Lezione 18

Compendio C/C++

Contenuto lezione

- Libreria standard C e C++
- Input/Output in C
- Macro e costanti
- Dichiarazioni e **`typedef`**
- Gestione memoria dinamica in C

Librerie 1/2

- Il solo insieme di istruzioni di un linguaggio di programmazione come il C/C++ è sufficiente a scrivere un qualsiasi programma in grado per lo meno di interagire con l'utente?
  - Decisamente no
- Gli oggetti *cin* e *cout* erano oggetti predefinti del linguaggio?

Librerie 2/2

- No, sono oggetti appartenenti ad una **libreria**
- Una libreria è una raccolta di funzioni ed oggetti che permettono di effettuare determinati insiemi di operazioni
  - Esistono ad esempio librerie per l'ingresso/uscita, librerie matematiche, librerie grafiche e così via

Libreria standard 1/2

- Sia nel linguaggio C che nel linguaggio C++ è prevista la disponibilità di una **libreria standard**
- La libreria standard del C++ è sostanzialmente un sovrainsieme di quella del C
- Sia la libreria standard del C che quella del C++ sono costituite da molti moduli, ciascuno dei quali è praticamente una libreria a se stante, che fornisce funzioni ed oggetti per un determinato scopo
- Per utilizzare ciascun modulo è tipicamente necessario includere un ben determinato *header file*

Libreria standard 2/2

- Alcuni moduli di base della libreria standard per il C ed il C++ sono:

Libreria	Header C	Header C++
Matematica	`math.h`	`cmath`
Ingresso/Uscita	`stdio.h`	`iostream`
Limiti numerici	`limits.h`	`limits`

Altre librerie

- Sia per il C che per il C++ sono disponibili molte librerie, utili per avere implementazioni alternative di alcuni degli oggetti forniti dalle librerie standard, o per realizzare compiti specifici
  - Costruire interfacce grafiche
  - Manipolare immagini, video o audio
  - ...
- Un insieme di librerie molto utilizzate per il linguaggio C++ sono le librerie *Boost*

Uso moduli C in C++

- Per utilizzare i moduli della libreria standard C++ a comune col C è bene includere degli header file il cui nome si ottiene, a partire dal nome del corrispondente header file per il C, eliminando il suffisso **`.h`** ed aggiungendo una **`c`** all'inizio del nome
  - Es.: la libreria matematica è presentata nell'header file **`math.h`** in C, mentre in C++ è presentata nell'header file **`cmath`**
  - Volendo, anche in C++ si possono includere gli header file originali del C, ma è una pratica sconsigliata
- Nel caso del C++, i nomi delle funzioni e degli oggetti di queste librerie sono definiti nello spazio dei nomi *std*
- In pratica, per usarli, bisogna aggiungere sempre la direttiva:

```
using namespace std ;
```

Ingresso/uscita formattato in C

- Diversamente dal C++, in C l'Input/Output formattato è realizzato mediante funzioni di libreria presentate in **`<stdio.h>`**
  - **`<cstdio>`** se volete utilizzare tali funzioni in C++
- Tra le funzioni principali:
  - **`printf`**: output formattato su *stdout*
  - **`scanf`**: input formattato da *stdin*

printf 1/4

```
void printf(const char format[], ... ) ;
```

Lista valori da stampare, opzionali

- La stringa **format** può contenere due tipi di oggetti:
  1) Caratteri ordinari (incluso quelli speciali), copiati tali e quali sullo *stdout*
     Esempio:

```
printf("Ciao mondo\n") ;
```

     scrive **Ciao mondo** sullo *stdout*

`printf` 2/4

2) *Specifiche di conversione*

- utilizzate solo se sono passati ulteriori parametri, contenenti valori da stampare, dopo la stringa **`format`**
- controllano l'interpretazione e quindi la traduzione in caratteri dei valori dei parametri aggiuntivi da stampare
  - E' necessario inserire una specifica per ogni valore da stampare

`printf` 3/4

- Nella posizione in cui appare una specifica di conversione nella stringa di formato
  - Verrà stampato, al posto di tale specifica, il valore passato come ulteriore parametro
  - Col formato determinato dalla specifica di conversione stessa
- Una specifica di conversione ha la forma:
  `%<`*sequenza di caratteri che specificano il tipo ed il formato del valore da stampare*`>`

printf 4/4

- Alcune delle specifiche di conversione più utilizzate sono:
  - `%d` Numero intero, da stampare in notazione decimale
  - `%g` Numero reale, da stampare in notazione decimale
  - `%c` Carattere, tipicamente codifica ASCII
  - `%s` Stringa, tipicamente codifica ASCII
- Esempio:

```
int a = 15; double b = 16.5 ;
printf("Il valore di a è %d, quello di b è %g\n", a, b) ;
```

- Equivale a

```
cout<<"Il valore di a è "<<a
    <<", quello di b è "<<b<<endl;
```

Domanda

- Quanti argomenti ha la funzione **`printf`**?

Numero argomenti `printf`

- Un numero di argomenti **variabile**
  - Per ogni specifica di conversione si può aggiungere un parametro attuale contenente il valore da stampare
- Esempi:

```
printf("Ciao\n") ;                    // un parametro
printf("%d", 3) ;                     // due parametri
printf("%d, %g, %d", 2, 2.5, 1) ; // quattro parametri
```

Funzioni variadiche

- Funzioni con un numero variabile di argomenti si definiscono **variadiche**
- Sia in C che in C++ si possono definire funzioni variadiche
  - Per dichiararle si utilizza una estensione della sintassi vista finora
  - Per accedere agli argomenti formali nel corpo di tali funzioni, si utilizzano tipicamente delle funzionalità fornite da una apposita libreria
  - Non vedremo ulteriori dettagli in questo corso

Confronto `printf` e << 1/2

- Come mai con l'operatore << non abbiamo utilizzato specifiche di conversione per stabilire come interpretare i valori degli operandi?

Confronto `printf` e << 2/2

- Perché l'operatore <<
  - Determina automaticamente il tipo dei valori, senza che sia necessario informarlo esplicitamente
  - Dal tipo decide autonomamente anche il formato
    - Che si può comunque modificare ulteriormente attraverso manipolatori e funzioni membro opportune

`scanf` 1/2

```
void scanf(const char format[], <indirizzo_variabile> ) ;
```

Indirizzo variabile in cui riversare ciò che si legge da stdin

- Vediamo solo questa forma semplificata, in cui la stringa **`format`** può contenere solo una specifica di conversione (in generale anche **`scanf`** è invece una funzione variadica)
  - Tale specifica controlla
    - l'interpretazione da dare ai caratteri letti da *stdin* per determinare il valore da memorizzare nella variabile passata come secondo argomento
    - il tipo che *si assume* abbia la variabile

scanf 2/2

Esempio:

```
int a ;
scanf("%d", &a) ; // equivale a cin>>a ;
```

- Si è utilizzata una delle seguenti specifiche di conversione:
  - %d Interpretare i caratteri sullo *stdin* come cifre di un numero intero, in notazione decimale, da memorizzare in un **`int`**
  - %lg Cifre di un numero reale, in notazione decimale, da memorizzare in un **`double`**
  - %c Carattere, tipicamente codifica ASCII, da memorizzare in un **`char`**
  - %s Stringa (lo spazio è un separatore), tipicamente codifica ASCII, da memorizzare in un **`char []`**

Errori

- Che succede se ci si sbaglia con le specifiche di conversione?
  - Errore logico
  - Errore di gestione della memoria
    - Era meno pericoloso nella **`printf`**
    - Estremamente dannoso nella **`scanf`**: corruzione della memoria
- Esempio:

```
char a ;
scanf("%d", &a) ; // corruzione della memoria
```

- Altro tipico errore molto pericoloso:

```
int a ;
scanf("%d", a) ; // corruzione della memoria
```

Confronto `scanf` e >>

- Con l'operatore >> non è necessario fornire specifiche di conversione perché l'operatore >> determina automaticamente il tipo dei valori, senza che sia necessario informarlo esplicitamente
  - Eliminata la possibilità di sbagliare tipo o fornire un indirizzo errato!
  - Dal tipo l'operatore >> decide autonomamente anche l'interpretazione da dare ai caratteri su *stdin*
    - Che si può comunque modificare attraverso manipolatori e funzioni membro opportune

Domanda

- Il controllo del tipo degli operandi effettuato automaticamente per gli operatori << o >> fa sì che il loro tempo di esecuzione sia maggiore di quello delle funzioni **`printf`** o **`scanf`**?

Risposta

- No, perché tale controllo e la conseguente scelta del codice da eseguire avviene a tempo di compilazione
- Si compila cioè direttamente (solo) il codice appropriato

File ed I/O formattato in C

- Per l'Input/Output formattato su file, in C si usano tipicamente due varianti di **`printf`** e **`scanf`**, chiamate **`fprintf`** ed **`fscanf`**
- In questo corso non vedremo dettagli su tali funzioni né in generale sull'Input/Output non formattato in C

Direttiva `#define`

- Solo negli ultimi standard del C è stato introdotto il qualificatore **`const`**
- Per definire costanti con nome in C si usa ancora spesso la direttiva **`#define`**
- Esempi:
```
#define a 5
#define b2 5.5
```
- E' una direttiva C/C++ per il preprocessore
- Comporta una sostituzione **testuale** del simbolo passato come primo argomento con qualsiasi sequenza di caratteri lo segua, prima della compilazione
  - Nessuna dichiarazione/controllo di tipo
    - Il simbolo sparisce **prima** della compilazione
  - Può essere utilizzata anche per sostituzioni più complesse

Tipo **`struct`** ed **`enum`** in C

- Anche in C si dispone dei tipi **`struct`** ed **`enum`**
- Però, data la dichiarazione di due tipi:

`struct` <*nome_tipo_struct*> { ... } ;
`enum` <*nome_tipo_enum*> { ... } ;

a differenza del C++, in C la definizione di oggetti dei due tipi va fatta ripetendo ogni volta rispettivamente **`struct`** ed **`enum`**:

`struct` <*nome_tipo_struct*> <*nome_variabile1*> ;
`enum` <*nome_tipo_enum*> <*nome_variabile2*> ;

`typedef` 1/2

- Sia in C che in C++ si possono definire dei sinonimi di tipi primitivi, oppure di tipi precedentemente dichiarati
  - Si fa mediante le dichiarazioni di nomi **`typedef`**
- Esempi:

Nuovo nome (sinonimo) per il tipo

```
typedef unsigned int u_int ;
u_int a ; // equivalente a unsigned int a ;

typedef struct persona Persona ;
Persona p ;

typedef enum colore colore_t ;
colore_t c ;
```

Solo in C vanno aggiunti **`struct`** ed **`enum`**

`typedef` 2/2

- Sia in C che in C++ le dichiarazioni typedef possono aiutare tantissimo a **migliorare la leggibilità** dei programmi
  - Permettono di evitare di dover ripetere in più punti dichiarazioni molto complesse
  - Permettono di sostituire nel programma nomi di tipo di basso di livello con nomi di tipo significativi nel dominio del problema
- Attenzione al fatto che nelle applicazioni in cui i problemi di *overflow* o in generale la conoscenza dei tipi di dato a basso livello sono importanti, le dichiarazioni **`typedef`** possono essere dannose
  - Perché non vedere direttamente il tipo di dato 'concreto' nelle dichiarazioni può rendere le cose più complicate

Allocazione array dinamici in C

- Mediante funzione di libreria **`malloc`**
  - presentata in **`<stdlib.h>`** (**`<cstdlib>`** se si vuole utilizzarla in C++)
  - prende in ingresso la dimensione, in byte, dell'oggetto da allocare
  - ritorna l'indirizzo dell'oggetto, oppure 0 in caso di fallimento (NULL in C)
- Allocazione di un array dinamico:

<nome_tipo> **`*`** *<identificatore>* **`=`**

`malloc(`*<num_elementi>* **`*`** **`sizeof(`***<nome_tipo>***`));`**

Deallocazione in C

- Mediante funzione di libreria **`free`**
  - presentata in **`<stdlib.h>`** (**`<cstdlib>`** se si vuole usarla in C++)
  - prende in ingresso l'indirizzo dell'oggetto da deallocare

- Deallocazione di un array dinamico:

  `free(`*<indirizzo_array>***`) ;`**

Confronto C/C++

- A differenza del C++, in C non ci sono operatori per allocazione/deallocazione della memoria, ma come si è visto due funzioni di libreria
- La funzione **`malloc`** opera ad un livello di astrazione più basso dell'operatore **`new`**
  - Alloca semplicemente una sequenza di byte, lunga quanto le comunichiamo
  - Al contrario all'operatore **`new`** possiamo chiedere esplicitamente di allocare un array di un certo numero di elementi di un dato tipo
    - Si preoccuperà lui di determinare il numero di byte necessari

Sorgenti in linguaggio C

- Suffisso tipico: .c
- Compilazione con gcc

```
gcc nome_file.c <altre-opzioni>
```