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 - 7

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

C PROGRAMMING LANGUAGE
(Part I)

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

TYPES AND DECLARATIONS

int i;
/*declares the *identifier* **i** to refer to a variable of type **int** ; creates (defines) the variable*/

All variables must be defined
All identifiers must be declared

An *identifier* can be up to 31 characters long(ANSI)
longer can be accepted, but maybe truncated
BUT GLOBAL SYMBOLS ...
contain letters,digits and _, starting with a letter or _
upper and lower-case letters are distinct

```
int a,A,VeryLongIdentifier;
char
  ATooLongIdentifierAsYouSeeUsedHere1,
   ATooLongIdentifierAsYouSeeUsedHere2;
        /* these could be considered
  the same
             by many compilers */
float x_3;
float 3x/*ILLEGAL!*/;
```


GLOBAL SYMBOLS : visible outside the file where they are defined
ANSI: are distinguished on the basis of their first 6 characters, case indipendent.WHY? Limits of system software-> not true on any UNIX or UNIX-like system

ELEMENTARY DATA TYPES

Data types: - set of values
- possible operations

Elementary data types provided by the language
Structured data types created by the programmer

C has very many to closely fit the hardware
-- possible portability problems
-- possible avoidance of portability problems (!)

- **void**
-*scalar types*
- *arithmetic types*
- *integral types*
- *floating types*
-*pointer types*
-*enumeration types*

---

void
- no values
--- to specify the type of a function that returns no value
void bye(...)
--- to specify a function without arguments
main (void)
--- to specify pointers to objects of any type

ARITHMETIC TYPES

Floating types

3 floating types (Old C : 2 only)
float
double
long double

NOTE: float : basic floating provided by hardware (32 bits almost everywhere, FORTRAN REAL)
double : at least the same precision and range than **float**, or better (REAL*8?)
long double : at least the same precision and range of **double**, or better (REAL*16)

- floating types are hardware: their behaviours and properties are implementation dependent (description in standard include file **<float.h>**)

- when mixing types in operations, obvious conversions:
-- **float a;**
double b;
... b*a
means:
convert **a** to double; perform product;
return double
and so on;
-- **a=b;**
means:
convert **b** to float; assign result to **a**

WARNING:
conversion of double to float can be impossible or an operation can yield a non representable result (Example: maxfloat+maxfloat) -> UNDEFINED BEHAVIOUR

FLOAT CONSTANTS

3.1 .33 3e2 5e-5 3.7e12
have type **double**
3.5f has type **float**
3.5e2L has type **long double**

ARITHMETIC TYPES

Integral types

char
short int or just **short**
long int or just **long**
int

each of the above can be modified by
signed or **unsigned**

char
- must contain (the numeric representation of) any character in the alphabet;
- must be at least 8 bits long.

Usually it is 8 bits long (but a chinese compiler could decide differently)

Excursus : the alphabet

ANSI defines the minimum alphabet of C
- Source alphabet (to write programs):
a-z A-Z 0-9 space tab form-feed newline
! " ' # % ^ () {} [] , . ; + - * / \ | ~ ? : < = > _ &
- Execution alphabet
Source alphabet + null alert(bell) backspace carriage return

Trigraphs:
Source alphabet is ASCII
ISO standard alphabet misses some characters
(national characters instead)

ANSI defines *trigraphs* to represent these missing characters in the source programs on computers not using full ANSI character set.
Ex. : **??=** can take the place of **#**
??(can take the place of **[**
trigraphs are a single character from any point of view, but only in source programs.

Again on **char:**

IT IS A (small) INTEGER!

8 bits, CAN be longer(implementation dependent)

signed char: range from -127 to 127 (?)
unsigned char: range from 0 to 256
char: whatever hardware prefers ("natural" representation) BUT(ANSI)
- guaranteed minimum range 0-127
- at least 8 bits
- value >0 if content is a character of the alphabet

CHARACTER CONSTANTS

'9' 'A'
'\n' '\t'
'\0','\107'
'*octal number**'**
'\x47'
'\xhexadecimal number**'**
'G' is '\107' is '\x47' if using ASCII

Because they are integers
9 == '9' - '0'; /*common trick, good only for ASCII machines*/
'a' == 'A' + 32; /* as above */
How to print a list of numeric values of letters?

```
#include <stdio.h>
main(void)
{ char c = 'a' ;
  while(c <= 'z'){
   printf("%c %d\n" , c , c) ;
   c = c+1 ;
}
```

Why **getchar()** is **int** and not **char** ?
(binary files)

---

int, short int **and** long int

short int : at least 2 bytes
long int : at least 4 bytes
int: "natural" hardware integer, at least 2 bytes

-short int used mainly for saving memory
-long int used mainly for range

---

unsigned **and** signed

unsigned short k;
unsigned long int j;
unsigned int l;

- Range from 0 to 2^{16}-1 or from 0 to 2^{32}-1
- Never overflows (arithmetic modulo 2^{16} or 2^{32})

WARNING : integer overflow gives undefined results!
Ex.:

```
short i = 256 ;
unsigned short n = 256;
n = n*n + 1;/* 256*256 is 0 mod 2^16 */
  /* n becomes 1*/
i  = i*i + 1 ;
  /* anything can happen*/
```

- Used for : -- exploiting all bits
  -- representing positive-only objects
  -- getting definite results with shifts
- Arithmetics can be slow

signed
useful only for **char** (could be same as **unsigned char**)

Integer constants

10 int, 10 decimal
50000 **long int** if int is 16 bits, else **int**, decimal
010 **int** 10 octal (8 decimal)
0x10 **int** 10 hex (16 decimal)
0x8000 **int** if int is 32 bits, else **unsigned int**
-1 **int** decimal
-035 **int** octal (-29 decimal)

RULE:
decimal take the type **int, long int** or **unsigned long int** (the smallest that fits)
hex and octal take the types **int, unsigned int, long int** or **unsigned long int** (the smallest that fits)

Explicit sizing:
10l **long int**
10u **unsigned int** (ANSI ONLY)

WARNING: DON'T BE TOO CLEVER

-1 is represented by 0xffff (16 bits)
BUT

```
short int n;
n=0xffff; /* is not -1*/
```

0xffff is a positive constant;
of type **unsigned int** (does not fit into an **int**)

assigned to an **int** -> **UNDEFINED**

EXPLICIT TYPE CONVERSION (CASTING)

(scalar type) expression

Ex:

```
float f = 3.5 ;
int i , n = 2 , m = 3 ;

i = (int)f;      /* i=3 */
f = m/n;     /* f=1.0 */
f = (float)m/n; /* f=1.5 */
```


MIXING TYPES IN ARITHMETICS AND ASSIGNMENT

```
int m , n ;
float a , b ;
char c ;
short q ;

a = m + n/a - b + c*q*b ;
```

Most reasonable:
short types converted to default
(short, char to **int)**

if operators involves different types,
convert to "most powerful"
(int + long, convert to **long ,** return **long)**
(int + float, convert to **float,** return **float)**
(int + unsigned, :convert to **unsigned,** return **unsigned)**

Convert result to the type of the variable
on the left of =, and assign.

IMPORTANT WARNING

```
int m = 256;
long int n ;

n = m*m ;/* UNDEFINITE RESULT if int is
  16 bits */
```

m*m is computed as **int**, but result overflows
result is converted to **long int**, but too late

```
n = (long)m*m ; /*works*/
```

WARNING: mixing unsigned and signed

```
unsigned int n=10;
int m;

m = n-15; /* 2^16-5*/
```

ENUMERATED TYPES

Like in Pascal: a set of names, holding costant values assigned by the compiler

```
enum { red, blue, gree, yellow }
  LightColor,PaperColor;/*LightColor,
  PaperColor are variables of an
  "anonymous" enumerated type */

enum Brightness {bright, medium, dark};
enum Brightness
  BulbIntensity,ScreenIntensity;
/*BulbIntensity, ScreenIntensity are
  variables of type "enum Brightness":
  Brightness is a type tag */

LightColor = red;
BulbIntensity = bright;
```

enum { *constant_identifiers* }
or
enum *tag* { *constant_identifiers*}

- constant identifiers bring integer values from 0 on
- a good compiler would issue a warning but never an error for a conflicting enum:


```
/* the following should cause a warning
  message */
  BulbBrightness=red;
  PaperColor=2;
```

- there is nothing specific to **enum** to go from one value to the next; adding 1 works (poor!);

```
enum day {sun, mon, tue, wed, thu, fri,
  sat };
enum day d;

for(d=sun; d<=sat; d=d+1)...
```

- **enum** can be used to give names to arbitrary integer constants, as follows:

```
enum{ Minimum=-12, DangerLow, Hot=98,
  Maximum=100} status;
/* DangerLow becomes -11*/
```

USAGE:
- Give names to constants:
  essential to improve readability. Important !
- To check against unreasonable type mixing

Old C style: use

#define MINIMUM (-12)

- Ok for readability
- No protection for type mixing (all integers!)
- Valid for a whole file (SCOPE problem)

POINTER TYPES

C uses extensively pointers : ESSENTIAL TO USE THE LANGUAGE

Hardware view of pointers

variables are memory cells
each one has an *address*
this address is some kind of integer
I can store the address of a variable in another variable
this one becomes a *pointer* to the first one

Example:

variable **i** IS memory cell 2347
contains the *value* 35
variable **j** IS memory cell 1398
j contains the *value* 2347

j is a pointer to **i**

C approach to pointers

Similar, BUT
- variables of different types use different "memory cells"
- addresses are not **int**
- addresses of objects of different types are of different types;

If **a** is a variable of type ***T***, **&a** is a pointer to it and has type "pointer to ***T***"

If **p** has type "pointer to ***T***" and is not null, ***p** is a variable of type ***T***

Example:


```
int i , j=15;
float a , b=1.4;
int *pi1 , *pi2; /*pi1,pi2 are pointers
  to int */
float *pf1;      /* pf1 is a pointer to
  a float */

pf1 = &a;     /* store the address of a
  in pi1 */
*pf1 = b;     /* is the same as a=b */
pf1 =b;       /* illegal */
b = *pf1 -0.25;/* is the same as b=a-
  0.25 */
pf1 = &b;
b = *pf1 - 0.25;/* is the same as b=b-
  0.25  WHY*/
pi1 = pf1;    /* is illegal : type
  mismatch */
pi1 = & i;
pi2 = pi1;
*pi2 = j;     /* is the same as i=j */
```


NULL POINTERS

A pointer containing a zero value does not point to anything.

```
int *p;
p = 0; /* ugly but legal */
p = (int *)0; /*better */
```

- *&anything* never returns 0;
- memory allocation facilities never return 0;

TYPELESS POINTERS

```
void * malloc(int size);
float * fptr;
fptr= malloc(sizeof(float));
```

- Every pointer can be assigned to a void pointer and reassigned from it and it will not be modified

```
void *ptr; float *fp1,fp2; double *dp1;
ptr=fp1;fp2=ptr;/* same as fp2=fp1 */
dp1=(double *)fp1;fp2=(float *)dp1;
  /*legal, but result dubious */
```

- can be dereferenced only after casting

```
void *ptr;
float a,b;
ptr=&a;/*ok*/
b=*ptr;/* illegal: *ptr has no type
  (type void) */
b=*(float*)ptr;/* ok*/
```

CONSTANT POINTERS

On some machines:

(char *)100 points to "memory position" 100

NOT STANDARD
require byte-addressed memory
(int *) would not work

POINTERS TO FUNCTIONS

```
float integrate(float (*f)(),float a,
  float b, float eps)
{
  ....
  y=f(x); /*yessir!*/
  ...
}
float sq(float x){return x*x;}
main(void)
{printf("%f\n",integrate(sq,-1.0,1.0))}
```

- a function cannot have another functions as arguments: it can have as an argument an address of a function:

float (*f)() means

- **(*f)(*args*)** is a float
- ***f** id a function returning a float
- ***f** is a pointer to a function returning a float
- a function can return a pointer to another function

- a pointer to a function is a legal variable

```
float (*pf)() ,sqi,sqc;/* pf is pointer
  to function returning float */
pf=sq;/* let's sq(x) return x*x */
sqi=integrate(pf,-1.0,1.0);
pf=mycos;/* another function returning
  cos(x)*/
sqc=integrate(pf,-1.0,1.0);
```

- **every** reference to a function name, except when it is being defined or declared, is intepreted as a pointer to the function; therefore when one calls **sq** - like **y=sq(a)**; - , **sq** is intepreted as a pointer to the function labeled by **sq**.

This is the reason of **integrate (sq,-1.0,1.0)** instead of **integrate (&sq,...)** and of **pf=sq;** instead of **pf=&sq;**, and of **y=f(x);** in the body of **integrate**

Possible use: a table of functions, selected on the basis of an input value:


```
float f1(float a, float b){...}
float f2(float a, float b){...}
float f3(float a, float b){...}
float f4(float a, float b){...}
main(void){
float (*(tabf[4]))()={f1,f2,f3,f4};
float z;
int n;
...
scanf("%d",n);
z=tabf[n](arg1,arg2);
...
}
```

- possible use: a 'compound variable' that includes the pointers to the functions needed for its own manipulation -> at the origin of OO programming

COMMENT : declarations

C declarations are "by example".

int *p;

means :

"***p** is an **int**", therefore "**p** is a pointer to an **int**"

For this reason:

```
int *pi1, *pi2, i, j; /* means i,j are
  int, pi1, pi2 are pointers to ints */
```

WARNING: common error:
int * p1, p2,p3;

USER DEFINED TYPES

typedef

```
/* type definitions*/
typedef unsigned int size ;
typedef int * P_to_i ;
typedef size * P_to_s ;
/* variable definitions */
size array_size, i;
P_to_i pi;
P_to_s ps;
```

typedef *declaration type-name*

afterwards ,*type-name* can be used as any predefined type

Can be used to parametrize programs:
size could be **unsigned long int** on 16 bits machines

Almost essential for complex type declarations (structured types)

IMPORTANT WARNING : difference with #define

```
#define PI int *
PI pi1,pi2;
```


expands to

```
int * pi1, pi2;
```

pi1 is pointer to int, pi2 is int !!!!!

BUT

```
typedef int* P_i;
P_i pi1,pi2; /* both are pointers to
  integers */
```

OPERATORS

- C has many
- C treats as operators things that other languages do not
- Contribute significantly to the complexity of the language

ALREADY MET

Arithmetic operators

+ - * /

- apply to arithmetic types; if types mismatch, arithmetic conversions
- actually, + - apply also to pointers (later);

%

- applies to integral types only

Meaning is obvious, except one WARNING

```
int n = -10 , m = 3 , p , q ;

p = m/n ;
q = n%m;
```

can yield either
p=-3 q=-1
or
p=-4 q=2

Relational operators

< > >= <= == !=

- apply to all scalar types
- operands must be of the same type
- return **int** 1 (true) or 0 (false); no Boolean-LOGICAL type in C!
- testing pointers for greater-less meaningful only if pointers to different elements of the same array or structure
- comparing pointers for equality with 0 legal (ugly: use cast)

- warning : written without intermediate spaces (== not = =)
- **WARNING: careful about checking floating types for strict equality:**


```
float a;

a=1.0/3.0;
if(1.0 == 3.0*a) ../*usually FALSE */
```

Address operator

& *identifier*

- applies to any variable except *bit-fields* and **register** variables
- returns a pointer to its operand

Dereferencing operator

***** *pointer*

Comma operator

expr1 , expr2

- applies to expressions of any type
- evaluates both its operands, and returns the value of expr2 (expr1 evaluated only for side effects)
- operands are evaluated in order (first operand evaluated first!)
- WARNING : this is NOT the comma that appears in function calls


```
int f(int *n1, int *n2);
...
f(&n1 , &n2) /*  is not the comma
  operator*/
f((n1=1 , &n1) , &n2) /*the first ,
  is an operator*/
```

GENERALITIES ON OPERATORS

1)Precedence Level

a + b / c is **a + (b/c)**

a < b - c is **a < (b-c)**

2)Associativity

if same level of precedence

2.0 / 3.0 / 4.0 is **(2.0 / 3.0) / 4.0**
LEFT ASSOCIATIVE

a = b = c is **a = (b = c)**
RIGHT ASSOCIATIVE

Highest precedence (precedence level **1**)

Postfix operators

Associativity: left to right

```
a[2].p is (a[2]).p
```

- **Array reference** []
-- if **a** is an array, **a[1]** is an element of it

- **Function call** ()
-- if **f** is a function, **f(...)** calls it and returns its value

- **Component selection** . ->
-- discussed with structures

Precedence level 2
Unary operators

Associativity: right to left

```
-*a is -(*a)
```

- **Address operator** &
- **Dereference operator** *

- **Minus** -
applies to arithmetic operands
changes the sign of its operand

- **Plus** + ANSI only

applies to arithmetic operands
forces immediate evaluation of its operand

--- mathematically, **a+(b+c) == (a+b)+c**
--- in these cases, associativity rules do not apply
--- compiler authorized to reorganize even removing unneeded parentheses
--- + can be used to force an order of evaluation
+(a + b) + c
or
a + +(b + c)

- **Logical negation** !
any scalar operand
returns **int** 1 if operand is 0, else returns 0

- **Bitwise complement** ~
any integral operand
returns bit complement of the operand

- **Increment and decrement operators** ++ --
VERY MUCH USED
HAVE SIDE EFFECTS
apply to any scalar variable (NOT TO AN EXPRESSION!)

m++ returns the current value of **m** AND
modifies **m** adding 1 to it
++m modifies the value of **m** adding 1 to it
returns the modified value

-- Example

```
int m , n = 3 ;
m = n++ ; /* is like m=n; n=n+1; */
m = ++n ; /* is like n=n+1; m=n; */
```

– WARNING

```
int m = 2;
m + m++ /*UNDEFINED :4 or 5*/
```

: NEVER use twice in the same expression
a variable subject to side effects of an operator

-m++ /* right associativity: means -(m++) */

–m++ /* illegal (why?)*/

- **sizeof** operator

applies to a name or expression of any type
returns the size of its argument in bytes
has two forms : operator and function


```
double f;

sizeof f /*applied to variable or
  expression*/
sizeof (double) /* applied to a type
  */
```

ESSENTIAL when dealing with dynamic objects (memory management)

Level 3
Associativity: right to left

- **Cast operator** (type)

```
if (p==(char *)0)...
f=(float)m/(float)n;
```

Level 4,5
Arithmetic operators
Associativity : left to right

Level 4: multiplicative operators * / %
Level 5: additive operators + -

Level 6
Shift operators
Associativity: left to right

- Left shift <<

- apply to integral operands
- right operand must be >=0 and <= number of bits of the left operand
- filling with 0

```
int i = 0x3 ;
i = i<<4 ;/* is 0x30 */
```

- Right shift >>

- as above, BUT
- if left operand is **unsigned** or >=0, zero filling;
- else, implementation dependent (zero or sign extension)

```
int m,n;/* int = 2 bytes */
m=715;/*binary 0000 0010 1100 1011*/
n=m>>2; /* n=715/4=178 */
m=-1;/* binary 1111 1111 1111 1111 */
n=m>>1 /* implementation dependent:
 -1 or 2^15 -1*/
```

DO NOT USE IN PLACE OF * , /

Levels 7,8
Relational operators

Associativity : left to right

Level 7 Comparison > < >= <=
Level 8 Equality == !=

Levels 9,10,11
Bitwise operators

Associativity: left to right

Apply to integral operands

Level 9	Bitwise and	&
Level 10	Bitwise exclusive or	^
Level 11	Bitwise inclusive or	|

Ex:

```
short int i1=0x0180 , mask=0x00c0;
short int masked_i1;
masked_i1 = i1 & mask  | 3; /*
 masked_i1 0x0083  */
```

Levels 12,13
Logical operators

Apply to scalar operands
Evaluate second operand only if needed

Level 12
- Logical and &&
returns 1 if both operands non-zero
EXACT:
if *op1* is 0 return 0
else if *op2* is 0 return 0
else return 1

Level 13
-Logical or ||
returns 1 if one operand non-0 else return 0
EXACT:
if *op1* non 0 return 1
else if *op2* non 0 return 1
else return 0

Example:

```
int *p;
...
if (p && (*p = getchar()) && *p !=
  EOF)...
```

getchar called only if **p** non NULL (non 0), otherwise disaster!
check for **EOF** only after **getchar** called, otherwise meaningless

The ONLY operators apart from comma with guaranteed order of evaluation;
therefore
the ONLY case, together with comma, where repeated use of variable affected by side effects is safe

```
int *p;
...
if (p && (*p = getchar(),*p !=EOF))
```

Note this one is probably right, the previous one is probably wrong. Even better, as we will see:

```
if (p && (*p=getchar())!=EOF)
```


Level 14
Associativity right to left
- Conditional Operator ? :

expr1 **?** *expr2* **:***expr3*
evaluates *expr1;*
if *expr1* is not 0, evaluate *expr2* and return its value;
else evaluate *expr3* and return its value

expr1 must be a scalar
expr2 and *expr3* must have compatible types

Example
max = x>y ? x : y ;

Level 15
Assignment operators

Associativity: right to left

WHY OPERATOR?

variable = *expression*

evaluate *expression*
convert its value to the type of *variable*
assign to *variable*
return the value assigned to *variable*

HAS SIDE EFFECTS

```
int p;
float a=3.5,b;

b=p=a;    /* means b=(p=a)
      p=3
      b=3.0
  */
```

```
int p1, status(FILE *fp);
if (p1=status(fp))printf("error
  %d\n",p1);
```


Compound assignment operators

+= -= *= /= %= <<= >>= &= ^= |=

binaryoperator=

```
int j,k;

j += k;   /* same as j = j+k */
```

Same operands as corresponding binary operator

VERY USEFUL (safer than simple assignment)

```
if (p && (*p = getchar())!=EOF)...
```

means:

if p not zero
then call getchar, put its value in *p,
then check the value assigned to *p
for equality with EOF, if different
then...

COMMENT: relation with assignment statement

Most language have assignement statement
In C, statement is any expression followed by ;

```
a=b; /*assignment statement */
```

From Example 1

```
while (( c=getchar()) != EOF)
```

c=getchar() assignment expression
modifies **c**
returns the value of **c**

(c=getchar()) parentheses needed because
precedence of = lower than
precedence of !=

(c=getchar())!=EOF relational expression,
evaluates to **1** or **0**
leaving useful value in **c**

WARNING : not exactly same as simple assignment: better!

Left operand evaluated only once

```
a[i++] = 3; /* a[i] is element i of
  array a */
```

means

```
a[i] = 3;
i = i+1; /*in this order */
```

```
a[i++] += 3;
```

means

```
a[i] = a[i]+3;
i = i+1; /* in this order */
```

BUT

```
a[i++] = a[i++] + 3;
```

is undefined (two occurrences of **i++** in the same expression)

Level 16

- Comma operator

CONCLUSION

great power at your fingertips
easy to make mistakes

- relational and logical operators returning integers
  if(a&b == c) /*legal : means a&(b==c) */
- side effects: use sparingly
  never use twice in an expression a variable affected by side effects, except if expression is "logical" one (&&, ||)
  b + (b=c) /*undefined:
  not even b + +(b=c) */
  i-- && a[i]=b /* OK because && */
  multiple unary operators
- precedence problems : use manuals and parentheses

STATEMENTS

SIMPLE STATEMENTS

expression ;

means:
evaluate *expression*
discard the result
USEFUL ONLY FOR SIDE EFFECTS

```
...
int i,j;

i=j;
i++;
i-j; /* legal but useless : no side
  effects*/
func(i,j); /* legal, even if func
  returns a value; useful or not ? */
(void)func(i,j);/* equivalent, but
  better programming style: why?*/
```

EMPTY STATEMENT

;

Ex:

```
if (i == j)
  ;
else
  j++;
```

GREAT OPPORTUNITY FOR MISTAKES
while(condition) ; {do something}
if(condition);{do something}

COMPOUND STATEMENTS

{
definitions and declarations;
statements ;
}


```
reverse(float a[], int n)
{
  int i=0,j=n-1;

  while ( i<j)
  {
  [illegible]
  }
}
/* NOTE : compound statements
 * terminated by
 *     }
 * not by ;
 */
```


- Compound statements can nest;

```
{ int i=1,j=n-1;
  while( i<n/2 )
  {
   float temp;
   temp = a[i];
   a[i] = a[j];
   a[j] = temp;
  }
}
```

- variables defined in a compound statement hide definitions of variables of same name outside;

```
{ int i,j;
  i=1;
  j=2;
  { float i; /* i redefined */
   i=3.0;
   printf ("%f %d\n",i,j);
   /* i from inner, j from outer
  definition */
  }
  printf ("%d,\n", i);
  /* here inner i is no longer
  existent */
}
```

- Functions bodies are a single compound statement

```
main(void)
{
......
}
```

COMMENT: definitions in compound statements should be used to keep variables definitions close to the place where they are used: readability

---

FLOW CONTROL

- conditional (2 statements)
- loops (2.5 statements)
- transfer of control (3 statements)

THE if STATEMENT

if (*expression*) *statement 1*

if (*expression*) *statement 1*
else *statement 2*

expression : must be of any scalar type.
If non 0,*statement 1* is executed
If 0, *statement 2* , or nothing if **else** missing

statement 1 and *statement 2* must be
one single statement
possibly a compound one

```
if( a == b ) j = 1;
if( a == b ){j=1;k=n;} /*note ;} */
```


WARNING

```
int k = 0, j =1;
float a =1.0 ;

if (k)
  if (j) a = 3.0 ;
else
  a = 2.0 ;
```

WHAT IS THE VALUE OF **a** ?

associativity:

if (*e1*) **if** (*e2*) *s1* **else** *s2*
means
if(*e1*) { **if** (*e2*) *s1* **else** *s2*} ***YES***
or
if (*e1*) { **if** (*e2*) *s1*} **else** *s2* *NO*

Do not trust indenting

USE COMPOUND STATEMENTS EVEN IF NOT NEEDED, TO BE SURE;

```
int k = 0, j =1;
float a =1.0 ;

if (k){
  if (j) a = 3.0 ;
}else{
  a = 2.0 ;
}/* no doubts! */
```

USEFUL COROLLARY

a specific **elsif** not needed

if (*expr1*) {
    statement1;
}else if (*expr2*) {
    statement 2;
}else {
    statement 3;
}

means exactly as intended:

if (*expr1*){
statement 1;
} else {
    if (*expr2*) {
        statement2;
    } else {
        statement3;
    }
}

THE switch **STATEMENT AND** break

switch (*expression*) {
    case *constant*$_1$:*statements*$_1$;
    case *constant*$_2$:*statements*$_2$;
    .....
    default : *statements* ;
}

- *constant labels* must be constant (known to the compiler)
- expression must be of any scalar type;
- execution jumps to the label whose constant value is equal to *expression,* or to **default** if none matches;
- if there is no **default** and *expression* does not match any label, nothing happens (poor style);
- execution does NOT end at the next label, but continues to the end;
- the **break** statement interrupts the flow of execution and jumps to the end of the **switch.**

Normal way of ending a case

WARNING: flow from one case to the other is dangerous. Should be used ONLY when many cases require the same action

switch (*expression*) {
    case *constant*$_1$:
    case *constant*$_2$:*statements*$_1$;**break;**
    case *constant*$_3$:*statements*$_2$;**break** ;
    .....
    default : *statements* ;
}

Ex.:

```
{/* convert a string to a decimal,
  stopping at first non-digit */
int num=0,c;
while((c=getchar())!=EOF)switch (c){
  case '0': case '1': case '2': case
  '3' : case'4': case '5': case '6';
  case '7'; case'8': case '9':
   num=10*num+c-'0';
   break;
default: ;
}
}
/* NOTE: very poor c; use "if"! */
```


THE while **AND** do **STATEMENTS**

while (***expr***) ***statement***

means:
    loop:
        evaluate *expr*
        if non 0 perform *statement*
        goto loop

- *expr* any scalar
- *statement* a single (possibly compound) statement

WARNING : common mistake
while (*expr*); statement;

---

do ***statement*** while (***expr***);
/* please note the final ; */

Like while, but *statement* executed before testing
USUALLY NOT NEEDED

Example:

```
char buf[500], *p=buf;
do *p++=getchar();
while(*p!='\n'&&*p!=EOF);
```


THE for **STATEMENT**

```
/* read 10 elements from input and
  copy them on output, summing them in
  the meantime: stupid problem with
  stupid solution
 */
main(void)
{
  int i, n , s ;

  for(i=0 , s=0 ; i<10 ; i++ , s+=n) {
   scanf("%d",&n);
   printf("%d + \n",n);
  }
  printf ("_____\n = %d\n",s);
}
```

In general
for (*expr1*; *expr2* ; *expr3*) *statement*

means (almost)

expr1 ; **/* evaluate as statement */**
while (*expr2*) {
    statement
    expr3 ;
}

Not like FORTRAN DO or Pascal for (fixed number of iterations with constant increment of the loop control variable)

for (i =*init*; i <= *end* ; i += *incr*)
    is same as FORTRAN
DO *label* **I=*init,end,incr***

WHY NOT USING **while?**
    Concentrates in a single place all the loop control information.

```
/* this function computes the
  factorial of  an integer; it uses
  "for" as a FORTRAN DO */
long int factorial(int val)
{
  int j, fact=1;

  for(j=2 ; j<=val ; j++)
   fact *= j;
  return fact;
}
```

BUT ALSO

```
/* this function reads a string of
  digits and  converts them to an
  integer. Stops at first non-digit
It uses the library function
  "isdigit", defined in the standard
  include file "ctype.h"
*/
#include <stdio.h>
#include <ctype.h>

int read_int(void)
{
  int num = 0 , d;
  for(d = getchar() ; d != EOF &&
  isdigit(d) ;d = getchar() ){
       num *= 10;
       num += d -'0';
  }
  return num;
}
```

FINAL REMARK
equivalence with **while** broken only in the following case

for (; ;)

means

while (1) /* while() would be incorrect */

Both used for infinite loops

COMMENT
each of the statements discussed above is a single statement: therefore:

```
for(...;....;...)
  while(....)
   if (...)
      a=b;
   else {
      b=c; d=e;
   }
```

BUT DANGEROUS: what if you add a statement before the above **if ?**

Usage of **{}** recommended for clarity and robustness if depending statement is complex.

```
for(...;....;...){
  while(....){
   if (...) {
      a=b;
   }else {
      b=c; d=e;
   }
  }
}
```


TRANSFER OF CONTROL

Theoreticians say : don't use it (PASCAL)

Dangerous
To be used only in anomalous situations (leave processing in case of error)

C needs it also to jump out of **switch** cases

CONTROLLED JUMPS : break **AND** continue

break;

already met. Jumps outside the surrounding **switch** or **for** or **while**

for (*expr1* ; *expr2* ; *expr3*){
.....
if (*error condition*) **break;**
.....
}

WARNING : exits from the innermost only

```
float a[100][100];
int i,j;
for(i=0;i<100;i++)
for(j=0;j<100;j++){
  if(a[i][j] >= 0.0){
   a[i][j]=sqrt(a[i][j]);
  }else {
   /* stop with the sqrt and print an
  error message : how? */
  }
}
```

continue;

```
for (i=-10; i<=10; i++){
  statement1;
  if (i==0) continue;
  statement2;/* skipped if i==0 */
}
```

When executed, jumps to the end of the surrounding **for** or **while,** and starts next iteration

```
/* read lines, skipping comments,
  that is lines starting with '#'
Uses the routine function gets,
  defined in "stdio.h", which copies
  an input line in a buffer, returning
  a pointer to the buffer or a null
  pointer if it hits the end of file
Uses NULL, defined as a 0 pointer in
  "stddef.h"
*/
.........
char buf[500];
char *p;
while((p=gets(buf))!=NULL){
  if(buf[0]=='#') continue;
  /* Instead of *p, buf[0] would work
  */
  /* start processing the line */
  ......
}
```

FINAL EXAMPLE

```
#include <stddef.h>
#include <stdio.h>
main(void){
char buf[500];
char *p;
while(1){
  p=gets(buf);
  if(p==NULL)break;
  if(*p=='#')continue;/* p==&buf[0]*/
  /* start processing */
......
}
}
```


labels: **any string followed by ":"**

- do not need to be pre-declared
- must be part of a statement (possibly empty)
  at end of compound statements
  label_at end : ; /* ; required*/
  }/* end of compound statement */
- visible only from inside the function where they are used

UNCONTROLLED JUMPS :

goto
label:

```
#include <stdio.h>
#include <math.h>
main(void)
{
float a[100][100];
/* fill a */
......
/* take square roots */
for (i=0;i<100;i++)
  for(j=0;j<100;j++){
   if(a[i][j] < 0) goto error;
   a[i][j] = sqrt (a[i][j]);
  }
/* here the rest of the program */
......
exit (0); /* normal end */
/* error handling area */
error: printf("%s %d %d", "a negative
  at", i,j);
exit (1);
}
```

break would not work because exiting from 2 loops

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*

```
int 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 (of course!)

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

IMPORTANT COMMENT
arrays of anything allowed (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 };
```


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

```
short s[10] , *ps;
double d[10] , *pd;
char c[10] , *pc;
ps = &s[0] ; pd = &d[0] ; pc = &c[0];
ps++ ; pd++ ; pc++ ;
  /* now:
   ps == &s[1]
   pc == &c[1]
   pd == &d[1] */
```

```
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" */
ar ++ ;  /* illegal: array names are
  "constant pointers "*/
p = &ar; /* illegal: ar is already a
  pointer to the array ; but all the
  compilers would understand and issue
  a warning only */
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;
float f_array[30], *pf=f_array;
p1 = &ar[10]; p2 = &ar[15];
q   = &br[5];

if(p1 != 0) printf("p1!=0\n");
 /*legal,true*/

if(p1 < p2)printf("p1<p2\n"); /*legal,
 true*/

if(p1+5 == p2) printf("p1+5==p2\n"); /*
 pointer+ int legal , yields pointer*/

if(p2-5 == p1) printf("p2-5==p1\n"); /*
 pointer - int legal, yields pointer */

printf("%d\n",p2-p1) ;
 /* pointer - pointer  legal, yields
 long*/

p1++;
 /*legal; now p1 points to ar[11] */

if(q!=p1) printf("q!=p1\n") ;
 /* legal, true*/

if(p1<q)printf("p1<q\n");
 /* result undefined */

q=p1- q;
/*result undefined*/
```

On the other hand

```
p1+p2          /* illegal */
2*p1           /* illegal */
q != 100       /* illegal */
p1==pf         /*illegal */
```

- No operations between pointers to different types (**p1** and **p2**)
- **< > <= >=** - meaningful only between pointers to different element of same array or structure
- tests for equality allowed for arbitrary pointers to the same type
- comparison with **int** 0 allowed for every pointer (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. (Fourth 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 *)** most likely 4

Not even if **ar** declared with a size

```
int sum_of_elem(int ar[100])
{ sizeof ar ....}
```

STYLE COMMENT

C ugly style

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

 for (pend=ar+n_elem; ar<pend; s+=
 *ar++) ;
 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 n_of_col){
  .... *( (int*)a + i*n_of_col + 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;
           /* exchange a[i] and a[i+1]
  */
           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;
           /* exchange *pj and *pj+1 */
           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'

Also:
"this" "is" "a single string"
compiler chains string constants
(used with preprocessor and # preprocessor operator)

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

Defining a string variable

```
char str1[10];
char str2[] = "string" ;
/* compiler makes str2  with 7
  elements , 6 of 's' 't' 'r' 'i' 'n'
  'g' + 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" (in system private area, like all constants)
- creates a character pointer variable(s)
- initializes s with the address of the constant

DIFFERENCE

```
str1 = p; /* illegal: str1 is array,
  i.e. constant pointer */
s = p; /* legal; the constant string
  attached to s in initialization is
  lost*/
```


STRING ASSIGNMENT

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

```
char carray1[10], carray2[10];

carray1 = "not ok" ; /*illegal:
  cannot assign to array name */
carray1[1] = 'a';   /* OK */
carray1[2] = '\0'; /*now carray1
  contains "a" */
```

```
carray2 = carray1; /* illegal */
```

```
{ register int i;
for(i=0;i<10&&carray1[i]!='\0';i++)
  carray2[i] = carray1[i];
if (i<10) carray2[i]='\0';
}
/* this is probably what you meant ,
  but there are better ways*/
```



```
char*p1="A string"; char *p2;
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"       /* illegal, 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;
}
```

COPYING STRINGS

```
#include <string.h>
char st1[20] , st2[]="wow!";

st1 = st2 ;/* illegal */

strcpy(st1 , st2) ; /* library
  function only way */
```

.....

```
char *
strcpy(char s1[],char s2[])
{
  register int i;
  for( i=0 ; s2[i] ; i++)
   s1[i] = s2[i];
  s1[i] = '\0';
  return s1;
}
```

```
char *
strcpy(char *s1, char *s2 )
{
  while ( *s2 ) *s1++ = *s2++;
  *s1 = '\0';
  return s1;
}
```

- **while (*s2)** means **while (*s2 != 0)**
- ***s2++** unary operators are right associative therefore ***(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

BETTER

```
char *
strcpy( char *s1 ,char *s2 )
{
  while ( *s1++ = *s2++) ;
  return s1;
}
```


THE STRING LIBRARY

#include <string.h>
contains:
strcpy(s1,s2)
char *s1,*s2;/* copy s2 to s1 */

strncpy(s1,s2,n)
char *s1,*s2;
int n; /* copy at most n characters from string s2 in s1 ; if s1 too short, may be not null-terminated or cause run-time error*/

int strlen(s1)
char *s1; /* return length of s1 */

strcat(s1,s2)
char s1[], *s2; /* concatenate s2 to the end of s1 */
WARNING: s1 must have the room for cat
char *s1="alfa", *s2="beta;
strcat(s1,s2) ; /*illegal, would cause execution-time problems */

---

char a[100]="alfa", *s2="beta";
strcat(a,s2);/*OK*/

strncat(s1,s2,n)
char s1[], *s2;
int n; /* concatenate at most n characters from s2 to the end of s1 */

int strcmp(s1,s2)
char *s1,*s2;/* compare s1 and s2: 0 if equal, <0 if s1 < s2, >0 is s1 > s2 */
WARNING:
strcmp(s1,s2) returns TRUE (!=0) if **s1 !=s2** !!!
if(strcmp(name,"Johnny")) used for **if(name equal to "Johnny")** is a very common mistake

int strncmp(s1,s2,n)
char *s1,*s2;
int n; /* compare at most n characters from s1 and s2: 0 if equal, <0 if s1 < s2, >0 is s1 > s2 */

char *index(s,ch)
char *s, ch;

char * rindex(s,ch)
char *s, ch;
/* returns pointer to first (index) or last (rindex) occurrence of ch in s, or NULL */

```
char * string="This is a sentence";
printf("last word of string is %s\n",
  rindex(string, ' ')+1) ;
  /*dangerous!*/
```

safer:

```
  char *p, *string="This is a
  sentence";
printf ("last word of string is
  %s\n", (p=rindex(string,' '))? p+1:
  "");
```

MORE ELEGANT BUT NOT SAFE

```
#define Rindex(s,c) (rindex(s,c)?(rindex(s,c)):(s+strlen(s)))
#define Index(s,c) (index(s,c)?(index(s,c)):(s+strlen(s)))
```

THE CHARACTER LIBRARY

Just because related:
#include <ctype.h>
contains definitions of macros
isdigit(c)
isalpha(c)
isalnum(c)
ispunct(c) /* not alnum not cntl */
iscntrl(c) /* '\0' to '\32' and '\127' */
isspace(c)/* space tab newline CR FF */
islower(c)
isupper(c)
isprint(c)