Lezione 19 e 20

- Allocazione dinamica della memoria
- Direttive al preprocessore
- Libreria standard
- Gestione delle stringhe

Valentina Ciriani (2005-2008)
Laboratorio di programmazione

Valentina Ciriani (2005-2008)
Laboratorio di programmazione

Lezione 19 e 20

Allocazione dinamica della memoria

Obiettivo:

- Creare dinamicamente spazio di memoria

I puntatori

- **I puntatori sono celle contenenti solo degli indirizzi e sono dichiarati esprimendo il tipo di dato puntato:**
  - esempio: **char * pc;**
- **Leggere l'indirizzo di una cella usando '&':**
  - esempio: **pc = &c;**
- **Accedere in lettura o scrittura ad una cella puntata da un puntatore mediante '*':**
  - esempio: ***pc = 'z';**

3

Assegnamento puntatori

- **Un puntatore è un indirizzo di memoria di una variabile di cui è noto il tipo**
- **Un puntatore può contenere:**
  - l'indirizzo di una variabile esistente allocata staticamente all'inizio dell'esecuzione
  - l'indirizzo di una variabile nuova allocata dinamicamente durante l'esecuzione

4

malloc

```
void* malloc(unsigned int n)
```

- **malloc** = **m**-emory **alloc**-ation:
  - alloca **n** byte di memoria
  - restituisce un puntatore alla memoria allocata
  - sotto forma di puntatore generico (void*) e quindi dovremo fare un cast per restituire il puntatore giusto

5

Esempio

- **Il codice qui riportato**

```
int* p;
p = malloc(sizeof(int));
```

è sbagliato, perché ?

- **Il codice qui riportato**

```
int* p;
p = (int*) malloc(sizeof(int));
```

è invece corretto poiché con il cast i tipi corrispondono

6

E se la memoria finisce?

- **Se non c'è più memoria**
  - l'allocazione fallisce
  - **`malloc`** restituisce il valore NULL
  - come la **`fopen`**
  - dobbiamo controllare il valore restituito dalla **`malloc`**
- **Allocazione corretta:**

```
int* p;
p = (int*) malloc(sizeof(int));
if (p == NULL) {
   printf("E' finita la memoria");
   exit(-1);
}
```

7

Utilizzo di `sizeof`

```
typedef struct {
  ...
} MissItalia;
MissItalia* p;
p = (MissItalia*) malloc(sizeof(MissItalia));
```

- **Il costrutto `sizeof` è molto utile con le `malloc`**
- **Va usato sempre, anche con i tipi base**
  - int, short, float, double ...
  - il numero di byte che occupano non è sempre fisso!

8

Deallocazione della memoria

```
void free(void* p);
```

- La funzione `free` libera la memoria che era stata allocata all'indirizzo p
- Importante: p deve essere il risultato di una allocazione dinamica!

- Se mi dimentico di disallocare, ho un memory leak
- In C non c'è alcuna garbage collection ovvero le celle di memoria non disallocate vanno perse

9

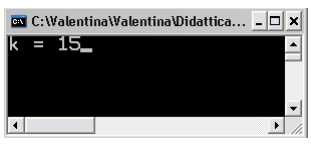
Esempio

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

int main()
{
  int k;
  k=15;

  printf("k = %d", k);

  fflush(stdin);
  getchar();
  return 0;
}
```



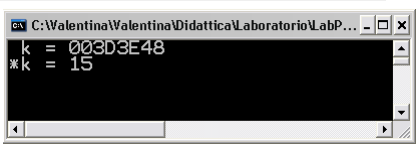

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   int* k;
   k = (int*)malloc(sizeof(int));
   *k = 15;

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

   free(k);

   fflush(stdin);
   getchar();
   return 0;
}
```



10

Problema

- **Stampare in ordine inverso gli interi passati da terminale. Ovvero se l'utente ci passa gli interi 2,67,8,45,6,10 dobbiamo stampare**
  - 10,6,45,8,67,2
- **Quando stiamo scrivendo il codice non sappiamo quanti interi l'utente ci passerà!**
- **A tempo di esecuzione possiamo chiedere come prima cosa quanti interi ci passa**

- **Come facciamo a memorizzarli?**
- **Se l'utente ci passa n interi**
  - dobbiamo allocare un vettore di n elementi interi
  - Soluzione: allocazione dinamica di un array!

`calloc`

```
(void*)calloc(unsigned int n, unsigned int s);
```

- **`calloc` = c-ontiguous alloc-ation**
- **Alloca n elementi contigui ciascuno di s byte**
- **In pratica, alloca un area di memoria grande**
  - n x s byte
- **Per il resto funziona come una `malloc`**
- **Esempio:**

```
int* p;
p = (int*) calloc(1000,sizeof(int));
```

- **Alloca un vettore di 1000 interi**

Dichiarazione dei vettori (1)

- **Abbiamo due tipi di vettori:**
  - vettori di lunghezza fissata a priori:

```
int v[1000]; //dichiarazione
```

  - vettori allocati dinamicamente:

```
int* v; //dichiarazione
v = (int*) calloc(1000,sizeof(int));
```

- **In entrambi i casi:**
  - ho un vettore di 1000 interi
  - l'i-esimo elemento di v è **v[i]**

13

Dichiarazione dei vettori (2)

- **Abbiamo due tipi di vettori:**
  - vettori di lunghezza fissata a priori:

```
int v[1000]; //dichiarazione
```

  - vettori allocati dinamicamente:

```
int* v; //dichiarazione
v = (int*) calloc(1000,sizeof(int));
```

- **Differenza 1:**
  -

```
int v[1000]; //devo fissare a priori la
             //lunghezza del vettore
```

  -

```
int* v; //dichiaro il vettore senza sapere la
        //sua lunghezza
scanf("%d", &size);
v = (int*) calloc(size,sizeof(int));
```

14

Dichiarazione dei vettori (3)

- **Abbiamo due tipi di vettori:**
  - vettori di lunghezza fissata a priori:

```
int v[1000]; //dichiarazione
```

  - vettori allocati dinamicamente:

```
int* v; //dichiarazione
v = (int*) calloc(1000,sizeof(int));
```

- **Differenza 2: se ho allocato, devo deallocare**

```
int* v = (int*) calloc(1000,sizeof(int) );
... /* usa v */
free(v);
```


Dichiarazione dei vettori (4)

- **Abbiamo due tipi di vettori:**
  - vettori di lunghezza fissata a priori:

```
int v[1000]; //dichiarazione
```

  - vettori allocati dinamicamente:

```
int* v; //dichiarazione
v = (int*) calloc(1000,sizeof(int));
```

- **Differenza 3: vettori di lunghezza fissata a priori sono più efficienti**
  - i vettori a lunghezza variabili occupano più memoria a causa dei puntatori

Dichiarazione dei vettori (5)

- **Abbiamo due tipi di vettori:**
  - vettori di lunghezza fissata a priori:

```
int v[1000]; //dichiarazione
```

  - vettori allocati dinamicamente:

```
int* v; //dichiarazione
v = (int*) calloc(1000,sizeof(int));
```

- **Differenza 4: vengono allocati in zone diverse della memoria**
  - i vettori di lunghezza fissata sono nel segmento delle variabili
  - i vettori allocati dinamicamente sono nel segmento heap

17

Esempio

- **Risolviamo il problema del vettore rovesciato**

```
int main()
{
   int n,i;
   int* v; //dichiaro il vettore ma non so la lunghezza

   printf("Quanti numeri mi passi? ");
   scanf("%d", &n); //l'utente mi dice la lunghezza
   v = (int*) calloc (n, sizeof(int)); // alloco il vettore
   if (v == NULL) {
      printf("E' finita la memoria");
      exit(-1);
   }
   for(i=0; i<n; i++){
     printf("inserisci v[%d]= ", i);
     scanf("%d", &v[i]);
   }
   printf("Stampo il vettore al c
   for(i=n-1; i>=0; i--){
     printf("\n %d", v[i]);
   }
   free(v); //disalloco la memori
   fflush(stdin);
   getchar();
   return 0;
}
```

```
C:\Valentina\Valentina\Didattica\Laboratorio\LabProg\codice\lez19e20\...
Quanti numeri mi passi? 5
inserisci v[0]= 0
inserisci v[1]= 1
inserisci v[2]= 2
inserisci v[3]= 3
inserisci v[4]= 4
Stampo il vettore al contrario
 4
 3
 2
 1
 0
```

Valentina Ciriani (2005-2008)
Laboratorio di programmazione

Lezione 19 e 20

Direttive al preprocessore

Obiettivo:

- Conoscere più nel dettaglio le direttive al preprocessore

`#include` e `#define`

- **Quelle che abbiamo visto sono principalmente due:**
  - includere le librerie
    (es: `#include <stdio.h>`);
  - definire delle etichette
    (es: `#define DIMENSIONEMATRICE 5`).
- **Vanno inserite in testa al programma (per motivi di buona programmazione)**

20

#include

- **La direttiva `#include` consente di includere la copia di un file specificato**
- **Esistono due forme di include:**
  - **`#include <nome file>`**
    - il preprocessore cerca il file nelle directory predefinite
    - usato per i file della libreria standard
  - **`#include "nome file"`**
    - il preprocessore ricerca il file nella stessa directory di quello da compilare
    - usato per i file definiti dall'utente

#define con le macro

- **Una macro del linguaggio C è un identificatore definito all'interno di una `#define`.**
- **Le macro possono essere con e senza argomenti**
- **Una macro senza argomenti è usata come costante**
  - Es **`#define PI() 3.14`**
- **In una macro con argomenti questi sono rimpiazzati all'interno del testo di sostituzione**
  - Es **`#define AREA_CERCHIO(x) (PI() * (x) * (x))`**
  - può essere chiamata con: **`AREA_CERCHIO(6)`**

Esempio

```
#define AREA_CERCHIO(x) (PI() * (x) * (x))
```

- **Perché le parentesi intorno alle x?**
  - `area = AREA_CERCHIO(3)`
    viene espansa dal precompilatore in
    `area = (3.14 * (3) * (3))`
  - `area = AREA_CERCHIO(y + 3)`
    viene espansa dal precompilatore in
    `area = (3.14 * (y + 3) * (y + 3))` (corretta!)
  - senza le parentesi avremmo avuto
    `area = (3.14 * y + 3 * y + 3)` (sbagliata!)

Funzioni vs macro

- **Possiamo usare le funzioni al posto delle macro**
- **Esempio**
  - `#define AREA_CERCHIO(x) (PI() * (x) * (x))`
  - ```
    double Area_Cerchio(double x){
        return 3.14 * x * x; // niente parentesi!
    }
    ```
- **Usando la macro si risparmia il tempo della chiamata di funzione**
- **La macro però valuta due volte l'argomento**
- **Regola: usare la macro se è composta da una sola linea di codice**
    ```

Espressioni condizionali

- **Sintassi:**

```
espr1 ? espr2 : espr3
```

- **Se la valutazione di `espr1`**
  - restituisce vero il risultato è **`espr2`**
  - restituisce falso il risultato è **`espr3`**
- **Esempio:**

```
a=7;
x = y ? a+1 : 3;
```

  - se y vale 4 allora x vale 8
  - se y vale 0 allora x vale 3

25

Macro con espressioni condizionali

- **Le espressioni condizionali sono molto utili nella definizione delle macro**
- **Esempio: usiamo una macro per definire la funzione minimo tra due numeri:**

```
#define min(x,y) (((x) < (y)) ? (x) : (y))
```

26

Valentina Ciriani (2005-2008)
Laboratorio di programmazione

Lezione 19 e 20

Librerie standard

Obiettivo:

- Conoscere le funzioni delle librerie standard

Libreria standard ANSI

- **Ogni funzione della libreria standard è associata a uno o più file di header che devono essere inclusi se si fa uso della funzione**
- **La definizione delle funzioni dichiarate nei file di header dipende dall'implementazione**
- **La libreria standard contiene funzioni per**
  - gestione input/output
  - gestione stringhe
  - funzioni matematiche
  - gestione memoria
  - conversioni
  - gestione data e ora ecc...

28

Gli header

- **In genere non è necessario importare l'intera libreria:**
  - si importano solo i singoli header che ci servono
  - Es. se uso una **`printf()`** importo **`<stdio.h>`**
- **Importiamo solo alcune funzioni della libreria standard quando ci servono!**
- **Molti header li abbiamo già incontrati**
  - **`<stdio.h>`**
  - **`<stdlib.h>`**
  - **`<math.h>, ecc..`**

29

Gli header

`stdio.h`	input/output
`stdlib.h`	funzioni di uso generale
`string.h`	stringhe
`math.h`	matematica
`time.h`	data e ora
`ctype.h`	classificazione e conversione caratteri
`assert.h`	asserzioni
`stdarg.h`	gestione funzioni a n variabile di argomenti
`setjmp.h`	salti non locali
`signal.h`	gestione eventi
`limits.h`	limiti e caratteristiche dei tipi interi
`float.h`	limiti e caratteristiche dei tipi reali
`errno.h`	gestione errori
`locale.h`	localizzazione
`stddef.h`	dichiarazioni di tipi utilizzati nella libreria

30

Valentina Ciriani (2005-2008)
Laboratorio di programmazione

Lezione 19 e 20

Gestione delle stringhe

Obiettivi:

- Conoscere più nel dettaglio le la gestione delle stringhe
- Conoscere le principali funzioni dell'header **`<string.h>`**
- Utilizzare correttamente le funzioni **`sprintf`** e **`sscanf`**

`string.h`

- **Le stringhe in C sono array monodimensionali di caratteri**
- **La maggior parte delle operazioni sulle stringhe sono nell'header `string.h`**
- **Una stringa termina SEMPRE con il carattere terminatore di stringa:**
  - '\0' (una cella tutta a zero).
- **Dichiarazione in due modi:**
  - **`char s[10];`**
  - **`char s[]= "stringa iniziale";`**

32

Funzioni per la gestione delle stringhe

- **Vediamo le più importanti:**
  - strcpy() copia una stringa in un'altra
  - strlen() restituisce la lunghezza di una stringa
  - strcmp() confronta due stringhe
  - strcat() concatena due stringhe

33

Copia di una stringa

- **Non si può copiare una stringa in un'altra con un assegnamento!**

```
Es: char s[128];
    s = "Laboratorio di Programmazione";// no!!!
```

- **Per copiare una stringa in un'altra si usa:**

```
char * strcpy(char *s1, const char *s2)
```

```c
#include <stdio.h>
main()
{
    char s[128];

    // strcpy(s_to,s_from) copia la stringa s_from in s_to
    // fino a quando trova un terminatore di stringa

    strcpy(s,"Laboratorio di Informatica applicata");

    printf("%s",s);
    getchar();
}
```

34

Lunghezza di una stringa

- **Per contare il numero di caratteri di una stringa che precedono il carattere '\0' si usa:**

```
size_t strlen(const char *s)
```

(`size_t` di solito è un `unsigned int`)

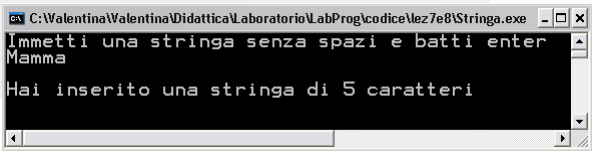
```
#include <stdio.h>
int main()
{
   char s[128];
   int lung;

   printf("Immetti una stringa senza spazi e batti enter\n");
   scanf ( "%s" , s ); // senza &!!!!!

   lung = strlen(s);

   printf("\nHai inserito una stringa di %d caratteri", lung);

   fflush(stdin);
   getchar();
   return 0;
}
```



35

Confronto tra due stringhe

- **Per confrontare due stringhe possiamo usare**

```
int strcmp(const char *s1, const char *s2)
```

- **Restituisce:**
  - 0 se le due stringhe sono uguali
  - valore <0 se s1 è minore di s2 (ordine lessicografico)
  - valore >0 se s1 è maggiore di s2
- **Esempio**

```
char s1[] = "domani";
char s2[] = "oggi";
c = strcmp(s1, s2); // c contiene -1
strcpy(s1, s2);
c = strcmp(s1, s2); // c contiene 0
```

36

Concatenazione

- **Per concatenare due stringhe si usa:**

```
char * strcat(char *s1, const char *s2)
```

- **accoda la stringa s2 alla stringa s1**
  - il primo carattere di s2 si sostituisce al carattere di terminazione di s1.
  - oltre a modificare s1 restituisce anche s1
- **Esempio**

```
char s1[] = "domani";
char s2[] = "oggi";
char *s3;
s3 = strcat(s1, s2); // s3 e s1 contengono
                     // "domanioggi"
```

37

Input /output

- **Nel header <stdio.h> ci sono delle funzioni utili per la gestione delle stringhe:**

	output	input
terminale	`printf`	`scanf`
stringhe	`sprintf`	`sscanf`
file	`fprintf`	`fscanf`

38

sprintf

- **Possiamo stampare su una stringa con la**

```
int sprintf(char *s, const char *format, ...)
```

- **È equivalente alla `printf` eccetto che l'output sarà immagazzinato nella stringa s**
- **Esempio:**

```
char *s1;
int x=3;
char s2[] = "matite";
sprintf(s1,"Ho %d %s", x, s2);
// s1 contiene la stringa "Ho 3 matite"
```

39

sscanf

- **Possiamo leggere da una stringa con la**

```
int sscanf(char *s, const char *format, ...)
```

- **È equivalente alla `scanf` eccetto che l'input è nella stringa s**
- **Esempio:**

```
char s1[] = "2 volte tre";
int x;
char *s2;
sscanf(s1,"%d %s tre", &x, s2);
//s2 contiene la stringa "volte", x il valore 2
```

40

Esempio (1)

- **Scrivere un programma che**
  - prende in input il file non vuoto di stringhe stringhe.txt
  - restituisce come output un file
    - contenente il numero di stringhe in stringhe.txt uguali alla prima stringa di stringhe.txt
    - e che abbia come nome "nome della stringa più lunga".txt

Esempio (2)

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

int main()
{
   char s[128]; //stringa
   char p[128]; // prima stringa
   char sl[128]; //stringa piu' lunga
   char nomefile[128]; //nome del file di output
   FILE *pfin, *pfout;
   int lung =1;


   /* apertura del file di input */
   pfin = fopen("stringhe.txt", "r");
   if (pfin == NULL) // controlla se il file viene aperto
   {
      printf("Non posso aprire il file %s\n", "stringhe.txt");
      exit(-1);
   }
```

Esempio (3)

Leggo la prima stringa

```
fscanf(pfin,"%s", p); //leggo la prima stringa
strcpy(sl, p); //la prima stringa per ora e' la piu' lunga

while(!feof(pfin)) //leggo tutte le stringhe
{
   fscanf(pfin,"%s", s);
   if (!strcmp(p,s)) // se sono uguali
      lung++; // incremento il contatore

   // calcolo la piu' lunga
   if (strlen(s)>strlen(sl))
      strcpy(sl, s);
}

fflush(pfin);
fclose(pfin);
```

Confronto con la prima stringa

Cerco la stringa piu' lunga

Esempio (4)

Scrivo la stringa piu' lunga nel nome del file

```
//nome file e' "stringa piu' lunga".txt
sprintf(nomefile, "%s.txt", sl);

/* apertura del file di output */
pfout = fopen(nomefile, "w");
if (pfout == NULL) // controllo se il file viene aperto
{
   printf("Non posso aprire il file %s\n", nomefile);
   exit(-1);
}
fprintf(pfout, "%d", lung);

fflush(pfout);
fclose(pfout);

return 0;
}
```

Scrivo lung nel file di output