INTERNATIONAL ATOMIC ENERGY AGENCY
UNITED NATIONS EDUCATIONAL, SCIENTIFIC AND CULTURAL ORGANIZATION
INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS
I.C.T.P., P.O. BOX 586, 34100 TRIESTE, ITALY, CABLE CENTRATOM TRIESTE

UNITED NATIONS INDUSTRIAL DEVELOPMENT ORGANIZATION
NTERNATIONAL CENTRE FOR SCIENCE AND HIGH TECHNOLOGY
INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS 34100 TRIESTE (ITALY) VIA GRIGNANO, 9 (ADRIATICO PALACE) P.O. BOX 586 TELEPHONE 040-224572 TELEFAX 040-224575 TELEX 460449 APH I

SMR/643 - 6

SECOND COLLEGE ON
MICROPROCESSOR-BASED REAL-TIME CONTROL -
PRINCIPLES AND APPLICATIONS IN PHYSICS
5 - 30 October 1992

C LANGUAGE - BASICS

(6 - 8)

A. NOBILE
International Centre for Theoretical Physics
Trieste
Italy

These are preliminary lecture notes, intended only for distribution to participants.

STRUCTURES AND UNIONS

To group heterogeneous objects (PASCAL"record"):

date: day month year

```
struct date{ int day, month, year; };
struct date today, yesterday,
   tomorrow;
```

- declare **structure** *tag* `date`(its not a `typedef!`)
- declares `today, yesterday` and `tomorrow` to be variables of the type "`struct date`"

Personal record:
name
social security number
date of birth : date

```
struct vitalstat
{   char vs_name[19],vs_ssnum[11];
    struct date vs_birth_date;
} vs1;

struct vitalstat vs2;
```

- declares **structure** *tag* `vitalstat`
- declares variables `vs1 vs2` of type `struct vitalstat`

- **struct** *tag_name* { *list of declarations*}

- **struct** components can be other **structs**
  WARNING : but of different types

```
struct infinite{ int count;
          struct infinite mytail;
} /*ILLEGAL*/
```

- *tag_name* is optional

```
struct {char a[10], b[10] ;} str1,
str2;
```

ACCESSING ELEMENTS OF A STRUCTURE

```
strcpy( vs.vs_name, "John Smith");
strcpy( vs.vs_ssnum, "035400245");
vs.vs_birth_date.day=17;
vs.vs_birth_date.month=9;
vs.vs_birth_date.year=1956;
```

variable name .component name

```
if (vs.vs_birth_date.month > 12 ||
    vs.vs_birth_date.day > 31 )
   printf ( "Illegal date. \n");
```

Structure components are normal variables

ARRAYS OF STRUCTURES

Arrays of anything!

```
#include <stdio.h>
typedef struct {float re,im;} Complex;
/* placed here to be GLOBAL, that is
apply to all functions in this file */
/* reads in two complex arrays */
main()
{

   Complex v1[10], v2[10];

   for ( i=0; i<10 ; i++ )
      scanf(" %f %f %f %f " ,
         &v1[i].re , &v1[i].im,
         &v2[i].re , &v2[i].im) ;
}
```

OR

```
#include <stdio.h>
struct complex { float re,im;}  ;
main()
{

   struct complex v1[10] , v2[10] ;

   for ( i=0; i<10 ; i++ )
        scanf(" %f %f %f %f " ,
         &v1[i].re , &v1[i].im,
         &v2[i].re , &v2[i].im) ;
}
```

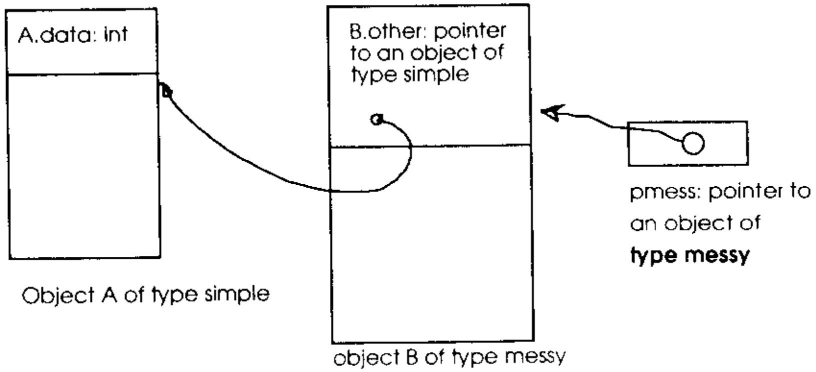
POINTERS TO STRUCTURES

pointers to anything !

```
#include <stdio.h>
typedef struct {float re,im;} Complex;

/* reads in one complex array
 * and computes its euclidean norm
 * squared */
main()
{

    Complex v1[1000];
    double cnorm2( );

    for ( i=0; i<10 ; i++ )
         scanf(" %f %f " , &v1[i].re ,
           &v1[i].im) ;
    dp=cnorm2(v1,10);
    printf (" %f \n" , dp );
}
double
cnorm2( v1,  n)
Complex v1[];
int n;
{
    double d=0;
    Complex *vend=&v1[n], *vp= v1;

    for( ; vp < vend; vp++ )
         d += (*vp).re * (*vp).re +
              (*vp).im * (*vp).im ;

    return d;
}
```

(*vp).re UGLY. CAN BE TERRIBLE :

```
struct simple { int data;
       ....
} A;
struct messy{ struct simple * other;
       ....
       } B;
struct messy * pmess;

(*(*pmess).other).data
```


A.data: int
B.other: pointer to an object of type simple
pmess: pointer to an object of **type messy**
Object A of type simple
object B of type messy

NEW OPERATOR ->

p->x IS **(*p).x**

EXAMPLE ABOVE: **pmess->other->data**

```
double cnorm2( v1 , n)
Complex v1[];
int n;
{
    double d=0;
    Complex *vend= &v1[n],  *vp=v1;
    for( ; vp < vend; vp++)
        d += vp->re * vp->re +
            vp->im * vp->im ;
    return d;
}
```

OPERATIONS ON STRUCTURES

- take a component (**.**)
- take the address (**&**)
- take the size (**sizeof**)

TO PASS TO FUNCTIONS: pass pointers

```
struct complex * cprod (cp1, cp2)
struct complex * cp1, *cp2;
{   struct complex product;
    product.re=cp1->re*cp2->re -
                cp1->im*cp2->im;
    product.im=cp1->re*cp2->im+
                cp1->im*cp2->re;
    return &product; /* DISASTER:
* returns pointer to local
* variable! */
}
```

Solution: use global or local static ?
Difficult! **cprod(cprod(*a,*b),*c) would not work either!**

DO NOT TRY TO RETURN AS VALUE (at least if use in of returned struct in an expression is conceivable)

```
cprod ( resultp, cp1, cp2)
struct complex *resultp, * cp1, *cp2;
{
    resultp->re=cp1->re*cp2->re -
                cp1->im*cp2->im;
    resultp->im=cp1->re*cp2->im+
                cp1->im*cp2->re;
    return ;
}
```

ANSI ONLY:

```
typedef struct(float re,im;) Complex;
Complex z1,z2;
```

- assignement:

```
z1=z2;
```

- passing as argument to functions

```
double cnorm ( Complex z1){
       return ( sqrt(z1.re * z1.re+
               z1.im * z1.im))
}
```

- being returned by a function

```
Complex csum (Complex z1,Complex z2){
    Complex s;
    s.re = z1.re + z2.re;
    s.im = z1.im + z2.im;
    return s;
}
```

- together with assignement:

```
Complex s,a,b;
Complex csum(Complex, Complex);
......
s=csum(a,b);
```

LAYOUT OF STRUCTURES IN MEMORY

Seldom useful; sometimes, with pointers...;

- Components are in sequential order, but not necessarily contiguous (holes -*padding*- possible to align objects to hardware required positions)
- No padding before first component: address of structure is address of first component

SELF-REFERENTIAL STRUCTURES

UNIONS

Like structures, but components share the same memory: only one can be *active* at any time.

Like Fortran EQUIVALENCE, Pascal variant record

```
union reint{
     float re;
     int i;
}

reint.re = 2.0; /* reint.int becomes undefined */
......
reint.i = 1; /* reint.re becomes undefined */
```

NORMALLY used inside a **struct,** togeteher with another variable holding an indicator,

```
struct {
     int type;
     union {  float r;
              int i;
     } v;
} var;

var.type = 0;
var.v.r = 1.0;
.....
var.type = 1;
var.v.i = 7;
.....
if (var.type == 0) x=var.v.r;
```

Bit fields

Not available on your compiler

```
struct {
     a: 3;
     b: 7;
     c: 2:
} s;
```

s.a is 3 bits wide;
s.b is 7 bits wide, and contiguous to s.a
s.c is 2 bits wide, contiguous to s.a

-- Each compiler can arrange bit fields in increasing or decreasing order in a computer word;

-- If a bit field would cross the boundary between two computer words, it is shifted to a new word

-- No bit field can be longer than a computer word

USAGE : sometimes to save memory
often to manipulate bit-sized objects
(hardware)

SCOPE RULES

```
#include <stdio.h>
typedef struct {float re,im;} Complex;
Complex arr[100];

main(){
   Complex x,y; /*OK:Complex global*/
   float normx = 0.0, normy = 0.0;
   int i;
   for (i = 0; i<100; i++){
      scanf(" %f %f", &arr[i].re,
&arr[i].im);
      if (norm() > normx)
         /* wrong: norm() unknown
          * assumed int
          */
        x = arr[i];
      if (norm() < normy)
        y=arr[i];
   }
}
float
norm(){
   return( arr[i].re*arr[i].re +
arr[i].im*arr[i].im);
   /* WRONG : i unknown */
}
```

i defined when norm called, but its *name* unknown outside function main

SCOPE of identifiers: where a NAME can be used

DIFFERENT BUT RELATED PROBLEM

```
main()
{ ......
   int *p, *f1();
   p=f1();
   ...
   f2(p);
}
int * f1(){
   int i=1;
   f2(&i);
   return &i;
}
f2(ip)
int *ip;
{
   printf ("%d",i);
}
```

- **i** *created* when **f1** called
  deleted when **f1** exits

- when **f2** called from **main,**
  i NO LONGER EXISTS

STORAGE CLASSES: when are variables created, deleted, initialized, etc.

SCOPE rules must be consistent with storage classes: non-existing variables cannot be named
- Pointers allow exceptions (AARGHH)

STORAGE CLASSES

1) **auto :**
normal declarations INSIDE compound statements.
Created and initialized before execution
of the compound statement, deleted at its end;

SCOPE: from tne declaration point to the end of the compound statement;

```
#include <stdio.h>
main(){
    int q[100];
    long int s;
    long int sum( );

    {
        int i=0 ; /* i created and
 initialized */
        for ( ; i<100 ; i++)
            scanf( "%d", &q[i]) ;
    }
    /* i no longer exists and is no
       longer accessible */
       s = sum( q, 100 );
 }
 long int
 sum ( arr, n )
 int arr[], n;
 {
     long int s = 0;
     /* s created and initialized */
     int i; /* i  created */

     for ( i=0 ; i<n ; i++)
         s += arr[i];
     return s ;/* i, s deleted */
 }
```

- NOTE : the body of functions is a compound statement!

- NOTE: the closest definition is the one that is considered (hides external ones)

Ex.: in the above the reading loop could be:

```
  {
       int s=0 ;
       /* s created and initialized
       *   "main"-wide s hidden
       */
       for ( ; s<100 ; s++)
           scanf( "%d", &q[s]) ;
  }
```

2) extern :

definitions outside any function, not marked "static". Created when program starts, survive till program end. Accessible from other files, through suitable *allusions (declarations).*

SCOPE:

- for a definition, the file in which the definition occurs, from the definition to the end;
- for an allusion :
- ---if the allusion is in a compound statement, the compound statement;

- ---if outside any function, the file from the allusion down to the end;

WARNING:

storage class <=> variable
scope <=> name

The name of an **extern** variable can have local scope if allusion (declaration) is inside compound statement

Do not identify EXTERN (storage class) and GLOBAL(scope)

File a.c:

```
#include <stdio.h>
struct complex {float re,im; } ;
    /*defines the tag complex :
    global to the file a.c*/
struct complex carr[10];
    /* defines an extern array of
        10 complex */
extern struct complex big_x;
    /* declares big_x as complex ,
        defined in another file;
        allusion */
main(){
    ....
    extern int fun();
    extern int errcode;
    /* allusions */
    int test();/* allusion? */
    struct complex z;
    /* struct complex has
        file scope */
    test();
    if(carr[1].re==0.0)errcode=1;
        /* carr has file scope */
}
int
test()/*definition of test */{
    if (carr[0].re > 100.0) {
        errcode=2;
        /* wrong : errcode has
            block scope */
            big_x.re=carr[0].re;
            big_x.im=carr[0].im;
        /* big_x, carr have file scope */
    }
```

File b.c

```
    struct complex {float re,im; };
    extern struct complex carr[10] ;
    /*allusion */
    int errcode=0; /definition of
    errcode */
    int fun(int i){
        .....
    }/*definition of fun */
```

COMMENTS:

- all the function names are by default **extern**
- types and tags have no storage associate to them->no allusions-> can be local to a block or global to file;
  #include to share among files (ALWAYS!).
- allusions are identified by the keyword **extern;**

3) static

Two uses:

3.1) Variables defined inside a block, but created and initialized at program start and deleted at program end;
keep their value from call to call (unlike **auto**)

SCOPE: the compound statement in which they are defined.

```
int ff(n)
int n;{
    static int first=1;
    ...
    if (first ){
        /*something to be done on
           first call */
        first=0;
    }
    ...
}
```

auto would not work (WHY?)

3.2) Variables AND FUNCTIONS defined like **extern** ones, but whose SCOPE is file only (cannot be *alluded*)

VERY USEFUL, HIGHLY RECOMMENDED

PROTECTS AGAINST *name clashes*

INFORMATION HIDING

Problem : set of routines to manipulate a list of names. The user should simply be able to add a name to the list (addnam), delete a name from the list (delnam), search the list for a name. The name is a string.

File lname.c:

```
#include <stdio.h>
/* basic data store not directly
 accessible from outside */
static struct vsstat *listOfNames ;
/* public procedures */
addnam(name)
char *name;
{
.......
}
```

```
int delnam(name)
char *name;
{
.....
}
struct vsstat * search(name)
char *name;
{
.....
}
/* private procedures
 * NAME CLASHES impossible !
 */
static compact_list(){
.....
}
static struct vsstat *
create_entry(name)
char *name;
{
.....
}
static error (errcode)
int errcode;
{
....
}
```

4) register

Like **auto**, but suggests to the compiler to put the variable in a hardware register if possible.
Can improve optimization a lot on old compilers.
Can inhibit it with optimizing compilers

- Since registers are limited, the first variable declared **register** has higher priority for allocation, and so on;
- You cannot take the address of a register variable

```
int arr[100] , k;

{   register int *pi , s=0;
    for(pi=&arr;pi<&arr[100];pi++)
        s += *pi;
    k=s;
}
```

TYPE QUALIFIERS (ANSI ONLY)

const

```
const float m=4.0;
const int *pci;
    /* pointer to const int */
m = 5.0;  /*error */
pci = &a; /*legal*/
*pci = a; /error*/
```

- Can be used on function arguments

```
float sum(const float arr[],
          const int n);
```

volatile

A **volatile** variable can be modified by the hardware or the O.S. , outside control fom the program.
THEREFORE, any store or load operation requested by the program MUST be actually performed (no optimization allowed)

Memory-mapped I/O: output by writing to address 500

```
    char a[100] ;
    int i ;
    char *out = (char *) 500 ;

    for(i=0; i<100; i++) *out = a[i] ;
```

most optimizers would translate into

```
    *out = a[99] ;
```

BUT

```
    char a[100];
    int i;
    volatile char *out =
        (volatile char *) 500;

    for( i=0; i<100; i++) *out = a[i];
```

COMMENT: can be combined

```
    extern volatile const int clock;
```

FUNCTIONS

Glossary

declaration : the point where a name gets a type associated with it

definition : a declaration that moreover associates some memory with the name. For functions, it is the place where you give a ***body*** for the function.

formal parameters
formal arguments : the names with which a function refers to its arguments

actual parameters
actual arguments : the names or values used when the function is actually called -> the values that *formal parameters* have on entry to the functions.

FUNCTION DECLARATION

Functions must be declared before being called

ANSI style: function prototype

```
char * isprint( char c );
static struct vsstat * createnode( char
* name );
```

Synopsis:

- Optional `static`; if not present, `extern` storage class is assumed
- function type (if missing, `int` assumed)
  - cannot be **array**
  - cannot be **function**
  - CAN be pointer to array or pointer to function
- function name

- list of declarations of formal arguments, in parentheses:
  like other declarations except:
  - only legal storage class is **register**;(ANSI)
  - an array declaration is interpreted as a pointer to an object of the same type of the array elements;
  - a function declaration is interpreted as a pointer to a function;
  - no initializers

IMPORTANT USE:

```
double sqrt( double x );
...
z=sqrt(1);
```

The compiler recognizes type mismatch and performs convertion of 1 to double

```
struct vsstat *add_to_list(char * name);
.....
p = add_to_list(1.0);
```

The compiler recognizes type mismatch and signals error

Old C style

```
char * isprint( );
static struct vssstat *
createnode( );
```

No information on arguments

```
p = createnode (1.0) ;
 /* AAARRGHHH */
```

FUNCTION DEFINITION

ANSI style

function prototype as above
function body (compound statement)

```
int factorial(int n)
{
   register long int p=1;
   register int i ;

   for (i = 2; i<=n; i++) p *= i;
   return p;
}
```

Old C style (accepted also by ANSI)

static (optional)
type name **(** *list of formal arguments names***)**
formal arguments declarations
function body

```
int factorial (n)
int n;
{
    ....
}
```

---

argument declarations: as in prototypes, plus:
--- **char** and **short** are treated as **int** + conversion
--- **float** are treated as **double** + conversion
<u>DEFAULT CONVERSIONS</u>

```
void a_func( c, x )
char c;
float x;
{ .....}
```

is handled as

```
void a_func( ext_c , ext_x )
int ext_c;
double ext_x;
{
   char c;
```

```
    float ext_x;

    c = (char) ext_c;
    x = (float) ext_x;
    ......
}
```

Seldom important to know, except for cross-language development. Can impact performance.

CALLING FUNCTIONS

1. evaluate expressions passed as arguments;
2. convert values according to function prototypes if any (ANSI) or according to default conversions;
3. use these values to initialize formal arguments
4. henceforth formal arguments behave like other local variables

```
void called_func( int , float );

main(){
    called_func ( 1, 2*3.5 );
}

void called_func (int iarg, float
farg){
    float tmp;
    tmp= iarg * farg;
}
```

CALL BY VALUE :

a copy of the value of the actual argument is passed, not the actual argument itself
-> function cannot modify the actual arguments
(unlike FORTRAN, Pascal **var** arguments)

```
int called_func( );

main(){
    int n=10, array[30];
    ......
    called_func ( array,10 );
}
called_func (arr,n)
int arr[],n;
{
    for(;n>=0;n--)
        printf("%d\n",arr[n]);
/*changing n is perfectly safe */
}
```

CALL BY REFERENCE:

passing the *address* of the actual argument.
Function MUST be written specially to accept it

```
float called_func(  );

main(){
    int i = 1, f;
    f=called_func ( &i , 2*3.5 );
}
float called_func (iarg,farg)
int *iarg; /* note int* */
float farg;
{
    float tmp;
    tmp= *iarg * farg;
    (*iarg )++ ; /* changes i */
    return tmp;
}
```

Arrays **are not** be passed by value:

```
void func(arr)
int arr[];
{......}
main()
{
int arr[10];
func(arr);
}
```

is identical to

```
void func(arr)
int *arr;
{ ...... }
main()
{int arr[10];
func(&arr[0]);
}
```

Functions are not be passed by value (WHAT?)

EXCURSUS : pointers to functions

Often used !

function name is constant pointer to function like array name

```
double fun(x)
double x;
{.....}
double operator(f)
double (*f)();
{/*do something with function f*/
...
}
main()
{
    double (*pf)(),s;
    /* pf pointer to function
    returning double */
    pf = fun ; /* pf = &fun wrong ;
             * pf = fun() wrong ;
             * pf = &fun() wrong ;
             * /
    s=operator(fun);
    /* same as s=operator(pf) */
    .....
}
```

Structures are passed by value (ANSI)

More on Default Conversions

If no function prototype used (Old C form of declaration or no declaration at all)

- `short` and `char` converted to `int`;
- `float` converted to `double`;

ANSI WARNING : mixing a prototyped declaration with a non-prototyped definition can cause problems

RETURNING FROM FUNCTIONS

```
void a_func(i,s)
int i;
float *s;
{
    if( !i ) return ;
    *s ++ ;
}
```

- `return`
- flow through the end

RETURNING A VALUE

```
double squareroot(x)
double x;
{
    double s;

    if ( x < 0.0 ) return 0;
    s =.../* compute square root */
    return s;
}
```

- type of returned expression automatically converted to type of function;

WARNING :
- mixing **return** *value* ; and **return;**
- mixing **return** *value*; and flow through end

is meaningless

EXCURSUS: COMPLEX DEFINITIONS

What's that

```
    int *(*(*x)())[5];
```

`*(*(*x)())[5]` is an `int`

[] has higher precedence than *

`(*(*x)())[5]` is a pointer to an `int`

`*(*x)()` is a 5-elements array of pointers to `int`

() has higher precedence than *

`(*x)()` is a pointer to a 5-elements array of pointers to `int`

`*x` is a function returning a pointer to a 5 - elements array of pointers to `int`

`x` is a pointer to a function returning .....

HORRIBLE
USE TYPEDEF

```
   typedef int *PI;
      /* a PI is pointer to int */
   typedef PI AP[5];
      /* an AP is a 5-elements array
      of PI, i.e. of pointers to int */
   typedef AP *FP() ;
      /* an FP is a function returning
      a pointer to an AP */
   FP  *x; /* x is a pointer to an FP */
```

INPUT-OUTPUT

Implemented through macros and functions, but **defined in the standard** as part of the standard library and standard header file **<stdio.h>**

GENERAL MODEL :

- **stream** : flux of *characters*
- each stream connected to an external *file* (operating system dependent)
- read or write take place at *file position indicator*
- *f.p.i.* moved after each read or write (sequential I-O)
- *f.p.i* can be manipulated directly to achieve direct access I-O
- Two basic types of streams : *text* and *binary* (ANSI)
  - *text* : sequence of lines, composed of printable characters. Programs see line separators as a single *newline* character (O.S. can use other conventions)
  - *binary:* sequence of non-interpreted characters.

THEY ARE THE SAME IN UNIX, OS/9, etc.

- streams can be *buffered*; buffering can be
  - block : data passed to/from O.S. when buffer full (file copying);
  - line : data passed to/from O.S. when end of line met (terminal I-O); ANSI
  - no buffer : data passed to/from O.S. immediately (screen editing).
- I-O operations are *syncronous* : program waits until completed

A key distinction:

O.S. services (calls):

read write lseek open close

Language constructs (stream-oriented)

`fread, fwrite, fseek, fopen, fclose`

- Old C programs often used system calls to do "binary" I-O (buffered unformatted)
- Better to avoid: portability
- With old compilers, could be unavoidable (fread, fwrite missing)

Therefore: in Unix and O.S. 9, O.S. uses "file numbers" (small integers) to identify files (`open(filename)` returns a filenumber, `read`, `write` require passing a filenumber, etc.) One field of the structure FILE identifying the C stream is the corresponding O.S. file number.

```
fileno (fp)
FILE *fp;
```

returns the file number attached to the stream fp; etc.

stdio.h

contains the definitions of the required types and macros, plus the prototypes of the functions, and the definitions of 3 standard streams.
Of general interest:

FILE **typedef:** the type of a `struct` containing stream control information.

EOF macro. A negative integral constant, used to signal end of file condition

stdin

stdout

stderr 3 objects of type **(FILE *)**,associated to the standard input (usually keyboard), standard output (usually screen) and standard error (usually screen). Open at program start.

NULL (char *) 0. ANSI moved it to **stddef.h**

ERROR HANDLING

- all I-O functions return error codes ;
- moreover error conditions and end-of-file on read are also recorded in a member of any FILE object;
- tested through **feof()** and **ferror()**, reset through **clearerr()**

- additional error information through system-defined extern `errno`, O.S. dependent

Ex.

```
/* this function tests error status
 * and resets it
 * it returns 0 if no error
 * 1 if end-of file
 * 2 if error
 * 3 if both
 */
#include <stdio.h>
#define EOF_FLAG 1
#define ERR_FLAG 2

char stream_stat( fp )
FILE *fp;
{
    char stat =0;

    if(ferror(fp))stat|= ERR_FLAG ;
    if(feof(fp))  stat|= EOF_FLAG ;
    clearerr(fp) ;
    return stat ;
}
```

DIRECT FILE MANIPULATION

ANSI

```
int remove ( const char *filename );
```

deletes the file. Returns 0 if success.

```
int rename ( const char *old, const
char *new);
```

changes file name. Valid file names are implementation dependent.

```
char * tmpnam(char *ε);
```

create a file name that is unique. On your compiler, analogous to **mktemp.**

```
FILE *tmpfile(void);
```

opens a temporary file which will be automatically deleted at program termination and has no name.

OPENING AND CLOSING

associate a *stream* with a *file*

fopen (***file_name*** , ***access_mode***)

returns a pointer to a **FILE** object or **NULL** (if failed)

```
FILE * fopen(file_name,access_mode)
char * file_name;
char * access_mode;
```

ACCESS MODES

for text streams

"r"	read only
"r+"	read-write (must exist)
"w"	write only. If existing, truncated to zero,else created
"w+"	write and read. If existing, truncated to 0,else created
"a"	append. Write only, but at the end of an existing file. Created if not existing.
"a+"	append and read . Created if not existing

binary streams (ANSI)

"rb", "r+b" etc.

Ex.

```
/* open with error message */
#include <stdio.h>
FILE *
openfile(fname, access)
char *fname, *access;
{
FILE *fp;
if((fp=fopen(fname,access))==NULL)
       fprintf( stderr,
"Error opening %s with access mode %s"
               ,fname,access);
return fp;
}
```

- WARNING : `(fp = fopen()) == NULL`
  parenthesis required! common mistake
- **fprintf** : like **printf** on a stream different from
  stdout

Ex:
Open file "pippo" for reading and writing; if it does'nt exist, create, if it exists, do not truncate

```
if((fp=fopen("pippo","r+"))==NULL)
       fp = fopen( "pippo", "w+");
```

reopen:
associates an open stream with a different file and/or with a different mode

```
FILE *
freopen( filename, mode, stream)
char *filename, *mode;
FILE * stream;
```

often used with standard streams

```
/*if flag set, output to disk file
"outfil"*/
....
int disk_flag;
.....
if ( disk_flag &&
freopen("outfil","w",stdout)==NULL)
       fprintf(stderr,
          "Error reopening");
....
```

IMPORTANT WARNING

Streams open for both read and write:

between a read and a write you MUST insert
a **fflush**, **fseek** or **rewind**

- - exception: write after read that hits End of File

---

fclose:

disassociates a stream from its file and makes the stream unusable

```
    int fclose(stream)
    FILE *stream;
```

NOTE : files are automatically closed at program termination

READING AND WRITING

formatted

unformatted : 1 character at a time
1 line at a time
1 block at a time

FORMATTED READ

```
    int scanf( format,...)
    char *format;

    int fscanf( stream, format, ...)
    FILE * stream;
    char *format;

    int sscanf ( in_string, format,...)
    char * in_string, * format;
```

Input-Output

fclose:

disassociates a stream from its file and makes the stream unusable

int fclose(stream)
FILE *stream;

NOTE : files are automatically closed at program termination

READING AND WRITING

formatted

unformatted : 1 character at a time
1 line at a time
1 block at a time

FORMATTED READ

int scanf(format,...)
char *format;

int fscanf(stream, format, ...)
FILE * stream;
char *format;

int sscanf (in_string, format,...)
char * in_string, * format;

NOTE : **scanf** IS **fscanf(stdin, ...)**

sscanf does conversion but not input,
using in_string as the source of characters
(FORTRAN INTERNAL FILE)

NOTE : arguments must be POINTERS to variables

ANSI INPUT FORMAT STRING

can contain three types of objects:

white space: skip input until next non-blank
ordinary character : next character in input MUST
match that character (seldom used)
conversion specifier:
LOOK IN THE MANUAL

function returns :

- EOF if EOF encountered before any conversion, OR
- number of successful conversions

UNFORMATTED INPUT-OUTPUT

ONE CHARACTER AT A TIME

Already met

```
int getchar( );
int putchar(c)
char c;
```

- refer to **stdin / stdout**

MORE GENERAL

```
int getc(fp)
FILE *fp;
int putc(c, fp)
char c ;
FILE *fp ;
```

special:

```
int ungetc(c, fp)
int c ;
FILE * fp;
```

- return EOF if error (getc/putc/ungetc) or end-of file (getc);
- otherwise return the character read or written

WARNING : in binary mode, EOF is a legal return value for getc,putc and ungetc: use ferror or feof to test for error!

- They are macros (defined in **stdio.h**)
- therefore expanded by preprocessor
- FAST

Note: `putchar(c)` is `putc( c , stdout )`
`getchar()` is `getc (stdin)`

--- WARNING

```
putc ( 'x' , fp[j++] ) ;
```

Macro expansion : more than one occurence of `fp[j++]` -> RESULTS UNDEFINED

For these cases, FUNCTION VERSION

```
int fgetc( fp)
FILE *fp ;
int fputc( c, fp)
char c ;
FILE *fp ;
```

Ex.:

```
#include <stdio.h>

#define FAIL 0
#define SUCCESS 1

int copyfile (infile,outfile)
char *infile, * outfile;
{
    FILE *fp1, *fp2;
    int c;

    if((fp1=fopen(infile,"rb"))==NULL)
        return FAIL;
    if((fp2=fopen(outfile,"wb"))==NULL)
    {   fclose (fp1);
        return FAIL;
    }
    while(c=getc(fp1), !feof(fp1)){
        putc( c , fp2 );
        if(ferror(fp2){
            fclose(fp1);
            fclose(fp2);
            return FAIL
        }
```

- note cleanup in case of failure
- **feof** needed in binary mode:

  getc returns EOF at End of File

  EOF is <0 -> not a letter, <u>if in text mode</u>

  COULD BE 8-bit pattern (often -1)

Example above could be slow (too many tests).

```
while(1){
    register int c;
    while((c=getc(fp1))!=EOF &&
           putc(c, fp2)!=EOF);
    /* EOF detected : why? */
    if(feof(fp1))break;/*finished*/
    if(ferror(fp1)||ferror(fp2)||
        /* if we are here, c==EOF but
         * no real EOF on fp1
         * therefore try to put it out
         */ (putc(c,fp2), ferror(fp2)){
           fclose(fp1);
           fclose(fp2);
           return FAIL;
    }
}
```

- why **c** needed? why not

```
    while(!feof(fp1))putc(getc(fp1), fp2);
```

?

Beware of off-by-one errors !!

ungetc:

pushes back the last character read
Ex.:

```
    /*skip until first non-blank */
    #include <stdio.h>
    #include <ctype.h>

    void
    bskip(fp)
    FILE *fp;
    {
        int c;
        while ( isspace(c=getc(fp)) );
        ungetc(c , fp) ;
    }
```

- only one character
- only after read
- it's not I-O: external file not changed
- **rewind** and other *f.p.i.* manipulations will cause the pushback to be forgotten

ONE LINE AT A TIME

MEANINGFUL ONLY IN TEXT MODE

```
char * fgets ( s,max_length)
char * s ;
int max_length;
FILE *stream;
int fputs ( s, fp)
char * s;
FILE *fp;
```

- and their stripped down versions (**stdin-stdout)**

```
char gets ( s )
char * s ;
int fputs ( s )
char * s ;
```

fgets

- reads until EOF or newline or **max_len-1** characters
- puts them in s
- adds a null at the end
- returns **s** or **NULL** if read error or EOF before anything read
- WARNING : input newline is included in s !

gets

- almost like **fgets** on **stdin** , but discards the newline (history...)

fputs

- writes **s** (as it is!) to **fp** discarding the terminating null
- returns 0 if successful, non-zero on error

puts

- almost as **fputs** on **stdout**, but adds a newline

NOTE: often implemented through calls to **fgetc/fputc ->** not faster than direct use of **getc/putc.**

ONE BLOCK AT A TIME

MAINLY BINARY
ANSI

```
#include <stdio.h>

    size_t fread( void * block, size_t
         size, size_t nelem, FILE
         *stream);

    size_t fwrite(const void * block,
         size_t size, size_t nelem,
         FILE *stream);
```

- `size_t` is a `typedef` in `stdio.h`: usually `unsigned int` or `unsigned long int`
- `nelem` elements of size `size` are transferred
  - WARNING : this is not the same as transferring **nelem * size** bytes !!
- return number of elements transferred
  - if returned < `nelem` on output, error
    on input, EOF or error (`feof`);

NOTE : implementation dependent. Can be very fast, or use **fgetc/fputc** and be very slow.

RANDOM ACCESS

Getting the current f.p.i.
Setting f.p.i. to beginning-of-file
Setting f.p.i. to an arbitrary value

Getting the current f.p.i.

```
long ftell (stream)
FILE *stream;
```

- returns the current *f.p.i.* as a `long int.`
-- binary: number of characters from start
-- text : "magic" (to be used only with `fseek`)
-- **-1L** if failure

Setting f.p.i. to beginning-of-file

```
rewind (stream)
FILE *stream;
```

Setting f.p.i. to an arbitrary value

```
int fseek( stream,offset,base_sel)
FILE * stream ;
long offset ;
int base_sel;
```

- positions the *f.p.i.* at a distance `offset` from a **`base`**:

--- **`base_sel`** selects the base:

`base_sel == SEEK_SET`
base is beginning of file

`base_sel == SEEK_CURR`
base is current *f.p.i.*

`base_sel == SEEK_END`
base is end of file

--- **`SEEK_SET, SEEK_CURR, SEEK_END`** macros defined in `stdio.h` (in old compilers, 0, 1, 2)

--- `offset` can positive or negative

--- if in **text** mode, `base` must be `SEEK_SET` and `offset` must be the output of `ftell`

--- in binary mode, `SEEK_END` could give strange results if system pads bynary files

COMMENT

`fseek/ftell` could not work if file length cannot be encoded in a `long int`

for this general case, 2 other functions ANSI only

```
int fgetpos( FILE *stream, fpos_t
*pos);

int fsetpos ( FILE *stream, const
fpos_t *pos);
```

FILE BUFFERING

File buffering: data are passed to-from the file only in chunks of fixed size (from 512 B to a few kB)

unbuffered : minimum latency
if file I-O used for control purposes
buffered : maximum I-O efficiency (less calls to O.S., device, etc)

WARNING : C buffering concerns passing data to O.S., NOT to device (O.S. can buffer by itself, or not, O.S. dependent)

By default, files buffered (buffer size implementation dependent)

`stderr` unbuffered

```
    #include <stdio.h>

    char c_arr [ BUFSIZE ];

    main(){

        FILE *fp;
        /* declarations */

        setbuf ( stderr, c_arr );
        /* stderr becomes buffered,
        c_arr is buffer */
         setbuf ( stdout, NULL);
        /* stdout becomes unbuffered */

        .....
    }
```

- `BUFSIZE` defined in `stdio.h`
(called `BUFSIZ` in your compiler)
- must be used after `fopen` and before any I-O operation

```
int fflush( stream)
FILE * stream;
```

- if stream is buffered, write content of buffer to O.S.
- if `stream == NULL`, applies to all open streams;
- returns 0 (success) or EOF (failure)

ANSI ONLY

```
    int setvbuf ( FILE * stream ,char
    *buf , int mode , size_t buf_size);
```

- arbitrary size of buffer and buffering mode
- mode can be

`_IOFBF`	Full buffering
`_IOLBF`	Line buffering
`_IONBF`	No buffering

- `setbuf ( stream, buf );`
  is (almost)
  `setvbuf(stream,buf,_IOFBF,BUFSIZE);`

and

- `setbuf ( stream , NULL ) ;`
  is (almost)
  `setvbuf(stream, NULL ,_IONBF ,0) ;`

SELDOM USED, BUT IMPORTANT