مقدمه‌ای بر کد های فرترن

این فصل مبتنی بر یادداشت‌های امین کیانی از کلاس سال ۱۳۸۷ در دانشگاه صنعتی اصفهان است.

اکنون می خواهیم با کلیات دستورات فرترن در غالب نوشتن یک برنامه‌ی کامپیوتری، ازساده ترین حالت ممکن یعنی یک برنامه ۲، ۳ خطی تا یک برنامه ۳۰، ۴۰ خطی آشنا شوید. در ادامه هم در مورد makefile اینکه چی هست، چه مزیت هایی دارد و چگونه ساخته می شود به شما توضیح خواهیم داد. این نوید را به شما می دهیم که بعد از این بخش شما به تنهایی قادر به نوشتن برنامه‌های طولانی خواهید بود.

خب، با ساده ترین برنامه ی ممکن آغاز می کنیم:

```
program test
   print*,"test"
end program
```

هر برنامه فرترن با کلمه program، سپس نام برنامه که با یک یا چند فاصله بعد از program می آید، شروع شده و با end program خاتمه میابد. البته ضرری هم ندارد که اسم برنامه را نیز بعد از end program بیاوریم. دستورات اجرایی هم بین این دو قسمت قرار می گیرد. در خط دوم، دستور print* کلمه ی test را برای شمادر صفحه ی ترمینال چاپ می کند. در فرترن، به طور کلی، برای چاپ یک کاراکتر یا کلمه یا ... بعد از دستور چاپ بایستی که آن را در یک " " یا' ' قرار دهیم. حالا بیایید و کمی کلاس برنامه مان را بالاتر ببریم.می خواهیم برنامه مان یک سری جمع و تقسیم را محاسبه کند. پس تایپ می کنیم:

```
program test
   a = 1 ; b = 2 ; c = 3
   y = a*b + c/a - b
   i = a*b + c/a - b
   p = a**b + c/b
   j = a**b + c/b
   print*, y, i, p, j
end program test
```

در فرترن * عملگر ضرب، + عملگر جمع، − عملگر تفریق، / عملگر تقسیم، ** عملگر توان، = عملگر تخصیص، == عملگر تساوی است. اگر برنامه را اجرا کنید نتیجه جالب و غیر منتظره خواهد شد. این برنامه مقدار y را برابر 3.000000 و مقدار i را برابر با 3 چاپ کرد. یعنی این که y متغیر اعشاری در نظر گرفته شده در حالی که i متغیر صحیح. از طرفی انتظار داشتیم که مقدار p و j با هم برابر باشند، در حالی که نتیجه‌ی اجرا برای p مقدار 2.500000 و برای j مقدار 2 را نشان می دهد! احتمالا دلیل، آن است که فرترن برخی از حروف یا کلماتی که با این حروف آغاز می شوند را به عنوان متغیر اعشاری و مابقی را به عنوان متغیر صحیح در نظر می‌گیرد. صحت این مطلب را می توانید به راحتی بررسی کنید. کافیست مقدار های گوناگون به این پارامتر ها تخصیص بدهید و بعد آن ها را چاپ کنید.(فعلا برای شروع بد نیست) شما می توانید با این کار نوع حروف یا کلمات را از نظر اعشاری یا صحیح بودن تعیین کنید. گاهی ممکن است که در یک برنامه ۲، ۳ هزار خطی همین مورد بالا (یعنی در نظر گرفتن متغیر اعشاری یا صحیح برای حروف و کلمات خاص به صورت پیش فرض) باعث ایجاد نتیجه‌ی نادرست شود، هر چند که برنامه شما از نظر الگوریتم هیچ اشکالی نداشته باشد. برای رفع این مشکل از دستور implicit none استفاده می کنیم. توصیه اکید می کنیم که در تمامی برنامه هایی که می نویسید از این دستور استفاده کنید. در واقع استفاده نکردن از این دستور برای من برنامه نویس به نحوی عذاب وجدان ایجاد می کند! ما با این دستور، عنان اختیار برای تعریف کردن نوع متغیرها را از فرترن می گیریم و این کار را خودمان انجام می دهیم. حالا بیایید و بعد از program test این دستور را وارد کنیم:

```
program test
   implicit none
   a = 1 ; b = 2 ; c = 3
   y = a*b + c/a - b
   i = a*b + c/a - b
   p = a**b + c/b
   j = a**b + c/b
   print*, y, i, p, j
end program test
```

برنامه را کامپایل کنید. به چند خطا برخورد می کنید، که اولین آن به صورت زیر است:

```
fortcom: Error: test.f90, Line 3: This name does not have a
type, and must have an explicit type. [A]
   a=1; b=2; c=3
---^
```

این را به این دلیل آوردیم که شما کم کم با این گونه خطاها آشنا شوید تا در آینده بتوانید آنها را در ۳ سوت رفع کنید. خطای بالا به ما می گوید که در خط سوم برنامه، نوع یک اسم مشخص نشده است و سپس دقیقا زیر کلمه‌ای که به آن خطا گرفته علامت قرار داده است. پس در اینجا لازم است که نوع پارامتر a را مشخص کنیم. خطاهای بعدی هم مشابه همین خطا هستند. بنابراین برنامه بالا را به صورت زیر اصلاح می کنیم.

```
program test
   implicit none
   integer :: a, b, y, i
   real :: c,p,j
   a = 1 ; b = 2 ; c = 3
   y = a*b + c/a - b
   i = a*b + c/a - b
   p = a**b + c/b
   j = a**b + c/b
   print*, y, i, p, j
end program test
```

در خط سوم از برنامه برای تعریف متغیرهای صحیح از دستور integer :: استفاده کردیم. در خط چهارم هم متغیر های اعشاری را تعریف کردیم که برای آن از دستور real :: استفاده نمودیم. البته گذاشتن :: ضرورتی ندارد ولی بهتر است که آن را قرار دهیم، چون فرم استاندارد آن به صورت گفته شده می باشد. حالا بیایید و کمی با برنامه بالا بازی کنیم. دستور implicit none را بعد از real :: قرار دهید. برنامه را کامپایل کنید. به یک خطا برخورد می کنید:

```
fortcom: Error: test.f90, Line 5: This IMPLICIT statment is
not positioned correctly within the scoping unit.
   implicit none
---^
```

اگر خطای بالا را به دقت بخوانید، متوجه خواهید شد که جایی که این دستور را قرار داده‌ایم نادرست است. پس این دستور بایستی حتما بعد از program test قرار بگیرد. شاید شما علاقمند نبودید که پارامترهای a, b, c را در داخل خود برنامه مقدار دهی کنید و دوست داشتید که این مقادیر را در هنگام آغاز اجرای برنامه، خودتان به عنوان ورودی به برنامه بدهید. برای این کار می توانید از دستور *read استفاده کنید که داده های ورودی شما را از صفحه ترمینال گرفته و در برنامه قرار می دهد. پس سطر ۵ در برنامه‌ی بالا حذف کنید و به جای آن دستور زیر را قرار دهید:

```
read*, a, b, c
```

این دستور اجازه خواهد داد که دو عدد صحیح و یک عدد اعشاری را که یکی پس از دیگری از صفحه کلید تایپ می کنید، پس از فشار دادن دکمه Enter به متغیرهای a, b, c اختصاص یابد. البته باید به طریق تایپ کردن این اعداد دقت کنید. می توانید سه عدد را با فاصله نوشته یا بین هر کدام یک کاما قرار دهید، سپس دکمه‌ی Enter را فشار دهید، یا این که بعد از تایپ هر عدد دکمه‌ی Enter را بزنید. البته دستور دیگری مخصوص خواندن ورودی و چاپ خروجی هم هست که در قسمت های بعد به تفصیل به آنها خواهیم پرداخت. یک نکته‌ی دیگر آنکه گذاشتن یک علامت ! در ابتدای هر خط باعث می‌شود که آن خط از برنامه کامپایل و اجرا نشود.

باز هم برنامه‌مان را تغییر می‌دهیم و دو دستور `: : real(8)` و `: : real(16)` را بعد از سطر چهارم تایپ می‌کنیم.اکنون، پارامتر p را در مقابل `: : real(8)` و پارامتر j را در مقابل `: : real(16)` قرار دهید و برنامه را اجرا کنید. نتیجه جالب خواهد بود. برنامه مقادیر y و i را برابر با 3، مقدار p را برابر با 2.50000000000000 و مقدار j را برابر با 2.50000000000000000000000000000000 چاپ می‌کند. به راحتی می‌توان تشخیص داد که دو دستور ذکر شده دقت را افزایش می‌دهند. اما در مورد `integer` چی؟ این مورد را به عهده‌ی خودتان واگذار می‌کنیم. می‌توانید با استفاده از help فرترن به پاسخ این سوال برسید. این بار هدف از تغییر دادن برنامه‌مان را محاسبه کردن روابط پیچیده‌تر قرار می‌دهیم. به جای روابط موجود در خطهای ۶ تا ۹ فعلا دو رابطه‌ی زیر را قرار می‌دهیم.

```
y = sin(a) + cos(b) + log(c)
p = sin(a) + cos(b) + log(c)
```

و در دستور مربوط به چاپ تنها دو متغیر y و p را می‌دهیم. برنامه را کامپایل کنید، خواهید دید که به دو تا خطا به صورت زیر برخورد خواهید کرد:

```
fortcom: Warning: test.f90, Line 9: This argument's data type
is incomptible with this intrinsic procedure;
procedure assumed EXTERNAL. [SIN]
   y = sin(a) + cos(b) + log(c)
-------^
fortcom: Warning: test.f90, Line 9: This name dose not have
a type, and must be have an explicit type. [SIN]
   y = sin(a) + cos(b) + log(c)
-------^
```

بیایید و خطای اول را با دقت بخوانیم و سعی کنیم آن را برطرف کنیم. خطای اول به ما می‌گوید که نوع داده‌ی ورودی آرگومان تابع با دستورات طبیعی موجود در فرترن سازگاری ندارد. برای رفع این مشکل از help فرترن استفاده می‌کنیم. کافیست که شما در آن کلمه sin را جستجو کنید. خواهید دید برای این که بتوانیم از تابع sin استفاده کنیم بایستی که حتما آرگومان آن اعشاری باشد. از طرفی مقداری هم که تابع sin به ما برمی‌گرداند، یک مقدار اعشاری است. به جدول زیر توجه کنید:

Type Result	Type Argument	Name Specific
REAL(۴)	REAL(۴)	SIN
REAL(۸)	REAL(۸)	DSIN
REAL(۱۶)	REAL(۱۶)	QSIN

اما ما آرگومان a را یک عدد صحیح تعریف کردیم و نمی‌خواهیم که نوع آن را از `integer` به `real` جابه‌جا کنیم. برای رفع این مشکل، یعنی هم a صحیح باشد و هم مقدار sin را به ما بدهد، می‌توانیم از دو دستور استفاده کنیم. یک دستور، `real` و دستور دیگر `float` است، که به صورت زیر از آن ها استفاده می‌کنیم:

```
y = sin(real(a)) + cos(real(b)) + log(c)
```

و یا

```
y = sin(float(a)) + cos(float(b)) + log(c)
```

اکنون برنامه را کامپایل کنید. مشاهده می‌کنید که نه تنها خطای اول، که خطای دوم هم رفع شد! و نتیجه‌ی اجرای برنامه هم به این صورت است که مقدار 1 را برای y و مقدار 1.52393639087677 را برای p چاپ می‌کند. با توجه به نوع متغیرهای y وp خودتان می‌توانید حدس بزنید که چرا نتیجه‌ی اجرا به این صورت شده است. در جدول بالا، دو تابع دیگر هم وجود دارد که هر دو مقدار sin را حساب می‌کنند اما این کجا و آن کجا! عبارت dsin مقدار sin را تا ۱۴ رقم بعد اعشار و qsin تا ۳۰ رقم بعد اعشار حساب می‌کند. خودتان این مطلب را بررسی کنید. قبل از این که به ادامه‌ی توضیحات بپردازیم، لازم است که توابع ریاضی مفید را برای شما در یک جدول به صورت فشرده ارائه کنیم. این که این توابع چه نوع داده‌هایی را به عنوان ورودی می‌گیرند و به عنوان خروجی بر می‌گردانند:

Type Result	Type Argument	Name Specific
REAL(۴)	REAL(۴)	SIN،COS،ASIN،ACOS
REAL(۴)	REAL(۴)	TAN،COTAN،ATAN
REAL(۴)	REAL(۴)	EXP،LOG۱۰،LOG،SQRT
REAL(۸)	REAL(۸)	DSIN،DCOS،DASIN،DACOS
REAL(۸)	REAL(۸)	DTAN،DCOTAN،DATAN
REAL(۸)	REAL(۸)	DEXP،DLOG۱۰،DLOG،DSQRT
REAL(۱۶)	REAL(۱۶)	QSIN،QCOS،QASIN،QACOS
REAL(۱۶)	REAL(۱۶)	QTAN،QCOTAN،QATAN
REAL(۱۶)	REAL(۱۶)	QEXP،QLOG۱۰،QLOG،QSQRT

اما توابع دیگری مثل ABS(x)(تابع قدرمطلق)، INT(x)(تبدیل به عدد صحیح)، floor(x)(تابعی که نزدیک‌ترین عدد صحیح که از x بزرگتر نباشد را به ما می‌دهد) و MOD(x,y) (باقی‌مانده‌ی x بر y). که در مورد این توابع در help فرترن جستجو کنید و نوع آرگومان ورودی و خروجی این توابع را همراه با میزان دقت آن ها پیدا کنید. این کار باعث می‌شود که کم کم دست شما در استفاده از help باز شود. اکنون برنامه را به صورت زیر اصلاح کنید:

```
program test
   implicit none
   integer :: a
   real :: c
   real(8) :: p
   read*, a
   p = sin(float(a))+cos(float(a))
   print*, a, p
end program
```

این برنامه اصلاح شده به ازای a مقدار sin(a)+cos(a) را برمی‌گرداند. مثلا برای a = 1 نتیجه‌ی اجرای این برنامه که در صفحه‌ی ترمینال چاپ خواهد شد به‌این صورت است:

```
 1          1.38177323341370
```

یک دستور دیگری که برای چاپ کردن می‌توان استفاده کرد، دستور write(*,*) است. با توجه به این که این دستور قابلیت های بیشتری نسبت به *prin دارد، از این جا به بعد از آن استفاده می‌کنیم. پس اگر خط هشتم برنامه را با دستور write(*,*) a, p جایگزین کنیم، نتیجه‌ی اجرای برنامه دقیقا همان فرم اجرای قبلی خواهد شد. ستاره اول در داخل پرانتز به ما می‌گوید که نتیجه‌ی اجرای برنامه در صفحه‌ی ترمینال چاپ خواهد شد و ستاره دوم هم مربوط به آن است که این دستور برای چاپ خروجی برنامه از فرمت آزاد (default) استفاده می‌کند. بعدا توضیح خواهیم داد که چگونه با فرمت های دلخواه به چاپ خروجی برنامه بپردازیم.
گاهی اوقات در برنامه نویسی این نیاز به وجود می‌آید که نتیجه‌ی اجرای برنامه در یک فایل ذخیره شود تا به بررسی و تحلیل داده ها بپردازیم و احیانا از این فایل در برنامه‌های دیگر به عنوان داده‌های ورودی استفاده کنیم. برای این کار بایستی تغییرات زیر را در برنامه اعمال کنیم:

- قبل از دستور write، دستور open(unit=1,FILE="data.txt") را تایپ کنید.
- در دستور write به جای ستاره اول در داخل پرانتز عدد 1 رابگذارید.
- دستور close(1) را بعد از دستور write قرار دهید.

در واقع با دستور open(unit=1,FILE="data.txt") ما یک فایل به اسم data.txt در واحد (unit) فرض شده‌ی شماره یک می‌سازیم. به سه نکته در مورد open توجه کنید:
اول این که نوشتن کلمه‌ی unit= کاملا اختیاری است. دوم، شماره‌ای که به unit اختصاص داده می‌شود دلخواه است، اما باید این عدد صحیح و مثبت باشد(خودتان بررسی کنید که مثلا اگر 1- قرار دادیم چه خواهد شد). سوم، این دستور تنها برای ذخیره خروجی برنامه به کار نمی‌رود و همان طور که بعدا خواهید برای این که ورودی یک برنامه را از یک فایل بگیریم، باید از این دستور استفاده کنیم.

برای این که نتایج برنامه در فایل data.txt ذخیره شود بایستی که شماره‌ی واحدی که در آن این فایل باز شده است را در آرگومان اول write قرار دهیم. در نهایت دستور close(1) فایل باز شده در واحد شماره‌ی یک را بعد از چاپ خروجی برنامه، می‌بندد. خودتان بررسی کنید، اگر دستور close(1) را از برنامه حذف کنیم، چه اتفاقی خواهد افتاد و این که آیا نتایج خروجی تغییر خواهد کرد یا نه؟
باز هم برنامه را تغییر می‌دهیم. این بار می‌خواهیم مقدار p را به ازای مقادیر مختلف c مثلا از۳ـ تا۳ حساب کنیم. برای این کار از دستور do enddo استفاده می‌کنیم. بنابراین در برنامه قبلی، اصلاحات زیر را اعمال می‌کنیم:
دستور *read را حذف کنید و عبارت زیر را در برنامه تایپ کنید:

```
do c = -3, 3, 0.01
   p = sin(c)+cos(c)
enddo
```

در این دستور مقدار c از۳ـ تا ۳ با گام‌های ۰۱,۰ ، ۰۱,۰ تغییر می‌کند. در واقع دستور do enddo یک حلقه‌ی تکرار است که مرتبه‌ی تکرار این حلقه توسط شما تعیین می‌شود. ساختار کلی این دستور قبلا توضیح داده شده است. قبل از این که به اجرای برنامه بپردازیم، بیایید و کمی در مورد محل قرار گرفتن دستورات close , open , write بحث کنیم. چون ما می‌خواهیم که به ازای هر مقدار از c یک مقدار از p به دست آوریم و آن را ذخیره کنیم، پس به نظر می‌رسد که بایستی این سه دستور را در داخل حلقه و پس از محاسبه‌ی p بنویسیم:

```
do c = -3, 3, 0.01
   p = sin(c)+cos(c)
   open(1, file="data.txt")
   write(1,*) c, p
   close(1)
enddo
```

حالا برنامه را اجرا کنید و سپس دستور more data.txt را در ترمینال تایپ کرده تا ببینید که چه اطلاعاتی در فایل ذخیره شده است. نتیجه آن چیزی نیست که ما انتظار داشتیم، چون تنها یک جفت عدد ذخیره شده است، یعنی برای c مقدار 3- و برای p مقدار 0.818806820335388- .
نتیجه‌ی اجرا نشان می‌دهد که تنها در دور اول از حلقه مقدار p محاسبه و در فایل ذخیره شده است. فکر می‌کنید که مشکل در کدام قسمت از برنامه باشد؟ همان طور که قبلا گفتیم دستور close(1) فایل باز شده در واحد شماره یک می‌بندد و پس از آن اجازه‌ی چاپ و ذخیره‌ی هیچ چیز دیگر را در آن نمی‌دهد. پس احتمالا در حلقه‌ی do enddo هنگامی که به دستور close(1) رسیده می‌شود، این دستور فایل باز شده را می‌بندد و دیگر اجازه‌ی ذخیره‌ی دیگر داده‌ها را نمی‌دهد. با این توضیحات بیایید و دستور close(1) را به بعد از حلقه‌ی do enddo تبعید کنیم. یعنی این دستور را دقیقا در سطر بعد از enddo تایپ کنید و برنامه را اجرا نمائید. اکنون فایل data.txt را باز کنید تا نتیجه‌ی اجرای برنامه‌تان را ببینید. پس مشکل از محل قرار گرفتن دستور close(1) بود. در اینجا نوشتن این دستور لزومی ندارد و اگر آن را از برنامه حذف کنید، هیچ مشکلی پیش نمی‌آید و نتیجه‌ی اجرای برنامه هیچ فرقی نخواهد کرد. همچنین می‌توان به جای این که دستور open(unit=1,FILE="data.txt") را درست قبل از write(1,*) و در حلقه بیاورید، آن را در ابتدای برنامه، ولی بعد از تعریف نوع پارامترها تایپ کنید (هیچ مشکلی به وجود نخواهد آمد). البته اگر نظر ما را بخواهید بهتر است که آن را در همان ابتدای برنامه تعریف کنید.
حالا می‌خواهیم از نتیجه‌ی برنامه‌مان لذت ببریم و نمودار sin(x)+cos(x) را رسم کنیم. برای این کار در صفحه‌ی ترمینال به ترتیب زیر تایپ کنید:

```
gnuplot
```

و دکمه‌ی Enter را بزنید. پس از آن تایپ کنید :

```
pl 'data.txt' w l
```

که در آن pl مخفف plot، w مخفف with و l مخفف line است و سطر بالا یعنی این که فایل data.txt را با خط رسم کند. بعدا در مورد gnuplot و قابلیت‌های فراوان آن توضیح خواهیم داد. شما می‌توانید با برنامه‌ی بالا بازی کنید و مثلا به جای cos(x) و sin(x) از توابع دیگر استفاده کنید.
اکنون با هدف توضیح پیرامون ساختار شرطی IF باز هم به دست کاری برنامه می‌پردازیم. ابتدا مختصری از ساختار کلی IF :

```
IF(logical expression)  Action
```

یا:

```
IF(logical expression) THEN
   Action
ENDIF
```

و یا:

```
IF(logical expression) THEN
   Action1
ELSE
   Action2
ENDIF
```

در مورد ساختار اول باید توجه کنید که تنها و تنها یک دستور بعد از if قرار می‌گیرد، در حالی که در دو ساختار بعدی هر تعدادی از دستورات فرترن که بخواهید می‌توانید از آن استفاده کنید. این ساختارها آن قدر گویا هستند که نیازی به توضیحات بیشتر نیست. حالا می‌خواهیم قدرمطلق p را در برنامه‌مان محاسبه و رسم کنیم. هر چند که در این‌جا می‌توانیم از تابع ABS(x) استفاده کنیم، اما برای این که با دستور if بیشتر آشنا شویم این کار را نمی‌کنیم و به طور مستقیم خودمان آن را محاسبه می‌نمائیم. پس برنامه را به صورت زیر اصلاح می‌کنیم:

```
open(1,file="data.txt")
do c = -5, 5, 0.01
   p = sin(c)+ cos(c)
   if(p < 0) p = -p
   write(1,*) c, p
enddo
```

برنامه را اجرا کنید و نمودار آن را ببنید!
این بار می‌خواهیم که تابع جزء صحیح p را محاسبه کنیم. برای این کار تایپ کنید:

```
do c = -5, 5, 0.01
   p = sin(c)+cos(c)
   if(p >=0 ) then
      p = int(p)
   else
      p = int(p)-1
   endif
   write(1,*) c, p
enddo
```

که این دستور شرطی if دقیقا معادل Floor(p) است.
اکنون یکی دیگر از شرطی‌ها را به شما معرفی می‌کنیم، شرطی‌های SELECT CASE :
قبل از این که دوباره به دستکاری برنامه‌مان بپردازیم بهتر است قبل از هر چیزی به بررسی ساختار این نوع شرطی بپردازیم. ساختار کلی این شرطی به صورت زیر است:

```
SELECT CASE (expression)
   CASE (selector)
      Action1
   CASE (selector)
      Action2
   .
   .
   .
END SELECT CASE
```

از این نوع شرطی می‌توان به جای شرطی‌های تو در توی if استفاده کرد، که پیچیدگی‌های آن را ندارد. در این شرطی ملاک تصمیم گیری برای این که کدام قسمت از دستورات فرترن انجام شود توسط expression تعیین می‌شود.
خب، برنامه را طوری تغییر می‌دهیم که به ازای مقادیر ۵ـ تا صفر جزء صحیح p و به ازای صفر تا ۵ قدر مطلق p محاسبه شود. از طرفی مقادیر قدر مطلق p در فایل data1.txt و مقادیر جزء صحیح p در فایل data2.txt ذخیره گردد. پس برنامه را به صورت زیر اصلاح می‌کنیم:

```
program test
   implicit none
   integer :: a
   real :: c
   real(8) ::p
   open(1, file = "data1.txt")
   open(2, file = "data2.txt")
   do c = -5, 5, 0.01
      p = sin(c)+cos(c)
      select case (c)
         case(-5:0)
            if(p >= 0) then
               p = int(p)
            else
               p = int(p)-1
            endif
            write(1,*) c, p
         case(1:5)
            if(p < 0) p = -p
            write(2,*) c, p
       end select case
   enddo
end program
```

قبل از این که به اجرا و کامپایل برنامه بپردازیم، منظور از case(1:5) آن است که به ازای تمامی اعداد صحیح ۱ تا ۵ دستور مورد نظر را اجرا کند. اگر برنامه را کامپایل کنید، به چند خطا بر خورد می‌کنید که اولین آن به صورت زیر است:

```
fortcom: Error: test.f90, Line 10: In a CASE statement, the
case_expr must be of type INTEGER, CHARACTER, or LOGICAL. [C]
      select case (c)
-------------------^
```

این خطا به طور واضح و آشکار به شما می‌گوید که عبارتی که می‌توانید در داخل select case بنویسید، باید یا از نوع INTEGER باشد یا از نوع CHARACTER و یا LOGICAL (برای دو مورد آخری خودتان از help فرترن کمک بگیرید). پس برای رفع اشکال از برنامه بایستی خط دهم برنامه را به صورت زیر اصلاح کنیم:

```
select case (int(c))
```

با این کار عبارت داخل پرانتز یک عدد صحیح خواهد بود. حال می‌توانید برنامه را با موفقیت کامپایل و سپس اجرا کنید. فعلا به طور موقت این برنامه را کنار می‌گذاریم. در ادامه می‌خواهیم که شما را با فرمت‌های چاپ و خواندن بیشتر آشنا کنیم. در واقع، فرمت مربوط به چگونگی مرتب کردن داده‌های عددی و یا کاراکتری که در ورودی یا خروجی برنامه قرار می‌گیرند، به کار می‌رود. همان‌طور که در قسمت‌های قبلی گفته شد، دستور write(*,*) مسئول چاپ خروجی برنامه در ترمینال است. درست در نقطه مقابل آن دستور read(*,*) وجود دارد که کار خواندن داده‌های ورودی را از ترمینال بر عهده دارد. در دستور read(*,*)، مشابه دستور write(*,*)، آرگومان اول مربوط به آدرسی که قرار است داده‌های ورودی از آن‌جا خوانده شود. آرگومان دوم هم مربوط به فرمتِ خواندن داده‌های ورودی است.
در فرترن فرمت I مخصوص چاپ اعداد صحیح، فرمت F مخصَوص چاپ اعداد اعشاری، فرمت E مخصوص چاپ اعداد به صورت نماد علمی و فرمت A مخصوص چاپ کاراکتر می‌باشد. اکنون در غالب یک برنامه به بررسی جزئیات مربوط به فرمت‌ها می‌پردازیم:

```
program test
   implicit none
   integer :: a=123, b=1e5
   real :: c=0.0001, d=1.d-4, e=1e-4
   real(8) :: f=123456789, g=0.0000001, h=1.d-7, i=0.0000001d0
   write(*, *) a
   write(*, "(i4)") a
   write(*, 13) a
   write(*, "(i6)") a
   write(*, "(i4.5)") a
   write(*, "(i4, i7)") a, b
   write(*, 19) a, b
```

```
   write(*, "(2i7)") a,b
   write(*, "(a20, i5)") "the value of a:", a
   write(*, "(a10, i5)") "the value of a:", a
   write(*, *) g,h,i
   write(*, "(f10.4)") c
   write(*, "(f10.4, f15.8, f7.4)") c, d, e
   write(*, 86) c, d, e
   write(*, "(3f10.5)") c, d, e
   write(*, "(f10.10)") f
   write(*, "(f20.10)") f
   write(*, "(e8.2)") c
   write(*, "(e15.5)") c
   write(*, "(e12.4e2)") c
   write(*, "(e15.5e5)") f
   13 format(i4)
   19 format(i4, i7)
   86 format(f10.4, f15.8, f7.4)
end program
```

در خط سوم،به b مقدار 1e5 را تخصیص دادیم که دقیقا معادل 100000 است (یعنی 100000 = 1e5). به طریق مشابه هم، می‌توانیم اعداد اعداد اعشاری را به این صورت بازنویسی کنیم. مثلا در خط چهارم، مقدار 1e-4 که معادل 0.0001 را به پارامتر e اختصاص دادیم. همچنین در همین خط از برنامه پارامتر d به صورت 1.d-4 مقدار دهی شده است که این کار باعث افزایش دقت در محاسبات و کمتر شدن درصد خطای محاسباتی در کامپیوتر می‌شود. در توضیح برنامه‌ی بالا بیشتر در این مورد صحبت می‌کنیم. اکنون برنامه را اجرا کنید. نتیجه‌ی اجرای خط ۶ برنامه به صورت زیر است:

```
---------123
         ^^^
```

پس در چاپ یک عدد صحیح با فرمت آزاد، یک میدان باعرض ۱۲ کاراکتر برای آن در نظر گرفته می‌شود که عدد 123 را از سمت راست در این میدان قرار داده و چاپ می‌کند.
برای نمایش اعداد صحیح در خروجی برنامه از فرم عمومی In استفاده می‌شود، که I مربوط به فرمت چاپ اعداد صحیح و n عرض میدانی است که عدد صحیح در آن چاپ می‌شود. البته فرم استاندارد چاپ با فرمت خاص که ماآن را در خط‌های ۷، ۱۱ و ۱۸ از برنامه استفاده نموده‌ایم، به صورت زیر است:

```
WRITE(*, number) a
number FORMAT(In)
```

که در خط هفتم number عدد 13 و n عدد 4 است. فرم دیگری که برای استفاده از فرمت در دستور write(*,*) استفاده می‌شود آن است که فرمت چاپ عدد را در داخل پرانتز و بین دو گیومه قرار داده و آن را در آرگومان دوم دستور چاپ می‌گذاریم (خط هشتم برنامه). بنابراین دو خط ۷ و ۸ برنامه نتیجه‌ی یکسانی را چاپ می‌کنند. اجرای برنامه برای خطوط ۷، ۸ و ۹ چنین است:

```
 123
 ^^^
 123
 ^^^
   123
---^^^
```

(در خط ۹ برنامه، عرض میدان ۶ کاراکتر در نظر گرفته شده است)
فرم دیگری که برای چاپ اعداد صحیح وجود دارد، In.m است که در آن m نشان‌دهنده‌ی حداقل تعداد ارقامی است که چاپ خواهد شد. مثلا، اگر فرمت خروجی برنامه I7.5 باشد، به ما می‌گوید که عدد صحیح a، اولا در یک میدان با عرض ۷ کاراکتر چاپ خواهد شد، ثانیا در این میدان ۵ رقم چاپ می‌گردد. یعنی:

```
  00123
--^^^^^
```

که توضیح بالا مربوط به اجرای خط ۱۰ برنامه می‌باشد. در خط بعدی، ما دو عدد صحیح a , b را یکی با فرمت i4 و دیگری را با فرمت i7 چاپ کردیم. یعنی به صورت زیر:

```
 123 100000
_^^^_^^^^^^
```

همچنین دو خط۱۱ و ۱۲ برنامه دقیقا معادل هم هستند که توضیح مشابه آن، در قسمت‌های بالاتر برای شما ارائه شد.
برای چاپ چند عدد صحیح با فرمت یکسان، می‌توان از فرم کلی pIn استفاده کرد که در آن p تعداد رقم‌هایی است که ما می‌خواهیم آن ها را چاپ کنیم. پس نتیجه‌ی اجرای خط ۱۳ برنامه به صورت زیر است:

```
    123 100000
____^^^_^^^^^^
```

برای چاپ متغیرهاو یا عبارت‌های کاراکتری از فرمت A.n استفاده می‌کنیم، که باز هم n نشان‌دهنده‌ی عرض میدان است. در خط ۱۳، عرض میدان تخصیص داده شده برای چاپ عبارت "the value of a:"، ۲۰ و در خط بعدی این عرض میدان ۱۰ در نظر گرفته شده است. نتیجه‌ی اجرا تفاوت این دو خط را به خوبی به شما نشان می‌دهد:

```
     the value of a:  123
_____^^^_^^^^^_^^_^^__^^^
the value   123
^^^_^^^^^___^^^
```

در خط ۱۵، سه متغیر i, h, g را با فرمت آزاد چاپ می‌کنیم. اینجاست که به اهمیت تخصیص اعداد با دقت بالا پی می‌بریم. به مقدار چاپ شده برای این سه متغیر توجه کنید:

```
1.0000000116860970E-007
1.0000000000000000E-007
1.0000000000000000E-007
```

همان‌طور که مشاهده می‌کنید مقدار g, h به طور دقیق چاپ شد اما i نه! و همین اعداد اضافی که از مرتبه هشتم اعشار به بعد ظاهر شده‌اند، ممکن است که در آینده دردسر ساز شوند. اما برای چاپ اعداد اعشاری به صورت کنترل شده، از فرمت کلی Fn.m استفاده می‌کنیم، که در اینجا F مربوط به فرمت چاپ اعداد اعشاری بوده، n نشان‌دهنده‌ی عرض میدان و m، تعداد ارقام بعد ممیز می‌باشد. مثلا در فرمت f10.4، تعداد ۱۰ کاراکتر برای چاپ یک عدد اعشاری اختصاص می‌یابد که تنها تعداد ۴ رقم بعد از اعشار آن چاپ خواهد شد. به نتیجه‌ی اجرای برنامه برای خط‌های ۱۷ و ۱۸ و ۱۹ از برنامه توجه کنید:

```
    0.0001
____^^^^^^
    0.0001     0.00010000 0.0001
____^^^^^^_____^^^^^^^^^^_^^^^^^
    0.0001     0.00010000 0.0001
____^^^^^^_____^^^^^^^^^^_^^^^^^
```

که دو خط ۱۷ و ۱۸ دقیقا معادل یکدیگرند.
اگر بخواهیم که چند عدد اعشاری را با یک فرمت یکسان چاپ کنیم، می‌توانیم از فرم کلی pFn.m استفاده کنیم، که در اینجا p تعدادی اعدادی است که می‌خواهیم آن‌ها را چاپ کنیم. مثلا در خط ۱۹ برنامه، 3f10.5 به این معنا است که برای هر کدام از ۳ عدد اعشاری e, d, c، عرض میدان ۱۰ کاراکتر را اختصاص داده و تنها تعداد ۵ رقم بعد از ممیز را چاپ کند. یعنی به صورت زیر است:

```
  0.00010   0.00010   0.00010
___^^^^^^^___^^^^^^^___^^^^^^^
```

در خط ۲۱ برنامه، هدفمان این است که عدد اعشاری f در میدانی با ۱۰ کاراکتر، تا ۱۰ رقم بعد اعشار از اعشار چاپ کنیم. دقت کنید که f یک یک عدد ۹ رقمی است و ما می‌خواهیم این عدد را تا ۱۰ رقم بعد از ممیز چاپ کنیم. پس با در نظر گرفتن ممیز اعشار، به یک میدان با عرض حداقل ۲۰ کاراکتر نیاز داریم، در صورتی که در این‌جا عرض میدان ۱۰ است. به همین دلیل نتیجه‌ای که حاصل از اجرای برنامه است به صورت زیر می‌باشد:

```
**********
----------
```

و اگر عرض میدانی برابر با ۲۰ را برای چاپ f با ۱۰ رقم بعد از ممیز اعشار در نظر بگیریم نتیجه صحیح خواهد شد:

```
123456789.0000000000
^^^^^^^^^^^^^^^^^^^^
```

یکی دیگر از فرمت‌هایی که برای چاپ اعداد اعشاری به کار می‌رود، استفاده از فرم کلی En.m است. این نوع فرمت اعداد را به صورت نماد علمی چاپ می‌کند. در این‌جا m , n دو عدد صحیح هستند که اولی عرض میدان را تعیین می‌کند و دومی تعداد رقم‌هایی را نشان می‌دهد که بعد از ممیز چاپ خواهند شد. به نتیجه‌ی اجرای خط‌های ۲۳ و ۲۴ برنامه توجه کنید:

```
0.10E-03
^^^^^^^^
    0.10000E-03
____^^^^^^^^^^^
```

اگردر نتیجه‌ی بالا دقت کرده باشید، قسمت نمایی تنها تا دو رقم بعد بعد از علامت ظاهر شده است. پس بیشینه و کمینه نمایی که می‌تواند به ما نشان دهد ۹۹ و ۹۹- خواهد بود. بنابراین برای این که این محدودیت برای چاپ اعداد را از بین ببریم از فرم کلی‌تر En.mEp استفاده می‌کنیم، که در آن p نشان‌دهنده‌ی تعداد رقم‌های قسمت نمایی است. مثلا در خط ۲۵ از برنامه برای چاپ عدد f، اولا میدانی با عرض ۱۲ کاراکتر در نظر می‌گیریم، ثانیا تعداد ۴ رقم بعد از ممیز مدّ نظر ماست، ثالثا تعداد ۲ رقم را برای قسمت نمایی عدد در نظر می‌گیریم. به نتیجه‌ی اجرای دو خط ۲۵ و ۲۶ برنامه توجه کنید:

```
  0.1235E+09
__^^^^^^^^^^
 0.12346E+00009
_^^^^^^^^^^^^^
```

از این‌جا به بعد می‌خواهیم در مورد آرایه‌ها صحبت کنیم. چیزی که شما به طور تقریبی در تمامی برنامه‌هایتان از آن استفاده خواهید کرد. اما ببینیم آرایه چی هست؟
آرایه یک مجموعه‌ای از داده‌هاست که نوع (type) تمامی این داده‌ها یکسان است. عضوهای یک آرایه هم، شبیه آنچه که در ریاضیات به صورت $A_n, ..., A_3, A_2, A_1$ است، اندیس گذاری می‌شوند. یک متغیر اندیس‌دار در فرترن به صورت مثلا a(8) نشان داده می‌شود که دقیقا معادل A_8 در ریاضیات است. آرایه‌های دو بعدی، در واقع به نوعی ماتریس هستند و از این جهت برای ما فیزیکدانان اهمیت پیدا می‌کند که همیشه به دنبال پیدا کردن ماتریس هامیلتونی در سیستم‌های مختلف فیزیکی هستیم تا به بوسیله‌ی آن به تحلیل آن سیستم فیزیکی بپردازیم. پس بحث در مورد آرایه‌ها را با نحوه‌ی تعریف کردن آن در آغاز برنامه شروع می‌کنیم.
به صورت استاندارد یک آرایه‌ی مثلا ۱ بعدی و یک آرایه‌ی ۲ بعدی را به صورت زیر تعریف می‌شود:

```
integer, dimension(5) :: array1
real, dimension(5,4) :: array2
```

خط اول به ما می‌گوید که اولا داده‌هایی که قرار است در آرایه‌ی array1 ذخیره شود از نوع integer می‌باشد، ثانیا تعداد در این آرایه می‌توانیم ذخیره کنیم ۵ تا است. یعنی به صورت array1(1) , array1(2) , array1(3) , array1(4) , array1(5) . در خط دوم هم، ما یک آرایه‌ی ۲ بعدی تعریف کردیم که نوع داده‌هایی که در آن دخیره می‌شود از نوع اعشاری است. از طرفی این آرایه قابلیت ذخیره کردن ۲۰ داده‌ی اعشاری را دارد. در واقع این آرایه یک ماتریس ۴ × ۵ است که اعضای آن به صورت زیر در این ماتریس قرار می‌گیرند:

array۲(۱،۴)	array۲(۱،۳)	array۲(۱،۲)	array۲(۱،۱)
array۲(۲،۴)	array۲(۲،۳)	array۲(۲،۲)	array۲(۲،۱)
array۲(۳،۴)	array۲(۳،۳)	array۲(۳،۲)	array۲(۳،۱)
array۲(۴،۴)	array۲(۴،۳)	array۲(۴،۲)	array۲(۴،۱)
array۲(۵،۴)	array۲(۵،۳)	array۲(۵،۲)	array۲(۵،۱)

اگر شما خواستید که آرایه‌هایی با ابعاد بالاتر تعریف کنید، کافیست که در داخل پرانتز dimension سه تا عدد را تایپ کنید که هر کدام مشخص کننده‌ی تعداد خانه‌هایی است که در یک بعد خاص برای ذخیره‌ی داده‌ها، اشغال می‌شوند. مثلا:

```
integer, dimension(3,4,5) :: array3
```

نکته‌ای که در این‌جا لازم به ذکر است، آن است که مثلا وقتی می‌نویسیم integer,dimension(5) :: array1، اندیس گذاری عضوهای آرایه به صورت پیش فرض، از عدد ۱ شروع شده و به ۵ ختم می‌شود.
گاهی اوقات شما دوست دارید که این اندیس گذاری را به صورت دلخواه و متناسب با برنامه‌تان انجام دهید، مثلا دوست دارید به جای ۱ تا ۵، اندیس گذاری به صورت ۱ـ تا ۵ـ یا ۵ تا ۱۰ و یا ... باشد. برای این کار کافیست که شما خودتان محدوده‌ی اندیس متغیرها را وارد کنید ولی به صورت زیر:

```
integer, dimension(-5: -1) :: array1
```

و یا

```
integer, dimension(5: 10) :: array1
```

بنابراین تعیین حد پایینی و بالایی آرایه دست شماست. البته توجه کنید که این اعداد بایستی حتما صحیح باشند. پس به طور کلی داریم:

```
INTEGER, DIMENSION(lower_bound: upper_bound) :: ARRAY
```

روش دیگری هم برای تعریف کردن آرایه‌ها در ابتدای برنامه وجود دارد و آن، این است که شما دیگر از کلمه‌ی dimension استفاده نکنید. به مثال‌های زیر توجه کنید:

```
REAL :: ARRAY1(lower_bound: upper_bound), ARRAY2(upper_bound)
INTEGER :: ARRAY3(-10: 10, -13: -7, 86:107)
```

به هر حال هر کدام از این تعریف‌ها، مزیت‌های خاص خود را دارد. مثلا اگر شما خواستید که چندین آرایه با یک نوع خاص و یک حد بالایی و پایینی تعریف کنید، بهترین روش، استفاده از dimension است. مثلا:

```
integer, dimension(-5: 5, -4:4) :: array1, array2, array3, array4
```

که در این مثال ما ۴ آرایه‌ی array4, array3, array2, array1 را همزمان با هم تعریف کردیم. اما اگر بخواهیم از روش دوم استفاده کنیم، باید بنویسیم:

```
integer :: array1(-5: 5, -4:4), array2(-5: 5, -4:4),
           array3(-5: 5, -4:4), array4(-5: 5, -4:4)
```

که همان‌طور که مشاهده می‌کنید، روش اول خیلی جمع و جورتر از روش دوم است و عقل و منطق حکم می‌کند که در این‌گونه موارد از روش اول استفاده کنیم.اما در غیر این صورت به نظر می‌رسد که روش دوم بهتر باشد (به هر حال شما صاحب اختیارید!). حالا بیایید و با این اطلاعات مقدماتی از آرایه‌ها، یک برنامه‌ای بنویسیم که طی آن عضوهای یک آرایه یک بعدی را که مثلا دارای ۵ عضو است، مقداردهی کرده و سپس چاپ کنیم. ابتدایی‌ترین و شاید احمقانه‌ترین روش برای مقداردهی کردن آرایه و سپس چاپ آن، به این صورت باشد که ما تک تک عضوهای یک آرایه را مقداردهی کنیم و سپس آن‌ها را در دستور write قرار دهیم تا کامپیوتر برای ما چاپ کند. یعنی مثلا بنویسیم:

```
array1(1) = 1; array1(2) = 2; array1(3) = 3
array1(4) = 4; array1(5) = 5
```

و سپس

```
write(*, *) array1(1), arrary1(2), ...
```

اگر کمی دقت کنیم، می‌بینیم که حداقل می‌توان در چاپ کردن این آرایه از یک حلقه‌ی do مدد بخواهیم. این طوری کمی برنامه‌مان بهتر می‌شود. پس فعلا این برنامه زیر را در نظر بگیرید:

```
program array_test
   implicit none
   integer :: i
   integer, dimension(1: 5) :: array1
   array1(1)=1; array1(2)=2; array1(3)=3
   array1(4)=4; array1(5)=5
   do i = 1, 5
      write(*, "(5i5)") array1(i)
   enddo
end program
```

اگر برنامه را اجرا کنید، مشاهده می‌کنید که این برنامه، آرایه‌ها را به صورت یک ماتریس ۱ × ۵ چاپ نموده است، یعنی ۵ سطر و ۱ ستون به صورت زیر:

```
    1
    2
    3
    4
    5
```

خب، تا این‌جا توانستیم یک آرایه را مقدار دهی کنیم و آن را چاپ کنیم. بیایید و نحوه‌ی مقداردهی را تغییر دهیم. برای مقداردهی این‌گونه آرایه‌ها می‌توانیم به صورت زیر از یک روش خلاصه شده استفاده کنیم:

```
array1 = (/1, 2, 3, 4, 5/)
```

این نوع مقداردهی را حتی می‌توانید یا در متن برنامه، یا در جایی که نوع آرایه را تعریف می‌کنید از آن استفاده نمایید. پس برنامه را به صورت زیر اصلاح می‌کنیم:

```
program array_test
   implicit none
   integer :: i
   integer, dimension(1: 5) :: array1 = (/1, 2, 3, 4, 5/)
   do i = 1, 5
      write(*, "(5i5)") array1(i)
   enddo
end program
```

گاهی اوقات می‌خواهیم این آرایه را که برنامه‌ی بالا آن را به صورت ستونی چاپ کرد، به صورت سطری و یا یک ماتریس ۵ × ۱ چاپ کند. برای این کار از حلقه‌ی do ضمنی استفاده می‌کنیم. این حلقه‌ی do ضمنی مشابه همان حلقه doای است که قبلا گفتیم، اما با کمی تفاوت در ساختار آن. ساختار کلی یک حلقه‌ی do ضمنی به صورت زیر است:

```
(ITEM, Do_var = initial, final)
```

و یا

```
(ITEM, Do_var = initial, final, STEP)
```

اما استفاده از این نوع حلقه‌ی do باعث اجرای دستورها(مثل خواندن داده‌ها، چاپ آن‌ها، مرتب کردن آن‌ها و ...) به صورت سطری می‌شود، مثلا اگر از آن در دستور `write` استفاده کنیم، خروجی برنامه را به جای ستونی، به صورت سطری چاپ می‌کند و یا اگر آن را دستور `read` مورد استفاده قرار دهیم، داده‌ها را به جای ستونی، به صورت سطری می‌خواند. این توضیح مهم را به شما بدهیم که کلیه‌ی عملیاتی که در فرترن صورت می‌گیرد اعم از خواندن و یا چاپ به صورت ستونی انجام می‌گیرد مگر این که ما دخالتی در آن کنیم و روند آن را به صورت سطری تغییر دهیم. با این توضیحات می‌خواهیم از حلقه‌ی do ضمنی در برنامه‌مان استفاده کرده تا ماتریسی با ۱ سطر و ۵ ستون را چاپ کنیم. بنابراین خط زیر را جایگزین خط‌های ۵، ۶ و ۷ از برنامه می‌کنیم:

```
write(*, "(5i5)") (array1(i), i = 1, 5)
```

که در این‌جا `array1(i)` همان `ITEM` و `i` همان `Do_var` در ساختار کلی حلقه‌ی do ضمنی است. اکنون به نتیجه‌ی برنامه توجه کنید:

```
    1    2    3    4    5
----^----^----^----^----^
```

حالا بیایید و کار را کمی سخت‌تر کنیم و بخواهیم که یک ماتریس مثلا ۴ × ۳ را مقداردهی کرده و آن را در فایل matrix.txt ذخیره کنیم. به نظر می‌رسد که این کار مشابه قسمت قبل باشد. پس برنامه‌ی بالا را به صورت زیر بازنویسی می‌کنیم:

```
program array_test
   implicit none
   integer, parameter :: row = 3, column = 4
   integer :: i, j
   integer, dimension(row: column) :: array2
   open(1, file = "matrix.txt")
   array2 = (/1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12/)
   write(1, "(12i5)") (array2(i, j), j = 1, column), i = 1, row)
end program
```

قبل از این که به کامپایل برنامه بپردازیم، لازم است که در مورد خط سوم توضیحی مختصر بدهیم. اگر در برنامه‌ای که نوشته‌اید تعداد عضوهای آرایه‌تان از اول تا آخر برنامه هیچ تغییری نمی‌کند، شما می‌توانید این تعداد را به صورت پارامتر(parameter) تعریف کرده و در آرگومان مربوط به آرایه به جای عدد، آن‌ها را قرار دهید. مثل همین کاری که ما در همین برنامه انجام دادیم. در خط ۷ برنامه، ما مشابه قسمت قبل به این آرایه مقداردهی کردیم و در خط بعد از آن هم از دو حلقه‌ی تودرتوی ضمنی برای چاپ آرایه کمک گرفتیم. اگر برنامه را کامپایل کنید به یک خطا برخورد می‌کنید که به صورت زیر است:

```
fortcom: Error: array.f90, line7: The shapes of the array
expression do not conform. [ARRAY2]
   array2 = (/1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12/)
---^
```

با توجه به توضیحات ارائه شده در این خطا، به نظر می‌رسد که این نوع نحوه‌ی مقداردهی، مخصوص آرایه‌های یک بعدی است. برای رفع این مشکل از help فرترن کمک می‌گیریم و کلمه‌ی shape را جستجو می‌کنیم. در این جستجو به یک تابع به اسم reshape برخورد می‌کنید که این تابع مسئولیت مقداردهی به آرایه‌ها را بر عهده دارد. ما آن را در برنامه‌مان استفاده می‌کنیم تا با نحوه‌ی کار با آن آشنا شوید. پس خط ۷ برنامه را به صورت زیر اصلاح می‌کنیم:

```
array2 = reshape((/1,2,3,4,5,6,7,8,9,10,11,12/),(/row,column/))
```

که پرانتز دومی عملا اعلام کردن تعداد سطرها و ستون‌های آرایه است. اکنون اگر برنامه را اجرا کنید، نتیجه‌ی ناامید کننده‌ای را مشاهده خواهید کرد:

```
    1    4    7   10    2    5    8   11    3    6    9   12
----^----^----^---^^----^----^----^---^^----^----^----^---^^
```

این نتیجه، احتمالا به دلیل استفاده‌ی نامناسب از حلقه‌ی do ضمنی، ایجاد شده است. همان‌طور که قبلا گفتیم، استفاده از do ضمنی در دستور write باعث چاپ خروجی برنامه به صورت سطری خواهد شد، در صورتی که استفاده از write در حلقه‌ی do معمولی باعث چاپ خروجی برنامه به صورت ستونی می‌گردد. اگر دقت کنید، ما می‌خواهیم که آرایه را با ۳ سطر و ۴ ستون چاپ کنیم، پس باید از تلفیقی از این دو حلقه‌ی do استفاده کرده تا نتیجه‌ی مطلوب را بدست آوریم. بنابراین خط هشتم برنامه را به صورت زیر اصلاح می‌کنیم:

```
do i = 1, row
   write(1, "(12i5)") (arrayw(i, j), j = 1, column)
enddo
```

اگر برنامه را مجدد کامپایل و اجرا کنیم، نتیجه‌ی زیر بدست می‌آید:

```
    1    4    7   10
    2    5    8   11
    3    6    9   12
```

نتیجه خیلی بهتر شد اما هنوز با آن چیزی که می‌خواهیم فاصله دارد. فکر می‌کنید که دلیل در چیست؟ اگر دقت کرده باشید ما در توضیحاتمان اشاره کردیم که در فرترن محاسبات به صورت ستونی صورت می‌گیرد، بنابراین تابع reshape که ما از آن برای مقداردهی استفاده کردیم به ترتیب، چهار مقدار اول، یعنی ۱، ۲، ۳ و ۴ را در ستون اول، چهار مقدار دوم، یعنی ۵، ۶، ۷ و ۸ را در ستون دوم و چهار مقدار سوم، یعنی ۹، ۱۰، ۱۱ و ۱۲ را در ستون سوم قرار می‌دهد. اکنون اگر اصلاح زیر را در برنامه‌مان انجام دهیم، نتیجه‌ی مورد نظر را به دست خواهیم آورد:

```
array2 = ((/1,5,9,2,6,10,3,7,11,4,8,12/),(/row,column/))
```

که نتیجه‌ی اجرای برنامه هم به این صورت است:

```
    1    2    3    4
    5    6    7    8
    9   10   11   12
```

حالا، شاید شما دوست نداشته باشید که در داخل خود برنامه آرایه را مقادیر عضوهای آرایه مشخص شود و می‌خواهید که خودتان از طریق ترمینال آن‌ها را تعیین کنید. برای این کار کافیست که شما خط هفتم از برنامه را حذف کرده و دستور زیر را جایگزین آن نمایید:

```
do i = 1, row
   do j = 1, column
      read(*, *) array2(i,j)
   enddo
enddo
```

نتیجه همان قبلی است. اما همان‌طور که می‌بینید از دو تا حلقه‌ی do برای انجام این کار استفاده شده است، در حالی که ما می‌توانیم این کار را بدون استفاده از حتی یک حلقه‌ی do انجام دهیم. این‌بار، شما بیایید و دستور زیر را به جای دستور بالا قرار دهید:

```
read(*, *) array2
```

برنامه را کامپایل و سپس اجرا کنید. اگر شما این ۱۲ مقدار را به صورت `1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12` در ترمینال تایپ کنید، نتیجه‌ی اجرا به صورت زیر خواهد بود:

```
    1    2    3    4
    5    6    7    8
    9   10   11   12
```

که باز هم دلیل آن است که اعداد خوانده شده، به صورت ستونی در آرایه قرار داده می‌شوند. اما چنانچه شما اعداد را به صورت `1, 5, 9, 2, 6, 10, 3, 7, 11, 4, 8, 12` وارد کنید، نتیجه‌ی مطلوب را به دست خواهید آورد. خب، تا این جای برنامه ما یک ماتریس ۴×۳ را در فایل matrix.txt ذخیره کردیم. حالا بیایید و برنامه‌مان را به گونه‌ای تغییر دهیم که این ماتریس از فایل matrix.txt خوانده شده و به طور صحیح چاپ شود. پس برنامه را به صورت زیر بازنویسی می‌کنیم:

```
program array_test
   implicit none
   integer, parameter :: row = 3, column = 4
   integer :: i, j
   integer, dimension(row: column) :: array2
   open(1386, file = "matrix.txt")
   do i = 1, row
      do j = 1, column
         read(1386, *) array2(i, j)
      enddo
   enddo
   do i = 1, row
      write(*, "(12i5)") (array2(i, j), j = 1, column)
   enddo
end program
```

اگر برنامه را کامپایل کنید، می‌بینید که از نظر برنامه نویسی هیچ مشکلی وجود ندارد. برنامه را اجرا می‌کنیم. خواهید دید که به یک خطا مطابق زیر برخورد می‌کنیم:

```
forrtl: severe (24): end-of-file during read, unit 1386,
file/home/amin/Desktop/array/matrix.txt
```

این خطا به وضوح اعلام می‌کند که در حین خواندن فایلی که آدرس آن واحد ۱۳۸۶ می‌باشد، داده‌های فایل قبل از خواندن آن به طور کامل به اتمام رسیده است. سپس در ادامه آدرس آن فایل را آورده است.
شاید این خطا از این‌جا ناشی شود که ما با فرمت مناسب داده‌های موجود در این فایل را نمی‌خوانیم. پس بیایید و فرمت 12i5 را در دستور read قرار دهیم و ببنیم آیا تغییری حاصل خواهد شد یا نه. اگر برنامه را اجرا کنیم، باز هم به همان خطای بالا برخورد می‌کنیم. پس اشکال از فرمت نیست. شاید اشکال از این باشد که ما از دو تا حلقه‌ی do معمولی برای خواندن فایل استفاده کردیم. به این دلیل که وجود دستور read در این دو حلقه منجر به این خواهد شد که داده‌ها از داخل فایل به صورت ستونی خوانده شوند، بنابراین از این فایل تنها داده‌هایی خوانده می‌شوند که در ستون اول قرار دارند. از آن‌جا که دستور خواندن ۱۲ مرتبه تکرار می‌شود و در فایل تنها ۴ عدد در ستون اول وجود دارد، مسلم است که در حین اجرای برنامه، سر و کله‌ی چنین خطایی پیدا می‌شود. برای رفع این مشکل باز هم از تلفیقی از دو حلقه‌ی do معمولی و ضمنی کمک می‌گیریم، تا عملیات خواندن، هم به صورت سطری و هم به صورت ستونی صورت بگیرد. پس سه خط ۷، ۸ و ۹ را از برنامه حذف می‌کنیم و خط زیر را جایگزین آن می‌کنیم:

```
read(1386, *) (array(i, j), j = 1, column)
```

با انجام این اصلاح در برنامه، نتیجه‌ی صحیح حاصل خواهد شد.
باز هم می‌خواهیم با برنامه‌مان بازی کنیم. این بار دستور مربوط به خواندن داده‌ها از فایل را به صورت زیر می‌نویسیم:

```
read(1386, *) array2
```

برنامه را اجرا می‌کنیم، بدون خطا، کامپایل و اجرا شد!! به نتیجه توجه کنید:

```
    1    4    7   10
    2    5    8   11
    3    6    9   12
```

این خیلی خوبه که برنامه‌مان خلاصه‌تر شده و از شر دو تا حلقه‌ی do هم راحت شدیم، اگر چه نتیجه صحیح نیست ولی آن را براحتی می‌توان با دست کاری در دستور write تصحیح کرد. کافیست دستور چاپ زیر را جایگزین دستور چاپ قبلی کنیم:

```
do j = 1, row
   write(*, "(12i5)") (array2(i, j), i = 1, column)
enddo
```

تنها کاری که کردیم این بود که جای i و j را با هم عوض نمودیم. (حالا فکر می‌کنید که چگونه اعداد از داخل فایل خوانده شده‌اند؟)
تا این جای کار، شما با نحوه‌ی مقداردهی به آرایه، چاپ آرایه و خواندن آن از یک فایل به طور نسبتا کامل آشنا شدید. از این جا به بعد می‌خواهیم در غالب یک سری برنامه‌های کوچک، کمی در مورد جبر بین آرایه‌ها و توابع تعریف شده در فرترن که مربوط به آن هاست، بیشتر آشنا شویم. لذا با برنامه‌ی ساده‌ی زیر شروع می‌کنیم:

```
program array_example
   implicit none
   integer, parameter :: row = 4, column = 4
   integer :: i, j
   integer, dimension(row, column) :: array1
   do i = 1, row
      do j = 1, column
         array1(i, j) = 0
      enddo
   enddo
   do i = 1, row
      write(*, "(16i5)") (array1(i, j), j = 1, column)
   enddo
end program
```

در مثال بالا، ما با استفاده از ۲ تا حلقه‌ی do مقدار صفر را به تمامی عضوهای این آرایه اختصاص دادیم. چنانچه برنامه را اجرا کنید، نتیجه‌ی زیر حاصل خواهد شد:

```
0     0     0     0
0     0     0     0
0     0     0     0
0     0     0     0
```

اما این روش صفر کردن عضوهای آرایه بسیار ابتدایی است. اگر تعداد عضوهای این آرایه بسیار زیاد باشد، وجود این دو تا حلقه‌ی do باعث افزایش مدت زمان اجرای برنامه می‌شود. خوشبختانه در فرترن ۹۰ این کار به سادگی هر چه تمام صورت می‌گیرد. کافیست شما چهار خط ۶، ۷، ۸ و ۹ را از برنامه‌ی بالا حذف کرده و به جای آن تایپ کنید:

```
array1 = 0
```

نتیجه همان قبلی است و این بسیار راضی کننده است. حتی شما می‌توانید به همین طریق به همه‌ی عضوهای آرایه و یا یک تعداد خاصی از آن‌ها مقدار دلخواه خود را نسبت دهید. مثلا اگر شما به جای دستور array1 = 0 قرار دهید:

```
array1(1: 2, 1:4) = 1 ; array1(3: 4, 1:4) = 2
```

دستور اول به ما می‌گوید که در عضوهایی که در کلیه‌ی ستون‌های دو سطر ۱ و ۲ ماتریس قرار دارند مقدار یک تخصیص میابد و در مابقی اعضا مقدار دو. پس برنامه ماتریس زیر را برای شما چاپ خواهد کرد:

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

به عنوان یک سرگرمی خوب، خودتان این آرگومان‌ها را تغییر دهید و ماتریس حاصل را چاپ کنید. یک نکته دیگر هم لازم به ذکر است که درصورتی که ما به آرایه مقداردهی نکنیم، خود فرترن به صورت پیش فرض، مقدار صفر را به هر کدام از عضوهای آرایه تخصیص می‌دهد. می‌توانید این مطلب را به راحتی بررسی کنید. اکنون بیایید و یک آرایه‌ی دو بعدی ۴ در ۴ دیگر به اسم array2 به برنامه‌مان اضافه کنیم. می‌خواهیم به عضوهای این آرایه به ترتیب، مقادیر ۱، ۲، ... و ۱۶ را نسبت دهیم. برای این کار از یک متغیر m که در ابتدای برنامه مقدار ۱ را به آن اختصاص داده‌ایم، کمک می‌گیریم. پس فعلا برنامه را به صورت زیر اصلاح می‌کنیم:

```
program array_example
   implicit none
   integer, parameter :: row = 4, column = 4
   integer :: i, j, m = 1
   integer, dimension(row, column) :: array1, array2
   array1 = 1
   do i = 1, row
      do j = 1, column
         array2(i, j) = m
         m = m+1
      enddo
   enddo
   do i = 1, row
      write(*, "(16i5)") (array2(i, j), j = 1, column)
   enddo
end program
```

نتیجه‌ی اجرا را ببینید:

```
 1    2    3    4
 5    6    7    8
 9   10   11   12
13   14   15   16
```

اکنون می‌خواهیم دو آرایه‌ی array1 و array2 را با هم جمع کنیم و نتیجه را در آرایه‌ی سومی به اسم array3 قرار دهیم. برای این کار اولین ایده‌ای که به ذهنمان می‌خورد، استفاده از دو حلقه‌ی do است که به واسطه‌ی آن عضوهای نظیر به نظیر دو آرایه‌ی array1 و array2 با هم جمع می‌شوند و در آرایه‌ی سوم ذخیره می‌گردند. پس ابتدا array3 را در خط ۴ برنامه تعریف می‌کنیم، سپس بعد از خط ۱۲ تایپ می‌کنیم:

```
do i = 1, row
   do j = 1, column
      array3(i, j) = array1(i, j)+array2(i, j)
   enddo
enddo
```

و آنگاه به جای چاپ array1 و array2، array3 را چاپ می‌کنیم. با انجام این اصلاحات در برنامه، نتیجه چنین خواهد شد:

```
 2   3   4   5
 6   7   8   9
10  11  12  13
14  15  16  17
```

باز هم این یک روش مبتدیانه است. می‌توانیم به جای این که از دو تا حلقه‌ی do برای جمع کردن عضوهای نظیر به نظیر دو آرایه استفاده کنیم به صورت زیر تایپ کنیم که:

```
array3 = array1+array2
```

و اگر array3 را چاپ کنید، نتیجه همان قبلی است.
بیایید و این بار برنامه‌ای بنویسیم که طی آن، مجموع عضوهای آرایه‌ی array3 را محاسبه کنیم. به نظر می‌رسد که این جا هم برای این کار، بایستی از دو تا حلقه‌ی do استفاده کنیم. پس فعلا برنامه‌مان را به صورت زیر بازنویسی می‌کنیم:

```
program array_example
   implicit none
   integer, parameter :: row = 4, column = 4
   integer :: i, j, m = 1, array_sum
   integer, dimension(row, column) :: array1, array2, array3
   array1 = 1
   do i = 1, row
      do j = 1, column
         array2(i, j) = m
         m = m+1
      enddo
   enddo
   array3 = array1+array2
   array_sum = 0
   do i = 1, row
      do j = 1, column
         array_sum = array_sum+array3(i, j)
     enddo
   enddo
   write(*, *) array_sum
end program
```

برنامه را اجرا کنید، نتیجه ای که برای شما چاپ می شود عدد ۱۵۲ خواهد بود. اما باز هم می توانیم برای اجتناب از دو تا حلقه‌ی do، برای جمع عضوهای آرایه از یک تابع به اسم SUM استفاده کنیم. این تابع مجموع عضوهای آرایه را به ما برمی‌گرداند. اکنون خط‌های ۱۴ تا ۱۹ برنامه را حذف کنید و به جای آن تایپ کنید:

```
array_sum = SUM(array3)
```

اگر برنامه را اجرا کنید، همان عدد ۱۵۲ را چاپ خواهد کرد.
این بار می‌خواهیم یک آرایه‌ی دو بعدی اعشاری را تعریف کنیم، به طوری که عضوهای آن مقادیر رندوم یا تصادفی داشته باشد. این مثال را از آن جهت آوردیم که شما در قسمت‌های بعد از اعداد رندوم به وفور استفاده خواهید کرد. در فرترن دستوری به صورت CALL RANDOM_NUMBER(n) وجود دارد که عدد رندوم n را برمی‌گرداند. این عدد مقداری بین صفر و ۱ خواهد داشت. خب، یک راه تولید آرایه‌ای که اعضای آن رندوم باشد استفاده از حلقه‌ی تکرار do به صورت زیر است:

```
do i = 1, row
   do j =1, column
      call random_number(n)
      array4(i, j) = n
   enddo
enddo
```

که array4 به صورت زیر تعریف می‌شود:

```
real, dimension(row, column) :: array4
```

اما ساده‌تر هم می‌شود این کار را انجام داد، کافیست دستور:

```
call random_number(array4)
```

را بعد از خط ۱۲ برنامه و

```
do i = 1, row
   write(*, "(16f7.2)") (array4(i, j), j = 1, column)
enddo
```

را به جای خط ۲۰ برنامه تایپ کنید. آنگاه نتیجه چنین خواهد بود:

```
   0.00   0.96   0.80   0.09
   0.03   0.84   0.83   0.89
   0.35   0.34   0.35   0.70
   0.67   0.92   0.87   0.73
```

حالا بیایید و کمی به ماجراجویی با این ماتریس بپردازیم. اول، برنامه‌ای بنویسیم که مقادیر کمینه و موقعیت آن‌ها در هر ستون این آرایه، را برگرداند و سپس آن‌ها را چاپ کند. دوم، همین کار را برای مقادیر بیشینه تکرار کنیم.
اولی را ما انجام می‌دهیم و دومی به عهده‌ی شما! برای پیدا کردن مقادیر کمینه و موقعیت آن‌ها ابتدا آرایه‌ی یک بعدی location(1: 4) را از نوع صحیح و آرایه‌ی یک بعدی minmimum(1: 4) را از نوع اعشاری تعریف می‌کنیم و بعد از مقداردهی رندوم آرایه‌ی array4، دستورات زیر را به برنامه‌مان اضافه می‌کنیم:

```
minimum = 1
location = 1
do i = 1, row
   do j = 1, column
      if(array4(i, j) < minmum(j)) then
         minimum(j) = array4(i, j)
         location(j) = i
      endif
   enddo
enddo
write(*, "(4f7.2)") (minimum(j), j = 1, column)
print*,"******************************"
write(*, "(4i7)") (location(i), j = 1, column)
print*,"******************************"
```

در این‌جا ما از یک الگوریتم کاملا معروف برای پیدا کردن کمینه استفاده کردیم لذا دلیلی نمی‌بینیم که آن را برای شما توضیح دهیم. اما در مورد مواردی که در دستور write قرار دادیم، باید این مطلب را ذکر کنیم که در دستور:

```
write(*, "(4f7.2)") (minimun(j), j = 1, column)
```

ما مقدار کمینه‌ی هر ستون را چاپ می‌کنیم. به همین دلیل است که در حلقه‌ی do ضمنی نوشته‌ایم: j = 1, column و در دستور

```
write(*, "(4i7)") (location(i), j = 1, column)
```

ما موقعیت مقدار کمینه در ستون zام را چاپ می‌کنیم، یعنی مقادیر ذخیره شده در آرایه‌ی location همان شماره سطرهایی است که مقادیر کمینه در آن ستون‌ها قرار گرفته‌اند. اکنون به نتیجه‌ی برنامه توجه کنید، دقیقا تایید کننده‌ی توضیحات گفته شده است.

```
   0.00   0.34   0.35   0.09
******************************
      1      3      3      1
******************************
   0.00   0.96   0.80   0.09
   0.03   0.84   0.83   0.89
   0.35   0.34   0.35   0.70
   0.67   0.92   0.87   0.73
```

همان‌طور که دیدید، در این برنامه ما از دو تا حلقه‌ی do و یک دستور شرطی if برای رسیدن به هدفمان استفاده کردیم. اگر ماتریس‌هایی با اندازه‌های بزرگتر داشته باشیم، وجود این حلقه‌ها و دستور شرطی if باعث اتلاف وقت بسیاری از ما خواهد شد. خوشبختانه در فرترن ۹۰ باز هم در این زمینه توابعی وجود دارد که دقیقا همان چیزی را که ما با چند خط برنامه نویسی بدست آوردیم را می‌دهد. این توابع به صورت زیر هستند:

```
MINVAL(array[, dim])
MINLOC(array[, dim])
```

اولا نتیجه‌ای که این دو تابع برمی‌گردانند آرایه‌های یک بعدی است ثانیا dim هم باید یک عدد صحیح باشد که مقدار این عدد نباید از بعد آرایه‌ای که در آرگومان این توابع قرار می‌گیرد، بیشتر باشد. در مورد مثال بالا اگر بعد از دستور print*,"**********" تایپ کنیم:

```
write(*, "(4f7.2)") (MINVAL(array4, dim = 1))
write(*, "(4f7.2)") (MINLOC(array4, dim = 1))
```

نتیجه‌ای که برای چاپ می‌کند، همانی است که ما بدست آوردیم. اما چنانچه dim = 2 را قرار دهیم، مقادیر کمینه و موقعیت آن‌ها را در هر سطر از آرایه پیدا کرده و برای ما چاپ می‌کند. اکنون به عنوان تمرین، برنامه‌ی دوم را خودتان نوشته و صحت نتایج خود را با دو تابع MAXVAL و MAXLOC مقایسه کنید.
گاهی اوقات ما نیاز داریم که دو تا ماتریس را در هم ضرب کنیم و یا این که علاقمند به محاسبه‌ی ضرب داخلی دو بردار هستیم. در فرترن توابعی که این دو کمیت را حساب می‌کنند عبارتند از MATMUL و DOT_PRODUCT . که اولی مخصوص ضرب ماتریسی دو آرایه‌ی دو بعدی و دومی هم مخصوص ضرب ماتریسی دو آرایه‌ی یک بعدی است. اکنون می‌خواهیم که ضرب داخلی دو بردار location و minimum ، و سطر اول آرایه‌ی array4 و سطر سوم آرایه‌ی array2 را حساب کنیم و مقدار آن را چاپ کنیم. برای این کار تایپ می‌کنیم:

```
dot_product(location, minimum)
dot_product(array4(1, :), array2(3, :))
```

که برای اولی مقدار 2.13 و برای دومی مقدار 19.46 را چاپ می‌کند. از طرفی اگر بخواهیم حاصل ضرب دو آرایه‌ی array2 و array4 را به دست آوریم، باید تایپ کنیم:

```
matmul(array4, array2)
```

که نتیجه‌ی زیر برای شما حاصل خواهد شد:

```
  3.78   7.31   6.98   6.91
  7.96  19.52  18.36  16.56
 12.14  31.72  29.74  26.22
 16.32  43.93  41.12  35.78
```

حالا می‌خواهیم یک برنامه‌ای بنویسیم که طی آن سه ماتریس قطری ۲ × ۲، ۳ × ۳ و ۴ × ۴ را بسازیم و آن‌ها را با فرمت آزاد در فایل‌های جدا ذخیره کنیم، به طوری که اعداد روی قطر این ماتریس‌ها دلخواه باشد. برای نوشتن این برنامه‌ی ساده اولین چیزی که به ذهنمان می‌خورد آن است که ما سه آرایه با ابعاد مناسب را تعریف کنیم، سپس با استفاده از ۶ تا حلقه‌ی do مقادیر مناسب را به عضوهای این آرایه نسبت داده و پس از آن باز هم با استفاده ۶ تاحلقه‌ی do (سه تا ضمنی و سه تا معمولی) به چاپ آن‌ها بپردازیم. یعنی به صورت زیر:

```
program array_dymnamic
   implicit none
   integer :: i, j, array1(2,2), array2(3,3), array3(4,4)
   array1 = 0; array2 = 0; array3 = 0
   open(2, file = "matrix1.txt")
   open(3, file = "matrix2.txt")
   open(4, file = "matrix3.txt")
   do i = 1, 2
      do j = 1, 2
         if(i == j) array1(i, j) = i
      enddo
```

```
   enddo
   .
   .
   .
   do i = 1, 2
      write(2, *) (array1(i, j), j = 1, 2)
   enddo
   .
   .
   .
end program
```

همان‌طور که دیدید، برای نوشتن این برنامه‌ی ساده ما از تعداد زیادی حلقه‌ی do و علاوه بر آن از ۳ تا آرایه‌ی جداگانه استفاده کردیم. ما می‌توانیم این برنامه را به گونه‌ای بسیار ساده‌تر بنویسیم که هم رضایت شما جلب شود هم ما! اما چون می‌خواهیم که در متن برنامه‌ی اصلاح شده از یک سری دستورات جدید، یعنی آرایه‌های دینامیکی، استفاده کنیم، ابتدا به توضیح آن‌ها می‌پردازیم سپس به اصلاح برنامه‌مان بپردازیم.
آرایه‌های دینامیکی، آرایه‌هایی هستند که اندازه‌ی آن‌ها، پس از انجام یک سری محاسبات در برنامه و یا از طریق داده‌های ورودی به برنامه تعیین می‌شوند. لذا ما از اندازه‌ی آن‌ها در همان ابتدای برنامه آگاهی نداریم. این نکته واقعا مهم است، به این دلیل که اگر قرار بود ما از همان ابتدا اندازه‌ی آرایه را تعیین کنیم، باید آن را به حدی بزرگ در نظر می‌گرفتیم که بعدها در حین اجرای برنامه با خطای اجرایی مواجه نشویم. از طرفی این بزرگ گرفتن اندازه‌ی برنامه منجر به اشغال حافظه‌ی زیادی از رایانه شده و سرعت اجرای برنامه را پایین خواهد آورد. در واقع استفاده از آرایه‌های دینامیکی، به نوعی مدیریت در حافظه‌ی رایانه می‌باشد. لذا در فرترن ۹۰ برای جلوگیری از این مشکلات، تسهیلات لازم در نظر گرفته شده است.
فرم کلی استفاده از آرایه‌های دینامیکی به صورت زیر است:

```
INTEGER, DIMENSION(: , :), ALLOCATABLE :: ARRAY
   ...
ALLOCATE(ARRAY(m:n, p:q))
   ...
DEALLOCATE(ARRAY)
   ...
ALLOCATE(ARRAY(ub1, ub2))
   ...
DEALLOCATE(ARRAY)
   ...
```

در این ساختار کلی، ابتدا نوع آرایه‌ی ARRAY را بدون مشخص کردن اندازه‌ی آن تعریف کرده، سپس با استفاده از دستور ALLOCATE اندازه‌ی آن را تعیین می‌کنیم. به عبارت دیگر، وقتی می‌نویسیم ALLOCATE(ARRAY(1:10, 5:10))، داریم به آرایه‌ی دو بعدی ARRAY تعداد ۱۰ خانه در سطر و ۵ تا هم در ستون برای ذخیره‌ی داده‌ها به آن اختصاص می‌دهیم. اگر در ادامه‌ی کار باز هم خواستیم از این آرایه اما با اندازه‌ای دیگر استفاده کنیم، با استفاده از دستور DEALLOCATE(ARRAY)، اندازه‌ی آرایه را که خودمان به آن تخصیص داده بودیم، رفع تخصیص می‌کنیم (حذف می‌کنیم) و مجدد با استفاده از دستور ALLOCATE اندازه‌ی جدید برای آرایه قرار می‌دهیم. به همین ترتیب ادامه می‌دهیم.
اکنون به برنامه‌ی اصلاح شده توجه کنید:

```
program array_dynamic
   implicit none
   integer :: i, j, N
   integer, dimension(: , :), allocatable :: array
   open(2, file = "matrix1.txt")
   open(3, file = "matrix2.txt")
   open(4, file = "matrix3.txt")
   do N = 2, 4
      allocate(array(N,N))
      array=0
      do i = 1, N
         do j = 1, N
```

```
            if(i == j) array(i, j) = i
         enddo
      enddo
   write(N, *) (array(i, j), j = 1, N)
   enddo
   deallocate(array)
   enddo
end program
```

در این برنامه، به جای این که ما از سه تا آرایه استفاده کنیم، تنها از یک آرایه استفاده نمودیم که با استفاده از دستور allocate اندازه‌ی مورد نظر را به آرایه تخصیص داده‌ایم. مثلا در مرتبه‌ی اول اندازه‌ی آرایه ۲ × ۲ بوده، بعد از انجام محاسبات و چاپ آن، با استفاده از دستور deallocate اندازه‌ی تعیین شده‌ی قبلی را پاک کرده و مجددا اندازه‌ی جدید را برای آرایه تعیین می‌کنیم و به همین صورت همین کار را ادامه داده‌ایم. نتیجه‌ی اجرای برنامه‌ی اصلاح شده، دقیقا همان نتیجه‌ی اجرای برنامه‌ی قبلی است، اما این کجا و آن کجا!!
تا این‌جای کار شما به قدری با آرایه‌ها آشنا شده‌اید که بتوانید آن‌ها را به راحتی در برنامه‌هایتان به خوبی و به جا استفاده کنید. در ادامه می‌خواهیم شما را با زیربرنامه‌ها و توابع که قسمت اعظم و البته مهمی از برنامه‌هایتان را در آینده تشکیل می‌دهد، آشنا کنیم. ابتدا ساختار کلی توابع:
ساختار کلی توابع در فرترن به صورت زیر است:

```
type FUNCTION function_name(arg1, arg2, ..., argn)
   IMPLICIT NONE
   [specification part]
   [execution part]
END FUNCTION function_name
```

همان‌طور که ملاحظه می‌کنید، شروع یک تابع با مشخص کردن نوع تابع آغاز می‌شود. این که این تابع چه نوع مقداری را بر می‌گرداند، صحیح، اعشاری یا .... بعد از تعیین نوع تابع، کلمه‌ی FUNCTION و سپس اسم تابع را مشخص می‌کنیم. آرگومان‌های argn, ..., arg2, arg1 که همگی ورودی‌های تابع هستند، بعد از اسم تابع آورده می‌شوند. در بدنه‌ی تابع هم، دستورات مربوط به تعریف متغیرها، پارامترها، آرگومان‌ها و به دنبال آن دستورات اجرایی قرار می‌گیرد. نکته‌ای که در این‌جا باید به آن توجه شود آن است که تمامی این آرگومان‌ها ورودی تابع هستند و خروجی آن هم، همان نام تابع است که بعد از کلمه‌ی FUNCTION نوشته می‌شود. حسن ختام هر تابع، END FUNCTION و سپس نام تابع بوده که البته در مورد آخری، یعنی اسم تابع، شما صاحب اختیارید که آن را استفاده کنید یا نکنید! در فرترن ۹۰ دو نوع تابع وجود دارد. یکی توابع داخلی و دیگری توابع خارجی. توابع داخلی توابعی هستند که با استفاده از دستور CONTAINS در ساختار کلی برنامه‌ی اصلی قرار می‌گیرند. یعنی به صورت زیر:

```
PROGRAM main
   IMPLICIT NONE
   ......
CONTAINS
   INTEGER, FUNCTION func1(arg1, arg2, ..., argn)
      IMPLICIT NONE
      ......
   END FUNCTION
   REAL, FUNCTION func2(arg1, arg2, ..., argm)
      IMPLICIT NONE
      ......
   END FUNCTION
END PROGRAM
```

و توابع خارجی هم، همان‌طور که از اسمشان پیداست، در خارج از بدنه‌ی اصلی برنامه قرار می‌گیرند. اکنون به مثال ساده‌ی زیر توجه کنید:

```
program func_example
   implicit none
   real :: a, b
   write(*, *) "two real number please -->"
   read(*, *) a, b
   write(*, *) dosomething(a, b)
contains
```

```
   real function dosomething(a, b)
      implicit none
      real :: a, b
      real :: res
      if( a>b )then
         res = sin(a)
      else
         res = sin(b)
      endif
      dosomething = res
   end function dosomething
end program
```

در این برنامه دو عدد a, b از ترمینال خوانده شده و مقدار Sin عدد بزرگتر چاپ می‌گردد. همان‌طور که می‌بینید متغیرهای a, b به عنوان ورودی تابع محسوب می‌شوند. بنابراین در خط ۱۰ برنامه برای این که تاکید کنیم که ما از این دو متغیر به عنوان ورودی تابع dosomething استفاده کرده‌ایم، دستور INTENT(in) را مورد استفاده قرار می‌دهیم. یعنی سطر زیر را جایگزین خط ۱۰ برنامه می‌کنیم:

```
real, intent(in) :: a, b
```

اما استفاده از این دستور مزیت دیگری هم دارد. به این مثال توجه کنید:
اگر شما به جای متغیر res، آرگومان a را قرار دهید و برنامه را اجرا کنید به دو خطا برخورد خواهید کرد که اولین آن به صورت زیر است:

```
fortcom: Error: dosomething.f90, line 12: A dummy argument
with the INTENT(IN) at
atribute shall not appear in a variable definition context. [A]
         a = sin(a)
---------^
```

بنابراین از این خطا این طور نتیجه می‌شود که پارامتر ورودی a که همراه با intent(in) تعریف شده است، نمی‌تواند دستخوش تغییر در تابع شود. پس هنگامی که با استفاده از دستور intent(in) دقیقا تاکید به ورودی بودن یک آرگومان می‌کنیم، از این لحاظ هم خیالمان راحت می‌شود که مقدار این آرگومان در طول اجرای برنامه در تابع تغییر نمی‌کند. توصیه‌ی اکید می‌کنیم که برای جلوگیری از به وجود آمدن مشکلات احتمالی ناشی از تغییر مقدار آرگومان‌های ورودی، حتما از این دستور در هنگام تعریف نوع آرگومان‌ها استفاده کنید.
در استفاده از توابع فراموش نکنید که مقدار محاسبه شده توسط تابع را در نام تابع ذخیره کنید. برای درک مطلب بالا بیایید و خط ۱۶ برنامه را حذف کنید. در این صورت در حین کامپایل، یک هشدار به شما داده خواهد شد:

```
fortcom: Warning: dosomething.f90, line 8: The return value
of this FUNCTION has not been defined. [DOSOMETHING]
   real function dosomething(a, b)
-----------------^
```

پس همیشه مقدار تابع، توسط نام تابع برمی‌گردانده می‌شود و شما بایستی در استفاده از توابع، مقدار محاسبه شده توسط آن‌ها را در نام آن‌ها ذخیره کنید. حالا بیایید و دقت مقدار محاسبه شده توسط تابع را ۲ برابر کنیم، یعنی تابع را به این صورت بازنویسی کنیم:

```
real(8) function dosomething(a, b)
   implicit none
   real(8), intent(in) :: a, b
   real(8) :: res
   if( a>b )then
      res = sin(a)
   else
      res = sin(b)
   endif
   dosomething = res
end function dosomething
```

در این صورت اگر برنامه را کامپایل کنید به دو خطای زیر برخورد خواهید کرد:

```
fortcom: Warning: dosomething.f90, line 6: The type of the
actual argument differs from the type of the dummy argument. [A]
   write(*, *) dosomething(a, b)
---------------------------^
```

```
fortcom: Warning: dosomething.f90, line 6: The type of the
actual argument differs from the type of the dummy argument. [B]
   write(*, *) dosomething(a, b)
-------------------------------^
```

این خطاها به وضوح به ما می‌گوید که نوع آرگومان‌های ورودی تابع، با نوع آن‌ها در برنامه‌ی اصلی متفاوت است. از این دو خطا نتیجه می‌گیریم که بایستی نوع آرگومان‌هایی که در برنامه‌ی اصلی و تابع تعریف می‌کنیم،هم از نظر نوع و هم از نظر دقت با هم همخوانی داشته باشند. لذا برای رفع این مشکل خط ۳ برنامه را با دستور زیر تعویض می‌کنیم:

```
real(8) :: a, b
```

در این حالت، برنامه به خوبی اجرا خواهد شد.
به عنوان حسن ختام در مورد مبحث توابع، برنامه‌ای را برای محاسبه‌ی فاکتوریل عدد n به صورت زیر نوشته‌ایم که تابع به کار گرفته شده در آن، یک تابع داخلی است.

```
program function_example
   implicit none
   integer :: n, r
   write(*, *) "two non-negativeintegers please -->"
   read(*, *) n, r
   write(*, *) n, '!=', factorial(n)
   write(*, *) r, '!=', factorial(r)
contains
   integer function factorial(n)
      implicit none
      integer, intent(in) :: n
      integer :: i, fact
      fact = 1
      do i = 1, n
         fact = fact*i
      enddo
      factorial = fact
   end function factorial
end program
```

از این جا به بعد می‌خواهیم شما را با زیربرنامه (Subroutine) آشنا کنیم. به طور کلی Subroutine قابلیت و کارایی بیشتری نسبت به تابع دارد. دلیل گفته‌مان را در ادامه متوجه خواهید شد. فعلا به ساختار کلی Subroutine توجه کنید:

```
SUBROUTINE subroutine_name(arg1, arg2, ...,argn)
   IMPLICIT NONE
   [Specification part]
   [Execution part]
END SUBROUTINE subroutine_name
```

و یا

```
SUBROUTINE subroutine_name()
   IMPLICIT NONE
   [Specification part]
   [Execution part]
END SUBROUTINE subroutine_name
```

همان‌طور که مشاهده می‌کنید، ساختار کلی یک subroutine مشابه تابع، شامل دو بخش است، بخش اول مربوط به تعریف نوع متغیرها و پارامترها است و بخش دوم هم مربوط به دستورات اجرایی.
subroutine ها هم مثل توابع می‌توانند داخلی یا خارجی باشند. یعنی، یا این که به صورت مستقل در خارج از برنامه‌ی

اصلی آورده می‌شوند و یا این که همراه با دستور CONTAINS در داخل برنامه‌ی اصلی نوشته می‌شوند. اما یک تفاوت مهمی که بین subroutine و تابع وجود دارد آن است که در تابع، تمامی آرگومان‌ها ورودی هستند و تنها یک خروجی وجود دارد که آن، اسم تابع می‌باشد. در حالی که در subroutine شما می‌توانید هر تعدادی که می‌خواهید آرگومان ورودی و خروجی داشته باشید. همچنین تفاوت دیگری که بین تابع و subroutine وجود دارد، در نحوه‌ی استفاده‌ی آن‌ها در برنامه‌ی اصلی است. ما از اسم تابع به عنوان نتیجه‌ی تابع استفاده می‌کنیم در حالی که برای استفاده از subroutine در برنامه‌ی اصلی باید، اولا از دستور

```
CALL subroutine_name(arg1, arg2, ...,argn)
```

یا

```
CALL subroutine_name()
```

استفاده کرده، ثانیا نتیجه‌ی subroutine در آرگومان خروجی آن ذخیره شده است و نه در اسم آن !
اما چند نکته در مورد آرگومان‌های subroutine:

- آرگومان یک subroutine را می‌توان با INTENT(IN)، INTENT(OUT) و یا INTENT(INOUT) اعلام کرد.
- اگر یک آرگومان به عنوان ورودی subroutine باشد، ما برای آن از INTENT(IN) استفاده می‌کنیم. واضح است که استفاده از این دستور باعث می‌شود که ما نتوانیم در داخل subroutine مقدار این آرگومان را تغییر دهیم.
- برای آرگومان‌های دیگر که به عنوان آرگومان‌های خروجی محسوب می‌شوند و نتیجه‌ی subroutine در آن‌ها ذخیره می‌شود، از INTENT(OUT) استفاده می‌کنیم.
- اگر یک آرگومان هم مسئولیت ورودی و هم مسئولیت خروجی subroutine را بر عهده داشته باشد، از INTENT(INOUT) برای آن استفاده می‌کنیم. به مثال زیر دقت کنید:

```
subroutine everything(a, b, c, d, e)
   implicit none
   integer, intent(in) :: a, b
   real, intent(out) :: c, d
   real, intent(inout) :: e
   ......
end subroutine everything
```

در این subroutine، سه آرگومان a، b و e به عنوان ورودی وارد subroutine می‌گردند، بعد از انجام محاسبات لازم، نتیجه در سه آرگومان c، d و e ذخیره شده و در خروجی subroutine قرار می‌گیرند.

اکنون می‌خواهیم برنامه‌ای بنویسیم که:
۱ـ با استفاده از Function، ترکیب دو عدد N و R را حساب کند.
۲ـ با استفاده از subroutine میانگین هارمونیک، میانگین حسابی و میانگین هندسی سه عدد a, b و c را به دست آورد.
فرض بر این است که این ۵ عدد، همگی از ترمینال خوانده می‌شوند. اما قبل از این که شروع به نوشتن برنامه کنیم، لازم است توضیح داده شود که:

- ترکیب N از R

$$\begin{pmatrix} N \\ R \end{pmatrix} = \frac{N!}{R!(N-R)!} \qquad (۱)$$

- میانگین هارمونیک

$$HM = \frac{۳}{\frac{۱}{a} + \frac{۱}{b} + \frac{۱}{c}} \qquad (۲)$$

- میانگین حسابی

$$AM = \frac{a+b+c}{۳} \qquad (۳)$$

- میانگین هندسی

$$GM = \sqrt[۳]{abc} \qquad (۴)$$

در نوشتن این برنامه ما از توابع و subroutine خارجی استفاده می‌کنیم. بنابراین:

```
program subfun_example
   implicit none
   integer :: N ,R
   real :: a, b, c, AM, GM, HM
   write(*, *) 'two non-negative integers<10 please -->'
   read(*, *) N, R
   write(*, *) 'three real numbers please -->'
   read(*, *) a, b, c
   write(*, 'i3, a2, i7') N, '!=', factorial(n)
   write(*, 'i3, a2, i7') R, '!=', factorial(r)
   if(R <= N) then
      write(*, 100) 'C(', N, ',', R, ')=', combin(N, R)
   else
      write(*, 100) 'C(', R, ',', N, ')=', combin(R, N)
   endif
   call mean(a, b, c, AM, GM, HM)
   write(*, '(a19, f6.2)') 'arithmatic average=', AM
   write(*, '(a19, f7.2)') 'geometric average=', GM
   write(*, '(a19, f8.2)') 'harmonic average=', HM
   100 format(a2, i3, a1, i3, a2, i7)
end program subfun_example
!*****************************************************
subroutine mean(x, y, z, AM, GM, HM)
   implicit none
   real, intent(in) :: x, y, z
   real, intent(out) :: AM, GM, HM
   real :: d
   AM = (x+y+z)/3.0
   GM = (x*y*z)**(1.0/3.0)
   d = (1.0/x+1.0/y+1.0/z)
   HM = 1.0/d
end subroutine mean
!*****************************************************
integer function factorial(n)
   implicit none
   integer, intent(in) :: n
   integer :: i, fact
   fact = 1
   do i = 1, n
      fact = fact*i
   enddo
   factorial = fact
end function factorial
!*****************************************************
integer function combin(N, R)
   implicit none
   integer, intent(in) :: N, R
   integer :: Cnr
   if(R >= 0 .and. R <= N) then
      Cnr = factorial(N)/factorial(r)/factorial(N-R)
   else
      Cnr = 0
   endif
   combin = Cnr
end function combin
```

به این دلیل در این برنامه از توابع factorial و combin استفاده کردیم که بتوانیم نحوه‌ی استفاده‌ی این توابع را به عنوان توابع خارجی به شما نشان دهیم. تا این‌جای کار، تنها تفاوتی که در مورد چگونگی استفاده از این توابع در برنامه‌ی اصلی ظاهر شده است، حذف شدن دستور contains از برنامه‌ی اصلی و قرار گرفتن این دو تابع بعد از آن می‌باشد. از طرفی subroutine mean هم مسئولیت محاسبه‌ی میانگین‌های حسابی، هندسی و هارمونیک را بر عهده دارد که این مقادیر را به ترتیب در آرگومان‌های AM، GM و HM ذخیره می‌کند.
خب، برنامه را کامپایل می‌کنیم. به ۳ تا خطا به صورت زیر برخورد می‌کنیم:

```
fortcom: Error: mainprogram.f90, line 9: This name does not
have a type, and must have an explicit type. [FACTORIAL]
   write(*, 'i3, a2, i7') N, '!=', factorial(N)
-----------------------------------^


fortcom: Error: mainprogram.f90, line 12: This name does not
have a type, and must have an explicit type. [COMBIN]
   write(*, 100) 'C(', N, ',', R, ')=', combin(N, R)
----------------------------------------^


fortcom: Error: mainprogram.f90, line 50: This name does not
have a type, and must have an explicit type. [FACTORIAL]
   Cnr = Cnr = factorial(N)/factorial(r)/factorial(N-R)
---------------^
```

این سه خطا از عواقب خارج کردن دو تابع factorial و combin از برنامه‌ی اصلی است. همان‌طور که از دو خطای اول به خوبی مشخص است، ما بایستی که نوع تابع‌های خارج شده از برنامه‌ی اصلی را در متن تعیین کنیم. در مورد خطای سوم هم باید این را بگوییم که چون در تابع combin از تابع factorial استفاده شده است، باید نوع این تابع را در تابع combin تعریف کنیم. پس برای اصلاح، کافیست factorial و combin را در خط سوم و factorial را در خط ۴۸ برنامه اضافه کنیم. با این کار برنامه بدون خطا کامپایل و اجرا می‌شود.
نکته‌ی آخر که در مورد subroutine و function ناگفته باقی مانده است، مربوط به استفاده از آن‌ها در آرایه‌ها است. برای استفاده از آرایه‌ها به عنوان آرگومان، چه ورودی و چه خروجی، نباید اندازه آرایه را نوشت. به مثال زیر توجه کنید:

```
subroutine alaki(arrayin, arrayout)
   implicit none
   integer, intent(in) :: arrayin(10, 20)
   real ,intent(out) :: arrayout(5, 10)
   ........
end subroutine alaki
```

ما فعلا به همین مثال‌های ساده بسنده می‌کنیم و بحث پیرامون توابع و زیرروال‌ها را به پایان می‌بریم. دو مورد دیگر پیرامون کدهای فرترن باقی‌مانده است که پس از یاد گرفتن آن‌ها، شما دیگر می‌توانید به راحتی از فرترن در انجام کارهای محاسباتی‌تان استفاده کنید. این دو مورد عبارتند از MODULE و INTRFACE. ابتدا از مدول شروع می‌کنیم.
ساختار کلی یک مدول به صورت زیر است:

```
MODULE module_name
   IMPLICIT NONE
   [Specification Part]
CONTAINS
   [Internal Function_1]
   [Internal Function_2]
   .
   .
   .
   [Internal Function_n]
END MODULE module_name
```

همان‌طور که مشاهده می‌کنید، ساختار یک مدول تقریبا شبیه ساختار یک برنامه است. یعنی هر مدول با کلمه‌ی MODULE شروع شده و با END MODULE هم خاتمه می‌یابد. البته این اختیاریست که نام مدول را بعد از END MODULE بیاوریم. نکته‌ی مهم در مورد مدول‌هااین است که در آن‌ها هیچ دستور اجرایی وجود ندارد، لذا مدول‌ها هیچ نتیجه‌ای را برنمی‌گردانند. بنابراین از یک مدول باید وقتی استفاده شود که با مدول‌های دیگر و علی الخصوص برنامه‌ی اصلی همراه باشد. به مثال‌های زیر در مورد مدول‌ها توجه کنید:

- مثال ۱

```
module someCTE
   implicit none
   real, parameter :: pi = 3.141592
   real, parameter :: g = 9.8
   integer :: counter
end module someCTE
```

همان‌طور که در این مثال ملاحظه می‌کنید، یک مدول می‌تواند فقط شامل تعریف نوع پارامترها و یا متغیرها باشد.

- مثال ۲

```
module fact_comb
   implicit none
contains
   integer function factorial(n)
      implicit none
      integer, intent(in) :: n
      integer :: i, fact
      fact = 1
      do i = 1, n
         fact = fact*i
      enddo
      factorial = fact
   end function factorial
   integer function combin(N, R)
      implicit none
      integer, intent(in) :: N, R
      integer :: Cnr
      if(R >= 0 .and. R <= N) then
         Cnr = factorial(N)/factorial(r)/factorial(N-R)
      else
         Cnr = 0
      endif
      combin = Cnr
   end function combin
end module fact_comb
```

این مثال نشان می‌دهد که یک مدول می‌تواند شامل توابع داخلی باشد.

- مثال ۳

```
module someCTE
   implicit none
   real, parameter :: pi = 3.141592
   real, parameter :: g = 9.8
   integer :: counter
   integer function factorial(n)
      implicit none
      integer, intent(in) :: n
      integer :: i, fact
      fact = 1
      do i = 1, n
         fact = fact*i
      enddo
      factorial = fact
   end function factorial
end module someCTE
```

که در این مثال یک مدول می‌تواند هم شامل توابع باشد و هم شامل برخی تعاریف مربوط به متغیرها و پارامترها.

اما برای این که شما بتوانید از توابع، پارامترها و یا متغیرهای موجود در یک مدول، در هر برنامه یا مدول‌های دیگر استفاده کنید، باید به صورت زیر عمل نمایید:

```
USE module_name
```

و یا

```
USE module_name ,ONLY: name_1, name_2, ..., name_N
```

این دو مورد را با مثال‌های ارائه شده در متن توضیح می‌دهیم.
مثال ۱)

```
module someCTE
   implicit none
   real, parameter :: pi = 3.141592
   real, parameter :: g = 9.8
   integer :: counter
end module someCTE
program main
   implicit none
   use someCTE
   counter = 1386
   write(*, *) pi, g, counter
end program
```

ما در این‌جا مدول را قبل از برنامه‌ی اصلی قرار دادیم. بعدا در مورد مکانی که مدول باید در آن‌جا قرار بگیرد هم بحث خواهیم کرد. فعلا، در این برنامه برای استفاده از مدول، دستور use someCTE را بعد از implicit none در خط ۹ برنامه تایپ کردیم. اگر برنامه را کامپایل کنیم، ۳ خطا حاصل خواهد شد، که اولین آن چنین است:

```
fortcom: Error: module_example.f90, line 9: This USE statement
is not positioned correctly within the scoping unit.
   use someCTE
---^
```

این خطا به وضوح به ما می‌گوید، مکانی که ما دستور use someCTE را قرار داده‌ایم، اشتباه است. پس بیایید و این دستور را به قبل از دستور implicit none تبعید کنیم! با این کار برنامه بدون خطا کامپایل و اجرا می‌شود. اکنون بیایید و مدول someCTE را بعد از برنامه‌ی اصلی قرار دهیم. با انجام این کار، وقتی که برنامه را کامپایل می‌کنیم، با خطای زیر مواجه خواهیم شد:

```
fortcom: Error: module_example.f90, line 8: Declaration of
module 'someCTE' comes after the use of that model. A module's
declaration must precede its use.
```

این خطا هم به وضوح به ما می‌گوید که اعلان مدول، باید جلوتر از استفاده‌ی از مدول واقع شود و نه بعد از آن. پس باید مدول را قبل از برنامه‌ی اصلی قرار داد و نه بعد از آن!
گاهی اوقات شما می‌خواهید که از یک سری پارامترها، متغیرها و یا یک سری توابع خاص که در مدول شما وجود دارند، استفاده کنید. برای این کار کافیست که بعد از دستور use module_name مواردی را که می‌خواهید انتخاب کنید. البته با رعایت قوانین و هنجارهای اجتماعی فرترن ۹۰ !، یعنی استفاده از دستور ONLY. برای توضیح این مطلب، همان مثال قبلی را در نظر بگیرید. این بار می‌خواهیم، تنها پارامتر pi و متغیر counter از مدول را در برنامه‌ی اصلی مورد استفاده قرار دهیم. به همین دلیل خط ۸ برنامه را به صورت زیر بازنویسی می‌کنیم:

```
use someCTE, ONLY: pi, counter
```

با کامپایل برنامه به خطای زیر برخورد می‌کنیم:

```
fortcom: Error: module_example.f90, line 11: This name does
not have a type, and must have an explicit type. [G]
   write(*, *) pi, g, counter
-------------------^
```

این خطا به ما نشان می‌دهد که پارامتر g که در مدول تعریف شده است در برنامه‌ی اصلی بدون نوع می‌باشد. بنابراین تنها پارامتر pi و متغیر counter، از برنامه‌ی اصلی مورد استفاده قرار گرفته‌اند و نه پارامتر g.
اما نکته‌ی دیگر در مورد مدول‌ها این که ما می‌توانیم subroutine ها را نیز در مدول‌ها به کار ببریم. لازمه‌ی آن، استفاده از دستور INTERFACE می‌باشد که بعد از توضیح این دستور، نحوه‌ی کاربرد subroutine در مدول را به شما یاد خواهیم داد. مورد آخر این که معمولا برنامه‌ی اصلی و مدول‌ها را در فایل‌های جداگانه قرار می‌دهند. مثلا برنامه‌ی اصلی در فایل main.f90 و مدول‌ها در فایل mymodules.f90. در این حالت دیگر نحوه‌ی کامپایل کردن این فایل‌ها اهمیت پیدا می‌کند. توضیح این مطلب را می‌گذاریم برای وقتی که می‌خواهیم نحوه‌ی ساخت makefile را به شما یاد دهیم. اکنون می‌پردازیم به توضیح پیرامون دستور INTERFACE:
گاهی اوقات، توابع موجود در برنامه، توابع خارجی هستند که در داخل برنامه‌ی اصلی قرار نمی‌گیرند. در این موارد، برنامه یا توابعی که از این گونه توابع، استفاده می‌کنند باید یک بلوک INTERFACE داشته باشند. در واقع با این دستور دیگر لازم نیست که در برنامه‌ی اصلی، نوع توابع را مشخص کنیم. ساختار کلی INTERFACE به صورت زیر است:

```
INTRFACE
   type FUNCTION function_name1(arg1, arg2, ..., argn)
      type, intent(in) :: arg1
      .
      .
      .
      type, intent(in) :: argn
   END FUNCTION function_name
   .
   .
   .
   ......other Functins......
END INTERFACE
```

همان‌طور که در ساختار کلی ملاحظه می‌کنید، یک بلوک interface، با کلمه interface آغاز شده و با end interface خاتمه می‌یابد. بلوک interface، شامل تیتر توابع و تعریف آرگومان‌های تابع است. دقت کنید! گفتیم که فقط آرگومان‌های تابع، نه سایر متغیرهایی که در تابع مورد استفاده قرار می‌گیرند. مثلا:

```
interface
   integer function factorial(n)
      integer, intent(in) :: n
   end function factorial
   integer function combin(n, r)
      integer, intent(in) :: n, r
   end function combin
end ifnterface
```

اما برای این که شما بتوانید subroutine ها را نیز وارد مدول دلخواه خود کنید، بایستی از بلوک interface استفاده کنید. مثلا:

```
module alaki
   inteface
      subroutine salam(a, b, c, d)
         implicit none
         real, intent(in) :: a, b
         real, intent(out) :: c
         real, intent(inout) :: d
      end subroutine salam
   end interface
end module
```

حالا بیایید و برنامه‌ی subfun_example را با مدول و interface بازنویسی کنیم. پس ابتدای برنامه لازم است که مدول زیر را تایپ کنید:

```
module myutils
   inteface
      subroutine mean(x, y, z, AM, GM, HM)
         implicit none
         real, intent(in) :: x, y, z
         real, intent(out) :: AM, GM, HM
      end subroutine mean
      integer function factorial(n)
         implicit none
         integer, intent(in) :: n
      end function
      integer function combin(n, r)
         integer, intent(in) :: n, r
      end function combin
   end interface
end module
```

اما برای استفاده از این مدول در برنامه‌ی اصلی دستور `use myutils` را قبل از `implicit none` در برنامه‌ی اصلی قرار دهید. بنابراین دیگر لازم نیست که ما نوع تابع‌های مورد استفاده را در داخل برنامه‌ی اصلی در داخل برنامه‌ی اصلی تعریف کنیم، لذا factorial و combin را که از نوع `integer` در برنامه‌ی اصلی تعریف کرده بودیم، حذف می‌کنیم. از طرفی در تابع combin ما از تابع factorial استفاده کردیم که در آن‌جا مجبور شدیم که نوع تابع factorial را در تابع combin تعریف کنیم. اکنون اگر در این تابع، قبل از `implicit none` تایپ کنیم:

```
use myutils, ONLY: factorial
```

دیگر لازم نیست که نوع factorial را در تابع combin تعریف کنیم، لذا دراین تابع و در تعریف نوع متغیرها، تنها متغیر Cnr را به عنوان `integer` و آرگومان‌های n و r را به عنوان `integer, intent(in)` معرفی می‌کنیم. اکنون اگر برنامه را کامپایل و اجرا کنید، همان نتیجه‌ی قبلی را به دست خواهید آورد.
اگر توجه کرده باشید، این برنامه چیزی در حدود ۷۰ خط شده است، در حالی که یک برنامه‌ی خوب، برنامه‌ای است که شما بتوانید تمامی آن را در صفحه‌ی مانیتور ببینید. بنابراین، بهتر آن است که به فایل بندی این برنامه‌ی نسبتا طولانی بپردازیم. ما مدول myutils را در فایل mymodule,f90، زیرروال mean را در mysubroutine.f90 و دو تابع factorial و combin را در فایل myfunctions.f90 ذخیره کردیم. این فایل‌ها به همراه برنامه‌ی اصلی که در فایل mainprogram.f90 می‌باشد را در یک folder به اسم mydocument قرار می‌دهیم. اکنون بیایید و برنامه را کامپایل کنیم. پس در صفحه‌ی ترمینال تایپ کنید:

```
ifrot mainprogram.f90
```

این دستور ۳ خطا را برای شما به ارمغان می‌آورد! که اولین آن به صورت زیر است:

```
fortcom: Error: mainprogram.f90, line 2: Error in openning the
Library module file. [MYUTILS]
   use myutils
-------^
```

این خطا به ما می‌گوید که فایل مدول استفاده شده در برنامه‌ی اصلی را نتوانسته پیدا کند. با این حساب، ابتدا بایستی که فایل mymodule.f90 اجرا شود تا فایل اجرایی آن ایجاد گردد و بدین ترتیب شرایط لازم برای استفاده از مدول در برنامه‌ی اصلی حاصل خواهد شد. لذا تایپ کنید:

```
ifort mymodule.f90
```

باز هم با مشکل مواجه می‌شوید، این بار خطایی که به کار شما گرفته می‌شود به صورت زیر است:

```
/opt/intel/fc/9.0/lib/for_main.o: In function 'main':
./src/libfor/for_main.c:(.text+0x39): undefined refrence to
'MAIN___'
```

احتمالا مشکل از آن‌جا ناشی می‌شود که ما مرجع اصلی را هنوز مشخص نکرده‌ایم. پس فعلا بیایید و تنها به کامپایل mymodule.f90 بپردازیم. یعنی:

```
ifort -c mymodule.f90
```

با این کار دو فایل mymodule.o و myutils.mod بوجود می‌آید. در ادامه هم می‌خواهیم، یکی یکی فایل‌ها را فقط کامپایل کنیم تا در نهایت با لینک کردن آن‌ها به هم، فایل اجرایی برنامه را تشکیل دهیم. پس در صفحه‌ی ترمینال تایپ کنید:

```
ifort -c mysubroutine.90 myfunctions.f90 mainprogram.f90
```

اکنون اگر دستور `ls -t *.o` را بزنید، ۴ فایل آبجکت مربوط به ۴ فایل کامپایل شده را مشاهده خواهید کرد. حالا به لینک کردن آن‌ها می‌پردازیم:

```
ifort mysubroutine.90 myfunctions.f90 mymodule.f90
mainprogram.f90 -o akbar.exe
```

و بالاخره بعد از کلی تلاش و کوشش، فایل اجرایی akbar.exe برای اجرای برنامه حاصل شد.
از آن‌جا که نوشتن ifort همزمان کار کامپایل کردن و ساختن فایل آبجکت را بر عهده دارد، شما می‌توانستید از همان ابتدا تایپ کنید:

```
ifort mysubroutine.90 myfunctions.f90 mainprogram.f90
mymodule.f90 -o akbar.exe
```

و با نوشتن `./akbar.exe` برنامه برنامه اجرا خواهد شد. اما همان‌طور که می‌بینید، یک قطاری از کلمات را پشت سرهم ردیف کردیم تا تک‌تک فایل‌ها کامپایل و سپس اجرا شود. ما می‌توانیم تمامی این کارها را با استفاده از makefile با سادگی هر چه تمام انجام دهیم!
برای ساختن یک makefile، بایستی که در همان folderای که قرار دارید یک فایل با نام makefile بسازید. تاکید می‌کنیم، دقیقا با نام makefile. درmakefile اولین کاری که می‌کنیم آن است که یک سری تارگت (target) را معرفی و سپس نحوه‌ی ساخته شدن آن‌ها را مشخص می‌کنیم. پس فعلا در makefile تایپ کنید:

```
mainprogram.o:
	ifort -c mainprogram.f90
mysubroutine.o:
	ifort -c mysubroutine.f90
mymodule.o:
	ifort -c mymodule.f90
myfunctions.o:
	ifort -c myfunctions.f90
```

در این‌جا منظور از تارگت، همان mainprogram.o، mysubroutine.o، mymodule.o و myfunctions.o است که در ابتدای هر سطر و قبل از ': ' قرار گرفته‌اند. به اسم این نوع تارگت‌ها که در آن‌ها کامپایل فایل‌هاصورت می‌گیرد، دقت کنید. حتما باید به صورت NAME.o باشند و نه به صورت مثلا NAME. از طرفی منظور از نحوه‌ی ساخته شدن آن‌ها همان کامپایل کردن برنامه‌ی source مربوط به آن‌ها می‌باشد. نکته‌ی دیگر آن که سطر بعد از هر تارگت باید ۸ کاراکتر از ابتدای خط فاصله داشته باشد که این فاصله میزان فاصله بایستی فقط و فقط با دکمه‌ی Tab ایجاد شود و نه با دکمه‌ی دیگری مثل Space. در غیر این صورت شما در کامپایل با خطا مواجه خواهید شد.
اکنون در ترمینال بزنید:

```
make mainprogram
```

با این کار فایل آبجکت mainprogram.o حاصل خواهد شد. برای اطمینان خودتان چک کنید!
بنابراین تایپ موارد:

```
make mymodule
make mysubroutine
make myfunctions
```

در ترمینال منجر به ساخته شدن فایل‌های آبجکت آن‌ها خواهد شد. اکنون تنها کاری که باید بکنیم، لینک کردن آن‌هاست، که کافیست مثل قبل عمل کنیم:

```
ifort mysubroutine.90 myfunctions.f90 mainprogram.f90
mymodule.f90 -o akbar.exe
```

و ... برنامه اجرا خواهد شد. اما چه فرقی کرد؟! تقریبا همان مقدار مشقت و سختی عایدمان شد. دلیل این است که ما به احمقانه‌ترین روش ممکن برای شما یک makefile نوشتیم! بیایید و کمی به بهبود سر و وضع آن بپردازیم. اولین کاری که می‌کنیم این است که یک تارگت دیگر به اسم clean به makefile اضافه می‌کنیم تا در صورت لزوم به تمیز کردن داخل folder بپردازیم:

```
clean:
        rm *~ *.mod *.o *.exe -f
```

دستور rm -f، همان دستور پاک کردن است که در این‌جا از آن در خواست کرده‌ایم که فایل‌های exe. ,o. ,mod. ,~ را از داخل folder پاک کند. اگر در ترمینال تایپ کنید make clean و سپس از داخل folder لیست بگیرید، به کارایی آن پی خواهید برد. حالا می‌خواهیم یک تارگت دیگر به اسم all بسازیم که به کمک آن، فایل‌های آبجکت شده را به هم لینک کرده، فایل اجرایی akbar.exe را بوجود آوریم. پس در خط اول تایپ می‌کنیم:

```
all:
        ifort mymodule.o mainprogram.o mysubroutine.o
              myfunctions.o -o akbar.exe
```

البته این را نیز متذکر شویم که در این نوع تارگت‌ها شما می‌توانید هر اسمی را که دوست دارید به جای all استفاده کنید. اکنون اگر در ترمینال تایپ کنید make all، سر و کله‌ی یک خطا به صورت زیر پیدا خواهد شد:

```
IPO link: can not find "mymodule.o"
ifort: Error: problem during multi-file optimization
compilation.
```

این خطا اشاره می‌کند که نتوانسته است که فایل آبجکت mymodule.o را پیدا کند و به تبع آن امکان کنار هم قرار دادن فایل‌ها و لینک کردن آن‌ها برای کامپایلر به وجود نیامده است. برای رفع این مشکل، باید در جلوی تارگت all تایپ کنیم: mymodule.o. با این کار، در واقع اعلام می‌کنیم که فایل آبجکت mymodule.o برای لینک کردن آماده است. حالا در ترمینال، دستور make clean و سپس make all را بنویسید... حال، به آنچه که صفحه‌ی ترمینال به شما نشان می‌دهد توجه کنید:

```
ifort -c mymodule.f90
ifort mymodule.o mainprogram.o mysubroutine.o
myfunctions.o -o akbar.exe
IPO link: can not find "mainprogram.o"
ifort: Error: problem during multi-file optimization
compliation.
```

همان‌طور که مشاهده می‌کنید، اضافه کردن mymodule.o در جلوی تارگت all باعث شد تا به طور خودکار فایل mymodule.f90 کامپایل شود. اما در ادامه، باز هم به یک خطایی دقیقا مشابه خطای قبلی برخورد می‌کنید. این بار به فایل آبجکت mainprogram.o گیر داده شده است. برای رفع آن مشابه قبل عمل می‌کنیم. به خاطر این که دیگر با این قبیل از خطاها مواجه نشویم، در جلوی تارگت all تایپ می‌کنیم:

```
all: mymodule.o mainprogram.o mysubroutine.o myfunctions.o
```

به نظر می‌رسد که تا این جای کار، خطایی در کار نباشد. اما اگر در ترمینال make all را بزنیم، این بار به ۳ خطای دیگر برخورد می‌کنیم، که اولین آن به صورت زیر است:

```
fortcom: Error: mainprogram.f90, line 2: Error in opennig the
Library module file. [MYUTILS]
    use myutils
--------^
```

ما قبلا هم با این خطا مواجه شده بودیم. اگر یادتان باشد برای رفع این مشکل مجبور شدیم که مدول را قبل از برنامه‌ی اصلی قرار دهیم تا ابتدا مدول کامپایل شود سپس برنامه‌ی اصلی (به هر حال برنامه‌ی اصلی وابسته به مدول است). اما در این‌جا که مدول و برنامه‌ی اصلی در فایل‌های جداگانه قرار دارند، چه کنیم؟
برای رفع این مشکل باید کاری بکنیم که ابتدا فایل mymodule.f90 کامپایل شده تا فایل آبجکت آن ایجاد شود، سپس کامپایل mainprogram.f90 آغاز شود. لذا در جلوی تارگت mainprogram.o ، mymodul.o را قرار دهید. یعنی:

```
mainprogram.o:  mymodule.o
        ifort -c mainprogram.f90
```

از آن جا که می‌دانیم، تابع combin هم به مدول myutils وابسته است، در جلوی تارگت myfunctions.o هم mymodule.o را تایپ می‌کنیم. حالا اگر تایپ کنید `make all` و سپس `./akbar.exe` برنامه برای شما اجرا خواهد شد. دیدید! تنها با ۲ دستور.!
اگر چه makefileمان، بهتر شده اما هنوز ناقص است. به این دلیل که اگر ما تغییراتی را در متن اصلی برنامه‌ها ایجاد کنیم، این تغییرات در کامپایل و اجرای برنامه هیچ اثری نخواهد داشت. چک کردن این مطلب به عهده‌ی شما و ارائه‌ی راه حل این مشکل از ما! برای رفع این مشکل ما بایستی در جلوی تارگت‌ها(یعنی در جایی که ما وابستگی‌ها را مشخص می‌کنیم)، نام فایل اصلی یا source آن تارگت را نیز بنویسیم. مثلا:

```
mainprogram.o: mainprogram.f90  mymodule.o
        ifort -c mainprogram.f90
```

خط بالا این را به ما می‌گوید که وابستگی به گونه‌ای است که اولا، فایل mymodule.f90 کامپایل شده و فایل آبجکت آن ایجاد شده باشد. ثانیا، اگر تغییری در برنامه‌ی اصلی ایجاد کردیم، در کامپایل برنامه و اجرای آن در نظر گرفته شود. به عبارت دیگر خطای بالا این مفهوم را به ما می‌رساند که mainprogram.o وابسته به برنامه‌ی mainprogram.f90 و mymodule.o می‌باشد. پس به طور کلی هر چی تارگت از نوع NAME.o داریم، به فایل source خودش وابسته است.
گاهی اوقات لازم است که برنامه را با استفاده از یک کامپایلر دیگر کامپایل کنیم، بنابراین لازم است که در تمام makefile اسم کامپایلر را عوض کنیم. ولی می‌توان این کار را راحت‌تر انجام داد. به این صورت که در همان ابتدای makefile می‌نویسیم:

```
FC = ifort
```

و در جایی که قرار است، برنامه کامپایل شود به جای `ifort` از `$(FC)` استفاده می‌کنیم. چنانچه شما بخواهید مثلا کامپالر f95 را به کار ببرید، کافیست که آن را با ifort در خط اول makefile عوض کنید. `FC` صرفا یک اسم یا علامت اختصاری است و نه چیز دیگر! همان طور که دیدیم، در جلوی تارگت all و خط بعد از آن، ما تعداد زیادی کلمه را به دنبال هم آوردیم، که نوشتن آن‌ها چنانچه باز هم بخواهیم در جای دیگری از makefile استفاده کنیم، زحمت‌آور است. برای سادگی در این کار، می‌توانیم به صورت زیر عمل کنیم:

```
mysubfun = mysubroutine.o myfunctions.o
myobjs = mymodule.o mainprogram.o
```

این دو تا خط را بعد از سطر اول (`FC=ifort`)، قرار می‌دهیم. با استفاده از این ۳ دستور، تارگت all را به صورت زیر بازنویسی می‌کنیم:

```
all: $(myobjs) $(mysubfun)
        $(FC) $(myobjs) $(mysubfun) -o akbar.exe
```

در این دو خط، منظور از `$(FC)` همان `ifort`، `$(myobjs)` دو فایل آبجکت mainprogram.o و mymodule.o و منظور از `$(mysubfun)`، فایل‌های myfunctions.o و mysubroutine.o می‌باشد.
همان طور که دیدید ما برای ساخت هر کدام از تارگت‌ها، مجبور بودیم که از دستور `ifort -c` استفاده کنیم. مثلا برای ساخت تارگت mymodule.o از دستور `ifort -c mymodule.f90` استفاده کردیم. خب، چنانچه تعداد این تارگت‌ها زیاد باشد، این کار هم طاقت فرسا خواهد شد. برای رفع این مشکل تدبیر زیر وجود دارد. قبل از تارگت all تایپ کنید:

```
.SUFFIXES:
.SUFFIXES: .F90.O
.F90.O:
        $(fc) -c $<
```

اما در مورد این ۴ خط،
نکته‌ی اول:
حروف مربوط به SUFFIXES. حتما باید، حروف بزرگ باشد.
نکته‌ی دوم:
مجموعه‌ی این چهار خط به ما می‌گوید که تمامی فایل‌های f90. را کامپایل کند و فایل‌های o. آن را تولید نماید.بنابراین از الان می‌توانیم که در تارگت‌ها دستورهای ifort -c را حذف کنیم.
نکته‌ی سوم:
هیچ چیز دیگر به جز f90.o.، نمی‌تواند در خط سوم قرار بگیرد. تاکید می‌کنیم که هیچ چیز دیگر به جز f90.o.!
بنابراین makefile اصلاح شده به صورت زیر در می‌آید:

```
FC = ifort
mysubfun = mysubroutine.o myfunctions.o
myobjs = mymodule.o mainprogram.o
.SUFFIXES:
.SUFFIXES: .F90.O
.F90.O:
        $(fc) -c $<
all: $(myobjs) $(mysubfun)
        $(FC) $(myobjs) $(mysubfun) -o akbar.exe
myfunctions.o: myfunctions.f90 mymodule.o
mainprogram.o: mainprogram.f90 mymodule.o
mymodule.o: mymodule.f90
mysubroutine.o: mysubroutine.f90
clean:
        rm *~ *.mod *.o *.exe -f
```

توجه کنید که جابه‌جا کردن جای تارگت‌ها با هم، هیچ اشکالی ندارد و شما آزادید که مکان تارگت‌ها را به دلخواه انتخاب کنید. اما نکته‌ای که در پایان بحث makefile باید به آن اشاره کرد، آن است که بر خلاف آنچه که در فرترن وجود داشت، حروف بزرگ و کوچک در makefile با هم تفاوت دارند و باید مراقب استفاده از آن‌ها باشید.