REPLAY PART

Il linguaggio di programmazione C

La programmazione

I linguaggi di programmazione sono:

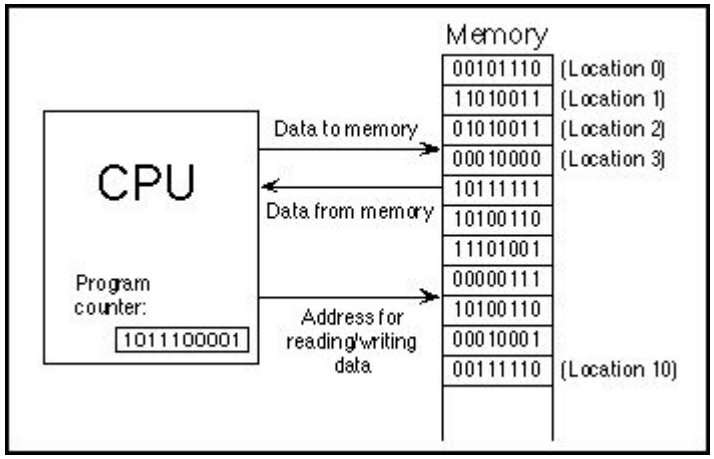
- Linguaggi formali (artificiali)
- Progettati per essere:
  - Sufficientemente "**espressivi**" per descrivere un'ampia gamma di algoritmi
  - **Comprensibili ai calcolatori in modo non ambiguo** e possibilmente "efficiente"
  - **Comprensibili a un programmatore umano** in modo non ambiguo e possibilmente "naturale"
- Descritti da una grammatica

Programmare significa in qualche modo dialogare con un elaboratore elettronico. Pertanto e' necessario imparare la lingua di tale strumento oppure il linguaggio predefinito di un possibile interprete che si pone tra noi e il computer.

Programmazione e livello di astrazione

Programmare significa implementare modelli astratti, strutture dati e algoritmi in una sequenza di istruzioni comprensibili per un calcolatore.

Ogni processore (CPU) per sua definizione esegue una lista di operazioni semplici che sono state predisposte come circuiti elettrici fisici al suo interno. Questo insieme di istruzioni primitive e' chiamato ***"instruction set"*** **(insieme istruzioni di macchina)**



L'insieme di istruzioni macchina (instruction set) è specifico di una particolare CPU

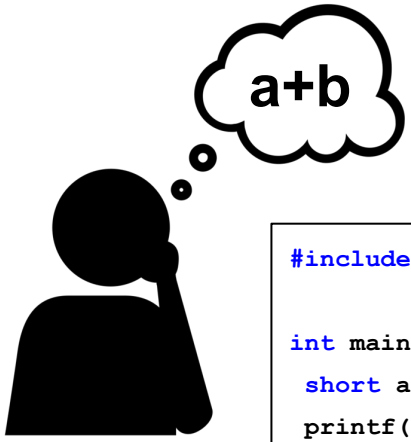
Per riferimento si puo' vedere la lista completa delle istruzioni macchina per intel x86 e' riportata in questo articolo: https://en.wikipedia.org/wiki/X86_instruction_listings

Programmazione e livello di astrazione

Sarebbe folle decidere di programmare ricordando tutte le 1503 istruzioni di base del processore, inoltre il codice non sarebbe portabile su diversi modelli di processore.

Una prima astrazione e' rappresentata dal linguaggio ***assembly*** dove le istruzioni di base sono sostituite con etichette mnemoniche (es: **addl**) .

Il codice **C** e' compatto, portabile e tuttavia molto vicino alla struttura di base della sequenza di operazioni macchina eseguita.




```c
#include <stdio.h>

int main(void) {
 short a = 3, b = 4;
 printf("%d",a+b);
 return 0;
}
```

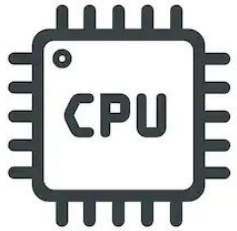
C

```
    .text
    .section    .rodata
main:
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp
    movw    $3, -4(%rbp)
    movw    $4, -2(%rbp)
    movswl  -4(%rbp), %edx
    movswl  -2(%rbp), %eax
    addl    %edx, %eax
    movl    %eax, %esi
    leaq    .LC0(%rip), %rdi
    movl    $0, %eax
    call    printf@PLT
    movl    $0, %eax
    leave
    ret
```

Assembly

```
0000000 457f 464c 0102 0001 0000 0000 0000 0000
0000010 0001 003e 0001 0000 0000 0000 0000 0000
0000020 0000 0000 0000 0000 02c8 0000 0000 0000
0000030 0000 0000 0040 0000 0000 0040 000d 000c
0000040 4855 e589 8348 10ec c766 fc45 0003 c766
0000050 fe45 0008 bf0f fc55 bf0f fe45 d001 c689
0000060 8d48 003d 0000 b800 0000 0000 00e8 0000
0000070 b800 0000 0000 c3c9 6425 0000 4347 3a43
0000080 2820 4e47 2955 3920 312e 302e 0000 0000
0000090 0014 0000 0000 0000 7a01 0052 7801 0110
00000a0 0c1b 0807 0190 0000 001c 0000 001c 0000
00000b0 0000 0000 0038 0000 4100 100e 0286 0d43
00000c0 7306 070c 0008 0000 0000 0000 0000 0000
00000d0 0000 0000 0000 0000 0000 0000 0000 0000
00000e0 0001 0000 0004 fff1 0000 0000 0000 0000
00000f0 0000 0000 0000 0000 0000 0000 0003 0001
0000100 0000 0000 0000 0000 0000 0000 0000 0000
0000110 0000 0000 0003 0003 0000 0000 0000 0000
0000120 0000 0000 0000 0000 0000 0000 0003 0004
0000130 0000 0000 0000 0000 0000 0000 0000 0000
0000140 0000 0000 0003 0005 0000 0000 0000 0000
0000150 0000 0000 0000 0000 0000 0000 0003 0007
0000160 0000 0000 0000 0000 0000 0000 0000 0000
0000170 0000 0000 0003 0004 0000 0000 0000 0000
0000180 0000 0000 0000 0000 0000 0000 0003 0006
0000190 0000 0000 0000 0000 0000 0000 0000 0000
00001a0 0007 0000 0012 0001 0000 0000 0000 0000
00001b0 0038 0000 0000 0000 000c 0000 0010 0000
00001c0 0000 0000 0000 0000 0000 0000 0000 0000
00001d0 0022 0000 0010 0000 0000 0000 0000 0000
00001e0 0000 0000 0000 0000 6100 6464 632e 6d00
00001f0 6961 006e 475f 4f4c 4142 5f4c 464f 5346
0000200 5445 545f 4241 454c 005f 7270 6e69 6674
0000210 0000 0000 0000 0000 0023 0000 0000 0000
… [ etc ]
```

bin code



Linguaggi compilati e interpretati (e ibridi)

1) LINGUAGGI COMPILATI: La **compilazione** e' l'operazione di trasformare un programma scritto in un linguaggio generico (ovvero astratto dalla architettura in cui sara' eseguito) in un codice direttamente eseguibile dal processore. Ad esempio, come abbiamo visto, un'operazione "+" nel codice sorgente potrebbe essere tradotta direttamente nella corrispondente istruzione "ADD" nel codice macchina.

Vantaggi linguaggi compilati:

- Il codice sorgente e' gia' tutto disponibile al compilatore e puo' essere applicata una forte **ottimizzazione**.
- Opportunita' di utilizzare **algoritmi nativi** per l'architettura scelta e rendere il codice molto efficiente.

2) Viceversa un linguaggio interpretato e' letto da in programma "interprete" (parser) che via via traduce le istruzioni che riceve (in run-time) in comandi preimpostati eseguibili dal processore. Evidentemente l'interprete e' a sua volta un programma compilato.

Vantaggi linguaggi interpretati:

- Generalmente **piu' semplici** da scrivere e semplice da portare su diverse architetture.
- **Il codice puo' essere eseguito al volo** senza alcun passaggio intermedio
- E' possibile l'**introspezione**, ovvero far si che il codice osservi se stesso.

3) LINGUAGGI IBRIDI: Linguaggi con compilatori JIT (Just in Time), il codice e' interpretato e compilato a blocchi. Ogni successivo riutilizzo di tali blocchi di codice usufruisce dei vantaggi della compilazione.

Versioni del linguaggio C

Sviluppato da Dennis Ritchie ai Bell Labs nel 1972 per realizzare il sistema operativo UNIX

K&R C: 1978 (prima versione, chiamato “K&R” dal nome degli autori del libro che lo ha divulgato: Brian W. Kernighan e Dennis M. Ritchie)

ANSI C: 1989 (alias: Standard C, C89)

ISO C: 1990 (quasi identico al C89, alias: C90)

C99: 1999 (Nuovo standard ISO)

C11: 2011

C18: 2018

Dove si utilizza il C

Nato insieme a Unix, è supportato dalla totalità dei sistemi operativi di largo uso impiegati ed è impiegato principalmente per la realizzazione di sistemi operativi, altri linguaggi di programmazione, librerie e applicazioni altamente performanti; è rinomato per la sua efficienza e si è imposto come linguaggio di riferimento per la realizzazione di software di sistema su gran parte delle piattaforme hardware moderne.

Alla base del sistema operativo della maggiorparte degli elaboratori elettronici.

Nei sistemi embedded

Per programmare device IOT

Il C per i sistemi di analisi

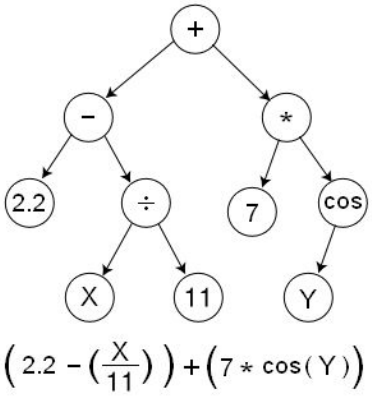
La quasi totalità degli odierni sistemi per l'analisi di dati sfrutta la capacità del C / C++ di essere molto vicino alla implementazione hardware per garantire la massima efficienza in ogni architettura.

Principali architetture di calcolo: **CPU**, **GPU**, **FPGA** e **ASIC** (TPU)

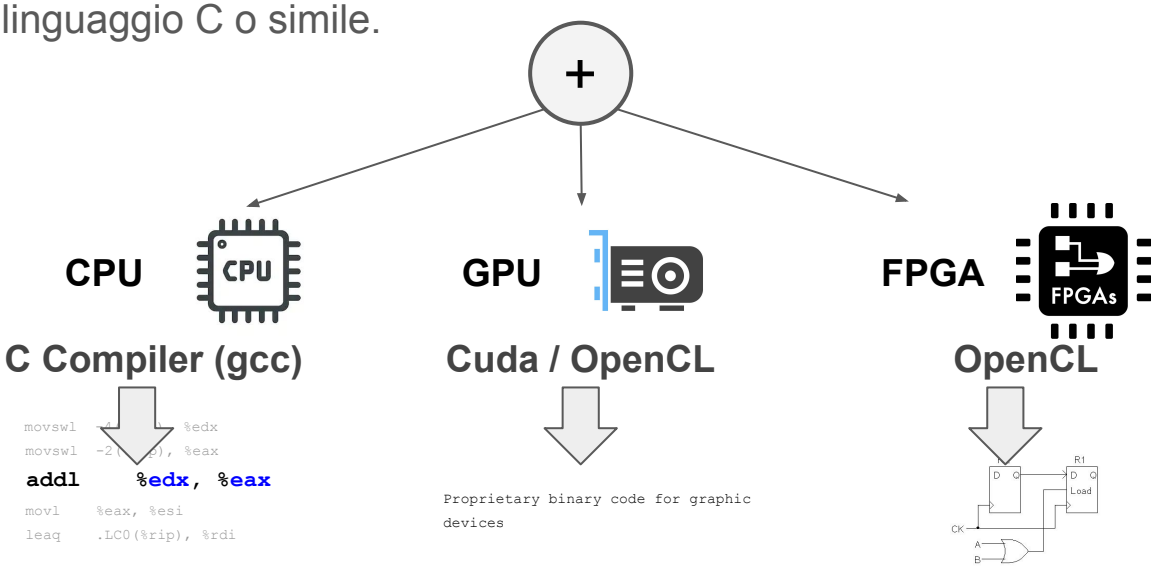
L'esigenza di sfruttare al massimo risorse di calcolo eterogenee ha portato alla diversificazione del codice.

Ad esempio: **TensorFlow**, **PyTorch**

Partendo da un linguaggio interpretato (python) sono in grado di costruire un albero delle operazioni e distribuire ogni singola operazione sulla architettura adeguata.



Per fare questo ogni operazione assume un carattere simbolico a cui corrisponde una implementazione diversa quasi sempre implementata in linguaggio C o simile.



Lezione 1: unità 4

Sintassi del C

32 parole chiave in C

Type definition:

char	double	enum	float	int
long	short	union	struct	void

Type qualifiers:

const	signed	typedef	sizeof	unsigned
volatile				

Storage classes:

auto	static	register	extern

Control flow:

break	case	do	else	for
goto	if	return	switch	while
default				

Sintassi di base

In C un programma e' definito da una sequenza di istruzioni (<statements>) che vengono eseguite una alla volta. Per una descrizione completa della sintassi del C si veda: https://en.wikipedia.org/wiki/C_syntax

Espressione semplice

```
<statement> ;
```

esempio

```
a = b + 1;
printf("Hello World!\n");
```

Espressione composta (detta '*blocco*')

```
{
    <statement> ;
    <statement> ;
}
```

esempio

```
int a = 0;
{
  int a = 1;
  printf("a within block scope: %d\n", a);
}
printf("a in function scope: %d\n", a);
```

istruzioni di preprocessore

```
// inclusione di un file
#include <file.h>

// definizione di una macro
#define NAME substituted expression

// chiamata diretta al preprocessore
#pragma parameter-list
```

esempio

```
// include un intero file come se fosse stato scritto qui
#include <stdio.h>

// sostituisce ovunque M_PI con 3.14 prima di compilare
#define M_PI 3.14

// warning per ogni funzione che non ritorna un valore
#pragma warn +rvl
```

Sintassi di base

dichiarazione delle variabili e assegnazione

```
type name = value;
```

esempio

```
int  who_u_call = 5552368;
```

dichiarazione di una funzione e chiamata

```
<return-type> functionName( <parameter-list> )
{
    <statements>
    return <expression of type return-type>;
}
```

```
functionName( <parameter-list> )
```

esempio

```
float my_pow( float x, unsigned int e ) {
    float res = x;
    int i;
    if (e == 0) return 1.;
    for ( i=0; i<e-1; i++ ) {
        res *= x;
    }
    return res;
}
```

```
printf("%f\n", my_pow(3.,3));
```

funzione principale

```
int main( int argc, char* argv[] ) {
    <statements>
    return status;
}
```

esempio

```
int main(int argc, char* argv[]) {
   for( int i=1; i<argc; i++) printf("%s",argv[i]);
   return 0;
}
```

Controllo di flusso

Istruzioni di diramazione

```
if (<expression>)
    <statement1>
else
    <statement2>
```

esempio

```
if ( res == 0 ) {
  printf("no errors.\n");
  do_something();
} else {
  printf("errors found.\n");
  handle_error(res);
}
```

```
<expression> ? <statement1> : <statement2>
```

```
res == 0 ? do_something() : handle_error(res);
```

Istruzioni di iterazione

```
for ( <init> ; <test> ; <advance> )
    <statement>
```

```
do <statement> while ( <expression> );
```

```
while ( <expression> )
    <statement>
```

esempio

```
for ( int i=0; i<100; i++ ) {
  printf("iterazione: %d", i);
  do_something_for(i);
}
```

```
int res;
do {
  res = do_something();
} while(res == 0);
```

```
while ( i > 0 ) {
  printf("some iterations left … ");
  do_something();
  i++;
}
```

Controllo di flusso (switch)

Istruzione switch

```
switch ( <expression> )
{
    case <label1> :
        <statements 1>
    case <label2> :
        <statements 2>
        break;
    default :
        <statements 3>
}
```

esempio

```
double n1, n2;
char operator;
switch(operator) {
        case '+':
            printf("%.1lf + %.1lf = %.1lf",n1, n2, n1+n2);
            break;

        case '-':
            printf("%.1lf - %.1lf = %.1lf",n1, n2, n1-n2);
            break;

        // operator doesn't match any case constant +, -, *, /
        default:
            printf("Error! operator is not correct");
  }
```

Istruzione break

L'istruzione break interrompe tutte le iterazioni successive dei costrutti **switch**, **while**, **do** e **for**.

Istruzione continue

L'istruzione continue interrompe l'iterazione corrente nei costrutti **while**, **do** e **for**.

Istruzione goto

Le regole del linguaggio C, definiscono anche l'istruzione di salto **goto**, sebbene il suo uso sia deprecato, poiché esce dagli schemi della programmazione strutturata. Comunque la sintassi e' la seguente:

```
goto identificatore;

identificatore:
  <statement>
```

Operatori

Operator name		Syntax	
Basic assignment		`a = b`	`int a = 1, b = 2;`
Addition		`a + b`	`c = a + 2; // c == 3`
Subtraction		`a - b`	`c = a - b; // c == 1`
Multiplication		`a * b`	`c = a * 2; // c == 2`
Division		`a / b`	`c = a / b; // c == 0`
Modulo (integer remainder)		`a % b`	`c = 8 % 3; // c == 2`
Increment	**Prefix**	`++a`	`a = 1; ++a; // a == 2`

	Postfix	`a++`	`a = 1; a++; // a == 1`
			`// a == 2`

Decrement	**Prefix**	`--a`	`a = 1; --a; // a == 0`

	Postfix	`a--`	`a = 1; a--; // a == 1`
			`// a == 0`

Operatori

Confronto	Syntax
Equal to	`a == b`
Not equal to	`a != b`
Greater than	`a > b`
Less than	`a < b`
Greater than or equal to	`a >= b`
Less than or equal to	`a <= b`

Logic	Syntax
Logical negation (NOT)	`!a`
Logical AND	`a && b`
Logical OR	`a || b`

Bitwise logic	Syntax
Bitwise NOT	`~a`
Bitwise AND	`a & b`
Bitwise OR	`a | b`
Bitwise XOR	`a ^ b`
Bitwise left shift	`a << b`
Bitwise right shift	`a >> b`

Per una lista completa degli operatori disponibili si veda: https://en.wikipedia.org/wiki/Operators_in_C_and_C%2B%2B

Tipi primitivi

DATA TYPE	MEMORY (BYTES)	RANGE	FORMAT SPECIFIER
short int	2	-32,768 to 32,767	%hd
unsigned short int	2	0 to 65,535	%hu
unsigned int	4	0 to 4,294,967,295	%u
int	4	-2,147,483,648 to 2,147,483,647	%d
long int	4	-2,147,483,648 to 2,147,483,647	%ld
unsigned long int	4	0 to 4,294,967,295	%lu
long long int	8	-(2^63) to (2^63)-1	%lld
unsigned long long int	8	0 to 18,446,744,073,709,551,615	%llu
signed char	1	-128 to 127	%c
unsigned char	1	0 to 255	%c
float	4		%f
double	8		%lf
long double	12		%Lf

Il tipo delle variabili

Il **tipo** assegnato alla variabile e' una indicazione univoca per caratterizzare il contenuto, esso serve sia al programmatore per sapere come trattare il dato che al compilatore per la gestione dello spazio di memoria dedicato.

Esempi:

```
int   a = -3;
float b = -3;
```

sizeof(a) == sizeof(b) == **4**

Per quanto riguarda la memoria occupata non c'e' nessuna differenza tra un intero (int) e un numero in virgola mobile (float); tuttavia il loro contenuto e' molto diverso:

int a = -3;	1111 1111	1111 1111	1111 1111	1111 1101
float b = -3;	1100 0000	0100 0000	0000 0000	0000 0000

Casting delle variabili

Per ***type casting*** si intende la modifica di una variabile da un tipo di dati ad un altro. Il compilatore applicherà un casting automatico se la conversione ha significato. Ad esempio, se si assegna un valore intero a una variabile in virgola mobile, il compilatore convertirà int in un float. Il cast come operatore permette di rendere esplicito questo tipo di conversione, o di forzarlo quando normalmente non previsto.

Cast implicito

```
#include <stdio.h>

int main()
{
    int a = -3;
    float b = a;

    printf("%f\n",b);
}
```

risultato

```
-3.000000
```

anche se il contenuto binario della variabile**a** non e' espresso in termini del tipo di **b** (float) il compilatore ha eseguito un cast implicito per mappare il valore di a allo stesso valore espresso in float.

Cast esplicito

```
#include <stdio.h>

int main() {
    int x;
    for(x=97; x<=122; x++)
        printf("%c ", (char)x);
    return 0;
}
```

risultato

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

Il contenuto della variabile intera x e' trasformato in un carattere e quindi convertito nel codice ASCII corrispondente per stampare le lettere dell'alfabeto

Esempio: stampare il formato a bit

CODING EXAMPLE

```c
#include <stdio.h>

void bin32(unsigned int n)
{
   unsigned int i;
   for (i = 1 << 31; i > 0; i = i / 2)
        (n & i) ? printf("1") : printf("0");
}


int main()
{
   printf("Show binary version of unsigned int\n");
   int a = -3;
   bin32( a ); printf("\n\n");

   printf("Show binary version of float from implicit cast\n");
   float b = a;
   bin32( *(unsigned int *)&b ); printf("\n\n");

   return 0;
}
```

```
Show binary version of unsigned int
11111111111111111111111111111101

Show binary version of float from implicit cast
11000000010000000000000000000000
```

tipi composti (il costrutto struct)

Una struttura è un tipo di dati definito dall'utente. Una struttura crea un tipo di dati che può essere utilizzato per raggruppare elementi di tipi possibilmente diversi in un unico tipo.

dichiarazione di una struttura e inizializzazione

```
struct newTypeName
{
    type name1,name2;
    type name3;
    ...
} inst1, inst2;
```

// inizializzazione 1

```
inst1.name1 = val1;
inst1.name2 = val2;
inst1.name3 = val3;
```

// inizializzazione 2

```
struct newTypeName inst3 = { val1, val2, val3 };
```

// inizializzazione 3 (C99)

```
struct newTypeName inst5;
inst5 = (struct newTypeName){
            .name1 = val1,
            .name2 = val2,
            .name3 = val3
          };
```

esempio

```
#include <stdio.h>

struct Point { float x,y; };

int main() {
 struct Point p1;

 // inizializzazione 1
 p1.x = 0.0;
 p1.y = 1.0;

 // inizializzazione 2
 p1 = (struct Point){ 0.0, 1.0 };

 // inizializzazione 3
 p1 = (struct Point){ .y = 1.0, .x = 0.0 };

 printf("point: (%f,%f)\n",p1.x,p1.y);
 return 0;
}



// RESULT
// ------
// point: (0.000000,1.000000)
//
```

Variables “storage classes”

Le classi di archiviazione (storage classes) vengono utilizzate per descrivere le caratteristiche di una variabile / funzione. Queste caratteristiche includono fondamentalmente l'ambito, la visibilità e la durata della vita che ci aiutano a tracciare l'esistenza di una particolare variabile durante il run-time di un programma.

var specifier	Storage target	Initial value	scope	life
auto (default)	stack	garbage	Within block	End of block
extern	Data segment	zero	Global	End of program
static	Data segment	zero	Within block	End of program
register	CPU reg	garbage	Within block	End of block

Storage classes: **auto** (classe di default)

questa è la **classe di archiviazione predefinita** per tutte le variabili dichiarate all'interno di una funzione o di un blocco. Se non e' specificata nessun qualificatore per la classe di archiviazione allora questa e' considerata auto di default. Pertanto, la parola chiave auto viene **utilizzata raramente (mai :-)** durante la scrittura di programmi in linguaggio C.

Le variabili automatiche sono accessibili **solo** all'interno del blocco o funzione in cui sono state dichiarate e non al di fuori di esse (che definisce il loro ambito). Naturalmente, è possibile accedervi all'interno di blocchi nidificati all'interno del blocco o funzione padre in cui è stata dichiarata la variabile automatica.

```
int a = -1;

void function()
{
    // declaring an auto variable (simply writing "int a=32;" works as well)
    auto int a = 1;
    printf("Value of the variable 'a' declared in a function: %d\n", a);
}

int main( void )
{
    auto int a = 0;
    function();
    {
        auto int a = 2;
        printf("Value of the variable 'a' within a block: %d\n", a);
    }
    printf("Value of the variable 'a' in main: %d\n", a);
}
```

Result:

Value of the variable 'a' declared in a function: 1
Value of the variable 'a' within a block: 2
Value of the variable 'a' in main: 0

NOTA IMPORTANTE:

La parola chiave **auto** ha assunto un significato diverso dalla versione **C11 in poi** dove viene usata per chiedere al preprocessore di ricavare il tipo dal contesto.

Storage classes: **extern**

extern dice semplicemente che la variabile è definita altrove e non all'interno dello stesso blocco in cui viene utilizzata. Quindi una variabile esterna non è altro che una variabile globale che viene dichiarata per essere utilizzata altrove.

Inoltre, una normale variabile globale può essere resa esterna inserendo la parola chiave 'extern' prima della sua dichiarazione in qualsiasi funzione. Questo significa sostanzialmente che non stiamo inizializzando una nuova variabile ma invece stiamo semplicemente accedendo alla variabile globale.

```
#include <stdio.h>

int a = -1;

void function()
{
    // declaring an auto variable (simply writing "int a=32;" works as well)
    extern int a;
    printf("Value of the variable 'a' declared in a function: %d\n", a);
}

int main( void )
{
    extern int a;
    function();
    {
        extern int a;
        printf("Value of the variable 'a' within a block: %d\n", a);
    }
    printf("Value of the variable 'a' in main: %d\n", a);
}
```

Result:

Value of the variable 'a' declared in a function: -1
Value of the variable 'a' within a block: -1
Value of the variable 'a' in main: -1

Nota

Lo scopo principale dell'utilizzo di variabili esterne è che è possibile accedervi tra due diversi file che fanno parte di un programma di grandi dimensioni.

Storage classes: **static**

questa classe di archiviazione viene utilizzata per dichiarare **variabili statiche** che vengono comunemente utilizzate durante la scrittura di programmi in linguaggio C. Le variabili statiche **hanno la proprietà di preservare il loro valore anche dopo che sono fuori dal loro *scope* e conservano il valore dell'ultimo utilizzo**.

Quindi possiamo dire che sono **inizializzate solo una volta ed esistono fino alla fine del programma**. Pertanto, non viene allocata alcuna nuova memoria perché non vengono dichiarate nuovamente. Il loro ambito di applicazione è locale rispetto alla funzione per la quale sono stati definiti.

Le variabili statiche globali sono accessibili ovunque nel programma. Per impostazione predefinita, viene assegnato il valore 0 dal compilatore.

```
#include <stdio.h>

int a = -1;

void function()
{
    // declaring an auto variable (simply writing "int a=32;" works as well)
    static int a = 0;
    printf("Value of the variable 'a' declared in function: %d\n", a);
    a++;
}

int main( void )
{
    function();
    function();
    function();
}
```

Result:

Value of the variable 'a' declared in function: 0
Value of the variable 'a' declared in function: 1
Value of the variable 'a' declared in function: 2

Storage classes: **register**

questa classe di archiviazione dichiara variabili di registro che hanno le stesse funzionalità di quelle delle variabili automatiche. L'unica differenza è che il compilatore tenta di memorizzare queste variabili nel registro del microprocessore se è disponibile un registro libero. Ciò **rende l'utilizzo delle variabili di registro molto più veloce** di quello delle variabili memorizzate durante il runtime del programma. Se non è disponibile un registro libero, questi vengono memorizzati solo nella memoria.

Di solito poche variabili alle quali è necessario accedere molto frequentemente in un programma vengono dichiarate con la parola chiave register che migliora il tempo di esecuzione del programma. Un punto importante e interessante da notare qui è che **non possiamo ottenere l'indirizzo di una variabile di registro usando i puntatori**.

```
#include <stdio.h>

int a = -1;

int main()
{
    register int a = 0;
    printf("Value of the variable 'a': %d\n", a);
    printf("Value of the address of 'a': %p\n", &a);
    return 0;
}
```

Result: compilation error

```
main.c: In function ‘main’:
main.c:18:5: error: address of register variable ‘a’ requested
    printf("Value of the address of 'a': %p\n", &a);
    ^~~~~~
```

I Puntatori

Nelle dichiarazioni il modificatore asterisco (*) specifica un tipo puntatore. Ad esempio, se lo specificatore int fa riferimento al tipo intero, lo specificatore int* si riferisce al tipo "puntatore all'intero". **I valori del puntatore associano due informazioni: un indirizzo di memoria e un tipo di dati**.

La seguente riga di codice dichiara una variabile **puntatore a intero** chiamata ptr: `int * ptr ;`

Referenziazione

Quando viene dichiarato un nuovo puntatore ad esso è associato un valore non specificato (random). L'indirizzo associato a tale puntatore **deve** essere modificato mediante assegnazione prima di utilizzarlo. Nel seguente esempio, *ptr* è impostato in modo che punti ai dati associati alla variabile *a* :

```
int a = 0 ;
int *ptr = &a ;
```

A tale scopo viene utilizzato l'operatore unario **&** ("indirizzo-di"). Esso restituisce la posizione di memoria dell'oggetto che segue.

Dereferenziazione

È possibile accedere ai dati puntati tramite tramite il valore del puntatore. Nel seguente esempio, la variabile intera *b* è impostata sul valore della variabile intera *a* , che è 10:

```
int a = 10;
int *p;

p = &a ;
int b = *p;
```

NOTA:

Attenzione quindi, l'operatore * assume quindi due significati:

1. Nella dichiarazione per definire un tipo puntatore.
2. Nella deferenziazione per ottenere il valore puntato dal puntatore.