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

INTERNATIONAL CENTRE FOR SCIENCE AND HIGH TECHNOLOGY
c/o 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/474 - 6

COLLEGE ON
"THE DESIGN OF REAL-TIME CONTROL SYSTEMS"
1 - 26 October

PROGRAMMING IN C LANGUAGE
(Parts 5 & 6)

A. NOBILE
International Centre for Theoretical Physics
Strada Costiera 11
34100 Trieste
Italy

ARRAYS , POINTERS, STRINGS

THE REAL THING !

C intertwins closely arrays and pointers

C handles strings as character arrays

ARRAYS

collection of variables of same type

double ar[1000];

ar is a 1000 elements array;
the elements are denoted **ar[0],ar[1]...ar[999]**
each of them is **double**
COMMENT: declaration by example:
can read:
1000-th element of **ar** *is* **double** (and the other too!)

WARNING : no way of specifying a range not starting from 0

WARNING : **ar[1000]** is NOT an element of the array! **ar[999]** is the last one !

WARNING: array size must be *constant*

```
f(int m){
char var_sized_array[m]; /* FORBIDDEN */
....
```

Like Pascal? (bleah) NOT QUITE

MULTIDIMENSIONAL ARRAYS

int t_d [2] [3];

[] associates left to right
means (**t_d [2]**) **[3]**
can read:
third element of second element of **t_d** *is* **int**
therefore
second element of **t_d** *is array of 3* **int**
t_d *is array of 2 arrays of 3* **int**

Valid elements:

t_d[0][0] t_d[0][1] t_d[0][2] t_d[1][0] t_d[1][1] t_d[1][2]

t_d[0] **t_d[1]**

WARNING
No way to refer to a column
Memory storage BY ROWS
- opposite of FORTRAN
- important to remember if using pointers
Seldom used

WARNING
What is the meaning of **t_d[0,1]** ?

COMMENT
arrays of anything (arrays of arrays special case)

INITIALIZING ARRAYS

ALREADY MET

```
int b = 1 ;
int *pi = &b;
```

ARRAYS

```
int a[6] = { 1, 0, -4, 4, 2, 7 };
```

ANSI : OK
Old C : only if static or global array (see later)

BUT ALSO (QUITE USEFUL)

```
int a[] = { 1, 0, -4, 4, 2, 7 }; /* assumed size */
```

The compiler will make **a** an array with 6 elements

MULTIDIMENSIONAL

```
int t_d [2] [3] = { { 0 , 1, 3 } ,
                    { -1, 4, 6 }
                  } ;
```

OR

```
int t_d [] [3] = {   { 0 , 1, 3 } ,
                     { -1, 4, 6 }
                 } ;
/* array of arrays : second dimension required */
```

ARRAYS AND POINTERS

```
int ar[5], *ip;

ip = &ar[0]; /* nothing new */
```

>>>> pointer arithmetics

FUNDAMENTAL

ip + 1 equals &ar[1]

if **ip** points to an element of an array of any type,
ip+1 points to the next one, and so on
- pointers are not integers
- pointers are not memory addresses

```
int ar[5], i;

for (i=0 ; i<5 ; i++) ar[i] = 0;
```

equivalent to

```
int ar[5], *ip;

for (ip = &ar[0] ; ip < &ar[5] ; ip++) *ip = 0;

/* legal: using the address of a[5] is legal even if
 * a[5] does not exist
 */
```

Is it better?
IF arrays are accessed sequentially,
pointers faster except if optimizer very good.
(Not true on vector machines, helas)

ONLY way of passing variable size arrays to functions

IN FACT ARRAYS DO NOT EXIST

C recognizes an array only
- in declarations
- as an operand to **sizeof**

```
long int arr[4], s;

s = sizeof arr; /* returns 16 */
```

IN ALL OTHER CONTEXTS,
ar is a pointer to **&ar[0]**
ar[i] is synonimous of ***(ar+i)**

BUT : array names are not pointer VARIABLES!

```
float ar[5], *p;

p = ar ; /*legal : p=&ar[0] */
ar = p ;/* illegal: array names are "constant
           pointers" */
&p = ar ;/*illegal: &p is a pointer value not a
           variable*/
ar ++ ;/* illegal: array names are "constant
           pointers "*/
ar[1] = *(p+3) ; /*legal*/
ar[1] = *(ar+4) ;/* legal, but crazy*/
*p = 5[ar] ; /* AARGHHH ... legal means ar[5]*/
```

MORE ON POINTER ARITHMETICS

```
int ar[20], *p1, *p2, br[10], *q;

p1 = &a[10];
p2 = &a[15];
q  = &br[5];

p1 != 0          /*true, legal*/
p1 < p2          /*legal*/
p1+5 == p2       /* pointer +integer */
p2-5 == p1       /* pointer -integer */
p2-p1 == 5       /* pointer - pointer */
p1+p2            /* illegal */
2*p1             /* illegal */
p1++ ;           /* now p1 points to a[11] */
p1 < q;          /* result undefined */
q - p1           /* result undefined */
q != p1          /* true, legal */
q != 100         /* illegal */
```

- No operations between pointers to different types
- **< > <= >=** - allowed only between pointers to different element of same array or structure (work anyhow...)
- tests for equality allowed for arbitrary pointers of the same type
- comparison with **int** 0 allowed (test for NULL pointer)

PASSING ARRAYS TO FUNCTIONS

- Always interpreted as pointers
- Array notation allowed

```
int sum_of_elem(int ar[] , int num_of_elem){
     int i , s=0 ;

     for(i=0 ; i<num_of_elem ; i++)
          s += ar[i] ;
          return s ;
}

int sum_of_elem(int *ar , int num_of_elem){
     int *p , s=0;

     for( p=ar ; p < ar+num_of_elem ; p++)
          s += *p;
          return s;
}

int sum_of_elem(int *ar , int n_elem){
     int *p , *pend , s=0;

     for( p=ar , pend=ar+n_elem; p < pend ; p++)
          s += *p;
          return s;
}
```

OR EVEN

```
int sum_of_elem(int ar[100] , int num_of_elem){
      int i , s=0;

      for(i=0 ; i<num_of_elem ; i++)
            s += ar[i];
            return s;
}
```

are exactly the same. (Third one different if compiler inserts array subscript checking)

Moreover

```
int sum_of_elem(int ar[]){
      int n,i,s;

      n = sizeof ar / sizeof(int) ;
      for (i=0 ; i < n; i++) ....
```

WOULD NOT WORK : **sizeof ar** is **sizeof (int *)**
Not even if **ar** declared with a size

```
int sum_of_elem(int ar[100])
```


STYLE COMMENT

C ugly style

```
int sum_of_elem(int *ar , int n_elem){
      int *p , *pend , s=0;

      for (pend=(p=ar)+n_elem; p<pend; s+=
            *p++) ;
      return s;
}
```

DOES WORK

PASSING MULTIDIMENSIONAL ARRAYS

```
int f(int a[][5])
 /* Note :second dimension requested */
{ ....a[i][j]...}
```

or

```
int f( int (*a)[5] )
/* pointer to a[0], which is an array of 5 ints */
{.... (*a+i)[j] ...}
```

or

```
int f ( int **a)
/* pointer to a[0], which is an array, therefore
 * a pointer to a[0][0] . COMMON FORM
 /
{     .... *( (int *)a + i*5 + j) ....
      /* any information about 5 lost */
```

or better

```
int ff( int **a, int size){
      .... *( (int*)a + i*size + j) ...
}
```

WHY C PROGRAMMERS AVOID MULTIDIMENSIONAL ARRAYS ?

Example

Sorting: bubblesort

```
/* sort an array of ints in ascending order */
#define FALSE 0
#define TRUE 1
void bubble_sort(int ar[], int size){
      int j, temp, sorted=FALSE;
      while ( !sorted ){
            sorted = TRUE; /*assume it's sorted */
            for (j = 0; j < size-1; j++){
                  if (ar[j]>ar[j+1]) {
                        sorted = FALSE;
                        temp = ar[j];
                        ar[j] = ar[j+1];
                        ar[j+1] = temp;
                  }
            }
      }
}
```

with pointers

```
/* sort an array of ints in ascending order */
#define FALSE 0
#define TRUE 1
void bubble_sort(int *ar, int size){
     int *pj, temp, sorted=FALSE;
     while ( !sorted ){
          sorted = TRUE; /*assume it's sorted */
          for (pj = ar; pj < ar+size-1; pj++){
               if (*pj > *(pj+1)) {
                    sorted = FALSE;
                    temp = *pj ;
                    *pj = *(pj+1) ;
                    *(pj+1) = temp;
               }
          }
     }
}
```

STRINGS

arrays of **char**
terminated by a **null ('\0')**

String constants
everything in quotes
" this is a string"

Compiler adds the terminating '\0'

NO specific string constructs
-> hard programming
-> very efficient code
-> library essential

Defining a string variable

```
char str1[10];
char str2[] = "string" ;/* compiler makes str2 7
                           elements , 6 of string + null
                           */
```

or

```
char str2[] = {'s','t','r','i','n','g','\0'};
char str3[10] = "one" ;/* Ok */
char str4[3] = "one" ;/* wrong, no room for null
                         */
```

BUT ALSO (OFTEN USED)

```
char *s="new string";
```

- creates a string constant containing the value "new string"
- creates a character pointer (**s**)
- initializes **s** with the address of the constant

DIFFERENCE

```
str1 = p; /* illegal */
s = p; /* legal; the constant string is lost */
```

STRING ASSIGNMENT

- strings are arrays or pointer to arrays
- arrays take value by filling -> *COPYING*
- assignment to pointers only

```
char carray1[10], carray2[10];
char *p1 = "10 spaces";
char *p2;

carray1 = "not ok" ; /*illegal: cannot assign to
                       array name */
carray1[1] = 'a';  /* OK */
carray1[2] = '\0'; /*now carray contains "a" */
carray2 = carray1; /* illegal */
p1 = "OK";         /* legal; creates new string and
                       puts its address in p1 */
p1[5] = 'c'        /* wrong */
p1[1] = 'N'        /* should transform OK in NK :
                   could not work, dangerous */
```



```
*p2 = "NO"        /* type mismatch */
p2 = "yes"
```

STRINGS vs. CHARS

```
char c = 'a';
char *s = "a";

*s = 'b';
*s = "b"; /* illegal */
s = "b"; /* OK */
s = 'b';   /* illegal */
```

DO NOT CONFUSE INITIALIZATION WITH ASSIGNMENT- with any type

```
float f;
float *pf = &f ; /* OK */
```

BUT

```
*pf = &f ; /*illegal */
```

COMPARING STRINGS

```
char arr1[]="str1" , arr2[]="str1";
char *s1=arr1, *s2=arr2;

if (s1 == s2)....

if (arr1 == arr2)
```

test fails, because compares character pointers: equal if they point to same object, not if they point to objects containing same value

```
if (*arr1 == *arr2)
```

wrong: compares only the first character

```
/* function to compare strings
 * return TRUE if equal
 */
int str_eq(char *s1, char *s2){
      while ( *s1 == *s2 ){
            if (  *s1 == '\0' )return 1;
            s1++;
            s2++;
      }
      return 0;
}
```


STRING COPY

```
char st1[20] , st2[]="wow!";

st1 = st2 ;/* illegal */

strcpy(st1 , st2) ; /* library function only way */
......
void
strcopy( char s1[], char s2[]){
      int i;
      for( i=0 ; s2[i] ; i++)
            s1[i] = s2[i];
      s1[i] = '\0';
}

void
strcopy( char *s1, char *s2){

      while ( *s2 ) *s1++ = *s2++;
      *s1 = '\0';
}
```

- **while (*s2)** means **while (*s2 != 0)**

- ***s2++** right associative: ***(s2++)**
  use current value of **s2**, then increment **s2**
- **s1** and **s2** are copies of the arguments passed, can be modified safely

- their values are the addresses of the arguments

being passed, that are actually modified

- **return** not needed if no value returned

BETTER

```
void
strcopy( char *s1, char *s2){
     while ( *s1++ = *s2++) ;
}
```

ARRAYS OF POINTERS

STRUCTURES AND UNIONS

To group heterogeneous objects:

name
social security number
date of birth : month day year

```
struct vitalstat
{    char vs_name[19], vs_ssnum[11] ;
     struct {
               short month, day, year;
             } vs_birth_date ;
} vs1;

struct vitalstat vs2;
```

- **declare structure** *tag***vitalstat**
- declare variables **vs1 vs2** of that type

- **struct** *tag name* **{** *list of declarations***}**

- **struct** components can be other **structs**
  WARNING : but of different types
- tag name is optional

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>
/* reads in two complex arrays */
main()
{
     typedef struct { float re,im} Complex;

     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>
main()
{
     struct complex { float re,im}  ;

     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

```
#include <stdio.h>
typedef struct { float re,im} Complex;
/* placed here it is GLOBAL for this file */

/* reads in two complex arrays
 * and computes their dot product
 */
main()
{

     Complex v1[10], v2[10];
     double dotprod( Complex *v1, Complex *v2,
                          int n);

     for ( i=0; i<10 ; i++ )
          scanf(" %f %f %f %f " ,
                 &v1[i].re , &v1[i].im, &v2[i].re , &
                 v2[i].im) ;
     printf (" %f ", dotprod(v1 , v2, 10) );
}
double
dotprod( Complex *v1, Complex *v2, int n);
{
     double d=0;
     Complex *vend=v1+n;

     for( ; v1 < vend; v1++ , v2++)
      d += (*v1).re * (*v2).re - (*v1).im * (*v2).im ;

     return d;
}
```

(*v1).re UGLY. CAN BE TERRIBLE

```
struct b { int data;
           ....
}
struct a{ struct b * other;
           ....
           }
struct a *pa;

(*(*pa).other).data
```

NEW OPERATOR ->

p->x IS **(*p).x : pa->other->data**

```
dotprod( Complex *v1, Complex *v2, int n);
{
     double d=0;
     Complex *vend=v1+n;

     for( ; v1 < vend; v1++ , v2++)
          d += v1->re * v2->re - v1->im * v2->im ;

     return d;
}
```

OPERATIONS ON STRUCTURES

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

ANSI ONLY:

- assignment:

```
struct complex z1,z2;

z1=z2;
```

- passing as argument to functions

```
double cprod ( Complex z1, Complex z2){
     return ( z1.re * z2.re - z1.im * z2.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;
}
```

NOTE : Old C allows passing and returning *pointers to structures* !

COMMENT : In Old C very wide use of pointers to structures.

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

LINKED LISTS

Each node declared

```
struct list_node{
      int data /* or whatever */;
      struct list_node *next;
}
```

- Contains *a pointer* to itself (allowed)

- **list_node** known as a *structure tag* as soon as encountered in first line , therefore **struct list_node *** understood;

- example of *partial* or *incomplete declaration*: declare an *tag* to refer to a function, then refer to it through pointers; complete declaration before declaring any variable

Example:

```
struct s1 ; /*incomplete */
struct s2 {
      int something;
      struct s1 * cross;
}
struct s2 {
      float something_else;
      struct s2 *cross2;
}
```

- **typedef** WOULD NOT WORK
  (ONLY place where *tags* NEEDED)

```
typedef { int data;
            List_elem *next;
          /*wrong: List_elem unknown */
} List_elem;/* List_elem known only after this */
```

WARNING:
partial declaration is just obtained by mentioning name:

```
struct abc{ struct xyz *p}; /*struct xyz now
                              partially declared */
```

NEVER USE, VERY DANGEROUS


```
/* this program creates a linked list and
 * prints it out following the pointers
 */

#include <stdio.h>
struct list_ele{
      int data;
      list_ele *next;
} ar [10];
main()
{
      struct list_ele *lp;

      ar[0].data = 5;
      ar[0].next = &ar[1];
      ar[1].data = 99;
      ar[1].next = &ar[2];
      ar[2].data = -7;
      ar[2].next = 0; /* null pointer: end of list */

      lp = ar;
      while (lp) {
            printf (" contents %d\n", lp->data);
            lp = lp->next;
      }
      exit(0);
}
```

- move from one element to the next following pointers
- array structure not used at all

DYNAMIC OBJECTS

Lists are typical example: array structure not used.

add node to the end of a list:

create a node
put new data in its **data** component
put a pointer to the new node in the (null) **next** component of the last element
put a **NULL** pointer in the **next** component of new node

delete a node from the end of a list:

detach the node from the list, putting a NULL in the **next** component of previous node
delete the node

CREATING an object

- obtain from the system enough memory to contain a copy of the object;
- handle that memory as if it was an object

```
struct list_ele *p;

l_sz = sizeof( struct list_ele );
p = ( struct list_ele *) malloc ( l_sz);
```

malloc (size) requires to the system to provide a block of **size** bytes, and returns a pointer to this block (NULL if memory not available)
(struct list_ele *) is a *cast* that transforms the pointer returned by **malloc** int a pointer to **list_ele.**

```
p->data = new_value;
```



```
p->next = NULL;
last->next = p; /* assuming last points to the
                   last element
                */
```

COMMENT:
ANSI says **malloc** is of type **void ***
meaning a pointer that can be casted to point to any type;
Old C says malloc is **char ***, but that it returns a value that can be casted to point to anything...;
In both cases casting does not cause any change in the returned pointer.

p useless:

```
last->next = ( struct list_ele *) malloc(l_sz);
last->next->next = NULL;
last->next->data = new_value;
last = last->next; /* if you need it again */
```

Deleting an object:

- Only if object created by **malloc;**
- **free (p);**

Deleting the first element of a list

```
struct list_ele *list;

if (list) /* if list empty, do nothing */
{    struct list_ele *temp;
     temp = list;
     list = list->next; /*first node unlinked*/
     free(temp);
}
```

Would the above be good as the body of a function?

WARNING:

```
struct list_ele *list, *list1;
     list1 = list;
.....
     if (list) /* if list empty, do nothing */
     {    struct list_ele *temp;
          temp = list;
          list = list->next; /*first node unlinked*/
          free(temp);
     }
.....
     list1 -> data .../* AAAARGGHHHH */
```

DANGLING POINTERS problem

WARNING 2

```
while ( something){
     allocate memory
     use it
     forget it (without freeing)
}
```

causes difficult to trace problems (memory can get exhausted depending from path in the program ,data, etc.)

MORE EXAMPLES ON LISTS

```
/* remove last element; if only one element,
 * list pointer cahnged to NULL
 */
#define NULL 0

void
remove_end(struct list_ele **list){
     /* we get the ADDRESS of the pointer to the
      * first element, because we could need to
      * modify it if there is only one element in
      * the list
      */
     list_ele  **previous = list , *curr = *list;
     /* "previous" is a pointer to the pointer used
      * to arrive to the element to which "curr"
      *  points: used to modify this pointer when
      * we delete *curr
      */
     if ( ! curr ) return ; /* list empty, do nothing */
          while (curr->next){ /*curr not last */
               previous = &curr->next ;
               /* "previous" points to the
                * pointer used to  access next
                * element
                */
               curr = curr->next ;
               /* access next element */
          }
          free (curr); /* delete last element */
          *previous=NULL;
           /* make pointer to it NULL */
     }
}
```



```
/*Assume list is ordered and insert in order
 * return NULL if failed
 * return pointer to beginning of list if successful
 */
#define LSZ  (sizeof struct list_ele)
#define NULL 0

struct list_ele *
add_ordered(struct list_ele * list, int newval){
     struct list_ele * temp, *curr;

     if (list == NULL) { /*empty list case */
          if (list = (struct list_ele *) malloc (LSZ);){
               /* if allocation successful, fill node */
               list->data=newval;
               list->next=NULL
               return list;
          }else{
               return NULL;
          }
     }
     /*normal case */
     curr = list;
     while (curr->next &&
               curr->next->data < newval)
          curr = curr->next ;/* skip */
     if (temp =  (struct list_ele *)malloc(LSZ)){
          /* if new node allocation OK
           * fill it ...
           */
          temp->data = newval;
          /* link with end of list ... */
          temp->next = curr->next;
```

```
        /* and link with previous part of list */
        curr->next = temp;

        return list;
    } else {
        return NULL;
    }
}
```

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 {re,im} Complex;
Complex arr[100];

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

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

SCOPE of identifiers: where a NAME can be used

DIFFERENT BUT RELATED PROBLEM

```
main()
{ ......

     f1();
     ...
     f2();
}
void
f1(void){
     int i;
     f2();
}
void
f2(void){
     printf ("%d",i);/*WRONG */
}
```

- 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 the declaration point to the end of the compound statement;

```
main(){
     int q[100];
     long int s;
     long int sum( int arr[], int n);

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

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


2) extern : definitions outside any function . Created when program starts, survive till program end. Accessible from other files, through suitable *allusions.*

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 it outside any function, the file from the allusion to the end;

WARNING:
storage class -> variable
scope -> name
extern variable can have local name

File a.c:

```
#include <stdio.h>
struct complex {float re,im} ;
    /*defines the tagcomplex : global to the file*/
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(int i);
        extern int errcode;
        /* allusions */
        void test(void);/* prototype*/
        struct complex z;
        /* struct complex has file scope */
        if(carr[1].re==0.0) errcode=1;
        /* carr has file scope */

}
void
test(void){
    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 {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(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


```
#include <stdio.h>
/* basic data store not directly accessible
 *  from outside
 */
static  struct vsstat  *list_of_names;
/* public procedures */
void
addname(char *name){
.......
}
int deletename(char *name){
.....
}
struct vsstat *
search(char *name){
.....
}
/* private procedures
 * NAME CLASHES impossible !
 */
static void
compact_list(void){
.....
}
static struct vsstat *
create_node(char *name){
.....
}
static void
error ( 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, outside control fom the program.
THEREFORE, any store or load operation requested by the program MUST be actually performed (no optimiziation 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 );
```

-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

DEFAULT CONVERSIONS

```
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;
     ......
}
```

CALLING FUNCTIONS

- evaluate expressions passed as arguments;
- convert values according to function prototypes if any or according to default conversions;
- use these values to initialize formal arguments
- 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)


```
called_func( int , float );

main(){
      int i = 1;
      called_func ( i , 2*3.5 );
}
void called_func (int iarg, float farg){
      float tmp;
      tmp= iarg * farg;
      iarg ++ ; /* no effect */
}
```

CALL BY REFERENCE

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

```
called_func( int * , float );

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

Arrays cannot be passed by value:

```
void func(int arr[]);

int arr[10];
func(arr);
```

is identical to

```
void func(int *arr);

int arr[10];
func(&arr[0]);
```

Functions cannot be passed by value (WHAT?)

EXCURSUS : pointers to functions
Often used !

function name is constant pointer to function
- like array name

```
int f(int n);
int op( int func()); /* interpreted as
                      * int op ( int (*func) ())
                      * DEFAULT CONVERSION
                      */
int (*pf)() /* pf pointer to function returning int */

pf = f ; /* pf = &f wrong ;
         * pf = f() wrong ;
         * pf = &f() wrong ;
         * /
```



```
op(f); /* same as op(pf) */
```

Structures are passed by value (ANSI)

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;**

WARNING : mixing prototyped declarations with non-prototyped definitions can cause problems

---

RETURNING FROM FUNCTIONS

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

- **return**
- flow through the end

RETURNING A VALUE

```
double squareroot(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

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; /* PI is pointer to int */
typedef PI AP[5]; /* AP is a 5-elements array of
                     pointers to int*/
typedef AP *FP() ;/* function returning pointer to
                     AP */

FP  *x;
```


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*

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

Two basic types of streams : *text* and *binary* (ASCII)

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

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 the type of an object 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

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

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( 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 *s);
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**

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

MODES

for text streams

"r"	read only
"r+"	read-write
"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 existing file. Created if not existing.
"a+"	append and read.

binary streams (ANSI)

"rb", "r+b" etc.

Ex.

```
/* open with error message */
#include <stdio.h>
#include <stddef.h> /*ANSI: NULL */
FILE *
openfile( char *file_name, char *access_mode){

     FILE *fp;
     if ( ( fp=fopen(file_name, access_mode) )
          == NULL )
          fprintf( stderr, "Error opening file %s "
               " with access mode %s",file_name,
               access_mode);
          return fp;
}
```

- WARNING : **(fp = fopen()) == NULL**
  parenthesis required! common mistake
- **"a" "b"** is **"ab"** (ANSI only)
- **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+");
```

OTHER

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

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

often used with standard streams

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

IMPORTANT WARNING

Streams open for 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

---

close: disassociates a stream from a file.

int fclose(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(const char *format, ...);

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

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

NOTE : **scanf** IS **fscanf(stdin, ...)**
sscanf does conversion but not input, using in_string as the source of characters (FORTRAN INTERNAL FILE)
NOTE : **...** is the new ANSI form to specify functions with a variable number of arguments.

NOTE : arguments must be POINTERS to variables

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

ferror or **feof** to check for status

FORMATTED WRITE

int printf (const char *format, ...);

int fprintf (FILE * stream, const char *format,...);

int sprintf (const char * in_string, const char * format, ...);

NOTE: **printf** is **fprintf(stdout,..)**

NOTE : arguments must be VALUES

OUTPUT FORMAT

can contain two types of objects:

ordinary character : copied to output
conversion specifier:

CHECK THE MANUAL

UNFORMATTED INPUT-OUTPUT

ONE CHARACTER AT A TIME

Already met

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

-refer to **stdin / stdout**

GENERAL

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

special:
int ungetc(int c , FILE *stream);

- RETURN EOF if error or end-of file on read
- - otherwise return the character read or written

- Macros (defined in **stdio.h**)
-- expanded by preprocessor
--- FAST

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

Ex.:


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

#define FAIL 0
#define SUCCESS 1

int copyfile (char *infile,char * 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 );
     fclose (fp1);
     fclose (fp2);
     return SUCCESS;
}
```

- note cleanup in case of failure
- **feof** needed in binary mode:
  getc returns EOF at End of File
  EOF is <0 -> not a letter, if in text mode
  COULD BE 8-bit pattern (often -1)

- why **c** needed? why not

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

---- Beware of off-by-one errors !!
- how to protect against output error (disk full?)

ungetc:

pushes back the last character read
Ex.:

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

void
bskip(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 ( char * s , int max_length ,
                FILE *stream );
int fputs ( const char * s, FILE *fp);
```

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

```
char gets ( char * s ) ;
int fputs ( const 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 **stream,** 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 ->** slower than direct use of **getc/putc.** CHECK

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 < **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 (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

void rewind (FILE *stream);

Setting f.p.i. to an arbitrary value

int fseek(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

could not work if file length cannot be encoded in a **long int**

for general case, 2 other functions ANSI only

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

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

FILE BUFFERING

unbuffered : minimum latency
if file I-O used for device control (?)
buffered : maximum I-O efficiency

WARNING : no buffer = data passed to O.S.
NOT to device (O.S. dependent)

By default, files buffered (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**
- must be used after **fopen** and before any I-O operation

int fflush(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

WARNING :
cannot replace operating system calls

THE C PREPROCESSOR

Already met:

#include

#define

ANSI greatly expanded it.
Here only elementary usage

Takes code containing preprocessors directives
Transforms it into legal C without them

Works line by line (C does not care newlines)
Does not obey scope rules:
    definition holds from definition point to
    end of file

--> USE SPARINGLY

Introduces C-specific things

--- > NOT A GENERAL-PURPOSE MACRO SYSTEM

#define
2 versions : function-like and not

#define EOF (-1)

.....

if (c == EOF)
is translated into
if (c == -1)

#define FMAC(a,b) a * b /*poor, see later*/

FMAC(p->data, q[4])
is translated into
p->data * q[4]

#define max(x,y)((x)>(y)?(x):(y))

#define UPPER(c) ((c)-'a'+'A') /*ASCII only */

RESCANNING

#define EOF (-1)
#define readc(c) ((c=getchar())!=EOF)

PRACTICAL RULE :
macro names (not function macros)
are all uppercase, C identifiers are lower case or mixed case

#define A a,b
#define strange(x) x-1
strange(A)

strange should be replaced
it has one argument , **A**
A should be replaced
A becomes **a,b**
strange now has 2 arguments?
ANSI says **no;** try yours
and what if
#define strange(A) something
Etc. : DON'T TRY

WARNING

#define PA (a)

This one defines PA to be (a), not PA(a) being nothing. SPACE BETWEEN NAME OF MACRO AND (

WARNING

#define FILENAME myprog.c
printf ("compiled from FILENAME\n");

Does not work : strings are a single object to preprocessor

#define FILENAME "myprog.c"
printf ("compiled from %s\n" , FILENAME);

ANSI has defined specific operators for this kind of situation.

WARNING

FMAC (p+q, l+m)
becomes
p + q * l + m
probably wrong.

#define DOUBLE(x) x+x

3* DOUBLE(x)
becomes
3* x + x
wrong

Correct format
#define DOUBLE(x) ((x) + (x))
#define FMAC(a,b) ((a) * (b))

WHY TO USE FUNCTION MACROS ?
- increase readability
- faster to evaluate than real functions

#undef *identifier*

Causes the definition to be forgotten

Ex.
#include <stdio.h>
#undef BUFSIZE
#define BUFSIZE 1024

#include <file>
#include "file"

#define NAME "file"
#include NAME

Includes can be nested

Conditional compilation

#ifdef ***identifier***
#ifndef ***identifier***
#if ***constant expression***
#else
#endif

- To select pieces of code that are machine dependent
- To turn on-off parts of code used for debugging

```
#define M6809
.....
#ifdef M6809
typedef long int Int;
#endif
#ifdef M68020
typedef int Int;
#endif
```

or (UNIX 1983 Source)

```
typedef struct {
#if vax || u3b
     int _cnt;
     unsigned char * _ptr;
#else
     unsigned char * _ptr;
     int _cnt;
#endif
     unsigned char *_base;
     char _flag;
     char _file;
} FILE
```

- **vax || u3b** is a constant expression. If defined and not 0, expression not 0, etc.

WARNING

```
#define NULL 0
#if NULL
```

would fail.

C LIBRARIES

DEFINED BY ANSI STANDARD

NOT REQUIRED:
- required in "hosted" systems
- can be missing in standalone systems ("bare" C)

STANDARD HEADERS: contain macro names and typesused by standard libraries

WARNING:
- identifiers defined in standard headers are reserved . Should not be redefined or reused (like all the C keywords)
- names starting with _ are reserved

<assert.h>	**<math.h>**	**<stdio.h>**
<ctype.h>	**<setjmp.h>**	**<stdlib.h>**
<float.h>	**<signal.h>**	**<string.h>**
<limits.h>	**<stdarg.h>**	**<time.h>**
<locale.h>	**<stddef.h>**	

Sections of the library:

I-O	**<stdio.h>**
String handl.	**<string.h>**
Debugging:	**<assert.h>**
Character handl.:	**<ctype.h>**
Time, date:	**<time.h>**
General utilities	**<stdlib.h>**
Implementation	**<limits.h>**
	<float.h>
	<locale.h>
Exceptions	**<signal.h>**
	<setjmp.h>
Var. num. of arg.	**<stdarg.h>**
Math.	**<math.h>**

ASSERTION CHECKING

#include <assert.h>
.....
    assert (a>b);
.....

if **a>b** nothing;
else
    print text of expression (**a>b** in this case)
        , file name, line number
    abort
If NDEBUG defined, expression not evaluated and test performed
#define NDEBUG
#include <assert.h>

all **assert** turned off

EXCEPTION HANDLING AND NON-LOCAL TRANSFER

EXCEPTION: occurs unexpectedly or infrequently
    (error conditions)
    can be originated outside program control

    - hardware exception: division by 0
    - user exception : interrupt key

In C : signals
-can be generated by the hardware or by the software
-cause the execution to be transferred to a *signal handler*
-programs can establish their own signal handlers
-a default signal handler (implementation dependent) is always available
-program can send a signal to themselves

NOTE : sending signal to another program is an O.S. problem

ANSI defines a minimum of 8 signals: more are implementation dependent.

They are **int,** defined in **<signal.h>**

SIGABRT	calling **abort** library function
SIGFPE	illegal floating point operation
SIGILL	illegal instruction
SIGINT	interrupt (from keyboard?)
SIGSEGV	illegal memory reference
SIGTERM	software termination (sent by another program?)

Default signal handler, called SIG_DFL,
defined in **<signal.h>,** typically aborts the program

Alternative signal handler, called SIG_IGN,
ignores the signal.

User defined signal handler:

```
void handler( int sig_number){
     ....
}
```

NOTE : SIG_DFL has type void (*SIG_DFL)(int);
Pointer to function returning void

Handlers associated to signals calling **signal**


```
#include <signal.h>
void fpe_handler(int sig_number);
main()
{
     ...
     signal(SIGFPE, fpe_handler);
}
```

signal returns a pointer to the old handler

```
#include <signal.h>
..
void int_handler(int sig_number);
{
     void (*old)(int) ;
     ...
     /* install handler only if signal not ignored */
     if (( old=signal(SIGINT, SIG_IGN)) !=
          SIG_IGN )
          signal(SIGINT,int_handler);
     ...
}
```

Later, **old** can be used to reinstall the original handler.

COMMENT
full prototype of **signal** is
void (*signal (int sig, void (*func)(int))) (int);

or (better)

typedef void (*func)(int) HANDLER;
HANDLER signal (int , HANDLER);

HANDLER STRUCTURE

- First, call **signal** again (usually, signals go back to default handler every time raised)
- Do whatever needed
- Either **return** (execution continues from point of exception
-- or jump somewhere else with **longjmp**

Example:


```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#define PR putchar(':')
void float_err(int), keyboard_intr(int);/*handlers*/

float a,b, result; char opr; /*global ->visible from
handlers*/
main(void)
{
     signal(SIGFPE, float_err);
     signal(SIGINT, keyboard_intr);

     while(PR,scanf("%f%c%f",&a,&opr,&b)!=
               EOF)
     {
          switch(opr){.....
               /* calculator as in lect. 1 */
               ....
          }
     }
}
void closing( char *message, int exit_code){
.....
}
void float_err(int i){
     signal (SIGFPE, float_err);
     printf(" Floating point error\n");
     a=b=1.0;/*safe value*/
}
```

```
void keyboard_intr(int i){
      signal(SIGINT, keyboard_intr);
      printf("Do you want to quit? Y or N:");
      switch (getchar()) {
            case 'Y' : case 'y' :
                  closing("Regular end", 0);
            default: printf("continue\n");
      }
}
```

- handlers can exit, or return; return resumes execution from exception point

- problem : scanf returns EOF if interrupted.
  resuming execution within **scanf** is not continuing.

-- > continue from a safer location

---

longjmp, setjmp

```
#include <setjmp.h>
/* defines type jmp_buf */
jmp_buf env;

int setjmp(jmp_buf env);
```

- stores in **env** all the information to resume execution from the point it is called
WARNING : not a checkpoint!
- returns zero

```
void longjmp(jmp_buf env, int val);
```

- if **env** filled by **setjmp**, jumps to the return point of setjmp, but returning **val**; if **val == 0**, returns 1.

Ex.

```
#include <setjmp.h>
jmp_buf env; /* global !*/
main(){
      int v;
      ......
      if ( v=setjmp(env) )
            printf(" Coming from longjmp "
                  " value = %d", v);
.....
}
other_function(){
...
      longjmp(env, 1 );
....
}
```

FINAL EXAMPLE

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <setbuf.h>
#define PR putchar(':')
void float_err(int), keyboard_intr(int);
/*handlers*/

float a,b, result; char opr; jmp_buf env;
 /*global to be visible from handlers*/
main(void)
{
     signal(SIGFPE, float_err);
     (void)signal(SIGINT, keyboard_intr);
     (void)setjmp(env);

     while(PR,scanf("%f%c%f",&a,&opr,&b)!=
               EOF)
     {
          switch(opr){
               ......
          }
     }
}
void closing( char *message, int exit_code){
.....
}
```



```
void float_err(int i){
     signal (SIGFPE, float_err);
     printf(" Floating point error\n");
     a=b=1.0;/*safe value*/
}
void keyboard_intr(int i){
     signal(SIGINT, keyboard_intr);
     printf("Do you want to quit? Y or N:");
     switch (getchar()) {
          case 'Y' : case 'y' :
               closing("Regular end", 0);
          default: printf("continue\n");
          longjmp(env,1);
     }
}
```