II Compitino di Programmazione (13.03.2013)

Domande di Teoria:

1) Considerate le due seguenti funzioni e il main() che le invoca:
```
void f(int*&a) {cout<<a[1]<<endl;}
void g(int*a) {cout<<a[2]<<endl;}
main() {int X[]={1,2,3}; f(X); g(X);}
```
Secondo voi c'è differenza tra *f* e *g*? Quale?
(Non si tratta dal fatto che stamperebbero elementi diversi di X)

2) Il *C++*, oltre al *cast* che eredita dal *C*, possiede alcune nuove operazioni di *cast* che sono diverse da quelle del *C*. Spiegare brevemente il motivo di questo fatto.

3) Cosa stampa il seguente programma?
```
int i=10;
for (i=0; i<4; i++)
 {
  cout<<i<<endl;
  if((i+3)%2)
   continue;
  else
   break;
 }
cout<<i<<endl;
```

Risposte (UFFICIALI*):

***** Esposte a lezione dal prof giovedì 14.03.2013, segue un report di quanto emerso.

1) (la risposta, giusta, vale 2 punti)
Le due funzione *f* e *g* differiscono poiché richiedono in ingresso un parametro *"a"* che rispettivamente in *f* è passato per riferimento e in *g* è passato per valore.
Si noti però che il parametro *"a"* della funzione *f* dovrebbe essere a tutti gli effetti un *alias* di X; ciò comporta delle complicazioni, tant'è che compilando questo:

```
1:  #include <iostream>
2:  using namespace std;
3:  void f(int*&a) {cout<<a[1]<<endl;}
4:  void g(int*a) {cout<<a[2]<<endl;}
5:  main() {int X[]={1,2,3}; f(X); g(X);}
```

Si ottiene il seguente messaggio di errore dal compilatore:
In function 'int main()':
[line]5: error: invalid initialization of non-const reference of type 'int&' from a temporary of type 'int*'*
[line]3: error: in passing argument 1 of 'void f(int&)'*

Ciò perchè la dichiarazione di un array (come *int X[]={1,2,3}*) è una costante.

Istruzioni come *X++;* sono rigettate dal compilatore poiché comporterebbero la perdita della "porta d'ingresso" all'array costituita dal suo stesso nome (appunto X).
Per evitare rischi il compilatore considera X come un puntatore costante al primo elemento dell'array.
Dunque se fosse possibile creare un alias (*"a"*) non costante di X, tramite la funzione (non costante) che lo riceve potremmo modificare suddetto alias e di riflesso anche X.
Ciò è risolvibile marcando come costante anche l'alias in tal modo il compilatore rassicurato non rigetterebbe più la funzione *f*.

2) (la risposta, giusta, vale 2 punti)
Per una risposta dettagliata ed esauriente si legga:
Cap. 9, Paragrafo 4 (9.4) "Conversione ed operatori di cast" dal pag. 124 a pag. 128 del libro.
In aggiunta si consideri che:
Il *C* ha un solo *cast*, della forma *(type)expression* (dove *type* è il tipo in cui si vuole convertire *expression*); tale operatore si basa sulla filosofia fondante del *C* ovvero *"il programmatore sa cosa sta facendo"*; per tale motivo il *cast alla C* viene sempre eseguito a prescindere dall'obbrobrio o meno che esso può comportare, il compilatore se ne lava le mani.
Il *C++* mette a disposizione più tipi di *cast*, essi sono più limitati e più controllati dal compilatore. Tutto ciò al fine di costringere il programmatore a riflettere bene su ciò che vuole ottenere con quella precisa richiesta di *cast* e poi di richiederla in modo esplicito, se la richiesta apparirà non coerente al compilatore esso la segnalerà obbligando il malcapitato ad un'ulteriore riflessione su cosa stia combinando.

3) (la risposta, giusta, vale 3 punti)
Fondamentalmente si tratta di accorgersi che c'è una ed una sola variabile *"i"* e dunque sia il blocco del *main()* (che la inizializza con *int i=10;*) sia il blocco del ciclo *for* (che al primo ingresso la azzera con *i=0*) utilizzano la stessa *"i"*.
Un eventuale errore di interpretazione è causato dalle istruzioni che seguono il *for*, nello specifico non è ***(int i=0**; i<4; i++)* ma semplicemente ***(i=0**; i<4; i++)*.
Morale: il programma compila, o meglio quanto segue compila:

```
1:    #include <iostream>
2:    using namespace std;
3:    main()
4:    {
5:    int i=10;
6:     for (i=0; i<4; i++)
7:      {
8:       cout<<i<<endl;
9:       if((i+3)%2)
10:       continue;
11:      else
12:       break;
13:     }
14:    cout<<i<<endl;}
```

e l'output è:
0 1 1

o meglio:
0 "endl" 1 "endl" 1 "endl"
cioè:
0
1
1

PS:
Per chi non conoscesse i seguenti comandi:
- *continue*: quando viene incontrato fa saltare alla fine del ciclo in cui è inserito, provvede ad incrementare il contatore se previsto (com'è nel ciclo *for*) e poi torna ad eseguire il ciclo dal principio, sostanzialmente permette di ignorare tutte le istruzioni che lo seguono fino alla fine del blocco.
- *break*: quando incontrata interrompe il programma e lo fa riprendere appena fuori dal ciclo in cui è inserito.
- *%*: operatore che restituisce un valore numerico, precisamente il resto della divisione tra il numero che lo precede e quello che lo segue, perciò *X%Y* è uguale al resto della divisione di *X* per *Y*.

PPS:
Per quanto detto a riguardo dell'istruzione *"continue"* il seguente codice è identico (fa e stampa esattamente la stessa cosa) di quello proposto nel compitino dal quale varia per l'omissione della sola istruzione *"else":*

```
1:    #include <iostream>
2:    using namespace std;
3:    main()
4:    {
5:    int i=10;
6:     for (i=0; i<4; i++)
7:      {
8:       cout<<i<<endl;
9:       if((i+3)%2)
10:       continue;
11:       break;
12:     }
13:    cout<<i<<endl;}
```