Breve introduzione al linguaggio C

Breve introduzione al linguaggio C

Come Java, il C é un linguaggio

strutturato ,

procedurale e

debolmente **tipizzato** .

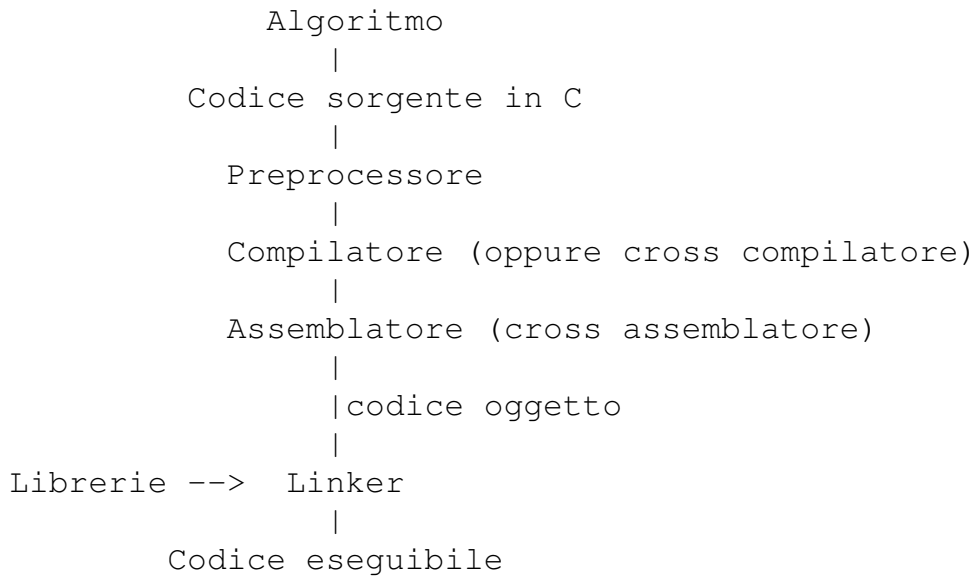
A differenza di Java, non é (esplicitamente) organizzato in **oggetti**

Struttura di un programma in C:

- Comandi del PREPROCESSORE
- Definizione dei tipi dei dati
- Prototipi di funzioni
- Variabili
- FUNZIONI (tra cui la funzione main())

L'entry point é la funzione main()

Generare codice eseguibile da un programma C



Programmi di compilazione:

funzione	Windows	Linux
Gestione progetto	nmake.exe	make
Preprocessore	cl.exe	cpp
Compilatore	cl.exe	gcc
Linker	link.exe	ld

Funzioni dei vari passi

Preprocessore:

- Gestisce le direttive del preprocessore:

```
inclusione di file:
#include <file>

definizione di macro:
#define simbolo

complazione condizionata
#ifdef simbolo
   ...istruzioni...
#endif
```

- elimina i commenti del sorgente

Compilatore: traduce il codice C in codice Assembler
Assemblatore: traduce il codice Assembler in codice oggetto privo di procedure di libreria (esempio cos(), ln()...)
Linker: collega tutti i codici oggetto provenienti dai sorgenti e dalle librerie,

Tipi di dati

Semplici

tipo	size in byte	range
signed char	1	$-128..+127$
unsigned char	1	$0..255$
short int	2	$-32768..32767$
unsigned short int	2	$0..65535$
long int (o int)	4	$-2^{31}..2^{31}-1$
unsigned long int	4	$0..2^{32}-1$
float	4	$-3.2 \times 10^{38}..3.2 \times 10^{38}$
double	8	$-1.7 \times 10^{308}..1.7 \times 10^{308}$
void	0	non definito

Complessi

tipo	size in byte
array monodimensionali	size in byte del tipo fondamentale $\times$ numero di elementi
array multidimensionali	size in byte del tipo fondamentale $\times$ numero di elementi
strutture (struct)	somma della dimensione dei dati costituenti
unioni (unions)	size in byte del massimo elemento
puntatori	size in byte degli indirizzi di memoria, tipicamente 4 byte

Tipi di dati complessi

Gli array sono insiemi di variabili dello stesso tipo dichiarati come: **tipo nomevar[numero elementi]**. Esempio:

```
int buf[10]; char stringa[20];
```

Il C alloca in memoria gli array in posizioni adiacenti

```
int buf[10]={1,2,3,4,5,6,7,8,9,0};
```

La **struct** é un tipo di dato astratto analogo alla classe. Esempio:

```
struct orografia{
   char nome[20];    //20 byte
   int longit;       //4 byte
   int latid;        //4 byte
   int altezza;      //4 byte
};
```

Una volta definito il tipo di dati astratto possiamo istanziare variabili:

```
struct orografia montebianco;
struct orografia monte[10];  //l'array monte[] occupa 360 byte
```

e accedere ai vari campi:

```
strcpy (monte[0].nome,"Monte Bianco"); monte[0].altezza=4800;
```

Tipi di dati complessi

L'istruzione **typedef** definisce nuovi tipi dai complessi. Esempio:

```
typedef struct{float pr, pi;} complesso;
```

A questo punto possiamo definire variabili complesse:

```
complesso p, carr[10];
carr[0].pr=1; carr[0].pi=1;
```

La **union** é un tipo di dato astratto che identifica oggetti di diversa dimensione e tipo ma che sono posizionati nella stessa zona. Esempio:

```
union all{
   struct orografia montebianco; //variabile di 36 byte
   complesso dato;               //variabile di 8 byte
   short int a;                  //variabile di 2 byte
};
```

Una volta definito il tipo union possiamo istanziare una variabile di quel tipo:

```
union all numero; //numero occuper 36 byte
```

Questa union é fatta per contenere il tipo maggiore (la struct di 36 byte). **Con la UNION Variabili di diverso tipo condividono lo stesso spazio di memoria**.

Visibilitá

- Variabili locali: dichiarate dentro ad un blocco di istruzioni, cioé entro due graffe ..istruzioni... Le variabili locali sono visibili solo dentro al blocco. Sono allocate sullo STACK.
- Variabili globali: dichiarate fuori da tutti i blocchi. Le variabili globai sono visibili da tutti i moduli dello stesso file. In altri file sono visibili SOLO se precedute dalla dichiarazione **extern**. Sono allocate solo nel punto dove sono dichiarate (senza **extern)**.
- Variabili formali: definiscono i parametri passati ad una funzione. Le variabili passate come tipi semplici sono passate per **valore**. I tipi complessi sono passati per **indirizzo**. Tutti i parametri passati ad una funzione sono passati sullo STACK.
- Incapsulamento : dichiarazione **static**. Una variabile statica viene allocata una sola volta ed é visibile solo all'interno del modulo dove é stata dichiarata.

Conversione di tipo

Due casi di conversione:

- Conversione automatica. C converte automaticamente la variabile a destra del segno = nel tipo a sinistra. Esempio, da intero a carattere: ch=i; oppure da float a intero: int=fl; o da intero a float: fl=int;
- Conversione esplicita: con il casting. Esempio: fl=(float)int;

DA	A	Informazione conservata	Possibile perdita info
char	signed char	7 bit meno significativi	valori $>$ 127
short int	char	byte meno significativo	valori $>$ 255
long int	char	byte meno significativo	valori $>$ 255
long int	short int	due byte -significativi	due byte + significativi
float, double	short int	parte intera se float $<$ 32767 altrimenti valore senza senso	parte frazionaria
double	float	valore arrotondato alla	arrotondamento
		precisione del float	

Operatori

Aritmetici	+ - * / % ++ −
Logici	&& || !
Relazionali	>>=<<===! =
abbreviati	+= -= *= /= %=
dimensione	sizeof()

Controllo del flusso

```
{

    if(.) then {.} else {.}

    switch { case }

    while() {.}

    do {.} while ()

    for(...){ }

    break/continue

}
```

Input e Output

Le funzioni di libreria sono definite in stdio.h Funzioni di input e output:

- int getchar(void); //lettura dallo standard input carattere per carattere
- int getc(file); //lettura dallo standard input carattere per carattere
- int scanf(formato,variabili); //lettura formattata dallo standard input
- int fscanf(file,formato,variabili); //lettura formattata dal file
- char *gets(char *); //lettura di una linea dallo standard input
- char *fgets(char * ch, int n, file); //lettura di una linea dal file
- char putchar(char ch); //scrittura sullo standard input carattere per carattere
- int putc(char, file); //scrittura sullo standard input carattere per carattere
- int printf(formato,variabili); //scrittura formattata dallo standard input
- int fprintf(formato,variabili); //scrittura formattata sul file
- fput(char *,file); //scrittura di una linea sul file
- int open(const char *pathname, int flags); //Linux system call
- size_t read(int fd, void *buf, size_t count); //Linux system call: lettura di count byte da file
- size_t write(int fd,void *buf, size_t count); //Linux system call: scrittura di count byte su file
- int close(int fd); //Linux system call

Input Output: formato

Il formato é una stringa di caratteri contenente caratteri di formato e variabili. Caratteri di formato:

- %d stampa un intero
- %10d stampa un intero e un minimo di 10 caratteri
- %f stampa un float
- %lf stampa un double
- %10f stampa un float e un minimo di 10 caratteri
- %10.5f stampa un float e un minimo di 10 caratteri con al massimo 5 cifre dopo il punto decimale
- %s stampa tutti i caratteri di una stringa (cioe' un vettore di caratteri con un 0 in fondo) fino a che incontra lo 0 finale
- %10s stampa almeno 10 caratteri di una stringa, con la stringa a destra
- %c stampa un singolo carattere
- caratteri di controllo: \n $\rightarrow$ NEW LINE (a capo linea), \t $\rightarrow$ TABULAZIONE, \b $\rightarrow$ BACKSPACE

Esempio:

```
int i=16; double g=107.13987; char *str="pippo";
printf("%d) %10.3lf \"%s\" \n",i,g,str);  --> 16) 107.139 "pippo"
```

Esempio di file unico

```
#include <stdio.h>              //direttive del preprocessore
#include <math.h>
#define pi 3.1415

float x, y;                     //variabili globali

void funct1(int i, int j);      //definizione funzioni
float funct2(float a);
int funct3(int i);

int main(void)
{
    short i,j;                  //variabili locali
    float res;
    for(i=0;i<7;i++){  j=funct3(i); funct1(i,j);  }
    res=(float)i*pi;
    printf("tangente(%d*pi)=%f\n",i,funct2(res));
}

void funct1(int i, int j)        //codice delle funzioni
{     printf("fattoriale di %d=%d\n",i,j);  }

float funct2(float a)
{     x=sin(a); y=cos(a);     return x/y;}

int funct3(int k)
{ if (k==0) return 1;  else   return k*funct3(k-1);  }
```

File diviso in moduli: prova1.c prova1.h funz.c

```
#include <stdio.h>              //direttive del preprocessore
#include "prova1.h"
#define pi 3.1415
int main(void)
{
    short i,j;                  //variabili locali
    float res;
    for(i=0;i<7;i++){       j=funct3(i); funct1(i,j);    }
    res=(float)i*pi;
    printf("tangente(%d*pi)=%f\n",i,funct2(res));
}
```

header file:

```
float x, y;                     //variabili globali
void funct1(int i, int j);      //definizione funzioni
float funct2(float a);
int funct3(int i);
```

funzioni

```
#include <stdio.h>              //direttive del preprocessore
#include <math.h>
extern float x,y;               //variabili locali allocate esternamente
void funct1(int i, int j)
{    printf("fattoriale di %d=%d\n",i,j); }
float funct2(float a)
{     x=sin(a); y=cos(a);    return x/y; }
int funct3(int k)
{   if (k==0) return 1; else return k*funct3(k-1); }
```

Compilazione in Linux

- Una libreria e un insieme di file oggetto riuniti in un file archivio.
- Le librerie sono contenute in /lib e /usr/lib con nome lib$<$ *nome* $>$.a
- Come si richiamano? Con la sintassi -l$<$ *nomelibreria* $>$
- Alcune librerie:
  - lc libreria standard. Viene caricata automaticamente
  - lcurses gestione del video ad alto livello
  - lm funzioni matematiche
- Se il numero di file sorgenti e maggiore di 10, allora diventa necessario usare il comando make per gestire il programma

Il comando make

- il comando make indica come vogliamo generare il programma.
- Il programma é lobiettivo (target) della operazione e dipende da uno o piú file, (componenti), i quali a loro volta possono dipendere da altri file.
- Il comando make deve conoscere questa gerarchia di dipendenze che vengono scritte nel file makefile o Makefile
- Le linee che contengono i due punti specificano le dipendenze. A sinistra dei due punti cé il target; a destra ci sono le componenti
- Le linee che cominciano con il tab sono linee comando; specificano come costruire i target
- make verifica le date per vedere se i file oggetto sono aggiornati. Se sono aggiornati non fá niente e scrive semplicemente

```
programma is up to date
```

Il comando make

Esempio: programma composto da tre sorgenti: prova1.c, funz.c, funz2.c

```
# questo e un commento
CC = gcc
programma : prova1.o funz.o funz2.o # dipendenza
    $(CC) -o programma  prova1.o funz.o funz2.o #comando
prova1.o : prova1.c # dipendenza
    $(CC) -c prova1.c # comando
funz.o : funz.c # dipendenza
    $(CC) -c funz.c # comando
funz2.o : funz2.c # dipendenza
    $(CC) -c funz2.c # comando
```

- Se lo script si chiama Makefile basta fare make
- Se lo script ha un nome generico make -f name

in pratica...

```
$ gcc prova.c -lm -o prova
$ ./prova
fattoriale di 0=1
fattoriale di 1=1
fattoriale di 2=2
fattoriale di 3=6
fattoriale di 4=24
fattoriale di 5=120
fattoriale di 6=720
tangente(7*pi)=-0.000649
$
$ gcc prova1.c funz.c -lm -o prova1
$ ./prova1
fattoriale di 0=1
fattoriale di 1=1
fattoriale di 2=2
fattoriale di 3=6
fattoriale di 4=24
fattoriale di 5=120
fattoriale di 6=720
tangente(7*pi)=-0.000649
```

File

Le operazioni fondamentali sono: apertura del file(open), eventuale posizionamento(seek), lettura/scrittura(read/write), chiusura del file(close). Esempio: mycat.c

```
#include <unistd.h>
#include <stdio.h>
#define BUFFSIZE 8192
int main(void)
{
    int n;
    char buf[BUFFSIZE];
    while ( (n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
       if (write(STDOUT_FILENO, buf, n) != n) {
           printf("write error"); exit(1);
       }
    if (n < 0) {
        printf("read error");
        exit(1);
    }
    exit(0);
}
```

File

Altro esempio: file mycp.c

```
/* copia di file. Tiene conto delle scritture incomplete */
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#define BUFSIZE 512
void copyok(char *from, char *to)
{ int fromfd, tofd, nread, nwrite, n;
        char buf[BUFSIZE];
        if((fromfd = open(from,0)) == -1)  syserr("from");
        if ((tofd = creat(to,0666)) == -1) syserr("to");
        while ((nread = read(fromfd, buf, sizeof(buf))) != 0) {
           if (nread == -1) syserr("read");
           nwrite = 0;
           do{ if ((n=write(tofd,&buf[nwrite], nread - nwrite)) == -1) syserr("write");
                nwrite += n;
           } while (nwrite < nread);
        }
        if(close(fromfd) == -1 || close(tofd) == -1) syserr("close");
}
main()
{
   char nome1[10], nome2[10];
   printf("nomi file? "); scanf("%s %s",nome1,nome2);
   printf("nomi letti: %s e %s\n", nome1,nome2);
   copyok(nome1,nome2);
}
```

Puntatori

L'uso dei puntatori é la vera, grande differenza tra Java e C.
In C ogni variabile caratterizzata da due valori: un indirizzo della locazione di memoria in cui sta la variabile, ed il valore contenuto in quella locazione di memoria, che il valore della variabile.
Un puntatore e' un tipo di dato, é una variabile c e contiene l'indirizzo in memoria di un'altra variabile, cioé un numero che indica in quale cella di memoria comincia la variabile puntata: *puntatore* $\rightarrow$ *variabile*
Per dichiarare un puntatore *p* ad una variabile di tipo tipo, l'istruzione e':

```
tipo *p;
```

L' operatore & fornisce l'indirizzo di una variabile, perció l' istruzione

```
p = &c
```

scrive nella variabile p l'indirizzo della variabile c, ovvero:

```
tipo c, *p; //dichiaro una var c di tipo tipo ed
            //un puntatore p a tipo
p = & c ;   //assegno a p l'indirizzo di c
```

Puntatori

L'operatore * viene detto operatore di indirezione o deriferimento. Quando una variabile di tipo puntatore é preceduta dall'operatore * , indica che stiamo accedendo all'oggetto puntato dal puntatore. Quindi con *p indichiamo la variabile puntata dal puntatore.

```
int c, *p; //dichiaro una var ed un puntatore p a int
p = &c ;   //assegno a p l'indirizzo di c c
c = 5;     //assegno a c il valore 5
printf("%d\n", *p); //stampo il valore puntato da p.
                    // viene stampato 5
```

Consideriamo gli effetti delle seguenti istruzioni:

```
int *pointer; /* dichiara pointer come  puntatore a int */
int x=1,y=2;
pointer= &x; /* assegna a pointer l'indirizzo di x */
y=*pointer; /* y = il contenuto dell'int puntato */
x=pointer /* assegna ad x l'indirizzo contenuto in pointer*/
*pointer=3; /* assegna 3 alla variabile puntata da pointer  */
```

Puntatori

Quindi con pointer possiamo considerare tre possibili valori:
1) pointer $\rightarrow$ contenuto o valore della variabile pointer cioé l'indirizzo della locazione di memoria a cui punta
2) &pointer $\rightarrow$ indirizzo fisico della locazione di memoria del puntatore
3) *pointer $\rightarrow$ contenuto della locazione di memoria a cui punta
Attenzione.

```
int *ip;
*ip=100;
```

$\rightarrow$ grave errore:
la locazione dove scrivo il valore 100 DEVE ESSERE ALLOCATA!
Altrimenti potrebbe cadere in una zona importante del sistema!
Come allocare spazio? o puntando ad una variabile allocata o usando

```
void *malloc(int nr_byte)
```

Aritmetica dei puntatori

- Uso degli operatori aritmetici +, - , ++ e – con puntatori.
- Non sono consentite operazioni aritmetiche tra puntatori, solo tra puntatori e numeri interi
- Sono consentiti confronti tra puntatori
- Ma: il risultato numerico di un'operazione aritmetica su un puntatore dipende delle dimensioni del tipo di dato a cui il puntatore punta.
- Infatti, incrementare un puntatore di uno significa spostare in avanti in memoria il puntatore di un numero di byte corrispondenti alle dimensioni del dato puntato dal puntatore.
- Ovvero: se p un puntatore di tipo puntatore a char (char *p) l'istruzione p++ aumenta effettivamente di un'unitá il valore del puntatore p, che punterá al successivo byte.
- Invece se p é un puntatore di tipo puntatore a **short int** (short int *p) l'istruzione p++ incrementa di 2 il valore del puntatore p, che punterá allo short int successivo a quello attuale.
- se il puntatore punta a void (void *p) il puntatore viene incrementato o decrementato a passi di un byte.

Puntatori e array

- Puntatori e Array: **il nome di un array é un puntatore alla prima locazione dell'array!**
- Esempio:

  `int buf[100];`

  → buf[10], *(buf+10) sono due modi EQUIVALENTI per accedere all'11esimo elemento di buf
- Esempio:

  `int arr[10]; while(*(arr++)=*(buf++));`

  → modo veloce per copiare l'array buf nell'array arr
- Attenzione: un array bidimensionale [M×N] **costruito dinamicamente** é un array di M puntatori che puntano a array monodimensionali di N elementi. Quindi é un puntatore di puntatori, definito come **p
- In questo caso, se la matrice é costruita dinamicamente ed ha M righe e N colonne, allora accedere all'elemento i,j puó essere fatto in uno dei seguenti modi: buf[i][j], *(buf[i]+j), (*(buf+i))[j], *((*(buf+i))+j), *(&buf[0][0] +M*i+j)
- altrimenti, se l'array é costruito staticamente, esempio float M[3][4], l'array é una successione di elementi. In questo caso accedere all'elemento [i][j] vuol dire

  `float *p=&M[0][0]; elem_i,j=*(p+i*N+j)`

Matrice bidimensionale dinamica

```
#include <stdio.h>          //direttive del preprocessore
float funct2(float **a)
{
     short i=0,j=1; printf("elem 0,1=%f\n", *(*(a+i*4)+j) );
     i=1;j=1; printf("elem 1,1=%f\n", *(a[i]+j) );
     i=2;j=2; printf("elem 2,2=%f\n", (*(a+i))[j] );
     i=3;j=3; printf("elem 3,3=%f\n", a[i][j] );
     return 0;
}
int main(void)
{
    short i,j;                    //variabili locali
    float p1[4]={1,2,3,4}, p2[4]={5,6,7,8};
    float p3[4]={9,10,11,12}, p4[4]={13,14,15,16};
    float *pp[4], **p=pp;;
    pp[0]=p1; pp[1]=p2;           //caricamento vettore di puntatori
    pp[2]=p3; pp[3]=p4;
    funct2(p);
    printf("fine\n");
}
```

Matrice bidimensionale statica

```
#include <stdio.h>          //direttive del preprocessore
#include <math.h>
float funct2(float *a)
{
     printf("elem 0,1=%f\n", *(a+0*4+1) );
     printf("elem 1,1=%f\n", *(a+1*4+1));
     printf("elem 2,2=%f\n", *(a+2*4+2));
     printf("elem 3,3=%f\n", *(a+3*4+3));
     return 0;
}

int main(void)
{
    short i,j;                    //variabili locali
    float res;
    float mat[4][4]={1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};
    float *p=&mat[0][0];
    printf("main elem 1,1=%f\n", *(p+1*4+1));
    funct2(p);
    printf("fine\n");
}
```

Codice estratto dallo schedulatore originale di Linux (Torvalds 1992)

```
void schedule(void)
{
    int i,next,c;
    struct task_struct ** p;
/* this is the scheduler proper: */
    while (1) {
        c = -1;
        next = 0;
        i = NR_TASKS;
        p = &task[NR_TASKS];
        while (--i) {
            if (!*--p)
                continue;
            if ((*p)->state == TASK_RUNNING \&\& (*p)->counter > c)
                c = (*p)->counter, next = i;
        }
        if (c) break;
        for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
            if (*p)
                (*p)->counter = ((*p)->counter >> 1) +
                        (*p)->priority;
    }
    switch_to(next);
}
```

Funzioni

- Forma generale della definizione di una funzione:

```
tipo_restituito nome_funzione (paramdef1, paramdef2, ...)
{
    dichiarazione variabili locali;
    istruzioni;
    return tipo_restituito;
}
```

- Passaggio di variabili da/a funzioni - per valore (copiando il valore sullo stack) - per indirizzo (passando il puntatore delle variabili sullo stack)
- Puntatori a funzione: sono variabili dove viene memorizzato l'indirizzo delle funzioni. Attraverso i puntatori si possono evidentemente eseguire le funzioni
- dichiarazione di un puntatore a funzione:

```
 <tipo_f> (* <nomeF>)(<lista_par_formali>);
dove:
 <nomeF>  l'identificatore del puntatore a funzione;
 <tipo_f>  il tipo di dato restituito dalla funzione puntata
 <lista_par_formali>  la lista dei parametri formali della funzion
```

- la chiamata della funzione avviene chiamando il nome del puntatore

Funzioni come parametri passati

Esempio:

```
#include <math.h>
double fun (double x) {
     return x*x;
}
double somma( double (*f)(double par), int start, int end) {
     int val;
     double result = 0.0;
     for (val=start; val<end; i++)
          result = result + (*f)(val);
     return result;
}
int main() {
     double num1,num2;

     num1 = somma(fun, 1, 1000);
     num2 = somma(sin, 1, 1000);
}
```