CORSO DI LABORATORIO DI PROGRAMMAZIONE

ANNO 2005 - 2006

CORSO DEL LINGUAGGIO C

Sviluppato con SO SUN OS 5.8 – compilatore unix sun – gcc version 3.2.2

Con note di utilizzo programmi con DEV++

Docente prof. Valter Cavecchia

Appunti dalle lezioni a cura di

Jyothi Zontini – Giovanni Sosi

Il primo programma in C

Il primo programma classico in C è quello che stampa “hello world!”
Lo chiameremo hello.c. Dovremo inserirci la libreria <stdio.h> tramite il preprocessore #include.
Devo scrivere almeno una funzione , la “main”; questa deve sempre essere presente in un programma in c (seguita poi eventualmente da molte altre).
Voglio scrivere “hello world!” sullo schermo; questo è complicato e devo delegare qualcuno a farlo, ovvero il SO con la funzione “printf”.

```
#include <stdio.h> // questa libreria va SEMPRE inclusa, in ogni programma!

int main(void) {
  (void)printf("hello world!\n");  /* il comando \n serve per andare a capo (n
                                di newline ) */
          return 0;

}

/* Si raccomanda sempre l'indentazione (rientro), in quanto con un programma ben indentato
risulterà molto piu facile scovare gli errori.*/
```

Per vedere il contenuto del file basta digitare >cat hello.c
Si usa >cat “hell .c” se uno mette spazi nel nome // rispetto al precedente ci sono le “”. La mancanza della “o” è voluta.
>mv mi permette di cambiare il nome del file

Compilazione del programma

Utilizzo il comando **gcc –o hello hello.c**

Dove gcc –o è il comando al compilatore , seguito dal nome del file in uscita compilato (hello) e poi dal nome utilizzato come codice sorgente (hello.c). In unix i file eseguibili non hanno estensione . Infatti con il comando gcc hello si ottiene in output un file eseguibile di nome a.out ; nel momento in cui ci si trovasse a compilare cosi due programmi diversi , avremmo due file di output con il medesimo nome, che quindi andrebbero a sovrapporsi. Utilizzando invece il comando –o, è possibile rinominare i file di output in fase di compilazione,in modo tale da evitare spiacevoli inconvenienti.
Stringhe di commento:
Ci sono due tipi di stringhe:
se commento su un’unica riga, digito

```
//commento su una riga
```

Se invece desidero commentare su piu righe, la stringa è la seguente:

```
/* qui inserisco il commento
   su piu righe */
```

Esempio: stampare i numeri pari < 30

Per poter stampare i numeri pari < 30 utilizzo la formula che i numeri pari sono divisibili per 2 senza resto.
Il programma sarà dunque:

```
#include <stdio.h>
#include <stdlib.h> // per chi usa windows questa libreria va sempre inclusa
int main(void) {

unsigned int i; // intero che mi designa il contatore
for (i=0;i<30;i++)  // per i che va da 0 a 30, incrementa i di uno ogni volta
  {
      if (i%2==0)    // se i è uguale a 0
          printf("%2u\n",i);}  // allora stampalo con formattazione unsigned
return 0; // il main non ha argomenti, quindi non ritorna nulla
}
```

Esempio: stampare le potenze del 2 con n < 30

```
#include <stdio.h>

int main(void) {

unsigned short int i;
p=1; // potenza del numero
for (i=0;i<30;i++){          // per i che va da 0 a 30, ogni volta stampa p

   printf("%d\n",p);
   p*=2;

}
return 0;
}
```

Esempio: stampare un triangolo di * (asterischi) del tipo:

```
*
**
***
****
*****
******
```

ecc ecc ecc

```
#include <stdio.h>

int main(void) {

unsigned int i,j;
for (i=0;i<30;i++){    // per i che va da 0 a 30
  for (j=0;j<=i;j++){  // per j che va da 0 ad i
    printf("*");       // stampami un asterisco
 }                     // quando j è = ad i
  printf("\n");        // vai a capo
}                      // quando i = 30
return 0;              // termina il main
}
```

Esempio: stampare lo stesso triangolo di * però rovescio:

```
******************
*****************
****************
***************
**************
```

ecc

```
#include <stdio.h>

int main(void) {

unsigned int i,j;
for (i=0;i<10;i++){
  for (j=10;j>i;j--){
   printf("*");
 }
   printf("\n");
}
return 0;
}
```

Esempio: dato un numero vedere la sua rappresentazione binaria:

```
#include <stdio.h>

int main(void) {

int x=20;
while(x>0)           // finchè x è > 0; termina quando x non è + maggiore di 0
  {
  printf("%d",(x%2));
  x=x/2;
  }
printf("\n");
return 0;
}
```

VARIABILI A BASSO LIVELLO – VARIABILI INTERE

Affrontiamo la gestione delle variabili a basso livello (parleremo solo di variabili intere)

Sono : char 1 byte
Int che può essere di tipo short – long – longlong
La parola , word = dipende dalla lunghezza dell'architettura del calcolatore

Long >= int >= short
Le architetture delle macchine moderne hanno un calcolo a virgola mobile. Normalmente non uso Floating Point, ma uniformo tutto a un valore: per esempio se devo fare una lista del peso delle persone di un'aula userò come unità di misura etti e non Kg.
Useremo nozioni di ottimizzazione, concetti per avere prestazioni migliori di un codice.
Per sapere come è lungo un intero in C c'è l'operatore:
sizeof(tipo) unsigned int

Esempio: visualizzare l'aspetto binario di un numero: (binari1.c)

```
#include <stdio.h>

int main(void) {

int i,n,m,x=8;
n=sizeof(unsigned int)*8;
for (i=n-1;i>0;i--)
{
   m=1<<i;
  (x&m)==m?printf("1"):printf("0");

/* equivale a scrivere:
                if ((x&m)==m)
                      printf("1");
                else
                      printf("0")
*/

}
printf("\n");
return 0;
}
```

Operatori bit a bit

In C e` possibile manipolare i valori di un byte agendo a livello dei singoli bit. Si tratta di costrutti usati raramente, sono però introdotti per completezza. Gli operatori bit a bit (anche detti *Bitwise Operator)* definiti in C sono:

Simbolo	Operazione realizzata
<<	Shift a sinistra
>>	Shift a destra
&	And bit a bit
|	Or bit a bit
^	Or esclusivo bit a bit
~	Complemento ad uno (Unario)

L'operatore << esegue lo **shift a sinistra** del suo operando di sinistra per il numero di posizioni specificate dal suo operando destro. I bit eccedenti (a sinistra) sono persi, i bit aggiunti (a destra) sono zero. Ad esempio:

7	<<	1	→	3
00000111				00000011
7	<<	3	→	0
00000111				00000000

L'operatore >> esegue lo **shift a destra** del suo operando di sinistra per il numero di posizioni specificate dal suo operando destro. I bit eccedenti (a destra) sono persi, i bit aggiunti (a sinistra) sono zero. Ad esempio:

7	>>	1	→	3
00000111				00000011
7	>>	3	→	0
00000111				00000000

L'operatore | esegue l'**or** bit a bit fra i suoi due operandi. Questo significa che il bit i-esimo risulter`a uguale ad uno se e soltanto se <u>almeno uno</u> dei bit i-esimi negli operandi e` uguale ad uno. Ad esempio:

7	|	12	→	15
00000111		00001100		00001111

L'operatore & esegue l'**and** bit a bit fra i suoi due operandi. Questo significa che il bit i-esimo risulter`a uguale ad uno se e soltanto se <u>ambedue</u> i bit i-esimi negli operandi sono uguali ad uno. Ad esempio:

7	&	12	→	4
00000111		00001100		00000100

L'operatore ^ esegue l'**or esclusivo** (anche detto xor) bit a bit fra i suoi due operandi. Questo significa che il bit i-esimo risulter`a uguale ad uno se e soltanto se uno solo dei bit i-esimi negli operandi e` uguale ad uno. Ad esempio:

7	^	12	→	11
00000111		00001100		00001011

L'operatore ~ esegue il **complemento ad uno** del suo operando. Questo significa che il bit i-esimo risulter`a uguale ad uno se e soltanto se il bit i-esimo dell'operando e` uguale a zero. Ad esempio (supponendo di avere un byte in complemento a due):

~7			→	-8
00000111				11111000

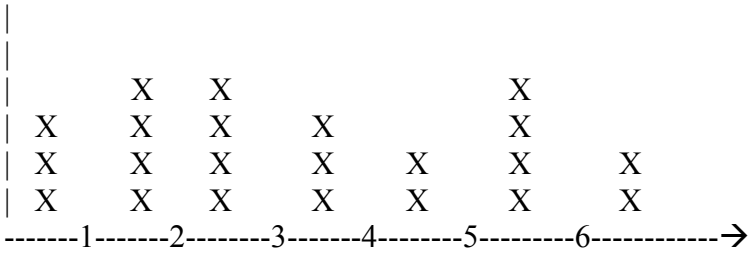
NUMERI PSEUDO CASUALI / RANDOM

Vogliamo prendere dei nomi e riordinarli ; per provarlo uso un generatore di numeri casuali. Rispolveriamo un po' di statistica: un numero casuale è un numero che non sono in grado di prevedere. Devo definire l'intervallo (range di validità) dove i numeri sono distribuiti e come sono distribuiti.
Esempio: uso un dado; suppongo che non sia truccato, cioè che tutte le facce abbiano la stessa probabilità di uscire e che la mano che lo lancia non bari.

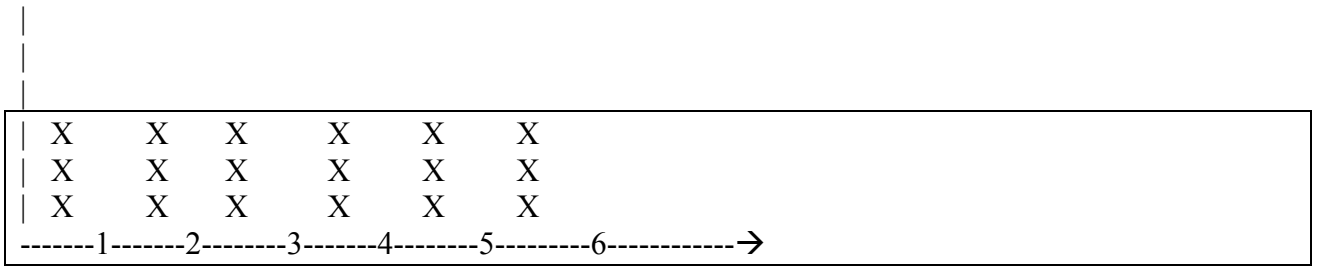
Lanci il dado per esempio 1000 volte; n = numero di lanci ; 1/6 è la speranza matematica della faccia 1 del dado. Poi conto effettivamente le volte in cui è uscita la faccia 1.
x= frequenza di volte in cui esce la faccia 1 (x!=1000/6)
➔ la frequenza tende alla speranza matematica e si avvicina sempre di più man mano che aumentano i lanci.
La media (delle volte in cui esce la faccia 1) è x segnato = (1/n) *{$\sum(i=1;n)[x_i]$}

Teorema del limite centrale:

Quando n è abbastanza grande, **la media si avvicina alla speranza matematica**.
La distribuzione dei numeri casuali ha a che fare con un oggetto detto "**istogramma**"

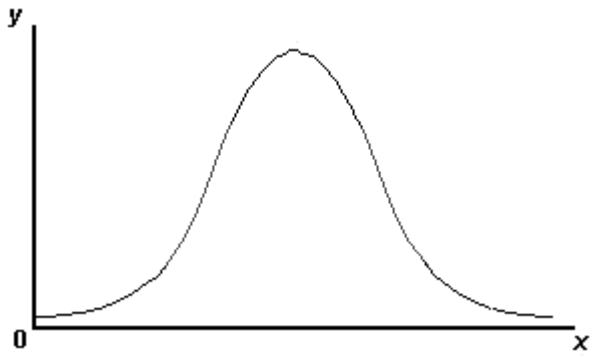


Supponendo che il dado non sia truccato, che la mano che lancia non stia barando, con un numero di lanci elevati ottengo un istogramma con una distribuzione rettangolare:

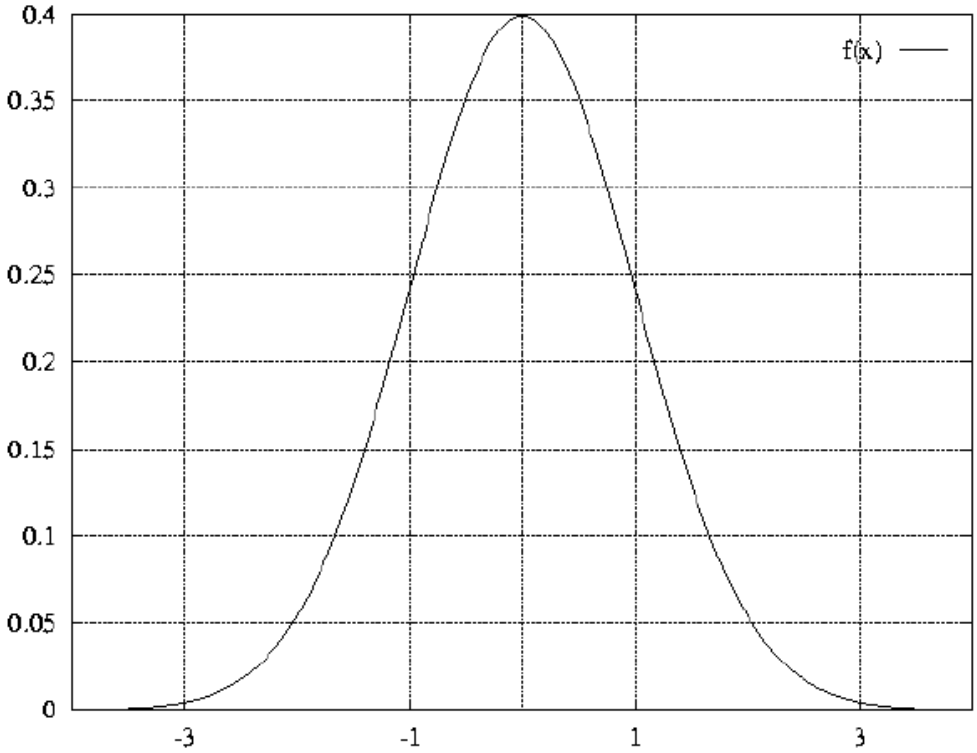


Altro esempio: se conto le radiazioni alfa, gamma e delta emesse da un campione radioattivo con un contatore geiger , la costruzione dell'istogramma diventa la:

Distribuzione di Poisson



Prendo ancora il dado, lo lancio 1000 volte e faccio la media; sarà 2 ½
<X> faccio la distribuzione dei valori medi; se faccio l'istogramma ottengo la curva di Gauss, che è la **distribuzione di probabilità**



L'integrale della curva di Gauss da –infinito a +infinito da valore=1; la funzione è $f(x)=e^{(-x^2)}$

Viene usato anche in elettrotecnica per i numeri complessi $e^{i\pi}$

Oltre al valore medio ho bisogno anche della varianza, che mi da lo scarto del valore medio (sigma).

Riassumendo :

Media – Varianza – Scarto quadratico medio

media = <X>=1/n * sommatoria di Xi con i=1 a n

varianza=sigma=1/(n-1) * $\sum$(Xi-<X>)^2 con i che va da 1 a n = **scarto quadratico medio**

NUMERI CASUALI

Torniamo ora alla generazione di numeri casuali; un calcolatore non è capace di generare numeri casuali, genera però numeri pseudocasuali

Significa che dall'input di un numero ne ottengo un altro. **Se non uso il seme l'output sarà sempre lo stesso** in quanto la libreria ne usa uno di default; quindi per ovviare a questo problema si ricorre ad un seed.

Per cambiarli devo modificare il numero iniziale = generation seed = seme di generazione.
Posso dare per esempio come numero iniziale da quanti millisecondi è acceso il computer, anche se questo non lo so. (ma non importa, l'importante è che il seme sia qualcosa di cui il computer sia al corrente.)

Vediamo ora alcuni esempi:
1) Qual è la probabilità che nell'aula ci siano studenti nati nello stesso giorno; o meglio, quante persone devo avere per avere la probabilità del 50%.
2) in giro ci sono casinò on-line; premettendo che statisticamente il casinò vince, cioè non bara, è stato scritto un programma pseudo-casuale basato sull'inizio del sistema.
Se noi riuscissimo a sapere (tramite una talpa) a che ora è partito il sistema e potessimo analizzare le sequenze di numeri casuali, potremmo ricostruire la generazione random e sapere che numeri usciranno…
Lo hanno fatto sbancando il casinò (waird ?)

Posso usare, se c'è, la **funzione random** e per vedere la sua sintassi uso :
/* per gli utenti windows con DEV++ la random() diventa rand();*/
>man random
Useremo long random(void)
Usare prima #include<stdlib.h>
La funzione ritorna numeri che vanno da 0 a 2^{31-1}

Esempio: fare un programma che stampe 10 volte numeri casuali
(in DEV++ windows la funzione è rand) rand1.c

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
int n;
long x;

for (n=1;n<=10;n++)
{
  printf("%ld\n",random());
}
printf("\n");
return 0;
}
```

Notare che i numeri generati sono sempre gli stessi; questo si è notato su più computers contemporaneamente, mentre su altri è diverso (altri orari di partenza del sistema?)
Per comodità posso avere un range di numeri casuali + ridotto, cioè i numeri fra MIN e MAX devo fare questa operazione:

Esempio con range da 0 a 5: (random3.c)

Per ridurre il range dei numeri casuali generati dovrò uniformarli dividendo i numeri stessi per la differenza di "max-min + 1" dove max= massimo numero consentito, min=minimo numero consentito; proviamo di seguito a verificare la validità …
fra 0 e 2 prendo il resto della divisione per 3 (spiegazione della formula)
fra 0 e MAX prendo il resto della divisione per MAX+1
fra -2 e 2 prendo il resto della divisione fra MAX-MIN+1

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {

int n,min,max;               /*definisco minimo e massimo */
long x;
float somma,medio;
somma=0;
medio=0;
for (n=1;n<=10;n++)
{
  min=1;
  max=6;
  x=(random()%(max-min+1));  /* vedi spiegazione sopra per la formula */

printf("%ld\n",x);
  somma=somma+x;
}
medio=somma/n;
printf("\n");
printf("valore medio %f\n",medio);
return 0;
}
```

Per avere la media devo avere la somma e dividere per 6
Sum = sum + x;
sum+x si può scrivere anche come sum+=x;
Però se divido 2 interi non mi va bene, perché nei risultati ho bisogno dei decimali;
a/b → a e b sono interi oppure sono float

alla fine possiamo mettere il randomizzatore, uso srand(intero senza segno)
per cercare nel manuale di unix posso usare il comando:
>apropos … NomeDaCercare…. Per usare la funzione il seme è **time(void)**
Ovvero **srandom(time(void))**;
NB: per utenti di DEV++ Windows la funzione diventa **srand(time(void))**

Esempio numeri casuali con cambio seme (random4.c)

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {

int n,min,max;
long x;
float somma,medio;
somma=0;
medio=0;
srandom(time(NULL));          // lancio la funzione di cambio "seme"
for (n=1;n<=10;n++)
{
  min=1;
  max=6;
  x=(random()%(max-min+1));
  printf("%ld\n",x);
  somma=somma+x;
}
medio=somma/n;
printf("\n");
printf("valore medio %f\n",medio);
return 0;
}
```

NUMERI REALI AL CALCOLATORE

Numeri Reali R
Esempio di numero reale: $\pm a10^{\pm b}$
Es: $1000 = 10^3 \rightarrow 1E(e)\ 3 = 1*10^3$

Ci sono poi dei numeri nel calcolatore ∞ oppure NaN (Not a Number→ forme indeterminate) 0/0
Nle linguaggio C ci sono due tipi di numeri reali:
float → precisione circa 10^{37} (6-7 cifre decimali)
double→ precisione circa 10^{200}

I calcoli sui floating point sono più complessi e richiedono più elaborazione al calcolatore (+ tempo), mentre con gli interi il calcolo è più veloce.
Nonostante questo per la maggior parte dei calcoli con i calcolatori va bene il floating point.
Il tipo floating point non va bene per i numeri reali, perché in questi ci sono quelli irrazionali (espressi tramite frazione) .
Esempi:
Numeri periodici
1/3 =0,3 periodico infinite cifre , non può essere rappresentato bene da una macchina
0,9 periodico = identica a 1 ==1
In matematica si può controllare se due cose sono uguali verificando che la loro differenza sia = 0.
|Xn-1|<En // E = epsilon
I numeri trascendenti (√3, ∏(pigreco)), ecc…
per esempio non è detto che $\sqrt{2}^2$ con il computer risulti uguale a due.
Con il calcolatore posso avere 2,00000000001 oppure 1,999999999, allora se devo eseguire una uguaglianza anziché assegnare if(a==b) è meglio utilizzare if(a-b<E), Se devo fare un'assegnazione è + corretto scrivere:

```
float a;
a=1.0;
```

che scrivere solamente:

```
float a;
```

Vari tipi di formattazione :
%F float
%lF double float
%g tronca i decimali
Si può anche scrivere %n.mF dove
n=lunghezza totale
m=numero dei decimali (il punto occupa un bit)

Esercizio: (float1.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int main(void) {
int i;
float x,r;

for (i=1;i<=10;i++)
{
  x=(float)i;
  r=(float)sqrt((double)x);
  printf("%f %f\n",x,r*r);
  }
return 0;
}
```

Se proviamo a togliere il link della libreria <math.h> , vediamo che il compilatore non da errore; in ambiente unix il linker è "ld", ed è quello che mi da errore se ci sono delle funzioni che stanno nelle librerie; la libreria matematica sta in un'altra libreria:
devo cioè scrivere nella riga di comando x compilare:
gcc -oprog prog.c –lm dove –lm è l'opzione per usare la "libreria matematica"

Esercizio: (radice2.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int main(void) {
int i;
float x,r;

for (i=2000;i<=2010;i++)
{
  x=(float)i;
  r=(float)sqrt((double)x);
  printf("%.10f %.10f\n",x,r*r);
  }
return 0;
}
```

Esercizio: (radice2a.c) prova senza la dichiarazione di #include <math.h>

Non avrebbe funzionato se non avessi eseguito l'operazione x=(float)i [operazione di casting]

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
int i;
float x,r;
for (i=2000;i<=2010;i++)
{
  x=(float)i;
  r=(float)sqrt((double)x);
  printf("%.10f %.10f\n",x,r*r);
  }
return 0;
}
```

Per vedere gli errori (o avvisi di warning) do il comando di compilazione:

`gcc –oprog prog.c –lm –Wall` **(-Wall mi da i warning della compilazione)**

Funzione printf()

In Unix posso vedere con il manuale:
>man print ➔ funzione printf() (=Print Formatted)
La funzione printf() ha un numero variabile di argomenti (minimo 1) separate da virgola ","
Printf("tipoargomenti in metacaratteri", argomenti) è la stringa di controllo con "meta caratteri"
"Meta caratteri" vuol dire che pur scrivendo dei caratteri questi non vengono stampati direttamente, ma interpretati e producono un output diverso da se stessi.

metacaratteri

\n = vai a capo; nuova linea (line feed)
\t = tabulazione orrizontale
\\ = \ (scrive una barra)(backslash)
\"= doppio apice
\'= singolo apice
\b=backspace (ritorno indietro di un carattere)

Gli argomenti printf(" ",a,b,c,d……) sono variabili, costanti, espressioni, ecc
Per ogni argomento bisogna indicare il corrispondente metacarattere di formattazione; cioè metto tanti metacaratteri quanti sono gli argomenti.

Tipi di formato:

%d = decimale con segno
%x = intero esadecimale senza segno
%o = intero ottale senza segno
%f = valore in virgola mobile
%e = esponenziale
Questi metacaratteri servono per stampare un intero in varie modalità.

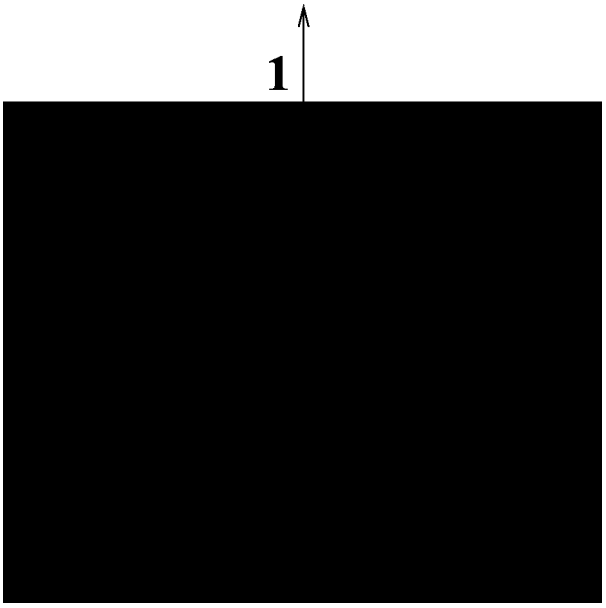
Prefissi tipi di formato:

%ld = long int
%lo = long ottale
%lx = long esadecimale

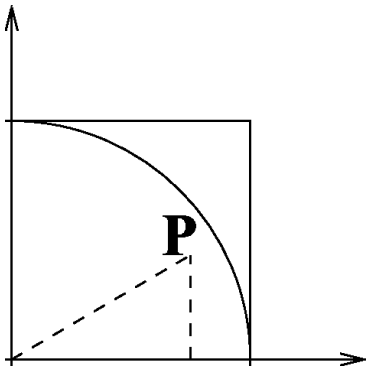
Calcolo valore di PiGreco col metodo di Montecarlo

Cerchiamo di ricavare il valore di π (PiGreco) (metodo di Montecarlo)
In questo esempio vedremo come l'utilizzazione di una particolare tecnica probabilistica, nota come **metodo Montecarlo,** ci permetterà di ottenere in modo molto semplice le prime cifre decimali di π. Questo metodo non permette di dire che le cifre ottenute sono sicuramente esatte, ma solo che sono esatte con una certa probabilità, sia pure molto elevata.

Il metodo Montecarlo si basa su un confronto tra aree. Supponiamo di avere un cerchio di raggio unitario e consideriamo il quadrato ad esso circoscritto (che di conseguenza viene ad avere lato 2) come riportato nella figura



Metodo Montecarlo: disposizione sul piano cartesiano del cerchio e del quadrato.
Supponiamo di prendere un quadrante del cerchio considerato, quello in alto a destra :



Visto che l'area del cerchio intero era π, possiamo dire che l'area di un quadrante vale π/4 e l'area del quadrato che lo inscrive vale 1.
Supponiamo di riempire in modo casuale questa area quadrata, ad esempio con il lancio di freccette; l'area si riempie in modo casuale di punti , parte nella superfice del pezzo di cerchio e parte in quella restante del quadrato.
Chiamo n il numero di lanci effettuati e nc i lanci che sono entrati nel pezzo di cerchio.
Si può trovare π con una proporzione:

$$nc:n = \pi/4:1 \rightarrow \pi = 4nc/n$$

Devo considerare alcune cose:

1) che i punti determinati dal lancio delle freccette sono compresi fra 0 e 1, ovvero messo in formalismo matematico x,y ϵ[0,1] devo avere una funzione pseudocasuale che mi ritorna x,y , sarà la funzione drand48()
2) Come faccio a sapere se le x,y stanno nel cerchio ? x,y stanno nel cerchio se $\sqrt{x^2+y^2}<1$ (ci sarà da discutere sui punti che cadono sulla circonferenza, ma possiamo dire che si possono trascurare, visto che la probabilità che risulti x,y=1 è bassissima.

Esempio: programma per il calcolo di **π**

Il programma seguente realizza il metodo descritto.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/*#define <limits.h>
#define drand48()
((double)rand())/((double)rand_max)*/
// righe inserite x usare con DEV++
int main(void) {
int n,i,nc;
n=0;
nc=0;
float x,y;
for (i=1;i<=1000;i++){
  x=drand48();
  y=drand48();
  if  (sqrt(x*x+y*y)<1)nc++;
  }

  printf("%f\n",4.0*nc/n);
return 0;
}
```

Un approssimazione di π=3,141592

Questo metodi sono detti di Montecarlo, ovvero **Metodi Stocastici**, basati cioè su numeri casuali. Possiamo dire che più grande sarà la misura dei numeri casuali, più preciso sarà il numero che risulterà, però non si può avere la certezza sulla bontà dell'approssimazione.
legge dei grandi numeri, assicura che, al crescere del numero dei punti, la probabilità di avere una cattiva approssimazione è sempre più bassa
Ricordiamo il calcolo degli integrali: praticamente il calcolo dell'integrale col il calcolatore diventa trovare l'area sottesa da una funzione al variare di x; devo trovare le ordinate e calcolare la $\sum$ delle varie aree date dalle figura con ordinata definita da una Δx che col tempo diventa un dx ; si chiama metodo di Newton di soluzione degli integrali: $\sum Ai = \int f(x)dx$

NB: Per poter usare il programma in windows, con DEV++ devo usare:
#define <limits.h>
#define drand48()
((double)rand())/((double)rand_max)

Proviamo ad aumentare il numero decimale aumentando I lanci…
Lo definisco a long se arrivo a 1 milione di lanci; mi pongo il problema, come posso sapere **quanto impiega ad essere eseguito un programma?**
In unix c'è una funzione :
>time *nome_programma* (tempo effettivo usato dalla cpu)

Come è il costrutto if ?

```
If(expr){
	Istruzioni;
} else {
	Istruzioni;
}
```

Esercizio: creare una tabellina numerica (tabella.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int main(void) {
int i,j;

for (i=1;i<=10;i++){
  for (j=1;i<=10;j++){
  printf("%d",i);
  }

  printf("%\n");
return 0;
}
```

Esercizio: fare una matrice di int a[10][10] con i valori che vanno da 0 a 9 (array2.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int main(void) {
int i,j;
int a[10][10];

for (i=0;i<10;i++){
  for (j=0;i<10;j++){
  a[i][j]=(i+1)&(j+1);
  }
  }
  printf("%d",a[i][j]);
  printf("%\n");
return 0;
}
```

FUNZIONI E ARRAY

Si utilizzano le funzioni per ridurre la complessità delle soluzioni dei problemi e concentrarsi così sulla soluzione di un piccolo problema, scomponendo fin che è possibile il problema da risolvere in tanti piccoli problemi = funzioni.
La funzione può essere vista come una scatola nera con un input e un output, senza sapere necessariamente cosa succede all'interno della scatola

Per far sapere come funziona una "funzione" devo comunicare il "**prototipo**" della funzione stessa.
Esempio: la funzione sqrt() = radice quadrata ritorna un numero in formato double, devo comunicare il "prototipo" scrivendo sqrt(double) ; inoltre lo devo anche dire al compilatore informandolo che devo usare la libreria #include<math.h>
Posso scrivere che sqrt(qualcosa) → qualcosa produce un elemento di tipo double.
Ci sono delle piccole eccezioni, certe funzioni possono non avere l'input o l'output.
Per esempio la funzione rand() , non avendo input ma solo output la scrivo → int rand(void);

sqrt()	1 input – 1 output
rand()	no input – 1 output
beep(void)	no input – no output

La funzione beep () (=cicalino) essendo void non la posso usare all'interno della struttura; la scrivo a basta, produce automaticamente l'effetto. Naturalmente non posso scrivere a=beep(), non esiste.
Esercizio:
funzione che somma 2 numeri; avrò 2 input e 1 output
scrivo il prototipo:

```
int     sum(int a, int b) {
        int c;
        c=a+b;
        return c;
}
```

Posso anche scrivere in alternativa

```
int     sum(int a, int b) {
        return a+b;
}
```

La chiamata alla funzione sarà x=sum(5,c);
Le variabili utilizzate dalla funzione sono locali alla funzione stessa; vengono distrutte all'uscita .
In C la varabili utilizzate all'interno sono dette "variabili automatiche"; come detto vengono create e distrutte all'interno della funzione.
Posso anche scrivere per esempio : auto int c;

Esempio: somma di due numeri

```
int main(void) {
int x;
printf(“%f\n”,sum(10,1));
x=sum(3,sum(4,-4));
printf(“%d\n”,x);
return 0;
}
```

Considerando la sequenza nella funzione costruita, posso dichiarare prima le funzioni utilizzate e poi la main(), in modo tale da trovarmi già dichiarate tutte le funzioni perché lette sequenzialmente prima della main.
Posso mettere”*void*” davanti alla funzione o meglio mettere *return 0;* alla fine della main.

Vedi esempio: (funzionesum2.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int sum(int a, int b){
         int c;
       c=a+b;

       return c;
}
int main(void) {
int x;
     x=sum(3,sum(4,-4));
     printf("Risultato = %d\n",x);
     return 0;
}
```

Esempio2: (funzione sum.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int sum(int a, int b){
         int c;
       c=a+b;

       return c;
}
int main(void) {

     printf("Risultato = %d\n",sum(10,-1));
}
```

Esempi su puntatori e array

Utilizzeremo array di interi
Esempio: int x[10] dichiarazione di un array di 10 interi

Esercizio: voglio creare un array di interi che:
1) contiene numeri a caso
2) stamparlo
3) trovare il minimo e massimo dei valori contenuti

Come informazione ho 2 elementi: il primo elemento (=nome array) e per secondo il numero di elementi; allora devo dare 2 argomenti alla funzione.

Esempio: (array2.c) creare un array e riempirlo di zeri

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

void aFill(int a[], int n,int min, int max) {
      int i;
      srand(time(NULL));
      for(i=0;i<n;i++) {
      a[i]=rand()%(max-min+1))+min;
      }
}

void aPrint(int a[], int n){
      printf("%d\n",a[i];
      return ;
}

int main(void) {
    int u[10];
    aFill(u,10,100,200);
    aPrint(u,10);
    return 0;
}
```

Consiglio di come usare i nomi da assegnare:

Nello scrivere il prototipo di una funzione cerco di usare nomi mnemonici che mi permettano in futuro di distinguere fra la varie funzioni quella/e che mi potrà servire ; es aFill = array Filler, ovvero funzione che riempie l'array.

```
Void   aFill(int a[], int n, int min, int max){
       For(i=0;i<n;i++)
       a[i]=(rand()%(max-min+1))+min;

       void aPrint(int[], int n) {
            printf("%d\n",a[i]);

//     trovo ora il + grande
```

aMax(int a[];int n){

NB: è + importante avere l'indice, più del valore del singolo elemento ; ritorneremo allora l'indice :

int aMax(int a[], int n){

Esercizio: come faccio a rovesciare un vettore?
Vettore originario → 1 2 6 9
Vettore rovesciato→ 9 6 2 1

Controllo la funzione vedendo se gli elementi sono pari o dispari; questo perché spostandomi dagli estremi
controlliamo se il nostro array ha numero elementi pari o dispari
Se sono 3 ottengo 0 1
Se sono 4 ottengo 0 2
Devo far scomparire l'elemento i-esimo con n-1-i
Però per poter scambiare 2 cose ho bisogno di una terza cosa, serve una variabile in più che possiamo chiamare "t"

La funzione che mi permette di rovesciare il contenuto di un vettore:

```
void aRev(int a[], int n){
  int i;
  int t;
  for(i=0;i<n/2;i++){
      t=a[i];
      a[i]=a[n-1-i];
      a[n-1-i]=t;
  }
```

Esercizio : fare un programma che inverte il contenuto di un array (arrayinverti.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
// creo array e riempio di zero
void aFill(int a[], int n,int xmin, int xmax) {
      int i;
      srand(time(NULL));
      for(i=0;i<n;i++) {
      a[i]=rand()%((xmax-xmin+1)+xmin);
      }
}
// stampo l'array
void aPrint(int a[], int n)
{
     int i;
     for(i=1;i<n;i++){
     printf("%d\n",a[i]);
}
      return ;
}
// trovo il massimo
int aMax(int a[],int n)
{
     int i;
     int xmax=0;
     for(i=1;i<n;i++){
     if(a[i]>a[xmax]) xmax=i;
     }
     return xmax;
}
// trovo il minimo
int aMin(int a[],int n)
{
     int i;
     int xmin=0;
     for(i=1;i<n;i++){
     if(a[i]<a[xmin]) xmin=i;
     }
     return xmin;
}
// rovescio un vettore
void aRev(int a[], int n){
  int i;
  int t;

  for(i=0;i<n/2;i++){
      t=a[i];
      a[i]=a[n-1-i];
      a[n-1-i]=t;
  }
}
// funzione main
int main(void) {
    int u[10];
    aFill(u,10,100,200);
    aPrint(u,10);
    printf("Minimo-> %d\n",u[aMin(u,10)]);
    printf("Maximo-> %d\n",u[aMax(u,10)]);
    return 0;
}
```

Esempio: fare la media dei valori di un vettore:

```
float Ave(int a[],int n) {
int a;
float sum=0.0;
for(i=0;i<n;i++){
    sum+=(float) a[i];
}
Return(sum/n);
}
```

Esercizio: generare un mazzo di carte

Per gestire un mazzo di carte dovrò dichiarare un vettore di 40
Int am[40]

```
int Amazzo(int am[],int n){
int i;
   for(i=0;i<n;i++){
   am[i]=i;
   }
}
```

Esercizio: mescolare un mazzo di carte (mescola.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

// funzione che riempie casualmente il vettore
void aFill(int a[], int n,int xmin, int xmax) {
      int i;
      srand(time(NULL));
      for(i=0;i<n;i++) {
      a[i]=rand()%((xmax-xmin+1)+xmin);
      }
}
// funzione che stampa il contenuto del vettore
void aPrint(int a[], int n)
 {
     int i;
     for(i=1;i<n;i++){
     printf("%d\n",a[i]);
}
      return ;
}


int Amazzo(int am[],int n){
int i;
   for(i=0;i<n;i++){
   am[i]=i;
   }
}



int main(void) {
    int u[10];
    aFill(u,10,100,200);
    aPrint(u,10);
    aPrint(Amazzo,40);
    int x;
    int i;
    int n;
    int t;
    for(i=0;i<n;i++){
       x=rand()%(40-i)+i;
       t=u[i];
       u[i]=u[x];
       u[x]=t;
   }
   aprint(Amazzo,40);
   return 0;
}
```

Esercizio: fare un istogramma visivo del lancio di un dado (istogramma.c)

Il programma simula il lancio del dado e devo far vedere con un istogramma le probabilità di uscita delle 6 facce del dado, modificando per alcune volte il numero di lanci simulati.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>


void aFill(int a[], int n,int xmin, int xmax) {
      int i;
      srand(time(NULL));
      for(i=0;i<n;i++) {
      a[i]=rand()%((xmax-xmin+1)+xmin);
      }

}

void aPrint(int a[], int n)
 {
     int i;
     for(i=1;i<n;i++){
     printf("%d\n",a[i]);
}
      return ;
}


int Amazzo(int am[],int n){
int i;
   for(i=0;i<n;i++){
   am[i]=i;
   }
}

int azero(int a[],int n){
    int i;
    for(i=0;i<n;i++){
    }
}

int main(void) {
    int matrix[1000],isto[6];
    aFill(m,1000,0,5);
    azero(isto,6);
    for(i=0;i<10000;i++)
    {
       isto[matrix[i]]++;
          printf("%d",j);
    }
      for(i=0;i<6;i++){
      {
        printf("%d) ",i+1);
      for (j=0;(j<isto[i]/100;j++)
     {
       printf("\n");
     }


}
```

ESEMPI VARI DI PROGRAMMAZIONE

Oggi faremo degli esempi di programmazione, semplici, quasi ridicoli, utilizzando il linguaggio c.
Gli esempi sono semplici, ma bisogna capire i concetti che ci sono sotto.
Esempio: facciamo una funzione che fa da contatore; cioè un programma che fa ogni tanto un beep e alla fine conta i beep emessi.
In input non avrò nessun parametro e in uscita un contatore.

```
int cont(void){
    int c:
    return c++;
}
```

La funziona fatta così non può funzionare, scriverò allora:

Esempio: contatore

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int count(void){
    int c;
    return c++;
}


int main (void){
int i;
    for(i=0;i<10;i++){
        printf("%d\n",count());
}
return 0;
}
```

La memoria del calcolatore è divisa in 2 categorie:
heap (=mucchio) e stack (=pila)

heap stack←--------→
|----------|----------|
Qui sono allocate tutte le variabili locali

Il valore di C parte da un numero (es: 1000 o 2010) poi scrivo +1 e diventa 2010, 2011; questo perché nessuno va ad occupare una zona dichiarata.
Questo succede perché una variabile non inizializzata ha valore zero.

Allora dalla funzione di tipo:

```
int count(void){
    int c=0;
    return c++;
}
```

Diventa del tipo:

```
int count(void){
    static int c=0;
    return c++;
//      oppure
//   return ++c;
}
```

Variabili statiche

Utilizzeremo una "variabile statica" che rimane messa in una allocazione fissa.
Le variabile statiche vanno usate con parsimonia. Se io vendo le funzioni che invento do anche il codice sorgente; cosa vuol dire dare il codice sorgente?
Esempio: vediamo la funzione di prima: devo dare anche i prototipi detti .h

Esempio: **(funzioni_v.c)**

```
int count(void){
   static int c=0;
   return c++;
}
```

Devo aggiungere anche (**funzioni_v.h)**

```
int count(void);
```

Avrò il mio programma **(cont.c)**
dove ho dovuto aggiungere la riga #include "funzioni_v.h"

```
#include "funzioni_v.h"

int main (void){
int i;
   for(i=0;i<10;i++){
      printf("%d\n",count());
         }
return 0;
}
// usata dichiarazione di include di funzioni_v.h
```

In unix vengono gestiti prima i comando con # = pre-processing
Se lancio la compilazione con gcc –E cont.c avrò degli errori, devo usare il comando di compilazione :
gcc –c funzioni_v.c → mi genera funzioni_v.o (object)
gcc –c cont.c → genera cont.o

compilazione e link di più oggetti

Ora devo fare la 3^ compilazione, fare cioè il link di più oggetti:

gscc –onome oggetto1.o oggetto2.o oggetto3.o ecc dove tutti gli oggetti elencati di seguito vengono linkati, cioè assemblati in un unico file.

Questo è anche il metodo da utilizzare per non dover ricompilare tutte le volte tutti gli oggetti (funzioni) ma solo linkarli fra loro.
Posso avere un sistema di compilazione solo dei file modificati ; in unix c'è un software che si chiame make che mi aiuta a ricompilare solo i files modificati.
NB:In ambiente windows (per esempio con DEV++) bisogna creare un progetto e dare un "empty project" e dire che è in C e non in C++.
In un pannello di sinistra posso aggiungere i files che utilizzo; il nome lo posso dare in base all'eseguibile e il DEV++ mi crea un "make file" x la compilazione in automatico.
Il progetto finale è meglio sia fatto da più file.

Esempio: Devo simulare in qualche modo il flusso dei fogli di una stampante, cioè per ogni foglio che entra nella stampante dal cassetto della carta ne esce un altro stampato; presuppongo che la stampante non si inceppi e che continuo a rifornire il cassetto di fogli.
Allora avrò una funzione che aggiunge 1 e una funzione che toglie 1
Farò allora 2 funzioni:
la **c_inc()** che incrementa di 1 e la **c_dec()** che decrementa di 1:

```
// inc=increment
int c_inc(void){
    Count++;
}
// dec=decrement
int c_dec(void){
    Count--;
}
```

Però se io dichiaro in due funzioni la stessa variabile (Count) NON sono la stessa variabile, perchè le variabili esistono solo nell'ambito della funzione in cui sono dichiarate e poi vengono distrutte.
Non posso avere il risultato voluto con le due funzioni così fatte ; dovrò allora dichiarare STATIC la variabile che uso:

```
// inc=increment
int c_inc(void){
    static inc;
     Count++;
}
// dec=decrement
int c_dec(void){
    Count--;
}
```

→ static
Il tipo di variabili → auto STORAGE CLASS
→ register

La mia variabile sta nella static (heat stack)

Scope : local = create all'interno di una funzione sono locali e automatiche
Global = sono dichiarate all'esterno delle funzioni

Global, static ← int count;
int inc(void){
local, auto ← int i;
non ha valore }
iniziale

int dec(void){
local, static ← static int j;
ha valore count--;
iniziale = 0 }

Le variabili globali si usano con parsimonia e si tende a dichiararle in cima al file.
E' buona norma scriverle in modo diverso, esempio con la prima lettera maiuscola, ad esempio int Count; ora riscriviamo meglio il nostro programma; cerchiamo di nascondere praticamente i dettagli dell'applicazione.

Costruiamo un contatore

La funzione c_get() mi permette di leggere un contatore

```
int c_get(void){
    return Count;
}
```

Mi serve anche un'altra funzione c_set() per poter scrivere un valore all'interno della sua variabile:

```
void c_set(int v){
    Count=v;
}
```

Poi anche una funzione c_reset() per mettere a zero la funzione:

```
void c_reset(void){
    c_set(0);
}
```

Nello sviluppo del programma voglio anche implementare la funzione di stampa c_print():

```
int c_print(void){
    printf("%d\n",Count);
}
```

Se io linko un programma con queste funzioni, Count, inc,dec,set,reset,print, sono globali; chiamo questo set di funzioni con **cont.c**
Chiamo le funzioni in questo modo:

Sono tutti simboli globali

dec( ) → c_dec (decrementa) inc( ) → c_inc (incrementa) set( ) → c_set (assegna valore) reset( ) → c_reset(azzerra) print( ) → c_print (stampa)

Sono globali, sia la variabile , sia la funzione.
Per esempio la funzione print() non voglio che si veda e neanche count().
Ho un file count.c e anche count.h contenente i prototipi.
Creo anche il programma main.c che si linka con count.h le funzoni create precedentemente.
Compilo main.c e funz_count.c
Provo ora a mettere in main.c cont=88; mi da variabile non dichiarata; ed è giusto che sia così perché devo andare a modificare la variabile in un file esterno con :
extern int count; (dichiarazione che serve nella main)
provo ora a metere "static int Count"; la parola "static" messa davanti ad una cosa globale la rende "riservata", cioè globale solo per quel file, invece static davanti la vedo anche fuori.

Variabili pubbliche e private

Per capire il concetto fra variabili pubbliche e private facciamo un esempio: se sono al ristorante posso ordinare gli spaghetti, non posso cuocere gli spaghetti, questo è delegato a qualcun altro (cuoco); posso solo ordinare al cuoco di cuocere gli spaghetti.
Percui certe funzioni sono private , altre sono pubbliche.
Tendenzialmente non si usano mai variabili pubbliche, se ne voglio una accessibile all'esterno uso un formalismo perticolare per non aveve omonimia o sovrapposizione , es: int __Count__
Se devo fare delle nuove implementazioni io dove "proteggere" il mio codice per poter anche mantenere la compatibilità delle funzioni che uso anche all'indietro, ovvero per versioni precedenti.
Ricordarsi del concetto di static nei due casi:
1) static davanti a una variabile locale → è nella heap della memoria
2) static davanti a un simbolo o funzione → privata nel singolo file

Vediamo la simbologia utilizzata nello scrivere i programmi:

#include ← è come scrivere dentro il programma il codice corrispondente

#include <……> dichiaro funzioni e/o librerie che stanno nella dir di sistema
#include "……" dichiaro funzioni che sono nella directory corrente

Per controllare nel sistema si possono usare i comandi “cat”, “more” e “less”
>cat /usr/include/math.h

Possiamo veder anche l’uso di define nel SO:
#define VMAJ
#define VMIN

Supponiamo di avere la necessità di utilizzare un valore molte volte, per esempio dover calcolare √3 tante volte, lo posso definire con double R3=sqrt(3.0);
In questo modo però utilizzo memoria ogni volta che eseguo il calcolo della radice quadrata, posso allora utilizzare questo modo:
#define R3 1,71 ← con questo formalismo il sistema assegna a R3 il valore di 1,71 e lo restituisce al programma senza fare ricalcoli e di conseguenza senza utilizzare memoria.

Atro esempio di scrivere il define# è:
#define BEGIN{
#define END }

```
if (A) begin
        I=1;
end
```

Ci sono dei modi d programmazione incomprensibili, nel senso che il significato standard delle varie istruzioni è sostituito da questo define# ; ci sono modi di programmazione tutto su una linea (oneliner) e occc (obfuskedcourse)

Con il define posso dichiarare il numero di elementi di un array in questo modo:

```
#define NA 10
Int a[NA];
for(i=o;i<NA;…){
}
```
Il preprocessore non centra con il linguaggio C, è staccato completamente.

PUNTATORI

I puntatori sono la parte più importante della programmazione in C, quella che permette di lavorare "a basso livello" (cioè agendo su singole istruzioni del processore), mantenendo però una praticità unica nel suo genere.
Cosa è un puntatore? **Un puntatore è una variabile che contiene l'indirizzo di memoria di un'altra variabile**. Quando dichiariamo una variabile, a questa verrà riservato un indirizzo di memoria, ad esempio la posizione 1000; un puntatore contiene, appunto, l'indirizzo di tale variabile (quindi il valore 1000). L'importanza risiede nel fatto che si possono manipolare sia il puntatore che la variabile puntata (cioè la variabile memorizzata a quell'indirizzo di memoria).
Per definire un puntatore è necessario seguire la seguente sintassi:

```
// variabile normale
int variabile;

// puntatore
int *puntatore;
```

L'asterisco (*) viene chiamato **operatore di indirezione o deferenziazione** e restituisce il contenuto dell'oggetto puntato dal puntatore; mentre l'operatore **&** restituisce l'indirizzo della variabile e va usato nella seguente forma:

```
// assegno al puntatore l'indirizzo di variabile
puntatore = &variabile;
```

Esempio:
a[i] = [j];
*(a+1) = *(a+j);

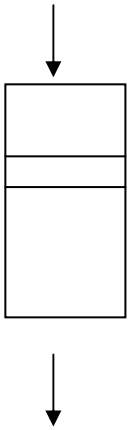
Esempio: (puntatori.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>


int main (void){
static int a[]={1,2};
int i=0;
int j=1;
  *(a+i) = *(a+j);
     printf("%d\n",a[0]);
     return 0;
}
```

Facciamo alcune considerazioni sulla formalità e significato delle assegnazioni in programmazione. Se scrivo x=x; o meglio ancora x=x+1; il valore di destra è uguale ad un valore numerico, quello di sinistra (--value) è interpretato come indirizzo, cioè dove il programma andrà a depositare il valore calcolata dalla parte destra (r-value).

l-value → x = x + 1 ; ← r-value

Come si differenziano i puntatori dagli altri tipi di variabili?
vediamo graficamente come si comporta la memoria col variare del puntatore a:
essendo un intero avrò una dimensione di 4 bit , per cui la locazione da 1000 aumentandola di 1 andrà a 1004:



"a" è sinonimo per indirizzo di memoria ; 1000 è un numero di unità di misura byte , cioè vale 1000 byte 1004 vale 1000+1 , cioò 1000 + 1 oggetti , però per sapere quanto vale un "dopo" devo sapere quanto è "lungo " un puntatore; devo sapere di cosa è fatto il puntatore, lo posso sapere con:
Se a è un intero **a+1←→a+1* sizeof(int)**
Allora posso anche dire che se a è un puntatore e b è un puntatore sono uguali se puntano allo stesso elemento.

casting

Non posso però mescolare puntatori fra di loro se non faccio le operazioni di CASTING.

Esempio (puntatatori2.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main (void){
static int a[]={128000,2};
char * c;   ←------------------------ puntatore di tipo carattere
     printf("%x\n",a[0]);
     printf("%p\n",a);
     return 0;
}
```

Il risultato può essere del tipo 1f400
Ora proviamo a stampare a col formato /p = formato puntatori

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int main (void){
static int a[]={128000,2};
char * c;
c=(char *)a;
     printf("%x\n",a[0]);
     printf("%p\n",a);
     return 0;
}
```

Mi da errore , allora eseguo il programma scritto così

Esempio:(puntatori4.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main (void){
static int a[]={128000,2};
char * c;
c=(char *)a;
     printf("%x\n",*a);
     printf("%p\n",a);
     printf("%p\n",c);
     printf("%x\n",*c);
     return 0;
}
```

Il programma mi da come risultato che *c = 0

A questo punto posso vedere la locazione di memoria successiva, usando cioè *(c+2):

Esempio (puntatori5.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main (void){
static int a[]={128000,2};
char * c;
c=(char *)a;
     printf("%x\n",*a);
     printf("%p\n",a);
     printf("%p\n",c);
     printf("%x\n",*c);
     printf("%x\n",*(c+1));
     printf("%x\n",*(c+2));
     return 0;
}
```

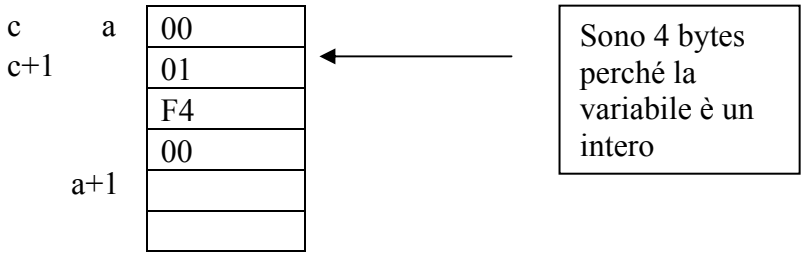
Come risultati posso ottenere:
1f400
20954
20954
0
ffffff4

Altra variazione : (puntatori6.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main (void){
static int a[]={128000,2};
char * c;
c=(char *)a;
     printf("%x\n",*a);
     printf("%p\n",a);
     printf("%p\n",c);
     printf("%2x\n",*c);
     printf("%2x\n",*(c+1));
     printf("%2x\n",*(c+2));
     printf("%x\n",(char)*(c+2));
     return 0;
}
```

Ottengo come risultati:
1f400
209b0
209b0
0
1
ffffff4
ffffff4

vediamo come è la situazione della memoria con il seguente schema:



Esempio: (puntatore7.c) provo lo stesso esempio assegnando a[]= 0x44ef1aF0,2}

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>


int main (void){
static unsigned int a[]={0x44ef1aF0,2};
unsigned char * c;
c=(unsigned char *)a;
     printf("%p\n",a);
     printf("%p\n",a+1);
     printf("%p\n",c);
     printf("%p\n",c+1);
     return 0;
}
```

2091c a
20920 a+1 diferenza =4

2091c c
2091d c+1 diferenza =4

c	44
c+1	ef
c+2	1a
	F0

Proviamo ora ad assegnare:
c=(unsigned char *)a;
c+ = 2;

Come posso portare in stampa questo?

*c	→	1a
c[0]	→	1a
c[1]	→	F0
*(c-1)	→	ef
c[-1]	→	ef
(-1)[c]	→	ef

Posso anche scrivere a[i] <-> *(a+1) <-> *(i+a) == > a[i] = i[a]

Vediamo ora l'esempio asegnando a[]={0x41424344,2};(puntatori8.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main (void){
static unsigned int a[]={0x41424344,2};
unsigned char * c;
c=(unsigned char *)a;
     printf("%p\n",a);
     printf("%p\n",a+1);
     printf("%p\n",c);
     printf("%p\n",c+1);
     printf("======\n");
     c+=2;
     printf("%c\n",*c);
     printf("%c\n",c[0]);
     printf("%c\n",c[1]);
     printf("%c\n",*(c-1));
     printf("%c\n",c[-1]);
     printf("%c\n",(-1)[c]);

     return 0;
}
```

Nota: per vedere la codifica ascii basta eseguire il comando unix:
>man ascii

Proviamo ora a scrivere :

```
void zero(int x){
x=0;
}
```

Il programma non può funzionare, perché quando faccio zero(a) valuto a che è uguale =1
Posso scrivere allora:

```
void zero(int * x){
*x=0;
}
```

Questo perché se a=1 → zero(&a) è corretto

MANIPOLAZIONE DI STRINGHE

Vediamo ora alcuni esempi di funzioni che usano i puntatori.
Fin'ora abbiamo usato valori semplici; si può usare il formalismo degli array o a quelli dei puntatori. Usiamo esempi di stringa di caratteri "ciao"; posso dichiarare la stringa in 2 modi:
char c[]="ciao";
char c[]={'c','i','a','o',0} nb: alla fine ho il carattere 0=zero che vuol dire fine stringa caratteri come vediamo sono 4 caratteri ma sono in realtà 5 numeri, perché lo standard di una stringa finisce con 0 (=zero) ;posso anche dichiarare la stringa con:
char * c = ……
In questo modo posso anche scrivere c=c+1 nei modi precedenti non avrei potuto perché mi avrebbe dato errore in quanto il nome dell'array c[ ] è COSTANTE.

Esempi di stringhe e relativa gestione

Facciamo alcuni esempi di stringhe e di funzioni per gestirle :

Per calcolare la lunghezza di una stringa uso la funzione mystrlen(char *s)
Provo altre funzioni del tipo :
mystrup() → funzione per mettere maiuscole
mystrlow() → funzione che mette minuscole

Esempio: stringa2.c) contiene varie versioni delle funzioni sopra citate :

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

// prima versione della funzione mystrlen()
int mystrlen(char *s){
int i=0;
while (*s !=0){
     i++;
     s++;
     }
 return i;
}
// seconda versione della funzione mystrlen()

int mystrlen2(char *s){
int i=0;
while (*s++){
     i++;
     }
 return i;
}


// terza versione della funzione mystrlen()

int mystrlen3(char *s){
   char *p=s;
   while (*s++);
   return s-p-1;
}

void mystrtoup(char *s){
```

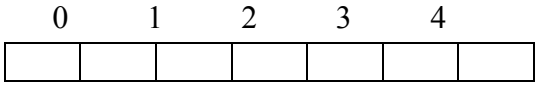
```
   while (*s){
   if(*s<='z'&& *s>='a'){
   *s-=' ';
   }
   s++;
   }
   }


void mystrtolow(char *s){
   while (*s){
   if(*s<='Z'&& *s>='A'){
   *s+=' ';
   }
   s++;
   }
   }



// funzione che mette lettere a caso in una stringa
void mystrrnd(char *s, int n){
int i;
for (i=0;i<n-1;i++){
   s[i]=(random()%('z'-'a'+1))+'a';
   }
s[n-1]=0;
}


// funzione che scambia il contenuto di una stringa
void mystrrev(char *s){
char t;
char *p=s+mystrlen(s)-1;
while(p>s){
t=*p;
*p=*s;
*s=t;
s++;
p--;
}
}



int main (void){
 char str[]="ciao a te";
// char c[]={'c','i','a','0','0'}
     printf("%s\n",str);
     printf("%d\n",mystrlen(str));
     str[4]=0;
     printf("%s\n",str);
     printf("%d\n",mystrlen(str));

     str[4]=' ';
     printf("%d\n",mystrlen2(str));
     str[4]=0;
     printf("%s\n",str);
     printf("%d\n",mystrlen2(str));

     str[4]=' ';
     printf("%d\n",mystrlen3(str));
     str[4]=0;
     printf("%s\n",str);
```

```
     printf("%d\n",mystrlen3(str));

// stampo la stringa maiuscola

     str[4]=' ';
     printf("%d\n",mystrlen3(str));
     str[4]=0;
     mystrtoup(str);
     printf("%s\n",(str));
     printf("%d\n",mystrlen3(str));

// stampo la stringa minuscola

     str[4]=' ';
     printf("%d\n",mystrlen3(str));
     str[4]=0;
     mystrtolow(str);
     printf("%s\n",(str));
     printf("%d\n",mystrlen3(str));

// riempio la stringa di caratteri casuali
     char q[30];
     mystrrnd(q,30);
     printf("%s\n",(str));
     return 0;
}
```

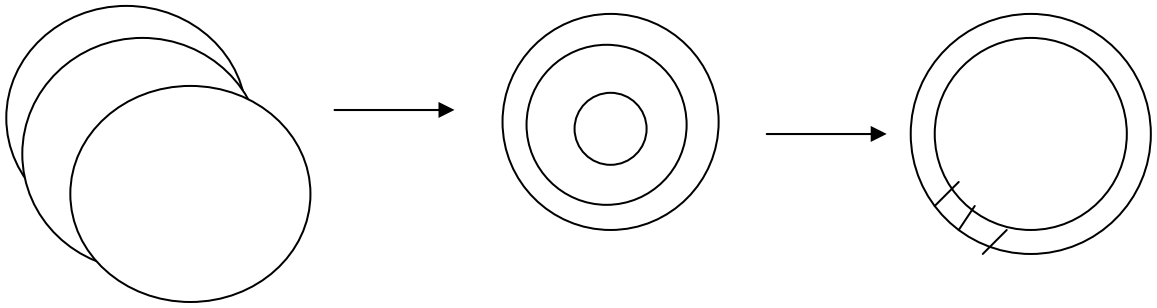
Esempio di applicazione: una calcolatrice

Premessa: proviamo oggi a fare un progetto e arrivare alla fine.
Un programma è un insieme di algoritmi e di strutture dei dati.
Per struttura dei dati non si intende la struct del C, ma i modi di memorizzare i dati.
A seconda del problema abbiamo struttura e organizzazione dei dati che meglio di adatta al tipo di problema che voglio risolvere.

Per esempio posso utlizzare un array:

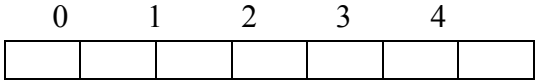


Se utilizzo un array il tempo di accesso a qualsiasi dei suoi dati è identico
Posso anche utilizzare l'esempio di un hard disk (=disco rigido o disco fisso); questo si può scomporre in più parti:



Avrò più dischi (cilindri) Avrò per ognuno Più corone tanti bytes in ogni corona

Questa struttura hardware assomiglia dopo la sua scomposizione ad un array di 3 dimensioni.
Naturalmente l'hardisk è una struttura hardware, l'array invece è una struttura concettuale(= struttura astratta).
Dipende dal linguaggio che si usa il modo che si fa la struttura.

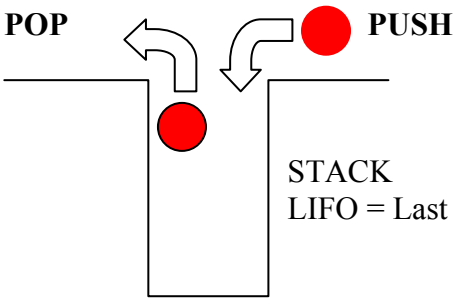
L'array → 

lo risolvo con int a[15];

Quando si deve pensare ad un nuovo programma, la prima fase di pensiero è su come organizzare i dati. L'array è una struttura dati che presenta vantaggi e svantaggi.
Vantaggi = il tempo di accesso è uguale per ogni elemento
Svantaggio = è complicato gestirlo , l'inserimenti di un dato in mezzo comporta lo shift a destra

Stack

Oggi implementiamo il concetto di stack (=catasta) Lo stack può essere visto come un pozzo o tubo verticale dove vengono inseriti i valori con l'operazione di push e estratti con quella di pop.



STACK ← Il concetto visivo di stack è un pozzo dove metto e tolgo
LIFO = Last In First Out

Ne consegue che l'utilizzo degli array è conveniente con strutture di dati statici e non dinamici. Praticamente l'idea è quella del pozzo dove con PUSH metto una pallina (dato) e con POP tolgo la pallina.

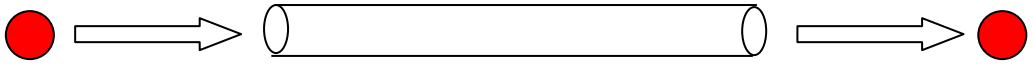
LIFO

La tipologia di funzionamento dello stack è di tipo LIFO , ovvero Last In (l'ultimo entrato) = First Out (Primo uscito) ; una analogia con la realtà può essere l'immagine di qualcuno che sta caricando il cassone un camion, dove l'ultima merce caricata è la prima ad essere scaricata.

Coda

Analogamente introduciamo anche il concetto di Queue = Coda

FIFO



Utilizziamo allora la queue per simulare delle situazioni di coda, classica la coda stradale in un tunnel , dove il concetto è il FIFO, = First In First Out, cioè il primo entrato è il primo a uscire.

Vediamo ora di fare un programma che simula una calcolatrice utilizzando il concetto di stack..
Noi siamo abituati ad utilizzare e a scrivere le formule matematiche per eseguire le varie operazioni con l'annotazione algebrica classica (uso di operatori e parentesi) es: 5x(2+(4/2))
Se voglio implementare una calcolatrice, cioè mettere in linea un programma, che controlla ed esegue i vari input che riceverà.
Non posso però spostarmi da destra e sinistra, rispettare le parentesi, ecc
Servirebbero le "rotazioni ad albero", ma lo faremo più aventi.

RPN = notazione inversa polacca

Utilizzeremo allora una tecnica detta RPN = Reverse Polish Notation = Notazione inversa polacca

Notazione polacca inversa

La notazione polacca inversa (reverse polish notation - RPN) è una sintassi utilizzata per le formule matematiche ed è dovuta a Jan Lukasiewicz, 1958.

Con la RPN è possibile effettuare qualsiasi tipo di operazione, con il vantaggio di eliminare le problematiche dovute alle parentesi e alla precedenza degli operatori (prima la divisione, poi l'addizione, ecc...). Alcune calcolatrici professionali (HP21-HP25) utilizzavano la RPN in quanto evita l'annotazione di risultati intermedi durante le operazioni; ancora oggi ne troviamo sul mercato.

Nella notazione polacca inversa, detta anche **notazione postfissa** in contrasto con la normale **notazione infissa**, prima si inseriscono gli operandi e poi gli operatori: un esempio di RPN è 3 2 + che equivale al classico 3+2, oppure 10 2 / che fornisce 5.

Quando si utilizza la RPN si fa conto di possedere una pila (stack) su cui pian piano si accumulano gli operandi: prima si impila il 3, poi il 2. Un operatore invece preleva dalla cima della pila tutti gli operandi di cui ha bisogno, esegue l'operazione, e vi rideposita il risultato. L'elemento più in basso è da considerarsi sempre l'operando sinistro. Se l'espressione completa e' corretta, alla fine di tutte le operazioni sulla pila si avra' un solo elemento, il risultato finale.

Questa pila permette, come gia' detto, di evitare l'utilizzo di parentesi per prioritizzare le operazioni, basta inserire nella parte sinistra della formula tutti gli operandi delle operazioni a parentesizzazione più esterna, al centro le operazioni più elementari, alla destra tutti gli operatori di combinazioni dei risultati delle operazioni centrali con gli operandi già presenti. Esistono infatti algoritmi di conversione sia dalla notazione infissa a quella postfissa che viceversa. Come si può notare, la RPN è facilmente implementabile sui Computer.

Un esempio: (10 * 2) + 5 → 5 10 2 * +

Prima della moltiplicazione sono presenti sulla pila 5, 10, 2. Il "*" recupera i primi due elementi (10, 2) li moltiplica e modifica la pila in modo che contenga 5, 20. L'operazione "+" addiziona 5 e 20, ora presenti nella pila, sostituendoli con il risultato: 25.

Altri esempi più complessi: ((10 * 2) + (4 - 5)) / 2 → 10 2 * 4 5 - + 2 /

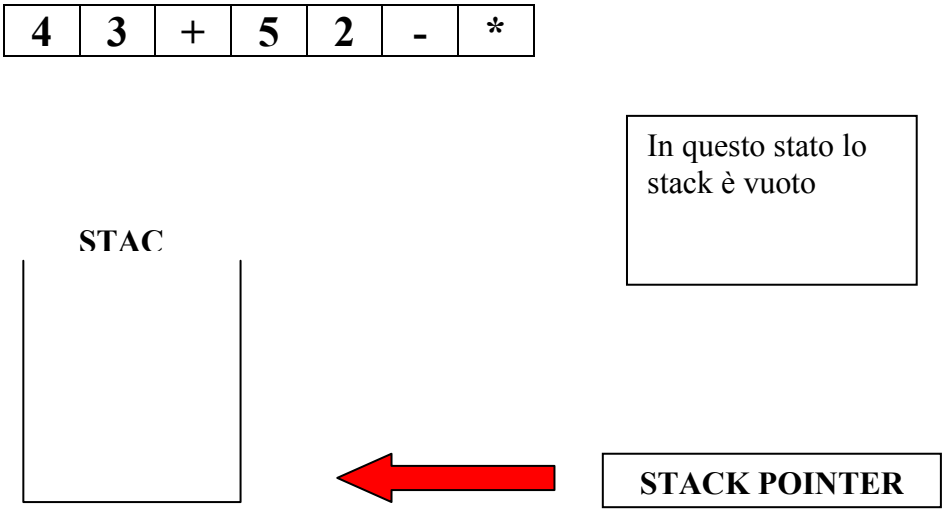
(7 / 3) / ((1 - 4) * 2) + 1 → 1 7 3 / 1 4 - 2 * / + oppure 7 3 / 1 4 - 2 * / 1 +

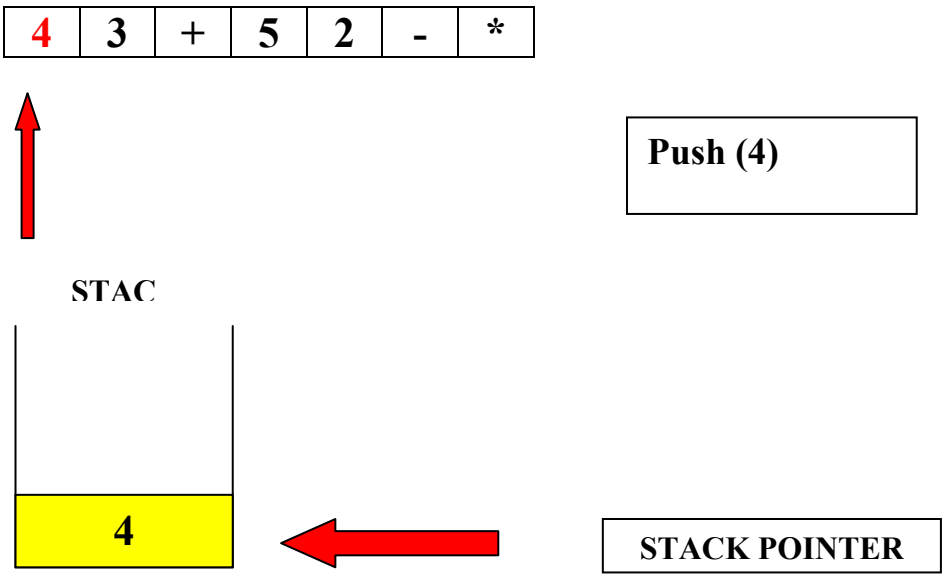
La notazione polacca inversa prende spunto dalla notazione polacca semplice, in cui gli operatori vengono posti prima degli operandi (quindi: + 1 2 invece di 1 2 +), ma solo la prima è facilmente implementabile in modo elettronico o via software, ed è quindi diventata ben più conosciuta.

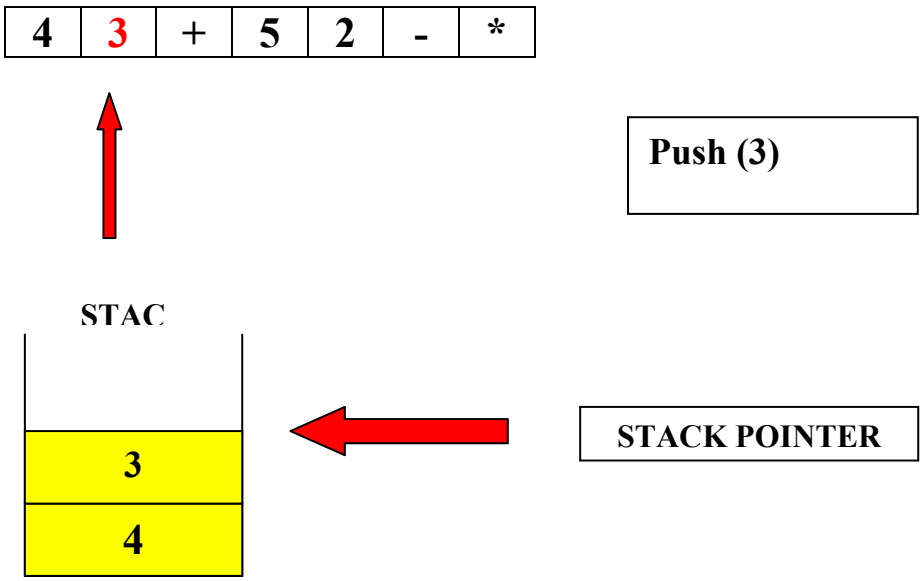
Esempio di calcolo RPN

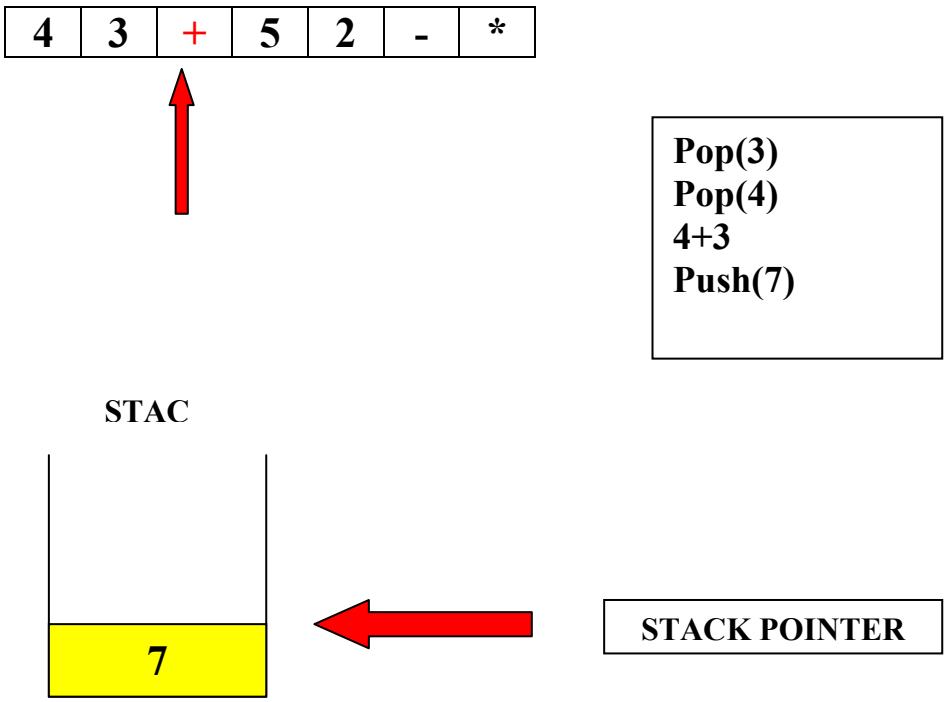
Vediamo ora una sequenza di slide con un esempio di calcolo con RPN:
L'espressione tradizionale da calcolare è (4+3)*(5-2)=21

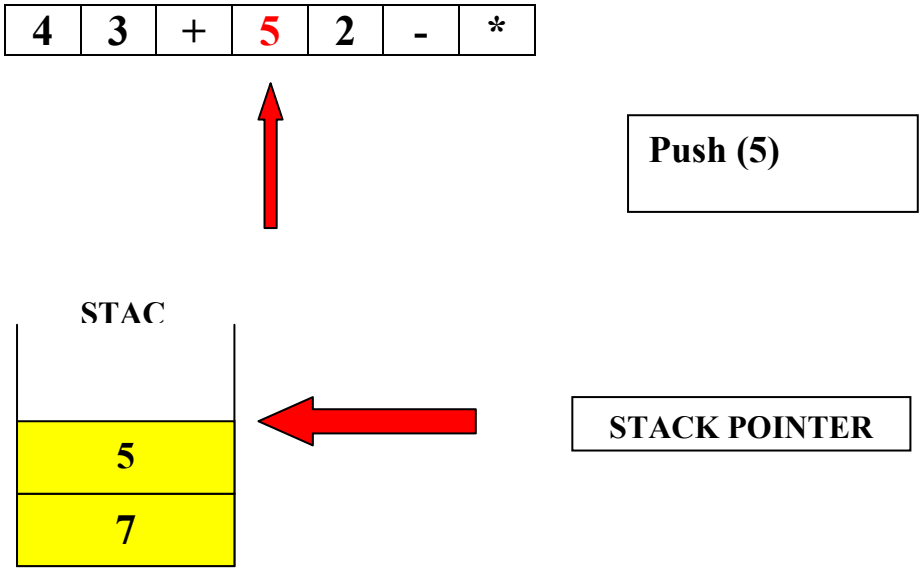
L'espressione tradotta in RPN da calcolare è 4 3 + 5 2 - *, ovvero :

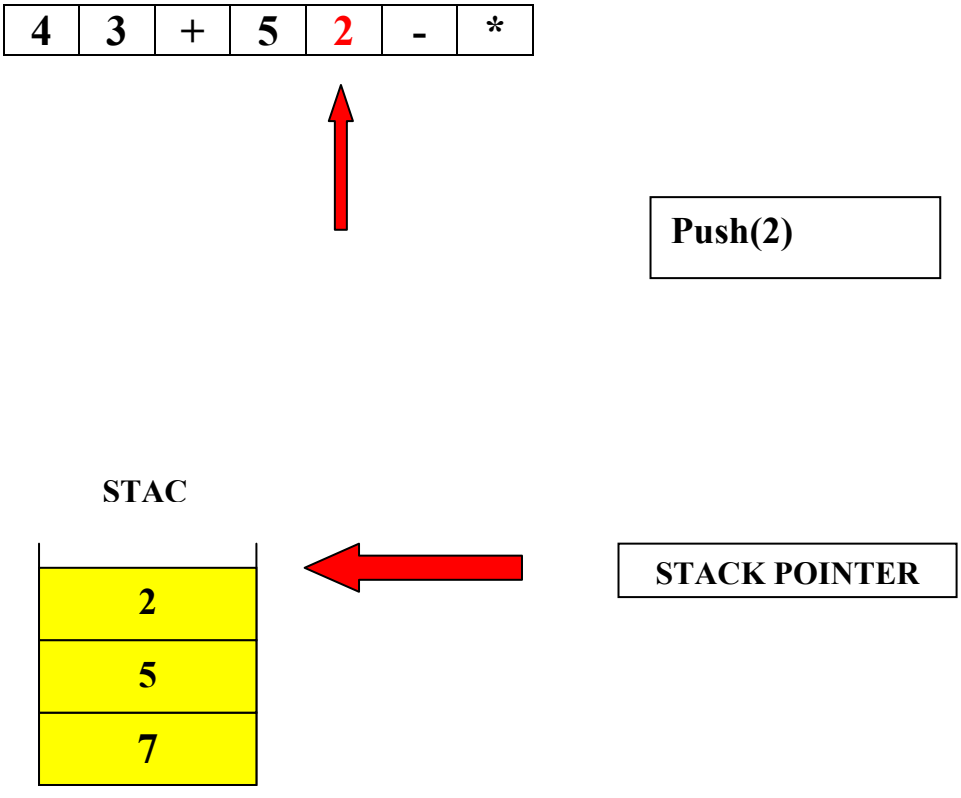


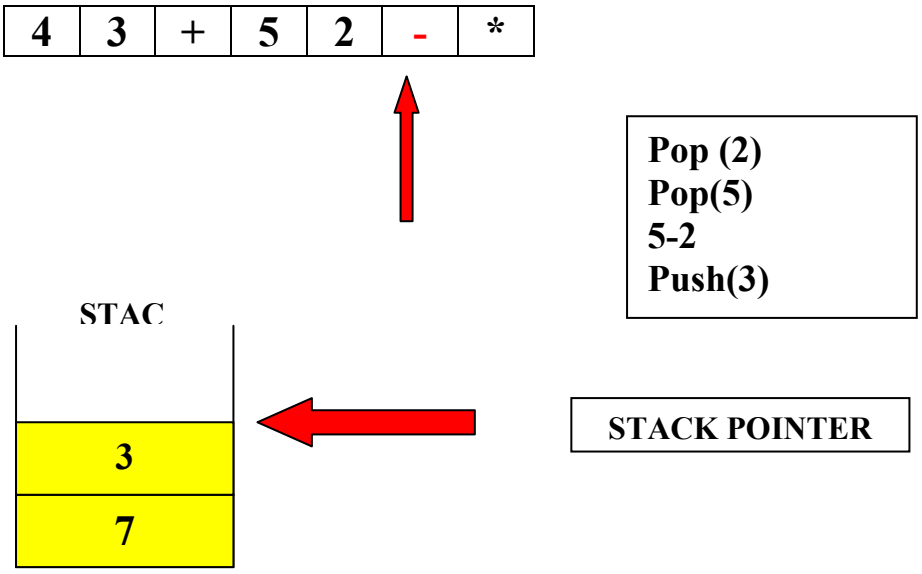


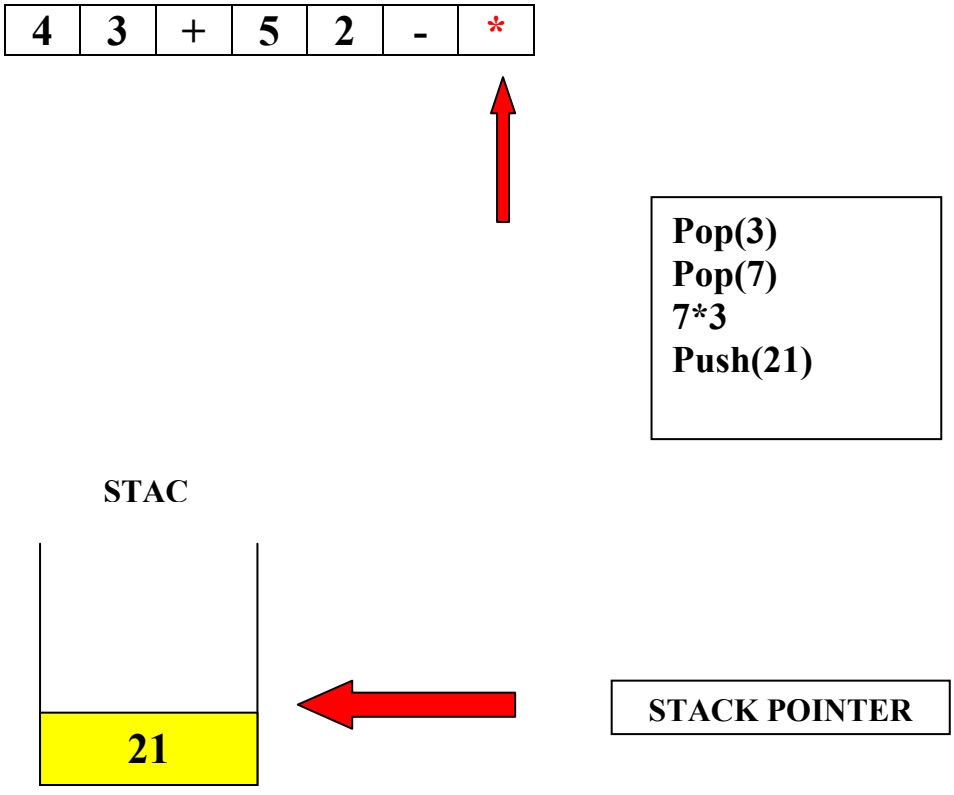

4
3
+
5
2
-
*
Push (3)
STAC
3
4
STACK POINTER


4
3
+
5
2
-
*
Pop(3)
Pop(4)
4+3
Push(7)
STAC
7
STACK POINTER


4
3
+
5
2
-
*
Push (5)
STAC
5
7
STACK POINTER


4
3
+
5
2
-
*
Push(2)
STAC
2
5
7
STACK POINTER


4
3
+
5
2
-
*
Pop (2)
Pop(5)
5-2
Push(3)
STAC
3
7
STACK POINTER


4
3
+
5
2
-
*
Pop(3)
Pop(7)
7*3
Push(21)
STAC
21
STACK POINTER

Come fare una calcolatrice

Nell'esempio che abbiamo visto i numeri vengono inseriti con PUSH (=operatore diatico)
Riassumendo, dopo avere visto la logica dello stack e l'applicazione al calcolatore del RPN, vediamo cosa devo fare per avere un programma che legge una riga di comando e esegue delle operazioni.

Regole per l'input, controllo degli errori

Devo stabilire delle regole per l'input.
Esempio : cosa succede se scrivo solo "*" o "SO10*" , come posso controllare questi input diversi?
Come posso intercettare le possibilità di errore che potrebbe incorrere il mio programma con degli input di vario tipo ?
Prima cercheremo di implementare le funzioni che dovrò usare:

sInit() loop che inizializza a zero lo stack
sPush() inserisce gli elementi nello stack e ritorna un valore che da al programma la possibilità di controllare se c'è stato un errore: 1 se nessun errore, 0 se c'è stato errore
sPop() estrae gli elementi dal mio stack
sCheckError() dice qual'è l'errore e poi toglie la condizione di errore
sCalc() effettua operazioni tra gli elementi nello stack

Funzione init()

```
/*****************************************/
/*sInit loop che inizializza a zero lo stack*/
/*****************************************/

void sInit(void)
{
     int i;
     for(i=0;i<=LEN;i++)                  /* per i volte (i da 0 da 10)
                                             inizializzo l'i-esimo elemento
                                             dello stack a 0;
                                          */

         {
                        stack[i]=0;
             }
     nelem=0;
     }
```

Funzione sPush()

```
/*****************************************************************/
/*sPush inserisce gli elementi nello stack e ritorna un valore      */
/*che da al programma la possibilità di controllare                 */
/*se c'è stato un errore: 1 se nessun errore, 0 se c'è stato errore */
/*****************************************************************/

void sPush(float x)
{                                    //lo stack sarà pieno se il numero di
elementi è = 10;
     if(nelem==LEN)
     {
        _error=s_full;
        return;
        }
     stack[nelem++]=x;               // x è il valore che vado ad inserire nell
stack
     return;

     }
```

Funzione sPop()

```
/*************************************/
/*sPop estrae gli elementi dal mio stack*/
/*************************************/

float sPop(void)
{
      float x=0.0;                 /* inizializzo il valore di x a 0;
                                       poichè è un float, devo specificare
                                       che i decimali sono a 0; */

           if(nelem==0)            // se  il numero di elementi nello stack è 0
           {
               _error=s_empty;     // errore: stack vuoto
               }

           x=(stack[nelem-1]);     /* assegno ad x il valore dell'elemento
                                      precedente ad nelem;
                                      supponiamo nelem 10--> ad x assegno il
                                      valore 9*/
           nelem--;                /* decremento nelem di uno:
                                      nelem diventa a 9;*/
           return x;                // ritorno x : x è = 9;

       }
```

Funzione sCheckerror

```
/*********************************************************************/
/*sCheckError dice qual'è l'errore e poi toglie la condizione di errore*/
/*********************************************************************/

int sCheckError(void)
{
    int t=_error;      // assegno a t il 'valore' _error
    _error=s_ok;       // controllo che errore sia ok ( cioè che non vi siano
errori )
    return t;          // ritorno _error ( cioè ritorno l'ok  )
    }
```

Funzione sCalc()

```
/*****************************************************/
/*sCalceffettua operazioni tra gli elementi nello stack*/
/*****************************************************/

float sCalc(char* op)
{
                                      /*devo riuscire a distinguere gli
operatori tra monadici e diadici
                                        quindi mi serve una serie di if che
controllano il tipo di operatori*/
      if(nelem<2)                     // per gli operatori diadici (+,-,*,/)
          {_error=s_empty;
          return 0;
          }

      if(strcmp(op,"+")==0)          //*se il confronto tra l'operatore
                                       ed il + è 0 */
          {
            stack[nelem-1]=stack[nelem-1]+stack[nelem-2];

            }

 else if(strcmp(op,"*")==0)           /* se il confronto tra l'operatore
                                         ed il + è 0 */
    {
    stack[nelem-1]=stack[nelem-1]*stack[nelem-2];

    }

 else if(strcmp(op,"-")==0)           /* se il confronto tra l'operatore
                                         ed il + è 0 */
    {
    stack[nelem-1]=stack[nelem-1]-stack[nelem-2];

    }

 else if(strcmp(op,"/")==0)            // se il confronto tra l'operatore ed il
+ è 0
    {
    stack[nelem-1]=((stack[nelem-1])/(stack[nelem-2]));
```

```
    }
 else if(stack[nelem-1]==0)            //se lo stack è vuoto segnala errore
     {
     _error=s_notz;
     return 0;
     }
                                    /* è possibile aggiungere una serie
                                       di elseif per ogni errore */

  else
    {
     _error=s_badop;

    return 0;
    }
}
```

Funzione sPrint()

```
/************************/
/*sPrint stampa lo stack */
/************************/

void sPrint(void)
{
     int i;
     for(i=0;i<=LEN;i++)                // per i volte con i da 0 a LEN (10)
     {
     printf("%d\n-->%f\n",i,stack[i]);   // ogni volta stampo i--> elemento i-
esimo
                                        /* esempio: 0 --> 21314
                                                    1 --> 23114
                                                    2 --> 03485
                                                    ecc...
                                                    10--> 3482
                                        */
     }
     printf("\n");                       // newline (endline)
    }
```

Stackmain.c – è la funzione main della mia applicazione

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>                      // da inserire per le funzioni di
operazioni tra stringhe
#include<float.h>                       // da inserire per la funzione _error
#include "stack.h"

int main()
{
    char s[80];                         // stringa caratteri
    float y;                            // variabile da tastiera
    int e=sCheckError();                // controllo errori
    for (;;)                            // ciclo infinito
        {
        printf("stack>");
        gets(s);                        // legge la stringa inserita da tastiera
        if((strcmp(s,"q"))==0)          // se premo q sulla tastiera
            exit(0);                    // il programma esce

        else if((strcmp(s,"p"))==0)     // se invece digito p
            sPrint();                   /* mi stampa a video la stringa
                                           precedentemente inserita */


        else if ((strcmp(s,"n"))==0)    // se digito n
            {
        y=(float)((strtod(s,NULL)));    / *converto la stringa in numero: come
                                           faccio a verificare che me l'abbia
                                           convertita in num? */
                 printf("%f\n",y);      // stampa la stringa convertita
                  //perror("");         /* si può inserire per un eventuale
                                           controllo sugli errori */
                  sPush(y);             // inserisce la stringa
            }
                                        // verifico se ci sono errori o meno
        else if ((strcmp(s,"c"))==0)                // se invece digito c sulla
tastiera
            {
               y=sCalc(s);              // chiamo la funzione sCalc

               if (e==s_ok)             // se non ci sono errori
               {
                printf("%f\n",y);       // stampo y
               }

               else if(e==s_badop)      // se ci sono errori di bad operator
               {
               sPush((float)strtod(s,NULL));  // inserisco una stringa di float
               }

              /* else
               {

                   }*/
            }                               // chiudo l'else if
((strcmp(s,"c")==0))
        }                                   // chiudo il for (;;)
system("pause");                            // solo per DEV++ windows
return 0;
}                                           // chiudo il main
```

Stack.c

Inserisco ora l'insieme completo delle funzioni presentate prima:

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<float.h>         // da inserire per la funzione _error
#define LEN 10            // dichiaro che lo stack ha massimo 10 elementi
#include "stack.h"

static float stack[LEN];  // rende lo stack di tipo private
static int nelem=0;       // inizializzo nelem a 0 e la rendo private
static int _error=s_ok;   // inizializzo _error ad ok (non errore)


/*****************************************/
/*sInit loop che inizializza a zero lo stack*/
/*****************************************/

void sInit(void)
{
     int i;
     for(i=0;i<=LEN;i++)                  /* per i volte (i da 0 da 10)
                                             inizializzo l'i-esimo elemento
                                             dello stack a 0;
                                          */

         {
                        stack[i]=0;
             }
     nelem=0;
     }

/*****************************************************************/
/*sPush inserisce gli elementi nello stack e ritorna un valore      */
/*che da al programma la possibilità di controllare                 */
/*se c'è stato un errore: 1 se nessun errore, 0 se c'è stato errore */
/*****************************************************************/

void sPush(float x)
{                                     //lo stack sarà pieno se il numero di elementi è =
10;
     if(nelem==LEN)
     {
        _error=s_full;
        return;
        }
     stack[nelem++]=x;               // x è il valore che vado ad inserire nell stack
     return;

     }

/*************************************/
/*sPop estrae gli elementi dal mio stack*/
/*************************************/

float sPop(void)
{
      float x=0.0;                   /* inizializzo il valore di x a 0;
                                        poichè è un float, devo specificare
                                        che i decimali sono a 0; */

           if(nelem==0)               // se  il numero di elementi nello stack è 0
           {
               _error=s_empty;        // errore: stack vuoto
               }

           x=(stack[nelem-1]);        /* assegno ad x il valore
                                         dell'elemento precedente ad nelem;
```

```
                                        supponiamo nelem 10--> ad x assegno
                                        il valore 9 */
          nelem--;                    // decremento nelem di uno: nelem diventa a 9;
          return x;                   // ritorno x : x è = 9;

       }

/*******************************************************/
/*sCalceffettua operazioni tra gli elementi nello stack*/
/*******************************************************/

float sCalc(char* op)
{
                                     /*devo riuscire a distinguere gli operatori tra
monadici e diadici
                                       quindi mi serve una serie di if che controllano
il tipo di operatori*/
      if(nelem<2)                    // per gli operatori diadici (+,-,*,/)
          {_error=s_empty;
          return 0;
          }

      if(strcmp(op,"+")==0)           // se il confronto tra l'operatore ed il + è 0
          {
            stack[nelem-1]=stack[nelem-1]+stack[nelem-2];

            }

 else if(strcmp(op,"*")==0)           // se il confronto tra l'operatore ed il + è 0
    {
    stack[nelem-1]=stack[nelem-1]*stack[nelem-2];

    }

 else if(strcmp(op,"-")==0)           // se il confronto tra l'operatore ed il + è 0
    {
    stack[nelem-1]=stack[nelem-1]-stack[nelem-2];

    }

 else if(strcmp(op,"/")==0)           // se il confronto tra l'operatore ed il + è 0
    {
    stack[nelem-1]=((stack[nelem-1])/(stack[nelem-2]));
    }
 else if(stack[nelem-1]==0)            //se lo stack è vuoto segnala errore
     {
     _error=s_notz;
     return 0;
     }
                                    /*è possibile aggiungere una serie
                                      di elsif per ogni errore */

  else
    {
     _error=s_badop;

    return 0;
    }
}
/******* FINE FUNZIONE sCalc*******/



/*************************************************************/
/*sCheckdice qual'è l'errore e poi toglie la condizione di errore*/
/*************************************************************/

int sCheckError(void)
{
    int t=_error;      // assegno a t il 'valore' _error
    _error=s_ok;       // controllo che errore sia ok ( cioè che non vi siano errori )
```

```
    return t;          // ritorno _error ( cioè ritorno l'ok  )
    }

/************************/
/*sPrint stampa lo stack */
/************************/

void sPrint(void)
{
     int i;
     for(i=0;i<=LEN;i++)                // per i volte con i da 0 a LEN (10)
     {
     printf("%d\n-->%f\n",i,stack[i]);   // ogni volta stampo i--> elemento i-esimo
                                        /* esempio: 0 --> 21314
                                                    1 --> 23114
                                                    2 --> 03485
                                                    ecc...
                                                    10--> 3482
                                        */
     }
     printf("\n");                       // newline (endline)
    }
```

Devo inoltre dichiarare il file **stack.h**, che contiene le dichiarazioni :

Esempio: stack.h

```
#define s_ok 0              // nessun errore
#define s_empty 1           // errore: lo stack è vuoto
#define s_full 2            // errore: lo stack è pieno
#define s_badop 3           // errore: l'operatore non è riconosciuto
#define s_notz 4            // errore: lo stack non è nullo

void sInit(void);
void sPush(float x);
float sPop(void);
float sCalc(char* op);
int sCheckError(void);
void sPrint();
```

Funzione stackmain

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>                        // da inserire per le funzioni di
operazioni tra stringhe
#include<float.h>                        // da inserire per la funzione _error
#include "stack.h"


int main()
{
    char s[80];                          // stringa caratteri
    float y;                             // variabile da tastiera
    int e=sCheckError();                 // controllo errori
```

```
    for (;;)                              // ciclo infinito
        {
        printf("stack>");
        gets(s);                         // legge la stringa inserita da tastiera
        if((strcmp(s,"q"))==0)           // se premo q sulla tastiera
            exit(0);                     // il programma esce

        else if((strcmp(s,"p"))==0)      // se invece digito p
            sPrint();                    // mi stampa a video la stringa
precedentemente inserita

        else if ((strcmp(s,"n"))==0)     // se digito n
            {
         y=(float)((strtod(s,NULL)));    /* converto la stringa in numero: come
                                            faccio a verificare che me l'abbia
                                            convertita in num? */
                printf("%f\n",y);        // stampa la stringa convertita
                 //perror("");           // si può inserire per un eventuale
controllo sugli errori
                 sPush(y);               // inserisce la stringa

            }

                                         // verifico se ci sono errori o meno
        else if ((strcmp(s,"c"))==0)     // se invece digito c sulla tastiera
            {

               y=sCalc(s);               // chiamo la funzione sCalc

               if (e==s_ok)              // se non ci sono errori
               {
                printf("%f\n",y);        // stampo y
               }

               else if(e==s_badop)       // se ci sono errori di bad operator
               {
               sPush((float)strtod(s,NULL));  // inserisco una stringa di float
               }

              /* else
               {

                   }*/
            }                                // chiudo l'else if
((strcmp(s,"c")==0))
        }                                    // chiudo il for (;;)
system("pause");
return 0;
}                                            // chiudo il main
```

RIEPILOGO - Corso Laboratorio di programmazione

Il corso consiste nella preparazione del linguaggio C (C++) che, con alcune differenze sintattiche coincide con java,java script, ecc

Il sistema operativo usato al corso è SUN OS 58 – il compilatore è il compilatore c di unix sun .
Le prime 2-3 settimane servono per prendere confidenza con il linguaggio. Le difficoltà saliranno dopo le prime lezioni.
L'esame consiste nel preparare un progetto (programma) su un argomento dato dal docente ca 20 giorni prima e nella sua discussione ed eventuale modifica; il alternativa produrre la documentazione del corso.(accordarsi col docente).

USO DI UNIX – NOTE PRELIMINARI

Entrare in tools/terminal
[ray]/home/student/2003-2004/*nome_utente*

Questa è la path che si ha dopo l'ingresso, è il percorso, la home directory assegnata dal sistema ad ognuno.
Ricordarsi che Unix è case sensitive , ovvero riconosce la differenza fra maiuscole e minuscole.
>pwd= print working directory = dà la directory di lavoro

Tutti I comandi che immettiamo sono presi da un command detto "shell"
La "shell" è di 2 tipi fondamentali:
la "sh" o bourne shell (bash)
la "csh" o c shell (tcsh)

Noi abbiamo una "tcsh" (la vedo eseguendo il commando >set)

Alcuni comandi di unix:
>cd = change directory
Nb: in unix la directory dove sono si chiama "." (=punto) ed è la current directory
La directory "in su" (superiore) è ".." (= doppio punto)
>man = manuale
>ls = list = per vedere il contenuto della cartella
Eseguo ls con l'opzione "ls –al" e ottengo:
[d][rwx][rwx[]rwx] [ ] [utente] [gruppo] [dimensioni byte] [data modifica] [nome]

il primo carattere indica che tipo di file (d=diretctory) spazio = file
I tre gruppi di rwx = Read Write Xecute sono riferiti in sequenza a Tutti,Gruppo,Utente

Altri comandi utili di "ls " sono:
> ls –alt t = ordine di data
> ls –altR R = recursive x sotto cartelle
> ls –altRF F = simbolo [*]vicino a file

>mkdir = make directory = crea directory
>rmdir +[nomedir] = remove directory = rimuovi directory
>rm +[file] = remove = rimuovi file

Per **vedere il contenuto del file** basta digitare >cat hello.c
Si usa >cat “hell .c” se uno mette spazi nel nome // rispetto al precedente ci sono le “”. La mancanza della “o” è voluta.
>mv mi permette di cambiare il nome del file

per **cercare nel manuale** di unix posso usare il comando:
>apropos … NomeDaCercare…. Per usare la funzione il seme è **time(void)**

quanto impiega ad essere eseguito un programma?
In unix c’è una funzione :
>time *nome_programma* (tempo effettivo usato dalla cpu)

Per **controllare nel sistema** si possono usare i comandi “cat”, “more” e “less”
>cat /usr/include/math.h

I programmi in C

In unix I programme in c devono finire con [.c], perchè il compilatore che useremo considera I file che finiscono con [.c].
Si può usare come editor “vi” o “nedit “
NB Per l’esame si può usare il proprio portatile
Useremo un compilatore che si chiama **gcc** dove:
g=gnu
c=c
c=compiler

gcc version 3.2.2 (c’è anche su linux con sistema di sviluppo)
per fermare lo scorrimento scrivo nel prompt ctrl+z e scrivo “bg” = background ; ma è meglio lanciare l’editor in background lanciandolo con “nedit&” [djgpp]
per andare in internet dal terminale di facoltà scrivere “tarantella”

nei vari sistemi ci sono funzioni “random” chiamate in modo diverso con i vari tipi di compilatori c.

TABELLA CODICE ASCII

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

USO PROGRAMMI IN C SU DEV++ (windows)

Si possono fare esercitazioni a casa utilizzando l'ottimo programma DEV++ della Bloodshed Software – Licenza tipo GNU .
Con questo sistema di sviluppo ho anche un ottimo editor con correzione della sintassi ; posso gestire programmi in C e C++.
Per fare il download http://www.bloodshed.net

Per far funzionare correttamente i programmi col compilatore di DEV++ devo avere alcune attenzioni sulla sintassi e funzioni non del tutto compatibili.

I programmi presso la facoltà lavorano a prompt di dos, ovvero sono in formato carattere; per vedere i risultati sul sistema DEV++ devo aggiungere una riga che "congela" momentaneamente i risultati a video prima di ritornare alla grafica. Tale riga dovrà essere così:

```
/* inserita la seguente riga x far funzionare con DEV++ sotto windows */
system("pause");
return 0
```

in fondo al programma di visualizzazione (stampa) inserisco la funzione system("pause") per poter così "fermare" il programma e vedere a video i risultati. Inoltre devo modificare alcune funzioni usate negli esempi:

UNIX	WINDOWS
random()	rand()
srandom()	srand()
drand(48)	#define <limits.h> #define drand48() ((double)rand())/((double)rand_max)