Software Security 03

Programmazione in C

Gianluca Amato
Università di Chieti-Pescara

https://cybersecnatlab.it

License & Disclaimer

License Information

Disclaimer

- We disclaim any warranties or representations as to the accuracy or completeness of this material.
- Materials are provided “as is” without warranty of any kind, either express or implied, including without limitation, warranties of merchantability, fitness for a particular purpose, and non-infringement.
- Under no circumstances shall we be liable for any loss, damage, liability or expense incurred or suffered which is claimed to have resulted from use of this material.

Il linguaggio C

Il linguaggio C

- Il C è un linguaggio di programmazione nato negli anni ‘70
  - per la scrittura di software di sistema, fino ad allora sempre scritti in assembly
- Privilegia:
  - Efficienza del codice compilato
  - Compattezza del codice sorgente
  - Controllo totale della macchina
- A scapito di:
  - Facilità di utilizzo
    - sintassi poco amichevole
    - necessità di gestire in maniera manuale la memoria
    - libreria standard minimalista
  - Portabilità
    - molti aspetti non sono completamente definiti dal linguaggio

C e Java

- A livello elementare, si può pensare a C come Java senza le classi.
  - I tipi "primitivi" di C e Java sono simili: `int`, `short`, `char`, `float`, ...
  - La sintassi delle istruzioni (`while`, `for`, `if`, ...) è simile.
  - Le *funzioni* C corrispondono ai *metodi statici* di Java.
  - Si usano le parentesi graffe per delimitare i blocchi come in Java.
  - L'esecuzione parte dalla funzione `main`.

Esempio: Java vs C

Java

Somma.java

```java
class Somma {

    // calcola la somma dei numeri da "a" fino a "b"
    public static int somma(int a, int b) {
        int somma = 0;
        for (int i = a; i <= b; i++) {
            somma += i;
        }
        return somma;
    }

    public static void main(String[] args) {
        int res = somma(1, 10);
        System.out.println("Somma: " + res);
    }
}
```

C

somma.c

```c
// importa le funzioni standard di input/output (printf)
#include <stdio.h>

// calcola la somma dei numeri da "a" fino a "b"
int somma(int a, int b) {
    int somma = 0;
    for (int i = a; i <= b; i++) {
        somma += i;
    }
    return somma;
}

void main(int argc, char *argv[]) {
    int res = somma(1, 10);
    printf("Somma: %d\n", res);
}
```

Esempio: Python vs C

Python

somma.py

```python
# calcola la somma dei numeri da "a" fino a "b"
def somma(a, b):
    somma = 0
    for i in range(a, b+1):
        somma += i
    return somma

res = somma(1, 10)
print("Somma:", res)
```

C

somma.c

```c
// importa le funzioni standard di input/output (printf)
#include <stdio.h>

// calcola la somma dei numeri da "a" fino a "b"
int somma(int a, int b) {
    int somma = 0;
    for (int i = a; i <= b; i++) {
        somma += i;
    }
    return somma;
}

void main(int argc, char *argv[]) {
    int res = somma(1, 10);
    printf("Somma: %d\n", res);
}
```

Differenze tra C e Java (e Python)

- Direttive del preprocessore
- Input e output
- Array
- Stringhe
- Puntatori
- Gestione della memoria

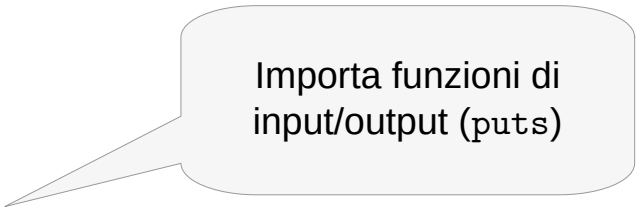
Le direttive

- Le direttive C sono delle speciali istruzioni che iniziano con #
- Non sono vere istruzioni C:
  - vengono prese in considerazione durante la fase di pre-elaborazione;
  - spariscono dal codice prima che venga effettivamente compilato.
- Vi ricordo che è possibile interrompere il compilatore alla fase di pre-elaborazione per vederne il risultato:
  - `gcc -E` *`nomefile.c`*

La direttiva `#include`

- `#include <file.h>`
  - Include il file indicato all'interno del file corrente.
  - In pratica: serve ad importare le funzioni di libreria, in maniera analoga

    all'istruzione `import` di Java e Python.
    - `stdio.h`: funzioni di input/output
    - `string.h`: funzioni per le stringhe
    - `stdlib.h`: funzioni per l'allocazione della memoria, conversioni di tipo, …




```c
#include <stdio.h>


void main() {
  puts("Hello world!");
}
```

La direttiva `#define`

- `#define` *NOME VALORE*
  - Definisce una **macro**.
  - Da ora in poi tutte le volte che nel programma compare *NOME*, esso verrà rimpiazzato da *VALORE*.
  - **Non è una variabile**, è solo una sostituzione sintattica.

Output – puts

- La funzione per l'output più semplice è `puts`:
  - `puts(s)`: manda in output la stringa `s`
- Esempio:

```
#include <stdio.h>

#define MESSAGE "Hello world!"

void main() {
  puts(MESSAGE);
}
```

Output – `printf` (1)

- La funzione per l'output più versatile e usata è `printf`
  - `printf(`*`format`*`, `*`param1`*`, `*`param2`*`, …)`
- La stringa *format* contiene la stringa da stampare mischiata con alcuni *specificatori di conversione*:
  - Sequenze di caratteri che iniziano con `%`
  - Determinano come interpretare i parametri *`param1`*, *`param2`*, …
  - Ad esempio, `%d` indica un parametro di tipo intero

Output – printf (2)

printf.c

```c
#include <stdio.h>

void main() {
    printf("%d - %x\n", 42, 42);
    printf("%d - %s\n", 42, "pippo");
    printf("%d - %x\n", 42, "pippo");
    printf("%d - %s\n", 42, 42);
}
```

```
42 - 2a
42 - pippo
42 - fc8700d
Segmentation fault (core dumped)
```

intero in esadecimale

stringa

lo specificatore di conversione dice che passo una stringa, ma in realtà passo un numero

Il manuale delle funzioni della libreria C

- Potete visualizzare la documentazione di una funzione C dalla shell di Linux con il comando `man`:
  - `man puts`
  - Per alcune funzioni, come `printf`, esiste una comando della shell con lo stesso nome. In tal caso usare il comando:
    - `man 3 printf`
    - il 3 chiarisce che siamo interessati alla sezione 3 del manuale
    - la sezione 3 è quella sulle funzioni della libreria C
- In alternativa: https://linux.die.net/man/

Gli array (1)

- Gli array in C sono simili a quelli Java ma:
  - In Java, se `a` è un array, `a.length` è la sua lunghezza
  - In C non esiste un modo per scoprire la lunghezza di un array
    - Non esattamente vero con l'introduzione dei "*variable length array*", ma noi non li utilizzeremo.
- In Python gli array non si usano:
  - Possiamo pensare agli array C come le liste Python
  - Come in Java, non possiamo conoscere la lunghezza di un array
  - Tutti gli elementi di un array devono essere dello stesso tipo

Gli array (2)

array.c

```c
#include <stdio.h>

int somma_array(size_t len, int a[]) {
    int somma = 0;
    for (size_t i = 0; i < len; i++) {
        somma += a[i];
    }
    return somma;
}

void main() {
    int mioarray[5] = {10, 20, 30, 40, 50};
    int s = somma_array(5, mioarray);
    printf("La somma dell'array è: %d\n", s);
}
```

`size_t` è il tipo da usare per indici e lunghezze di array

Funzione che calcola la somma degli elementi di un array `a`. Notare che devo passare separatamente come parametro anche la lunghezza dell'array.

Passaggio di parametri (1)

- In generale in C il passaggio dei parametri è per valore:
  - Se passo ad una funzione un valore di tipo `int`, in realtà passo una copia di quel valore.
  - Le modifiche alla copia non si riflettono all'originale.
- L'unica eccezione è se passo un array ad una funzione:
  - In questo caso, passo il suo indirizzo (passaggio per riferimento).
  - Come avviene in Python per tutti i tipi.
  - O come avviene in Java per tutti i tipi non primitivi.
- Modifiche all'array nella funzione si riflettono sul chiamante.

Passaggio di parametri (2)

array2.c

```c
#include <stdio.h>

void change(size_t len, int a[]) {
    for (size_t i = 0; i < len; i++) {
        a[i] = 0;
    }
    len = 999;
}

void main() {
    size_t l = 5;
    int mioarray[5] = {10, 20, 30, 40, 50};
    change(l, mioarray);
    printf("l: %zu v: %d\n", l, mioarray[0]);
}
```

`l` non viene modificato, ma `mioarray` sì

zu è il codice da usare per le variabili di tipo `size_t`

Stringhe

- In C non esiste un vero tipo per le stringhe.
- Le stringhe vengono implementate come array di byte che terminano con il byte zero.
  - La corrispondenza tra byte e caratteri stampabili dipende dal set di caratteri in uso nel sistema (ASCII, UTF-8, etc...)
  - Quando si manipolano le stringhe bisogna stare attenti alla dimensione massima.
  - Vedremo quanti problemi creerà questo fatto.

Input – `fgets` (1)

- La funzione più semplice (tra quelle sicure) per l'input è `fgets`.
  - `fgets(`*`str`*`, `*`len`*`, stdin)`
    - legge una riga, e mette il risultano nell'array `str`
    - la lunghezza massima della stringa è `len-1`
    - in caratteri in eccesso vengono ignorati
    - l'eventuale carattere di andata a capo finale diventa parte di `str`
    - `stdin` denota lo standard input (normalmente la tastiera): va specificato perché fgets può essere usata anche per leggere da un file.

Input – `fgets` (2)

fgets.c

```c
#include <stdio.h>

#define SIZE 30

void main() {
    char nome[SIZE];
    printf("Inserisci il nome: ");
    fgets(nome, SIZE, stdin);
    printf("Ciao %s\n", nome);
}
```

Inserisci il nome: **Gianluca**
Ciao Gianluca

Nelle slide, metto in grassetto quanto digitato dall'utente

Non si vede nella slide, ma c'è una riga vuota perché vengono stampati due caratteri di andata a capo: quello digitato dall'utente e che si trova in `nome`, e il `\n` di `printf`.

Input – `fgets` (3)

- Se l'array destinazione della stringa è troppo piccola… c'è una vulnerabilità.

fgets2.c

```
#include <stdio.h>

#define SIZE 30

void main() {
    char nome[10];
    printf("Inserisci il nome: ");
    fgets(nome, SIZE, stdin);
    printf("Ciao %s\n", nome);
}
```

dimensione dell'array `nome` ridotto a soli 10 caratteri

Inserisci il nome: **Questo nome è probabilmente troppo lungo**
Ciao Questo nome è probabilmente
Segmentation fault (core dumped)

Input e conversioni di tipo

- Se si deve leggere in input un numero, si può leggere una stringa e poi convertirla in numero.

fgets3.c

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

#define SIZE 30

void main() {
    char buffer[SIZE];

    printf("Immetti lato di un quadrato: ");
    fgets(buffer, 30, stdin);
    int lato = atoi(buffer);

    printf("Area: %d\n", lato * lato);
}
```

per la funzione `atoi`

`atoi` restituisce l'intero rappresentato nella stringa buffer

```
Immetti lato di un quadrato: 20
Area: 400
```

Esercitazione

Esercizio

Trovare la password per il programma `mychallenge2`

Puntatori e allocazione dati

Utilizzo della memoria (1)

- La maggior parte dei linguaggi di programmazione consentono al programmatore di utilizzare tipi di dati senza preoccuparsi di come sono rappresentati in memoria.
- Allo stesso modo, i programmatori ignorano dove i dati si trovano in memoria (in gergo più tecnico, si parla di allocazione dei dati)
  - I compilatori prendono queste decisioni di concerto col sistema operativo.

Utilizzo della memoria (2)

- In alcuni linguaggi, come il C, l'utilizzo della memoria è controllabile dal programmatore.
  - Si possono allocare e deallocare zone di memoria in maniera manuale.
  - Si può scoprire dove risiede in memoria un certa variabile.
  - Si può manipolare liberamente il contenuto della memoria.
- In C, se `x` è una variabile, `&x` è l'indirizzo nella memoria dove `x` è memorizzato
  - Il primo degli indirizzi, se `x` richiede più di un byte

Allocazione memoria (1)

allocazione1.c

```c
#include <stdio.h>

void main() {
    int i;
    char c;
    short s;
    long l;

    printf("i è allocata all'indirizzo %p ed occupa %zu byte\n", &i, sizeof(i));
    printf("c è allocata all'indirizzo %p ed occupa %zu byte\n", &c, sizeof(c));
    printf("s è allocata all'indirizzo %p ed occupa %zu byte\n", &s, sizeof(s));
    printf("l è allocata all'indirizzo %p ed occupa %zu byte\n", &l, sizeof(l));
}
```

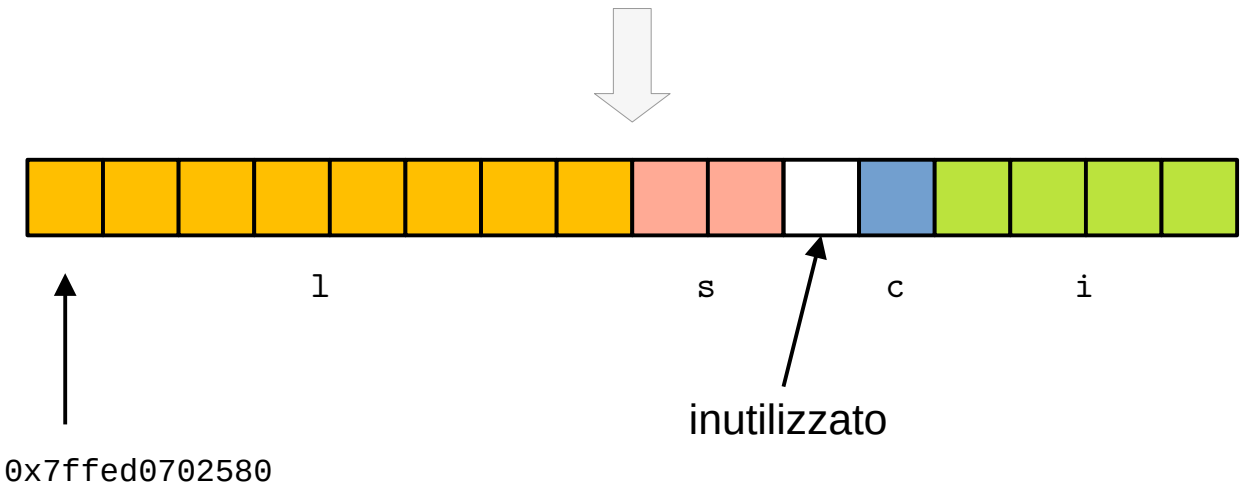
`%p` è lo specificatore per gli indirizzi di memoria

`sizeof` restituisce la quantità di byte occupati da una variabile o un tipo.

```
i è allocata all'indirizzo 0x7ffed070258c ed occupa 4 byte
c è allocata all'indirizzo 0x7ffed070258b ed occupa 1 byte
s è allocata all'indirizzo 0x7ffed0702588 ed occupa 2 byte
l è allocata all'indirizzo 0x7ffed0702580 ed occupa 8 byte
```

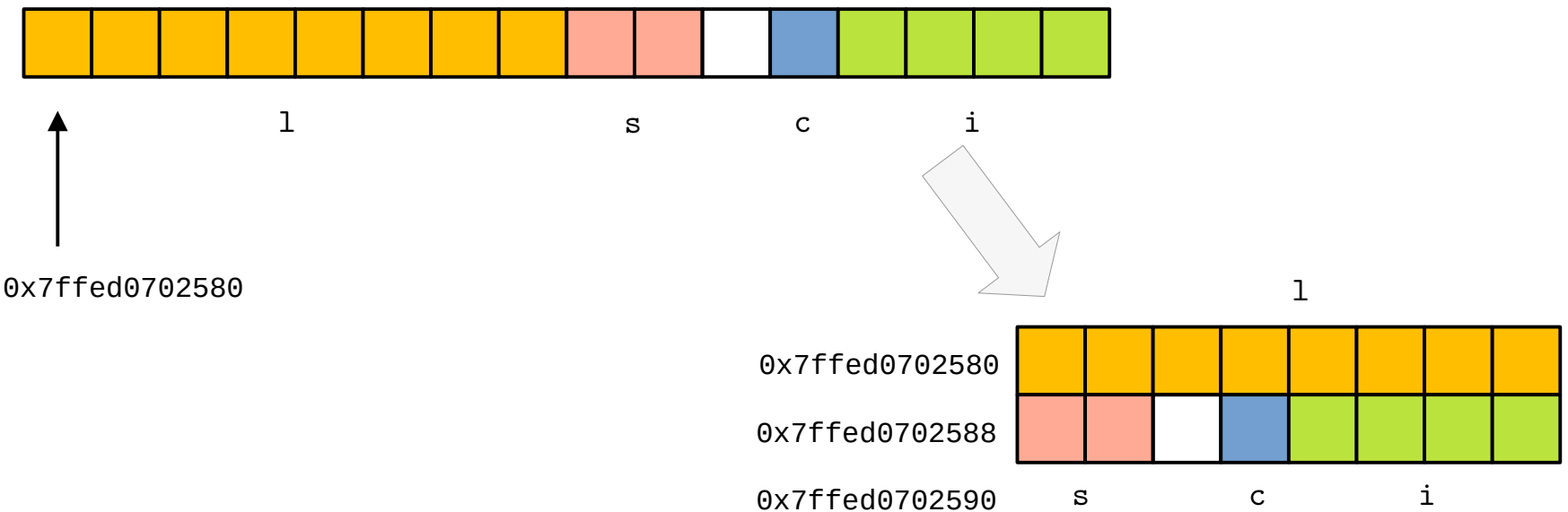
Allocazione memoria (2)


```
i è allocata all'indirizzo 0x7ffed070258c ed occupa 4 byte
c è allocata all'indirizzo 0x7ffed070258b ed occupa 1 byte
s è allocata all'indirizzo 0x7ffed0702588 ed occupa 2 byte
l è allocata all'indirizzo 0x7ffed0702580 ed occupa 8 byte
```



Allocazione memoria (3)

- Visto che stiamo lavorando con una CPU a 64 bit, può essere conveniente pensare la memoria divisa a gruppi di 8 byte (64 bit)



Allocazione memoria (4)

- Gli indirizzi possono cambiare ad ogni esecuzione:
  - Ma la disposizione relative tra di loro è costante

```
amato@banzai:~$ ./allocazione1
i è allocata all'indirizzo 0x7ffc9687648c ed occupa 4 byte
c è allocata all'indirizzo 0x7ffc9687648b ed occupa 1 byte
s è allocata all'indirizzo 0x7ffc96876488 ed occupa 2 byte
l è allocata all'indirizzo 0x7ffc96876480 ed occupa 8 byte
amato@banzai:~$ ./allocazione1
i è allocata all'indirizzo 0x7ffef392066c ed occupa 4 byte
c è allocata all'indirizzo 0x7ffef392066b ed occupa 1 byte
s è allocata all'indirizzo 0x7ffef3920668 ed occupa 2 byte
l è allocata all'indirizzo 0x7ffef3920660 ed occupa 8 byte
amato@banzai:~$ ./allocazione1
i è allocata all'indirizzo 0x7ffe5a9567dc ed occupa 4 byte
c è allocata all'indirizzo 0x7ffe5a9567db ed occupa 1 byte
s è allocata all'indirizzo 0x7ffe5a9567d8 ed occupa 2 byte
l è allocata all'indirizzo 0x7ffe5a9567d0 ed occupa 8 byte
```

Allocazione memoria (5)

- Tuttavia, alcune opzioni del compilatore possono anche cambiare la distanza tra gli indirizzi.
  - Ad esempio, con l'opzione `-O` del compilatore che serve ad ottimizzare il codice generato.
- Compilazione standard

```
i è allocata all'indirizzo 0x7ffed070258c ed occupa 4 byte
c è allocata all'indirizzo 0x7ffed070258b ed occupa 1 byte
s è allocata all'indirizzo 0x7ffed0702588 ed occupa 2 byte
l è allocata all'indirizzo 0x7ffed0702580 ed occupa 8 byte
```

- Compilazione con `-O2`

```
i è allocata all'indirizzo 0x7fffa11375d4 ed occupa 4 byte
c è allocata all'indirizzo 0x7fffa11375d1 ed occupa 1 byte
s è allocata all'indirizzo 0x7fffa11375d2 ed occupa 2 byte
l è allocata all'indirizzo 0x7fffa11375d8 ed occupa 8 byte
```

Allineamento dati

- Il compilatore può introdurre spazio non usato (padding)
  - In generale, per questioni di efficienze è normale che un dato lungo *n* byte inizi ad una locazione divisibile per *n*

allocazione2.c

```c
#include <stdio.h>

void main() {
    long l1;
    char c;
    long l2;

    printf("l1: %p\n", &l1);
    printf(" c: %p\n", &c);
    printf("l2: %p\n", &l2);
}
```

```
l1: 0x7ffe6bf43a08
 c: 0x7ffe6bf43a07
l2: 0x7ffe6bf439f8
```

7 byte inutilizzati tra `l2` e `c`

Puntatori (1)

- In C esiste il tipo "*puntatore a*", che si indica con un * prima del tipo
  - `int *`: puntatore a intero
  - `char *`: puntatore a carattere
  - `void *`: puntatore a un oggetto di tipo non specificato
- Un puntatore:
  - non contiene un valore
  - ma l'indirizzo della locazione di memoria dove il valore è memorizzato.

Puntatori (2)

- Se `i` è una variabile intera (tipo `int`)
  - `&i` (quello che fin'ora abbiamo chiamato semplicemente *indirizzo* di `i`) è effettivamente un puntatore ad `i`
- Se `p` è una variabile puntatore a intero (tipo `int *`)
  - `*p` è il valore puntato dal puntatore `p`.
- Gli operatori `*` e `&` sono uno l'inverso dell'altro:
  - `&(*p) == p`
  - `*(&i) == i`

Esempio: puntatori

puntatori.c

```c
#include <stdio.h>

void main() {
    int i = 42;
    int *p = &i;
    printf("%d\n", i);
    printf("%d\n", *p);
    printf("%p\n", p);

    *p += 5;
    printf("%d\n", i);

    p = 0;
    printf("%d\n", *p);
}
```

```
42
42
0x7ffdc842941c
47
Segmentation fault (core dumped)
```

leggo dalla locazione 0,
ma non gradisce...

Input – `scanf` (1)

- Un altro modo per leggere da tastiera è la funzione `scanf`:
  - `scanf(format, param1, param2, …)`
- La stringa *format* contiene la stringa che ci si aspetta in input, assieme agli *specificatori di conversione*:
  - Sequenze di caratteri che iniziano con %
  - Determinano che tipo di valore ci si aspetta dall'utente
    - Ad esempio, `%d` indica un valore intero da memorizzare su 4 byte
  - Le variabili `param1`, `param2`, etc… contengono i puntatori alla zona di memoria dove memorizzare i valori letti da tastiera

Input – scanf (2)

scanf.c

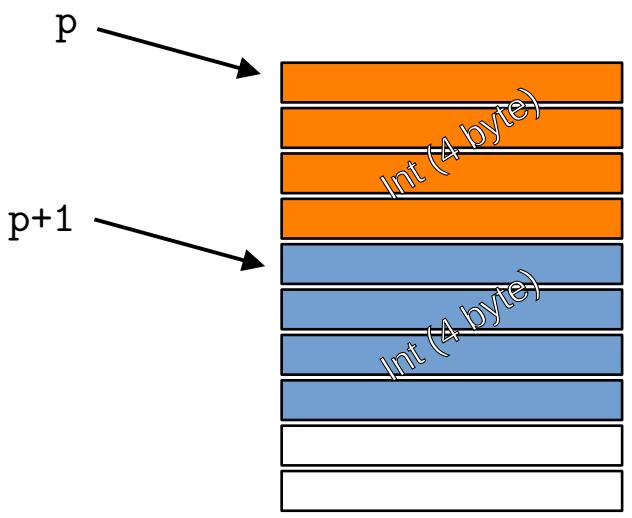
```c
#include <stdio.h>

void main() {
    int base, altezza;

    printf("Immetti base e altezza di un rettangolo: ");
    scanf("%d %d", &base, &altezza);
    printf("Area: %d\n", base * altezza);
}
```

Ci si aspetta due numeri interi separati da uno spazio

```
Immetti base e altezza di un rettangolo: 23 4
Area: 92
```

Aritmetica dei puntatori (1)

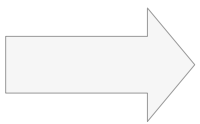
- Somma di una numero ad un puntatore (`p`)
  - `p+1` non è la locazione di memoria immediatamente successiva a `p`
  - La locazione puntata da `p+1` dipende dal tipo di `p`
    - Un variabile `int` occupa 4 byte
    - Se `p` è di tipo `*int`, `p+1` non punta alla locazione successiva a `p`, ma all'intero successivo, senza sovrapposizioni.



Aritmetica dei puntatori (2)

puntatori2.c

```c
#include <stdio.h>

void main() {
    int i = 42;
    int *p1 = &i;
    int *p2 = p1 + 1;
    void *p3 = p1;
    void *p4 = p3 + 1;

    printf("dimensione int: %lu\n", sizeof(int));

    printf("%p\n", p1);
    printf("%p\n", p2);
    printf("%p\n", p3);
    printf("%p\n", p4);

    printf("%d\n", *p1);
    printf("%d\n", *p2);
}
```



```
dimensione int: 4
0x7fff111fbcdc
0x7fff111fbce0
0x7fff111fbcdc
0x7fff111fbcdd
42
287292637
```

Dimensioni dei tipi in C

- Le dimensioni dei tipi dipendono dalla CPU e dal sistema operativo.
- Nei sistemi Linux a 64 bit:
  - `int`: 32 bit (4 byte)
  - `long`: 64 bit (8 byte)
  - `void` / `char`: 8 bit (1 byte)
  - puntatore: 64 bit (8 byte)
- Nei sistemi Linux a 32 bit, come per i sistemi a 64 bit, ma:
  - puntatore: 32 bit (4 byte)
- In generale `sizeof(tipo)` in C è la lunghezza in byte del tipo specificato

Array e puntatori (1)

- L'aritmetica dei puntatori viene usata spesso per accedere agli elementi di un array.

puntatori3.c

```c
#include <stdio.h>

void main() {
    int a[] = { 10, 33, 87, -4 };
    int *p = &a[0];

    printf("Elemento 0: %d\n", *p);
    printf("Elemento 1: %d\n", *(p+1));
    *(p+1)= 15;
    printf("Di nuovo elemento 1: %d\n", a[1]);
}
```

```
Elemento 0: 10
Elemento 1: 33
Di nuovo elemento 1: 15
```

Array e puntatori (2)

- Array e puntatori in C sono praticamente la stessa cosa.
- Se `p` è un puntatore:
  - `*p` si può scrivere come `p[0]`
  - `*(p+1)` si può scrivere come `p[1]`
  - In generale, `*(p+n)` si può scrivere come `p[n]`
- Al contrario, se `a` è un array
  - `a[0]` si può scrivere come `*a`
  - `a[n]` si può scrivere come `*(a+n)`
- L'unica vera differenza è che una variabile array non si può modificare:
  - `a+=1` genera errore
  - `p+=1` funziona

Scorrere un array con i puntatori

Sintassi basata su array

```c
#include <stdio.h>

int somma_array(size_t len, int a[]) {
    int somma = 0;
    for (size_t i = 0; i < len; i++) {
        somma += a[i];
    }
    return somma;
}

void main() {
    int mioarray[5] = {10, 20, 30, 40, 50};
    int s = somma_array(5, mioarray);
    printf("La somma dell'array è: %d\n", s);
}
```

Sintassi basata su puntatori

puntatori4.c

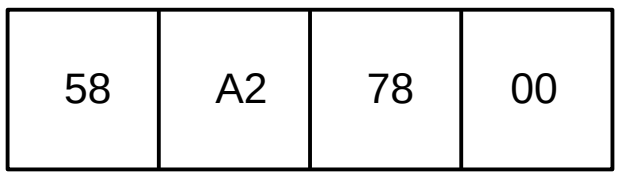
```c
#include <stdio.h>

int somma_array(int *start, int* end) {
    int somma = 0;
    for (int *p = start; p < end; p++) {
        somma += *p;
    }
    return somma;
}

void main() {
    int mioarray[5] = {10, 20, 30, 40, 50};
    int s = somma_array(mioarray, mioarray+5);
    printf("La somma dell'array è: %d\n", s);
}
```

Big endian e little endian (1)

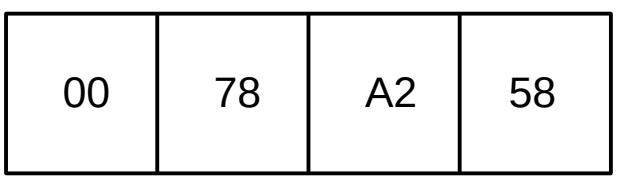
- Un valore di tipo `int` occupa 4 byte: ma come sono disposti in memoria ?
- Little endian:
  - Standard sulle architetture x86 e x86-64
  - Il byte meno significativo viene salvato negli indirizzi più bassi
- Big endian:
  - Il byte più significativo viene salvato negli indirizzi più bassi
- Questi nomi traggono origine dal romanzo “I viaggi di Gulliver” !

Big endian e little endian (2)

- Esempio: 0x0078A258
  - Little endian:



  - Big endian:



Big endian e little endian (3)

endianess.c

```c
#include <stdio.h>

void main() {
    int v = 0;
    char *p = (char*) &v;
    p[1] = 'A';
    printf("%d\n", v);
}
```

type casting: convince il compilatore a trattare un puntatore a intero come fosse un puntatore a char

```
16640
```

???

Accesso fuori dai margini (1)

- In Java e Python, un tentativo di accedere ad un elemento inesistente di un array o di una lista causa un errore.
- In C non si genera nessun errore!
  - Si accede semplicemente a zone della memoria situate al di fuori dall'array.
  - Qusto fenomeno è noto come buffer overflow, ed è probabilmente la vulnerabilità più comune nei programmi in C.

Accesso fuori dai margini (2)

oob.c

```c
#include <stdio.h>

void main() {
    int x = 0;
    char s[] = "ciao";
    s[5] = 10;
    printf("%d\n", x);
}
```

```
10
```

Esercitazione

Esercizio

Trovare l'input che fa rispondere "Ce l'hai fatta" al programma `mychallenge3`

Esercizi

Svolgere la challenge (fattibile)

SS_2.01 Digital billboard

(è una versione più difficile di `mychallenge3`*)*

Esercizi

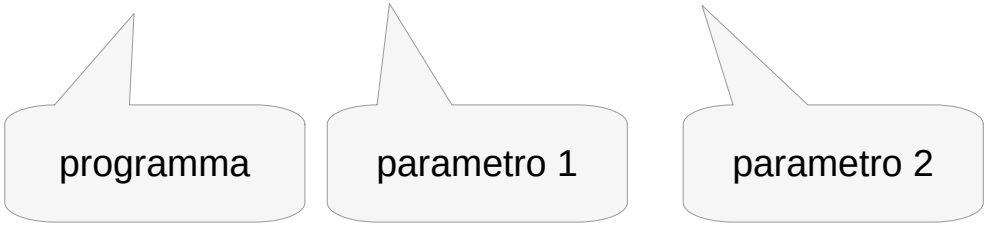
Svolgere la challenge (media)

SS_1.04 Unbreakable AES

(è una versione più difficile di `mychallenge2`*)*

Parametri su riga di comando (1)

- Quando si lancia un programma, si possono passare parametri sulla riga di comando
  - Esempio: `apt install default-jdk`



- I parametri `argc` e `argv` della funzione `main` contengono il valore dei parametri su riga di comando.
  - In maniera analoga al parametro `args` del metodo `main` in Java.

Parametri su riga di comando (2)

- `argc` contiene il numero di parametri
- `argv` è un array di puntatori a caratteri
  - Ovvero, in C, un array di stringhe
  - Ogni stringa è un parametro della riga di comando
    - `argv[0]` è il nome del programma
    - `argv[1]` è il primo parametro
    - e così via

Parametri su riga di comando (3)

args.c

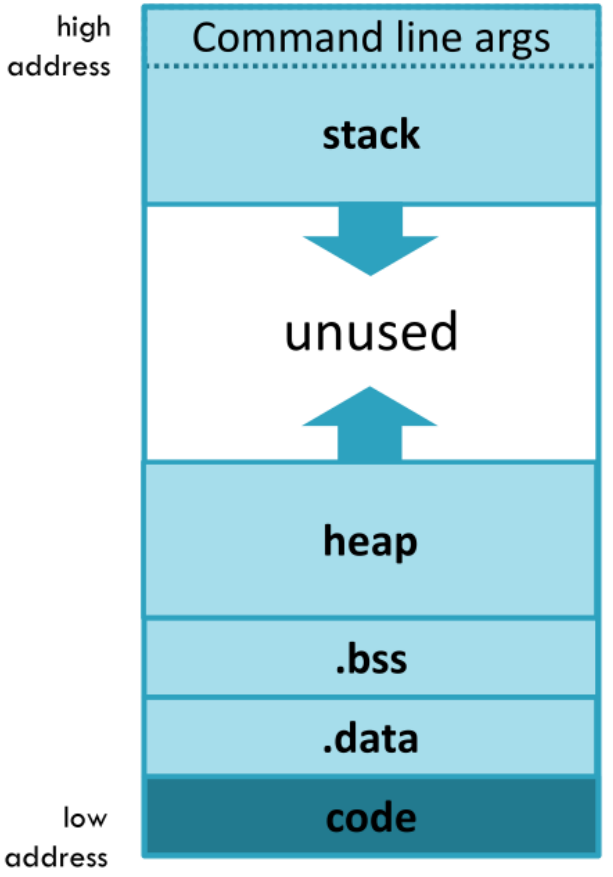
```c
#include <stdio.h>

void main(int argc, char *argv[]) {
    for (int i = 0; i < argc; i++) {
        printf("argv[%d]: %s\n", i, argv[i]);
    }
}
```

```
amato@atomino:~$ ./args Ciao parametro
argv[0]: ./args
argv[1]: Ciao
argv[2]: parametro
```

Gestione della memoria

Segmenti di memoria (1)

- Ogni processo ha della memoria allocata per memorizzare dati e codice.
- Questa memoria è divisa in aree specifiche:
  - stack: per le variabili locali
  - heap: per la memoria allocata dinamicamente
  - `bss`: variabili globali non inizializzate
  - `data`: variabili globali con valori iniziali
  - `rodata`: dati inizializzati in sola lettura (stringhe)
  - area codice: per le istruzioni del programma



Esercitazione

- Usando `readelf`:
  - disegnare una mappa di memoria dettagliata di questo programma;
  - verificare la sezione in cui si trovano memorizzate le variabili del programma.

memarea.c

```c
#include <stdio.h>

int i;
char *p = "Ciao mondo sola lettura";
char s[] = "Ciao mondo";
int a[100];

void main() {
    int x;

    printf(" i: %p\n", &i);
    printf(" p: %p\n", &p);
    printf("*p: %p\n", p);
    printf(" s: %p\n", &s);
    printf(" a: %p\n", &a);
    printf(" x: %p\n", &x);
}
```

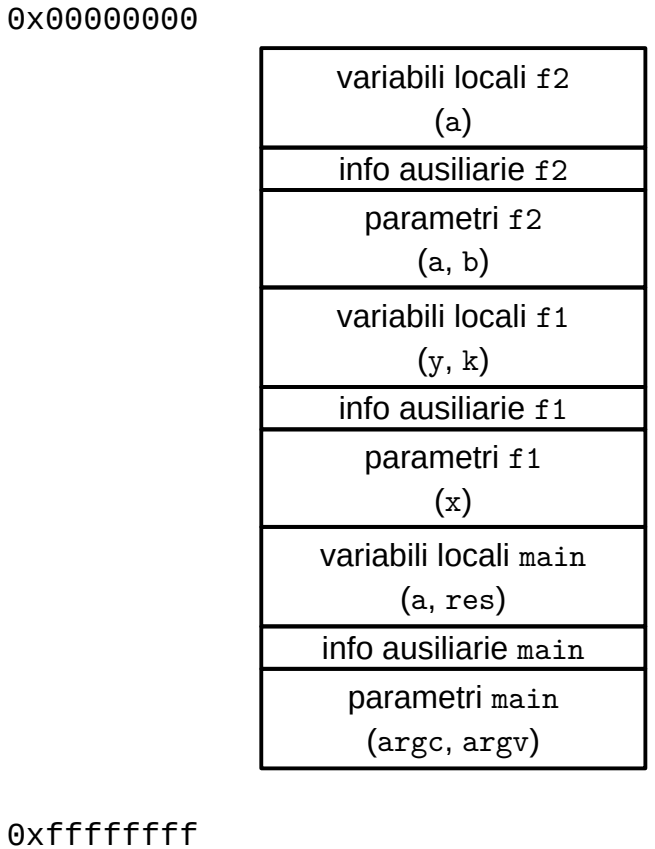
I record di attivazione (1)

- Ogni volta che una funzione viene chiamata, viene creato un "record di attivazione" nello stack
  - Ogni record di attivazione contiene le variabili locali alla funzione, più altre informazioni ausiliarie
  - A differenza di Python o Java, il record di attivazione contiene proprio i valori delle variabili, non un riferimento allo heap.
- I record di attivazione si trovano in una area di memoria chiamata stack che cresce verso indirizzi di memoria più bassi.

I record di attivazione (2)

stack.c

```c
#include <stdio.h>

int f2(int a, int b) {
    int z = a + b;
    return z;
}

int f1(int x) {
    int y = 2 * x;
    int k = f2(x, y);
    return k;
}

void main() {
    int a = 2;
    int res = f1(a);
    printf("%d\n", res);
}
```



Funzioni che restituiscono array ? (1)

- Attenzione
  - una funzione non deve restituire puntatori a dati presenti nello stack
  - quando la funzione termina questi dati potrebbero non esistere più!

stack2.c

```c
#include <stdio.h>

int *unit_point() {
    int point[2] = { 1, 1 };
    return point;
}

void main() {
    int *point = unit_point();
    printf("%d %d\n", point[0], point[1]);
}
```

l'array `point` è memorizzato nello stack

Potrebbe funzionare o generare "*segmentation fault*"

Funzioni che restituiscono array ? (2)

- Per questo, di solito in C le funzioni non restituiscono array
  - Prendono invece come parametro l'array su cui devono operare

stack3.c

```c
#include <stdio.h>

void unit_point(int point[]) {
    point[0] = 1;
    point[1] = 1;
}

void main() {
    int point[2];
    unit_point(point);
    printf("%d %d\n", point[0], point[1]);
}
```

Allocazione dinamica (1)

- Ma perché in Java e Python non c'è problema a restituire un array (o simile)?

```c
#include <stdio.h>

int *unit_point() {
    int point[2] = { 1, 1 };
    return point;
}

void main() {
    int *point = unit_point();
    printf("%d %d\n", point[0], point[1]);
}
```

```python
def unit_point():
    point = [1, 1]
    return point

point = unit_point()
print(point[0], point[1])
```

Allocazione dinamica (2)

- Perché in Java e Python:
  - array e simili non sono creati nello stack ma nell'heap
  - non subiscono la sorte del record di attivazione
  - in Java l'operatore `new` indica che stiamo allocando qualcosa nell'heap
- Si può allocare memoria nell'heap in C ?
  - Sì, con la funzione `malloc` (e similari)
  - `void *malloc(size_t size)`
    - Riserva una quantità di memoria pari a `size` byte, e restituisce il puntatore a questa zona di memoria

Allocazione dinamica (3)

malloc.c

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

int *unit_point() {
    int *point = malloc(2 * sizeof(int));
    point[0] = 1;
    point[1] = 1;
    return point;
}

void main() {
    int *point = unit_point();
    printf("%d %d\n", point[0], point[1]);
}
```

importa le funzioni di allocazione della memoria

alloca memoria sufficiente per 2 interi

la memoria allocata da `malloc` è sempre valida

Liberare la memoria allocata (1)

- Ma chi libera la memoria allocata per un dato che non serve più ?
  - In Java e Python ci pensa una componente dell'interprete chiamato "garbage collector"
- In C il garbage collector non esiste:
  - Tutta la memoria allocata con malloc va liberata con la funzione free
  - Altrimenti rimane in vita fino al termine del programma

Liberare la memoria allocata (2)

free.c

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

#define SIZE 2000000

void main() {
{
    int count = 0;
    while (1) {
        count += 1;
        printf("count: %d\n", count);
        int *p = malloc(SIZE * sizeof(int));
        for (size_t i = 0; i < SIZE; i++) {
            p[i] = 0;
        }
    }
}
```

prima o poi viene ucciso dal sistema operativo

free.py

```python
SIZE = 2000000
count = 0

while True:
    count += 1
    print("count:",count)
    l = [0] * SIZE
```

continua l'esecuzione per sempre

Liberare la memoria allocata (3)

free2.c

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

#define SIZE 2000000

void main() {
{
    int count = 0;
    while (1) {
        count += 1;
        printf("count: %d\n", count);
        int *p = malloc(SIZE * sizeof(int));
        for (size_t i = 0; i < SIZE; i++) {
            p[i] = 0;
        }
        free(p);
    }
}
```

aggiungere `free(p)` per risolvere il problema

free.py

```python
SIZE = 2000000
count = 0

while True:
    count += 1
    print("count:",count)
    l = [0] * SIZE
```

continua l'esecuzione per sempre

Liberare la memoria allocata (4)

- La cosa però non è sempre semplice:
  - Quando deallocare la memoria ?
  - Se si dealloca quando ancora la si sta usando, è un problema
  - Se si dealloca due volte, è un problema

free3.c

```
#include <stdlib.h>

void main() {
    int *p = malloc(sizeof(int));
    free(p);
    free(p);
}
```

```
free(): double free detected in tcache 2
Aborted (core dumped)
```

Esercizi

Svolgere la challenge (difficile)

SS_1.03 Flag Checker

Esercizi

Svolgere la challenge (difficile)

SS_1.06 pacman

Esercizi

Svolgere la challenge (molto difficile)

SS_1.05 morph

Software Security 03

Programmazione in C

FINE

Gianluca Amato
Università di Chieti-Pescara

https://cybersecnatlab.it