
# Linguaggi di Programmazione AA 2022-2023
# Introduzione a C e C++ (3)

Marco Antoniotti
Gabriella Pasi
Fabio Sartori

# Introduzione al C ed al C++ (3ª parte)

- Ulteriori elementi di C e C++
  - Modifiche alle dichiarazioni
    - Direttive per il linking (`extern`) e linking
    - Costanti e, per il C++, `const`
  - Input/Output e files

# Modificatori di dicharazione

- Per la libreria QUPCP ci ritroviamo alla fine le seguenti dichiarazioni

```
#ifndef _QUPCP_H
#ifndef _QUPCP_H

extern const int qupcp_N;
extern bool qupcp_find(int, int);
extern void qupcp_unite(int, int);

#endif
```

nel file di interfaccia, e le dichiarazioni

```
static int set_id_of[qupcp_N]
static int find_root(int x);
```

nel file di implementazione

# Modificatori di dicharazione

- Il due modificatori hanno il seguente significato
  - `extern`
    la dichiarazione seguente ha una definizione non locale, ovvero la definizione dell'oggetto dichiarato si trova più in là nel file od in un altro file
  - `static`
    la dichiarazione o definizione seguente viene "fissata" nello spazio di memoria globale e non è visibile al di fuori del file in cui essa appare
    - Il modificatore `static` ha anche un altro significato che vedremo successivamente
- Quindi, `extern` si usa essenzialmente per dichiarazioni da mettere in file di interfaccia, mentre `static` si usa soprattutto per definizioni globali in un file

# Modificatori di dicharazione

- Il modificatore `static` di fatto fissa la dichiarazione o definizione seguente nello “scope” cha la racchiude
- Variabili dichiarate `static` mantengono il loro valore tra una chiamata di funzione e la successiva
- Esempio

```
int foo() {
    static int x = 42;
    return x++;
}
```

questa funzione restituisce in sequenza gli interi

42, 43, 44, 45, …

# Costanti

- L'ultima versione di C (C18 ANSI) ed il C++ ci permettono di modificare le dichiarazioni di vari oggetti con l'attributo **`const`**
  - Solo le versioni più recenti di C ammettono questo attributo

- Esempi

```
const int qd = 42;                  // Costante intera.
const char s[10] = "qwerty";        // s[i] e` costante.
const double pi  = 3.14L;           // Constante `double'.
```

- Le costanti devono essere inizializzate (a meno che non siano dichiarate `extern`)
- Le conseguenze degli esempi precedenti sono le sequenti

```
qd = 3;             // Errore
s[2] = 'Z';         // Errore
```

# Costanti

- Questi sono esempi semplici
  - Quando si cominciano a considerare i puntatori e le funzioni, le complicazioni aumentano
  - Quando si usa un puntatore vi sono due oggetti da considerare
    - Il puntatore
    - L'oggetto puntato
  - La dichiarazione `const` deve poter distinguere tra i due

# Costanti

Esempi

```
char* p;
char  s[] = "Ford Prefect";

const char* pc = s;
pc[3] = 'X';                    // Errore.
pc = p;

char *const cp = s;
cp[3] = 'Y';
cp = p;                         // Errore.

const char *const cpc = s;
cpc[3] = 'Z';                   // Errore.
cpc = p;                        // Errore.
```

# Costanti

- Un importante uso di `const` è il seguente

```
char* strcpy(char* p, const char* q);
// Gli elementi di q non si possono modificare.
```

- Vi è un altro uso di `const` in congiunzione con “member functions” (metodi) di classi C++

- Nonostante la complessità derivante dalle combinazioni di const con puntatori di varia natura, è senz’altro molto utile usare costanti quando ve ne fosse la necessità (ovvero, in alternativa a `#define`)

# “Streams”, Input, Output e Files

- C e C++ hanno librerie molto sofisticate per la gestione di input, output e files

- Vedremo solo gli aspetti più importanti di queste librerie

# “Streams”, e “file handlers”

- Le librerie di I/O in C/C++ sono molto legate alla piattaforma sottostante, ed in particolare al “file system”

- In C tutte le operazioni di I/O coinvolgono *streams* (di solito associati a *files*)

- Nella libreria C (in **`stdio.h`**) esistono tre streams fondamentali
  - **`stdin`** standard input
  - **`stdout`** standard output
  - **`stderr`** standard error

- Altri streams vengono creati ed associati a strutture di tipo **`FILE`** (anch’esso definito in **`stdio.h`**) tramite le funzioni di libreria

# Output

- Le funzioni di output più semplici sono le seguenti
  - `int fputc(int c, FILE* ostream)`
    scrive il carattere `c` (notare la conversione) su `ostream`
  - `int fputs(const char* s, FILE* ostream)`
    scrive la stringa `s` su `ostream`
  - `int fprintf(FILE* ostream, const char* format, …)`
    scrive la stringa `format` su `ostream` dopo averla "interpretata" sulla base degli argomenti (in numero variabile) forniti
- La funzione **`printf`** che abbiamo già visto è equivalente a **`fprintf`** con il primo parametro pari a **`stdout`**

```
printf("foo %d\n", 42) === fprintf(stdout, "foo %d\n", 42)
```

# Output

- La funzione **`fprintf`** interpreta la stringa **`format`** sostituendo alle direttive '`%`' delle stringhe che dipendono dal parametro corrispondente
- La stringa format può contenere (*) le seguenti sotto-stringhe e direttive
  - `\n` stampa una "newline"
  - `\t` stampa un carattere di tabulazione
  - `%d` stampa un intero **`int`**
  - `%f` stampa un numero floating point **`float`**
  - `%s` stampa una stringa **`char*`**

(*) La specifica di **`fprintf`** è lunga 8 pagine nel documento ANSI C99

# “Streams” e “file handlers” in C++

- In C++ le operazioni di I/O coinvolgono *streams* che sono istanze di classi di libreria
- Nella libreria C++ (in **`iostream`**) esistono tre stream fondamentali
  - **`std::cin`** standard input
  - **`std::cout`** standard output
  - **`std::cerr`** standard error
- Altri streams vengono creati ed associati a files nel file system creando istanze delle classi **`ifstream`** (input file stream) e **`ofstream`** (output file stream)

# Output in C++

- In C++ si usa l'operatore '**<<**' (detto "put") per scrivere qualcosa su un output stream
- L'operatore è associativo a destra e prende due parametri di input
  - Un output stream
  - Un valore di un qualche tipo riconosciuto
- Esempi

```
cout << "Il numero " << 42 << endl;
cout << 3.14L << endl;
```

- La stringa `endl` è di fatto un puntatore ad una funzione, che l'operatore **<<** riconosce e richiama; il suo effetto è di scrivere un carattere newline sullo stream e di assicurarsi che esso sia "svuotato" (*flush*-ed)

# Output in C++

- Gli esempi precedenti sono da leggersi nel sequente modo

```
((cout << "Il numero ") << 42) << endl;

(cout << 3.14L) << endl;
```

# Output in C++

- L'operatore **<<** può essere specializzato a seconda dei casi; così come il metodo Java `Object.toString()`
- Ad esempio consideriamo la seguente struttura e supponiamo di avere a disposizione una versione specializzata dell'operatore **<<**

```
struct person {
  char name[80];
  int age;
  char trusted_assistant[80];
} p = {"Salvo Montalbano", 42, "Catarella"};
```

- L'espressione seguente stamperà

```
cout << p << endl;
==>
<Persona "Salvo Montalbano" "Catarella" 42>
```

# Output in C++

- La specializzazione dell'operatore **<<** (e di **>>**) ha come prerequisito la definizione delle nozioni di **`operator`** e di *overloading* in C++

# Input in C

- La gestione dell'input di un programma è sempre molto più complicata della gestione dell'output!
- In C le funzioni principali dedicate alla gestione dell'input sono le seguenti

  - `int` **`fgetc`**`(`**`FILE`**`* istream)`
    Legge un carattere da `istream` e lo restituisce

  - `char*` **`fgets`**`(char* s, int n,` **`FILE`**`* istream)`
    Legge al più `n` caratteri da `istream` nella stringa `s` e la restituisce; se c'è un newline o se `istream` è alla fine (end of file) l'operazione di lettura si ferma; un puntatore nullo viene restituito in caso di errore
    - Quindi bisognerebbe sempre controllare il risultato di una chiamata a **`fgets`**

# Input in C

- Esiste una funzione simmetrica a **`fprintf`**

  - `int` **`fscanf`**`(`**`FILE`**`* istream,`
    `const char* format,`
    `…)`

    Legge l'input da `istream` nella sotto il controllo del contenuto della stringa `format`; i parametri passati devono essere dei puntatori ad aree di memoria dove è possibile depositare il valore letto
    **`fscanf`** è in grado di interpretare (parse) le stringhe lette in input e di tradurle nel format interno corretto

  - La funzione **`scanf`** è simmetrica a **`printf`**

    **`scanf`**`("%d", &x) ===` **`fscanf`**`(`**`stdin`**`, "%d", &x)`

# Input in C

- Esempi

```
int read_int_from_stream(FILE* in) {
     int x = 0;
     fscanf(in, "%d", &x);
     return x;
}
```

la chiamata

```
int an_int = read_int_from_stream(stdin);
```

con input "42" (stringa) deposita il numero 42 nella variabile `an_int`

# Input in C

- Esempi

```
char* read_string_from_stream(FILE* in) {
     char* x = (char*) malloc(80 * sizeof(char));
     fscanf(in, "This is a %s", x);
     return x;
}
```

la chiamata

```
char* a_string = read_int_from_stream(stdin);
```

con input "`This is a 42`" (stringa) deposita la stringa "`42`" nella variabile `a_string`

# Input in C

- La descrizione di **`fscanf`** è lunga 7 pagine nella specifica ANSI C99

- Il comportamento di **`fscanf`** è piuttosto complicato, ma, al tempo stesso abbastanza sofisticato da permetterci di trattare formati di input di moderata complessità

# Input in C++

- In C++ si usa l'operatore '**>>**' (detto "get") per leggere qualcosa da un input stream

- L'operatore è associativo a destra e prende due parametri di input
  - Un input stream
  - Un puntatore (o una referenza) ad un oggetto di qualche tipo riconosciuto

- **Esempi**

```
int x;
cin >> x;
cout << "Il numero e`" << x << endl;
```

# Input in C++

- Un esempio più “complesso”

- Costruiamo un operatore speciale per la lettura di numeri complessi della forma
  *f*
  *(f)*
  *(f, f)*
  dove *f* è un numero “floating point”

- Assumiamo
  - L’esistenza di una classe **`complex`** con relativi costruttori
  - Di conoscere la semantica della creazione di operatori specializzati tramite la dichiarazione **`operator`**

# Input in C++: numeri complessi

```
istream& operator>>(istream& s, complex& a) {
      double re = 0, im = 0;
      char c = 0;

      s >> c;
      if (c == '(') {
            s >> re >> c;
            if (c == ',') s >> im >> c;
            if (c != ')') s.clear(ios_base::badbit);
      } else {
            s.putback(c);
            s >> re;
      }
      if (s) a = complex(re, im);
      return s;
}
```

# File I/O in C

- Le funzioni di input ed output agiscono su stream che possono essere associati a files
- Per associare uno stream ad un file si usa la funzione **`fopen`**
- Per rompere questa associazione (ovvero per chiudere) un file si usa la funzione **`fclose`**
- Un file può essere distrutto usando la funzione **`remove`**

# File I/O in C

- La funzione **`fopen`**

```
FILE* fopen(const char* filename, const char* mode);
```

- Il parametro filename è il nome (completo) del file da “aprire”

- Il parametro mode controlla come il file viene aperto
  - `"r"` apre un file testo in lettura
  - `"w"` apre azzerando, o crea un file in scrittura
  - `"a"` “append”; apre o crea un file di testo in scrittura (alla fine del file)

- **`fopen`** ritorna un puntatore ad uno stream **`FILE`** od il puntatore nullo se viene segnalato un qualche errore

# File I/O in C

- La funzione fclose

```
int fclose(FILE* stream);
```

- Semplicemente segnala al file system che il file associato a `stream` non verrà più usato dal programma (a meno di richiamare **`fopen`**)

- Il valore ritornato è `0` se la chiamata va a buon termine; **`EOF`** in caso contrario

# File I/O in C++

- In C++ l'input e l'output su files si basa sulla costruzione di istanze delle classi **`ifstream`** ed **`ofstream`** ("input file stream" ed "output file stream" definite in `<iostream.h>`)
- In quanto streams, tali istanze possono essere manipolate tramite gli operatori **>>** e **<<**
- **Esempio**
  Supponiamo che il file **`bar.txt`** contenga la stringa `42`

```
ifstream bar("bar.txt");
int qd;

bar >> &qd;
bar.close();
cout << qd;
```

la stringa `42` apparirà sullo standard output

# File I/O in C++

- **Esempio**

  Supponiamo che il file **`bar.txt`** contenga la stringa `42`

```
ofstream bar("bar.txt", "a");

bar << '\n' << "Vogons" << endl;
bar.close();
```

  a questo punto il file conterrà anche la stringa "`Vogons`", preceduta da un newline

# Abbreviazioni: `typedef`

- Il nome `FILE` definito in `<stdio.h>` è in realtà un'abbreviazione di una struttura più complessa
- Queste abbreviazioni si creano usando la direttiva **`typedef`** con una sintassi che ricorda le dichiarazioni

- **Esempi**

```
typedef char buffer[1024];
buffer x;    /* x e` dichiarata di fatto come char x[1024] */

struct _persona { char* nome; int eta; char coll[80]; };
typedef struct _persona* Persona;
Persona p = (Persona) malloc(sizeof struct _persona);

p->eta = 42;/* p e` di fatto un puntatore a struct _persona */
```

# Riassunto

- Dichiarazioni `extern` e `const`

- Input/Output
  - Streams
  - I/O in C
  - I/O in C++
    - Cenno all'estensibilità degli operatori

- Abbreviazioni con `typedef`