# FROM CODE TO EXECUTION: I

High-level code with preprocessors statements

This is the starting point for a program.

It is organized in files with extensions specifying the language; e.g., `fu.f` or `bar`

Passed to a preprocessor.

Pure high-level code.

Passed to a compiler.

Assembly language code

The files have the extension `.s`

Passed to an assembler.

# FROM CODE TO EXECUTION: II

Machine language code

> These object files contain machine language with unresolved cross references to o
> programs and library routines.
>
> They have the extension `.o`.
>
> Several such files are passed to a linker.

Executable program

> This file can have any name (default `a.out` on Unix).
>
> Passed to a loader.

A running process

> During its execution, the process is supported by a run-time environment, which,
> among other things, links the process dynamically to shared libraries and aids in
> storage allocation and deallocation.

Commo

# COMMONALITIES

We will be concerned with C, Fortran 77, and Fortran 95

    And to a lesser extent Matlab.

In spite of very real difference, the languages have much in common.

# FORMAT

Fortran uses lines to delimit statements. C uses

```
if (a .ne. 0) then        if (a != b){
   c = b/a                   c = b/a;
end if                    }
```

A keyword is a word that is part of the language itself.

An identifier is a name that the program creates to represent variables, functions, etc

In C the keywords are reserved.

There are no reserved words in Fortran.

Data Types and Specif

# DATA TYPES AND SPECIFICATIONS

Our languages have the following data types: integer, single and double-precision floating-point, logical (Fortran only), and character.

C has a richer variety of integers.

Fortran supports floating-point complex numbers, which C does not.

There are constants of all types.

All types can be declared as arrays.

Arrays in Fortran and C are different animals.

C has structures, Fortran 95 has derived types.

All but Fortran 77 have pointers that contain the address of a variable or function.

Identifiers are typed by declarations or specification statements.

```
double x, y, z;
```

Fortran has implicit typing. (e.g., `num` is by default an integer.)

Exp

# EXPRESSIONS

Arithmetic operators

+, -, *, -.

Exponentiation ** (Fortran). Modulus % (C).

Usual precedence: a + b*c is not (a + b)*c.

Left to right evaluation: a + b + c is (a + b) + c.

Promotion of types

if a is double and n is int then a + n is double.

Relation operators

a .le. b in Fortran and a <= b in C.

Logical operators

.and., .or. in Fortran and && and || in C.

C is richer than Fortran (eg., x++).

The assignment operator is = in all languages.

# CONTROL STATEMENTS

`if`, `then`, `else`, `else if`

`switch` (C); `select case` (Fortran 95).

Looping: `for` (C); `do` (Fortran); `while`.

`goto` (Fortran).

> Disparaged by Dijkstra.

> Useful in the following situations.

>> For simulating control statements that are not in the language at hand.

>> For breaking out of deeply nested constructs — especially loops.

>> For directing control from several places in a program to a common point — example, to a clean-up before quitting a section of code or a subprogram.

# SUBPROGRAMS

C has only functions. Parameters are passed by value.

Fortran has functions and subroutines.

Functions return a value.

Subroutines return values through the argument list.

In both cases, parameters are passed by reference.

Call b

# CALL BY VALUE

Consider the following Matlab function.

```
function [d] = dot(n, x, y)
   d = 0.0;
   for i=1:n
      d = d + x(i)*y(i);
   end
   n = 0;
return
```

If we execute the following code

```
size = 5;
vec1 = ones(size, 1); vec2 = 2*vec1;
a = dot(size, vec1, vec2)
size
```

we get

```
a =
    10
size =
    5
```

# CALL BY REFERENCE

If Matlab were to use call by reference in dot:

```
function [d] = dot(n, x, y)
   d = 0.0;
   for i=1:n
      d = d + x(i)*y(i);
   end
   n = 0;
return
```

then on executing

```
size = 5;
vec1 = ones(size, 1); vec2 = 2*vec1;
a = dot(size, vec1, vec2)
size
```

we get

```
a =
    10
size =
     0
```

Internal and Static V

# INTERNAL AND STATIC VARIABLES

A subprogram can declare internal variables for its own use.

Ordinarily, the values of these variables is lost when the subprogram returns.

However, all our languages have can declare variable to be static, so that they keep t
values until when the subprogram returns.

Static variables can cause a function to return different results when they are execute
with the same arguments.

Such inconsistencies are called side effects.

Side effects can make it difficult for compilers to optimize code.

# GLOBAL VARIABLES

Global Variables are variables that can be accessed by a set of subprograms.

They provide a common workspace for the subroutines that can access them.

All our languages have mechanisms for implementing global variables.

They are a rich source of side effects.

# ARRAYS

All our languages have arrays of their respective data types.

For example the Fortran statement

```
double precision x(20), a(5,12)
```

defines x to be a linear array with 20 elements and a to be a two-dimensional array consisting of 5 rows and 12 columns.

The arrays of Fortran and C are quite different.

# STRUCTURES

Fortran 95 and C can define structures that bundle together variables and other struct

For example the F95 statement

```
type point
   real x     ! The x-coordinate
   real y     ! The y-coordinate
end type point
```

defines a derived type that can represent points in a plane.

Then if we define

```
type(point) A, B, C
```

We can refer to the x-coordinate of A by A%x.

The corresponding construction in C is called a structure.

# POINTERS

A pointer is a variable that contains an address of a memory location.

Languages that have pointers have a mechanism for retrieving and altering the memo
locations they point to.

For example, if p is a C pointer then *p represents the object pointed to by p.

Pointers are useful for returning new objects not defined at compile time.

The statement

```
dp = (double *)(100, sizeof(double))
```

returns a pointer to an array of 100 doubles.

We can reference the array by dp[i].

Pointers can be used deliberately or unintentionally to write other parts of memory.

```
dp[500] = 1;
```

Pointer can make it difficult for compilers to optimize.

Pointers in F95 are restricted to prevent such problems.

Re

# RECURSION

The recursive Matlab program

```
function fct = fact(n)
   if n==0
      fct = 1;
   else
      fct = n*fact(n-1);
   end
return
```

computes the factorial of n.

Recursion is not as expensive as it once was.

There is no recursion in F77.

# MEMORY MANAGEMENT

Declarations of variables causes memory for the variables to be made available.

In many applications more memory is needed during execution.

For example, a work array whose size depends on the input to the program.

Memory comes from the stack or the heap.

The programs that supply the memory are part the run-time environment.

In F77 all memory must be declared at compile time.

# PREPROCESSORS

A preprocessor is a program that takes high-level language code and changes it accor
to preprocessor directives before it is compiled.

For example, in C the preprocessor line

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

brings in a system file containing specification statements that are used in I/O.

Another example.

```
#define PI 3.14159265
```

cause the occurrence of the name `PI` in the text of the program to be replaced by
`3.14159265`.

More elaborate definitions, called macros, can take arguments.

```
#define pythag(a, b) sqrt(a*a + b*b)
```

# SEPARATE COMPILATION AND LIBRARIES

Our languages have provisions for breaking large programs into smaller modules, whic
may be compiled separately.

The reasons:

Managing short files is easier than managing large files.

The parts of the program that are solid do not have to be recompiled.

On Unix systems the compilations can be coordinated by `make`.

Our languages have libraries to evaluate mathematical functions like sin or exp, to
manipulate strings, and many other things.

# FORTRAN 77

Fortran (for Formula Translator) was the earliest successful programming language.

It went through numerous revisions

Fortran II, Fortran IV, Fortran 66, Fortran 77

Fortran 90, Fortran 95, Fortran 2003

FII–F77 all had a fixed input format with certain columns restricted to certain functio

Initially Fortran had very primitive control constructs

F77 has everything but the `select case` contstruc.

The later Fortrans dropped the fixed format and added pointers, memory managemer considerable polymorphism, and modules.

F2003 has OOP.

```fortran
1.          double precision function ddot(n,dx,incx,dy,incy)
2.  c
3.  c       forms the dot product of two vectors.
4.  c
5.          double precision dx(*),dy(*),dtemp
6.          integer i,incx,incy,ix,iy,m,mp1,n
7.  c
8.          ddot = 0.0d0
9.          dtemp = 0.0d0
10.         if(n.le.0)return
11.         if(incx.eq.1. and. incy.eq.1) go to 20
12. c
13. c       code for unequal increments or equal increments
14. c       not equal to 1
15. c
16.         ix = 1
17.         iy = 1
18.         if(incx.lt.0)
19.        *   ix = (-n+1)*incx + 1
20.         if(incy.lt.0)
21.        *   iy = (-n+1)*incy + 1
22.         do 10 i = 1,n
23.           dtemp = dtemp + dx(ix)*dy(iy)
24.           ix = ix + incx
25.           iy = iy + incy
26.      10 continue
27.         ddot = dtemp
28.         return
29. c
30. c       code for both increments equal to 1
31. c
32.      20 do 30 i=1,n
33.             dtemp = dtemp + dx(i)*dy(i)
34.      30 continue
35.         ddot = dtemp
36.         return
37.         end
```

# PROGRAMS

Each program must must have a single program unit of the form

```
program <name>
    <specification statements>
    <executable statements>
end
```

This is the equivalent of the `main` function in C.

# FUNCTIONS

A function returns a value; e.g.,

```
double precision function ddot(n,dx,incx,dy,incy)
```

Functions get their arguments by reference, and can then change them. But it is not good idea. Consider:

```
integer function fu(m)      integer function bar(m)
integer m                   integer m
   fu = 5                      bar = m
   m = 10                   end
end
```

When these functions are used in the sequence

```
m = 1
if (fu(m) .lt. bar(m)) ...
```

If `fu` is evaluated first, the expression in the if statement will evaluate to `.true.`

If `bar` is evaluated first, the expression will evaluate to `.false.`.

The standard does not help this problem by specifying an order of evaluation.

3

# FUNCTIONS IN C

A C function has the form

```
<type> <name>(<parameter list>){
   <declarations>
   <statements>
}
```

For example,

```
double ddot(int n, double x[], int incx, double y[], int incy)
```

Parameters are passed by value.

We can get around this limitation by using pointers.

One can return from a program by

1. `return <expression>;`

2. `exit(int);`

# POINTERS

If a is a variable, then &a is its address. It is called a pointer.

If b is a pointer, then *b is equivalent to a variable whose value is found at b.

& is the address operator.

* is called the indirection or dereferencing operator.

The sequence

```
int a, *b, c;
b = &a;
a = 6;
c = *b/2;
*b = c;
```

results in the value of   a   being set to 2.

# POINTERS AND FUNCTIONS: I

We can use pointers to return values from functions via the parameter list. For exam

```
void swap(int *a, int *b){
    int temp;
    temp = *a;
    *a = *b
    *b = temp;
    return
}
```

If the int's p==5 and q==3, then the invocation

```
swap(&a, &b)
```

makes p==3 and q==5.

# POINTERS AND FUNCTIONS: II

Pointers to functions can be used to pass functions to to functions.

```
squarefx(double x, double (*f)(double)){
    double temp;
    temp = (*f)(x);
    return temp*temp;
}
```
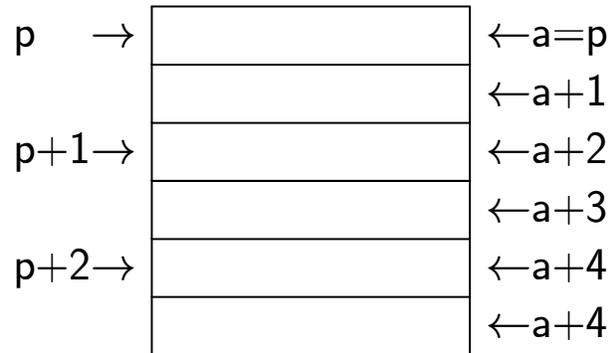
Note the difference!

double (*f)(double)  says that f is a pointer to a function of a double that returns double.

double *f(double)  says that f is a function of a double that returns a pointer to double.

# POINTER ARITHMETIC

One can add integers to pointers to get a new pointer. Suppose p is a double. Then

```
p      →  ┌─────────────┐  ←a=p
          ├─────────────┤  ←a+1
p+1→      ├─────────────┤  ←a+2
          ├─────────────┤  ←a+3
p+2→      ├─────────────┤  ←a+4
          └─────────────┘  ←a+4
```

Pointer arithmetic recognizes the size of the type of the pointer.

# STRUCTURES: I

Consider the following structure that represents points in the plane.

```
struct point{
    double x;
    double y;
};
```

The following function computes the distance between two `point`'s.

```
double dist(point A, point B){
    double dx, dy;
    dx = B.x - A.x;
    dy = B.y - A.y;
    return sqrt(dx*dx + dy*dy);
}
```

```
double dist(point A, point B){
    double dx, dy;
    dx = B.x - A.x;
    dy = B.y - A.y;
    return sqrt(dx*dx + dy*dy);
}
```
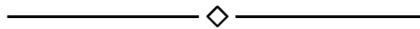
———————◇———————

The problem is that the structures A and B must be copied. If we use pointers, we ge

```
double dist(point *A, point *B){
    double dx, dy;
    dx = (*B).x - (*A).x;
    dy = (*B).y - (*A).y;
    return sqrt(dx*dx + dy*dy);
}
```

```
double dist(point *A, point *B){
    double dx, dy;
    dx = (*B).x - (*A).x;
    dy = (*B).y - (*A).y;
    return sqrt(dx*dx + dy*dy);
}
```

———————◇———————

C provides a more convenient notation

```
double dist(point A, point B){
    double dx, dy;
    dx = B->x - A->x;
    dy = B->y - A->y;
    return sqrt(dx*dx + dy*dy);
}
```

# THE FORTRAN ARRAY

Here are some typical Fortran arrays

```
integer n(1000)
real a(100), b(-5:5), c(25,30)
```

Arrays can be up to 7-dimensional.

The limits of arrays can be variable.

Arrays are stored in column major order

```
double precision c(2,2,2)
```

```
c(1,1,1), c(2,1,1), c(1,2,1), c(2,2,1),
        c(1,1,2), c(2,1,2), c(1,2,2), c(2,2,2)
```

# MORE ON ARRAY STORAGE

```
    a      a + 40  a + 80   a + 120
 A(1,1)  A(1,2)  A(1,3)   A(1,4)

 a + 8  a + 48  a + 88   a + 128
 A(2,1)  A(2,2)  A(2,3)   A(2,4)

 a + 16 a + 56  a + 96   a + 136
 A(3,1)  A(3,2)  A(3,3)   A(3,4)

 a + 24 a + 64  a + 104  a + 144
 A(4,1)  A(4,2)  A(4,3)   A(4,4)

 a + 32 a + 72  a + 112  a + 152
 A(5,1)  A(5,2)  A(5,3)   A(5,4)
```

───────◇───────

If a is the starting address of A, then the address of A(i,j) is

a + 8*(5*(j-1) + i-1).
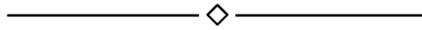
# IN GENERAL

If `A` is `m` `x` `n` then the address of `A(i,j)` is

`a + size*(m*(j-1) + i-1)`.

The number `m` is called the leading dimension of A (often) abbreviated `lda`

The trailing dimension is not necessary to calculate the address of `A(i,j)`

# ARRAY TRICKS I

```
integer m, n, mmax, nmax
parameter (mmax=100, nmax=50)
data m/75/, n/25/
double precision a(nmax), b(nmax), c(mmax,nmax), d(mmax,nmax)
call fu(m, n, mmax, nnax, a, b(3), c, d(1,5))
```

———————◇———————

```
subroutine fu(m, n, ldc, nmax, aa, bb, cc, dd)
integer m, n, ldc, nmax
double precision aa(nmax), bb, cc(ldc, nmax), dd
```

The arrays are called adjustable.

nmax has been renamed ldc.

bb and dd have been declared as scalars.

```
integer m, n, mmax, nmax
parameter (mmax=100, nmax=50)
data m/75/, n/25/
double precision a(nmax), b(nmax), c(mmax,nmax), d(mmax,nmax)
call fu(m, n, mmax, nnax, a, b(3), c, d(1,5))


subroutine fu(m, n, ldc, nmax, aa, bb, cc, dd)
integer m, n, ldc, nmax
double precision aa(*), bb, cc(ldc,*), dd
```

The arrays aa and cc are assumed size arrays.

The value of nmax does not need to be passed.

```
integer m, n, mmax, nmax
parameter (mmax=100, nmax=50)
data m/75/, n/25/
double precision a(nmax), b(nmax), c(mmax,nmax), d(mmax,nmax)
call fu(m, n, mmax, nnax, a, b(3), c, d(1,5))


subroutine fu(m, n, ldc, nmax, aa, bb, cc, dd)
integer m, n, ldc, nmax
double precision aa(*), bb(*), cc(ldc,*), dd(ldc,*)
```

bb(i) corresponds to b(3+(i-1)).

Equivalently bb(1:n-3+1) corresponds to b(3:n).

dd(1:m,1:n-5+1) corresponds to d(1:m,5:n).

```
integer m, n, mmax, nmax
parameter (mmax=100, nmax=50)
data m/75/, n/25/
double precision a(nmax), b(nmax), c(mmax,nmax), d(mmax,nmax)
call fu(m, n, mmax, nnax, a, b(3), c, d(1,5))
```

```
subroutine fu(m, n, ldc, nmax, aa, bb, cc, dd)
integer m, n, ldc, nmax
double precision aa(*), bb(*), cc(ldc,*), dd(*)
```

The change is in the declaration of dd, which has become a linear array, whose startir address is that of d(1,5).

In other words dd(1:m) is simply the fifth column of dd.

Computir

## COMPUTING A'*A

$$
B = A^{\mathrm{T}}A = \begin{pmatrix} a_1^{\mathrm{T}} \\ a_2^{\mathrm{T}} \\ \vdots \\ a_n^{\mathrm{T}} \end{pmatrix} \begin{pmatrix} a_1 & a_2 & \cdots & a_n \end{pmatrix} = \begin{pmatrix} a_1^{\mathrm{T}}a_1 & a_1^{\mathrm{T}}a_2 & \cdots & a_1^{\mathrm{T}}a_n \\ a_2^{\mathrm{T}}a_1 & a_2^{\mathrm{T}}a_2 & \cdots & a_2^{\mathrm{T}}a_n \\ \vdots & \vdots & & \vdots \\ a_n^{\mathrm{T}}a_1 & a_2^{\mathrm{T}}a_2 & \cdots & a_n^{\mathrm{T}}a_n \end{pmatrix}
$$

—————◇—————

The calling sequence for ddot is

```
double precision function ddot(n, dx, incx, dy, incy)
```

It is called as follows.

```
   do 20 j=1,n
      do 10 i=1,j
         b(i,j) = ddot(m, a(1,i), 1, a(1,j), 1)
         b(j,i) = b(i,j)
10    continue
20 continue
```

The 1-dimensional

# THE 1-DIMENSIONAL C ARRAY

$$\text{double x[10]}$$

———————◇———————

The above statement defines x to be an array of 10 doubles.

Because indexing in C begins at zero, x[9] is the last element.

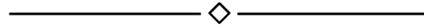x is a pointer constant. Hence x[i] and *(x+i) are the same.

The construction

```
int fu(int n){
    int work[n];
    .  .  .  .  .  .
}
```

is impossible in C.

```
double A[5][7]
```

───────◇───────

The above statement defines $A$ to be an 5 x 7 array of doubles.

Because C arrays are stored row-wise, the expressions `A[i][j]` and `*(x + j + 7*i)` are equivalent

Unfortunately, the construction

```
double fu(int lda), double A[][ldt]
```

is impossible.

But you can always pass pointers and use pointer arithmetic.