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

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

SMR/474 - 4

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

PROGRAMMING IN C LANGUAGE
(Lectures 1 to 4)

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

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

C Language:
- Very popular
- Widely used
- Hated by computer language theorists
--- as FORTRAN, BASIC...
--- unlike Algol, Pascal, Ada...

WHY?

- Complex syntax (lots of operators, symbols etc.)

- More serious: no strict rules: allows writing horrible things;
  allows playing with hardware;
  allows writing incredibly unreadable code
  (annual prize for most cryptic C program...);

REQUIRES PROGRAMMER'S DISCIPLINE

SOME HISTORY

Originated from Brian Kernighan and Dennis Ritchie, ATT Bell Labs, about 1972, as systems programming language-used to implement UNIX(TM) since then.
Their book(1977) the only reference for many years.
Slow expansion (**void** data type, **enum**)
Many minor variants.
ANSI standard : started 1983, ready from 1988, but many compilers not compliant are yet around (this course uses one).

THE **C** SYSTEM

1) macro processor ("C preprocessor")
2) language/compiler
3) standard include files
4) standard library

+

syntax checker ("lint")

THE STRUCTURE OF A C PROGRAM

Multiple source files

Each **file** is composed of **functions** (and other things...)

functions cannot be nested (unlike Pascal)

Subdivision in **files** is part of the language (unlike FORTRAN)

OVERVIEW OF C

```
/* Two penny calculator */
#include <stdio.h>
#define PROMPT putchar(':')

main(void)
{
	float a, b;
	char opr;
	float result;
	void bye(char message[], int exit_code);

	while( PROMPT, scanf("%f%c%f",&a,&opr,&b) != EOF){
		switch (opr) {
			case '+': result = a+b; break;
			case '-' : result = a+b; break;
			case '*' : result = a*b ; break;
			case '/' : result  = a/b ; break;
			default: bye("ERROR, bad operator\n",1);
		}
		printf (" result is %f\n", result);
	}
	bye("Normal exit\n",0);
}
void
bye(char message[], int error_code)
{
	printf ("%s",message);
	exit (error_code);
}
```

COMMENTS:

```
/* anything */
```

Can span many lines;
Cannot be nested: warning when "commenting out"

```
/* a=1;
/* a comment */   AAAARGHHH!
b=1; */
```

Compiler treats them as 1 space (ANSI)
(undefined in Old C)

```
a/* something */b is the same
                    as
a b               (ANSI)
ab                  (usually, in Old C)
```

PREPROCESSOR STATEMENTS

```
#include <standard header file>
#include "local header file"
```

standard header file **stdio.h**:
Required in (essentially) every program;
contains all the I-O related definitions, macros, etc.;

```
#define MACRO definition
```

ends with newline

MACRO will be replaced by its expansion in every occurrence,
then rescanned for further substitutions, etc.

PROMPT replaced by **putchar**
putchar is macro, defined in <stdio.h>

WARNING : infinite recursion
MACRO will not be replaced again in its own expansion (ANSI ONLY)

```
#define A A+B    /* VERY UNWISE; but would work in
                    ANSI , would hang your compiler*/
```

WARNING: common mistake

```
#define MACRO=definition /*wrong; = part of MACRO*/
#define MACRO = definition /*wrong; same*/
```

FUNCTIONS

analogous to FORTRAN subroutines-functions:
- encapsulate an operation
- give it a name
- call it from anywhere

2 functions in this example

main
bye

main required in each program
bye : user function.

FUNCTION DEFINITION:

header
body

body :
{
0 or more*definitions* and *declarations*
1 or more *statements*
} **/* this is a compound statement */**

header is very different in ANSI and Old C

ANSI

type returned by the function
(optional ; default : **int**)
(**void** means no value returned
(FORTRAN subroutine))
name of the function
(
argument type argument name, argument type argument name etc.
)
or
(void) meaning no arguments

OLD C
type returned by the function
(optional ; default : **int**)
name of the function
(
argument name, argument name etc.
)
argument type argument name;**/* please note the ; */**
argument type argument name **;**
etc. for all the arguments

or

() meaning no arguments

Example:
int sum (n1,n2) / *"int" can be omitted, it's the default*/
int n1,n2;
....

To run on your compiler, example above must be written

```
main()
{
......
}
bye(message,error_code)
char message[];
int error_code;
....
```

FUNCTIONS RETURNING A VALUE

return*value*; /* value should be of the type of the function */

```
int sum(int n1 , int n2){
      int s;
      s = n1 + n2;
      return s;
}
```

INVOKING FUNCTIONS

Simply by naming them, followed by the appropriate argument list; Functions without arguments must be followed by an empty argument list;

```
bye("this is a message",27);
bye();/* WRONG: ARGUMENT LIST MUST MATCH
         THE ONE IN FUNCTION DEFINITION.
         ERROR DETECTED BY ANSI C
      */
```


Functions can be recursive:

```
int fac(int n)
{ if (n == 0)
      return(1);
  else
      return (n*fac(n-1));/* note recursive call */
}
```

VARIABLE DECLARATION

Each variable has to be declared before being used (and before any "ordinary" statement):

type variable name ;/* **possibly other names, separated by "," */**

type :
int, float, char

int i1,i2;

FUNCTION DECLARATION (PROTOTYPING)

- Required if function defined after being used (like **bye**);
- Can be omitted : function in this case assumed to return type **int;**
- ANSI : should be identical to the function header; compiler uses it to check call;

- Old C: should declare only type returned:
  int bye();
-- often omitted if type **int** or neglected

THE **while** STATEMENT

while (*expression*)
  statement

Loops repeating *statement* until *expression* becomes 0

THE "," OPERATOR

expression1 ,expression2
evaluate expression1
discard its value (?!!!)
evaluate expression2
return its value

Example:

PROMT -> **putchar(:)**
  standard I-O function
  puts a character on output
  value : error code
    BUT
  side effects ! (in this case, display :)
scanf(....) != EOF
  relational expression
  Its value retained and tested by **while**

RELATIONAL OPERATORS

== test for equality
!= test for unequality
> greater
>= greater or equal
< less
<= less or equal
  return 1 if satisfied
  return 0 if not satisfied

C has no "logical" ("Boolean").
ALL CONDITIONALS TEST FOR 0 (false)/ NON 0 (true)

why C is dangerous?

if (i != j) i = i+1;
  can be written
i = i + (i != j); /*LEGAL (AAARGHH)*/

ARITHMETIC OPERATORS

+, -, *, / as usual
%: remainder (for **int** only)
int a,b;
(a/b)*b+a%b == a /*(careful if a <0)*/

THE **switch** STATEMENT AND **break**

switch (*expression*) {
    case *constant 1* :*statements* ;
    case *constant 2* :*statements* ;
    ....
    default : *statements* ;
}

- execution jumps to the label whose constant value is equal to *expression*, or to **default;**
- 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.**

THE ASSIGNEMENT STATEMENT

As usual:

variable =*expression* ;

THE SEMICOLON ;

- Every statement must be terminated by **;**
-- except COMPOUND STATEMENTS (that is **{ .... }**)
- Declarations and definitions must be terminated by **;**
- Function headers must NOT be terminated (why?)

WARNING

while(*expression* **)** is not a statement by itself -> no **;**
while (*expression* **) a=a+1;**

while (*expression* **) {.....}**

WHAT DOES THIS MEAN?

while (i < 10) ; i = i+1 ; /*AAARGHH!*/

CHARACTERS AND STRINGS

'a' is a character constant
char a; /*character variable */
a='a';

NON-PRINTABLE CHARACTERS:
'\n' is the *newline* character
'\t' is the *tab* character

STRINGS
"this is a string" is a string constant

- C has no basic string type
- a string variable is an **array of characters**
-- char message[];
- a string is always terminated by '\0' (ASCII NULL)
-- the compiler adds the '\0' to string constants

```
char s[4];
s[0]='b';
s[1]='y';
s[2]='e';
s[3]='\0';
/* Now s is a valid string */
```

THE BASIC I-O LIBRARY

C has no bult-in I-O operations
All I-O performed through library calls
Basic I-O macros and functions declared in **<stdio.h>**
STANDARD HEADER FILE

EOF : macro defined in <stdio.h>
an **int** that would not be a valid **char**
WARNING : never use -1. EOF is implementation dependent

int
putchar(char c)

puts the character c on the standard output
If it succeeds, it returns the value of **c;**
If it fails, it returns **EOF**

int
printf(char format[],...)

-A function with a variable number of arguments.
-The values of arguments from 2 on are converted according to the format (argument 1) and placed on standard output.
-If successful, returns the number of characters printed; otherwise, a negative value.

format: a string. It is copied on the output, except the conversion directives, that have the form
%d (integer conversion) or **%f** (floating conversion)
or **%s** (string conversion) or **%c** (character conversion)

Example:

```
printf (" These are %d things",2);
```

prints

These are 2 things

```
printf(" PI is %f",3.14);
```

prints

PI is 3.14

```
printf (" These are %d %s",2,"things");
```

prints

These are 2 things

\n in format prints a newline

int
scanf(char format[],...)

-A function with a variable number of arguments.
-The arguments from 2 on must be ADDRESSES of variables.
-The input is scanned and converted according to format, and values are placed, one after the other, in the other arguments. White spaces in format indicate skipping of whites and tabs in input.

Example

```
float a,b;
char opr;
scanf("%f%c%f",&a,&opr,&b);
```

input line:
32.5*12.0\n

%f converts 32.5 to a float (stops at *, illegal for float) and puts in a
%c reads 1 character '*' to opr
%f converts 12.0 to a float (stops at \n) and puts in b

```
float a,b;
char opr;
scanf("%f %c %f",&a,&opr,&b);
```

input line:
32.5   *   12.0\n

%f converts 32.5 to a float (stops at ' ', illegal for float) and puts in a
blank skips all blanks up to *
%c reads 1 character '*' to opr
blank skips all blanks up to 1
%f converts 12.0 to a float (stops at \n) and puts in b

THE & OPERATOR

Takes the address of its argument

C functions receive the VALUE of their scalar arguments ("call by value"): that is a copy of them. If they have to modify them, they must receive their addresses.

TYPES AND DECLARATIONS

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

All variables 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 :
ANSI: are distinguished on the basis of their first 6 characters, case indipendent.
WHY? Limits of system software

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 (ANSI)**
- *scalar types*
  - *arithmetic types*
    - *integral types*
    - *floating types*
  - *pointer types*
  - *enumeration types*(ANSI)

---

void

- no values

--- to specify the type of a function that returns no value

void bye(...)

--- to specify a function without arguments

main (void)

WARNING : Old C's **main()** ---

--- to specify pointers to objects of any type
(Old C uses pointers to chars, but...)

ARITHMETIC TYPES
Floating types

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

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>,** ANSI only)

- 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**

ANSI ONLY
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 (ANSI ONLY) or **unsigned**

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

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:**

BUT 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 real character of the alphabet

CHARACTER CONSTANTS

'9' 'A'

'\n' '\t'
'\0','\107'
'*octal number*'
'\x47'
'\x*hexadecimal number*' **ANSI ONLY**
'G' is '\107' is '\x47'

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()
{     char c = 'a' ;
      while(c <= 'z'){
            printf("%c %d" , c , c) ;
            c = c+1 ;
}
```

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

---

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 ; /* 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 (ANSI only)
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

```
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**)
(**float** to **double**, Old C only)

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**) (ANSI)
convert to **int,** return **int** (old C)
)

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; /*UNDEFINED, overflow */
```

ENUMERATED TYPES
(ANSI only)

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

```
enum { red, blue, gree, yellow }
     LightColor,PaperColor;
enum Brightness{bright, medium, dark} ;
enum Brightness BulbIntensity, ScreenIntensity;
```

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;

```
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}stat;
/* 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

Ex:
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;
```

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

CONSTANT POINTERS

On most machines:

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

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

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

---

typedef

USER DEFINED TYPES

```
/* 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
- operand 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;
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 <u>side effects</u>)
- WARNING : this is NOT the comma that appears in function calls

int f(int *n1, int *n2);
...
f(&n1 , &n2) /* it 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

- 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

- 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 <u>even removing unneeded parentheses</u>
--- **+** 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 operand

  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 */

- **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 **()**

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)

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)
check for **EOF** only if **getchar** called

ONLY operators with guaranteed order of evaluation;
therefore
ONLY case where repeated use of variable affected by side effects is safe

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();/* Old C style*/

if (p1=status(something))...
```

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**

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)

WARNING : not exactly same as simple assignment

Left operand evaluated only once

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

means

```
a[i] = 3;
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 ;

mean:
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 ? */
```

empty statement

```
;
```

Ex:

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

GREAT OPPORTUNITY FOR MISTAKES

COMPOUND STATEMENTS


```
{
      definitions and declarations;
      statement list;
}

{     int i,j;

      for (i=1,j=n ; i<=n/2 ; i++,j--){
            float temp;

            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
      }
}
/* NOTE : compound statements terminated by
 *           } not by ;
 */
```

- Compound statements can nest;
- Variables defined in a compound statement hide definitions of variables of same name outside;
- Functions bodies are a single compound statement;

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 no ; */
```


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*

USE COMPOUND STATEMENTS EVEN IF NOT NEEDED, TO BE SURE;
or
ALWAYS USE **else**, POSSIBLY WITH A NULL STATEMENT

THE **switch** STATEMENT AND **break**

switch (*expression*) {
  case *constant1* :*statements1*;
  case *constant2* :*statements2* ;
  ....
  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 *constant1* :
  case *constant2* :*statements1* ;**break**;
  case *constant3* :*statements2* ;**break** ;
  ....
  default : *statements* ;
}

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: refer to lecture 1

THE **for** STATEMENT

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

	for(i=0 , s=0 ; i<10 ; i++ , s+=n) {
		scanf("%d",&n);
		printf("%d",n);
	}
	printf ("Sum is %d",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
 */
#include <stdio.h>
#include <ctype.h>

int make_int(void)
{
    int num = 0 , digit;
    for(digit = getchar() ;
        digit != EOF && isdigit(digit) ;
        digit = getchar() )
    {
        num *= 10;
        num += digit -'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 above is a single statement:

```
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

---

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

---

UNCONTROLLED JUMPS : **goto**

```
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]);
    }
error: printf("%s %d %d", "negative at", i,j);
exit (1);
```

break would not work because exiting from 2 loops labels: any string followed by ":"

- do not need to be pre-declared
- must be part of a statement
---at end of compound statements

```
        label_at end : ; /* ; required*/
    }
```

- visible only from inside the function where they are used

ALMOST NEVER USED