# Osservazioni sulla programmazione in C

Moreno Marzolla

*Versione 1.2 del 14/6/2016*
Seconda revisione
*Versione 1.1 del 11/2/2015*
Prima revisione, aggiunta “bonus track”
*Versione 1.0 del 10/2/2015*
Prima versione

*Immagine da Wikipedia*

“*Io ho visto cose che voi umani non potete immaginare:*
*array dinamici in fiamme al largo dei bastioni di Orione.*
*E ho visto errori logici balenare nel buio vicino alle porte di Tannhäuser.*
*E tutti quei momenti andranno perduti nel tempo, come lacrime nella pioggia.*”

Moreno Marzolla, al termine della correzione dei progetti di Algoritmi Avanzati

L'*incipit* un po' melodrammatico di questo documento serve solo ad attirare l'attenzione, dato che le valutazioni dei progetti del corso di Algoritmi Avanzati sono state più che positive. Ho apprezzato l'impegno dimostrato nello svolgimento del progetto (che comunque era piuttosto semplice), ma ho anche “visto cose che voi umani non potete immaginare”, sotto forma di gravi errori di programmazione che non mi sarei aspettato da studenti della laurea Magistrale. In questa sessione ho valutato “bonariamente” (cioè, ho penalizzato poco) questo tipo di errori, concentrandomi sugli aspetti più legati al corso (scelta delle primitive di comunicazione, analisi delle prestazioni, e simili). Dalla prossima tornata di progetti non intendo continuare a sorvolare.
Ricordo che la conoscenza del linguaggio C è un prerequisito essenziale di questo modulo; a lezione avevo fornito un riferimento bibliografico (Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language 2nd edition*, conosciuto semplicemente come *K&R*) per coloro che non si sentono a proprio agio con questo linguaggio. In questo documento voglio essere più esplicito, proponendo un semplice quiz per aiutarvi a capire se il vostro livello di padronanza del C è sufficiente. Qui sotto trovate tre brevissimi programmi, tutti ispirati a codice che ho visto nei progetti, che compilano correttamente con `gcc -Wall` ma contengono ciascuno un errore grave, o un aspetto comunque critico. Esaminateli con attenzione: se non vi salta all'occhio nulla di strano, vi suggerisco di approfondire la conoscenza del C prima di mettervi a scrivere il firmware di un pacemaker, o il software del pilota automatico di un aereo (o consegnare il prossimo progetto di Algoritmi Avanzati!). Le risposte sono presenti in fondo a questo documento.
Invito **tutti** gli studenti a provare questo test di autovalutazione. Coloro che consegneranno i prossimi progetti scopriranno alcuni errori da evitare; chi ha già consegnato il progetto consideri queste note come un monito a tenere sempre alto il livello di attenzione quando si scrive codice.

## Primo programma

Questo programma contiene un errore. Riuscite a individuarlo?

```
#include <stdio.h>
int main( void )
{
	float v[5] = {-1.0, 0.3, 9.8, 0.2, 0.9};
	int i;
	/* scan inclusivo di v, memorizzando il risultato in v stesso */
	for ( i=0; i<5; i++ ) {
		v[i] = v[i] + v[i-1];
	}
```

```
        printf(“l'ultima somma prefissa vale %f\n”, v[4]);
        return 0;
}
```

## Secondo programma

Questo programma dovrebbe calcolare la somma dei valori dell'array *v*. Contiene tuttavia un errore piuttosto subdolo, nonostante sembri funzionare correttamente sulle piattaforme che ho provato. Riuscite a individuare il problema?

```
#include <stdio.h>
int main( void )
{
        float v[5] = {-1.0, 0.3, 9.8, 0.2, 0.9};
        float s;
        int i;
        for ( i=0; i<5; i++ ) {
                s += v[i];
        }
        printf(“la somma vale %f\n”, s);
        return 0;
}
```

## Terzo programma

Questo programma, di per se, puo' considerarsi corretto. Tuttavia contiene un problema potenzialmente insidioso. A cosa è necessario prestare attenzione?

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

/* restituisce un nuovo array di n elementi contenente le somme prefisse di v. Assumo n>0 */
float* scan(float* v, int n)
{
        float* s = (float*)malloc( n * sizeof(float) );
        int i;
        s[0] = v[0];
        for (i=1; i<n; i++) {
                s[i] = s[i-1] + v[i];
        }
        return s;
}

int main( void )
{
        float v[5] = {-1.0, 0.3, 9.8, 0.2, 0.9};
        float* s = scan(v, 5);
        printf(“l'ultima somma prefissa vale %f\n”, s[4]);
        return 0;
}
```

## Bonus Track

La funzione seguente calcola le somme prefisse inclusive dell'array *v* di *n* elementi passato come parametro; il risultato viene memorizzato in *v*. La funzione non presenta errori, ma il codice è indicativo di un ragionamento poco “lineare”. Qualche osservazione?

```
void scan( float* v, int n )
{
        int i;
        for ( i=0; i<n; i++ ) {
                if ( i>0 ) {
                        v[i] = v[i] + v[i-1];
                }
        }
}
```

## Soluzioni

Il primo programma presenta un accesso *out-of-bound* all'array *v*; infatti alla prima iterazione del ciclo (*i* = 0) si accede a *v*[-1], che ovviamente risulta al di fuori dello spazio di memoria allocato per *v*. Il comportamento del programma è indefinito: potrebbe funzionare (sul mio sistema funziona e stampa il risultato corretto), potrebbe produrre un risultato errato, oppure potrebbe produrre un crash del processo, in base alla combinazione compilatore / sistema operativo / processore usato. Il programma diventa corretto se la variabile *i* parte da 1 anziché da 0.

Nel secondo programma la variabile *s* non è inizializzata, ciò significa che ha valore *indefinito* in base alla specifica del linguaggio (http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1256.pdf pagina 126 paragrafo 10: "*If an object that has automatic storage duration is not initialized explicitly, its value is indeterminate*"). Il programma produce comunque il valore corretto sul mio sistema, ma non è garantito che con una diversa combinazione di compilatore / sistema operativo / processore il risultato sia lo stesso. Molti compilatori possono generare un warning in presenza di codice che fa uso di variabili non inizializzate; nel caso di gcc è necessario usare il flag `-Wuninitialized`; purtroppo ci possono essere casi in cui l'uso di variabili non inizializzate non viene correttamente riconosciuto dal compilatore, per cui è sempre necessario prestare molta attenzione. E' interessante osservare che altri linguaggi di programmazione, ad esempio Java, inizializzano sempre le variabili con valori di default. Nel caso di Java, le variabili numeriche sono automaticamente inizializzate a zero, quindi la versione Java del programma funzionerebbe correttamente.

Il terzo programma non è tecnicamente sbagliato, ma contiene una insidia. Notate che la funzione *scan*() alloca dinamicamente (cioè sullo heap con *malloc*) un nuovo array *s*, lo inizializza correttamente, e restituisce al chiamante il puntatore al primo elemento. Tale array pero' non viene mai deallocato con la funzione *free*. Nel nostro caso non è un problema, dato che la terminazione del programma rilascia comunque tutta la memoria occupata dal processo. Immaginate pero' cosa succederebbe se *scan*() venisse invocata all'interno di un ciclo, passando in input array di grandi dimensioni: ad un certo punto il programma potrebbe esaurire la memoria a disposizione.

Veniamo infine alla "Bonus Track". Come scritto, non c'è nulla di sbagliato nella funzione `scan()` proposta, che anzi evita il problema di accesso out-of-bound di cui al primo programma. Tuttavia, il test "if (i>0)" puo' essere facilmente omesso se facciamo partire l'indice *i* da 1 anziché da 0:

```
void scan( float* v, int n )
{
	int i;
	for ( i=1; i<n; i++ ) {
		v[i] = v[i] + v[i-1];
	}
}
```

Molti di voi saranno tentati di bollare questo esempio come "cercare il pelo nell'uovo". Ripensateci: in ogni riga di codice si può celare un bug; più codice superfluo c'è, maggiore è la probabilità di introdurre errori. In particolare, la presenza di costrutti condizionali superflui aumenta la complessità ciclomatica del programma, aumentando il numero di casi di test da approntare per testare tutti i percorsi del grafo di flusso (Arthur H. Watson and Thomas J. McCabe (1996). *"Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric"*. NIST Special Publication 500-235).
In generale, codice ridondante o poco comprensibile non ha solo effetti "estetici", ma ha anche un costo reale e misurabile: (1) è più difficile da comprendere da parte di terzi, (2) puo' avere impatto negativo sulle prestazioni, (3) aumenta la complessità della fase di test, e (4) può introdurre errori.