

"بسمه تعالی"



FORTRAN

Programming language

Autumn 1399



دانشکده فنی و مهندسی

مهندسی کامپیوتر گرایش نرم افزار

عنوان:

"بررسی زبان برنامه نویسی Fortran"

استاد:

"سرکار خانم ساناز کشوری"

تهیه شده توسط:

"اتوسا طغیانی"

تاریخ پروژه:

99/10/19

فهرست:

Introduction	5
History	5
Types of versions	6
interpreter OR Compiler	8
Syntax	10
Basic	11
Operator	11
Terminology	13
Enumerator	15
Data type	15
Variable	17
Data object	18
Characters	21
Binding	22
Conditionals	23
Loop	28
Pointer	33
File	36
Derived data type	39
Record	42
Modul	44
Arrays	47
Procodure	53
Intrinsic functions	54

Class	56
OOP	57
Static and Dynamic Scoped	59
Library	62

مقدمه:

فرترن (Fortran) یک زبان برنامه‌نویسی کامپایل شونده و دستوری می‌باشد که برای محاسبات عددی بسیار مناسب است. فرترن در گذشته به صورت حروف بزرگ FORTRAN نوشته می‌شده است که کوتاه‌شده FORMula TRANslation است. طراحی Fortran پایه و اساس بسیاری از زبان‌های برنامه‌نویسی دیگر بود. از جمله شناخته شده ترین موارد BASIC است که مبتنی بر FORTRAN II است.

از ویژگی‌های مهم فرترن سرعت بالا در کامپایل کردن کدهای محاسباتی است و دلیل این امر نیز بهینه‌بودن دستورات این زبان نسبت به زبان‌های دیگر است و با توجه به کمینه بودن تعداد کاراکترهای مورد نیاز برای ایجاد یک دستور و امکانات ترجیحی و اختیاری فراوان، کدهای نوشته‌شده با این زبان از حجم کمی برخوردارند و در زمان کوتاهی اجرا شده و نتیجه می‌دهند؛ این امر برای انجام محاسبات عددی سنگین اهمیت ویژه‌ای دارد؛ بیش از چندین دهه است که برای محاسبات پیچیده، پیش‌بینی آب و هوا، دینامیک سیالات و ... مورد استفاده قرار می‌گیرد.

ویژگی‌های فرترن:

- سرعت بالای محاسبات : در برنامه‌های محاسباتی علاوه بر امکانات باید به حجم محاسبات نیز دقت شود؛ حجم محاسبات بعضی از پدیده‌های فیزیکی مثل مدل‌سازی ستاره و کهکشان‌ها، هواشناسی و ... بسیار زیاد است؛ فرترن برای انجام چنین محاسباتی بسیار خوب عمل می‌کند.
- سادگی نسبت به دیگر زبان‌ها: نسبت به زبان متلب و ++C ، کار با فرترن آسان‌تر است.
- سادگی کار با آرایه‌ها: به خصوص آرایه‌های چند بعدی.
- قابل حمل بودن

تاریخچه:

در اواخر ۱۹۵۳ یک پیشنهاد جهت جایگزین کردن یک برنامه عملی‌تر به جای زبان اسمبلی، برای برنامه نویسی کامپیوتر اصلی IBM 704 توسط جان بکوس (John Backus) در شرکت IBM ارائه شد.

تیم برنامه‌نویسان فرترن متشکل از: Richard Goldberg, Sheldon F. Best, Harlan, Herrick Peter Sheridan, Roy Nutt, Robert Nelson, Irving Ziller, Harold Stern, Lois Haibt, David Sayre بود. برخی از این برنامه‌نویسان بازیکن شطرنج بودند و با این تفکر که ذهن منطقی دارند ، برای کار در IBM انتخاب شدند.

هدف ایجاد زبان فرترن ورود آسان تر معادلات به کامپیوتر بود. اولین کتابچه راهنمای آن در اکتبر ۱۹۵۶ آماده شد و اولین کامپایلر فرترن در آوریل ۱۹۵۷ ارائه شد؛ این کامپایلر اولین کامپایلر بهینه‌ساز بود و می‌توان فرترن را جزء اولین زبان‌های برنامه نویسی سطح بالا دانست.

در فرترن، تعداد عبارات برنامه نویسی لازم برای کار با ماشین با ضریب ۲۰ کاهش داده شده است و همین امر سبب شد که توجه برنامه‌نویسان به آن جلب شود و مورد پذیرش قرار گیرد.

جان بکوس در طی مصاحبه‌ای گفت: "بخش عمده‌ای از کار من از تنبلی ناشی شده است. من نوشتن برنامه‌ها را دوست نداشتم، بنابراین هنگامی که در IBM 701 کار میکردم، برنامه‌هایی برای محاسبات می‌نوشتیم. پس شروع به کار بر روی یک سیستم برنامه نویسی کردم تا نوشتن برنامه‌ها آسان تر شود".

تا سال ۱۹۶۰ نسخه‌های فرترن برای رایانه‌های IBM 109، IBM 650، IBM 1620 و IBM ۷۰۹۰ در دسترس بود. محبوبیت روزافزون این زبان، سبب شد که تا سال ۱۹۶۳، بیش از ۴۰ کامپایلر فرترن وجود داشت؛ به همین دلایل، FORTRAN اولین زبان برنامه نویسی بین پلتفرمی پرکاربرد در نظر گرفته می‌شود.

توسعه فرترن به موازات سیر تکاملی کامپایلر بود و به همین دلیل بسیاری از پیشرفت‌های تئوری و طراحی کامپایلر در جهت رفع نیازهای فرترن برای ایجاد کد کارآمدتر بود.

انواع نسخه‌ها:

FORTRAN II: در سال ۱۹۵۸ عرضه شد؛ پیشرفت اصلی این نسخه، پشتیبانی از برنامه نویسی رویه‌ای با اجازه دادن به زیر برنامه‌ها و توابع نوشته شده توسط کاربر است که مقادیر را با پارامترهای منتقل شده توسط مرجع باز می‌گرداند.

FORTRAN III: این نسخه هرگز به عنوان یک محصول منتشر نشد چون مانند FORTRAN ۷۰۴ و FORTRAN II شامل ویژگی‌های وابسته به ماشین بود که باعث می‌شد کدهای نوشته شده در آن از دستگاه به دستگاه دیگر قابل حمل نباشد.

FORTRAN IV: شرکت IBM در این نسخه ویژگی‌های وابسته به ماشین را حذف کرد و ویژگی‌های جدیدی همچون نوع داده منطقی، عبارات منطقی بولی و عبارات منطقی IF اضافه کرد. سرانجام FORTRAN IV در سال ۱۹۶۲، ابتدا برای رایانه ("Stretch") IBM 7030 و سپس نسخه‌های IBM 7090، IBM 7094 و بعداً برای IBM 1401 در ۱۹۶۶ منتشر شد.

FORTRAN 66: شاید مهمترین پیشرفت در اوایل تاریخ FORTRAN ، تصمیم انجمن استاندارد آمریکا (ANSI) برای تشکیل کمیته‌ای تحت حمایت BEMA ، برای توسعه استاندارد آمریکایی Fortran باشد. دو استاندارد حاصل ، در مارس ۱۹۶۶ تصویب شد ، دو زبان FORTRAN (بر اساس FORTRAN IV که به عنوان یک استاندارد عمل می‌کرد) و Basic FORTRAN (بر اساس FORTRAN II ، اما از ویژگی‌های وابسته به ماشین آن محروم بودند) تعریف کرد. FORTRAN 66 در واقع به اولین نسخه استاندارد صنعت برای FORTRAN تبدیل شد.

FORTRAN 77: پس از انتشار استاندارد FORTRAN 66 ، فروشندگان کامپایلر چندین مورد را به Standard Fortran اضافه کردند و باعث شد کمیته ANSI X3J3 در سال ۱۹۶۹ تحت حمایت CBEMA ، (BEMA سابق) کار خود را در مورد اصلاح استاندارد ۱۹۶۶ آغاز کند. استاندارد جدید ، با نام FORTRAN 77 در آوریل ۱۹۷۸ منتشر شد. در این بازنگری در استاندارد ، تعدادی از ویژگی‌ها به گونه ای حذف شده یا تغییر داده شده‌اند که ممکن است برنامه‌های سازگار با استاندارد قبلی را بی‌اعتبار کند.

Fortran 90: جانشین بسیار با تأخیر FORTRAN 77 ، که به طور غیررسمی به عنوان Fortran 90 شناخته می‌شود (و قبل از آن Fortran 8X) ، سرانجام با استاندارد ISO / IEC 1539: 1991 در ۱۹۹۱ و ANSI Standard در ۱۹۹۲ منتشر شد. علاوه بر تغییر رسمی این نسخه اصلی از FORTRAN به Fortran تغییر یافت.

Fortran 95: به طور رسمی با عنوان ISO / IEC 1539-1: 1997 منتشر شد ، یک ویرایش جزئی بود ، بیشتر برای حل برخی از مشکلات برجسته از استاندارد Fortran 90.

Fortran 2003: به طور رسمی با عنوان ISO / IEC 1539-1: 2004 منتشر شده است ، یک نسخه اصلی است که بسیاری از ویژگی‌های جدید را ارائه می‌دهد.

Fortran 2008: به طور رسمی با عنوان ISO / IEC 1539-1: 2010 ، که به طور غیررسمی به عنوان Fortran 2008 شناخته می‌شود ، در سپتامبر ۲۰۱۰ تأیید شد. همانند Fortran 95 ، این یک ارتقا جزئی است که شامل توضیحات و اصلاحات Fortran 2003 و همچنین معرفی برخی از قابلیت‌های جدید است.

Fortran 2018: آخرین تجدید نظر در زبان (Fortran 2018) قبلاً به عنوان Fortran 2015 شناخته شده بود. این یک بازنگری قابل توجه است و در تاریخ ۲۸ نوامبر ۲۰۱۸ منتشر شد.

گونه‌های مختلف:

عرضه‌کنندگان کامپیوترهای علمی، افزونه‌های فرترن را اضافه کردند تا از قابلیت‌های سخت‌افزاری مانند حافظه نهان دستور، pipeline و آرایه‌های برداری بهره ببرند. برای مثال: یکی از کامپایلرهای فرترن شرکت آی‌بی‌ام، سطحی از بهینه‌سازی را داشت که ترتیب دستورها را عوض می‌کرد تا واحدهای محاسباتی داخلی مختلف را همزمان مشغول نگه دارد.

یک مثال دیگر CFD است، یک گونه‌ی خاص از فرترن که مخصوص ابرکامپیوتر ایلیاک ۴ (ILLIAC IV) ساخته شده است. این کامپیوتر در مرکز تحقیقات ایمز متعلق به ناسا قرار دارد.

نرم‌افزارهای مورد نیاز:

– Microsoft Visual Studio

– intel Fortran

ورژن نسخه intel Fortran باید بالاتر از Microsoft Visual Studio باشد.

Compiler OR interpreter ؟

Compiler و interpreter انواع مترجم زبان هستند. مترجم زبان، نرم‌افزاری است که برنامه‌ها را از یک زبان مبدا که به صورت قابل خواندن توسط انسان (معمولاً یک زبان برنامه‌نویسی سطح بالا) است به یک برنامه معادل در زبان ماشین ترجمه می‌کنند.

• Fortran یک زبان کامپایلری است.

کامپایلر، اصطلاحات زبان را برای صحت آن تجزیه و تحلیل می‌کند؛ اگر نادرست باشد، خطایی را وارد می‌کند. خطاها، به صورت همزمان و در انتهای برنامه نمایش داده می‌شوند. با این حال، کامپایلر فقط یک برنامه است و نمی‌توان خطاهای موجود در آن برنامه را برطرف کند؛ خطاهای برنامه را بدون مراجعه به کد منبع نمی‌توان تغییر داد.

کامپایلرهای زبان Fortran:

- Intel Fortran Compiler
- GNU Fortran
- GNU Compiler Collection
- MinGW
- G95
- Silverfrost FTN95
- Intel C++ Compiler
- Mingw-w64
- Absoft Fortran Compilers
- Digital Visual Fortran Programmer's Guide

Compiler Software/ Fortran/ Windows



: Syntax

Syntax برنامه اصلی به شرح زیر است:

```
program program_name
implicit none

! type declaration statements
! executable statements

end program program_name
```

مانند:

```
1 program addNumbers
2
3 ! This simple program adds two numbers
4 implicit none
5
6 ! Type declarations
7 real :: a, b, result
8
9 ! Executable statements
10 a = 12.0
11 b = 15.0
12 result = a + b
13 print *, 'The total is ', result
14
15 end program addNumbers
```

بررسی Syntax زبان Fortran :

- تمام برنامه‌های Fortran با **keyword program** شروع می‌شوند و در جلوی آن اسم برنامه قرار می‌گیرد و انتهای برنامه نیز با **keyword end program** و نام برنامه در جلوی آن به اتمام می‌رسد.
- در شروع هر برنامه‌ای باید از عبارت **implicit none** استفاده شود؛ این عبارت به کامپایلر اجازه می‌دهد تا بررسی کند که انواع متغیر به درستی اعلام شده است.
- نظرات (comment) با "!" شروع می‌شوند و همه کاراکترهای بعد از آن توسط کامپایلر نادیده گرفته می‌شود.
- دستور **print *** دستورات را بر روی صفحه نمایش چاپ می‌کند.
- Fortran به حروف کوچک و بزرگ حساس نیست.

: Basic

کاراکترهای اصلی زبان Fortran شامل:

- حروف A ... Z و a ... z
- ارقام ۰ ... ۹
- خط زیرین (_)
- کاراکترهای خاص: ? , < , > , & , % , ! , ' , \$, . , [,] , (,) , * , - , blank +

: Token

تشکیل شده از مجموعه‌ای از کاراکترهاست. یک Token می‌تواند یک keyword, identifier, constant, string literal, symbol باشد.

: Identifier

یک Identifier (شناسه) نامی است که برای شناسایی متغیر توسط کاربر تعریف می‌شود؛ باید ویژگی‌های زیر را داشته باشد:

- نمی‌تواند بیش از ۳۱ کاراکتر باشد.
- باید از حروف الفبا (تمام حروف الفبا و ارقام ۰ تا ۹) و زیر خط (_) تشکیل شده باشد.
- اولین کاراکتر باید یک حرف (letter) باشد.

: Operator

عملگر، نمادی است که به کامپایلر می‌گوید دستکاری‌های ریاضی یا منطقی خاصی را انجام دهد؛ انواع عملگر در Fortran:

- Arithmetic Operators : + , - , * (Mult) , / (Div) , ** (Power)
- Relational Operators : < , > , <= , >= , /= , ==
- Logical Operators : .and. / .or. / .not. / .eqv. / .neqv.

:Arithmetic Operators

عملیات ریاضی در فرترن :

اگر $A = 3$ و $B = 2$ باشد، داریم:

Operator	Example
+	$A + B = 5$
-	$A - B = 1$
*	$A * B = 6$
/	$A / B = 1.5$
**	$A ** B = 9$

:Relational Operators

بین متغیرهای عددی صورت میگیرد و شامل موارد زیر است:

- EQ. برابر است با "=="
- NE. برابر است با "!="
- LT. برابر است با "<"
- LE. برابر است با "<="
- GT. برابر است با ">"
- GE. برابر است با ">="

در جدول زیر با مثال عملکرد عملگرها نشان داده شده است:

A	B	A .EQ. B A == B	A .NE. B A != B	A .LT. B A < B	A .LE. B A <= B	A .GT. B A > B	A .GE. B A >= B
5	2	F	T	F	F	T	T
2	5	F	T	T	T	F	F
5	5	T	F	F	T	F	T

:Logical Operators

بین متغیرهای منطقی صورت می گیرد، شامل موارد زیر است:

- .AND. -
- .OR. -
- .XOR. -
- .EQV. -
- .NOT. -

عملکرد عملگرها در جدول زیر قابل مشاهده است:

A	B	A .AND. B	A .OR. B	A .XOR. B	A .EQV. B	.NOT. A
T	T	T	T	F	T	F
T	F	F	T	T	F	
F	T	F	T	T	F	T
F	F	F	F	F	T	

:Terminology

در C / C ++ / Java ، اصطلاحات **"reserved word"** و **"keyword"** معنی یکسانی دارند، اما :
در Fortran :

keyword کلمه‌ای است که فقط در زمینه‌های خاص استفاده می‌شود؛ مانند:

Real VarName -

reserved word، کلماتی هستند که نمی‌توان به عنوان نام متغیر توسط کاربر تعیین شوند؛ مانند:

Wile , for, do, break -

Keywords:

Keywords کلمات کلیدی هستند که مخصوص زبان هستند و از این Keywords ها نمی‌توان به عنوان نام متغیر یا شناسه استفاده کرد. Keywords های زبان Fortran :

The non-I/O keywords				
allocatable	allocate	assign	assignment	block data
call	case	character	common	complex
contains	continue	cycle	data	deallocate
default	do	double precision	else	else if
elsewhere	end block data	end do	end function	end if
end interface	end module	end program	end select	end subroutine
end type	end where	entry	equivalence	exit
external	function	go to	if	implicit
in	inout	integer	intent	interface
intrinsic	kind	len	logical	module
namelist	nullify	only	operator	optional
out	parameter	pause	pointer	private
program	public	real	recursive	result
return	save	select case	stop	subroutine
target	then	type	type()	use
Where	While			
The I/O related keywords				
backspace	close	endfile	format	inquire
open	print	read	rewind	Write

Constants:

ثابت‌ها به مقادیر ثابتی اشاره دارند که برنامه نمی‌تواند در حین اجرا آن‌ها را تغییر دهد. این مقادیر ثابت را literals نیز می‌نامند.

ثابت‌ها می‌توانند از هر یک از انواع داده‌های اصلی مانند یک integer constant ، floating constant ، character constant ، complex constant یا string literal باشد. فقط دو logical constants وجود دارد: true و false.

با ثابت‌ها دقیقاً مانند regular variables رفتار می‌شود ، با این تفاوت که مقادیر آن‌ها پس از تعریف قابل اصلاح نیستند.

دو نوع ثابت وجود دارد:

- Literal constants
- Named constants

! Literal constant دارای یک مقدار است اما هیچ نامی ندارد.

! Named constant دارای یک مقدار و همچنین یک نام است.

: ENUMERATOR type

Fortran2003 نوع ENUMERATOR را شبیه C enum تعریف می‌کند.

برخلاف C ، یک enumerator نمی‌تواند یک نوع تعریف شده خاص داشته باشد.

Syntax آن به شرح زیر است:

```
ENUM, BIND(C) :: COLOR
  ENUMERATOR :: RED = 4, BLUE = 9
  ENUMERATOR YELLOW
END ENUM

TYPE(COLOR) :: MYCOLOR = BLUE
```

برخلاف derived type ، این نوع scalar است و هیچ syntax انتخاب عضو (member-selector) ندارد.

:Data Types

Fortran پنج نوع داده ذاتی را ارائه می‌دهد:

- Integer type
- Real type
- Complex type
- Logical type
- Character type

: Integer type

فقط می‌توانند مقادیر عدد صحیح را نگه دارند؛ تابع `huge()` بیشترین عددی را که می‌تواند توسط نوع خاص داده صحیح نگهداری شود، می‌دهد.

: Real type

این اعداد شناور مانند `2.0`, `3.1415`, `-100.876` و غیره را ذخیره می‌کند؛ به طور سنتی دو نوع `real` مختلف وجود دارد: نوع پیش فرض `real` و نوع `double precision`.

: Complex type

این برای ذخیره اعداد مختلط استفاده می‌شود. یک عدد مختلط دارای دو قسمت است؛ دو واحد ذخیره عددی متوالی این دو قسمت را ذخیره می‌کنند.
به عنوان مثال، عدد مختلط `(3.0, -5.0)` برابر با `"3.0 - 5.0i"` است.

: Logical type

فقط دو مقدار منطقی وجود دارد: `true` و `false`.

: Character type

رشته‌ها و کاراکترها را ذخیره می‌کند. طول رشته را می‌توان با مشخص کننده `len` تعیین کرد. اگر طول مشخص نشده باشد، ۱ است.

انواع متغیر (Variable) :

متغیرها می‌توانند شامل حروف، اعداد و (_) باشند.

قوانین متغیرها در Fortran :

- متغیرها باید از حروف الفبا (تمام حروف الفبا و ارقام ۰ تا ۹) و زیر خط (_) تشکیل شده باشند.
- اولین کاراکتر یک متغیر باید یک حرف باشد.
- متغیرها نمی‌توانند بیش‌تر از ۳۱ کاراکتر داشته باشند.
- به حروف کوچک و بزرگ حساس نیستند.

اعلان ضمنی (implicit): به طور پیش فرض در Fortran، متغیرها و ثابت‌هایی که نام آنها با I - N شروع می‌شود از نوع INTEGER هستند و سایر متغیرها از نوع REAL هستند ولی بهتر است که هر متغیر و ثابت صریحاً اعلام شود.

اعلان صریح (explicit): Fortran، پنج نوع متغیر و ثابت را تشخیص می‌دهد؛ شامل:

نوع متغیر	توضیح به اختصار
Integer	برای مقادیر صحیح استفاده می‌شود. (متغیر عددی)
Real	برای اعداد اعشاری (float number) استفاده می‌شود. (متغیر عددی)
Complex	برای اعداد مختلط استفاده می‌شود.
Logical	برای مقادیر بولی استفاده می‌شود. (متغیر منطقی)
Character	برای استفاده از کاراکترها و رشته‌ها مورد استفاده قرار می‌گیرد. (متغیر کاراکتری)

Syntax for variable:

type-specifier :: variable_name

```
integer :: total
real    :: average
complex :: cx
logical :: done
character(len = 80) :: message ! a string of 80 characters
```

برای مثال:

```
Execute | > Share | main.f95 | STDIN
1 program variableTesting
2 implicit none
3
4 ! declaring variables
5 integer :: total
6 real :: average
7 complex :: cx
8 logical :: done
9 character(len=12) :: message ! a string of 12 characters
10
11 !assigning values
12 total = 40000
13 average = 145.60
14 done = .true.
15 message = "Hello World!"
16 cx = (1.35, 5.0) ! cx = 1.35 + 5.0i
17
18 Print *, total
19 Print *, average
20 Print *, cx
21 Print *, done
22 Print *, message
23
24 end program variableTesting
```

بعد از کامپایل کد بالا داریم:

```
Result
$gfortran -std=gnu *.f95 -o main
$main
40000
145.600006
(1.35000002,5.00000000)
T
Hello World!
```

دسته‌بندی متغیرها (Data Object):

– استاتیک:

قبل از شروع حافظه را به متغیر اختصاص می‌دهد و در طول اجرا حافظه در اختیار متغیر است و در FORTRAN 77 به این صورت بود.

- **مزیت:** کارایی بالا، ادرس دهی مستقیم، پشتیبانی از زیر برنامه حساس به تاریخ.
- **عیب:** عدم انعطاف‌پذیری (بدون بازگشت)، نمی‌توان حافظه را به صورت اشتراکی برای متغیر استفاده کرد.

- پویا:

در زمان ترجمه (کامپایل) میزان حافظه اختصاص داده شده به آن مشخص نیست و در طول زمان اجرا حافظه تخصیص داده می‌شود؛ مانند:

آرایه پویا (dynamic array) : برای استفاده از آرایه‌ها باید ابعاد آرایه ذکر شود، با این وجود برای تخصیص حافظه به چنین آرایه‌ای از تابع allocate استفاده می‌شود.

```
real, dimension (:,:), allocatable :: darray
```

```
allocate ( darray(s1,s2) )
```

پس از استفاده از آرایه ، در برنامه ، حافظه ایجاد شده باید با استفاده از تابع deallocate آزاد شود.

```
deallocate (darray)
```

! در ادامه به‌طور کامل به توضیح آرایه‌ها در زبان Fortran می‌پردازیم.

نمونه کد:

The following example demonstrates the concepts discussed above.

```
program dynamic_array
implicit none

!rank is 2, but size not known
real, dimension (:,:), allocatable :: darray
integer :: s1, s2
integer :: i, j

print*, "Enter the size of the array:"
read*, s1, s2

! allocate memory
allocate ( darray(s1,s2) )

do i = 1, s1
  do j = 1, s2
    darray(i,j) = i*j
    print*, "darray(",i,",",j,") = ", darray(i,j)
  end do
end do

deallocate (darray)
end program dynamic_array
```

نتیجه کد بالا پس از کامپایل:

When the above code is compiled and executed, it produces the following result –

```
Enter the size of the array: 3,4
darray( 1 , 1 ) = 1.00000000
darray( 1 , 2 ) = 2.00000000
darray( 1 , 3 ) = 3.00000000
darray( 1 , 4 ) = 4.00000000
darray( 2 , 1 ) = 2.00000000
darray( 2 , 2 ) = 4.00000000
darray( 2 , 3 ) = 6.00000000
darray( 2 , 4 ) = 8.00000000
darray( 3 , 1 ) = 3.00000000
darray( 3 , 2 ) = 6.00000000
darray( 3 , 3 ) = 9.00000000
darray( 3 , 4 ) = 12.00000000
```

عملیات در Fortran :

- انتساب:

FORTRAN از "=" به عنوان عملگر انتساب استفاده می کند ، به عنوان مثال : $x = 6$.

- الحاق دو رشته:

از عملگر "///" برای پیوستن دو رشته به یک رشته استفاده می شود، مانند:

```
j = ' Join these toget'///'her'
```

: Characters

کاراکترها می‌توانند هر نمادی باشند که از مجموعه کاراکترهای اصلی گرفته شده‌اند ، یعنی از حروف ، رقم اعشار ، زیر خط و ۲۱ کاراکتر خاص.

اعلام کاراکتر:

```
Type-specifier :: variable_name
```

مثل:

```
Character :: pl , atousa
```

برخی از تابع‌های کاراکتر:

Function and Description	
len(string) طول یک رشته کاراکتر را برمی‌گرداند.	1
index(string,sustring) محل sustring را در string دیگری پیدا می‌کند ، اگر پیدا نشود "0" را برمی‌گرداند.	2
achar(int) این یک عدد صحیح را به یک کاراکتر تبدیل می‌کند.	3
iachar(c) این یک کاراکتر را به یک عدد صحیح تبدیل می‌کند.	4
trim(string) رشته را در حالی که قسمت‌های خالی عقب برداشته شده برمی‌گرداند.	5
scan(string, chars) "رشته" را از چپ به راست جستجو می‌کند. این یک عدد صحیح را بازمی‌گرداند که موقعیت آن کاراکتر را می‌دهد ، یا اگر هیچ یک از شخصیت های "chars" پیدا نشده باشد ، صفر است.	6
verify(string, chars) مانند بالایی با این تفاوت که از راست به چپ جستجو می‌کند.	7
len_trim(string) یک عدد صحیح برمی‌گرداند که برابر است با طول رشته منهای فضاهای خالی.	8

انقیاد (Binding) :

وابسته بودن یا ارتباط داشتن یک چیز با چیزهای دیگر است.

زمان انقیاد در زبان Fortran :

انقیاد تنها در **زمان ترجمه** (هنگام کامپایل) صورت می گیرد؛ که در این صورت یک سری معایب و مزایا دارد.

- **مزیت:** کارایی بسیار بالایی دارد .

- **عیب:** انعطاف پذیری کاهش می یابد.

در دسته بندی انقیاد در زبان ترجمه، Fortran در دسته "**توسط برنامه نویس**" قرار می گیرد؛ زیرا برنامه نویس باید نوع (Type) متغیرها را مشخص کند.

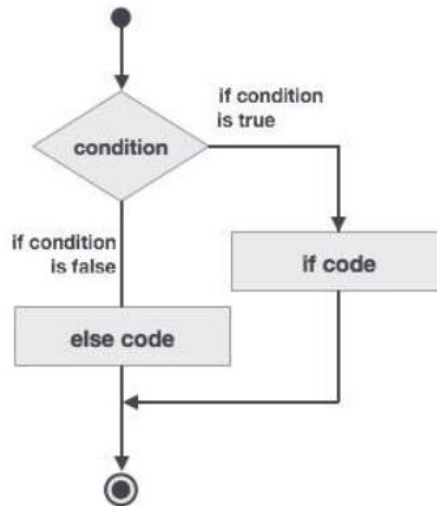
انواع انقیاد:

نوع انقیاد زبان Fortran **زودرس (early)** است؛ زیرا انقیاد در زمان ترجمه صورت می پذیرد.

ساختارهای تصمیم‌گیری:

۱. ساختار IF :

دیگرام if به صورت زیر است:



: Syntax

```
if (logical expression) then
    statement(s)
else
    other_statement(s)
end if
```

مثال:

```
1 program ifElseTest
2 implicit none
3
4     ! local variable declaration
5     integer :: a = 18
6
7     ! check the logical condition using if statement
8     if ( a > 10 ) then
9
10        ! if condition is true then print the following
11        print*, "You Passed!"
12
13    else
14
15        ! if none of the conditions is true
16        print*, "You failed!"
17
18    end if
19
20
21 end program ifElseTest
```

بررسی ساختار if:

Program

```
If ( a > 10) then  
    Print* , "You Passed!"  
Else  
    Print* , "You Failed!"
```

End

گرامر:

<Example> -> <program> S <end>

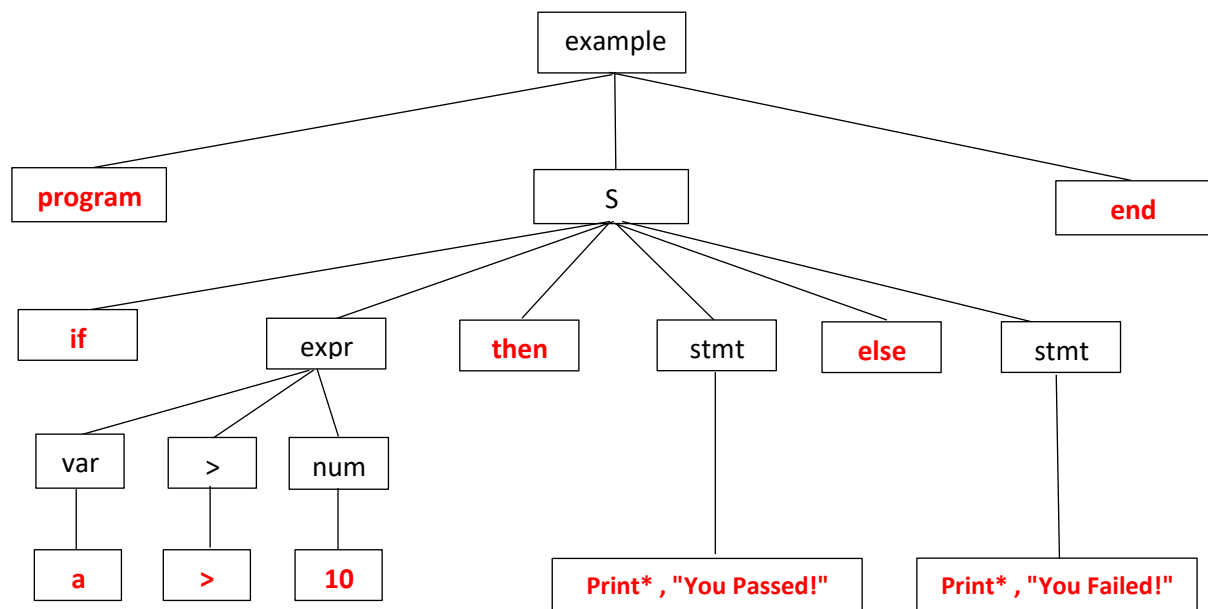
<S> -> if <expr> then <stmt> else <stmt>

<expr> -> <var> > <num>

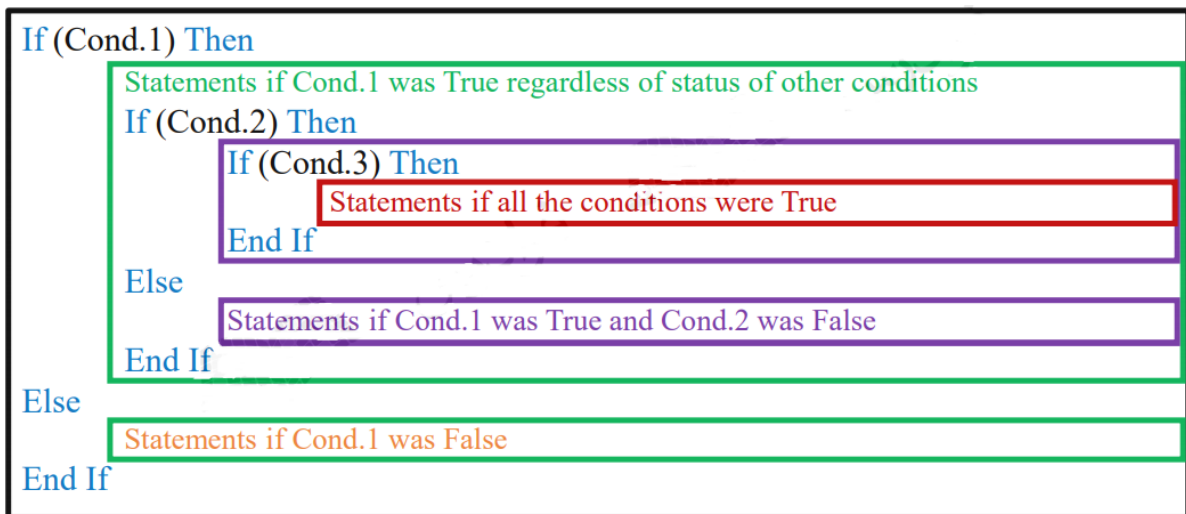
<num> -> 10

<var> -> a

<stmt> -> Print* , "You Passed!" | Print* , "You Failed!"



II. ساختار if های تو در تو:



ابتدا شرط if اولی مورد بررسی قرار می گیرد، اگر شرط برقرار بود وارد if دوم می شود و اگر شرط برقرار نبود، دستور else را اجرا می کند. به این ترتیب تمام دستورها اجرا می شوند.

مثال:

```
1 Program elseifex
2 implicit none
3
4 INTEGER :: x
5 CHARACTER(LEN=1) :: Grade
6
7 x = 67
8
9 IF (x < 50) THEN
10   write(*,*) "Grade = 'F'"
11 ELSE
12   if (x < 60) then
13     write (*,*) "Grade = 'D'"
14   ELSE
15     if (x < 70) then
16       write(*,*) "Grade = 'C'"
17     ELSE
18       if (x < 80) then
19         write(*,*) "Grade = 'B'"
20       ELSE
21         write(*,*) "Grade = 'A'"
22       end if
23     end if
24   end if
25 end if
26
27 End Program elseifex
28
```

III. ساختار if else :

```
if (Cound.1) then
    Statement1
ElseIf (Cound.2) then
    Statement2
ElseIf (Cound.3) then
    ...
Else
    Statement if all the condition wew False
End If
```

مثال:

```
1 Program elseifex
2 implicit none
3
4 INTEGER :: x
5 CHARACTER(LEN=1) :: Grade
6
7 x = 67
8
9 IF (x < 50) THEN
10     write(*,*) "Grade = 'F'"
11 ELSE IF (x < 60) THEN
12     write(*,*) "Grade = 'D'"
13 ELSE IF (x < 70) THEN
14     write(*,*) "Grade = 'C'"
15 ELSE IF (x < 80) THEN
16     write(*,*) "Grade = 'B'"
17 ELSE
18     write(*,*) "Grade = 'A'"
19 END IF
20
21 End Program elseifex
22
```

خروجی هر دو کد بالا (مثال‌های Elseif و IF های تو در تو) یکسان است اما نوشتن به صورت Elseif ساده‌تر و امکان اشتباه کمتری در نوشتن کد وجود دارد.

```
Grade = 'C'
```

IV. ساختار Select Case:

یک Select Case اجازه می‌دهد تا یک متغیر برای برابری در برابر لیستی از مقادیر آزمایش شود. به هر مقدار یک Case گفته می‌شود و متغیر انتخاب شده برای هر Case انتخاب شده بررسی می‌شود.

```
Select Case (VariableName)
  Case(Cound.1)
    Statement1
  Case(Cound.2)
    Statement2
  ...
  Case Default
    Statements if all the conditions were False
End Select
```

مثال:

```
1 Program week
2 implicit none
3 integer :: day
4
5 write(*,*) "Enter a number between 1 and 7:|"
6 read(*,*) day
7
8 select case (day)
9   case(1)
10    write(*,*) "Sunday"
11   case(2)
12    write(*,*) "Monday"
13   case(3)
14    write(*,*) "Tuesday"
15   case(4)
16    write(*,*) "Wednesday"
17   case(5)
18    write(*,*) "Thursday"
19   case(6)
20    write(*,*) "Friday"
21   case(7)
22    write(*,*) "Saturday"
23 end select
24
25 end Program week
26
```

پس از کامپایل کد بالا:

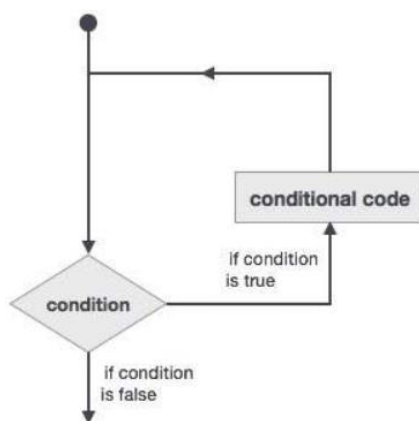
```
Enter a number between 1 and 7:
3
Tuesday
```

:LOOP

ممکن است شرایطی پیش بیاید که نیاز به اجرای چندین بلاک از کد باشد. به طور کلی ، دستورات به صورت پی در پی اجرا می شوند: اولین دستور در یک تابع ابتدا اجرا می شود ، پس از آن جمله دوم و غیره اجرا می شود.

یک دستور حلقه (LOOP) به ما امکان می دهد تا یک دستور یا گروهی از دستورات را چندین بار اجرا کنیم.

ساختار حلقه ها:



i. حلقه Do

ساختار حلقه do ، یک دستور یا یک سری دستورات را قادر می سازد تا به صورت تکراری انجام شوند ، در حالی که یک شرط مشخص درست است.

```
do var = start, stop [,step]
  ! statement(s)
  ...
end do
```

- متغیر loop var باید یک عدد صحیح باشد.
- start مقدار اولیه است.
- stop مقدار نهایی است.
- [,stop] ، اگر این مورد حذف شود ، متغیر var با واحد افزایش می یابد.

مثال:

```
1 program factorial
2 implicit none
3
4     ! define variables
5     integer :: nfact = 1
6     integer :: n
7
8     ! compute factorials
9     do n = 1, 10
10        nfact = nfact * n
11        ! print values
12        print*, n, " ", nfact
13    end do
14
15 end program factorial
```

خروجی کد بالا:

1	1
2	2
3	6
4	24
5	120
6	720
7	5040
8	40320
9	362880
10	3628800

ii. حلقه Do While:

این یک عبارت یا یک گروه از عبارات را تکرار می‌کند در حالی که یک شرط مشخص درست است. این شرایط را قبل از اجرای بدنه حلقه آزمایش می‌کند.

Do While (Condition)

[Statements]

End Do

مثال:

```
1 program factorial
2 implicit none
3
4     ! define variables
5     integer :: nfact = 1
6     integer :: n = 1
7
8     ! compute factorials
9     do while (n <= 10)
10        nfact = nfact * n
11        n = n + 1
12        print*, n, " ", nfact
13    end do
14 end program factorial
```

خروجی کد بالا همان خروجی کد قبل است. (فاکتوریل اعداد ۱ تا ۱۰ را چاپ می‌کند.)

iii. Loop های تو در تو:

می‌توان از یک یا چند Loop در داخل ساختار Loop دیگری استفاده کرد. همچنین می‌توان labels را روی حلقه‌ها قرار دهید.

مثال:

```
1 program nestedLoop
2 implicit none
3
4     integer:: i, j, k
5
6     iloop: do i = 1, 3
7         jloop: do j = 1, 3
8             kloop: do k = 1, 3
9
10                print*, "(i, j, k): ", i, j, k
11            end do kloop
12        end do jloop
13    end do iloop
14
15
16 end program nestedLoop
```

خروجی کد بالا پس از کامپایل:

(i, j, k):	1	1	1
(i, j, k):	1	1	2
(i, j, k):	1	1	3
(i, j, k):	1	2	1
(i, j, k):	1	2	2
(i, j, k):	1	2	3
(i, j, k):	1	3	1
(i, j, k):	1	3	2
(i, j, k):	1	3	3
(i, j, k):	2	1	1
(i, j, k):	2	1	2
(i, j, k):	2	1	3
(i, j, k):	2	2	1
(i, j, k):	2	2	2
(i, j, k):	2	2	3
(i, j, k):	2	3	1
(i, j, k):	2	3	2
(i, j, k):	2	3	3
(i, j, k):	3	1	1
(i, j, k):	3	1	2
(i, j, k):	3	1	3
(i, j, k):	3	2	1
(i, j, k):	3	2	2
(i, j, k):	3	2	3
(i, j, k):	3	3	1
(i, j, k):	3	3	2
(i, j, k):	3	3	3

iv. دستورات کنترلی Loop:

code is compiled	Example	Control Statement
<pre>(i, j, k): 1 1 1 (i, j, k): 1 1 2 (i, j, k): 2 1 1 (i, j, k): 2 1 2 (i, j, k): 3 1 1 (i, j, k): 3 1 2</pre>	<pre>program nestedLoop implicit none integer :: i, j, k iloop: do i = 1, 3 jloop: do j = 1, 3 kloop: do k = 1, 3 print*, "(i, j, k): ", i, j, k if (k==2) then exit jloop end if end do kloop end do jloop end do iloop end program nestedLoop</pre>	<p>Exit</p> <p>از حلقه خارج شده و اجرای برنامه در اولین عبارت اجرایی پس از شرط پایان انجام می‌شود.</p>
<pre>1 2 3 4 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20</pre>	<pre>program cycle_example implicit none integer :: i do i = 1, 20 if (i == 5) then cycle end if print*, i end do end program cycle_example</pre>	<p>Cycle</p> <p>برنامه در آغاز تکرار بعدی ادامه می‌یابد.</p>
<pre>1 2 3 4</pre>	<pre>program stop_example implicit none integer :: i do i = 1, 20 if (i == 5) then stop end if print*, i end do end program stop_example</pre>	<p>Stop</p> <p>اگر می‌خواهید اجرای برنامه متوقف شود ، می‌توانید یک عبارت Stop وارد کنید.</p>

: pointer in Fortran

در بیشتر زبان‌های برنامه نویسی، یک متغیر اشاره‌گر آدرس حافظه یک شی را ذخیره می‌کند. با این حال، در Fortran، یک اشاره‌گر یک شی داده است که ویژگی‌های آن بیشتر از ذخیره آدرس حافظه است. این شامل اطلاعات بیشتری در مورد یک شی خاص است، مانند type، rank، extents و memory address.

اعلام متغیر اشاره‌گر:

یک متغیر اشاره‌گر با ویژگی‌های زیر اعلام می‌شود. مانند:

```
integer, pointer :: p1 ! pointer to integer
real, pointer, dimension (:) :: pra ! pointer to 1-dim real array
real, pointer, dimension (:,:) :: pra2 ! pointer to 2-dim real array
```

یک اشاره‌گر می‌تواند به موارد زیر اشاره کند:

- منطقه ای از حافظه اختصاص یافته به صورت پویا.
- یک شی داده از همان نوع اشاره‌گر، با ویژگی هدف.

یک اشاره‌گر می‌تواند:

- Undefined
- Associated
- Disassociated

اختصاص دادن فضا برای یک اشاره‌گر:

Allocate به شما امکان می‌دهد برای یک شی اشاره‌گر فضا اختصاص دهید. به عنوان مثال:

```
program pointerExample
implicit none
  integer, pointer :: p1
  allocate(p1)

  p1 = 1
  Print *, p1

  p1 = p1 + 4
  Print *, p1
end program pointerExample
```

پس از اجرای کد بالا داریم:

```
1
5
```

می‌توان فضای ذخیره سازی اختصاص داده شده توسط اشاره گر را توسط `deallocate` را خالی کرد؛ هنگامی که دیگر نیازی به آن نیست و از تجمع فضای حافظه استفاده نشده و غیر قابل استفاده جلوگیری کرد.

Targets

`Targets` متغیر طبیعی دیگر است که فضای کافی برای آن در نظر گرفته شده است. یک `Targets` هدف باید با صفت (`attribute`) هدف اعلام شود.

می‌توان با استفاده از عملگر ارتباط (`=>`) ، یک متغیر `pointer` را با متغیر `Targets` مرتبط کرد.

مانند قطعه کد زیر:

```
Execute | > Share | main.f95 | STDIN
1 program pointerExample
2 implicit none
3
4     integer, pointer :: p1
5     integer, target :: t1
6     integer, target :: t2
7
8     p1=>t1
9     p1 = 1
10
11    Print *, p1
12    Print *, t1
13
14    p1 = p1 + 4
15
16    Print *, p1
17    Print *, t1
18
19    t1 = 8
20
21    Print *, p1
22    Print *, t1
23
24    nullify(p1)
25    Print *, t1
26
27    p1=>t2
28    Print *, associated(p1)
29    Print*, associated(p1, t1)
30    Print*, associated(p1, t2)
31
32    !what is the value of p1 at present
33    Print *, p1
34    Print *, t2
35
36 end program pointerExample
```

خروجی کد بالا :

```
Result
$gfortran -std=gnu *.f95 -o main
$main
      1
      1
      5
      5
      8
      8
      8
T
F
T
      0
      0
```

در برنامه فوق ، با استفاده از عملگر (=) نشانگر p1 با هدف t1 مرتبط شده است.

دستور `nullify` یک اشاره گر را از یک هدف (target) جدا می کند.

Nullify اهداف را خالی نمی‌کند زیرا ممکن است بیش از یک نشانگر به همان هدف وجود داشته باشد. با این حال، خالی کردن اشاره‌گر به معنای لغو نیز است.

فایل‌ها در Fortran :

Fortran این امکان را می‌دهد که داده‌ها از روی فایل‌ها خوانده شوند و در فایل‌ها نوشته شوند.

می‌توان روی یک یا چند فایل عمل خواندن و نوشتن صورت گیرد، توسط دستورات WRITE, READ و باز کردن و بستن فایل‌ها از طریق OPEN, CLOSE صورت می‌گیرد.

باز کردن و بستن فایل‌ها:

قبل از استفاده از فایل باید فایل را باز شود. از دستور open برای باز کردن فایل‌ها برای خواندن یا نوشتن استفاده می‌شود. ساده‌ترین فرم دستور به شکل زیر است:

```
Open (unit = number, file="name").
```

یک فرم کلی‌تر نیز دارد:

```
Open (list-of-specifiers)
```

در جدول زیر مشخصه‌های بیشتری مورد بررسی قرار گرفته است:

Specifier & Description	
[UNIT=] u شماره واحد u می‌تواند هر عددی در محدوده ۹-۹۹ باشد و این نشان‌دهنده فایل است، می‌توان هر شماره‌ای در این بازه انتخاب شود اما هر فایل باز شده در برنامه باید یک شماره منحصر به فرد داشته باشد.	1
IOSTAT= ios این شناسه وضعیت I / O است و باید یک متغیر صحیح باشد. اگر دستور open موفقیت‌آمیز بود، مقدار ios برگشتی صفر است و در غیراین صورت مخالف صفر است.	2
ERR = err این برجسبی است که در صورت بروز هرگونه خطا، کنترل به آن می‌پرد.	3
FILE = fname نام فایل، یک رشته کاراکتر است.	4
STATUS = sta این وضعیت قبلی فایل را نشان می‌دهد. یک رشته کاراکتر و می‌تواند یکی از سه مقدار OLD، NEW یا SCRATCH را داشته باشد. با ساخته شدن و حذف یک فایل SCRATCH در آن هنگام برنامه بسته می‌شود یا خاتمه می‌یابد.	5
ACCESS = acc این حالت دسترسی به فایل است. می‌تواند از هر دو مقدار SEQUENTIAL یا DIRECT برخوردار باشد. پیش فرض SEQUENTIAL است.	6
FORM = frm وضعیت قالب بندی فایل را نشان می‌دهد. می‌تواند از هر دو مقدار FORMATTED یا UNFORMATTED برخوردار باشد. پیش فرض UNFORMATTED است.	7
RECL = rl طول هر رکورد را در یک فایل دسترسی مستقیم مشخص می‌کند.	8

پس از باز شدن فایل، با استفاده از دستورات خواندن و نوشتن می‌توان به آن دسترسی پیدا کرد. پس از اتمام، باید با استفاده از دستور `close` ، فایل بسته شود.

دستور `close` به شرح زیر است:

```
Close ([UNIT = ]u[,IOSTAT = IOS,ERR = err,STATUS = sta])
```

! پارامترهای موجود در براکت‌ها اختیاری است.

مثال:

```
program outputdata
implicit none

real, dimension(100) :: x, y
real, dimension(100) :: p, q
integer :: i

! data
do i=1,100
    x(i) = i * 0.1
    y(i) = sin(x(i)) * (1-cos(x(i)/3.0))
end do

! output data into a file
open(1, file = 'data1.dat', status = 'new')
do i=1,100
    write(1,*) x(i), y(i)
end do

close(1)

end program outputdata
```

در قطعه کد بالا ، یک فایل به نام `data1.dat` ایجاد می‌شود و مقادیر آرایه‌های `x` و `y` را در آن می‌نویسد و سپس فایل را می‌بندد.

خواندن و نوشتن در فایل‌ها:

```
read ([UNIT = ]u, [FMT = ]fmt, IOSTAT = ios, ERR = err, END = s)
write([UNIT = ]u, [FMT = ]fmt, IOSTAT = ios, ERR = err, END = s)
```

مشخص کننده `END = s` یک برچسب دستور است که وقتی برنامه به انتهای فایل می‌رسد ، می‌پرد.

مثال:

```
1 program outputdata
2 implicit none
3
4 real, dimension(100) :: x, y
5 real, dimension(100) :: p, q
6 integer :: i
7
8 ! data
9 do i=1,100
10 | x(i) = i * 0.1
11 | y(i) = sin(x(i)) * (1-cos(x(i)/3.0))
12 end do
13
14 ! output data into a file
15 open(1, file = 'data1.dat', status='new')
16 do i=1,100
17 | write(1,*) x(i), y(i)
18 end do
19 close(1)
20
21 ! opening the file for reading
22 open (2, file = 'data1.dat', status = 'old')
23
24 do i = 1,100
25 | read(2,*) p(i), q(i)
26 end do
27
28 close(2)
29
30 do i = 1,100
31 | write(*,*) p(i), q(i)
32 end do
33
34 end program outputdata
```

در قطعه کد بالا، فایلی که در مثال قبلی ایجاد شده بود را باز شده و سپس عمل خواندن و نوشتن را در آن انجام شده است و در انتها فایل بسته شده است.

پس از کامپایل کد بالا داریم: (نمونه ایی از خروجی)

```
Result
$gfortran -std=gnu *.f95 -o main
$main
0.100000001      5.54589933E-05
0.200000003      4.41325130E-04
0.300000012      1.47636665E-03
0.400000006      3.45637114E-03
0.500000000      6.64328877E-03
0.600000024      1.12552457E-02
0.699999988      1.74576249E-02
0.800000012      2.53552198E-02
0.900000036      3.49861123E-02
1.000000000      4.63171192E-02
1.100000002      5.92407584E-02
1.200000005      7.35742524E-02
1.300000007      8.90605897E-02
1.399999998      0.105371222
1.500000000      0.122110792
1.600000002      0.138823599
1.700000005      0.155002132
1.800000007      0.170096487
1.899999998      0.183526158
2.000000000      0.194692180
2.100000014      0.202990443
2.200000005      0.207826138
2.299999995      0.208628103
2.400000010      0.204863429
2.500000000      0.196052119
2.600000014      0.181780845
2.700000005      0.161716297
2.799999995      0.135617107
2.900000010      0.103344671
3.000000000      6.48725405E-02
```

وراثت:

انواع داده مشتق شده (Derived data type):

در زبان‌های شی‌گرا (object-oriented) مانند C++، می‌توان کلاس‌هایی را تعریف کرد که هم شامل داده باشند و هم روش‌هایی که روی آن داده کار می‌کنند. سپس می‌توان نمونه‌های جداگانه‌ای از کلاس را ایجاد کرد که هرکدام داده‌های خاص خود را دارند. متدی که از یک نمونه از کلاس فراخوانی می‌شود، بر روی داده‌های نگهداری شده توسط آن نمونه خاص کار خواهد کرد.

Fortran این امکان را می‌دهد که انواع داده‌های مشتق شده را تعریف کنید. نوع داده مشتق شده، structure نیز نامیده می‌شود و می‌تواند شامل data objects از انواع مختلف تشکیل شده باشد؛ انواع داده مشتق شده برای نشان دادن یک رکورد استفاده می‌شود.

تعریف نوع داده مشتق شده:

برای تعریف نوع داده مشتق شده، از دستورات type و end type استفاده می‌شود. عبارت type نوع جدیدی را تعریف می‌کند که بیش از یک عضو برای برنامه دارد. قالب عبارت type به این صورت است:

```
type type_name
  declarations
end type
```

به عنوان مثال: می‌خواهید کتاب‌های خود را در کتابخانه پیگیری کنید، شاید بخواهید مشخصات زیر را در مورد هر کتاب ردیابی کنید:

```
type Books
  character(len = 50) :: title
  character(len = 50) :: author
  character(len = 150) :: subject
  integer :: book_id
end type Books
```

دسترسی به اعضای structure:

یک شی از نوع داده مشتق شده را structure می گویند.

```
type(Books) :: book1
```

میتوان از طریق (%) ، دسترسی پیدا کرد به components های structure ، مانند:

```
book1%title = "C Programming"
book1%author = "Nuha Ali"
book1%subject = "C Programming Tutorial"
book1%book_id = 6495407
```

توجه !: قبل و بعد "%" فاصله ای وجود ندارد.

مثال:

```
1 program deriveDataType
2
3 !type declaration
4 type Books
5     character(len = 50) :: title
6     character(len = 50) :: author
7     character(len = 150) :: subject
8     integer :: book_id
9 end type Books
10
11 !declaring type variables
12 type(Books) :: book1
13 type(Books) :: book2
14
15 !accessing the components of the structure
16
17 book1%title = "C Programming"
18 book1%author = "Nuha Ali"
19 book1%subject = "C Programming Tutorial"
20 book1%book_id = 6495407
21
22 book2%title = "Telecom Billing"
23 book2%author = "Zara Ali"
24 book2%subject = "Telecom Billing Tutorial"
25 book2%book_id = 6495700
26
27 !display book info
28
29 Print *, book1%title
30 Print *, book1%author
31 Print *, book1%subject
32 Print *, book1%book_id
33
34 Print *, book2%title
35 Print *, book2%author
36 Print *, book2%subject
37 Print *, book2%book_id
38
39 end program deriveDataType
```


خروجی قطعه کد بالا:

```
Result
$gfortran -std=gnu *.f95 -o main
$main
C Programming
Nuha Ali
C Programming Tutorial
6495407
Telecom Billing
Zara Ali
Telecom Billing Tutorial
6495700
```

:Array of Structures

همچنین می‌توان آرایه‌هایی از نوع مشتق شده (arrays of a derived type) ایجاد کرد، مانند:

```
type(Books), dimension(2) :: list
```

:Record

ساختار record در نسخه‌های قبل از Fortran95/90 بیشتر رایج بود ولی بعد از آن عملکرد آن با داده‌های استاندارد مشتق شده (derived type) جایگزین شده است.

رکوردها را می‌توان به راحتی برای قابلیت جابه‌جایی به ساختارهای نوع داده مشتق شده تبدیل کرد؛ اما می‌توان به همان شکل گذشته نیز مورد استفاده قرار گیرند؛ در بیشتر موارد می‌توان از Fortran record و Fortran 95/90 derived type به جای هم استفاده کرد.

ساختار رکورد یک موجودیت کل است که شامل یک یا چند عنصر است؛ به عناصر رکورد components نیز می‌گویند.

ایجاد رکورد یک فرآیند دو مرحله ای است:

- باید فرم رکورد را با یک اعلامیه ساختار چند طبقه تعریف کرد.
- برای اعلام سابقه به عنوان موجودی با نام ، باید از دستور RECORD استفاده کرد. (بیش از یک عبارت RECORD می تواند به یک ساختار معین اشاره داشته باشد).

: ساختار Record

```
STRUCTURE /item/  
  Integer  num  
  Character(len=200)  message  
  Real     price  
END STRUCTURE  
  
! Define two variables, an single record of type "item".  
! named "pear", and an array of items named "store_catalog".  
RECORD /item/ pear, store_catalog(100)
```

برای تبدیل record به derived type:

- به جای `STRUCTURE /structure-name` از `TYPE type-name` استفاده شود.
- برای دسترسی به components، علامت `(%)` جایگزین علامت `(.)` شود.

Record structure	Fortran 95/90 derived type
<pre> STRUCTURE /item/ integer id character(len=200) description real price END STRUCTURE ! Define two variables, an single record of type "item" ! named "pear", and an array of items name "store_catalog" RECORD /item/ pear, store_catalog(100) ! components pear.id = 92316 pear.description = "juicy D'Anjou pear" pear.price = 0.15 store_catalog(7).id = 7831 store_catalog(7).description = "milk bottle" store_catalog(7).price = 1.2 ! We can also manipulate the whole structure store_catalog(12) = pear print *, store_catalog(12) </pre>	<pre> TYPE item integer :: id character(len=200)::description real :: price END TYPE ! "RECORD /name/ variable" becomes "TYPE(name)variable" TYPE(item) pear, store_catalog(100) ! components pear%id = 92316 pear%description = "juicy D'Anjou pear" pear%price = 0.15 store_catalog(7)%id = 7831 store_catalog(7)%description = "milk bottle" store_catalog(7)%price = 1.2 ! Assignments of a whole variable do not change store_catalog(12) = pear print *, store_catalog(12) </pre>

:Modul

ماژول مانند بسته‌ای است که در آن می‌توان عملکردها و زیر برنامه‌ها را حفظ کرد، در صورتی که یک برنامه بسیار بزرگ باشد، از توابع یا زیر برنامه‌های آن می‌توان در بیش از یک برنامه استفاده کرد؛ ماژول‌ها راهی برای تقسیم برنامه‌ها بین چندین فایل ارائه می‌دهند.

ماژول‌ها برای:

- زیر برنامه‌ها (subprograms) ، بلوک‌های داده و رابط بسته بندی (interface blocks).
- تعریف داده‌های global که بیش از یک روال می‌تواند از آنها استفاده کند.
- اعلام متغیرهایی که می‌توانند در هر روال انتخابی در دسترس قرار بگیرند.
- وارد کردن ماژول به طور کامل ، برای استفاده ، به برنامه یا زیر برنامه دیگری.

:Syntax of a Module

از دو قسمت تشکیل شده است که با دستور CONTAINS از هم جدا می‌شوند:

- بالای عبارت contains ، جایی است که داده‌ها و متغیرهای global تعریف می‌شوند و قابلیت دسترسی به داده‌ها و رویه‌ها تعیین می‌شود.
- عبارات زیر contains ، جایی است که متدها، خصوصیات و سازنده‌های اجرای کد در آن قرار می‌گیرند.

```
module name
  [statement declarations]
  [contains [subroutine and function definitions] ]
end module [name]
```

استفاده از ماژول در برنامه خود:

با استفاده از دستور Use می‌توان یک ماژول را در یک برنامه یا زیر برنامه خود قرار دهید.

Use name

نقش ماژول:

ماژول از نسخه Fortran90 وارد Fortran شد .

ماژول‌ها ویژگی‌های Abstraction و encapsulation را پوشش می‌دهند.

چند نکته در رابطه با ماژول‌ها:

- میتوان به تعداد مورد نیاز ماژول اضافه کرد ، هر یک در فایل‌های جداگانه قرار می‌گیرند و به طور جداگانه وارد می‌شوند.
- از یک ماژول می‌توان در برنامه‌های مختلف استفاده کرد.
- از یک ماژول می‌توان بارها در همان برنامه استفاده کرد.
- متغیرهای اعلام شده در یک قسمت مشخصات ماژول ، برای ماژول `global` هستند.

مثال:

```
1 module constants
2 implicit none
3
4     real, parameter :: pi = 3.1415926536
5     real, parameter :: e = 2.7182818285
6
7 contains
8     subroutine show_consts()
9         print*, "Pi = ", pi
10        print*, "e = ", e
11    end subroutine show_consts
12
13 end module constants
14
15
16 program module_example
17 use constants
18 implicit none
19
20     real :: x, ePowerx, area, radius
21     x = 2.0
22     radius = 7.0
23     ePowerx = e ** x
24     area = pi * radius**2
25
26     call show_consts()
27
28     print*, "e raised to the power of 2.0 = ", ePowerx
29     print*, "Area of a circle with radius 7.0 = ", area
30
31 end program module_example
```

! توجه شود که ماژول‌ها قبل از برنامه اصلی نوشته می‌شوند.

خروجی برنامه بالا پس از کامپایل:

```
Pi = 3.14159274
e = 2.71828175
e raised to the power of 2.0 = 7.38905573
Area of a circle with radius 7.0 = 153.938049
```

در دسترس بودن متغیرها و زیر برنامه‌ها در یک ماژول:

به طور پیش فرض ، تمام متغیرها و زیرروال‌ها در یک ماژول با استفاده از دستور `use` در دسترس برنامه‌ای قرار می‌گیرد که از کد ماژول استفاده می‌کند.

با این حال، می‌توان قابلیت دسترسی به کد ماژول را با استفاده از خصوصیات `private` و `public` کنترل کرد. وقتی برخی متغیرها یا زیرروال‌ها را `private` اعلام می‌شوند، خارج از ماژول در دسترس نیستند.

مثال در مورد **دسترسی خصوصی** متغیرها در ماژول:

```
1 module constants
2   implicit none
3
4   real, parameter,private :: pi = 3.1415926536
5   real, parameter,private :: e = 2.7182818285
6
7 contains
8   subroutine show_consts()
9     print*, "Pi = ", pi
10    print*, "e = ", e
11  end subroutine show_consts
12
13 end module constants
14
15
16 program module_example
17 use constants
18 implicit none
19
20 real :: x, ePowerx, area, radius
21 x = 2.0
22 radius = 7.0
23 ePowerx = e ** x
24 area = pi * radius**2
25
26 call show_consts()
27
28 print*, "e raised to the power of 2.0 = ", ePowerx
29 print*, "Area of a circle with radius 7.0 = ", area
30
31 end program module_example
```

پس از کامپایل کد بالا داریم:

```
ePowerx = e ** x
1
Error: Symbol 'e' at (1) has no IMPLICIT type
main.f95:19:13:

    area = pi * radius**2
    1
Error: Symbol 'pi' at (1) has no IMPLICIT type
```

آرایه‌ها در Fortran:

آرایه‌ها می‌توانند مجموعه‌ای متوالی از عناصر یک نوع با اندازه ثابت را ذخیره کنند. از آرایه‌ها برای ذخیره مجموعه‌ای از داده‌ها استفاده می‌شود، اما اغلب استفاده از آرایه‌ها به عنوان مجموعه‌ای از متغیرهای همان نوع بسیار مفیدتر است.

همه آرایه‌ها از مکان‌های حافظه مجاور تشکیل شده‌اند. کمترین آدرس مربوط به اولین عنصر و بالاترین آدرس به آخرین عنصر است.

Numbers(1)	Numbers(2)	Numbers(3)	Numbers(4)	Numbers(5)
------------	------------	------------	------------	------------	-----	-----

آرایه‌ها می‌توانند یک بعدی (مانند بردارها)، دو بعدی (مانند ماتریس‌ها) باشند و Fortran این امکان را می‌دهد تا آرایه‌های ۷ بعدی ایجاد شود.

اعلام آرایه‌ها:

آرایه‌ها با مشخصه بعد (dimension) اعلام می‌شوند.

به عنوان مثال، برای اعلام یک آرایه یک بعدی به نام number، از نوع real numbers شامل ۵ عنصر، مانند زیر نوشته می‌شود:

```
real, dimension(5) :: numbers
```

برای ایجاد یک ماتریس دو بعدی 5*5 از نوع integer، مانند زیر عمل می‌کنیم:

```
integer, dimension (5,5) :: matrix
```

همچنین می‌توان یک آرایه با کران (محدوده) مشخص تعیین کرد، مانند:

```
real, dimension(3:7) :: numbers  
integer, dimension (-3:2,0:4) :: matrix
```

اختصاص مقدار:

می‌توان به تک تک اعضای آرایه مقدار مشخصی را اختصاص داد.

```
numbers(1) = 2.0  
numbers(3) = 1.7
```

یا می‌توان از حلقه برای مقداردهی استفاده کرد:

```
do i = 1, 5
  numbers(i) = i * 2.0
end do
```

آرایه‌های یک بعدی را می‌توان مستقیماً مقداردهی کرد، مانند:

```
Numbers = (/ 1.6, 4.0 , 2.3, 9.1 , 0.8 /)
```

توجه!: بین پرانتز و "/" نباید فاصله باشد.

تخصیص مقادیر یک آرایه به آرایه دیگر مجاز است به شرطی که هر دو آرایه مورد نظر دارای بعد فیزیکی یکسان باشند. مثلاً:

```
Real , dimension(10) :: A , B
A = (/ 1,2,3,4,5,6,7,8,9,10 /)
B = A
```

مقادیر عناصر آرایه A را به آرایه B اختصاص می‌دهد.

برای اختصاص مقادیر به عناصر منفرد آرایه می‌توان از **where** به صورت زیر استفاده کرد:

```
Where (logical argument)
  Sequence of array assignments
Elsewhere
  Sequence of array assignments
End Where
```

به عنوان مثال اگر مقادیر A به صورت زیر باشد:

```
A = (/ (1, 1 = 1, 10) /)
```

سپس، می‌توان عناصر آرایه B را به صورت زیر در نظر گرفت:

```
Where ( A > 5 )
  B = 1
Elsewhere
  B = 0
End Where
```

مقادیر آرایه B به صورت زیر می‌شود:

```
B = (/ 0, 0, 0, 0, 0, 1, 1, 1, 1, 1 /)
```


عملگرها در آرایه:

عملگرهایی که به طور معمول در عبارات ساده اعمال می‌شوند ، ممکن است در آرایه‌هایی که تعداد عناصر یکسانی دارند نیز اعمال شوند. چنین عملیاتی بر اساس عنصر به عنصر (element by element) انجام می‌شود. مثلاً:

$$A = A + B$$

$$C = 2 * C$$

برخی از اصطلاحات مربوط به آرایه:

اصطلاح	معنی
Rank	تعداد ابعاد یک آرایه است. مثلاً برای ماتریس بالا Rank ، ۲ است و برای Numbers ، ۱ است.
Extent	تعداد عناصر در امتداد یک بعد است. مثلاً Extent در Numbers ، ۵ است.
Shape	یک آرایه صحیح یک بعدی است که شامل تعداد عناصر در هر بعد است. مثلاً برای ماتریس shape ، (۳, ۳) و برای numbers ، (۵) است.
Size	تعداد عناصر کلی یک آرایه است . برای ماتریس ۹ و برای Numbers ، ۵ است.

اختصاص حافظه به آرایه:

می‌توان برای آرایه نیز با دستور allocatable ، به صورت زیر حافظه اختصاص داد.

```
Real, dimension(:), ALLOCATABLE :: A
```

برای اطمینان از دسترس بودن حافظه کافی برای آرایه می‌توان از گزینه STAT از دستور ALLOCATE استفاده کرد:

```
ALLOCATE ( A(N), STAT = AllocateStatus)
IF (AllocateStatus /= 0 STOP "*** Not enough memory ***")
```

با استفاده از دستور DEALLOCATE می‌توان یک آرایه را از حافظه آزاد کرد.

```
DEALLOCATE (A , STAT = DeAllocateStatus )
```

مثال از آرایه‌ها:

```
Execute | Share main.f95 STDIN
1 program arrayProg
2
3   real :: numbers(5) !one dimensional integer array
4   integer :: matrix(3,3), i , j !two dimensional real array
5
6   !assigning some values to the array numbers
7   do i=1,5
8     numbers(i) = i * 3.0
9   end do
10
11  !display the values
12  do i = 1, 5
13    Print *, numbers(i)
14  end do
15
16  !assigning some values to the array matrix
17  do i=1,3
18    do j = 1, 3
19      matrix(i, j) = i+j
20    end do
21  end do
22
23  !display the values
24  do i=1,3
25    do j = 1, 3
26      Print *, matrix(i,j)
27    end do
28  end do
29
30  !short hand assignment
31  numbers = (/1.6, 4.0 , 2.3, 9.1 , 0.8 /)
32
33  !display the values
34  do i = 1, 5
35    Print *, numbers(i)
36  end do
37
38 end program arrayProg
```

خروجی:

```
Result
$gfortran -std=gnu *.f95 -o main
$main
3.00000000
6.00000000
9.00000000
12.00000000
15.00000000
2
3
4
3
4
5
4
5
6
1.60000002
4.00000000
2.29999995
9.10000038
0.80000012
```

بخش‌های آرایه:

تا اینجا به روال کلی آرایه‌ها اشاره شد، اما Fortran یک روش آسان برای ارجاع چندین عنصر یا بخشی از یک آرایه را با استفاده از یک عبارت ارائه می‌دهد.

برای دسترسی به یک بخش آرایه، باید قسمت پایین و قسمت فوقانی بخش و همچنین یک stride (افزایش) را برای تمام ابعاد ارائه مشخص کرد. به این علامت subscript triplet گفته می‌شود:

array ([lower]:[upper][:stride], ...)

! هنگامی که قسمت پایین و بالا ذکر نشده باشد، به طور پیش فرض همان مقداری که تعریف کردید را انتخاب می‌کند؛ هنگامی که برای stride مقدار ذکر نشده باشد، پیش فرض ۱ است.

مثال:

```
1 program arraySubsection
2
3   real, dimension(10) :: a, b
4   integer :: i, asize, bsize
5
6   a(1:7) = 5.0 ! a(1) to a(7) assigned 5.0
7   a(8:) = 0.0 ! rest are 0.0
8   b(2:10:2) = 3.9
9   b(1:9:2) = 2.5
10
11  !display
12  asize = size(a)
13  bsize = size(b)
14
15  do i = 1, asize
16    | Print *, a(i)
17  end do
18
19  do i = 1, bsize
20    | Print *, b(i)
21  end do
22
23  end program arraySubsection
```

خروجی:

```
Result
$gfortran -std=gnu *.f95 -o main
$main
5.00000000
5.00000000
5.00000000
5.00000000
5.00000000
5.00000000
5.00000000
0.00000000
0.00000000
0.00000000
2.50000000
3.90000010
2.50000000
3.90000010
2.50000000
3.90000010
2.50000000
3.90000010
2.50000000
3.90000010
```

:Procedure

Procedure به گروهی از عبارات گفته می‌شود که وظیفه‌ای کاملاً مشخص را انجام می‌دهند و می‌توانند از برنامه فراخوانی شوند. اطلاعات (یا داده‌ها) به عنوان آرگومان به برنامه فراخوانی منتقل می‌شوند.

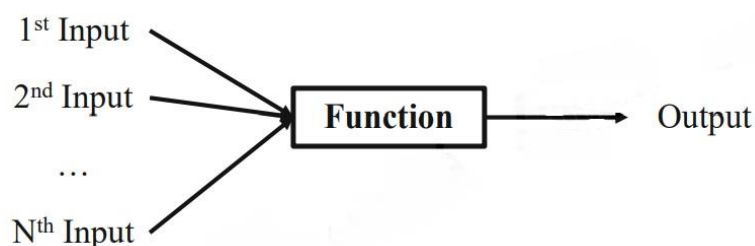
دو روش وجود دارد:

- Functions
- Subroutines

:Functions

تابع روشی است که یک مقدار واحد را برمی‌گرداند. یک تابع نباید آرگومان‌های خود را اصلاح کند؛ مقدار برگشتی به عنوان مقدار تابع شناخته می‌شود و با نام تابع مشخص می‌شود.

هدف از یک تابع این است که تعدادی از مقادیر یا آرگومان‌ها را در اختیار شما قرار می‌دهد، برخی محاسبات را با این آرگومان‌ها انجام می‌دهید و سپس یک نتیجه واحد را برمی‌گرداند.



برخی از توابع وجود دارد که در FORTRAN نوشته شده‌اند و می‌توانند بدون هیچ گونه تلاش خاصی توسط شما، برنامه نویس، مورد استفاده قرار گیرند؛ آن‌ها **intrinsic functions** نامیده می‌شوند. بیش از ۴۰ **intrinsic functions** در FORTRAN وجود دارد و آنها عمدتاً مربوط به توابع ریاضی هستند.

روش کلی فعال سازی یک تابع استفاده از نام تابع در یک عبارت است. به دنبال نام تابع، لیستی از ورودی‌ها که به آنها آرگومان نیز گفته می‌شود، در پرانتز قرار می‌گیرند.

یک تابع ممکن است یک یا چند آرگومان داشته باشد اما تنها یک نتیجه می‌دهد.

:Syntax

```
function name(arg1, arg2, ....)
  [declarations, including those for the arguments]
  [executable statements]
end function [name]
```

مثال:

```
function func(i) result(j)
  integer, intent(in) :: i ! input
  integer              :: j ! output
  j = i**2 + i**3
end function

program main
  implicit none
  integer :: i
  integer :: func
  i = 3
  print *, "sum of the square and cube of", i, "is", func(i)
end program
```

! ویژگی `intent(in)` در آرگومان `i` به این معنی است که `i` را نمی‌توان در داخل تابع تغییر داد و در مقابل ، مقدار برگشتی `z` دارای `intent(out)` خودکار است. توجه داشته باشید که نوع بازگشتی عملکرد باید اعلام شود. اگر این مورد حذف شود ، برخی از کامپایلرها کامپایل نمی‌کنند.

خروجی کد بالا:

```
Sum of the square and cube of 3 is 36
```

: intent

هنگام اعلام متغیرها در داخل توابع و زیر برنامه‌ها که باید به داخل یا خارج منتقل شوند ، ممکن است `intent` به اعلامیه اضافه شود.

Explanation	Used as	values
به عنوان مقدار ورودی استفاده می‌شود اما در تابع مقدارش تغییر نمی‌کند.	<code>intent(in)</code>	<code>in</code>
به عنوان مقدار خروجی استفاده می‌شود و با نادیده گرفتن مقادیر اصلی به تابع ارسال می‌شود.	<code>intent(out)</code>	<code>out</code>
متغیر با یک مقدار وارد می‌شود و با یک مقدار خارج می‌شود.	<code>intent(inout)</code>	<code>inout</code>

توابع ذاتی (Intrinsic Functions):

در زیر به برخی از توابع ذاتی فورترن اشاره شده است.

Math functions

<code>abs(x)</code>	absolute value.
<code>conjg(z)</code>	complex conjugate.
<code>dim(x, y)</code>	maximum of x-y or 0.
<code>dprod(x, y)</code>	double precision product.
<code>exp(x)</code>	exponential.
<code>log(x)</code>	natural logarithm.
<code>log10(x)</code>	base 10 logarithm.
<code>max(x1, x2[, x3...])</code>	returns the maximum value.
<code>min(x1, x2[, x3...])</code>	returns the minimum value.
<code>mod(x, p)</code>	remainder modulo p, i.e. $x - \text{int}(x/p) * p$.
<code>modulo(x, p)</code>	x modulo p.
<code>sign(x, y)</code>	absolute value of x times the sign of y.
<code>sqrt(x)</code>	square root.

Trig functions

<code>acos(x)</code>	inverse cosine.
<code>asin(x)</code>	inverse sine.
<code>atan(x)</code>	inverse tan.
<code>atan2(x, y)</code>	inverse tan.
<code>cos(x)</code>	cosine.
<code>cosh(x)</code>	hyperbolic cosine.
<code>sin(x)</code>	sine.
<code>sinh(x)</code>	hyperbolic sine.
<code>tan(x)</code>	tan.
<code>tanh(x)</code>	hyperbolic tan.

Random number

<code>random_number(x)</code>	fill in x with random numbers in the range [0,1]. x may be a scalar or array.
<code>random_seed([size][, put][, get]</code> <code>)</code>	initialise or reset the random number generator.

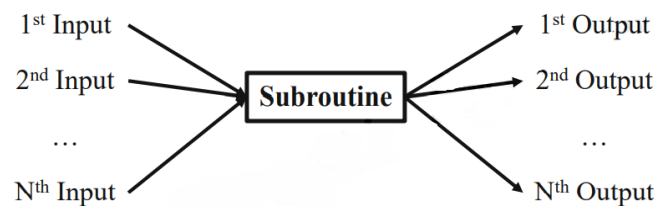
Character functions

<code>achar(i)</code>	returns the ith character in the ascii character set (like C's chr).
<code>char(i[, kind])</code>	returns the ith character in the machine specific character set (like C's chr).
<code>iachar(c)</code>	returns the position of the character in the ascii character set (like C's ord).
<code>ichar(c)</code>	returns the position of the character in the machine specific character set (like C's ord).

بسیاری از زبان‌های برنامه نویسی تفاوتی بین توابع و زیرروال‌ها قائل نمی‌شوند (به عنوان مثال C / C ++، Python، Java). اما در Fortran، **functions** و **subroutines** متفاوت است: اولی مقداری را برمی‌گرداند در حالی که دومی این کار را نمی‌کند.

: subroutines

subroutines مقداری را بر نمی‌گرداند، با این حال می‌تواند آرگومان‌های خود را اصلاح کند.



:Syntax

```

subroutine name(arg1, arg2, ....)
  [declarations, including those for the arguments]
  [executable statements]
end subroutine [name]
  
```

:subroutines فراخوانی

با استفاده از دستور **call** فراخوانی می‌شود.

مثال:

```

subroutine square_cube(i, isquare, icube)
  integer, intent(in) :: i           ! input
  integer, intent(out) :: isquare, icube ! output
  isquare = i**2
  icube   = i**3
end subroutine

program main
  implicit none
  external square_cube           ! external subroutine
  integer :: isq, icub
  call square_cube(4, isq, icub)
  print *, "i,i^2,i^3=", 4, isq, icub
end program main
  
```

خروجی کد بالا:

i,i^2,i^3=	4	16	64
------------	---	----	----

:Class

در زبان‌های شی‌گرا مانند C++، می‌توان کلاس‌هایی را تعریف کرد که هم شامل داده باشند و هم روش‌هایی که روی آن داده کار می‌کنند. سپس می‌توان نمونه‌های جداگانه‌ای از کلاس ایجاد کرد، هرکدام داده‌های خاص خود را دارند. متدی که از یک نمونه از کلاس فراخوانی می‌شود، روی داده‌های نگهداری شده توسط آن نمونه خاص کار خواهد کرد.

در Fortran، ماژول‌ها ممکن است حاوی داده باشند، اما تصویری از موارد جداگانه یک ماژول وجود ندارد. برای به دست آوردن رفتاری مانند کلاس، می‌توان یک ماژول را که حاوی روش‌های کار بر روی کلاس است با یک derived type حاوی داده‌ها ترکیب کرد. از این نوع "instances" جداگانه‌ای وجود دارد که می‌توان متغیرهای زیادی را از آن نوع اختصاص داد که می‌توانند به عنوان پارامتر به روش‌های موجود در ماژول منتقل شوند.

```
Execute | Share | main.f95 | STDIN
1 module class_Circle
2   implicit none
3   private
4   real :: pi = 3.1415926535897931d0 ! Class-wide private constant
5
6   type, public :: Circle
7     real :: radius
8     contains
9       procedure :: area => circle_area
10      procedure :: print => circle_print
11   end type Circle
12 contains
13   function circle_area(this) result(area)
14     class(Circle), intent(in) :: this
15     real :: area
16     area = pi * this%radius**2
17   end function circle_area
18
19   subroutine circle_print(this)
20     class(Circle), intent(in) :: this
21     real :: area
22     area = this%area() ! Call the type-bound function
23     print *, 'Circle: r = ', this%radius, ' area = ', area
24   end subroutine circle_print
25 end module class_Circle
26
27
28 program circle_test
29   use class_Circle
30   implicit none
31
32   type(Circle) :: c ! Declare a variable of type Circle.
33   c = Circle(1.5) ! Use the implicit constructor, radius = 1.5.
34   call c%print ! Call the type-bound subroutine
35 end program circle_test
```

Result

```
$gfortran -std=gnu *.f95 -o main
```

```
$main
```

```
Circle: r = 1.50000000 area = 7.06858349
```


شی‌گرایی در Fortran:

- Module:

در چند مبحث قبل به Module ها به طور کامل پرداخته شده است .

- Data access:

سه روش برای دسترسی وجود دارد:

- **Public**: کد خارج اجازه خواندن و نوشتن را دارد.
- **Private**: کد خارج، اجازه دسترسی ندارد.
- **public, protected**: کد خارج، اجازه دسترسی خواندن را دارد.

```
1 module data_access_m
2   implicit none
3
4   private
5
6   public  a, b
7   protected b
8   private c
9
10  integer :: a=1
11  integer :: b=1
12  integer :: c=1
13 end module
14
15 program main
16   use data_access_m
17
18   ! accessing public object works
19   print *, a
20
21   ! editing public object works
22   a = 2
23
24   ! accessing protected object works
25   print *, b
26
27   ! editing protected object does not work
28   !b = 2 <- ERROR
29
30   ! accessing private object does not work
31   !print *, c <- ERROR
32
33   ! editing private object does not work
34   !c = 2 <- ERROR
35 end program
```

```
Result
$gfortran -std=gnu *.f95 -o main
$main
1
1
```

• Derived data types:

در Fortran می توان ساختارهایی را از ساختارهای دیگر استخراج کرد که اصطلاحاً انواع داده های مشتق شده نامیده می شوند. که در مباحث قبل به توضیح ان پرداختیم.

• Polymorphism in Fortran 2003:

رابطه "is a" است همچنین به ما کمک می کند متغیرهای چند شکلی (polymorphic variables) با پسوندهای نوع ارتباط (type extensions) برقرار کنند. کلمه کلیدی CLASS به برنامه نویسان F2003 امکان ایجاد متغیرهای چند شکل را می دهد. **یک متغیر چند شکل متغیری است که نوع داده آن در زمان اجرا پویا است.** باید یک متغیر اشاره گر ، یک متغیر قابل تخصیص یا یک استدلال ساختگی باشد. در زیر یک مثال آورده شده است.

```
class(shape), pointer :: sh
```

در مثال بالا ، "sh" می تواند یک نشانگر برای "shape" یا هر نوع پسوند نوع آن باشد. بنابراین ، می تواند یک اشاره گر به یک "shape" ، یک "rectangle" ، یک "square" یا هر نوع "shape" در آینده باشد. تا زمانی که نوع هدف اشاره گر "is a" باشد ، "sh" می تواند به آن اشاره کند.

:Basic types of polymorphism

- **procedure polymorphism**: با procedures سروکار دارد که می تواند روی انواع مختلفی از داده ها و متغیرها کار کند.
- **data polymorphism**: با متغیرهای برنامه سروکار دارد که می تواند انواع مختلفی از داده ها و متغیر را ذخیره کند و روی ان ها کار کند.

Static scope

دامنه از نظر statically تعریف می‌شود. هر ارجاع به یک متغیر از نظر ایستایی به یک اعلامیه متغیر خاص متصل می‌شود. Fortran ، Algol60 ، Pascal ، Ada ، C ، C ++ ، Scheme ، Common Lisp از استاتیک استفاده می‌کنند.

Dynamic scoped

دامنه اتصال در زمان اجرا تعیین می‌شود. در Lisp ، APL ، Snobol و Perl استفاده می‌شود.

تنها قانون واقعی Scoping در F90

- هیچ دامنه عمومی در Fortran وجود ندارد.
 - تنها مکانی که به نوعی دامنه وجود دارد ، استفاده از توابع داخلی (internal functions) است.
- یک تابع داخلی ممکن است به تمام متغیرهای تعریف شده در enclosing function دسترسی داشته باشد. (مگر اینکه متغیرهای محلی (local variables) دیگری را به همین نام حذف شوند).
- با این حال ، enclosing function نمی‌تواند به متغیرهای تعریف شده در internal functions دسترسی داشته باشد.

مثال:

```
FUNCTION pythagoras(a, b)      ! The enclosing function
  IMPLICIT NONE

  REAL :: pythagoras, a, b
  REAL :: s

  s = square(a) + square(b)
  pythagoras = sqrt(s)
  RETURN

CONTAINS                      ! internal functions follow...

  REAL FUNCTION square(x)
  REAL :: x

  print *, "a = ", a, ", b = ", b !! You can access outer variables

  square = x*x
  return
END FUNCTION square

END FUNCTION pythagora
```

```

1  FUNCTION pythagoras(a, b) ! The enclosing function
2  IMPLICIT NONE
3
4  REAL pythagoras
5  REAL a, b
6
7  REAL s
8
9  s = square(a) + square(b)
10 pythagoras = sqrt(s)
11 RETURN
12
13 CONTAINS ! internal function follow...
14
15 FUNCTION square(x)
16
17 REAL square
18 REAL x
19
20 print *, "a = ", a, ", b = ", b
21
22 square = x*x
23 return
24 END FUNCTION !! END "FUNCTION" is required
25
26 END FUNCTION pythagoras !! END required, FUNCTION is optional
27
28 ! =====
29
30 PROGRAM internal
31 IMPLICIT NONE
32
33 REAL :: x
34 REAL, EXTERNAL :: pythagoras
35
36 x = pythagoras(3.0, 4.0)
37 PRINT *, "X = ", x
38
39 ! x = square(3.0) !! Error: cannot call internal function
40
41 END program internal

```

خروجی برنامه:

```

Result
$gfortran -std=gnu *.f95 -o main
$main
a = 3.00000000 , b = 4.00000000
a = 3.00000000 , b = 4.00000000
X = 5.00000000

```

دامنه (Scope) و طول عمر (Lifetime) متغیرهای "Local":

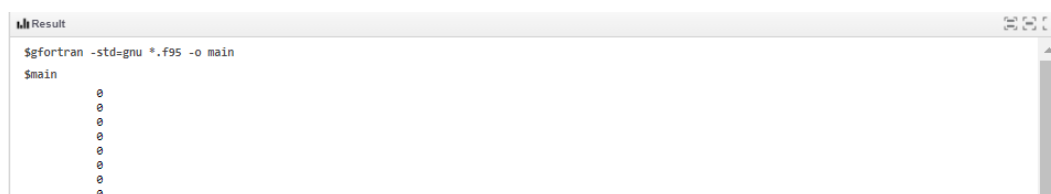
- فقط به صورت محلی در داخل یک تابع قابل دسترسی هستند.
- فقط در داخل آن تابع و توابع داخلی آن قابل دسترسی هستند.
- در توابع Fortran با شروع اجرای برنامه ایجاد می شوند.

Fortran 77: دامنه یک متغیر Local محدود به یک subroutine است. دامنه یک متغیر global کل متن برنامه است ، مگر اینکه توسط یک اعلامیه متغیر Local با همان نام متغیر پنهان شود.

مثال :

```
1  INTEGER FUNCTION myFunction1()
2  IMPLICIT NONE
3
4  INTEGER :: i
5
6  myFunction1 = i
7  i = i + 1
8  RETURN
9
10 END FUNCTION
11
12 ! =====
13
14 INTEGER FUNCTION myFunction2()
15 IMPLICIT NONE
16
17 INTEGER :: i
18
19 myFunction2 = i
20 i = i + 1
21 RETURN
22
23 END FUNCTION
24
25 ! =====
26
27 PROGRAM internal
28 IMPLICIT NONE
29
30 interface
31   function myFunction1()
32     integer myFunction1
33   end function
34
35   function myFunction2()
36     integer myFunction2
37   end function
38 end interface
39
40
41 PRINT *, myFunction1()
42 PRINT *, myFunction1()
43 PRINT *, myFunction1()
44 PRINT *, myFunction1()
45
46 PRINT *, myFunction2()
47 PRINT *, myFunction2()
48 PRINT *, myFunction2()
49 PRINT *, myFunction2()
50
51 END program internal
```

از آنجا که متغیرهای محلی i در ابتدای برنامه ایجاد می‌شوند ، مقدار بین فراخوانی عملکردها را حفظ می-
کنند. در نتیجه ، خروجی کد بالا:



```
Result
$gfortran -std=gnu *.f95 -o main
$main
0
0
0
0
0
0
0
0
```

کتابخانه‌ها در Fortran:

ابزارها و کتابخانه‌های مختلف Fortran وجود دارد. برخی رایگان و برخی خدمات پولی هستند. در زیر چند کتابخانه رایگان آورده شده است.

- RANDLIB, تولیدکننده‌های توزیع آماری تعداد و تصادفی
- BLAS
- EISPACK
- GAMS–NIST راهنمای نرم افزار ریاضی موجود
- NIST برخی از برنامه‌های آماری و دیگر برنامه‌های
- LINPACK
- MINPACK
- MUDPACK
- NCAR کتابخانه ریاضی
- Netlib مجموعه نرم افزارهای ریاضی ، مقالات و پایگاه‌های داده
- ODEPACK
- ODERPACK, مجموعه‌ای از برنامه‌های معمول برای رتبه بندی و سفارش
- Expokit برای محاسبه نماهای ماتریس
- SLATEC
- SPECFUN
- STARPAC
- StatLib کتابخانه آماری
- TOMS
- Sorting and merging strings

کتابخانه‌های زیر رایگان نیستند:

- The NAG Fortran numerical library
- The Visual Numerics IMSL library
- Numerical Recipes