         // Power
var
   S: String;
   X, Y: Real;
begin
{Читаем строку из edInput и выделяем из нее два вещественных чис-
ла, разделенных хотя бы одним пробелом}
   S := edInput.Text;
   if (S = '') or (pos(' ',S) =0) then
      Exit;          // Нет текста или в нем нет
                     // пробела - прекращаем дальнейшую работу
   try
      // Выделяем первое число:
      X := StrToFloat(copy(S, 1, pos(' ', S) - 1)) ;
      // Если успешно, удаляем символы до пробела
      // и выделяем второе число:
      Delete (S, 1, pos (' ', S)) ;
      Y := StrToFloat(Trim(S));
   except
         Exit;    // Завершаем работу при ошибке преобразования
   end;

      mmOutput.Lines.Add(FloatToStr(Power(X, Y)));
      mmOutput.Lines.Add(FloatToStr(Power(X, -Y)));
   end;
```

Для вызова функции *Power* ее указали в качестве параметра при обращении к стандартной функции преобразования вещественного числа в строку *FloatToStr*. Параметры *X* и *Y* в момент обращения к функции *Power* – это фактические параметры. Они подставляются

вместо формальных параметров *A* и *B* в заголовке функции и затем над ними осуществляются нужные действия. Полученный результат присваивается специальной переменной с именем *Result,* которая в теле любой функции интерпретируется как то значение, которое вернет функция после окончания своей работы. В программе функция *Power* вызывается дважды – сначала с параметрами *X* и *Y,* а затем *X* и *-Y*.

Окно программы POWER показано на рисунке



Сохраните проект (*File/Save All*).

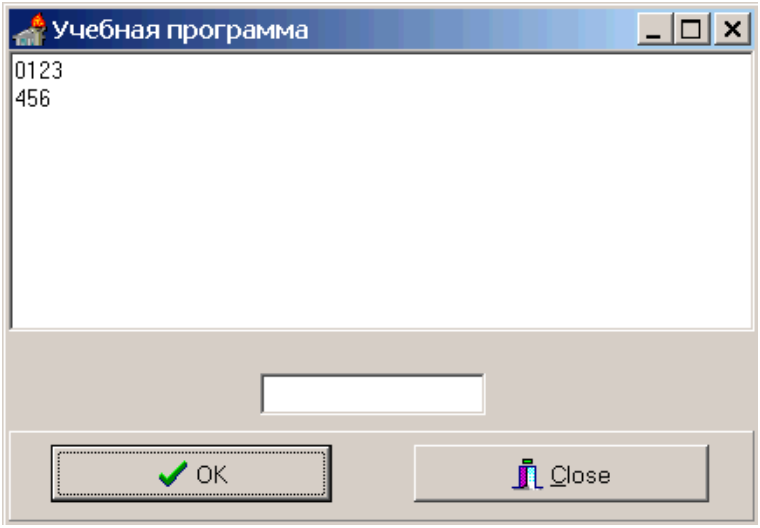
3.2. Программа вывода содержимого одномерных массивов переменной длины

Откройте новый проект и сохраните его под именем *Array.pas* в папке *C:\Projects\Array*, а модуль под именем *Main.pas* в той же папке. Введите в обработчик события *OnClick* кнопки *Run* следующий код:

```
procedure TfmExample.bbRunClick(Sender: TObject);
{Иллюстрация использования открытых массивов: программа выводит в
компонент mmOutput содержимое двух одномерных массивов разной дли-
ны с помощью одной процедуры ArrayPrint}
procedure ArrayPrint(aArray: array of Integer);
var
   k: Integer;
   S: String;
begin
   S  := '';
   for k := 0 to High(aArray) do
       S := S + IntToStr(aArray[k]);
   mmOutput.Lines.Add(S);
end;
const
   A: array [-1..2] of Integer = (0,1,2,3);
   B: array [5..7] of Integer = (4,5,6);
begin
   ArrayPrint(A);
   ArrayPrint(B);
end;
```

Как видно из этого примера, фактические границы массивов *A* и *B,* передаваемых в качестве параметров вызова процедуре *ArrayPrint,* не имеют значения. Однако размерность открытых массивов (количество индексов) всегда равна *1* – за этим следит компилятор.

Окно программы ARRAY показано на рисунке



Сохраните проект (*File/Save All*).

3.3. Передача процедур в качестве фактических параметров вызова

Откройте новый проект и сохраните его под именем *Relation.pas* в папке *C:\Projects\Relation*, а модуль под именем *Main.pas* в той же папке.

Программа выводит на экран таблицу двух функций:

Sin1(x) = (Sin(x) + 1) × Exp(-x) и *Cos1(x) = (Cos(x) + 1) × Exp(-x).*

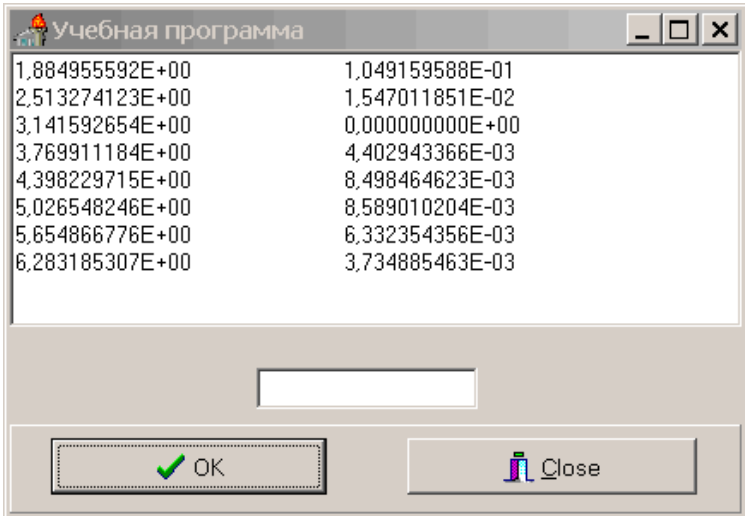
Вычисление и печать значений этих функций реализуются в процедуре *PrintFunc,* которой в качестве параметров передается количество *NP* вычислений функции в диапазоне *X* от 0 до 2π и имя нужной функции.

```
function Sin1(X: Real): Real;
begin
     Result := (Sin(X) + 1) * Exp(-X)
end; // Sin1

function Cos1(X: Real): Real;
begin
     Result := (Cos(X) + 1) * Exp(-X)
end; // Cos1

procedure TfmExample.bbRunClick(Sender: TObject);
type
    Func = function(X: Real): Real; // Процедурный тип
procedure PrintFunc(NP: Integer; F: Func) ;

var
   k: Integer;
   X: Real;
begin
  for k := 0 to NP do
```

```
  begin
    X := k * 2 * pi / NP;
   mmOutput.Lines.Add(FloatToStrF(X, ffExponent, 10, 2) +
      #9 + #9 + FloatToStrF(F(X), ffExponent, 10, 2)) ;
  end;
end;          // Print Func

begin        // bbRunClick
     mmOutput.Lines.Add(#9'Функция SIN1:') ;
     PrintFunc(10, Sinl) ;
     mmOutput.Lines.Add(#9'Функция COS1:');
     PrintFunc (10, Cosl) ;
end;
end.
```

Обратите внимание: передаваемые подпрограммы не могут быть локальными, т. е. процедурами или функциями, объявленными внутри другой подпрограммы. Вот почему описание подпрограмм *Sin1* и *Cos1* размещаются вне обработчика *bbRunClick,* но выше него по тексту модуля. Символ #9 – это символ табуляции, который вставляется в формируемые строки для разделения колонок с цифрами.

Окно программы RELATION показано на рисунке



Сохраните проект (*File/Save All*).

3.4. Пример рекурсивной программы вычисления факториала

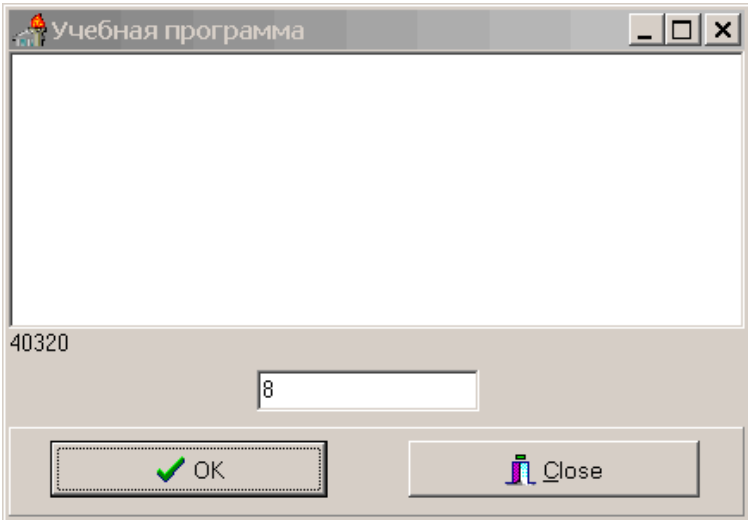
Откройте новый проект и сохраните его под именем *Factorial.pas* в папке *C:\Projects\Factorial*, а модуль под именем *Main.pas* в той же папке.

Программа получает от компонента *edInput* целое число *N* и выводит в компонент *IbOutput* значение *N!,* которое вычисляется с помощью рекурсивной функции *Factorial.*

```
procedure TfmExample.bbRunClick(Sender: TObject);
function Factorial(N: Word): Extended;
begin
     if N = 0 then
        Result := 1
     else
        Result := N * Factorial(N-1)
```

```
end;
var
   N: Integer;
   begin
        try
          N := StrToInt(Trim(edInput.Text));
        except
           Exit;
   end;
   lbOutput.Caption := FloatToStr(Factorial(N))
end;
```

Окно программы FACTORIAL показано на рисунке



Сохраните проект (*File/Save All*).