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

The United Nations University

SMR/774 - 9

THIRD COLLEGE ON MICROPROCESSOR-BASED REAL-TIME CONTROL - PRINCIPLES AND APPLICATIONS IN PHYSICS
26 September - 21 October 1994

C PROGRAMMING LANGUAGE
Part II

Alvise NOBILE
International Centre for Theoretical Physics
P.O. Box 586
34100 Trieste
Italy

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!**)
- defines **today, yesterday** and **tomorrow** as 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**
- defines 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

```
struct vitalstat vs;
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(void)
{

 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(void)
{

 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) ;
}
```

Note: 1) difference between structure tag and type name
   2) No support for struct in I-O

POINTERS TO STRUCTURES

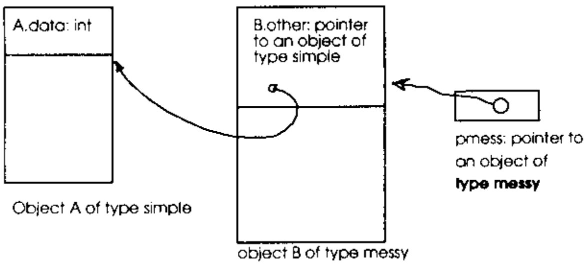
pointers to anything !

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

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

 Complex v1[1000];
 double cnorm2(Complex v[], int l);

 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( 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;
}
```

Structures 5

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

```
struct simple { int data;
   ....
} A;
struct messy{ struct simple * other;
   ....
   } B;
struct messy * pmess;
A.data=1;
B.other=&A;
pmess=&B;/* pmess points to B, whose
  field "other" points to A: we want
  the "data" field of A */
(*(*pmess).other).data
```



Structures 6

NEW OPERATOR ->

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

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

```
double cnorm2( 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**)
- assignment (**s1=s2;**)
- pass to function as argument
- return from function as value

Structures 7

```
struct complex {floar re,im;}
struct complex cprod (struct complex
  cp1, struct complex 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;
}
main(void)
{
...
struct complex c1,c2,c3,c4;
...
c1=cprod(c2,cprod(c3,c4));
...
```

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

---

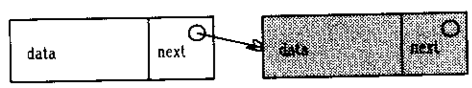
Structures 8

SELF-REFERENTIAL STRUCTURES

LINKED LISTS

Structures cannot contain themselves as components
Structures can contain pointers to anything as components, even pointers to themselves

```
struct list_node{
  char name[100];
  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** * is understood;

- example of *partial* or *incomplete declaration*: declare a *tag* to refer to a structure, then refer to it through pointers; complete declaration of structure before declaring any variable; can be used in general;

Example:

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


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

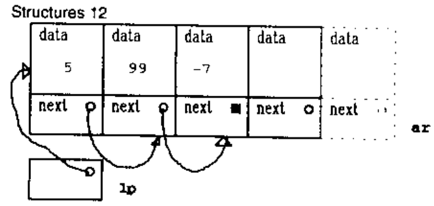
```
typedef { int data;
     ListElem *next;/*wrong:
       ListElem unknown here*/
} List_elem;/* type List_elem known
  only after this point */
```

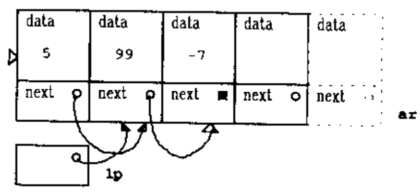
WARNING:
partial declaration is obtained just by mentioning name:

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

DANGEROUS: mistyping can be interpreted as legal partial declaration...


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

#include <stdio.h>
struct list_elem{
  int data;
  list_elem *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 to
  next: no next, end of list */

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

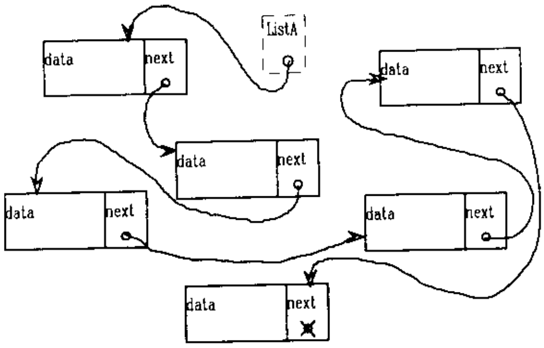





- move from one element to the next following pointers (lp=lp->next, NOT lp++)
- array structure not used at all

DYNAMIC OBJECTS

Lists are typical example: array structure not used.



Please note: most often, list referred to through a variable pointer to their first element:

```
list_elem *ListA;
```

Sometimes, through couple of variable pointers to first and last element (

```
struct list_id {
  struct list_elem *first, *last;
  }ListA;
```

Both approaches help dealing with empty list case.

New problems:
add node to the list (for instance,at the beginning):

- ***create*** a node
- put new data in its **data** component
- put a pointer to the current first node in the **next** component of the new node
- make the list to point to the new node

delete the first node of a list:

- detach the node from the list, making the list to pint to the node to which the **next** component of the first node points
- ***delete*** the node

CREATING an object of type ***T***

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

DELETING an object :

- return its memory to the system

```
#include <stdlib.h>
...
struct list_elem *ListA;
.... /* assume ListA point to first
  element of list */
struct list_elem *p;
int l_sz;
/* add a new element with content
  "new_value" to the beginning of
  listA */
l_sz = sizeof( struct list_elem );
p=(struct list_elem *)malloc (l_sz);
```

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


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

COMMENT:
malloc is of type **void *** meaning a pointer that can be casted to point to any type;
In both cases casting does not cause any change in the returned pointer.

Deleting an object:

- Only if the object was created by `malloc;`
- `free(p)` /* p pointer to the object to be deleted */

Deleting the first element of a list

```
struct list_elem *ListB;

if (ListB) /* if list empty, do
nothing */{
  struct list_elem *temp;
  temp= ListB;/*keep address of
            first node*/
  ListB=ListB->next; /*first node
            unlinked from ListB*/
  free(temp);/*delete old first
            node*/
 }
```

Would the above be good as the body of a function that performs list-element removal?
WARNING:

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

DANGLING POINTERS problem.

WARNING 2: Passing lists to functions:
Lists <=> pointers to their first elements
passing by reference if lists have to be be modified
=> passing pointer to lists => passing pointers to pointers to first element

```
function remove_first(List)
list_elem **List;
{ list elem *temp;
  if(*List){
  temp=*List;
  *List=(*List)->next;
  free (temp);
}
```

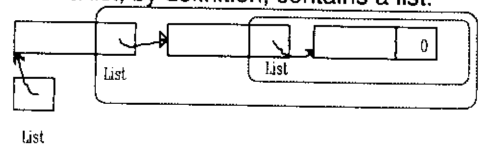
WARNING 3

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

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

List and recursion

If a list is a pointer to a struct one of which fields is a pointer to an object of the same type, then a list, by definition, contains a list:



Recursive programming: a function applied to a list can often be programmed like this

```
struct listelem;
typedef struct listelem *List;
struct listelem{int data; List next};

f1(List l){
if(l){
  dosomething(l->data);f1(l->next);
}
```

or

```
f2(List l)
{if(l){if(l->data has some property)
      dosomething;
      else f2(l->next);
}
```

Example: scan the whole list, printing all elements

```
void printlist(List l){
if(l){printf("%d\n",l-
  >data);printlist(list->next);}
}
```


Count nodes in a list

```
int n_nodes(List l){
if(l)return 1+n_nodes(l->next);
else return 0;
}
```

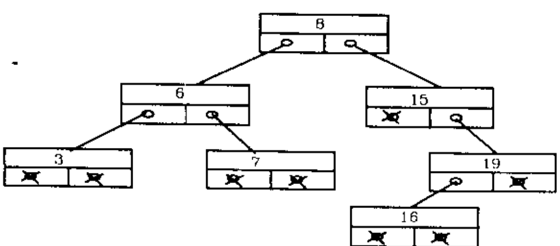
Very elegant, very powerful, not so efficient:

```
for(;l;l=l->next){ ... }
```

usually (much)faster

```
void printlist(List l){
  for(;l;l=l->next)
   printf("%d\n",l->data);
}
int n_nodes(List l){
  int c=0;
  for(;l;l=l->next)c++;
  return c;
}
```

Is recursion essential?
Binary trees



Search trees: for every node, (the data fields of) all the nodes in its left subtree are 'less' and (the data fields of) all the nodes in the right subtree are 'greater' then (the data field of) the node itself


```
typedef structure tree_node{
  int data;
  struct tree_node *left,*right;
} Treenode;

typedef Treenode *Tree;

/* print a binary tree of the above
  type in ascending order */
tree_print(Tree tree)
{ if(tree){
   tree_print(tree->left);
   printf("%d\n",tree->data);
   tree_print(tree->right);
  }
}
```

Simple!

```
#include <stddef.h>
/*return a pointer to a tree node
  whose data field equals the one
  passed as argument */
Tree
TreeSearch(int data,Tree tree)
{ if(!tree)return NULL;
  if(tree->data==data)return tree;
  if (tree->data>data)
  return TreeSearch(data,tree->left);
  else
  return TreeSearch(data,tree->right);
}
```

```
#include <stddef.h>
#include <stdlib.h>
/* Inserts 'newdata' in tree */
Tree
tree_add(Tree *tp, int n)
{
 if(!tp){ return NULL;}
/* NULL pointer passed: error */
 if(!*tp){ /* empty tree */
  if(
  *tp=(Tree)malloc(sizeof(Treenode))){
    (*tp)->data = n;
    (*tp)->left = (*tp)->right = NULL;
  }
  return *tp;/* if malloc failed *tp
  is NULL! */
 }
 if((*tp)->data > newdata)
  return tree_add(&((*tp)->left), n);
 else if ((*tp)->data < n)
  return tree_add(&((*tp)->right),n);
 else return *tp;
}
```


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

```
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(void){
  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(void){
  return( arr[i].re*arr[i].re +
  arr[i].im*arr[i].im);
  /* WRONG : i unknown */
}
```


i has a value when **norm** called, but its ***name*** unknown outside function **main**
The compiler detects the error

SCOPE of identifiers: where a NAME can be used

DIFFERENT BUT RELATED PROBLEM

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

COMPILES CORRECTLY

- **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 variables declared 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;


```
#include <stdio.h>
main(void){
  int q[100];
  long int s;
  long int sum(int arr[], int n
  );/*declaration, not definition!*/
  {
   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 );
  printf("%f\n",s);
}
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 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 (or external):
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 **external** variable can have local scope if allusion (declaration) is inside compound statement
Do not identify EXTERN (storage class) and GLOBAL(scope)

WARNING:
etern keyword is <u>not</u> the characterization of an extern variable!: it is the marker of an *allusion* to an extern variable.

<u>File a.c</u>:

```
#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(void){
  ....
  extern int fun(int i);
  extern int errcode;
  /* allusions */
  int test(void);/* declaration */
  struct complex z;/*definition: auto*/
  /* struct complex has file scope */
  test();
  if(carr[1].re==0.0)errcode=1;
       /* carr has file scope */
}
int
test(void)/*defines test: extern*/{
  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 */
```



```
}
```

<u>File b.c</u>

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

COMMENTS:

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

IMPORTANT LIMITATION:
each **extern** object should be *defined* in exactly **one** file! (that is its name should appear with the keyword **extern** in all files except one)
extern object are initialized to 0 by default

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,** like **extern**)

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;
  }
  /* normal processing */
  ...
}
```

auto would not work (WHY?)

3.2) Variables AND FUNCTIONS defined "at top level" like **extern** ones, but whose visibility is limited to the file of definition(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 */
struct vsstat{...};
static struct vsstat *listOfNames ;
/* public procedures */
int
addnam(char *name)
{
.......
}
int
delnam(char *name)
{
.....
}
struct vsstat *
search(char *name)
{
.....
}
```



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

RULE: define 'static' every "top level" object, unless you want it to be shared

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

const

```
const float m=4.0;
const int *pci;
  /* pointer to const int
    *  Note : int * const pci;
    *  what is the difference ? */
m = 5.0; /*error */
pci = &a; /*legal*/
*pci = a; /error*/
```

- Can be used on function arguments

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

- Helps the compiler to identify mistakes
- Gives a lot of informations to optimizers

volatile

A **volatile** variable can be modified by the hardware or the O.S. , outside control of 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 standard 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 vsstat *
createnode( );
```

No information on arguments

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

FUNCTION DEFINITION

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;
}
```

Functions 21

```
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

Functions 22

```
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;
3. use these values to initialize formal arguments
4. henceforth formal arguments behave like other local variables

Functions 23

```
float called_func( int , float );

main(void){
  called_func ( 10.0/3.0, 2*3.5 );
}

float called_func (int iarg, float
  farg){
  float tmp=1.0;
  while (iarg --)tmp *= float;
  return tmp;
}
```

Functions 24

CALL BY VALUE :
a copy of the value of the actual argument is passed, not the actual argument itself
-> function <u>cannot</u> modify the actual arguments (unlike FORTRAN, Pascal **var** arguments)

```
int called_func( int a[], int n);

main(){
  int n=10, array[30];
  ......
  called_func ( array,10 );
}
called_func (int arr[],int 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( int *i, float x );

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


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

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

is identical to

```
void func(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(double x)
{.....}
double integrate(double (*f)(), double
  a, double b)
{/*integral of f(x) from a to b*/
  ...
}
main(void)
{
  double (*pf)(),s;
/* pf pointer to function returning
  double */
  pf = fun ; /* pf = &fun wrong ;
              * pf = fun() wrong ;
              * pf = &fun() wrong ;
              * /
  s=integrate(fun, -1.0, 1.0);
/* same as s=integrate(pf, -1.0. 1.0)
  */
.....
}
```

Structures are passed by value

More on Default Conversions

<u>If</u> <u>no</u> <u>function</u> <u>prototype</u> <u>used</u> (Old C form of declaration or no declaration at all)

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

WARNING : mixing a prototyped declaration with a non-prototyped definition 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

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, DOS, 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 descriptors" (small integers) to identify files (`open(filename)` returns a file descriptor, `read`, `write` require passing a file descriptor,

etc.) One field of the structure FILE identifying the C stream is the corresponding O.S. file descriptor.

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

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

OS calls (ioctl) must be done on file descriptor; 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.

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

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

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

DIRECT FILE MANIPULATION

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. .

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(char *file_name, char * access)

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

"rb", "r+b" etc.

Ex.

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

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( char *format,...)

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

int
sscanf ( char *in_string, 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 : arguments must be POINTERS to variables

format string

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

FORMATTED WRITE

```
int
printf ( char *format, ...)

int
fprintf ( FILE *stream, char
*format,...)
```



```
int sprintf(char *out_string, char
*format,...)
```

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

NOTE : arguments must be VALUES

OUTPUT FORMAT STRING

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

- refer to **stdin / stdout**

MORE GENERAL

```
int getc(FILE *fp)

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

special:

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

- return EOF if error (getc/putc/ungetc) or end-of file (getc);
- otherwise return the character read or written (as **unsigned char** converted to **int**)

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

Ex.:


```
#include <stdio.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))!=EOF){
   if ((c=putc( c , fp2 ))==EOF){
        fclose(fp1); fclose(fp2);
        return FAIL;
   }
 }
  if (ferror(fp1)) {
    fclose(fp1);fclose(fp2);
    return FAIL;
  }
  fclose (fp1);
  fclose (fp2);
  return SUCCESS;
}
```


- note cleanup in case of failure
  getc returns EOF at End of File or in case of error, **ferror** needed
  putc returns EOF in case of error

- 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(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 ( char * s, FILE *fp )
```

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

```
char *gets ( char *s )

int puts ( 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 non-negative if successful, EOF 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

```
#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 1 object of **nelem * size** bytes!!
- return number of elements transferred
  - if returned < **nelem**
    - on output, error
    - on input, EOF or error (**feof** to check);

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

```
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

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 [ BUFSIZ];

main(void){

  FILE *fp;
  /* declarations */

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

  .....
}
```

- **BUFSIZ** 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)

```
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

```
char c1[65536], c2[65536];
f1=fopen(...);
f2=fopen(...);
setvbuf(f1,c1,_IOFBF,65536);
setvbuf(f2,c2,_IOFBF,65536);
while((c=getc(f1))!=EOF &&
  (c=putc(f2))!=EOF));
```

...

```
f1=fopen("mydevice","wb");
setbuf(f1,NULL);
....
fputc(C,f1);
```

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);
```

or, better

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

a) Preprocessor operator #: converts its argument to a string
b) "a " "**string**" for C is the same as "**a string**" (constant strings written one after the other are treated as a single string);

Name generation operator

```
#define Genericswapdef(type) \
/* type must be a type name (basic\
 * type or typedef */\
void type##swap(type *a, type *b)\
{type temp;\
temp=*a;\
*a=*b;\
*b=temp;\
}
#define Gswap(a,b,type)\
type##swap(&a,&b);
....
typedef char * charptr;
Genericswapdef(int);
Genericswapdef(charptr);
charptr p1,p2;
int alfa,beta;
...
Gswap(&alfa, &beta,int);
Gswap(&p1,&p2,charptr);
```

Which then is translated into

```
....
typedef char * charptr;
void intswap(int *a, int *b){int
  temp; temp=*a; *a=*b; *b=temp;}
void charptrswap(charptr *a, charptr
  *b){charptr temp; temp=*a; *a=*b;
  *b=temp;}

char *p1,p2;
int alfa,beta;
...
intswap(&alfa, &beta);
charptrswap(&p1,&p2);
```


WARNING

```
#define FMAC(A,b) a*b
....
a=FMAC ( p+q, 1+m);
```

becomes

```
a=p + q * 1 + 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

Useful gcc extension:

```
static inline int
inc (int *a)
{(*a)++;}
```

or better


```
__inline__ static int
inc (int *a)
{(*a)++}
```


#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
#elif 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.

defined(x) returns 1 if x is defined, else 0
 #ifdef a <=> #if defined(a)

WARNING

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

would fail (#if defined NULL would succeed!)

#error diagnostic message

causes the preprocessor to abort

```
#if wordsize==4
...
#elif wordsize==2
...
#elif wordsize==8
....
#else
#error "wordsize is strange"
#endif
```

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** do nothing;
else
 print text of expression (**a>b** in this case)
 , file name, line number
 abort
If NDEBUG defined, expression not evaluated and test not 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, like SIG_IGN, are pointers to such functions: they are declared:
void (*SIG_DFL)(int);

Handlers are associated to signals by a call to to the library function **signal**


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

signal returns a pointer to the old handler

```
#include <signal.h>
..
void int_handler(int sig_number);
{
  void (*old)(int);/* pointer to a
  function */
  ...
  /* install handler only if signal
  was 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 (*HANDLER)(int);
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**
- -- or exit the program **(exit());**

Example:

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
void keyboard_intr(int);/*handlers*/

int number_of_cycles;
main(void)
{
  (void)signal(SIGINT, keyboard_intr);

  while (....){
   ....;
   number_of_cycles++;
  }
}
void keyboard_intr(int i){
  signal (SIGINT, keyboard_intr);
  printf(" Really quit?(y/n)");
  switch (getchar()) {
   case 'Y' : case 'y' :
       printf(
   "interrupted after %d
  cycles\n",number_of_cycles);
        exit(0);
   default: printf("continue\n");
  return;
  }
}
```

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


```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
void float_err(int);
main(void){
int a,b;
(void)signal(SIGFPE,fpe_error);
while(scanf("%d %d",&a,&b)==2)
   printf("%4.1f\n",(float)a/b*100);
exit (0);
}
void fpe_error(int i){
(void)signal(SIGFPE,fpe_error);
printf("No samples\n");
return;
}
```

Likely wrong: execution resumes from a/b, causes exception again
Solution 1: let handler do clean_up

```
void fpe_error(int i){
(void)signal(SIGFPE,fpe_error);
printf("No samples\n");
a=0; b=1;
return;
}
```

Wrong: a,b local to main. Must be to global;
Unsafe: at the point of error, b could be already in a register, modifying its value in memory pointless;
Ad-hoc: generally handler does not know where problem happens->cannot clean_up.


```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
void keyboard_intr(int);/*handler*/

main(void)
{
  (void)signal(SIGINT, keyboard_intr);

  while (scanf("%d %d",&a,&b)==2){
   ....;
  }
}
void keyboard_intr(int i){
  signal (SIGINT, keyboard_intr);
  printf(" Really quit?(y/n)");
  switch (getchar()) {
   case 'Y' : case 'y' :
        exit(0);
   default: printf("continue\n");
  return;
  }
}
```

What if signal arrive during scanf?
- **scanf returns EOF (MANY implementations) (OS problem: interruptible system calls return error code, library routines then...)**
- interference of getchar and scanf

- return from signal handlers dangerous, in particular if signal generated by error
- in these cases either exit or jump to safe place to clean up

In every case , if you plan to recover after a signal, be careful about interrupted system calls:
- Check which calls are interruptible and recovery possibilities (OS manuals)
- Check if SIG_IGN blocks the signal at the OS level (it should). In this case consider protecting critical sections by 'masking' signals

- **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(void){
  int v;
  ......
  if ( v=setjmp(env) )
   printf(" Coming from longjmp "
       " value = %d", v);
......
}
other_function(){
...
  longjmp(env, 1 );
....
}
```


Associated with signal handler

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <setjmp.h>
void fpe_err(int);/*handlers*/
int a,b;
jmp_buf env;
main(void)
{

  (void)signal(SIGFPE, float_err);
  setjmp(env);
  while(scanf("%d %d"),&a,&b)==2)
   printf("%4.2d\n", float(a)/b);
}
void fpe_error(int i){
(void)signal(SIGFPE,fpe_error);
printf("No samples\n");
longjmp(env,1);
}
```

FINAL EXAMPLE (Exercise)
Modify the calculator of Exercise 1. Insert two signal handlers, one for SIGFPE and one for SIGINT. Let both the signal handlers to try to recover the program (in case of interrupt, by asking the user as in previous example). Use **setjmp/lngjmp** to return to a safe place from the signal handler.
What is the difference with checking for 0 in the division? Is this solution more or less safe than the one with the check for 0? Is it more or less efficient?


```
#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)
{
  (void)signal(SIGFPE, float_err);
  (void)signal(SIGINT, keyboard_intr);
  (void)setjmp(env);

  while(PR,scanf("%f %c %f",&a, &opr,
  &b) ==3)
  {
   switch(opr){
       ......
   }
  }
}
void closing( char *message, int
  exit_code){
.....
}
```



```
void float_err(int i){
  (void)signal (SIGFPE, float_err);
  printf ("Floating point error\n");
  longjmp(env,1);
}
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);
  }
}
```