INTEGER OVERFLOW IN C

LEONARDO TAMIANO

TABLE OF CONTENTS

INTRODUZIONE

Consideriamo il seguente snippet di codice C

```c
#include <stdio.h>

int main(int argc, char **argv) {

  int normal_value = 4321;
  int overflowing_value = (int) (4294967296);

  printf("[INFO] - Normal value      = %d\n", normal_value);
  printf("[INFO] - Overflowing value = %d\n", overflowing_value);

  return 0;
}
```

Una volta eseguito otteniamo la seguente risposta

```
[INFO] — Normal integer value       = 4321
[INFO] — Overflowing integer value  = 0
```

Notiamo che anche se avevamo assegnato alla variabile `overflowing_value` il valore `4294967296`, alla fine il valore della variabile una volta stampato è `0`.

$$4294967296 \longrightarrow 0$$

Se invece assegnavamo il valore `4294967296 + 1` il valore stampato sarebbe stato `1`.

$$4294967297 \longrightarrow 1$$

In questi casi diciamo che la variabile è andata in **overflow**.

Dato poi che la variabile è un intero, si parla di **integer overflow**.

Perché succede questo?

SULLA FINITEZZA DELLA MEMORIA

Per capire questo tipo di comportamento dobbiamo ricordarci il fatto che

la memoria di un computer è una risorsa finita.

In particolare questo significa che c'è sempre un limite superiore a tutto ciò che possiamo memorizzare tramite un computer.

Questo fatto, che potrebbe sembrare banale, ha molte conseguenze. Tra queste troviamo anche gli **integer overflows** mostrati nell'esempio in precedenza.

Nei linguaggi di programmazione a basso livello come C/C++, ogni variabile ha una dimensione limitata.

Nelle architetture hardware moderne questa dimensione viene specificata in **bytes**.

Un singolo byte contiene **8 bit**, e permette di rappresentare i numeri da 0 a 255.

$$
\begin{aligned}
00000000 &\longrightarrow 0 \\
00000001 &\longrightarrow 1 \\
00000010 &\longrightarrow 2 \\
&\vdots \\
11111111 &\longrightarrow 255
\end{aligned}
$$

SIZEOF()

In C possiamo utilizzare l'operatore `sizeof()` per vedere la dimensione, intesa come numero di bytes, associata alle variabili di un particolare tipo.

```c
#include <stdio.h>

int main(int argc, char **argv) {

  printf("[INFO] - Size of char      = %d\n", sizeof(char));
  printf("================================\n");
  printf("[INFO] - Size of short     = %d\n", sizeof(short));
  printf("[INFO] - Size of int       = %d\n", sizeof(int));
  printf("[INFO] - Size of long      = %d\n", sizeof(long));
  printf("[INFO] - Size of long long = %d\n", sizeof(long long));
  printf("================================\n");
  printf("[INFO] - Size of float     = %d\n", sizeof(float));
  printf("[INFO] - Size of double    = %d\n", sizeof(double));

  return 0;
}
```

Eseguendo il codice otteniamo

```
[INFO] – Size of char      = 1
============================
[INFO] – Size of short     = 2
[INFO] – Size of int       = 4
[INFO] – Size of long      = 8
[INFO] – Size of long long = 8
============================
[INFO] – Size of float     = 4
[INFO] – Size of double    = 8
```

Il fatto che il size di una variabile di tipo `int` è `4`, significa che ad ogni variabile di tipo `int` saranno associati `4` particolari byte della memoria.

NOTA BENE: Il particolare size associato ad ogni variabile non è fisso, ma dipende da varie cose, tra cui:

- dal compilatore.
- dal sistema operativo.
- dall'architettura hardware sottostante.

EXTRA: INDIRIZZO DELLE VARIABILI LOCALI

Per vedere l'indirizzo in memoria di una variabile possiamo utilizzare l'operatore &.

```c
#include <stdio.h>

int main(int argc, char **argv) {
  int var1 = 10;
  int var2 = 20;

  int *addr1 = &var1;
  int *addr2 = &var2;

  printf("[INFO] - Value of var1   = %d\n", var1);
  printf("[INFO] - Value of var2   = %d\n", var2);
  printf("============================\n");
  printf("[INFO] - Address of var1 = %p\n", addr1);
  printf("[INFO] - Address of var2 = %p\n", addr2);

  return 0;
```

Eseguendo il codice otteniamo

```
[INFO] – Value of var1   = 10
[INFO] – Value of var2   = 20
============================
[INFO] – Address of var1 = 0x7ffdd52b4830
[INFO] – Address of var2 = 0x7ffdd52b4834
```

Notiamo in particolare che la differenza tra gli indirizzi è proprio di 4 bytes.

```
0x7ffdd52b4834 - 0x7ffdd52b4830 = 0x4 = 4
```

Questo significa che nella memoria le variabili `var1` e `var2` sono memorizzate una dopo l'altra.

ESEMPI PRATICI

Andiamo adesso a vedere qualche esempio pratico.

OVERFLOW #1: INT

Per mandare in overflow un intero ci dobbiamo ricordare che per memorizzare un intero tipicamente si utilizzano 4 bytes.

Consideriamo quindi tutti i bit che sono associati ad un intero.

Dato che un singolo byte può essere spezzato in 8 bit, in totale per un intero abbiamo a disposizione il seguente numero di bit

$$8 \times 4 = 32$$

Poniamoci ora la seguente domanda:

qual è il numero massimo che possiamo rappresentare con 32 bit?

L'idea è che con 32 bit posso rappresentare tutti i numeri da 0 a $2^{32} - 1$.

```
0000 0000 0000 0000 0000 0000 0000 0000  <---> 0
0000 0000 0000 0000 0000 0000 0000 0001  <---> 1
0000 0000 0000 0000 0000 0000 0000 0010  <---> 2
0000 0000 0000 0000 0000 0000 0000 0011  <---> 3
0000 0000 0000 0000 0000 0000 0000 0100  <---> 4
0000 0000 0000 0000 0000 0000 0000 0101  <---> 5
0000 0000 0000 0000 0000 0000 0000 0110  <---> 6
0000 0000 0000 0000 0000 0000 0000 0111  <---> 7
...
...
...
1111 1111 1111 1111 1111 1111 1111 1111  <---> 2^32 - 1 = 4294967295
```

Cosa succede però quando dobbiamo memorizzare, ad esempio, il numero 2^{32}?

Dato che non ci sono più bit a disposizione, il processore effettua un **overflow**, ovvero resetta il contenuto della memoria e ritorna al valore 0.

```
1111 1111 1111 1111 1111 1111 1111 1111  <---> 2^32 - 1

0000 0000 0000 0000 0000 0000 0000 0000  <---> 2^32
```

È come se la macchina dicesse che

$$2^{32} = 0$$

Notiamo che questa equazione non ha senso da un punto di vista matematico. Eppure la macchina potrebbe funzionare esattamente in questo modo.

Osservazione 1: Sono proprio questi gli aspetti che distinguono l'informatica dalla matematica, e che rendono l'informatica una materia molto pratica: **la finitezza del mondo fisico**.

È proprio per questo che il codice iniziale ha trasformato il valore 4294967296 nel valore 0, perché

$$2^{32} = 4294967296$$

Osservazione 2: Se l'architettura (o anche il compilatore) avesse associato più o meno bytes per memorizzare un intero, il numero dopo il quale la variabile ritorna a 0, causando un overflow, sarebbe diverso.

OVERFLOW #2: SHORT

Dato che uno **short** nella nostra architettura viene memorizzato tramite 2 bytes, per mandarlo in overflow basterà assegnarli il valore.

$$2^{(8\times 2)} = 2^{16} = 65536$$

Il seguente esempio mostra un **short overflow**.

```c
#include <stdio.h>

int main(int argc, char **argv) {

  short normal_value = 20;
  short overflowing_value = (short) (65536);

  printf("[INFO] - Normal short value       = %hd\n", normal_valu
  printf("[INFO] - Overflowing short value  = %hd\n", overflowing

  return 0;
}
```

Che una volta eseguita ci ritorna

```
[INFO] — Normal short value       = 20
[INFO] — Overflowing short value  = 0
```

OVERFLOW #3: LONG

Dato che uno **long** nella nostra architettura viene memorizzato tramite 8 bytes, per mandarlo in overflow basterà assegnarli il valore.

$$2^{(8\times 8)} = 2^{64} = 18446744073709551616$$

Il seguente esempio mostra un **long overflow**.

```c
#include <stdio.h>

int main(int argc, char **argv) {

  long normal_value = 4294967296;
  long overflowing_value = (long) (18446744073709551616);

  printf("[INFO] - Normal long value       = %ld\n", normal_value
  printf("[INFO] - Overflowing long value  = %ld\n", overflowing_

  return 0;
}
```

Che una volta eseguita ci ritorna

```
[INFO] – Normal long value       = 4294967296
[INFO] – Overflowing long value  = 0
```

E QUINDI?

In generale per mandare una variabile in overflow ci dobbiamo chiedere quanti bytes sono utilizzati per memorizzarla.

Se per memorizzare una variabile abbiamo bisogno di n bits, allora per mandare la variabile in overflow basterà farle raggiungere il valore di 2^n.

$$n \text{ bits} \longrightarrow 2^n \text{ per overflow}$$

COSA POTREBBE SUCCEDERE IN CASO DI OVERFLOW?

L'esempio più significativo di cosa potrebbe succedere in caso di integer overflow ci è offerto dal volo del razzo `Ariane 5`, accaduto nel 4 giugno del 1996.

Il razzo girava infatti del codice utilizzato per la versione precedente (`Ariane 4`) e un integer overflow ha causato il disastro.

https://www.youtube.com/watch?v=qnHn8W1Em6E

Preso da: Hackaday – the-7-billion-dollar-overflow

There were two bits of code. One that measured the sideways velocity, and one that used it in the guidance system. The measurement side used a 64 bit variable, but the guidance side used a 16 bit variable. The code was borrowed from an earlier, slower rocket whose velocity would never grow large enough to exceed that 16 bits. The Ariane 5, however, […] quickly overflowed this value.

Preso da: Hackaday – the-7-billion-dollar-overflow

The code that caused the overflow was actually a bit of pre-launch software that aligned the rocket. It was supposed to be turned off before the rocket firing, but since the rocket launch got delayed so often, the engineers made it timeout 40 seconds into the launch so they didn't have to keep restarting it.

EXTRA: PYTHON BIGNUM

Notiamo che in `python3` non ci sono apparenti limiti ai numeri che possiamo memorizzare in una variabile.

```
value = 184467440737095516161844674407370955161618446744073709551
```

Questo comportamento è conseguenza del fatto che l'interprete di `python3` utilizza un metodo chiamato `bignum` per gestire numeri con un numero arbitrario di cifre.

Ovviamente anche utilizzando `bignum` siamo comunque limitati dalla quantità di memoria fisica (RAM) presente nel computer e messa a disposizione dal sistema operativo.