М6О-121С-21 Полковников Д.Н.

МЕТОДИЧКА ПО FORTRAN

Пособие по программированию на языке Fortran

Москва 2022 г.

Оглавление

1. Немного из истории, концепция языка.

1.1 Немного из истории.

Фортран – первый язык программирования высокого уровня, получивший практическое применение.Создан в период с 1954 по 1957 год группой программистов под руководством Джона Бэкуса в корпорации IBM. Название ***FORTRAN*** является сокращением двух слов: **FOR**mula и **TRAN**slator (Транслятор формул).

Фортран редко используют в индустрии, в основном этот язык программирования используют учёные для больших вычислений. К примеру, его могут использовать для крупномасштабных симуляций физических систем, молекулярной динамики, климатический моделей и т.д.

Фортран до сих пор используется благодаря своей долгой и богатой истории. На нём написано огромное количество программ, алгоритмов, библиотек и подпрограмм для любого рода вычислений. За всё время существования этого языка, все коды этих программ были доведены до их наилучшего состояния. Поэтому сейчас переделывать то, что и так хорошо работает дорого и бессмысленно. Также ***фортран*** является самым быстрым языком программирования, когда дело касается к примеру огромных многомерных матриц.

1.2 Концепция языка.

Программы на языке фортран начинаются с **PROGRAM** с объявлением имени и заканчиваются **END** с возможностью обращения к заданному имени**.** Пояснения можно писать после «**!**» (восклицательного знака). Выводимая информация после **PRINT** выделяется верхними запятыми **"выводимая информация "**. Запись этих символов (***,**) является синтаксисом фортрана и указывает бесформатный вывод. (О форматах в п.8) По умолчанию файл с текстом написанной в свободной форме программы имеет расширение F90.

Общий пример программы:
program (имя программы) **…** **<Блок операторов и конструкций>** **…** **end (имя программы)**
Полный пример программы:
program Hello **print*, "Hello World"** **end program Hello**

Имя объекта это выстроенная по определенным правилам последовательность символов.

В Фортране-90/95 действуют следующие правила записи имен:

1) *Прописные и строчные буквы алфавита эквивалентны в наборе символов Fortran. Другими словами, Fortran нечувствителен к регистру.*

Как следствие, переменные **a** и **A** являются одной и той же переменной. В принципе можно написать программу следующим образом:

pROgrAm MYproGRaM

...

enD mYPrOgrAM

Но потом самому придётся потратить не мало времени чтобы разобраться в своих записях.

2) *Символьная длина имени не должна превышать 31 символ.*

3) *Имя может состоять из алфавитно-цифровых символов (букв и цифр) и символа подчеркивания – «_».*

4) *Первым символом имени обязательно должна быть буква.*

Для имен (идентификаторов) и связанных с ними объектами данных, в Фортране существует правило неявного определения типов (Про типы данных п.2). Если в программе используется именованный объект данных (скаляр или массив), тип которого не определен явно, то объект данных считается:

1) *Целым (INTEGER), стандартной разновидности целого типа, если первым символом его имени будет одна из букв: 'I', 'J', 'K', 'L', 'M' 'N'.* ***Например****: J, IND, LSTREAM и т.д.*

2) *Вещественным (REAL), стандартной разновидности вещественного типа, если первым символом его имени будет любая другая буква.* ***Например****: X, DIN, EPSILON*

3) *Правил по неявному определению других типов в Фортране не существует.*

Неявное определение типов можно отключить с помощью **IMPLICIT NONE** его необходимо писать сразу же после **PROGRAM**.

2. Типы данных, сравнение вещественных чисел, полученных в результате вычислений.

2.1 Типы данных

В математике целые числа являются частью множества вещественных чисел. В Фортране термины "целый'' и вещественный" используются для обозначения двух различных типов констант (чисел). Есть и другие типы данных, но оставим в рассмотрении самые простые из них.

Целая константа – это последовательность цифр без десятичной точки, со знаком или без него. Для хранения целой константы в памяти ПЭВМ, как правило, используется 4 байта, т.е. 32 двоичных разряда, один из которых используется для хранения знака числа. Поэтому целая константа не может превышать 2^31–1. Константа без знака или со знаком + (плюс) воспринимается как положительная, со знаком – (минус) как отрицательная. Примеры целых констант: 145, -9363, +5434, 0 .

Вещественные константы с фиксированной точкой - эти константы состоят из целой части, точки и дробной части, например: 532.6527, -6256.33, 0.0 и т.д. Правило знаков для вещественных констант то же, что и для целых (знак "минус" обязателен, знак "плюс" не обязателен). Отсутствующую целую или дробную части можно не писать, но точка должна быть обязательно: 2.00 = 2.0 = 2.; 0.01 = .01; +0.25 = .25; -0.25 = -.25; 0.0 = 0. = .0.

2.2 Сравнение вещественных чисел, полученных в результате вычислений.

На результат арифметических операций сказывается то, что все числа хранятся в памяти компьютера с конечным числом значащих цифр. В силу этого деление зачастую нельзя выполнить точно (например: 1./3.=0.333333...), а при сложении, вычитании и умножении пропадают младшие разряды результата. При этом следует иметь в виду, что во всех случаях в Фортране происходит не округление, а отбрасывание (усечение) младших разрядов результата, не умещающихся в ячейке памяти.

При вычислении вещественных числе может накапливаться погрешность.

```
program vesh
real a,b,c
c=6.38
a=8.94*c
b=a/8.94
print*, b
end program vesh
```

В результате получим `6.38000011`, хотя по идее мы должны были получить 6.38000000. при дальнейших вычислениях погрешность может увеличится.

```
program vesh
real a,b,c
c=6.38
a=8.94*6.38
b=a/8.94
b=b*3
b=(b/3)
print*, b
if(b>6.38) print*,'bol'
if(b<6.38) print*,'men'
end program vesh
```

```
 6.37999964
men
```

Тут Фортран уже считает полученное число меньше исходного. Хотя математически мы ничего не изменили.

Для сравнения подобных очень похожих чисел удобно использовать разность между ними и сравнение результата с довольно маленьким числом.

```
program vesh
real a,b,c
c=6.38
a=8.94*6.38
b=a/8.94
b=b*3
b=(b/3)
print*, b
if((b-c)<0.000001) print*,'ravn'
if((b-c)>0.000001) print*,'b bol'
if((c-b)>0.000001) print*,'b men'
end program vesh
```

```
 6.37999964
ravn
```

В итоге мы получаем результат, точность которого вполне достаточна в обыденных задачах.

3. Базовые алгоритмы, вложенные структуры. Конструирование алгоритмов.

3.1 Базовые алгоритмы, вложенные структуры.

Алгоритмы могут быть записаны в виде 3-х основных структур:

1) Блок операторов.

Это просто подряд идущие операторы, которые исполняются сверху вниз.

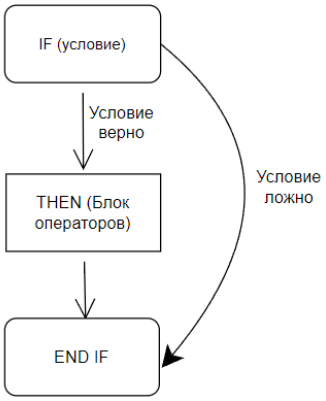
Пример:
program Block **integer :: a, b, c** **a=1** **b=2** **c=a+b** **print*, c** **end program Block** 3
Здесь операторы исполняются один за другим, сначала присваивания, потом присваивание суммы и в конце вывод.

***2) Ветвления. (Условный оператор* IF).**

В этом варианте структуры программа выбирает один из предложенных путей в зависимости от заданного условия. Записывается в виде:

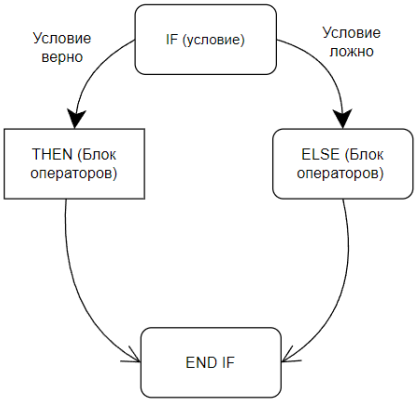
```
if (условие) then
        <блок операторов>
end if
```

Схематично выглядит так:

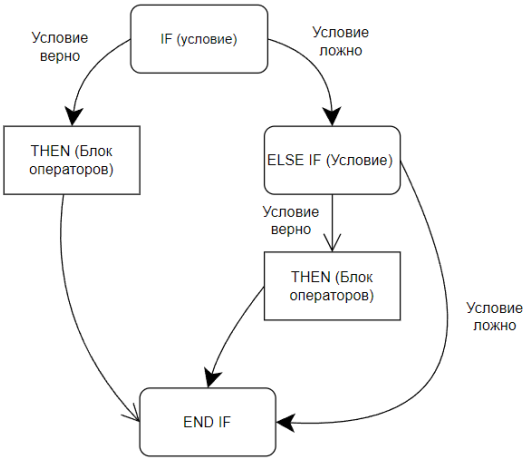


Существует ещё несколько вариантов ветвления: с использованием **ELSE** и **ELSE IF**. **ELSE** означает ИНАЧЕ (т.е. при не выполнение условия программа не завершает условный

оператор, а совершает иное действие). В первом случае просто включает другой блок операторов при ложном главном условии:



А во втором случае происходит проверка ещё одного условия. И в этом случае только если оно верно, то включается другой блок операторов.



Подобное ветвление может быть довольно большим, и их также можно комбинировать.

```
if (условие 1) then
        <Блок операторов 1>
else if (условие 2) then
        <Блок операторов 2>
else if (условие 3) then
        <Блок операторов 3>
        ……
else
        <Блок операторов>
end if
```

```
program Esli
integer :: a, b, c, h
a=1
b=2
c=a+b
h=0
if (a>b) then
        h=1
else if (a>c) then
   h=2
else if (b>c) then
   h=3
else
   h=4
end if
print*, h
end program Esli
```

```
 4
```

Вложенный условный оператор

Условные операторы могут быть вложенными друг в друга.

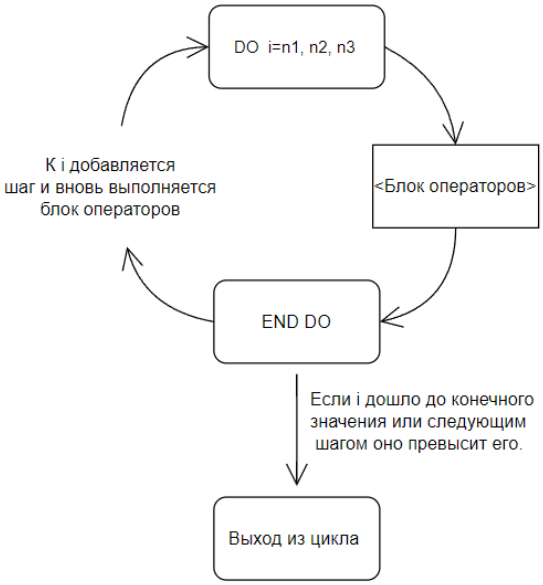
if (условие 1) then
<Блок операторов 1>
if (условие 2) then
…
<Блок операторов n >
…
end if
end if
	program esli_vloj
	integer :: a,b,c,h
	a=1
	b=2
	c=a+b
	h=0
	if (c>a) then
	h=h+1
	if(c>b) then
	h=h+2
	end if
	end if
	print*, h
	end program esli_vloj
	3

3) ***Цикл (DO)***

Бывают ситуации, когда приходится сделать какую-либо операцию, или несколько операций определённое количество раз. В этом случае на помощь приходит цикл. Он «прогоняет» блок операторов заданное нами количество раз (или полученное в ходе программы).

Есть ***Цикл с шагом*** *(цикл проходит от одного числа до другого с определённым шагом, по умолчанию это один):*	
do i=n1, n2, n3 **<Блок операторов>** **end do** **n1 –** Начальное значение переменной цикла **i** **n2 -** Конечное значение переменной цикла **i** **n3 –** Шаг с которым цикл идёт от начального значения к конечному.	**program Cycle** **integer :: b** **b=2** **do i=1, 7, 2** **b=b*2** **end do** **print*, b** **end program Cycle** 32
А есть ***Цикл пока*** или ***Цикл с условием*** *(он работает пока некоторое условие остаётся истинным):*	
do while (условие) **<Блок операторов>** **end do**	**program Cycle** **integer :: b** **b=2** **do while (b<100)** **b=b*2** **end do** **print*, b** **end program Cycle** 128

Схематично циклы выглядят так:



Вложенный цикл.

Цикл также может быть вложен в другой цикл.

```
do i=n1, n2, n3
<Блок операторов 1>
do j=n1, n2, n3
        <Блок операторов 2>
        …
end do
end do
```

```
program Cycle_vloj
do i=1, 2
do j=1, 2
                print*, i,j
end do
end do
end program Cycle_vloj
```

```
1           1
1           2
2           1
2           2
```

3.2 Конструирование алгоритмов.

ЭВМ не сможет выполнить задачу если ей просто задать условие, программисту необходимо составить алгоритм (последовательность действий), следуя которому компьютер выдаст нужный нам ответ.

1)*Для составления программы прежде всего необходимо уяснить суть задачи.*

2)*Затем понятным для себя языком расписать ход её решения (обычно решение полностью можно описать словами).*

3)*По написанному составляем алгоритм, описывая в отдельности совершаемые действия.*

4)*И в самом конце после проверки логики алгоритма (т.е. насколько правильно он работает и нет ли каких-нибудь недочётов, из-за которых всё может «сломаться») переводим написанное на язык программирования (пишем код), используя вышеперечисленные структуры (п.3.1).*

4. Итерационные алгоритмы. Ряды. Смена знака.

4.1 Итерационные алгоритмы.

Итерация - циклическая управляющая структура, которая содержит композицию и ветвление, предназначенное для организации повторяющихся процессов обработки последовательности значений переменных.

Итерационный цикл - оператор цикла, для которого число повторений тела цикла заранее неизвестно. В итерационных циклах на каждом шаге вычислений происходит последовательное приближение и проверка условия достижения искомого результата. Выход из итерационного цикла происходит если заданное условие принимает значение **ложь**.

Алгоритм, в состав которого входит итерационный цикл, ***называется итерационным алгоритмом.*** Итерационные алгоритмы используются при реализации итерационных численных методов. В итерационных алгоритмах необходимо обеспечить обязательное достижение условия выхода из цикла ***(сходимость итерационного процесса).*** В противном случае произойдет ***зацикливание*** алгоритма, т.е. не будет выполняться основное свойство алгоритма – не будет конечного результата.

4.2 Ряды.

Для вычисления частичной суммы ряда удобно использовать ***метод итераций***. В итерационных алгоритмах решение задачи реализуется путем последовательного приближения к искомому результату. Процесс является циклическим, поскольку заключается в многократных вычислениях. Начальное приближение Y_0 выбирается заранее или задается по определенным правилам. Заканчивается итерационное вычисление при выполнении условия $|Y_i - Y_{i-1}| <$**eps**

где **eps** - допустимая ошибка вычисления (Заданная точность).

Пример вычисления частичной суммы ряда $\sum_{i=0}^{n} \frac{1}{2^n}$ **:**

```
program ryad
real :: eps, s_new, s_old, n
eps=0.00001
s_new=0
s_old=0.1
n=0

do while(abs(s_old-s_new)>eps)
        s_old=s_new
        s_new=s_new+1./(2**n)
        n=n+1
end do

print*, n, s_new   ! Количество итераций и искомая сумма
end
```

```
 18.0000000       1.99999237
```

4.3 Смена знака.

Существуют задачи, для вычисления которых необходимо производить смену знака.

Выясним как сделать это эффективней (где время вычислений меньше та программа и эффективней):

```
real(4) :: start_time, finish_time
call cpu_time(start_time)

k=(-1)**100

print*, k
call cpu_time(finish_time)
print *, 'Время вычислений = ', finish_time - start_time
end
```

```
           1
Время вычислений =     3.10000032E-05
```

```
program smena
real(4) :: start_time, finish_time
call cpu_time(start_time)
s=0
k=1
do n=1, 100
   k=k*(-1)
end do
print*, k
call cpu_time(finish_time)
print *, 'Время вычислений = ', finish_time-start_time
end
```

```
           1
Время вычислений =     4.60000010E-05
```

Как видим быстрее выполняется первый вариант, соответственно его использовать выгоднее, может при небольших вычислениях разница не существенна, при крупномасштабных подсчётах использование первого варианта будет намного выгоднее.

5. Одномерные массивы - сортировки, поиск элементов. Динамические массивы.

5.1 Одномерные массивы

Одномерные массивы – массивы, в которых элементы пронумерованы последовательно по порядку: первый элемент, второй, третий и т.д. Для обозначения элементов одномерного массива используется один индекс. Индексы пишутся в скобках; если индексов несколько, то они разделяются запятыми. Индексы всегда идут от начального до максимального значения подряд. В качестве индексов могут использоваться константы, переменные и арифметические выражения. По аналогии с математикой ***одномерные числовые массивы*** часто называют векторами. Память ЭВМ устроена линейно (как одна огромная строка): для массива выделяется линейный участок памяти, в котором подряд располагаются его элементы.

Описание массивов производится, например, с помощью неисполняемого оператора **DIMENSION.**

5.1.1 Сортировки.

Сортировка – это упорядочивание набора однотипных данных по в некотором порядке (например: по возрастанию или убыванию). Она применяется для эффективного выполнения вычислений, предполагающих упорядоченность данных (построение функций, поиск) и часто облегчает интерпретацию результатов.

Сортировка по возрастанию путём передвижения бОльших элементов вправо.	***Сортировка по убыванию***, всё тоже самое, но с исполнением справа на лево.

program sortirovka_po_vozrastaniyu
integer a(100)
print*, 'dlina massiva n'
read*, n
print*, 'zapolnite massiv'
read*, (a(i),i=1,n)
print*, 'ish massiv'
print*, (a(i),i=1,n)
do k=1, n-1
do i=1,n-k
if (a(i).gt.a(i+1)) then
t=a(i)
a(i)=a(i+1)
a(i+1)=t
endif
enddo
enddo
print*, 'ism massiv'
print*, (a(i),i=1,n)
end
	program sortirovka_po_ubyvaniyu
	integer a(100)
	print*, 'dlina massiva n'
	read*, n
	print*, 'zapolnite massiv'
	read*, (a(i),i=1,n)
	print*, 'ish massiv'
	print*, (a(i),i=1,n)
	do k=1, n-1
	do i=n, 1+k, -1
	if (a(i).gt.a(i-1)) then
	t=a(i)
	a(i)=a(i-1)
	a(i-1)=t
	endif
	enddo
	enddo
	print*, 'ism massiv'
	print*, (a(i),i=1,n)
	end

ish massiv 3 4 1 5 2 ism massiv 1 2 3 4 5	ish massiv 3 1 5 2 4 ism massiv 5 4 3 2 1

5.1.2 Поиск элементов.

Поиск элементов в массиве является довольно частой задачей, чаще всего необходимо найти максимальный и минимальный элементы. Вот эффективные алгоритмы их поиска:

Поиск ***максимального*** элемента(идёт сравнение элементов, начиная с первого, с элементом, который на данный момент выбран максимальным).	Поиск ***минимального*** элемента(идёт сравнение элементов, начиная с первого, с элементом, который на данный момент выбран минимальным).
program Max_el **integer a(100)** **print*, 'dlina massiva n'** **read*, n** **print*, 'zapolnite massiv'** **read*, (a(i),i=1,n)** **print*, 'ish massiv'** **print*, (a(i),i=1,n)** **max=a(1)** **do i=2, n** **if (a(i)>max) max=a(i)** **enddo** **print*, 'otv'** **print*, max** **end**	**program Min_el** **integer a(100)** **print*, 'dlina massiva n'** **read*, n** **print*, 'zapolnite massiv'** **read*, (a(i),i=1,n)** **print*, 'ish massiv'** **print*, (a(i),i=1,n)** **min=a(1)** **do i=2, n** **if (a(i)<min) min=a(i)** **enddo** **print*, 'otv'** **print*, min** **end**
ish massiv -34 345 0 176 otv 345	ish massiv -54 0 43 876 otv -54

5.2 Динамические массивы

Массивы могут быть ***статическими*** (именно их мы и рассмотрели выше) и ***динамическими.*** Под ***статические массивы*** на этапе компиляции выделяется заданный объем памяти, которая занимается массивом во все время существования программы. Память под ***динамические массивы*** выделяется в процессе работы программы и при необходимости может быть изменена или освобождена.

<u>Основные операторы для использования динамического массива:</u>

INTEGER,ALLOCATABLE,DIMENSION (:) :: MASS
! Объявление динамического массива, размерность указана в скобках (также может быть **(:,:)***;* **(:,:,:)***; и т.д). Вместо* **INTEGER** *может быть* **REAL**.

ALLOCATE(A(N))
! Выделение ячеек памяти под динамический массив, их количество равно значению переменной в скобках.

…

DEALLOCATE(A)
! Очистка памяти, которую занимал динамический массив.

Простейший пример динамического массива:

```
program Din_mass
integer,allocatable,dimension(:) :: a
print*, 'dlina massiva n'
read*, n
allocate(a(n))
print*, 'zapolnite massiv'
read*, a
print*, 'ish massiv'
print*, a
deallocate(a)
end
```

```
 dlina massiva n
4
 zapolnite massiv
765
23
-65
0
 ish massiv
         765          23         -65           0
```

В этой программе выделилось такое количество памяти, которое нам необходимо и нет никаких лишних, незаполненных ячеек, что кстати и позволило нам упростить написание ввода и вывода массива. (Мы написали просто «**read*, a**» вместо «**read*, (a(i), i=1, n)**», то же с «**принтом**»). Переменная «**n**» может быть вычислена в ходе программы.

6. Структуризация в программировании – подпрограммы.

6.1 Структуризация в программировании.

Сложная задача всегда разбивается на простые и сводится к минимальному числу базовых структур (последовательности, ветвления и циклы). Итоговая программа обязательно должна быть структурирована - тело цикла, блоки условных операторов - сдвинуты вправо. Команды, по логике задачи, идущие линейно, должны быть написаны с одной позиции. Перед вводом и выводом информации должна быть пояснительная надпись (что и для чего вводится/выводится).Конечная программа будет иметь вид «лестницы»:

```
PRINT*, 'Vvedite razmernost matrici NxM'
READ*,  N, M

PRINT*, 'Zapolnite matricu razmerom NxN'
DO I=1,N
    READ*, (A(I,J), J=1,M)
ENDDO

PRINT*, 'Ishodnaya matrica:'
DO I=1,N
    PRINT*, (A(I,J), J=1,M)
ENDDO

MIN=A(1,1)
DO I=1,N
    DO J=1,N
        IF (A(I,J).LT.MIN) THEN
            MIN=A(I,J)
            Imin=I
            Jmin=J
        ENDIF
    ENDDO
ENDDO
```

Структурное программирование — технология разработки программного обеспечения, в основе которой лежит представление программы в виде иерархической структуры логически целостных фрагментов (блоков).

Основные принципы структурного программирования заключаются в том, что:

1) *любая программа строится из трёх базовых управляющих конструкций: последовательность, ветвление(**IF**), цикл(**DO**);*

2) *в программе базовые управляющие конструкции могут быть вложены друг в друга произвольным образом;*

3) *повторяющиеся фрагменты программы можно оформить в виде **подпрограмм** (процедур и функций). В виде **подпрограмм** можно оформить логически целостные фрагменты программы, даже если они не повторяются;*

4) *все перечисленные конструкции должны иметь один вход и один выход;*

5) *разработка программы ведётся пошагово, методом «сверху вниз».*

Вспомогательный алгоритм — это алгоритм, целиком используемый в составе другого алгоритма. Запись ***вспомогательных алгоритмов*** в языках программирования осуществляется с помощью ***подпрограмм***.

6.2 Подпрограммы.

Вообще говоря, в реальных задачах, программы выглядят как набор вызываемых ***подпрограмм***. В фортране существуют ***подпрограммы функции*** и просто ***подпрограммы***.

Подпрограмма-функция является отдельной программной единицей и может быть откомпилирована отдельно от основной программы и записана в библиотеку подпрограмм. ***Подпрограмма-функция*** может вычислить только одно число, являющееся значением функции в зависимости от значений аргументов. Т.е. это отображение n-мерного пространства множества значений аргументов на одномерное пространство множества значение функции.

Подпрограммой называется поименованная часть программы, имеющая формальные параметры. ***Подпрограмма*** является отдельной программной единицей и может быть откомпилирована отдельно от основной программы и записана в библиотеку подпрограмм. ***Подпрограмма*** является расширением ***подпрограммы – функции,*** и ее можно рассматривать как математическое отображение n-мерного пространства множества значений аргументов на m-мерное пространство множества значений отображения. Обычно ***подпрограмма*** имеет ***входные*** и ***выходные параметры***. ***Входные параметры*** – это такие параметры, которые остаются без изменения в подпрограмме***. Выходные параметры***, это параметры, которые при вызове ***подпрограммы*** имеют неопределенное значения, а в ходе выполнения программы получают определенное значение, зависящее от входных параметров. Правда, часто параметры бывают одновременно и входными, и выходными.

Основные операторы для использования подпрограмм-функций:

FUNCTION имя (a1,a2,...,an)
неисполняемые операторы (операторы описания).
исполняемые операторы, среди которых должен быть оператор.
присваивания: имя= значение или выражение.
END
Вызывается подпрограмма функция по **имени** (в примере будет понятней).

Пример:

```
function sq(x)          ! Как мы видим результат вычисления подпрограммы- функции
implicit none           ! вызывается по имени функции sq
real x
real :: sq
sq=x*x
end function sq
!-----------------------
program Func
implicit none
real :: a, b=5, c=10, sq
a = b + sq(c)
print*, a
end program
```

```
 105.000000
```

Основные операторы для использования подпрограмм:

SUBROUTINE имя (a1,a2,...,an) **!** ***Оператор описания подпрограммы***
неисполняемые операторы (операторы описания)
исполняемые операторы
END

где **имя** - имя подпрограммы; **ai** - формальные параметры (аргументы).

CALL ИМЯ(y1,y2,....,yn) **!** ***Оператор вызова подпрограммы***

где **y1,y2,...,yn** - фактические параметры, которые могут быть константами (если формальные параметры являются только входными параметрами), простыми переменными любого типа, элементами массивов, массивами, структурированными элементами, арифметическими выражениями, именами подпрограмм или подпрограмм-функций, именами встроенных функций или подпрограмм. Естественно, между формальными и фактическими переменными должно быть соответствие типов

Пример:

```
program main
implicit none
 real :: a, b,r
 a = 4.0
 b = 12.0
 call multiply(a,b,r)
 print*, r
end program main
!---------------------------------
subroutine multiply(x,y,rez)
 implicit none
 real x,y,rez
 rez = x*y
end subroutine multiply
```

```
48.0000000
```

7. Двумерные массивы. Передача двумерных массивов в подпрограммы.

7.1 Двумерные массивы.

Двумерные массивы **–** это массивы данные в которых представлены в виде таблицы (в математическом представлении - матрица чисел), где положение каждого элемента задаётся номером строки и номером столбца. Следовательно для обозначения элемента двумерного массива необходимо два индекса: **a(i,j),** где **i** – номер строки, **j** – номер столбца.

Двумерный массив можно выводить по столбцам и по строкам.

Вывод двумерного массива по строкам.

```
integer a(100,100), n,m
print*, 'vvedite razmernost matrici nxm'
read*,  n,m
print*, 'zapolnite matricu razmerom nxm'

do i=1,n                    ! Блок ввода двумерного массива
   read*, (a(i,j), j=1,m)
enddo

do i=1,n                    ! Блок вывода двумерного массива по строкам
   print*, (a(i,j), j=1,m)
enddo

end
```

```
 vvedite razmernost matrici nxm
2 3
 zapolnite matricu razmerom nxm
1
2
3
4
5
6
           1           2           3
           4           5           6
```

Вывод двумерного массива по столбцам.

```
integer a(100,100), n,m
print*, 'vvedite razmernost matrici nxm'
read*,  n,m
print*, 'zapolnite matricu razmerom nxm'

do j=1,m          ! Блок ввода двумерного массива
   read*, (a(i,j), i=1,n)
enddo

do j=1,m          ! Блок вывода двумерного массива по столбцам
   print*, (a(i,j), i=1,n)
enddo

end
```

```
 vvedite razmernost matrici nxm
2 3
 zapolnite matricu razmerom nxm
1
2
3
4
5
6
           1           2
           3           4
           5           6
```

Для прохождения по всем элементам массива используют следующие вложенные циклы:

Для прохождения по строкам:

```
…
do i=1,n
   do j=1, m
     <Блок операторов>
   end do
end do
…
```

Для прохождения по столбцам:

```
…
do j=1,m
   do i=1, n
     <Блок операторов>
   end do
end do
…
```

7.2 Передача двумерных массивов в подпрограммы.

Для передачи двумерного массива в подпрограмму необходимо передай ей данные которыми описывается этот двумерный массив, а именно имя массива и размерность.

Пример:

```
program Peredacha
integer A(10,10)
print*, 'vvedite razmernost matrici nxm'
read*,  n,m
print*, 'zapolnite matricu razmerom nxm'
do i=1,n                    ! Блок ввода двумерного массива
   read*, (A(i,j), j=1,m)
enddo
call mass (A,n,m)           ! Вызов подпрограммы
end
!=================
subroutine mass (A,n,m)   ! Подпрограмма
integer A(10,10)
do i=1,n                    ! Блок вывода двумерного массива по строкам
   print*, (A(i,j), j=1,m)
enddo
end subroutine
```

```
 vvedite razmernost matrici nxm
2
2
 zapolnite matricu razmerom nxm
1
2
3
4
           1           2
           3           4
```

Здесь **A –** имя массива. **n, m –** его размерность.

8. Форматный ввод, вывод.

Форматный ввод/вывод данных позволяет управлять шириной полей ввода вывода, количество значащих цифр в числовых данных, а также отступами, пробелами, табуляцией переводами строк и другими параметрами представления – т.е. форматом данных при их вводе и выводе.

Перевод данных из внутреннего представления в текстовое задается ***дескрипторами преобразований (ДП)***. Дескрипторы преобразования содержатся в ***спецификации формата***. ***Спецификация формата*** включает заключенный в скобки список ***ДП***.

Для форматного ввода/вывода данных используют оператор **FORMAT**, на который операторы ввода/вывода ссылаются при помощи метки.

Общий вид оператора: **m FORMAT(c1,c2, . . . , cn)**

где **m** - обязательная метка оператора (любое целое число); **c1,c2, . . . , cn** – список ДП. Метка (в операторе **PRINT*,)** записывается вместо звёздочки: **PRINT m, 'Какой-то текст'** .

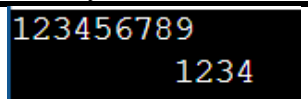
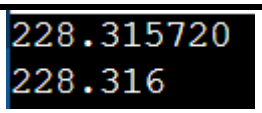
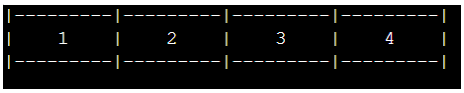
Для каждого типа данных существует своя спецификация: для ***целых*** величин - спецификация **I** и **G**; ***для вещественных*** величин - **F, E** и **G**; для ***вещественных*** величин ***двойной точности*** - **D** и **G**; для ***логических*** величин - **L** и **G**; для ***текстовых*** величин спецификация **A**. Спецификация **X** имеет вид **wX** и предназначена для пропуска **w** позиций. Символы управления переходом на новую строку **/**, **$**. При встрече символа **/** (слеш) происходит переход к следующей строке, а при встрече символа **$** происходит запрещение перехода к следующей строке, и следующий вывод производится на данной строки.

Запись повторяющихся спецификаций:

m FORMAT (n(c1, c2,…, cn)) где ***n*** **–** количество раз исполнения спецификации в скобке после ***n***

Некоторые Дескрипторы преобразования данных.		
Дескриптор	***Тип аргумента***	***Внешнее представление***
Iw	Целый	Целое число
Fw.d	Вещественный	Вещественное число
Lw	Логический	T и F, .T и .F, .TRUE. .FALSE.
A[w]	Символьный	Строка символов
Xw	——	Пробел
$	——	запрет перехода к следующей строке
/	——	переход к следующей строке

В таблице использованы следующие обозначения:
- w - длина поля, отведенного под представление элемента В/В(ввода вывода);
- d - число цифр после десятичной точки (d < w).

Примеры:	
Программа	***Результат***
program Form !Форматный ввод целого числа **integer :: n** **read 1, n** **print*, n** **1 format (i4)** **end**	
program Form !Форматный вывод вещественного числа **read *, t** **print 1, t** **1 format (f7.3)** **end program**	
program Form ! Печать следующего символа на **integer :: a(5) !следующей строке** **a = (/1,2,3,4,5/)** **print 1, a** **1 format (i10,/)** **end**	
program Form ! Запрет перехода на следующую строку **real :: a(10)** **read*, n** **do i=1, n** **read *, a(i)** **end do** **do i=1, n** **print 4, a(i)** **end do** **4 format (f10.1,$)** **end program**	
program Form ! Вид таблицы созданной **integer m(4) ! форматным выводом** **m = (/1,2,3,4/)** **print 2,** **print 1, m** **print 2,** **1 format (5('|', i5,4x))** **2 format (4('|---------'),'|')** **end program**	

9. Что такое эффективный алгоритм?

С понятием эффективности связано понятие сложности. Они взаимообратны. Чем более эффективен алгоритм, тем он менее сложен и наоборот. Будем употреблять их как синонимы.

Эффективность алгоритма определяется несколькими компонентами:

Интеллектуальная эффективность. При анализе интеллектуальной сложности алгоритма исследуется понятность алгоритмов и сложность их разработки.

Описательная эффективность является характеристикой способа, которым задается алгоритм, его описания, программы (объем программы, длина записи, число команд и т.д.)

Вычислительная эффективность характеризует сложность переработки алгоритмом каждого значения исходных данных, к которым он применим (время работы, емкость памяти и т.д.)

Мы, анализируя алгоритм с точки зрения вычислительной эффективности будем говорить о двух ее составляющих: памяти (или пространство) и времени.

Пространственная эффективность измеряется количеством памяти, требуемой для выполнения алгоритма.

Компьютеры обладают ограниченным объемом памяти. Если две программы реализуют идентичные функции, то та, которая использует меньший объем памяти, характеризуется большей пространственной эффективностью. Иногда память становится доминирующим фактором в оценке эффективности программ. Однако в последние годы в связи с быстрым ее удешевлением эта составляющая эффективности постепенно теряет свое значение.

Временная эффективность алгоритма определяется временем, необходимым для ее выполнения.

Все эти формы сложности обычно взаимосвязаны. Как правило, при разработке алгоритма с хорошей временной оценкой сложности приходится жертвовать его пространственной и/или интеллектуальной сложностью. Например, алгоритм быстрой сортировки существенно быстрее, чем алгоритм сортировки выборками. Плата за увеличение скорости сортировки выражена в большем объеме необходимой для сортировки памяти. Необходимость дополнительной памяти для быстрой сортировки связана с многократными рекурсивными вызовами.

Также можно добавить, что для алгоритма нужно найти наиболее важный фактор, который сильнее всего влияет на эффективность алгоритма и дальнейшие улучшения проводить в его сторону. Т.е. если задача не требует высокой скорости исполнения (наш ресурс - «Время», довольно велик) мы можем сократить к примеру объём используемой памяти, увеличив тем самым время работы алгоритма и так получается, что в определённой задаче именно это решение будет более эффективным.

Список литературы.

1. Бартеньев О. В. - Современный Фортран. 2000г. (3-е изд., допополненное и переработанное)

2. А.М. Купфер, Ж.И. Мехалая, Ю.В. Осипов - ФОРТРАН Учебное пособие 2004г.

3. Ю.И.Рыжиков – Программирование на фортране powerstation для инженеров. Практическое руководство.

4. Н.А. Берков Н.Н. Беркова - Учебное пособие. Алгоритмический язык фортран 90. 1998г.