Alcuni dei migliori modi per farsi male con il linguaggio "C"

Massimo Bernaschi

Istituto per le Applicazioni del Calcolo "Mauro Picone"

Consiglio Nazionale delle Ricerche

Viale Manzoni, 30 - 00185 Roma - Italy

`http://www.iac.cnr.it/~massimo`

e-mail: *m.bernaschi@iac.cnr.it*

Web Resources

- `http://www.eskimo.com/~scs/C-faq/faq.html`
- `http://www.andromeda.com/people/ddyer/topten.html`
- scaricare gli esempi: `http://twin.iac.rm.cnr.it/cppf.tgz`
  (da linea comandi si può utilizzare `wget`)

per spacchettarli: `tar zxvf cppf.tgz`

per accederli: `cd cppf`

Libri

- A Book on C: Programming in C (4th Edition)
  by Al Kelley, Ira Pohl
  Publisher: Addison-Wesley Pub Co; 4th edition (December 29, 1997) ISBN: 0201183994

- C: A Reference Manual (5th Edition)
  by Samuel P., III Harbison, Guy L., Jr. Steele
  Publisher: Prentice Hall; 5th edition (February 21, 2002) ISBN: 013089592X

- Expert C Programming
  by Peter van der Linden
  Publisher: Prentice Hall PTR; 1st edition (June 14, 1994) ISBN: 0131774298

- C Traps and Pitfalls
  by Andrew Koenig
  Publisher: Addison-Wesley Pub Co; 1st edition (January 1, 1989) ISBN: 0201179288

Gli sviluppatori di compilatori hanno cercato di aumentare la produttività dei programmatori C rilevando e “notificando” (tramite *warning*) frammenti di codice *potenzialmente* errati.

1. **Commenti non-terminati** , “accidentalmente” terminati da successivi commenti:

   ```
   a=b; /* this is a bug
   c=d; /* c=d will never be executed */
   ```

   Esempio: comment.c

   Per compilare: `gcc -c comment.c`
   Aggiungere poi l’opzione `-Wall`: `gcc -c comment.c -Wall`

2. **Un classico...**
   Accidental assignment/Accidental Booleans

```
if(a=b) c;      /* a always equals b, but c will be executed if b!=0 */
```

A seconda del punto di vista, il “baco” è che l’operatore di assegnazione si confonde troppo facilmente con l’operatore di confronto.

Oppure che il C non presta troppo attenzione a ciò che costituisce un’espressione booleana (o logica):
(a=b) non è un’espressione booleana! (ma il C *doesn’t care*).

Considerate il seguente frammento di codice:

```
if( 0 < a < 5) c;  /* Cosa fa questo statement? */
```

3. **Unhygienic macros 1**

```
#define average(a,b) a*0.5+b*0.5
average(x, y+128)
```

diventa

x*0.5+y+128*0.5 /* una strana media... */

Quale è la soluzione?

(Cioè la regola da ricordare quando si scrive una macro)

4. Unhygienic macros 2

```
#define MAX(a,b) ((a)>(b)?(a):(b))


        biggest = x[0];
        i = 1;
        while (i < n) {
           biggest = MAX (biggest, x[i++]);
        }
```

Supponiamo di avere `x[0]=2, x[1]=3, x[2]=1`.
Funziona il programma? (Provare **macro.c**).

5. Phantom returned values

Supponiamo di avere il seguente frammento di codice:

```
int foo(int a) {
    ...
    if (a) return(b);
} /* buggy, because sometimes no value is ret.  */
```

Anche qui il compilatore può segnalare che c’è qualcosa che non va se è attivo il giusto livello di diagnostica.
Quello che accade all’esecuzione dipende dal particolare compilatore ed è perfettamente possibile che il programmi “sembri” funzionare correttamente.
Immaginate quello che può succedere se la funzione è pensata per ritornare un puntatore!

6. **Indefinite order of evaluation**

```
foo(pointer->member, pointer = &buffer[0]);
```

Funziona come “ci si aspetta” con il **gcc** ma, possibilmente, non con altri compilatori. La ragione è che il **gcc** valuta gli argomenti di funzione da sinistra a destra ma...

il K&R e le specifiche ANSI/ISO C **non** definiscono l’ordine di valutazione degli argomenti di funzione. Può essere `sinistra-destra`, `destra-sinistra` o qualsiasi altra cosa. Questo significa non avere portabilità neanche all’interno della stessa piattaforma!

7. **Easily changed block scope**

```
if( ... )
  foo();
else
  bar();
```

Se si aggiunge, ad esempio, uno *statement* per il debugging, diventa

```
if( ... )
  foo();
else
  printf( "Calling bar()" ); /* oops! l'else si ferma qui */
  bar();                     /* oops! bar viene sempre eseguita */
```

C'è un'ampia classe di errori simili legati all'uso non corretto dei delimitatori.

Il motto del `C`: *who cares what it means? I just compile it!*

Un esempio “illuminante”:

```
switch (a) {
int var = 1;

case A: ...
case B: ...
}
```

Esempio: leggere, compilare ed eseguire **switch.c**.

8. Unsafe returned values

```
 char *f() {
   char result[80];
   sprintf(result,"anything will do");
   return(result); /* Oops! result e' allocato sullo stack! */
}

int g() {
   char *p;
   p = f();
   printf("f() returns: %s\n",p);
}
```

Anche questo codice bacato molte volte sembra funzionare!
Almeno fino a quando non viene riutilizzato il particolare blocco di stack occupato da *result*.

9. **Undefined order of side effects.**

Anche all’interno di una singola espressione, il C non definisce l’ordine dei *side effects*.
Come conseguenza, a seconda del compilatore, `i/++i` potrebbe valere 0 oppure 1.
Vediamo questo frammento di codice:

```
#include <stdio.h>

int foo(int n) {printf("Foo got %d\n", n); return(0);}
int bar(int n) {printf("Bar got %d\n", n); return(0);}

int main(int argc, char *argv[]) {
  int m = 0;
  int (*(fun_array[3]))();

  int i = 1;
  int ii = i/++i;

  printf("\ni/++i = %d, ",ii);

  fun_array[1] = foo; fun_array[2] = bar;

  (fun_array[++m])(++m);
}
```

Stampa `i/++i` = 1 o `i/++i`=0;

Stampa “Foo got 2” o “Bar got 1”

Esempio: foobar.c (compilare con e senza l’opzione -Wall)

Più in generale,

10. **Il seguente statement è ben definito?**

```
a[i] = i++;
```

11. Uninitialized local variables

Questo è un altro baco classico...
Consideriamo prima il caso più semplice:

```
void foo(a) {
     int b;
     if(b) {/* bug! b is not initialized! */ }
}
```

Ormai qualsiasi compilatore “decente” indica un errore ovvio come il precedente.
Comunque ci vuole poco per essere più “furbi” del compilatore:

```
void foo(int a)  {
   char *p;
   if(a) { p=malloc(a); }
   if(p) { *p=...; } /* BUG! p potrebbe non essere inizializzato */
}
```

12. È possibile confrontare direttamente due strutture?

NO. Non esiste una maniera semplice per un compilatore di implementare un confronto implicito per strutture (cioè di supportare l’operatore ==).

Notare come sia però possibile usare l’operatore di assegnamento =.

13. **Qual è il risultato di questa operazione?**

```
double half = 1/2;
```

14. **Il seguente programma è corretto?**

```
#include <stdio.h>
#include <math.h>
hsqrt(double arg) {
        return sqrt(arg)/2.;
}
int main(int argc, char *argv[]) {
       double halfroot=hsqrt(2.0);
       printf("Half root equal %f\n",halfroot);
       return 0;
}
```

Esercizio: compilare ed eseguire **halfroot.c**

15. **Quale dei seguenti statement è corretto?**

```
if (flags & FLAG)
```

```
if (flags & FLAG != 0)
```

Quale è il loro effetto?

16. Precedenza (ed associatività) degli operatori

- Gli operatori che hanno precedenza più alta sono quelli che non sono operatori veri e propri: subscripting, chiamate a funzione e selezione di elementi di strutture. Questi associano tutti a sinistra.
- Vengono quindi gli operatori unari. Gli operatori unari sono associativi a destra
- Vengono quindi i veri operatori binari (con due operandi)
  - Ogni operatore logico ha precedenza minore rispetto ad ogni operatore di comparazione.
  - Gli operatori di shift hanno precedenza maggiore degli operatori di comparazione ma inferiore a quelli aritmetici.
- L’assegnamento è un operatore con il quale si fanno spesso errori:

```
while (c=getc(stdin) != EOF) { putc(c, stdout); ... }
```

Solo i 4 operatori &&, ||, ?:, e , specificano un ordine di valutazione.

- `&&` e `||` valutano prima l’operando a sinistra poi quello a destra solo se necessario.
- L’operatore ?: prende tre operandi `a ? b : c`; valuta `a` prima, e quindi valuta o `b` o `c`, a seconda del valore `a`.
- L’operatore `,` valuta l’operando a sinistra, scarta il valore e quindi valuta l’operatore a destra.

Tutti gli altri operatori C valutano gli operandi in un ordine “indefinito”. Ad esempio il seguente meccanismo per copiare i primi n elementi di x in y non funziona:

```
i = 0;
while(i<n) { y[i] = x[i++]; }
```

()	[]	->	.								*l2r*
!	~	++	–	+	-	*	&			sizeof	*r2l*
*	/	%									*l2r*
+	-										*l2r*
<<	>>										*l2r*
<	<=	>=	>								*l2r*
==	!=										*l2r*
&											*l2r*
^											*l2r*
|											*l2r*
&&											*l2r*
||											*l2r*
?:											*r2l*
=	+=	-=	*=	/=	%=	&=	^=	—=	<<=	>>=	*r2l*
,											*l2r*

17. **Il seguente frammento di codice è corretto?**

```
struct foo {
        int x;
        ......
}

f(int y) {
        ......
}
```

18. **Il seguente frammento di codice è corretto?**

```
((condition) ? a : b) = complicated_expression;
```

19. ***p++ incrementa p, o ciò a cui punta p?**

20. **Qual è la differenza tra queste inizializzazioni (ammesso che ci sia una differenza)?**

```
char p[] = "string literal";
char *p  = "string literal";
```

e le seguenti?

```
char ch = 'A';
char ch = "A";
```

21. **Qual è il risultato di sizeof('a')?**
22. **Operatori di Shift**
    - In uno shift a destra, i bit “mancanti” sono sostituiti con zeri o con copie del bit di segno?
    - Quali valori sono permessi per il contatore di shift?

La risposta alla prima domanda è semplice ma in qualche modo *implementation-dependent*.

Se il valore shiftato è `unsigned`, le posizioni vacanti sono riempite con zeri.

Se il valore è `signed`, le posizioni vacanti possono essere riempite con zeri **oppure** con copie del bit di segno.

La risposta alla seconda domanda è (per una volta...) intuitiva: se l'oggetto da shiftare è lungo `n` bit, allora lo shift è $>= 0$ e $<$ `n`.

Esercizio: leggere, compilare ed eseguire **shift.c**

23. Puntatori a funzioni

Un array di puntatori a funzione può essere dichiarato in questo modo:

```
void ( *foo[4] ) ();
```

Se si conoscono i nomi delle funzioni, si può inizializzare l’array in questo modo:

```
extern int a(), b(), c(), d();
int ((*foo)[]) () = {a, b, c, d};
```

Una funzione (ad esempio `b()`) può essere quindi invocata attraverso un puntatore contenuto nell’array in almeno due modi:

```
(*foo[1])();
foo[1]();
```

24. Variabili ed inizializzazione di array

Si **può** inizializzare come segue:

```
char *fruit="apple"
char *fruits[]={"apple","orange","banana","strawberry"};
```

Non si può inizializzare come segue:

```
double *p=3.14;
int *lists[] = { {1, 2, 3, 4}, {5, 6, 7} };
```

Esercizio: leggere, compilare ed eseguire **initialize.c**.

Eliminare l’errore in fase di compilazione.

Modificare la macro per abilitare il ramo `else` dell’esecuzione condizionale.

25. Arrays e Puntatori sono la stessa cosa, o no?

array.c:

```
#define LEN 4
int list[LEN] = {1, 2, 3, 4};
```

pointer.c:

```
extern int *list;
main() {
        printf("Third element of list is %d\n",list[2]);
}
```

Esercizio: compilare ed eseguire **array.c pointer.c**:

```
gcc -o arraypointer array.c pointer.c
```

26. **Array e Puntatori (riassunto dell’interscambialità)**

- Un accesso ad un array `a[i]` è sempre “riscritto” o interpretato dal compilatore come un accesso tramite puntatore `*(a+i)`.
- I puntatori sono sempre soltanto puntatori!
  Non sono mai riscritti come array.
- Un array che è argomento di funzione è sempre modificato, dal compilatore, in un puntatore all’array.
- In tutti gli altri casi, le definizioni devono essere coerenti con le dichiarazioni!

27. Array Multidimensionali

```
int ma[4][6];
```

```
          [0] [1] [2] [3] [4] [5]
    ma[0] | 0 | 1 | 2 | 3 | 4 | 5 |
      [1] | 6 | 7 | 8 | 9 | 10| 11|
      [2] | 12| 13| 14| 15| 16| 17|
      [3] | 18| 19| 20| 21| 22| 23|

    ma[0] | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10| 11|
```

`ma[i][j]` è equivalente a `* ( *(ma + i) + j );`

Esercizio: leggere, compilare ed eseguire **multiarr.c**

28. **Passaggio di array multidimensionali ad una funzione**.

- Un array unidimensionale di qualsiasi tipo può essere usato come argomento di una funzione.

  Il parametro è riscritto come puntatore al primo elemento.

- Bi (o multi) dimensionali array sono più *tricky*, dal momento che l'array è riscritto come puntatore alla **prima** riga.

  Non c'è modo per passare un generico array multidimensionale ad una funzione.

  È necessario indicare la dimensione di una singola riga.

```
main(...) {
   int m[32][16]...
   ...
   f(m);
}

f(int m[][16]) {
}
```

29. **Attenzione a non lasciare caratteri nel buffer di input!**

Considerate il seguente codice:

```
int x;
char st[31];

printf("Enter an integer: ");
scanf("%d", &x);
printf("Enter a line of text: ");
fgets(st, 30, stdin);
```

Usare sempre **fgets()** e quindi **sscanf**
(variante di `scanf` che legge da stringa).

```
int x;
char st[31];
printf("Enter an integer: ");
fgets(st, 30, stdin);
sscanf(st,"%d", &x);
printf("Enter a line of text: ");
fgets(st, 30, stdin);
```

Esercizio: leggere, compilare ed eseguire **readinp.c**.

Definire la macro `USEFGETS` e riprovare.

30. `char *s, *t, *u`

    è uguale a

    `char* s, t, u`?

31. Cosa significa `a+++++b`?

32. ```
    if (a == 1)
        if (b == 2)
            printf("***\n");
        else
            printf("###\n");
    ```

    l'`else` a quale `if` fa riferimento?

    Esercizio: leggere, compilare ed eseguire **ifif.c**
    ```

33. Cosa fa questo frammento di codice?

```
while (c == '\t' || c = ' ' || c == '\n') c = getc(f);
```

Meglio scrivere:

```
while ('\t' == c || ' ' = c || '\n' == c) c = getc(f);
```

cioè mettere a sinistra le costanti.

Esercizio: leggere e compilare **getc.c**

34. **Come dichiarereste un array di puntatori a funzione che ritornano puntatori a funzione che ritornano puntatori a carattere?**

Si può usare il programma `cdecl` che “traduce” l’inglese in C e vice versa:

```
cdecl> declare a as array of pointer to function returning
       pointer to function returning pointer to char
char *(*(*a[])())()
```

Come dichiarare e definire variabili globali e funzioni

Ci possono essere molte “dichiarazioni” (in diverse “unità di compilazione”) di una singola variabile o funzione globale (o meglio `extern`) ma ci deve essere esattamente una “definizione”.

(La definizione è la dichiarazione che alloca spazio ed eventualmente fornisce un valore di inizializzazione).

La soluzione migliore è avere ogni definizione in un file `.c` specifico (ad esempio `global.c`) con una dichiarazione che è in un header file `.h` (ad esempio `extern.h`) incluso ogni volta che la dichiarazione è necessaria.