Capítulo 2

Lenguajes de Programación para las ciencias

En este texto hemos decidido utilizar tres medios para programación científica (C o $\mathbb{C}$, Fortran90 o $\mathbb{F}$90 y Matlab) ya que es imposible cubrir todas las posibilidades . Existen muchos otros lenguajes o aplicaciones que han ganado especio en la s ciencias como Python, C#, Mathematica, etc.

$\mathbb{F}$90 y $\mathbb{C}$ son lenguajes de programación que requieren compilación, Matlab es una aplicación para programación, lo cual lo hace distinto. Es muy importante que el científico de hoy en día sepa por lo menos uno de los languajes de programción que requieran compilación. $\mathbb{F}$90 y $\mathbb{C}$ son estos lengaujes que queremos presentar. El uso de Matlab, aunque mucho más sencillo y con muchas funciones y programas prediseñados puede no ser ideal ya que una gran cantidad de programas y librerías que el lector puede utilizar en el futuro no están disponibles en Matlab. No es muy apropiado pedirle a los colegas que le expliquen un programa en C o Fortran, ya que solo sabe Matlab.

Este texto no es un texto sobre $\mathbb{F}$90 , $\mathbb{C}$ o Matlab y no pretende cubrir todas las propiedades y capacidades de cada uno de ellos. Esto es solo un tutorial para posteriormente aplicar los conceptos básicos en problemas numéricos en ciencias.

2.1. Elección del lenguaje

Desde el comienzo, FORTRAN (Formula Translator) fue el principal languaje en las ciencias computacional y de análisis numérico. El uso de multiples librarías desarrolladas con el tiempo (EISPACK, LAPACK, LINPACK, etc.) hicieron de Fortran un gran lenguaje para uso en ciencias. En los últimos tiempos sin embargo la entrada en vigencia de otras estructuras para la computació como arreglos dinámicos, servidores, visualización de datos y arquitecturas para paralelización hacen que Fortran 77 (la versión anterior) perdiera importancia. Otros lenguajes como C (y también C++) empezaron a ganar relevancia. Fortran ha

introducido Fortran90 para mejorar esas deficiencias. En esta sección se compara de manera algo superficial entre Fortran (F77 y $\mathbb{F}$90) y $\mathbb{C}$ (C, y C++) para que el lector si no conoce ninguno de los dos lenguajes pueda tomar una decisión medianamente informada. Al final de la sección se discute sobre Matlab.

Hoy en día C domina como lenguaje de programación, gracias en parte a que los departamentos de ciencias computacionales lo prefieren (por ejemplo Unix fue escrito en C).

Por qué Fortran?

1. Un gran número de programas (vieja escuela) están escritos en $\mathbb{F}$90 , y en muchos casos la comunicación con los científicos puede ser más sencilla si el lector sabe Fortran.

2. Hay una extensa librería de subrutinas de Fortran en la comunidad científica. Una gran parte de los paquetes de algebra lineal fueron escritos en $\mathbb{F}$90 . Muchos de estas subrutinas seguramente las veremos e incluso usaremos en el libro.

3. Variable compleja y doble precisión están incluidos.

4. Una gran cantidad de las capacidades en C/C++ han sido implementadas en Fortran90/95, tales como alocación dinámica de memoria, estructuras de datos, pointers, etc. Una gran cantidad de cosas que se pueden hacer en C se pueden hacer hoy en día en Fortran90 e incluso es posible conbinar los dos lenguajes (llamar C desde $\mathbb{F}$90 , o llamar fortran desde C).

Por qué C?

1. La gran mayoría de estudiantes y profesionales manejan C.

2. Algunas personas y gente de ingeniería de sistemas cree que C es mejor .

3. C puede ser muy útil para manejar bases de datos, creación de directorios, etc.

Por qué no Fortran?

1. No tiene graficador incorporado.

2. La mayoría de los nuevos programas estan escritos en otros lenguajes.

Por qué no C?

1. No tiene graficador incorporado.

2. Se requieren librerías para gran cantidad de problemas numéricos.

Y que de Matlab?

Matlab no es un lenguaje de programación, es un programa (que puede ser muy costoso) en el cual se puede programar problemas numéricos de manera simple. Tiene grandes ventajas comparado con $\mathbb{C}$ y $\mathbb{F}90$ ya que posee graficador (y bastante bueno), tiene una gran cantidad de funciones incorporadas, y está optimizado para algebra lineal. Si se tiene un problema numérico, en Matlab es posible resolverlo rápidamente y graficar los resultados.

Pero Matlab también tiene desventajas. Cuando el problema numérico es importante y de gran tamaño, Matlab no es la herramiento óptima. $\mathbb{F}90$ y $\mathbb{C}$ pueden hacer el trabajo muchisimo más rápido. $\mathbb{C}$ y $\mathbb{F}90$ se pueden paralelizar y actualmente son los lenguajes utilizados para simulaciones numéricas en centros de investigación en el mundo. Matlab está desarrollando herramientas para paralelizar pero esto seguramente venga con un costo adicional.

2.2. $\mathbb{F}90$, $\mathbb{C}$ y Matlab

Para poder compilar programas en $\mathbb{C}$ o $\mathbb{F}90$ y poder abrir el programa Matlab, naturalmente estos deben estar instalados en el computador en uso. Matlab es un programa que se puede adquirir a través de la empresa MathWorks, $\mathbb{C}$ y $\mathbb{F}90$ se pueden obtener de múltiples proveedores (por ejemplo el compilador de Intel `ifort` es uno de los mejores actualmente ya que Intel domina el mercado de procesadores). $\mathbb{C}$ y $\mathbb{F}90$ se pueden adquirir de manera gratuita descargando `gcc`, `g95` o `gfortran`.

Si el usuario está trabajando en un computador Unix, el compilador $\mathbb{C}$ y $\mathbb{F}90$ estan probablemente preinstalados. Por razón de costos los ejemplos en este libro se realizarán con los compiladores gratuitos (Matlab si debe ser adquirido por el lector). Se puede utilizar el emulador de Matlab llamado `Octave`, que se puede adquirir gratis.

En principio, todos los programas que se trabajarán acá deben ser independientes del tipo de computador en el cual se realicen. para la edición y generación de programas para cada uno de los lenguajes es necesario tener un editor de texto (idealmente NO Microsoft Word) tal como se discutio en el capítulo anterior.

2.3. El primer programa explicado

Lo primero que se necesita comprender es que para Fortran90 y C, se requiere de un archivo de texto ascii, el cual se llama el código fuente. Para $\mathbb{F}90$ el nombre del archivo debe terminar con `.f90`, para $\mathbb{C}$ con `.c`. Se recomienda seguir este patrón ya que la mayoría de compiladores siguen esta regla.

En Matlab, una vez abierto, el usuario puede dar comandos los cuales generan un resultado o respuesta de Matlab. En este texto vamos a generar códigos fuente para Matlab, con terminación `.m`.

Para que el programa pueda correr, en $\mathbb{C}$ y $\mathbb{F}90$ es necesario compilarlo para a partir del archivo de texto se genere un archivo ejecutable (el programa en sí).

Programa 2.1 Primer programa `hello_world` en 𝔽90 , ℂ y Matlab.

```
! hello_world.f90
program hello_world

   print *,"hello, world"

end program hello_world

/* hello world.c */
#include <stdio.h>
int main()
{
   printf("hello, world\n");
   return 0;
}

% hello_world.m
disp('hello, world')
```

Un ejemplo es el programa `hello_world.xx` (donde xx depende de si es 𝔽90 , ℂ o matlab) que se puede ver en Programa C.2. Para compilar en fortran

```
pperez > gfortran hello_world.f90 -o hello_world
pperez >
```

y para compilar en C

```
pperez > gcc hello_world.f90 -o hello_world
pperez >
```

Si en la Terminal no se obtiene ninguna respuesta, esto es positivo. Si hay algún error de sintaxis, este error sería detectado durante la compilación y un mensaje de error sería desplegado.

Si todo sale bien, un archivo ejecutable (binario) del programa llamado `hello_world` se crea y el programa se puede correr somplimente digitando

```
pperez > hello_world
hello, world
pperez >
```

La diferencia entre el código fuente y el ejecutable es lo fundamental en lenguajes como ℂ o 𝔽90 , y a su vez es lo que hace que los programas sean mucho más rápidos que en lenguajes no-compilados como Matlab. En Matlab, tendramos

```
matlab > hello_world
hello, world
```

2.3.1. Explicación Detallada

Para el caso de Fortran

El programa esta escrito así:

```
! hello_world.f90
program hello_world

   print *,"hello, world"

end program hello_world
```

La primera linea es un comentario. Cualquier texto que esté después del signo `!` en una linea es un comentario y no es leído por el compilador. La siguiente linea se vuelve a leer. Eso quiere decir que uno puede tener comentarios después d eun comando, por ejemplo

```
   print *,"hello, world"   ! comentario
```

también sería valido. Tambiés es importante tener en cuenta que las lineas en blanco no afectan la compilación ni la velocidad del programa y mas bien pueden ayudar a que el código fuente sea mas lejible.

La siguiente linea es el nombre del programa

```
program hello_world
```

El nombre `hello_world` en realidad no es utilizado por $\mathbb{F}$90 , pero es considerado una costumbre de buen estilo de programación. Siempre un programa debe ser terminado con un `end`, y para mejor estilo

```
end program hello_world
```

Nuevamente no es necesario el término program y el nombre del programa, pero es considerado más lejible.

La siguiente linea ya representa el cuerpo del programa

```
   print *,"hello, world"
```

y note que acá el cuerpo del programa está indentado con tres espacios, que nuevamente ayuda a la lejibilidad del código fuente (aunque no es necesario, si lo recomendamos). Este comando entonces hace que el programa *imprima* los caracteres en comillas a la pantalla.

Para finalizar, todo programa debe terminar con un comando `end` y que nosotoros completemos con `program hello_world`. Este cominezo y terminación de programas no es obligatorio pero si ayuda a la persona leyendo el código fuente a saber que programa empieza y cual termina, especialmente cuando tenemos códigos de miles de lineas de longitud.

Para el caso de C

El programa esta escrito así:

```
/* hello world.c */
#include <stdio.h>
int main()
{
   printf("hello, world\n");
   return 0;
}
```

Similar al caso de 𝔽90 , la primera linea es un comentario. En este caso cualquier texto que esté encerrado entre `/*` y `*/` es un comentario. Esto quiere decir que en ℂ uno puede tener comentarios de múltiples lineas (a diferencia de 𝔽90 .

La siguiente linea:

```
#include <stdio.h>
```

es bien importante (note que esta linea no termina con `;` (punto y coma). ℂ es un lenguaje que por si solo es un lenguaje de bajo nivel que no puede hacer muchas de las cosas que uno esperaría que un lenguaje pueda hacer. Para expandir sus capacidades del lenguaje, muchas librerías están disponibles en instalaciones de C, pero con la necesidad de cargarlas antes de empezar el programa. Las librerías se cargan al comienzo del programa y la librería `stdio.h` facilita el input/output y es la librería de mayor uso. En otros casos es necesario el uso de la librerá `math.h` para la mayoría de cálculos matemáticos.

```
int main()
```

Todo programa de ℂ es en sí una serie de funciones. Cada programa debe tener una función llamada `main` que le dice al compilador donde se comienza a ejecutar. El término `int` define la función como un número entero y que va a regresar un valor o número entero. El paréntesis `()` vacio es usado para los argumentos de la función, y en este caso no tiene. Finalmente, el contenido o cuerpo de la función se incluye dentro de los parentésis curvos { y }. Para hacer el código más claro, usualmente se indenta el cuerpo del programa.

El cuerpo del programa o función tiene

```
   printf("hello, world\n");
```

el cual envia a la pantalla `'hello, world \n'`. Las comillas sirven para definir un arreglo de caracteres y `\n` es notación ℂ para un caracter de nueva linea y así el cursor avanza a la siguiente linea. Si no se utiliza el `\n` el cursor se mantendrá en la misma linea. Eso quiere decir que el equivalente escribir

```
printf("hello, ");
printf("world\n");
```

y se obtendría el mismo resultado. No olvidar el \n, un error muy común.

Por último, se TIENE que cerrar un comando con el ;, lo cual no es natural para el usuario habitual de 𝔽90 , ya que para una función ℂ una nueva linea no significa nada. Por ejemplo, el siguiente programa

```
/* hello world.c */
#include <stdio.h>
int main()
{
   printf
   ("hello, world\n");
   return 0;
}
```

correría sin ningún problema. Pero no se resomienda. Tampoco se recomiendo u nprograma así:

```
/* hello world.c */
#include <stdio.h>
int main() {printf ("hello, world\n");return 0;}
```

que aunque más compacto, es muy difícil de seguir. El hecho que el código fuente ocupe menos lineas no significa que el ejecutable sea más rápido.

Para el caso de Matlab

En este caso el programa es simple. El signo % es usado para comentarios y el comando disp()

```
disp('hello, world')
```

despliega en la pantalla el set de caracteres que esten entre los paréntesis.

Sobre Makefiles

Si el lector tiene un programa, compilarlo es simple

```
gfortran prog1.f90 -o prog1
```

para el caso de 𝔽90 y

```
gcc prog1.f90 -o prog1
```

para C. Pero si en el futuro se requiere compilar cientos de programas, muy rápidamente se vuelve complicado digitar este comando muchas veces. Por esto se recomienda el manejo de make, un comando Unix a través de archivos conocidos como Makefile. Este archivo debe estar ubicado en el directorio donde se encuentra el programa y debe aparecer así

```
%:%.f90
        gfortran $< -o $*

%:%.c
        gcc $< -o $*
```

IMPORTANTE: El truco acá es que el espacio antes de `gfortran` o de `gcc` TIENE que ser un TAB, no simplemente espacios.

Una vez se tenga el Makefile, se digita

```
make prog1
```

y automáticamente el programa se compila.

`Makefiles` son de gran utilizadad para seguir el rastro y personalizar las opciones del compilador y para hace*link* con subrutinas, librerías, etc. En al Apéndice A una descripción más completa (aunque muy básica) sobre el uso de `Makefies` es proporcionada.

2.4. Multiplicación de números enteros

El programa para multiplicar dos números enteros (multiplicar 2 y 3) se puede ver en Programa 2.2.

En los tres programa se utilizan tres variables, las letras `a, b` and `c`. Lo longitud del nombre de las variables puede llegar a ser de hasta 31 caracteres y puede formarse a partir de la combinación entre letras, números y _.

Hay dos diferencias fundamentales entre $\mathbb{C}$ -$\mathbb{F}$90 y Matlab. En los dos primeros es necesario **definir** las variables, mientras que Matlab las define de acuerdo a lo que el usuario ponga como valores. En $\mathbb{F}$90 y $\mathbb{C}$ las variables pueden definirse como enteros (integer o int), reales (real o float) y otras definiciones que veremos más adelante.

Sobre declaración de variables

Aunque no es obligatorio declarar las variables, se sugiere siempre utilizar en $\mathbb{F}$90 un `implicit none`. Si las variables no son declaradas, el compilar las asigna como enteros o reales, dependiendo de la primera letra de la variable. Las variables iniciando con `i` y `n` (INteger obviamente) se asumen como enteros y las demás variables como reales.

En muchas códigos antiguos, se utiliza esta convención y es importante que el lector la conozca. En esas casos solamente se declaraban las variables que rompían la regla (por ejemplo una variable como `year`, que es un entero). Estas variables no declaradas se denominan *declared implicitly*, pero esto se puede cambiar con comandos como

```
implicit real (a-z)
```

Programa 2.2 Programa para multiplicar dos números enterors

```
(multint.f90)
program multint

   implicit none
   integer :: a, b, c      ! declare variables

   a = 2
   b = 3

   c = a * b
   print *,"Product = ",c

end program multint

(multint.c)
/* test program to multiply two integers */
#include <stdio.h>
int main()
{
   int a, b, c; /* declare variables */
   a = 2;
   b = 3;

   c = a * b;
   printf("Product = %d \n",c);
   return 0;
}

(multint.m)
 % Script to multiply two integers

a = 2;
b = 3;

c = a * b;
disp(['Product = ', num2str(c)])
```

que declaran todas las variables como reales. Esto puede hacer que el programa sea aún más confuso y nosotros (y prácticas de programación modernas) sugerimos utilizar

```
implicit none
```

al comienzo de TODO programa. Si el código utiliza una variable no definida y el comando `implicit none` ha sido utilizado, se desplegará un error durante compilación, lo cual evita errores difćiles de coregir.

Caso de Fortran

Al comienzo del programa se utiliza el `implicit none`. La siguiente linea

```
integer :: a, b, c      !declare variables
```

declara las variables como números enteros. Para un compilador típico en Linux o Sun el rango de números que se pueden expresar con enteros de 4-byte es -2,147,483,648 a 2,147,483,647. Para números reales, la declaración es

```
real(4) :: a, b, c
```

donde el rango en este caso es 1.175494e-38 to 3.402823e+38 (para mayor precisión se puede utiliazr real(8), ttambién conocido como *double precision*).

Las siguiente lineas son bastante obvias

```
a = 2

b = 3

c = a * b

print *,"Product = ",c
```

donde `*` es multiplicación, como casi en los otros lenguajes. Suma es `+`, resta `-` y división `/`. En $\mathbb{F}$90 un número `x` a la potencia de `b` se escribe `x**b` donde `x` y `b` pueden ser números reales.

Caso de C

Similar al caso anterior, las tres variables deben ser declaradas al comienzo de la función

```
int a, b, c; /* declare variables */
```

donde el comando `int` los declara enteros. Si se requiere que sean números reales, entonces

```
float a, b, c;
```

con rangos iguales a los de 𝔽90 . Los valores de `a` y `b` están definidos en lineas separadas, aunque pueden ser definidas en una sola linea, no es lo que se recomienda en este texto.

```
c = a * b;
```

define la multiplicación como en 𝔽90 . Para imprimir o desplegar el resultado se usa

```
printf("Product = %d \n",c);
```

con una lista de caracteres `Product =` seguidos del valor de `c` y el caracter de nueva linea. El término `%d` no se despliega, simplemente define el formato del valor de `c` que se despliega. Algunos formatos son:

- %d imprimir número entero
- %8d imprimir número entero con al menos 8 caracteres de longitud (incluido espacios en blanco)
- %f imprimir número real (floating point)
- %7f imprimir número real (floating point) con al menos 7 caracteres de longitud (incluyendo el punto decimal.
- %7.2f imprimir número real (floating point) con 7 caracteres de longitud y dos números después del punto decimal.
- %c imprimir caracter sencillo
- %s imprimir caracter como cadena de caracteres
- %i a veces se utiliza para números enteros, aunque no es recomendado.

2.5. Una tabla trigonométrica usando `loops`

Cualquier lenguaje de programación debe proveer una forma de hacer *loops* a lo largo de una serie de valores de una variable. En ℂ y Matlab esto se hace a través del comando `for` y en 𝔽90 con el comando `do` [ver Programa 2.3].

Caso Fortran

Es importante notar acá que para Fortran90 no es recomendable (e incluso algunos compiladores NO lo aceptan) hacer `do` loops con valores reales, y solo se recomiendo hacer loops con valores enteros. El standard F90 es con números enteros exclusivamente.

```
do i = 0, 89, 1
   ...
enddo
```

Programa 2.3 Programa para generar tabla trigonométrica

```
(trigtable.f90)
program trigtable

!*****************************
   implicit none
   real(4) :: theta, stheta, ctheta, ttheta, degrad
   integer :: i
   real(4), parameter :: pi=3.1415927
!*****************************

   degrad = 180./pi

   do i = 0, 89, 1

      theta = real(i)

      ctheta = cos(theta/degrad)
      stheta = sin(theta/degrad)
      ttheta = tan(theta/degrad)

      print "(f5.1,1x,f6.4,1x,f6.4,1x,f7.4)",  &
      theta, ctheta, stheta, ttheta

  enddo

end program trigtable

(trigtable.c)
#include <stdio.h>
#include <math.h>
int main()
{
   float theta, stheta, ctheta, ttheta, degrad;

   degrad = 180./3.1415927;
   for (theta=0.0; theta<89.5; theta=theta+1.0){
      ctheta = cos(theta/degrad);
      stheta = sin(theta/degrad);
      ttheta = tan(theta/degrad);
      printf("%5.1f %6.4f %6.4f %7.4f \n",
                theta,ctheta,stheta,ttheta);
   }
   return 0;
}
```

... continua **Programa 2.3**

```
% Script for trigtable
degrad = 180/pi;  % pi pre-definido en matlab
for i = 1:89
   theta = i;

   ctheta(i) = cos(theta/degrad);
   stheta(i) = sin(theta/degrad);
   ttheta(i) = tan(theta/degrad);

end

[ctheta' stheta' ttheta']
```

comienza el `do` loop con valores de `i` desde 0 hasta 89, saltando 1 a 1. Los loops se cierran con un `enddo` y es recomendable indentar los comandos dentro del loop.

Tanto F90 como C y Matlab usan radianes (no grados) como argumento en funciones trigonométricas. Además las funciones trigonométricas trabajan con números reales. Por estas razones se utiliza la variable real `theta` que equivale al valor de `i` (pero real).Y dentro del cálculo de las funciones trigonométricas se transforma de grados a radianes a través de la variable `degrad`. Note que `pi` está definida al comienzo como una variable `real, parameter` es decir que su valor no puede cambiar en el programa.

```
   theta = real(i);

   ctheta = cos(theta/degrad);
   stheta = sin(theta/degrad);
   ttheta = tan(theta/degrad);
```

Dentro de cada loop entonces se comienza con un valor de `i` = 0, aumentando de a 1 unidad hasta llegar a 89. La variable `theta` toma entonces valores de 0,0, 1,0, 2,0, ..., 89,0 para cada loop.

Dentro de cada loop se cálcula el coseno, seno y la tangente de `theta` una vez se corrige a radianes. Para el despliegue en pantalla se tiene

```
   print "(f5.1,1x,f6.4,1x,f6.4,1x,f7.4)", &
               theta, ctheta, stheta, ttheta
```

para obtener un output con un formato específico. Acá `f5.1` significa lo mismo que en C, un valor real con cinco dígitos y solo un número después del punto decimal. Los valores están justificados a la derecho y se ponen espacios en blanco si es necesario. El valor `1x` se usa para forzar un espacio en blanco libre y evitar que los números se peguen.

Por último, el caracter `&` se usa para continuar una linea si el espacio no es suficiente y que se pueda leer el código fuente con facilidad.

En $\mathbb{F}$90 los formatos siguen reglas similares a $\mathbb{C}$ (exceptuando el salto de linea)

- i5 imprimir número entero con 5 caracteres.
- i5.4 igual al anterior pero se ponen zeros para completar 4 espacios (88 se imprime 0088)
- f8.3 número real con 8 espacios, 3 despues del punto decimal
- e12.4 valor real con exponentes, 4 espacios después del decimal (por ejemplo ␣-0.2342E+02) donde existe un espacio en blanco a la izquierda para completar 12 espacios.
- a8 imprimir 8 caracteres en cadena. (si el número de caracteres de la variable supera 8 espacios, los 8 de la izquierda se imprimen.
- 2x genera espacios en blanco para separar variables
- / comienzo de una nueva linea

Si el número de espacios proporcionado no es suficiente para desplegar el número, `****` aparecerá.

Caso C

La primera observación es que para el uso de las funciones trigonométricas toca incluir la librería `math.h`. En algunos compiladores, para acceder a las librerías de matemática es obligatorio incluir la opción `-lm` durante la compilación

```
gcc -lm trigtable.c -o trigtable
```

Si al compilar sin esta opción hay errores asociados con las funciones trigonométricas entonces es neceario utilizar la opción. Note que esto puede automatizarse en el `Makefile`.

El `for` loop es similar al de $\mathbb{F}$90 , aunque en este caso si se hace el loop usando valores reales (float)

```
for (theta=0.0; theta<89.5; theta=theta+1.0){
   ...
}
```

La variable `theta` asume entonces valores (0,0, 1,0, 2,0, ..., 88,0, 89,0), dentro del loop se calculan las funciones trigonométricas convirtiando también a radianes los ángulos.

Los resultados se despliegan en la pantalla con el formato determinado

```
   printf("%5.1f %6.4f %6.4f %7.4f \n",
             theta,ctheta,stheta,ttheta);
```

para tener los valores alineanos en columnas.

Una posible forma alternativa de escribir el loop es con el uso del comando `while` así:

```
theta = 0.0;
while (theta<89.5){
   ctheta = cos(theta/degrad);
   stheta = sin(theta/degrad);
   ttheta = tan(theta/degrad);
   printf("%5.1f %6.4f %6.4f %7.4f \n"
           theta,ctheta,stheta,ttheta);
   theta=theta+1.0;
}
```

aunque a nuestro parecer no es tan claro.

La variable de conversión de grados a radianes `degrad`, posee un valor único y así como en $\mathbb{F}$90 se usa `real, parameter`, en $\mathbb{C}$ también se puede definir una `constante simbólica` que puede ser usada a lo largo de todos los programas. El código fuente comenzaría

```
#include <stdio.h>
#include <math.h>
#define DEGRAD 57.29577951
int main()
{
  ...
  ctheta = cos(theta/DEGRAD);
  ...
```

y se considera una buena práctica de programación. Por convención las constantes simbólicas van en mayúsculas.

Vale la pena tener en cuenta que

```
#define DEGRAD 180./3.1415927
```

no funciona y que

```
#define DEGRAD (180./3.1415927)
```

si. En nuestro caso es más rapido usar el valor exacto 57,29577951, ya que es más rápido.

Caso Matlab

En Matlab la programación de este problema es muy similar a los anteriores. Note que en este caso las variables (`ctheta, stheta, ttheta`) son vectores.

Esto para después de terminado el loop se pueda desplegar los resultados de manera sencilla. Note además que el valor `pi` no hay que definirlo ya que Matlab lo tiene definido. Por último, en Matlab el signo `;` a final de una linea sirve de manera similar a C, pero adicionalmente le dice a Matlab que NO despliegue en la pantalla el valor o resultado del comando.

Cuadro 2.1: Tabla con algunas funciones presentes en lenguajes de programación.

Fortran90	**C**	**Matlab**	**Función**
acos(x)	acos(x)	acos(x)	arcocoseno
asin(x)	asin(x)	asin(x)	arcoseno
atan(x)	atan(x)	atan(x)	arccotangente
atan2(x)	atan2(x)	atan2(x)	arcotangente de y/x en el cuadrante correcto
cos(x)	cos(x)	cos(x)	coseno
cosh(x)	cosh(x)	cosh(x)	coseno hiperbólico
exp(x)	exp(x)	exp(x), e^(x)	exponencial
log(x)	log(x)	log(x)	logaritmo natural (base e)
log10(x)	log10(x)	log10(x)	logaritmo base 10
sin(x)	sin(x)	sin(x)	seno
sinh(x)	sinh(x)	sinh(x)	seno hiperbólico
sqrt(x)	sqrt(x)	sqrt(x)	raiz cuadrada
tan(x)	tan(x)	tan(x)	tangente
tanh(x)	tanh(x)	tanh(x)	tangente hiperbólica

2.6. Interacción con el teclado

En el ejemplo Programa 2.2 sólo se puede multiplicar dos números, el 3 y el 2. Si el usuario desea multiplicar el 4 y el 2, tiene que cambiar el programa, recompilar. Para mejorar la capacidad de los programas es necesario hacerlos más interactivos, de tal forma que el usuario pueda decidir que valores entran dentro del programa [ver Programa 2.4].

En el caso de $\mathbb{F}$90 pedir *input*del usuario es bastante sencillo, simplemente

```
print *, "Enter two integers"
 read *, a, b
```

mientras que en el caso de $\mathbb{C}$ es

```
printf("Enter two integers \n");
scanf("%d %d", &a, &b);
```

Al correr los programas en Fortran:

Programa 2.4 Programa con `input` para multiplicar dos números enteros

```
(usermult.f90)
program usermult

   implicit none
   integer :: a, b, c

   print *, "Enter two integers"
   read *, a, b

   c = a * b
   print *, "Product = ", c

end program usermult

(usermult.c)
#include <stdio.h>
int main(){
   int a, b, c;
   printf("Enter two integers \n");
   scanf("%d %d", &a, &b);
   c = a * b;
   printf("Product = %d \n",c);
   return 0;
}

(usermult.m)
function c = usermult (a,b);

c = a*b;

disp(['Product = ', num2str(c)])
```

```
pperez > usermult
 Enter two integers
2 5
 Product =           10
```

C

```
pperez > usermult
Enter two integers
2 4
Product = 8
```

y Matlab

```
matlab > c = usermult(2,4);
Product = 8
```

También es válido en $\mathbb{F}90$ y $\mathbb{C}$ digitar

```
> usermult
Enter two integers
2
5
Product =  10
```

y esto sucede cuando uno olvida que debe digitar dos números.

Que sucede si el usuario digita un valor real (4.2) por ejemplo? O si digita un caracter diferente a un número? Para $\mathbb{C}$ y $\mathbb{F}90$ el resultado es distinto (en Matlab, el error es más evidente ya que en realidad lo se creo fue una función).

Abajo, ejemplos de lo que sucede con errores en el input, para Fortran:

```
> usermult
pperez > usermult
 Enter two integers
2 4.2
At line 7 of file usermult.f90 (unit = 5, file = 'stdin')
Fortran runtime error: Bad integer for item 2 in list input
```

e igual sucede si se digita un caracter diferente a números enteros. Para C:

```
pperez > usermult
Enter two integers
4.3 2
Product = 2130723144
pperez > usermult
Enter two integers
2 a
Product = -1082122076
```

Lo preocupante en el caso de ℂ es que no genera un error, sino que da una respuesta incorrecta. Esto hace que el programa no sea muy robusto, y esto se debe a que en ℂ no hay el equivalente al comando *free-format read* (p.e., `read *, a, b`). Esto se puede evitar con condicionales que se discutirán más adelante.

2.7. Condicionales, `if` statements

Para explicar el uso de condicionales o `if` statements, hacemos una variación del programa `usermult` [Programa 2.4] para que el usuario pueda continuar digitando el par de números hasta que quiera detener el programa [ver Programa 2.5].

En el caso de 𝔽90 , se genera un `do` loop sin argumentos (es decir que hace loops infinitamente) y un bloque de código que repetidamente se lleva a cabo hasta que el comando `exit`es ejecutado. El programa permite que el usuario digita dos números enteros para ser multiplicados, pero cuando se desea que el programa termine el usuario debe digitar dos veces el cero. El `if` hace la validación de los argumentos

```
if (a == 0 .and. b == 0) exit
```

y si es el caso, sale del `do` loop con el comando `exit`.

El uso de la sintaxis de F77 todavía es valida en F90, por lo tanto también es valido

```
if (a .eq. 0 .and. b .eq. 0) exit
```

Los `if` pueden validar condiciones mas complejas, como por ejemplo

```
if ( (a > b .and. c <= 0) .or. d == 0) z = a
```

donde los parentesis se usan para darle claridad al orden de condiciones.

En el código ℂ hay varias observaciones

1. Como se discutio anteriormente, si el usuario en ℂ digita un caracter no numérico, ℂ lo lee como un número. Para hacer el código más robusto se aprovecha el hecho que `scanf` es un a función que retorna el número de input que haya leído. En nuestro caso [Programa 2.5] el valor esperado es 2.

2. `while(1)` se usa para un loop while que continue eternamente. Para validar una condición se requiere un falso y un verdadero, en ℂ el verdadero se le asigna el valor de 1; por lo tanto al tener `while(1)` el loop continua ejecutandose.

3. Se usa el comando `break` para manualmente salir del loop cuando la condición a=0 y b= 0 sea cierta. `break` es un comando sumamente útil y funciona como el `exit`en 𝔽90 .

Programa 2.5 Programa con `input` para multiplicar dos números enteros.

```
(usermult2.f90)
program usermult2

   implicit none
   integer :: a, b, c

   do
      print *, "Enter two integers (zeros to stop)"
      read *, a, b

      if (a == 0 .and. b == 0) exit

      c = a * b
      print *, "Product = ", c
   enddo

end program usermult2

(usermult2.c)
#include <stdio.h>
int main(){
   int a, b, c;
   while (1) {
      printf("Enter two integers (zeros to stop)\n");
      if ( scanf("%d %d", &a, &b) != 2){
         printf("***Error in input \n");
         fflush(stdin);
      } else if (a == 0 && b == 0)
         break;
      else {
         c = a * b;
         printf("Product = %d \n",c);
      }
   }
   return 0;
}
```

4. El comando `fflush(stdin)` limpia los comandos del teclado después de un error en la digitación del usuario (por ejemplo si pone una letra), para que cuando vuelva a ser ejecutado no quede en la memoria la última digitación. Puede ser de gran ayuda siempre poner un `fflush(stdin)` antes de un comando `scanf`.

2.7.1. Comandos `i++` y `i+=`

Para el usuario de 𝔽90 o Matlab, la forma para incrementar el valor de una variable por una unidad es:

```
i = i + 1;
```

Esta sintaxis es normal en BASIC o 𝔽90 . En ℂ sin embargo, se permite una expresión más corta y compacta por medio del comando u operador de incremento `++` asi:

```
i++;
```

que tiene exactamente el mismo significado. Otra forma de uso es

```
(fortran)
j = i
i = i + 1
(C)
j = i++;
```

Note que en C, primero se iguala `j` con `i` y después se le añade 1 a `i`. En otro caso

```
(fortran)
i = i + 1
j = i
(C)
j = ++i;
```

la adición se da antes de que `j` sea igual a `i`. En ℂ también hay operadores de decremento (`--`), con operaciones como `i--` o `--i` con resultados obvios.

Otra característica de C, es que s epuede expresar `i = i + 2` así

```
i += 2;
```

En este caso `+=` se le conoce como el `assignment operator` o operador de asignación. Operaciones similar pueden ser

```
n -= 10;       significa      n = n - 10
x *= 2.0;      significa      x = 2.0 * x
zeta /= 8.3;   significa      zeta = zeta/8.3
```

En todos los casos, los argumentos pueden ser números enteros o reales (floats).

Algunos textos de ℂ sugieren que el uso de operados de incremento o de asignación hace que el código puede correr más rápido. Y aunque esto puede ser cierto (y en casos que hemos probado no es así), esto hace que el código sea más dificil de leer para un novato en C. Es importante sin embargo saber y poder manejarlos ya que muchos de los códigos que en el futuro se encuentren probablemente tengan estos operadores.

2.7.2. Condicionales `If, then, else`

En el ejemplo anterior [Programa 2.5] un único condicional es ejecutado cuando la condición es verdadera. Un programa con mayor verstilidad tiene un mayor número de condiciones o posibles resultados

```
if (logica condicional) then
   (block of code)
else if (logica condicional) then
   (block of code)
else if (logica condicional) then
   (block of code)
...
else
   (block of code)
end if
```

Los bloques de código (block of code) pueden tener muchas lineas de código si se requiere y tantos `else if` como sea necesario se pueden usar. Note que el orden de validación va de arriba hacia abajo, es decir que si la expresión lógica 2 es verdadera, el programa no continua para validar las expresiones 3, 4, etc. Al final se puede tener un `else` cuyo bloque de código se ejecuta si ninguna condición se cumple.

Un ejemplo del uso de condicionales donde el usuario se le pide un número real (positivo) para retornar la raíz cuadrada de dicho número. Si el usuario digita un número negativo, el programa debe solicitar un número nuevamente, si es positivo entonces ccalcula la raiz y si se digita el vlor `0` el programa se detiene. Note en 𝔽90 como el comando `cycle` le dice al programa que ejecute la siguiente iteración del `do` loop, mientras que el comando `exit`se sale del loop. El uso de estos comandos es nuevo en Fortran90 y es recomandado utilizar. No se recomienda el uso de `goto`, ya que es considerado una mala costumbre de programación (y en muchos casos algunos científicos abusaron del goto).

2.8. Ejemplo del máximo común denominador

Haciendo uso de los discutido hasta el momento, el programa 2.7 calcula el máximo común denominador de dos números enteros. El ejemplo no es necesariamente muy eficiente pero funciona de manera rápida y se puede leer fácilmente para números pequeños.

Programa 2.6 Programa para calcular la raíz cuadrada de un número positivo

```
(usersqrt.f90)
   implicit none
   real :: a, b

   do
      print *, 'Enter positive real number (0 to stop)'
      read *, a

     if (a < 0) then
         print *,'This number is negative!'
         cycle
     else if (a == 0) then
         exit
     else
         b = sqrt(a)
         print *, 'sqrt = ', b
     end if

   end do

(usersqrt.c)
#include <stdio.h>
#include <math.h>
int main(){
   float a, c;
   while (1) {
      printf("Enter two integers (zeros to stop)\n");
      if ( scanf("%f", &a) != 1){
         printf("***Error in input \n");
         fflush(stdin);
      }
      else if (a == 0.)
         break;
      else if (a < 0.)
         printf("This number is negative! \n");
      else {
         c = sqrt(a);
         printf("Product = %f \n",c);
      }
   }
   return 0;
}
```

Tabla 2.1 Una lista de operaciones de relación en varios lenguajes

F77	F90	C	MATLAB	significado
.eq.	==	==	==	igual a
.ne.	/=	!=	~=	no es igual a
.lt.	<	<	<	menor a
.le.	<=	<=	<=	menor o igual a
.gt.	>	>	>	mayor a
.ge.	>=	>=	>=	mayor o igual a
.and.	.and.	&&	&	y
.or.	.or.	||	|	o

Caso Fortran

Algunos comandos nuevos:

- `min(a,b)` calcula el valor mínimo entre `a` y `b` usando la función intrinseca de 𝔽90 `min`. Obviamente también existe la función `max`, y ambas pueden tener más de dos argumentos.
- `mod(a,i)` retorna el restante de la división de `a` por `i`, también llamado el módulo. Si `mod(a,i)` es cero, entonces `a`es divisible por `i`.

Caso C

El comando `%` es el operador de módulo que funciona igual el de 𝔽90 pero se escribe `a% i`.

Probablemente la parte del código fuente más complicada de entender es la parte donde se define el mínimo

```
#define MIN(A,B) ( (A) < (B) ? (A) : (B) )
```

C no tiene una función intrínseca para el mínimo entre dos números por lo que es necesario escribirlo nosotros mismos. Una opción sería

```
if (y > x)
z = x;
else
z = y;
```

Programa 2.7 Cálculo del máximo común denominador entre dos números enteros

```
(gcf.f90)
   implicit none
   integer :: a, b, i, imax

  do

      print *, 'Enter two integers (zeros to stop)'
      read *, a, b

      if (a == 0 .and. b == 0) then
         exit
      else

        do i = 1, min(a,b)
            if (mod(a,i) == 0 .and. mod(b,i) == 0) imax=i
         end do
         print *, "Greatest common factor = ", imax

      end if
  enddo

(gcf.c)
#include <stdio.h>
#define MIN(A,B) ( (A) < (B) ? (A) : (B) )
int main()
{
   int a, b, i, imax;
   while (1) {
      printf("Enter two integers (zeros to stop)\n");
      if ( scanf("%d %d", &a, &b) != 2){
         printf("***Error in input \n");
         fflush(stdin);
      } else if (a == 0 && b == 0)
         break;
      else {
         for (i=1; i<=MIN(a,b); i++){
            if (a % i == 0 && b % i == 0) imax=i;
         }
         printf("Greatest common factor = %d \n",imax);
      }
   }
   return 0;
}
```

En ℂ es posible algo llamado expresión condicional,

```
z = (x < y) ? x : y;
```

donde ? es un operator ternario (*tternary operator*), de tal manera que `min(A,B)` se asigna dependiendo si `(x<y)`. El comando en general se usa así:

```
expr1 ? expr2 : expr3
```

`expr1` se evalua primero, si es verdadero, se le asigna el valor de `expr2`, y si no, se le asigna el valor de `expr3`.

Tabla 2.2 Algunas funciones útiles en los diferentes lenguajes

F90	C	MATLAB	significado
abs(a)	abs(a)	abs(a)	valor absoluto de a
sign(a,b)			abs(a) por el signo de b
real(a)			conversion a real (float)
int(a)		fix(a)	conversion a entero
nint(a)		round(a)	entero mas cercano
ceiling(a)		ceil(a)	entero mas cercano por arriba
floor(a)		floor(a)	entero mas cercano por debajo

2.9. Funciones propias

A medida que la complejidad y longitud de un programa de computador aumenta, es una buena idea partir el problema en pedazos más pequeños, definiendo **funciones** o **subrutinas** (fortran) para que hagan esas pequeñas partes del programa. Esto da multiples ventajas como por ejemplo

- El científico puede evaluar cada parte del programa individualmente para saber si esta funcionando como se espera y evitar tener que correr el programa completo para saber lo mismo.
- El código fuente es más modular y más fácil de entender.
- Es más fácil usar las partes del programa por separado en otros programas

Como ejemplo del uso de funciones propias, el programa 2.8 calcula el máximo común denominador pero usando funciones propias.

Caso Fortran

Para hacer el cálculo del `gcf` se utiliza una función propia, que en este caso llamamos `getgcf`. La función requiere dos argumentos `a` y `b` que provienen del programa principal. La función se llama

```
gcf = getgcf(a,b)  (fortran)
```

Y la función comienza con

```
integer function getgcf(x, y) result(z)
...
end function getgcf
```

Programa 2.8 Máximo común denominador con funciones propias

```
(gcf2.f90)
program gcf2
   implicit none
   integer :: a, b, getgcf, gcf

   do

      print *, 'Enter two integers (zeros to stop)'
      read *, a, b

      if (a == 0 .and. b == 0) then
         exit
      else
         gcf = getgcf(a,b)
         print *, "Greatest common factor = ", gcf
      end if

   enddo
end program gcf2

integer function getgcf(x, y) result(z)
   implicit none
   integer, intent(in):: x,y
   integer :: i, z

   do i = 1, min(x,y)
      if (mod(x,i) == 0 .and. mod(y,i) == 0) z = i
   end do
end function getgcf
```

Las variables `x` y `y` en la función asumen el valor que proviene del programa principal. Los argumentos tienen que ajustarse en el tipo de variable y el número de variables, pero no tienen que tener el mismo nombre. Dentro de la función se puede poner

```
intent(in):: x,y
```

con lo cual se evita que la función (o subrutina) cambie los valores de las variables. Esto quiere decir que esas variables no pueden cambiar de valor dentro

de la función. Esto es una herramiento muy poderosa pues evita muchos errores difíciles de encontrar.

... continua **Programa 2.8**

```
(gcf2.c)
#include <stdio.h>
#define MIN(A,B) ( (A) < (B) ? (A) : (B) )
int getgcf(int a, int b); /* computes greatest common factor */
int main()
{
   int a, b, c;
   while (1) {
      printf("Enter two integers (zeros to stop)\n");
      if ( scanf("%d %d", &a, &b) != 2){
         printf("***Error in input \n");
         fflush(stdin);
      } else if (a == 0 && b == 0)
         break;
      else {
         c = getgcf(a,b);
         printf("Greatest common factor = %d \n",c);
      }
   }
   return 0;
}

int getgcf(int a, int b)
/* MIN must be available as global macro */
{
   int i, imax;
   for (i=1; i<=MIN(a,b); i++){
      if (a % i == 0 && b % i == 0) imax=i;
   }
   return imax;
}
```

El comando `result(z)` indica que el resultado pasará de vuelta al programa como la variable `z`. En este caso no hay necesidad de definir o declarar una variable `getgcf`.

Caso C

En C, la función propia es `getgcf`. Un programa de $\mathbb{C}$ puede tener tantas funciones propias comose requiera y todas ellas pueden llamarse entre sí.

La linea justo antes del programa `main` es

```
int getgcf(int a, int b);
```

se conoce como *function prototype* y alerta al compilador que existe esta función y cuales son sus argumentos. El uso de *function prototype* fue introducido a ℂ para poder detectar errores. Los programas antiguos no incluyen esta característica y esto ahce que la función debe estar **arriba** del programa `main`, lo cual hace que si el usuario tiene 100 funciones propias, todas ellas deban ir antes del programa principal.

El termino `int` indica que la función retorna un número entero. Los argumentos se encuentran dentro del parentesis indicando el tipo de número que representan (enteros en este caso). También en este caso el nombre de las variables es indiferente, pero el tipo de variable si debe ser consistente entre programa principal y función.

A diferencia de 𝔽90 , los valores de `a` y `b` no son devueltos al programa principal, solamente el resultado `imax`. Por esto es que en 𝔽90 se prefiere usar `intent(in)` para evitar este tipo de problemas.

2.10. Subrutinas (solo `Fortran`)

En 𝔽90 , las funciones tienen una importante limitación, están desiñadas para pasar SÓLO un valor de vuelta al programa principal. Una opción más general es la *subrutina* que permite el paso de un número ilimitado de variables desde y hacia la subrutina.

Un ejemplo del uso de la subrutina se presenta en el Programa 2.9, donde a partir de la fecha juliana (Julian day o día del año) se determina el mes y el día del mes correspondiente.

La subrutina en este caso [Programa **??**] debe estar ubicada en el mismo archivo o código fuente del programa principal (`whatday`), y se llama así

```
call jday2mday(year,julian,month,day)
```

En este caso las variables `yeary` `julian` son pasadas del programa principal a la subrutina, y la subrutina devuelve las variables `monthy` `day` al programa principal. Note sin embargo que las variables de entrada NO son alteradas (`intent(in)) y que las variables de salida tienen {tt intent(out)` de tal manera que el compilador puede producir un error si estas variables de salida no existen.

Adicionalmente es una buena práctica de programación siempre documentar las subrutinas, funciones, etc. del usuario. Si hay aproximaciones, limitaciones, casos donde la subrutina o función fallan, es acá donde se debe poner claramente.

2.11. Enlace de subrutinas y funciones externas

Uno de los aspectos más útilies de las subrutinas y funciones propias es que pueden ser usadas por múltiples programas sin la necesidad de tener la subrutina en el código fuente de todos los programas. Es decir, el usuario puede tener **solo**

Programa 2.9 Programa para determinar la fecha (ao, mes día) a partir del año y dia del año.

```
(whatday.f90)
program whatday
   implicit none
   integer :: year, julian, month, day, i

   do
      print *, 'Type year and day of the year (zeros to exit) '
      read(5,*) year, julian
      if (year == 0 .and. julian == 0) exit
      call jday2mday(year,julian,month,day)
      print *, 'The date is ', year, '/', month, '/', day
   enddo
end program whatday

subroutine jday2mday(yr,jdy,mo,dy)
! Returns the month and the day of the month
! from the year and day in the year (julian day),
! Input:      year - integer (e.g., 1999)
!             jdy - integer, Julian day
! Output      mo - integer, month of the year
!             dy - integer, day of month

   implicit none
   integer, intent(in)  :: yr,jdy
   integer, intent(out) :: mo, dy
   integer :: i, lpyear
   integer, dimension(12) :: mondy

! Set first days of month
   mondy =  (/0,31,59,90,120,151,181,212,243,273,304,334/)
! Check if leap year
   if( (mod(yr,400) == 0).or.(mod(yr,4) == 0).or.   &
      (mod(yr,100) == 0) ) then
      lpyear = 1
   else
      lpyear = 0
   endif
! Set month and calendar day
   do i = 1, 12
      if (mondy(i) < jdy) mo = i
   enddo
   if (mo>2) then
      dy = jdy - mondy(mo) - lpyear
   else
      dy = jdy - mondy(mo)
   endif
end subroutine jday2mday
```

una copia o versión de una subrutina o función sin la necesidad de hacer *copy-paste*. Por ejemplo, más adelante se introducirá el tema de métodos de Fourier, para lo cual la transformada rápida de Fourier es básica. Lo ideal es que en un directorio se tengan todas las funciones relacionadas con Fourier y que nunca se tengan que editar, cambiar o copiar, simplemente se hace un *enlace* entre los programas y las funciones y subrutinas de Fourier.

El ejemplo anterior [Programa 2.9] puede ahora ser mucho mas corto. Al

Programa 2.10 Programa para determinar la fecha (ao, mes día) a partir del año y dia del año.

```
(whatday2.f90)
program whatday2
   implicit none
   integer :: year, julian, month, day, i

   do
      print *, 'Type year and day of the year (zeros to exit) '
      read(5,*) year, julian
      if (year == 0 .and. julian == 0) exit
      call jday2mday(year,julian,month,day)
      print *, 'The date is ', year, '/', month, '/', day
   enddo
end program whatday2
```

compilar sin embargo, toca indicar donde se encuentra la subrutina `jday2mday`. Lo que se quiere hacer es enlazar (*link*) algo denominado un *archivo objeto* con las funciones o subrutinas de interes. Los archivos objeto tienen terminación en `.o` y deben ser compilados a partir de un código fuente. En nuestro caso, el archivo `dates.f90`, ubicado en el folder `objs` tiene la subrutina (o función en el caso de C) así

```
subroutine jday2mday(yr,jdy,mo,dy)

! Subroutine returns the month and the day of the month
! from the year and day in the year (julian day),
! taking into account leap years.
...
end subroutine jday2mday
```

Para crear el archivo objeto, se digita entonces

```
pperez > gfortran -c objs/dates.f90 -o objs/dates.o
```

y para C

```
pperez > gcc -c objs/dates.c -o objs/dates.o
```

Una vez el objeto este compilado, se debe compilar el programa principal

```
pperez > gfortran whatday2.f90 objs/dates.o -o whatday2
```

o

```
pperez > gcc whatday2.c objs/dates.o -o whatday2
```

Esto también se puede personalizar en el `Makefile` de tal forma que no sea necesario saber de memoria donde se encuentra el archivo objeto de interés. Por ejemplo se puede editar el Makefile:

```
OBJS1=  objs/dates.o

 %: %.f90
     gfortran $< $(OBJS1) -o $*
```

[ver apéndice sobre Makefiles].

2.12. Arreglos (Arrays)

Los arreglos o *arrays* son, como su nombre lo indica, arreglos de números (o caracteres), como si fueran vectores o columnas de números. Se definen en Fortran

```
real, dimension(100) :: b
integer, dimension(50,2) :: index
```

y en C

```
float  b[100];
int index[50][2]
```

donde `b` es un arreglo de 100 elementos (b(1) a b(100)) para el primer caso y de b[0] a b[99] para el segundo (C). La variable index es un arreglo de 2-dimensiones de 50x2 elementos. $\mathbb{C}$ **asume que los índices del arreglo comienzan en cero**. Esta es una gran diferencia entre $\mathbb{F}$90 y $\mathbb{C}$, puede resultar muy confuso. Por ejemplo el elemento b[100] estaría fuera del rango en C, pero el elemento b(100) en $\mathbb{F}$90 es valido.

Note que en $\mathbb{C}$ los arreglos empiezan en 0 y en $\mathbb{F}$90 en 1. Además note que los elementos de los arreglos se escriben en parentesis en $\mathbb{F}$90 y con paréntesis cuadrados en C.

Una característica útil (pero que no recomendamos por que se presta para errores) es que en $\mathbb{F}$90 se puede especificar los límites del arreglo en cuento a sus índices, así

```
real, dimension(-100:100) :: a, b
integer, dimension(0:50, 2) :: index
```

en cuyo caso `a`y `b` son arreglos de 201 elementos (de a(-100) a a(100)) y de manera similar para arreglos multi-dimensionales.

El programa 2.11 muestra el uso de arreglos para determinar el número de primos que han en un rango específico. El método utilizado es a veces llamado la *criba de Erastóstenes*, un matemático Griego del 3er siglo AC.

El método comienza con un arreglo `prod` de 100 números (desde el 1 hasta el 100) y a todos se les considera primos y se les asigna un 0. Después se empieza con el número 2 (primo) y se buscan y eliminan todos sus múltiplos hasta 100 (si es múltiplo, se le asigna el valor de 1). El siguiente número es el 3 (primo) y se buscan y eliminan sus multiplos (asigna valor a 1). Así a medida que se va aumentando solo quedan con valor $prod = 0$ los números primos. Note por ejemplo que el 4, ya tiene el valor de 1 (multiplo de 2) por lo cual no es necesario buscar sus múltiplos. Solo es necesario mirar números hasta `sqrt(100)` ya que los factores más altos ya habrán sido eliminados.

Cuando el programa llega al final, se despliegan todos los números (la posición o índice) en `prod` que contenga el valor 0. Se cuantan cuantos hay y se despliegan los números primos. Note como en $\mathbb{C}$ se utiliza cada 10 valores primos el comando de linea nueva, en $\mathbb{F}$90 esto es posible pero más complicado. Se requeriría la creación de un nuevo vector donde se ubican los números primos a desplegar y se despliegan 10 a la vez [ver ejemplo en programa 2.12].

En el caso de $\mathbb{F}$90 este programa nuevamente se utiliza el `parameter` para que el programa tenga definido una única vez el tamaño de los arreglos.

```
integer, parameter :: maxnum=100
```

Si es necesario o el usuario así lo desea se cambia este valor una única vez para que el programa corra con 1000, 10000 puntos.

Note como es importante antes de calcular raiz cuadrada por ejemplo (o cualquier otra función trigonométrica o matemática), siempre convertir el número entero a un número real.

```
max_i = floor(sqrt(real(maxnum)))
```

En muchas funciones el argumento que debe entrar a la función debe ser real, o real(8).

Note como en $\mathbb{F}$90 definimos el número de valores que son desplegados en pantalla

```
print '(10i5)', pnum
```

Para asignar valores a un arreglo en $\mathbb{F}$90 se puede hacer de varias maneras

```
integer, dimension(3) :: x = (/ 1, 2, 3 /), y
x = (/ 15, 30, 40 /)
y = 2
```

donde `x` es definida en su asignación inicial `x = (/ 1, 2, 3 /)` o dentro del cuerpo del programa `x = (/ 15, 30, 40 /)`. La variable `y` que también tiene 3 valores, puede ser asignada sencillamente como `y = 2` lo cual equivale a `y =`

Programa 2.11 Programa calcula el número de primos, usando un arreglo.

```
(prime.f90)
...
   integer, parameter :: maxnum=100
   integer :: i, j, max_i, max_j, nprime
   integer, dimension(maxnum) :: prod

   nprime = 0
   prod = 0

   max_i = floor(sqrt(real(maxnum)))      !maxnum=100, max_i = 10

   do i = 2, max_i
      if (prod(i) == 0) then          ! for i=2 - 3 ....
         max_j = maxnum/i             ! maxj = 50  - 33 - 25 ...
         do j = 2, max_j              ! 2,50  - 3,33 - 4,25 - ....
            prod(i*j) = 1             ! 4,6,8....100
        enddo
      end if
  end do

   do i = 2, maxnum
      if (prod(i) == 0) then
         nprime = nprime + 1
         print "(i4)", i
      end if
   enddo
   print *, 'Number of primes found = ',nprime

(prime.c)
...
#define MAXNUM 100
...
   int i, j, prod[MAXNUM+1], nprime=0;
   for (i=1; i <= MAXNUM; i++) prod[i] = 0; /* set prod array to zero */
   for (i=2; i*i <= MAXNUM; i++){
   if (prod[i] == 0) {
      for (j=2; i*j <= MAXNUM; j++) prod[i*j] = 1;
      }
   }
   for (i=2; i <= MAXNUM; i++){
   if (prod[i] == 0) {
      nprime++;
      printf("%3d ", i);
      if (nprime % 10 == 0) printf("\n");
      }
   }
   printf("\n Number of primes found = %d \n",nprime);
...
```

Programa 2.12 Programa 𝔽90 que calcula el número de primos, usando un arreglo. Despliega diez números por linea (similar a la versión ℂ en Programa 2.11

```
(prime2.f90)
...
   integer, parameter :: maxnum=100, ndisp=10
   integer :: i, j, max_i, max_j, nprime, iloc
   integer, dimension(maxnum) :: prod
   integer, dimension(ndisp)  :: pnum

   nprime = 0
   prod = 0
...
   iloc = 0
   do i = 2, maxnum
      if (prod(i) == 0) then
         nprime = nprime + 1
         iloc = iloc + 1
         pnum(iloc) = i
         if (iloc == 10) then
            print '(10i4)', pnum
            iloc = 0
            cycle
         endif
      end if
   enddo
   print '(10i4)', pnum(1:iloc)
...

RESULTADO AL CORRER
pperez> prime2
   2   3   5   7  11  13  17  19  23  29
  31  37  41  43  47  53  59  61  67  71
  73  79  83  89  97
 Number of primes found =           25
```

`(/2, 2, 2/)`. Obviamente el número de valores entre `(/ /)` debe ser igual al del arreglo.

NOTA: Una posible variación del programa 2.12 sería utilizar el comando `write (*,"(i4)",advance='no')` por medio del cual al desplegar en la pantalla no se avanza la linea, y se puede entonces desplegar 10 dígitos por linea sin crear un nuevo vector.

2.12.1. Arreglos como argumentos de funciones y subrutinas

Suponga que se ha definido un arreglo `x` con 100 elementos en $\mathbb{F}$90 o $\mathbb{C}$:

```
integer :: x(100)
int x[100];
```

Si se usa `x` como argumentos en una funcion (C) o en una subrutina (Fortran) sin incluir los paréntesis

```
call SUMTOT(x,n,y)    (f90)
y = sumtot(x);         (C)
```

En ambos casos esto quiere decir que el arreglo se pasa de y hacia la funcion/subrutina por completo. En $\mathbb{F}$90 la subrutina debe tener definido un arreglo del mismo tamaño (o libre `dimension(:)`). En $\mathbb{C}$ `x` pasa como un `pointer` del primer elemento del arreglo (mas adelante ver sección de `pointers`). En C, a diferencia de argumentos que no son arreglos, cualquier cambio al arreglo dentro de la función genera un cambio en el arreglo en el programa principal. Recuerde que el comportamiento de $\mathbb{C}$ para variables en arreglos y variable (no arreglo) es distinto.

2.12.2. Arreglos en dos dimensiones

Arregos en 2-dimensiones (y en Fortran90 más) se definen así

```
integer, dimension(2,3) :: a          !F90 convention
int a[2][3];                          /* C convention */
```

por medio del cual se define una matriz en el cual el primer índice va de 1 a 2 y el segundo de 1 a 3. Recuerde que en $\mathbb{C}$ los arreglos comienzan con índice 0.

El programa 2.13 ilustra como se definen los elementos dentro de los arreglos tanto en $\mathbb{C}$ como en $\mathbb{F}$90 .

Para $\mathbb{F}$90 se utiliza al *array constructor* de la forma `(/ 1, 2, 3 /)` con una secuencia de valores en una sola dimensión. Por esto se hace necesario utilizarlo para cada dimensión de interés,

```
a(1, 1:3) = (/ 1, 2, 3 /)
```

Para $\mathbb{C}$ los arreglos se pueden inicializar cuando son definidos, y se pueden construir todos las dimensiones a la vez.

```
int a[4][3] = { {5, 3, 4},
{9, 1, 1},
{8, 7, 2},
{1, 1, 3} };
```

Programa 2.13 Programa sobre matrices en $\mathbb{F}$90 y $\mathbb{C}$.

```
(testmatrix.f90)
program testmatrix
   implicit none
   integer, dimension(2,3) :: a

   a(1, 1:3) = (/ 1, 2, 3 /)
   a(2, 1:3) = (/ 4, 5, 6 /)
   print *, a

end program testmatrix

(testmatrix.c)
#include <stdio.h>
int main()
{
   int i, j, k;
   int a[2][3] = { {1, 2, 3},
   {4, 5, 6} };
   for (i=0; i<2; i++){
      for (j=0; j<3; j++) printf("%d ",a[i][j]);
   }
   printf("\n");
   return 0;
}

pperez > testmatrix
   1   4   2   5   3   6
```

Note que se ocupan los espacios manteniendo el primer índice intacto y llenando los espacios del segundo índice. Una forma de entender esto, es pensar en la matrix `a[i][j]` como `(a[i])[j]`. De hecho la matrix se puede escribir así

```
int a[4][3] = {5, 3, 4, 9, 1, 1, 8, 7, 2, 1, 1, 3};
```

Finalmente, note como en $\mathbb{F}$90 se despliega l amatrix con el comando `print *,`, que los despliega en el orden como son guardados los elementos en memoria. $\mathbb{F}$90 , al imprimir guarda primero variando los valores de la primera dimensión, después la segunda, etc.

```
a(1,1), a(2,1), a(1,2), a(2,2), a(1,3), a(2,3)
```

Note que el usuario puede utilizar las matrices similar a Matlab, por ejemplo cuando se quiere llamar a una subrutina

```
call filter(a(:,i),npts)
```

donde `i` es el número de la columna de la matriz, y `npts` el número de elementos en dicha columna. La subrutina tendría una estructura

```
subroutine filter(c, n)
   ...
   real, dimension(n) :: c
   ...
```

Y aunque la forma alternativa `a(1,:)` también es valida, no es eficiente debido a la forma como se almacenan los elementos en una matriz.

Para $\mathbb{C}$ la asignación de valores es bastante simple.

```
int a[2][3] = { {1, 2, 3},
{4, 5, 6} };
```

Note que el usuario puede definir un vector (o matriz) así

```
int a[][3] = { {1, 2, 3},
{4, 5, 6} };
```

sin necesidad de determinar el primer índice. PRECAUCIÓN: Esto NO funciona `int a[][] = ....... .` I finalmente estas dos asignaciones son válidas y equivalentes

```
int a[2][3] = { {5, 3, 2}, {8, 0, 0} };
int a[2][3] = {5, 3, 2, 8};
```

Los programas [2.14 - 2.15] muestran un ejemplo de multiplicación de matrices en $\mathbb{F}$90 (dos versiones) y en C. $\mathbb{F}$90 posee una función intrínseca para la multiplicación de matrices [ver Programa 2.14]. Los ejemplos realizan la multiplicación de matrices, en $\mathbb{C}$ y $\mathbb{F}$90 . Tener en cuenta que en $\mathbb{F}$90 existe la función intrínseca.

2.13. Cadenas de caracteres

Una cadena de caracteres se puede declarar tanto en $\mathbb{C}$ como en $\mathbb{F}$90 , así:

```
character (len = 20) :: name      (Fortran)
char name[]="Bill Clinton";       (C)
```

donde la variable `name` es una cadena de caracteres de longitud 20 (fortran), libre (C). En C, para un caracter sencillo (cadena de 1 solo caracter) se debe definir

```
char grade;
grade = 'A';
```

Programa 2.14 Programa 𝔽90 para multiplicación de matrices.

```
program matmult

   implicit none
   real, dimension(3,3) :: a, b, c
   integer :: i, j, k

   a(1,1:3) = (/ -5.1,  3.8,  4.2 /)
   a(2,1:3) = (/  9.7,  1.3, -1.3 /)
   a(3,1:3) = (/ -8.0, -7.3,  2.2 /)

   b(1,1:3) = (/  9.4, -6.2,  0.5 /)
   b(2,1:3) = (/ -5.1,  3.3, -2.2 /)
   b(3,1:3) = (/ -1.1, -1.8,  3.0 /)

   do i = 1, 3
      do j = 1, 3
         c(i,j) = 0.0
         do k = 1, 3
            c(i,j) = c(i,j) + a(i,k) * b(k,j)
         enddo
      enddo
   enddo

   print *, "Matrix a follows"
   call printmat(a)

   print *, "Matrix b follows"
   call printmat(b)

   print *, "Matrix c = a*b follows"
   call printmat(c)

end program matmult

subroutine PRINTMAT(x)

   real, dimension(3,3) :: x

   do i = 1, 3
      print "(3f8.3)", (x(i,j), j=1,3)
   enddo

end subroutine PRINTMAT
```

continua Programa 2.14

```
program matmult

   implicit none
   real, dimension(3,3) :: a, b, c
   integer :: i, j, k
   ...
   c = matmul(a,b)

   print *, "Matrix a follows"
   ...
```

donde se usan las comillas sencillas y no las dobles (las dobles son para cadenas de caracteres). Otra característica en $\mathbb{C}$ es que no se puede realizar

```
name = "Juan Manuel Santos";    /* NO FUNCIONA!!! */
```

Uno podría poner los caracteres uno a uno

```
{\tt name[0]='J'; name[1]='u'; .... name[__]='\0'})
```

pero en este caso se recomienda usar la función `strcpy` (string copy) usando la librería <`string.h`> [ver Programa 2.16].

Fortran puede definir arreglos de cadenas de caracteres

```
   character (len = 20), dimension(100) :: name_array
```

y a diferencia de C, la variable de caracteres se puede definir

```
  name = "Juan Manuel Santos"
```

y los comillas sencillas si funcionan. Si los 20 espacios que se han definido no son utliizadas, se dejan como espacios en blanco. Si el texto es más largo que el espacio disponible, este se corta.

Caso Fortran

La longitud de una cadena de caracteres se puede obtener con el `len`,

```
len("Juan") da como resultado 4
len("  ") da  2
```

La función `len` no es siempre ideal, ya que si la variable está declarada como de longitud 20, `len(bariable)` va a dar como resultado 20, independiente del número de espacios en blanco. Para algo mejor, se utiliza `len_trim`

Programa 2.15 Programa ℂ para multiplicación de matrices.

```
(matmult.c)
#include <stdio.h>
#define DIM 3
int printmat(float a[DIM][DIM] );
int main()
{
   int i, j, k;
   float a[DIM][DIM] = { {-5.1, 3.8, 4.2},
   { 9.7, 1.3, -1.3},
   {-8.0, -7.3, 2.2} };
   float b[DIM][DIM] = { { 9.4, -6.2, 0.5},
   {-5.1, 3.3, -2.2},
   {-1.1, -1.8, 3.0} };
   float c[DIM][DIM];
for (i=0; i<DIM; i++){
   for (j=0; j<DIM; j++){
      c[i][j] = 0.0;
      for (k=0; k<DIM; k++) c[i][j] += a[i][k] * b[k][j];
      }
   }
   printf("Matrix a follows \n");
   printmat(a);
   printf("Matrix b follows \n");
   printmat(b);
   printf("Matrix c = a*b follows \n");
   printmat(c);
   return 0;
}

int printmat(float a[DIM][DIM] )
{
   int i, j;
   for (i=0; i<DIM; i++){
      for (j=0; j<DIM; j++) printf("%7.2f ",a[i][j]);
      printf("\n");
   }
   return 0;
}
```

Programa 2.16 Programa ℂ para asignación de cadenas de caracteres.

```
#include <stdio.h>
#include <string.h>
int main()
{
char name[81];
strcpy(name,"Alvaro Uribe");
printf("Nuestro presidente anterior fue %s \n",name);
strcpy(name,"Juan Manuel Santos");
printf("Nuestro presidente actualmente es %s \n",name);
return 0;
}
```

```
   ...
   character (len=20) :: name
   name = 'Juan Manuel'
   print *, len(name)
   print *, len_trim(name)
   ...
Resultado
pperez> test_len
          20
          11
```

Otra función intrínseca en 𝔽90 es `trim` que retorna una cadena de caracteres sin los espacios blancos al final.

Sub-cadenas se pueden obtener en fortran

```
name(1:4) es "Juan"
name(12:17) es "Santos"
name(6:6) is "M"
```

y por lo tanto se puede cambiar solo una parte de la cadena de caracteres

```
name(12:17) es "Uribe "
```

Note que se debe poner espacios en blanco para reemplazar la `s` de Santos. Esto cambia la posición de la cadena:

```
   name(2:13) = name(1:12)
```

PRECAUCIÓN: El caracter 13 no es alterado.

Para encontrar la posición de una cadena dentro de una variable se puede utilizar

```
   name = 'Santos'
   print *, index(name,'ant') es 2
```

```
print *, index(name,'tos') es 4
print *, index(name,'tas') es 0 (indicate que cadena no existe)
```

Para *pagar* dos cadenas de caracteres (concatenación) se usa el operador `//`:

```
text1 = "Juan Manuel Santos"  // " fue elegido en 2010."
```

para formar una frase más completa. Pero tenga cuidado que la longitud de la variable debe ser lo suficientemente larga. Note que entre el nombre y el resto hay un espacio en blanco (para no quedar con `Santosfue`.

Cuando las variables tienen espacios en blanco al final, por ejemplo la variable `nombre` fue declarada con 40 caracteres

```
name = "Juan Manuel Santos"
text1 = name // " fue elegido en 2010."
```

con resultado

```
Juan Manuel Santos              fue elegido en 2010
```

mientras que

```
text1 = trim(name) // " fue elegido en 2010"
```

```
Juan Manuel Santos fue elegido en 2010.
```

produce el resultado esperado.

El Programa 2.17 ilustra el uso de cadenas de caracteres en 𝔽90 (y C) con interacción del usuario. El comando para leer digitación del teclado

```
read "(a)", name
```

no tiene que definir cuantos carcteres de longitud tienen `name`, aunque `a80` sea valido. El uso de `free format read statement`:

```
read *, name
```

ya que solo se *lee* hasta el primer espacio en blanco, es decir con múltiples nombres como `Juan Manuel Santos`, solo se obtendría en la variable `name` hasta `Juan`.

Caso C

Una lista de comandos para el manejo de cadenas de caracteres en C. Se asume que las cadenas están ternminadas con `\n`.

```
strcpy(s1,s2)      copia s2 a s1, retorna s1
strncpy(s1,s2,n)   copia n caracteres de s2 a s1
strcat(s1,s2)      pega s2 al final de s1, retorna s1 modificado
strncat(s1,s2)     pega max n caracteres de s2 a s1
strlen(s1)         retorna longitud de s1, sin contar el \n
```

Programa 2.17 Programa para uso de caracteres en $\mathbb{F}90$ y $\mathbb{C}$.

```
(vote.f90)
program vote

   implicit none
   character (len = 80) :: name

   print *, 'Por qui\'en vot\'o en las elecciones presidenciales 2010?'
   read "(a)", name

   if (index(name,'ill') /= 0 .or.index(name,'lin') /= 0) then
      print *, "Then you are likely a Democrat."
   else if (index(name,'eor') /= 0 .or.index(name,'ush') /= 0) then
      print *,"Then you are likely a Republican."
   else
      print *,"Then you are likely an independent voter."
   end if

end program vote

(vote.c)
#include <stdio.h>
#include <string.h>
int main()
{
   char name[81];
   printf("Who did you vote for? \n");
   fflush(stdin);
   fgets(name,81,stdin);
   printf("Then you are likely a");
   if (strstr(name,"ill") != NULL || strstr(name,"lin") != NULL)
      printf(" Democrat \n");
   else if (strstr(name,"eor") != NULL || strstr(name,"ush") != NULL)
      printf(" Republican \n");
   else
      printf("n independent voter \n");
   return 0;
}
```

```
strcmp(s1,s2)      compara s1 y s2, con resultado
                   cero si s1 y s2 son iguales
                   valor negativo si s1<s2
                   valor positivo si s2>s1
strncmp(s1,s2,n)   similar a strcmp, compara max n caracteres
strstr(s1,s2)      retorna un pointer a la primera ocurrencia
                   de s2 en s1, NULL si no existe
```

El Programa 2.17 ilustra el uso de cadenas de caracteres en 𝔽90 (y C). En ℂ hemos optado por utilizar la función `fgets` para leeer lo digitado en el teclado (`stdin` en nuestro caso). El valor 81 nos dá el número máximo de caracteres a ser leidos. Algunos programas utilizan el comando `gets` para leer cadenas de caracteres, pero se corre el riesgo (si se digitan más de 81 caracteres) de sobreescribir en memoria.

`fgets` también tiene su desventaja ya que incluirá el caracter `\n` como el penúltimo caracter (justo antes de `\0` al final). La función `scanf` también puede ser utilizada para leer del teclado

```
scanf("%s",name);
```

con la desventaja que en nuestro ejemplo la lectura sólamente se haría hasta el primer espacio en blanco (ver caso similar en fortran). La segunda y demás palabras estarán en memorioa, por lo cual se recomienda el uso de `fflush(stdin)` para limpiar el *buffer*.

2.13.1. Arreglos de cadenas de caracteres

En 𝔽90 , una cadena de caracteres se puede considerar como un arreglo de cadenas de 1 caracter. Es posible entonces un arreglo 2D de caracteres, como por ejemplo:

El programa 2.18 puede guardar hasta 44 nombres, cada uno con máximo 80 caracteres. Note el uso de la función `trim`al desplegar los nombres para evitar demasiados espacios en blanco.

De manera similar en C, las cadenas de caracteres se pueden trabajar como arreglos. NOte que el último indices del arreglo DEBE ser la longitud de la cadena de caracteres deseada. En este caso, cada cadena se puede accesar con `name[0]`, `name[1]`, etc.

2.14. Input/Output (I/O)

Hasta ahora los ejemplos han involucrado entrada y salida de resultados del teclado y a la pantalla respectivamente. Ahora es necesario poder leer y escribir archivos (ascii o binarios). El programa [2.19] lee un archivo con dos columnas, los multiplica y guarda los números originales y el producto en un nuevo archivo. El nombre de los archivos se le pregunta al usuario.

Para abrir el archivo se usa el comando

Programa 2.18 Programa 𝔽90 con arreglos 2D de caracteres.

```
(stringarray.f90)

   implicit none
   character (len = 80), dimension(44) :: name
   ...
   name(5) = "Cesar Gaviria"
   name(4) = "Ernesto Samper"
   name(3) = "Andrs Pastrana"
   name(2) = "Alvaro Uribe"
   name(1) = "Juan Manuel Santos

   print *, "Nuestro presidente actual es ", trim(name(1))
   print *, "Recientes presidentes de Colombia "
   print *, trim(name(2))
   print *, trim(name(3))
   print *, trim(name(4))
   print *, trim(name(5))
   ...

(stringarray.c)
#include <stdio.h>
#include <string.h>
...
   char name[10][81];
   strcpy(name[4],"Cesar Gaviria");
   strcpy(name[3],"Ernesto Samper");
   strcpy(name[2],"Andres Pastrana");
   strcpy(name[1],"Alvaro Uribe");
   strcpy(name[0],"Juan Manuel Santos");
   printf("Nuestro presidente actual es %s \n",name[0]);
   printf("Recientes presidentes de Colombia \n",name[1]);
   printf("%s \n",name[1]);
   printf("%s \n",name[2]);
   printf("%s \n",name[3]);
   printf("%s \n",name[4]);
...
```

```
   open (11, file=infile, status='old')     (f90)
   fp1 = fopen(infile,"r");                          (C)
```

En 𝔽90 , los archivos que se abren se les asigna un *número de asignación* o `unit`, un número entero que en nuestro caso es `11`o `12`. En la mayoría de sistemas, `unit 5` está definido como standard a entrada del teclado, `unit 6` definido standard como salida a pantalla (desplegar), así que no se deben utilizar estos números. Habitualmente se usan valores desde el 11, 12, en adelante.

Programa 2.19 Programa con Input Output de archivos.

```
(fileinout.f90)
...
   implicit none
   character (len=100) :: infile, outfile
   real(4) :: x, y, z
   integer :: ios

   print *, 'Enter input file name'
   read "(a)", infile
   open (11, file=infile, status='old')
   print *, 'Enter output file name'
   read "(a)", outfile
   open (12, file=outfile)
   do
      read (11,*, iostat = ios) x, y
      if (ios < 0) exit
      z = x * y
      write (12,*) x, y, z
   enddo
   close (11)
   close (12)
...
```

El comando `status='old'` es opcional pero se recomienda su uso ya que confirma primero si el archivo existe. Si no existe el programa genera un error y se ejecución se termina, mientras que sin ese comando, 𝔽90 abre un nuevo archivo, y lee de ese archivo vacío, generando un error posterior que no es tan simple de entender. Otro `status` es `'new'` el cual ayuda a evitar sobreescribir archivos (note que en nuestro caso no se usa `open(12,file=outfile)`.

En C, para abrir y cerrar archivos se requiere definir `pointers` (ver mas adelante). Los `pointers` en el programa deben ser definidos, independiente de la definición de la cadena de caracteres del nombre de los archivos (`infile`, `outfile`).

```
FILE *fp1, *fp2;
```

Esto es diferente a 𝔽90 , ya que no se usa `unit`. Para pedirle al usuario el nombre de los archivos, se utiliza `scanf` ya que no se espera que el nombre del archivo tenga espacios. Como siempre `fflush(stdin)` antes de solicitar el segundo nombre. No se recomienda el uso de `fgets` porque adiciona un `\n` al final del nombre del archivo.

... continua **Programa 2.19**

```
(fileinout.c)
...
   FILE *fp1, *fp2;
   char infile[100], outfile[100];
   float x, y, z;
   printf("Enter input file name \n");
   scanf("%s",infile);
   fp1 = fopen(infile,"r");
   if (fp1 == NULL) {
      printf("***Can't open file: %s\n", infile);
      return 1;
   }
   printf("Enter output file name \n");
   fflush(stdin);
   scanf("%s",outfile);
   fp2 = fopen(outfile,"w");
   if (fp2 == NULL) {
      printf("***Can't open file: %s\n", outfile);
      return 1;
   }
   while (1){
   if (fscanf(fp1, "%f %f", &x, &y) == EOF) break;
      z = x * y;
      fprintf(fp2,"%8.3f %8.3f %8.3f \n", x, y, z);
   }
   fclose(fp1);
   fclose(fp2);
...
```

Para abrir finalmente el archivo:

```
fp1 = fopen(infile,"r");
```

donde el segundo argumento se usa para determinar permisos

```
"r" abre archivo existente para leer
"w" abre archivo existente o crea nuevo para sobreescribir
"a" abre archivo para editar, appending.
```

Se asume en todos los casos que el archivo es tipo `ascii`. Para archivos binarios usar `rb`, `rw` y `ra`.

Si el programa no puede abrir los archivos, el `pointer`se asigna `NULL` para lo cual se hace un control

```
if (fp2 == NULL) {
   printf("***Can't open file: %s\n", outfile);
   return 1;
}
```

y se retorna un valor diferente de 0.

Para leer las variables de los archivos en 𝔽90 se utiliza

```
read (11, *, iostat = ios) x, y                    (f90)
```

donde se utiliza nuevamente el `free-format read` para `unit 11`. EL `*`puede ser reemplazado por un formato específico si este se conoce(por ejemplo `'(i2,2f6.2)'`). Una complicación en 𝔽90 es que es necesario saber cuando se llega al final de un archivo (EOF por sus siglas en inglés). El comando `iostat=ios` asigna el valor de `0` normalmente, un valor positivo si hay algún error de lectura y un valor negativo si se ha llegado el final del archivo. Por esto se ejecuta

```
if (ios < 0) exit
```

para salir del `do` loop cuando se llegue al final del archivo. En programas antiguos (f77) se usaba el `goto`

```
read (11, *, end = 123) x, y
```

pero esto ya no es recomendado (aunque sigue siendo válido en f90).

Para guardar las variables en el nuevo archivo en 𝔽90 (usando el `free-format write` para `unit 12`.

```
write (12,*) x, y, z
```

Note que `x` y `y` no necesariamente van a tener el mismo formato que en el archivo original. Finalmente, después de llegar al final del archivo (y final del `do`loop, siempre se recomienda cerrar los archivos utilizados

```
close (11)
close (12)
```

Aunque esto se hace automáticamente cuando el programa concluye, es una buena práctica de programación.

En C, para leer de un archivo se utiliza el comando `fscanf`, que funciona de manera similar a `scanf` con el teclado.

```
if (fscanf(fp1, "%f %f", &x, &y) == EOF) break;
```

Note que cuando el final del archivo se alcanza `fscanf` retorna un `EOF` y en ese momento se puede culminar el `while`loop (sin esto el loop sería infinito). Las variables de entrada pueden ser asignadas como pointers (`&x` y `&y`). Para guardar el archivo de manera similar

```
fprintf(fp2,"%8.3f %8.3f %8.3f \n", x, y, z);
```

Para terminar, los archivos se cierran con `fclose` (igualmente no es obligatorio en nuestro caso, pero es buena práctica de programación).

2.14.1. I/O de manera rápida

Tanto 𝔽90 como ℂ pueden realizar grabación y lectura de archivos de manera muy rápida, por medio de operación de *read/write* con archivos binarios.

Programa 2.20 Programa con Input Output de archivos.

```
(testio.f90)
   implicit none
   real, dimension(100000,10) :: a = 1.1
   integer :: i, j

   open (12, file='testio1.dat')
   do i = 1, 100000
      write (12, '(10e12.4)') (a(i,j), j=1,10)
   enddo
   close (12)

   open (12, file='testio2.dat', form='unformatted')
   do i = 1, 100000
      write (12) (a(i,j), j=1,10)
   enddo
   close (12)

   open (12,file='testio3.dat',form='unformatted')
   write (12) a
   close (12)
...
(testio.c)
NATALIA GOMEZ PONER VERSION C ACA
```

Suponga que tenemos un arreglo de 100000×10 que deseamos guardar en el disco duro [ver Programa 2.20]. Para el primer caso, guardando el archivo como *ascii*, se utilizan 12 MB de memoria (3 veces el espacio para número real de 4-bytes) y tarda 1 segundo en un Mac Portátil (2.7 GHz Intel i7). Sólamente en el primer ejemplo se muestra el uso de la función intrínseca `system_clock` para saber cuanto tarda una sección del programa.

Resulta más eficiente tanto en cuanto a tiempo como en cuanto a espacio de disco guardar los datos en formato binario

```
   open (12, file='testio2.dat', form='unformatted')
```

```
   do i = 1, 100000
      write (12) (a(i,j), j=1,10)
...
```

El tamaño usado es de 4.6 MB, un poco mas de los 4MB esperados. El tiempo requerido es de 0,14 segundos, un factor de 7 más veloz. Y una mejora mas se puede obtener si se guarda el arreglo entero directamente:

```
      open (12,file='testio3.dat',form='unformatted')
      write (12) a
      close (12)
```

ocupando 4,000,008 bytes en un tiempo record de 0,001 segundos.

En algunos casos los datos que se tienen no son exclusivamente de un solo tipo (enteros, caracteres, reales) para lo cual se puede introducir las `estructuras`, que en el caso de 𝔽90 fueron introducidas en Fortran90. Ms adelante sobre esto.

2.14.2. Ascii vs Binary files

En el caso de archivos binarios y ascii existe un *trade-off*. Los archivos ascii son fáciles de leer y faciles para trabajar ya que cualquier editor de texto los puede abrir y el usuario puede leer el archivo y entender su formato. Si el problema es pequeño o la velocidad de procesamiento no es importante, este es el formato de los datos que se recomienda. Sin embargo, para bases de datos muy grandes o el tiempo de procesamiento es importante, los archivos binarios son los ideales, ya que permiten I/O muy rápido y con tamaño de memoria requirida mucho menor. Tiene como desventaja que el usuario debe saber con precisión el formato (por ejemplo $\%f14{,}5$) y si hay `byte-swap` dependiendo de la arquitectura del sistema (ver little-endian vs big-endian en Google!).

2.15. Estructuras

En algunos casos es útil tener un arreglo de variables de diferente tipo que el usuario pueda manipular. Las variables en este caso prodrían ser de diferente tipo, una real, una entera, una cadena de caracteres, etc. En ℂ esto se le conoce como *Structures* o estructuras y en Fortran90 como *Derived Data Type*.

Suponga que se quiere tener una base de datos con la información de los estudiantes de un curso y varias características de cada persona, incluyendo el nombre, la altura, el peso, la edad, etc. Para esto se requeriría que cada una de esas características tenga una variable asignada. El programa 2.21 es un ejemplo del uso de estructuras en ℂ y 𝔽90 y puede definir la persona con menor peso.

Para crear una estructura y un `defined type` en ℂ y 𝔽90 respectivamente se utiliza el comando `typedef struct` o `type person` respectivamente. La estructura tiene una cadena de caracteres de 20 espacios, edad (entero) y altura y peso (float o real). Cada una de estas variables se les conoce como `miembros` o `componentes` de la estructura.

Programa 2.21 Programa para el manejo de estructuras en $\mathbb{C}$ y $\mathbb{F}$90 .

```
(namebase.f90)
...
   integer, parameter :: nmax=3
   type person
      character (len=20) :: name
      integer :: age
      real :: height, weight
   end type person
   type(person), dimension(nmax) :: student
   integer :: i
   real :: weightmin

   do i = 1, nmax
      print *, 'Enter first name, age, height, weight '  &
               '(e.g., Bob 27 69 183)'
      read *,   student(i)%name,   student(i)%age,     &
                student(i)%height, student(i)%weight
   enddo
   print *, 'Lightest student = ',    &
          student(minloc(student%weight))%name
...
(namebase.c)
...
#define NAMELENGTH 12
#define NMAX 5
...
   typedef struct { char name[NAMELENGTH];
      int age;
      float height, weight;
      } person_type;
   person_type students[NMAX];
   int i,ii;
   float weightmin;
   for (i=0; i<NMAX; i++){
      printf("Enter first name, age, height, weight ");
      printf("(e.g., Bob 27 69 183) \n");
      scanf("%s %d %f %f", &students[i].name,
         &students[i].age,
         &students[i].height,
         &students[i].weight );
   }
      ...  /* find lightest person */
   }
   printf("%s is the lightest student \n",students[ii].name);
...
```

```
type person (f90)
   character (len=20) :: name
   integer :: age
   real :: height, weight
end type person

typedef struct { char name[NAMELENGTH]; (C)
   int age;
   float height, weight;
   } person_type;
```

En nuestro ejemplo es importante tener en cuenta que la definición de una estructura no es suficiente, y se debe asignar o definir una variable que tenga esa estructura, `student`en nuestro caso.

```
type(person), dimension(nmax) :: student   (f90)
person_type students[NMAX];                (C)
```

Note como en ambos caso, la definición de una variable como estructura es similar a definir la variable como `real`, o `int`. Se pueden definir más de una variable como estructura, e incluso se puede definir un arreglo de estructuras:

```
(f90)
   type(person), dimension(nmax) :: grads, undergrads
   type(person) :: teacher
(C)
   person_type grads[NMAX], undergrads[NMAX], teacher;
```

donde `teacher` es una estructura sencilla y `grads`y `undergrads`son arreglos de estructuras (aunque todas son el mismo tipo de estructura).

Los miembros de una estructura son referenciados de manera diferente en C, 𝔽90 e incluso Matlab. Para 𝔽90 los miembros de una estructura se referencian pegando `%miembro`, después del nombre de la variable, donde `miembro`se refiere al miembro específico. En C, se adiciona `.miembro` al final del nombre de la variable.

Por ejemplo, `student(1)%age` o `student[1].age` es la edad del estudiante 1. Note que el índice del arreglo va ANTES que el `%miembro` o antes de `.miembro`. En 𝔽90 el `%` se le conoce como el *component selector*, en ℂ el operador punto `.` se le conoce como *structure member operator* o el `member access operator`.

Note que en 𝔽90 el uso de funciones intrínsecas como `minloc`o `maxloc` permiten ubicar los máximos o mínimos de manera rápida, sin la necesidad de un `do`loop, lo cual no existe en C.

Aunque el programa 2.21 sirve como ejemplo del manejo de las estructuras, no muestra la ventaja de usar estructuras. Hay múltiples ventajas, incluyendo el poder *mover* todos los miembros de una estructura con un solo comando o guardar en un archivo binario una estructura, que al leerla posteriormente tiene toda la información de la estructura. Esto permite entonces tener i/o de manera rápida al guardar las estructuras, teniendo todavía la capacidad de accesar las partes o miembros de las estructuras.

2.16. Números Complejos

Una de las grandes ventajas de $\mathbb{F}90$ (y Matlab) es que los números complejos son una característica standard, mientras que $\mathbb{C}$ no posee un método directo para manipular variable compleja. Desde el punto de vista científico esto es una gran desventaja, ya que la variable compleja es esencial en muchos problemas en las ciencias. En $\mathbb{C}$ afortunadamente, por medio del uso de estructuras, es posible implementar funciones propias del usuario para aritmética compleja.

Programa 2.22 Manejo de variable compleja en $\mathbb{F}90$.

```
program testcomplex

   implicit none
   complex :: a, b, c

   print *, 'Enter first complex number'
   read *, a
   print *, 'Enter second complex number'
   read *, b
   c = a*b
   print *, 'Product = ', c
   print *, 'abs of product = ', abs(c)
   print *, 'sqrt of product = ', sqrt(c)

end program testcomplex
```

El programa 2.22 muestra de manera sencilla el uso de variable compleja en $\mathbb{F}90$. Note que los complejos deben ser digitados así:

```
(x, y)
```

en paréntesis y separados con una coma. También se puede ver como las funciones intrínsecas de $\mathbb{F}90$ pueden actuar sobre variable compleja, sin necesidad de hacer ningún cambio. Un resultado ejemplo es

```
pperez > testcomplex
 Enter first complex number
(1, 1)
 Enter second complex number
(-5, 1)
 Product =  ( -6.0000000    , -4.0000000    )
 abs of product =    7.2111025
 sqrt of product =  ( 0.77817172    , -2.5701268    )
```

PRECAUCIÓN: Niinguno de los siguientes es válido

```
            1, 1
o           1  1
o incluso  (1  1)
```

y se obtiene un error.

También es posible a partir de variables reales, crear un número complejo, o al contrario, es decir a partir de un número complejo extraer su componente real e imaginario como variables reales [ver Programa 2.23]. y su resultado en

Programa 2.23 Manejo de variable compleja en $\mathbb{F}90$.

```
program testcomplex2

   implicit none
   complex :: a
   real :: ar, ai

   print *, 'Enter real part'
   read *, ar
   print *, 'Enter imaginary part'
   read *, ai

   a = cmplx(ar, ai)
   print *, 'a = ',a
   a = exp(a)
   print *, 'Exp(a) = ', a
   ar = real(a)
   ai = aimag(a)

   print *, 'Real part = ', ar
   print *, 'Imag part = ', ai

end program testcomplex2
```

pantalla

```
pperez > testcomplex2
 Enter real part
1.
 Enter imaginary part
2.
 a =  (  1.0000000    ,  2.0000000    )
 Exp(a) =  ( -1.1312044    ,  2.4717267    )
 Real part =   -1.1312044
 Imag part =    2.4717267
```

En el caso de C, es necesario utilizar funciones que puedan manejar variables complejas. Para esto, utilizamos una serie de funciones de dominio público disponibke en *Numerical Recipes* a través del archivo `complex.h` o `complex.c` (ver Apéndice B). Las rutinas o funciones tienen la siguiente forma

```
typedef struct FCOMPLEX {float r,i;} fcomplex;
fcomplex Cadd(fcomplex a, fcomplex b);
   retorna la suma de dos complejos

fcomplex Csub(fcomplex a, fcomplex b);
   retorna la diferencia entre dos complejos

fcomplex Cmul(fcomplex a, fcomplex b);
   producto de dos complejos

fcomplex Complex(float re, float im);
   genera un complejo a partir de dos floats (real e imaginario)

fcomplex Conjg(fcomplex z);
   Conjugado complejo de variable complejo

fcomplex Cdiv(fcomplex a, fcomplex b);
   dividir dos complejos

float Cabs(fcomplex z);
   valor absoluto de complejo

fcomplex Csqrt(fcomplex z);
   retorna raiz cuadrada compleja

fcomplex RCmul(float x, fcomplex a);
   retorna el producto complejo de dos variables complejas
```

y el Programa 2.24 muestra el uso de estas funciones como ejemplo

Note que la *estructura* `fcomplex`está definida en `complex.c` así que no es necesario que aparezca dentro del programa. Sólo es necesario definir las variables como complejas

```
fcomplex a, b, c;
```

Los valores real e imaginario de las variables (por eejmplo `b` se accesan con `b.r` y `b.i`. Estas funciones o rutinas permiten la manipulación de variable compleja en $\mathbb{C}$ cas´tan fácil como en $\mathbb{F}90$. La 'nica desventaja sería que el código fuente es más complicado de leer.

Programa 2.24 Manejo de variable compleja en ℂ .

```
(complexmult.c)
#include <stdio.h>
#include <math.h>
#include "complex.c"
int main()
{
   fcomplex a, b, c;
   float x, y, cabs;
   printf("Enter 1st complex number (real + imag) \n");
   scanf("%f %f", &x, &y);
   a = Complex(x, y);
   printf("Enter 2nd complex number (real + imag) \n");
   scanf("%f %f",&b.r, &b.i);
   c = Cmul(a, b);
   cabs = Cabs(c);
   printf("product = (%f, %f), cabs = %f \n", c.r, c.i, cabs);
   return 0;
}
```

2.17. Otras opciones en ℂ

2.17.1. Pointers

Aunque los `pointers` actualmente también existen en 𝔽90 , son una característica fundamental en ℂ . Cualquier variable definida en un programa de ℂ es guardada en alguna parte de la memoria del computador. Su `pointer` es simplemente la dirección de la ubicación de ese espacio de memoria. TODOS los programas y lenguajes usan `pointers`, solo que algunos (como 𝔽90 o Matlab) lo hacen detrás de bambalinas.

ℂ es un lenguaje muy orientado al uso de `pointers` y es necesario entender como funcionan para poder manejar ℂ apropiadamente. Un `pointer` de una variable se referencia al adicionar como prefijo un `&`, por ejemplo `&x` se refiere al `pointer` de la variable llamada `x`. En un programa

```
x = valor de x
&x = pointer (memoria) para x
```

Es también posible definir `pointers` y darles un nombre regular (son el &). Los pointers deben referirse a variables específicas (int, float, char, etc.) y esto debe estar definido cuando se declara el `pointer`. así se definiría un `pointer` a un entero

```
pointer xpt
int *xpt
```

Programa 2.25 Manejo de `pointers` en ℂ .

```
(testpoint.c)
#include <stdio.h>
int main()
{
int x, y;
int *pt;
x = 1;
pt = &x; /* pt1 now points to x */
y = *pt; /* y is now set to 1 */
*pt = 2; /* x is now set to 2 */
printf("%d %d %d \n",x,y,*pt);
return 0;
}
```

El programa 2.25 ilustra el uso de `pointers` de manera sencilla con resultados como se esperaría.

```
2 1 2
```

Se pueden utilizar `pointers` si se desea que una función cambie los valores de variables (algo que ℂ no hace (solo si es un arreglo).

El programa 2.26 muestra el uso de estos conceptos para calcular tanto el área como la circumferencia de un círculo.

La variable `r` pasa de manera normal hacia la función, sin embargo el área y la circumferencia `area, circum` pasan a través de su `pointer` (es decir la posición en la memoria). Cuando estos valores son cambiados dentro de la función, también son cambiados en el programa principal. Esto es útil cuando lo que se quiere de una función es que de como resultado más de una sola variable (similar a una subrutina en 𝔽90).

2.17.2. Pointers y Arreglos

Los pointers y los arreglos están intimamente ligados en ℂ . El nombre de un arreglo es de hecho un pointer al primer elemento del arreglo. Por ejemplo, si se define un arreglo `x`

```
int x[5];
```

entonces un simple `x` (sin índice) es un `pointer` el primer valor del arreglo, `x[0]`. Si se le agrega 1 al `pointer` entonces mostrara el siguiente elemento del arreglo como lo muestra el Programa 2.27.
El resultado de este programa es:

```
pperez > testarraypt
11 12 13
11 12 13
```

Programa 2.26 Manejo de `pointers` en ℂ .

```
(circle.c)
#include <stdio.h>
#define PI 3.1415927
int areacircum(float r, float *area, float *circum);
int main()
{
   float r, area, circum;
   r = 1.33;
   areacircum(r, &area, &circum);
   printf("r, area, circum = %f %f %f \n", r, area, circum);
   return 0;
}

int areacircum(float r, float *area, float *circum)
{
   *area = PI * r*r;
   *circum = 2. * PI * r;
   return 0;
}
```

Pero, cómo sabe el programa cuál es la siguiente dirección en la memoria para el siguiente índice? La respuesta depende del tipo de variable que se le asigne al `pointer`. Si por ejemplo es un arreglo de enteros, usualmente se moverá 4 bytes, si es un arreglo de caracteres solo se moverá 1 byte, etc. Afortunadamente el compilador ℂ sabe el tipo de variable y cuanto debe moverse, y por esto es *fundamental* declarar los `pointers` a el tipo de variable al que se refieren.

Programa 2.27 Manejo de `pointers` en ℂ .

```
(testarraypt.c)
#include <stdio.h>
int main()
{
   int z[5] = {11, 12, 13, 14, 15};
   int *pt;
   pt = z;
   printf("%d %d %d \n", *pt, *(pt+1), *(pt+2) );
   printf("%d %d %d \n", *z, *(z+1), *(z+2) );
   return 0;
}
```

También es posible usar operadores como `++` o `--` con `pointers` de manera similar como con números enteros, por ejemplo

```
pt++;
```

mueve el `pointer` al siguiente elemento del arreglo. Esto demuestra el porque al pasar un arreglo a una función, los cambios hechos dentro de la función afectan los valores del arreglo dentro de del programa principal (los arreglos pasan a las funciones como `pointers`.

PRECAUCIÓN: Los compiladores ℂ interpretan comendos

```
x[10]
```

como

```
*(x+10)
```

sinverificar si `x` tiene las dimensiones suficientes, es decir que el valor en memorioa puede no estar asignado a `x`. No hay una manera simple de verificar si el arreglo se salio de sus límites.

2.17.3. Pointers vs índices de arreglos

En ℂ es posible hacer operaciones con arreglos por medio de dos caminos,

1. `pointer`
2. índice del arreglo

En algunos textos se sugiere que el uso de `pointers` hace que el código corra más rápido, pero hoy en día y con compiladores modernos la diferencia es mínima e incluso en algunos casos que hemos hecho, el manejo de arreglos con índices es más veloz (aunque por poco).

El programa 2.28 muestro dos rutinas para suma de valores de un arreglo de tamaño X, con resultados en valocidad muy similares independiente del número X. Es probable que los compiladores actuales optimicen el código en ambos casos. Nosotros recomendamos el uso de índices, ya que la programación con índices es similar a la matemática detrás de los algoritmos y esto hace que el código fuente sea más fácil de leer.

2.18. Otras opciones en 𝔽90

2.18.1. Procedimientos internos

Las funciones y subrutinas en 𝔽90 en general se les conoce como externas ya que se ubican por fuera del programa principal, ya sea como un objeto, librería o dentro del mismo código fuente pero por fuera de (begin-end) program. Con rutinas externas, todas las variables que pasan del programa a la rutina tienen que hacer parte de la lista de argumentos al llamar a la rutina.

En algunos casos sin embargo es útil tener dichas rutinas internas. Estas rutinas estan **antes** del final del programa, antes de `end program`. La ventaja

Programa 2.28 Comparación de velocidad entre manejo de arreglos con `pointers` y con índices.

```
float sumarray1(float a[], int n){
   int i;
   float sum=0.0;
   for (i=0; i<n; ++i) sum = sum + a[i];
   return sum;
}

float sumarray2(float a[], int n){
   float *point, *pointmax;
   float sum=0.0;
   point = a; /* pointer for a[0] */
   pointmax = point + n - 1;
   while (point <= pointmax){
      sum += *point++;
   }
   return sum;
}
```

es que los nombres de las variables son compartidas y por lo tanto la lista de argumentos no es necesaria. Los procedimientos internas se adjuntan después del comando `contains`. Las variables declaradas en el programa principal no requieren ser declaradas en la rutina o función.

Nuestra experiencia nos sugiere que el uso de procedimientos internos en muchos casos hace que los programas sean más complicados de entender (ya que no se sabe que entra y que sale de una rutina por ejemplo) y solo se ha visto utilidad cuando hacen parte de un gran número de rutinas que requieren enlaces complicados (algo que está más allá del objeto de este texto).

2.18.2. Precisión doble

De manera standard $\mathbb{F}90$ guardaa variables reales usando 4-bytes, haciendo que dichas variables tengan una precisión de 6 a 7 decimales, con números entre 1.175494e-38 y 3.402823e+38. En algunos casos esta precisión es insuficiente para el problema que se está resolviendo y se requiere de variables con mayor precisión. Esto se puede lograr en $\mathbb{F}90$ por medio de la declaración `real(8)` y en código fuente antiguo `double precision`. El programa 2.29 muestra un ejemplo del uso de estas variables. cuyos resultados en pantalla se muestran a continuación:

```
a4 =   8.899999618530273437500000000000000000000
a8 =   8.900000000000000355271367880050092935562l
aa =   8.900000000000000355271367880050092935562l
```

```
k4 =   8.899999618530273437500000000000000000000
k8 =   8.900000000000000355271367880050092935562l
k16=   8.900000000000000000000000000000003081488
```

y donde `a4` y `k4` son variables `real(4)` y aproximan el valor `8.9` con el número `8.8999996`. Mayor precisiń se puede obtener al declarar valores `real(8)` como se ve en los resultados, e incluso algunos compiladores pueden declarar `real(16)` o cuadruple precisión (note que los compiladores sin costo como `gfortran`o `g95` no permiten cuadruple precisión todavía.

Programa 2.29 Programa de 𝔽90 para generar números reales con doble precisión.

(testdouble.f90)

```
   implicit none
   character (len = 30) :: fmt = "(a5,f44.40)"
   real    :: a4
   real(8) :: a8
   double precision aa

   real (kind=4)  :: k4
   real (kind=8)  :: k8
   real (kind=16) :: k16

   a4 = 8.9
   print fmt, 'a4 = ', a4
   a8 = 8.9d0
   print fmt, 'a8 = ', a8
   aa = 8.9d0
   print fmt, 'aa = ', aa

   k4 = 8.9
   print fmt, 'k4 = ', k4
   k8 = 8.9_8
   print fmt, 'k8 = ', k8
   k16 = 8.9_16
   print fmt, 'k16= ', k16
```

El programa muestra varias formas como se pueden declarar las variables con la precisión deseada, pero note ante todo que para una variable de doble precisión se requiere digitar `d0` para que el compilar sepa que se trata de un valor con doble precisión. También es posible definir la variable compleja como doble precisión!!

2.18.3. Aritmética en Arreglos en F90

$\mathbb{F}$90 permite la aplicación de operaciones sobre vectores e incluso matrices (muy similar a como se hace en Matlab) con un solo comando y sin la necesidad de hacer `do` loops sobre los índices del vector o matriz. Algunas operaciones se muestran en el Programa 2.30.

F90 allows many operations on vectors and matrices to be performed in single statements without the necessity of writing do loops over the array indices (as was necessary in F77). Here is an example program that demonstrates some of these operations on vectors. Note que $\mathbb{F}$90 puede distinguir entre vectores y escalares y puede fácilmente adaptar la operación a lo que requiere el usuario.

Operaciones en matrices funcionan de manera similar en el Programa **??** Se usan operaciones como `matmul`, `maxval` y `maxloc`, `transpose` y `size` con resultados obvios. Note que `matmul(a,b)` es la multiplicación matricial y no equivale a `a*b` que representa la multiplicación de los elementos individuales de las matrices.

La función `maxloc` devuelve la posición del valor máximo (tambi'en hay `minloc`),

```
loc2 = maxloc(a23)
```

donde `loc2` es un arreglo de enteros con `dimension(2)`. PRECAUCIÓN: Si `maxloc` se aplica a un arreglo 1D, el ejemplo

```
loc = maxloc(a)
```

debe tener `loc` como un arreglo `dimension(1)` (no puede ser una variable entera adimensional).

2.18.4. Asignación de Memoria dinámica (Allocatable)

Una de las nuevas características presenten en $\mathbb{F}$90 es que se puede asignar memoria a arreglos durante o en medio de un programa y no se tiene que preasignar. En inglés esto se la llama *dynamic memory allocation* y en $\mathbb{C}$ se usa `malloc`. El Programa 2.32 muestra un ejemplo del uso de memoria dinámica en $\mathbb{F}$90 .

Aunque de gran ventaja, es importante tener en cuenta que el acceso de variables `allocatable` en $\mathbb{F}$90 puede ser más lento, por lo cual se recomiendo solo utilizarlo si es necesario (aunque en nuestra experiencia en muchos casos lo es).

2.18.5. Módulos

Los módulos o `modules` son otra adición a $\mathbb{F}$90 y pueden ser de gran utilizadad, por ejemplo para generar una `Explicit interface` entre argumentos que pasan de un programa a una subrutina.

Suponga que tiene la siguiente subrutina que teien como tarea desplegar los valores de sus argumentos:

Programa 2.30 Programa de 𝔽90 sobre aporaciones en arreglos

```
program vectormath

   implicit none
   real, dimension(5) :: a = (/ 1.0, 2.0, 3.0, 4.0, 5.0 /), b = 2.0, c
   real :: x

   print *, 'a = ', a

   c = a + 1
   print *, 'a + 1 = ', c

   c = 2 * a
   print *, '2 * a = ', c

   c = a * a
   print *, 'a * a = ', c

   c = sqrt(a)
   print *, 'sqrt(a) = ', c

   c = sin(a)
   print *, 'sin(a) = ', c

   c = exp(a)
   print *, 'exp(a) = ', c

   print *, 'b = ', b
   c = a + b
   print *, 'a + b = ', c

   c = a * b
   print *, 'a * b = ', c

   x = sum(a)
   print *, 'sum(a) = ', x

   c = a
   c(4:5) = 0.0
   print *, 'a with two zeros on end = ', c

   x = dot_product(a, b)
   print *, 'a dot b = ', x

   x = sum( (a - sum(a)/5. )**2)
   print *, 'sum of squares of difference from mean = ', x

end program vectormath
```

Programa 2.31 Programa de $\mathbb{F}90$ sobre aporaciones en arreglos 2D.

```
program matrixmath

   implicit none
   real, dimension(2, 3) :: a23, b23, c23
   real, dimension(3, 2) :: a32, b32, c32
   real, dimension(2,2) :: a22, b22, c22
   integer, dimension(2) :: loc2
   integer :: i, j, k

   a23(1,1:3) = (/ -5.1,  3.8,  4.2 /)
   a23(2,1:3) = (/  9.7,  1.3, -1.3 /)
   print *, 'Matrix a23 follows'
   call PRINTMAT(a23, 2, 3)

   b32(1:3, 1) = (/  9.4, -6.2,  0.5 /)
   b32(1:3, 2) = (/ -5.1,  3.3, -2.2 /)
   print *, 'Matrix b32 follows'
   call PRINTMAT(b32, 3, 2)

   c22 = matmul(a23, b32)        !this works but matmul(b32, a23) does not
   print *, "Matrix c22 = matmul(a,b) follows"
   call PRINTMAT(c22, 2, 2)

   print *, 'maxval(a23) = ', maxval(a23)
   print *, 'maxloc(a23) = ', maxloc(a23)
   loc2 = maxloc(a23)
   print *, 'loc2 = ', loc2
   print *, 'a23(loc2(1), loc2(2)) = ', a23(loc2(1), loc2(2))

   b23 = a23 + transpose(b32)
   print *, 'Matrix b23 = a32 + transpose(b32) follows'
   call PRINTMAT(b23, 2, 3)

   print *, 'size(a23) = ', size(a23)
   print *, 'size(a23(1,:)) = ', size(a23(1,:))

end program matrixmath

subroutine PRINTMAT(x, m, n)
   real, dimension(m, n) :: x
   integer :: m, n

   do i = 1, m
      print "(10f8.3)", (x(i,j), j=1,n)
   enddo
end subroutine PRINTMAT
```

Programa 2.32 Uso de memoria dinámica en 𝔽90 .

```
(setarray.f90)
program setarray

   implicit none
   real, dimension(:), allocatable :: x
   real, dimension(:, :), allocatable :: y, yy
   real :: xsum
   integer :: n, i, j, m

   print *, 'Enter number of points'
   read *, n
   allocate (x(n))

   do i = 1, n
      call random_number(x(i))
   enddo

   xsum = sum(x)
   print *, 'sum = ', xsum
   deallocate(x)        !this frees up memory

   print *, 'Enter m, n (# rows, # columns)'
   read *, m, n

   allocate (y(m,n))
   allocate (yy(n,n))

   do i = 1, m
      print *, 'Enter row # ', i
      read *, (y(i, j), j = 1, n)
   enddo

   yy = matmul(y, transpose(y))
   print *, 'y * yt = '

   call PRINTMAT(yy, m, m)

   deallocate(y)
   deallocate(yy)

end program setarray
```

```
subroutine print_numbers( r, i )

   implicit none
   real, intent(in) :: r
   integer, intent(in) :: i

   print *, "Real number is ", r
   print *, "Integer number is ", i

end subroutine print_numbers
```

y el programa principal sería

```
...
   real_number = 42.0
   integer_number = 42

   call print_numbers( real_number, integer_number )
...
```

el resultado es como el esperado. Pero si por alguna razón el usuario comete un error y digita en el programa principal

```
   call print_numbers( integer_number, real_number )
```

el resultado sería

```
Real number is  5.9E-44
Integer number is  1109917696
```

claramente equivocado. Note que al compilar no hay información sobre los argumentos que deben entrar en la subrutina, por lo tanto al compilar el programa principal no se genera error alguno.

Este problema se puede eviatar creando un módulo (en vez de una subrutina sencilla)

```
module arguments_wrong
...
contains
subroutine print_numbers( r, i )

   implicit none
   real, intent(in) :: r
   integer, intent(in) :: i
...
end subroutine ...
end module
```

lo cual puede escribirse en el encabezado el programa o como un archivo de código de fuente independiente (que debe ser compilado claro está).

```
gfortran -c arguments_wrong.f90
```

generando un archivo `arguments_wrong.mod`. El programa principal debe entonces *LLAMAR* al módulo

```
program ...

   use arguments_wrong
   implicit none
   real :: real_number
   integer :: integer_number
...
   call print_numbers( integer_number, real_number )


end program ...
```

Al compilar este programa se genera un error

```
Error: Type mismatch in parameter 'r' at (1).
   Passing INTEGER(4) to REAL(4)
```

Este ejemplo muestra una de las ventajas de usar módulos, ya que la `explicit interface` permite diagnosticar problemas antes de tiempo.

Los módulos pueden ser usados para *llaver* información sobre variables globales es decir que todos los programas o subrutinas que utilicen el módulo tendría las mismas variables disponibles y si alguna de las subrutinas cambia el valor de dicha variable, cambia para todas. PRECAUCIÓN: Nosotros NO recomendamos el uso de módulos con este fin, ya que aunque muy útil, hace que sea muy complicado entender el programa por un tercero.

Una de los usos que nosotros hemos dado a los módulos en nuestra programación científica es el de interfases. En 𝔽90 , cuando el usuario digita

```
sin(x)
```

𝔽90 calcula el seno de `x`, sea `x` una variable real, compleja, un arreglo de reales, un arreglo de `real(8)`, etc., y este proceso es invisible para el usuario. Por medio de módulos es posible crear funciones y subrutinas **propias** que tengan la habilidad de adaptarse de acuerdo a los argumentos del programa principal. La forma como esto se puede lograr está más allá de este texto, pero es importante para el lector saber que esa posibilidad existe.

Preguntas

1. Escriba un programa (ℂ , 𝔽90 o Matlab) que despliegue su frase celebre favorita.
2. Escriba un programa donde se multiplique los valores 2,46 y 3,85 y dse despliegue el resultado con dos decimales.

3. Escriba un programa que despliegue una tabla con `x`, `sinh(x)` y `cosh(x)` para valores de `x` entre 0.0 y 6.0 en incrementos de 0.5. Use un formato de despliegue apropiado. NOTA: NO convierte `x` a radianes ni aplique la corrección `degrad`.

4. Escriba un programa que repetidamente la pida al usuario por un número positivo. EL programa debe parar si el usuario digita 0. Para el número en cuestión, despliegue TODAS las posibles raices cuadradas que tenga. Haga que el programa sea robusto y tenga la menor cantidad de errores posibles.

5. Modifique el programa 2.7 para calcular el mínimo común múltiplo de dos números enteros.

6. Igual al anterior, pero utilizando una función o subrutina **propia**.

7. Escriba un función (o subrutina) que calcule el volúmen de una esfera, dado el radio (a través del teclado).

8. Los compiladores generalmente incluyen un generador de números aleatorios. En $\mathbb{C}$:

```
(listrand.c)
#include <stdio.h>
#include <stdlib.h> /* contains srand and rand functions */
int main()
{
   int i;
   float xran;
   srand(5); /* initialize generator */
   printf("RAND_MAX = %d \n",RAND_MAX);
   for (i=1; i<=20; i++) {
      xran = (float)rand()/(float)(RAND_MAX+1);
      printf("%f \n",xran);
   }
   return 0;
}
```

   donde `rand()` retorna un valor entero entre 1 and RAND_MAX. Note como se convierte de entero a float po medio de un `float`. No se recomienda el uso de este generador de variable aleatoria para fines investigativos, es importante saber las especificaciones del generador, en algunos casos no es de la mejor calidad.

   En $\mathbb{F}$90

```
...
   integer :: i
```

```
   real :: xran
   do i = 1, 20
      call random_number(xran)
      print *, xran
   enddo
...
```

genera la variable aleatoria con distribución uniforme entre 0 y 1.

9. Escriba un programa que genere 10000 valores aleatorios entre 0 y 1 y evalue la *aleatoreidad* haciendo un conteo de cuantos valores caen en diferentes rangos entre 0 y 1 (ejemplo entre 0,0−0,1, 0,1−0,2, etc.). Despliegue los resultados en la pantalla, así

```
0 1009
1 1048
2 1001
3 1038
4 1008
5 959
6 993
7 925
8 1017
9 1002
```

AYUDA: Cree un arreglo o vector de N elementos que en un principio tenga todos los valores 0 y vaya sumando a medida que algún rango es cubierto por el número aleatorio.

10. Escriba un programa que lea de un archivo de texto con 5 números por linea. Calcule el valor promedio de los 5 datos y restele el promedio a cada uno de ellos. Guarde un nuevo archivo con los datos *demeaned* (con promedio 0,0). Permite que el usuario digite el nombre del archivo de entrada y de salida. Por ejemplo, si el archivo de entrada es:

```
10 20 30 40 50
1 2 3 4 5
2 4 6 8 10
5 5 5 5 5
```

el programa debe generar un archivo de salida similar a

```
-20.000 -10.000 0.000 10.000 20.000
-2.000 -1.000 0.000 1.000 2.000
-4.000 -2.000 0.000 2.000 4.000
0.000 0.000 0.000 0.000 0.000
```

11. Escriba un programa que repetidamente le pida al usuario tres valores (reales) a, b, c, para la ecuacion cuadratica

$$ax^2 + bx + c = 0,$$

Usando la famosa formula para las raíces de este problema, el programa debe reconocer si hay raíces reales y calcular estos valores. Como resultado, muestre el número de raíces reales y sus valores. El programa debe parar si todos los valores son 0. También debe ser robusto, si la función es imcompatible, si `a=0`, etc.

12. Modifique el program 2.14 de tal forma que la multiplicación matricial sea hecha en una función ($\mathbb{C}$) o subrutina ($\mathbb{F}90$). Genere otra subrutina o función que para matrices cuadradas genere la transpuesta de la matriz.

En $\mathbb{C}$, no intente retornar la matriz `c` en el argumento `return`, se sugiere usar algo similar a

```
int matmult(float a[D][D],float b[D][D],float c[D][D])
{
.
< stuff to do multiply here >
.
return 0;
}
```

Recuerde que en $\mathbb{C}$, cambios en los valores de un **arreglo** dentro de la función se ven reflejados en el programa principal.

13. Escriba un programa para verificar la función exponencial compleja (standard en $\mathbb{F}90$, esribirla en $\mathbb{C}$), y compare con otras formas equivalente

$$e^z = e^{(x+iy)} = e^x e^{(iy)} = (e^x cosy) + i(exsiny)$$

que deben ser adaptadas dentro del programa. Asegurese de obtener los resultados esperados

```
exp( 1.1 + 2.3i) = -2.002 + 2.240i
exp( 0.0 + 1.2i) = 0.362 + 0.932i
exp(-0.5 + 0.0i) = 0.607 + 0.000i
```

Permite que el usuario digite los valores real e imaginarios que desee.