

Processi

Unità 1 – Lezione 1 Processi sequenziali e paralleli

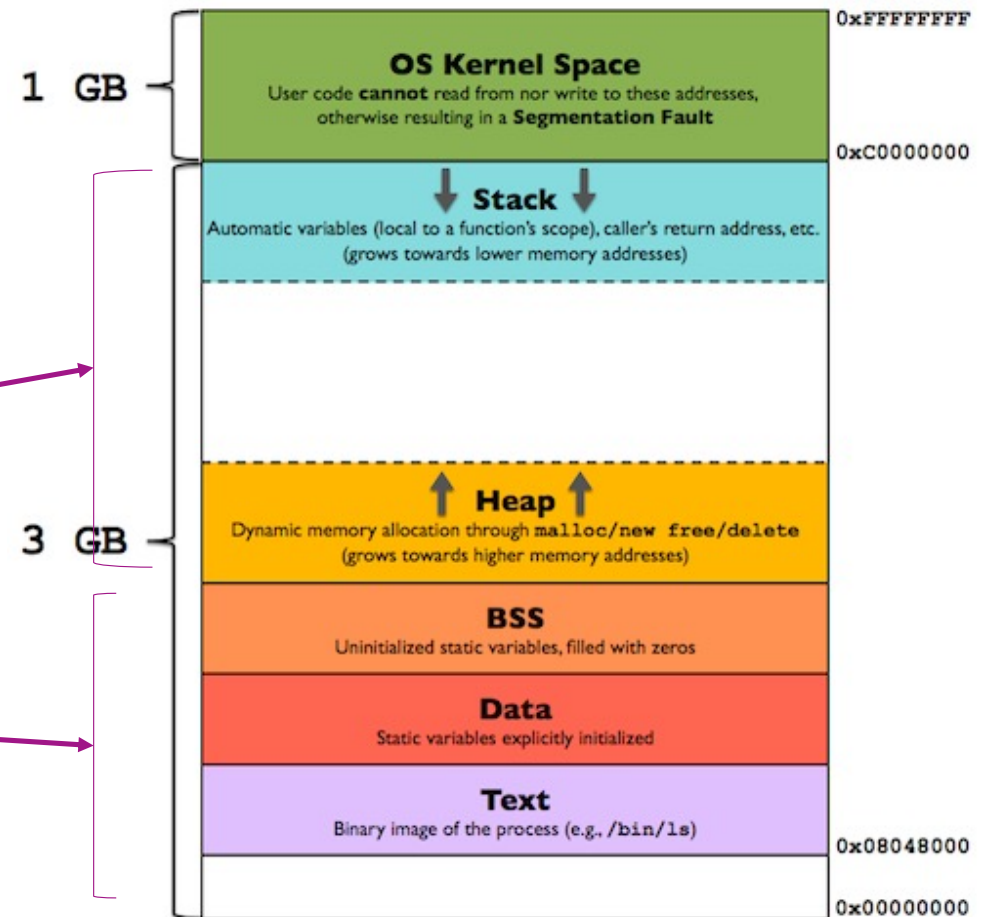
Memory Layout – la struttura della memoria di un processo

Processi

- Programma entità statica
 - una serie di istruzioni memorizzate su un dispositivo di archiviazione come un disco rigido o un SSD.
- Processo entità dinamica
 - un'istanza di un programma in esecuzione
 - il sistema operativo alloca risorse (come memoria e tempo di processore) per eseguire quel programma
 - quando viene creato ad un processo viene assegnato un PID: Process Identifier e una struttura dati chiamata Process Control Block PCB che il sistema operativo usa per mantenere e gestire le informazioni relative a un processo in esecuzione
 - ha uno stato che può cambiare nel tempo (come in esecuzione, in attesa, terminato, ecc.) e occupa **spazio nella memoria RAM** del computer mentre è in esecuzione.

Memory Layout di un processo ...

- ...ovvero com'è organizzata la memoria assegnata dal Sistema Operativo al processo in esecuzione
- Una parte dinamica
 - Stack
 - Heap
- Una parte statica
 - Area codice
 - Area dati globali



Memory Layout di un processo – parte statica

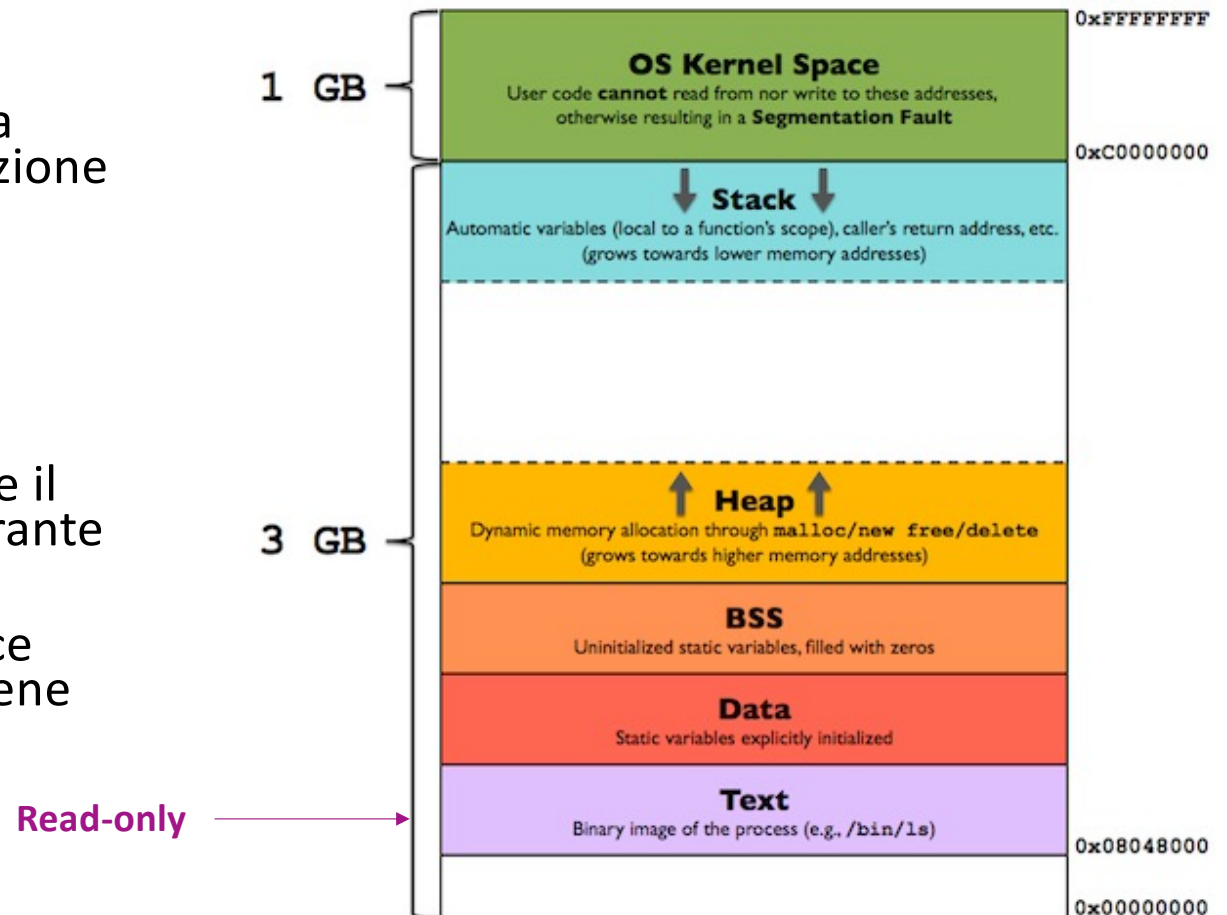
- ...ovvero com'è organizzata la memoria assegnata dal Sistema Operativo al processo in esecuzione

Area Text

Questa parte contiene il codice eseguibile del programma.

È di sola lettura per prevenire che il programma si auto-modifichi durante l'esecuzione.

Questo segmento include il codice compilato del programma che viene eseguito dal processore.



Memory Layout di un processo – parte statica

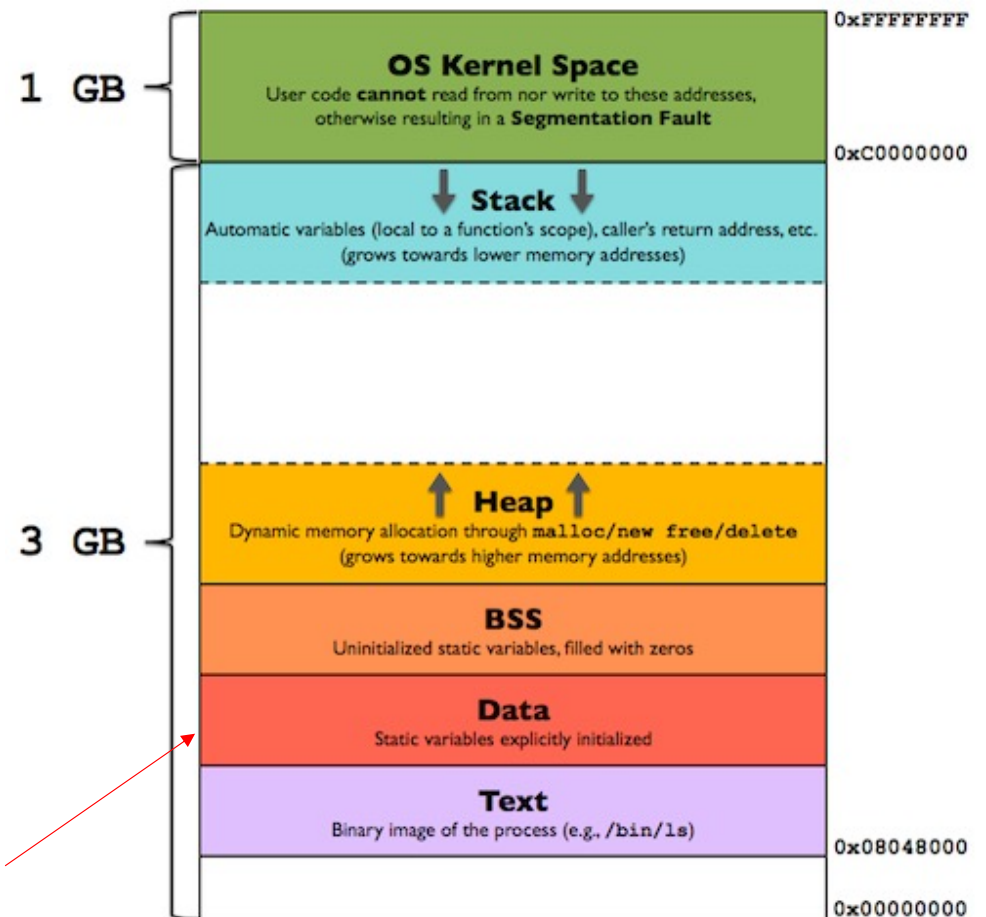
- ...ovvero com'è organizzata la memoria assegnata dal Sistema Operativo al processo in esecuzione

Data Segment

Le **variabili globali** (cioè quelle definite al di fuori di tutte le funzioni che compongono il programma) e le **variabili statiche** (cioè quelle dichiarate con l'attributo static).

Questo segmento è preallocata all'avvio del programma e inizializzato ai valori specificati nelle dichiarazioni. Il segmento può essere ulteriormente suddiviso **nell'area dei dati globali** **inizializzati** e in sola lettura (**read-only**), e **nell'area dei dati globali** **inizializzati modificabili** (**read-write**).

Read-only & Read-write



Esempio in C

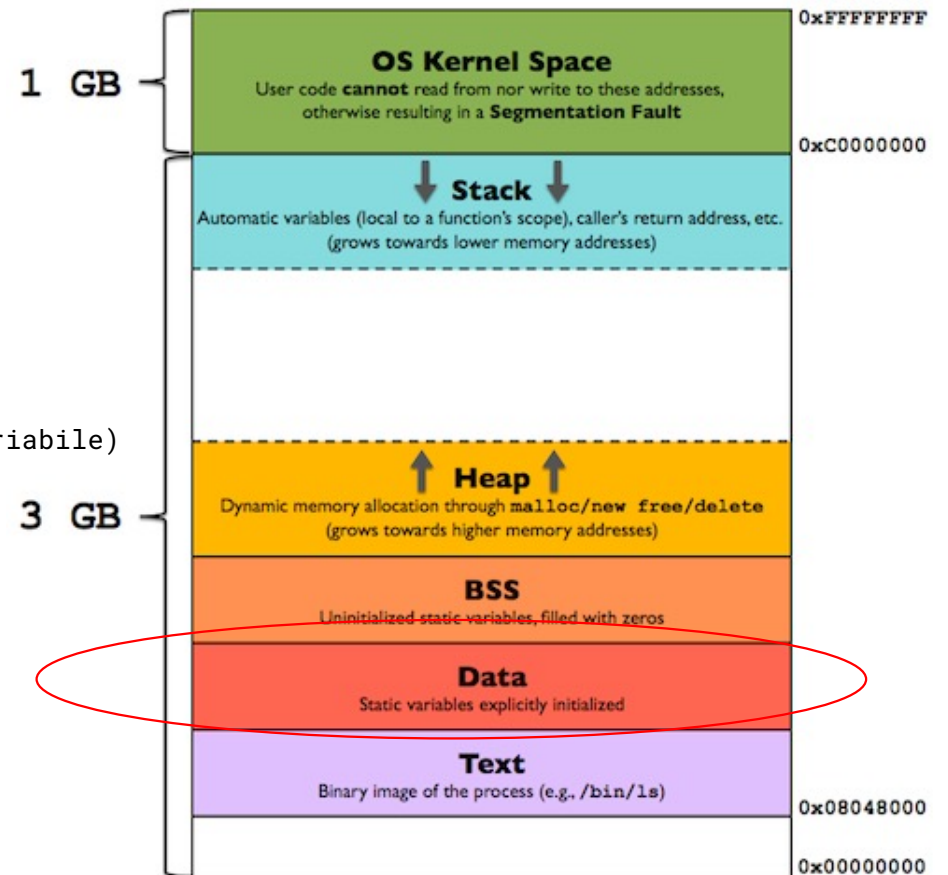
```
#include <stdio.h>
#include <stdlib.h>

// Variabile globale read-write Data Segment
int variabile_globale = 10;
char s[] = "hello world";

// Variabile statica read-write Data Segment
static int variabile_statica = 20;

//String literal "hello world" read-only
//puntatore str read-write (posso modificare il contenuto della variabile)
char* str = "hello world";
str = "ciao mondo";

int main(int argc, char *argv[]){
    ...
}
```



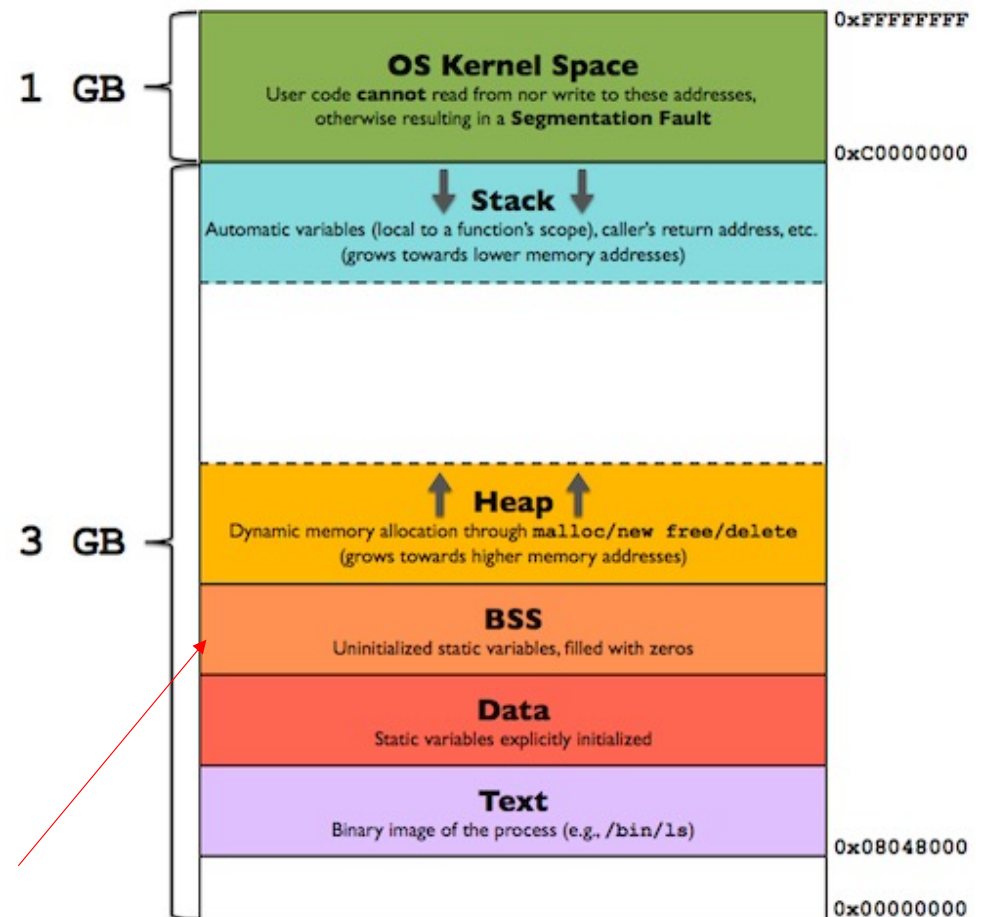
Memory Layout di un processo – parte statica

- ...ovvero com'è organizzata la memoria assegnata dal Sistema Operativo al processo in esecuzione

BSS

Il segmento BSS contiene le **variabili globali o statiche non esplicitamente inizializzate** dal programmatore, e il suo nome deriva da un vecchio operatore del linguaggio assembly nominato Block Started by Symbol. Di solito questo segmento inizia alla fine del segmento Data e i dati in questo segmento vengono inizializzati dal kernel al valore 0 (o un valore di default, null per puntatori) prima che il programma venga eseguito, e sono sempre dati modificabili (**read-write**).

Read-write



Esempio in C

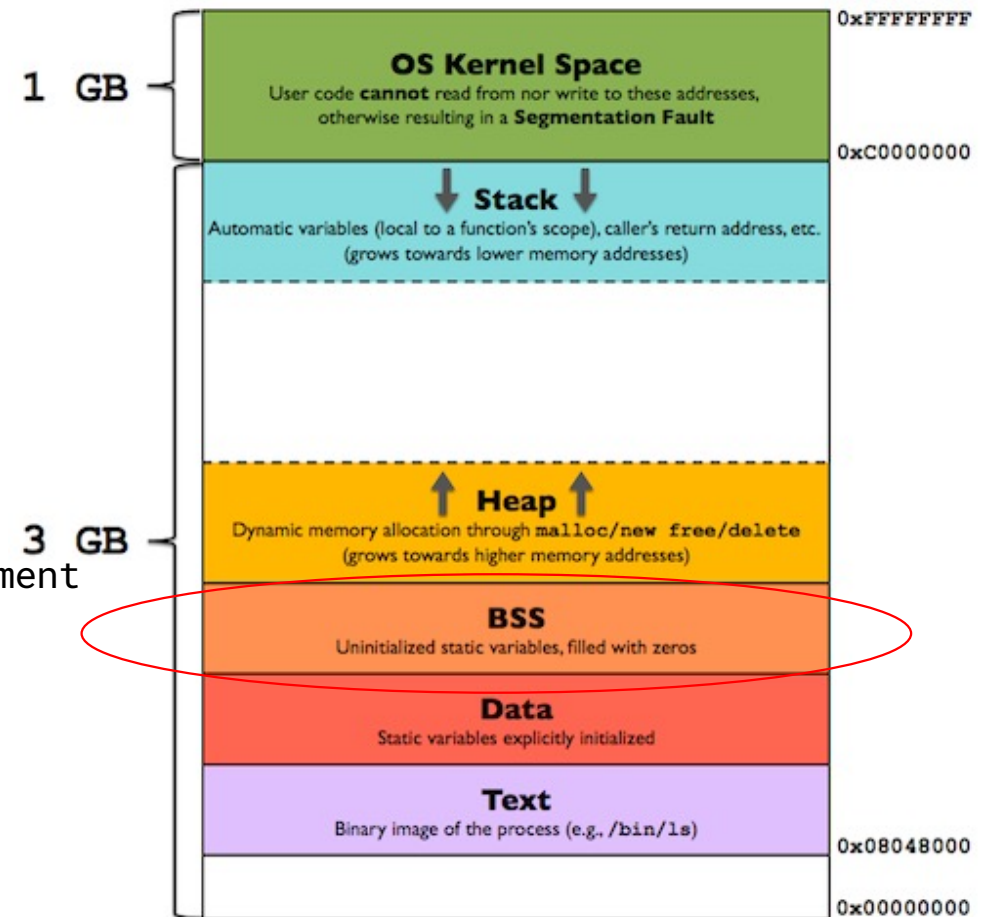
```
#include <stdio.h>
#include <stdlib.h>

// Variabile globale non inizializzata
// read-write BSS
int i;

// Variabile statica read-write BSS
// non inizializzata
static int variabile_statica;

//dipende da compilatore, a volte in Data Segment
int vect[100];

int main(int argc, char *argv[]){
    ...
}
```



Linux size command

Size (without the -m option) prints the (decimal) number of bytes required by the __TEXT, __DATA and __OBJC segments. All other segments are totaled and that size is listed in the 'others' column.

The final two columns is the sum in decimal and hexadecimal. If no file is specified, a.out is used.

memoryLayout.c

```
int variabile_globale = 10;
```

```
int main(){  
    return 0;  
}
```

gcc memory-layout.c -o memory-layout
size -m memoryLayout

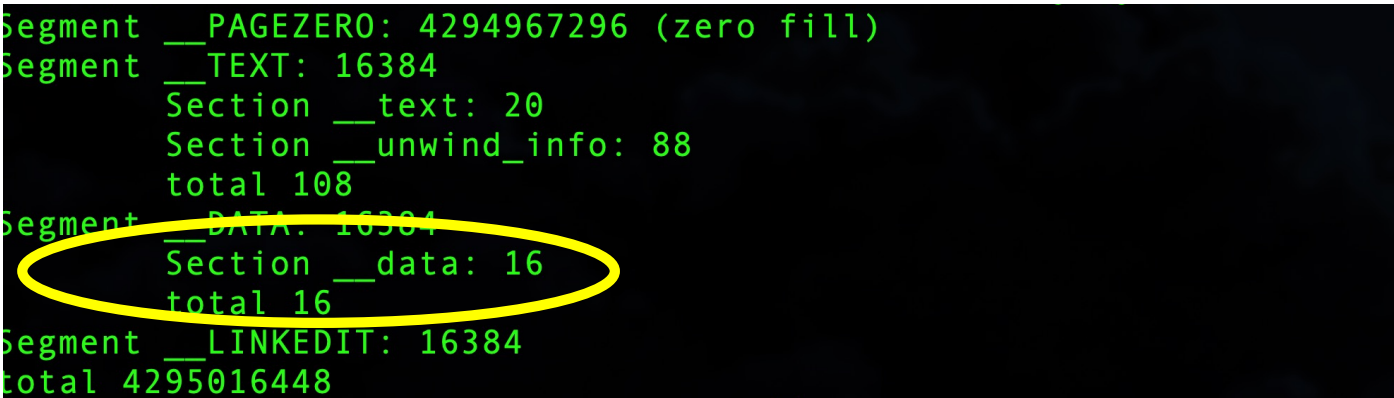
```
Segment __PAGEZERO: 4294967296 (zero fill)  
Segment __TEXT: 16384  
    Section __text: 20  
    Section __unwind_info: 88  
    total 108  
Segment __DATA: 16384  
    Section __data: 4  
    total 4  
Segment __LINKEDIT: 16384
```

memoryLayout.c

```
int variabile_globale = 10; //4 byte
char s[] = "hello world"; // 1 byte * 11 caratteri + terminazione null \0 inclusa in automatico

int main(){
    return 0;
}
```

gcc memory-layout.c -o memory-layout
size -m memoryLayout



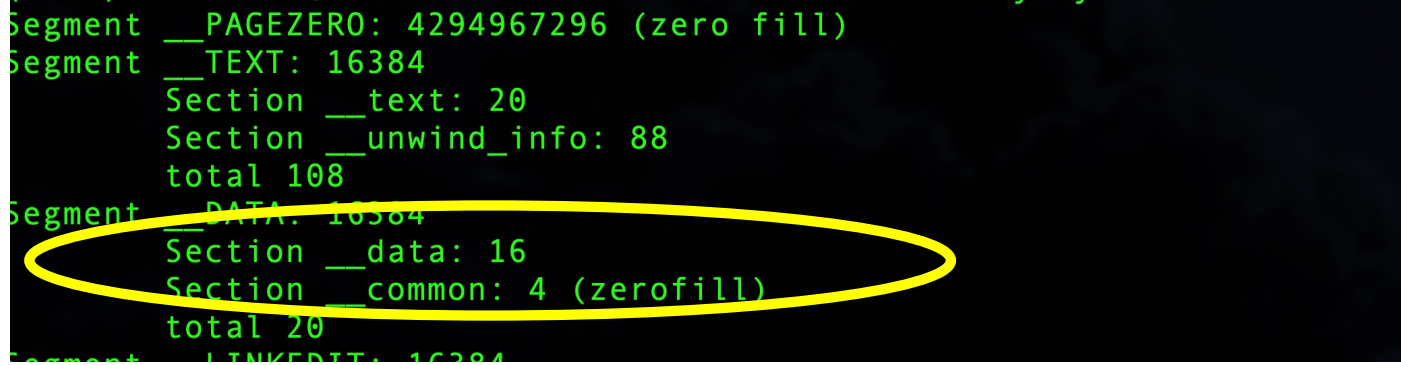
```
Segment __PAGEZERO: 4294967296 (zero fill)
Segment __TEXT: 16384
    Section __text: 20
    Section __unwind_info: 88
    total 108
Segment __DATA: 16384
    Section __data: 16
    total 16
Segment __LINKEDIT: 16384
total 4295016448
```

memoryLayout.c

```
int variabile_globale = 10; //4 byte
char s[] = "hello world"; // 1 byte * 11 caratteri + terminazione null \0 inclusa in automatico
int i;

int main(){
    return 0;
}
```

gcc memory-layout.c -o memory-layout
size -m memoryLayout

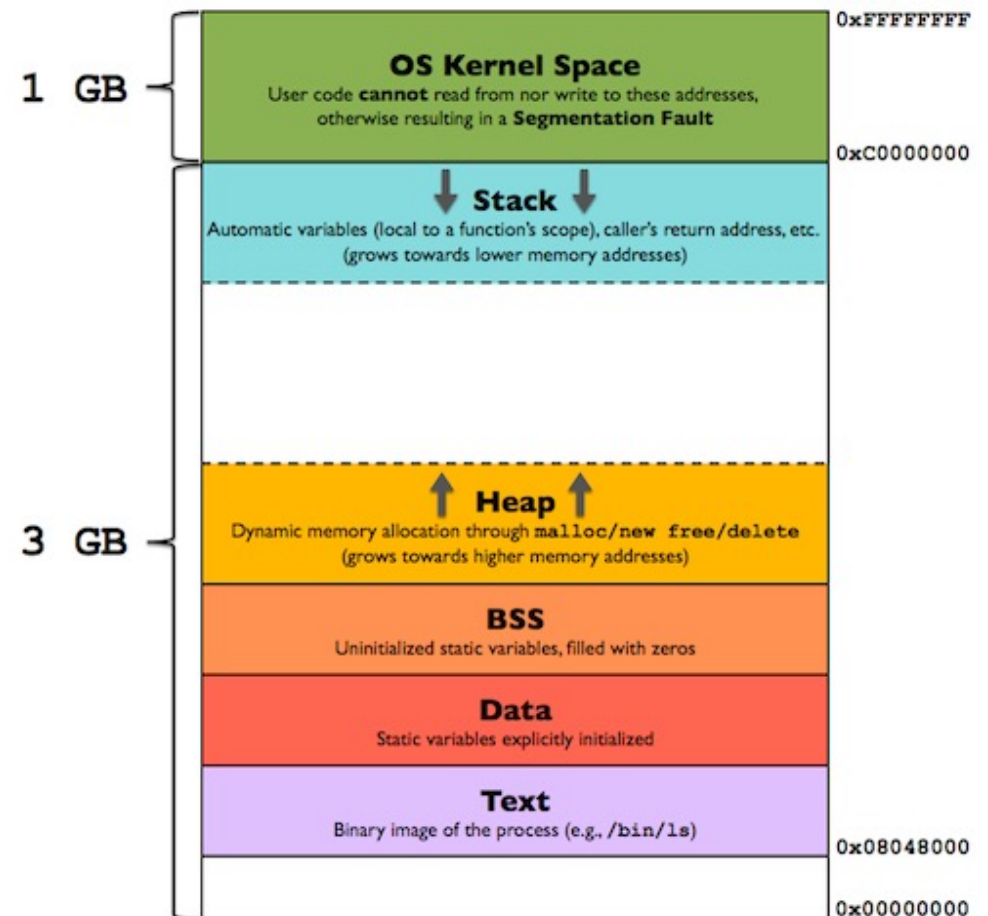


```
Segment __PAGEZERO: 4294967296 (zero fill)
Segment __TEXT: 16384
    Section __text: 20
    Section __unwind_info: 88
    total 108
Segment __DATA: 16384
    Section __data: 16
    Section __common: 4 (zerofill)
    total 20
Segment __LINKEDIT: 16384
```

The screenshot shows the memory layout of the compiled program. A yellow oval highlights the details of the `__DATA` segment, which includes the `__data` section (16 bytes) and the `__common` section (4 bytes, zero-filled).

Memory Layout di un processo – parte dinamica

- ...ovvero com'è organizzata la memoria assegnata dal Sistema Operativo al processo in esecuzione
- Lo stack e l'heap **non sono parte dell'eseguibile stesso**, ma aree di memoria che vengono allocate e gestite dal sistema operativo al momento del caricamento del processo in memoria.



Stack & Heap

- Quando un programma viene eseguito, il sistema operativo crea un processo per quel programma e alloca lo spazio per lo stack e l'heap come parte dello spazio di indirizzamento del processo. Queste aree di memoria sono dinamiche e la loro dimensione può cambiare durante l'esecuzione del programma (fino a raggiungere il Massimo).
- **Per esempio -> Impostazioni del Compilatore:** Alcuni compilatori permettono di specificare la dimensione dello stack durante la compilazione del programma. Ad esempio, in GCC (GNU Compiler Collection), puoi usare l'opzione `-Wl,--stack,size` per impostare la dimensione dello stack su Windows.

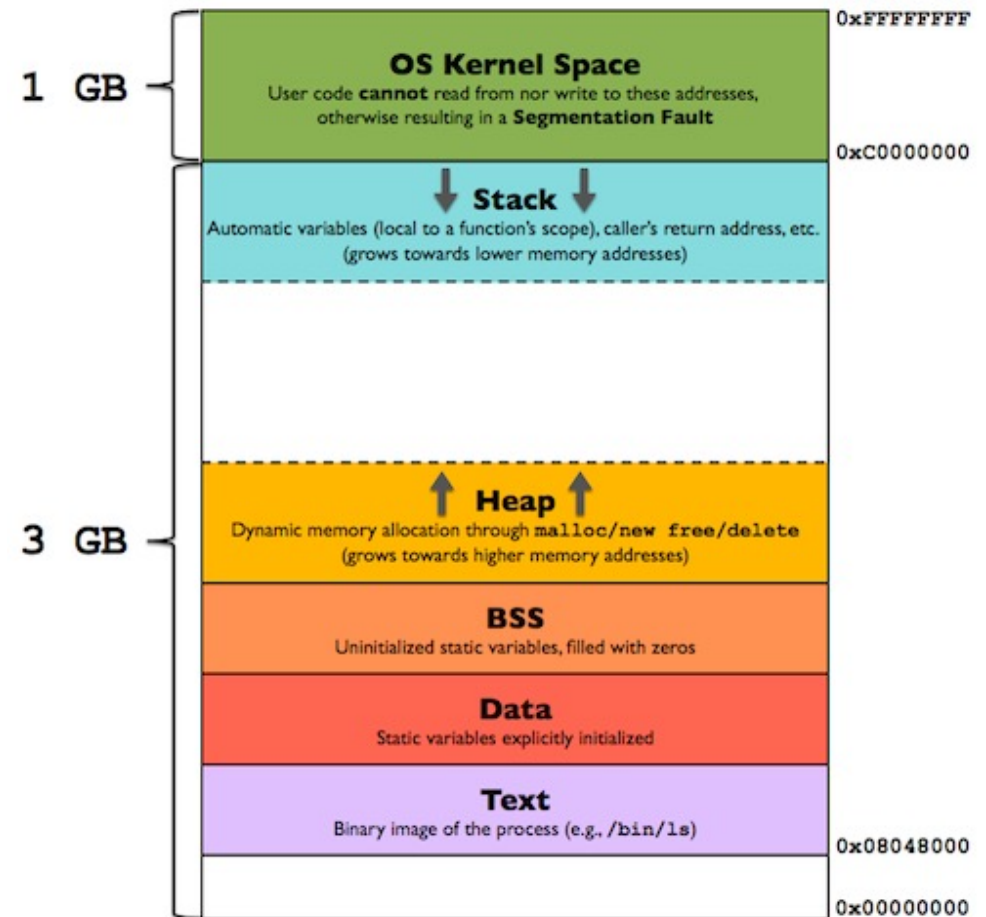
Memory Layout di un processo – parte dinamica

- ...ovvero com'è organizzata la memoria assegnata dal Sistema Operativo al processo in esecuzione

Lo stack

Struttura **LIFO (Last In First Out)**

Questo segmento è usato per memorizzare tutti i dati necessari ad una chiamata di funzione del programma. L'insieme dei valori di una funzione che vengono inseriti nello Stack viene chiamato **stack frame** e contiene l'indirizzo di ritorno del chiamante, i parametri formali con i relativi valori attuali e le variabili locali della funzione.



memoryLayout.c

```
int variabile_globale = 10;  
char s[] = "hello world";  
int i;
```

```
int main(){  
    int a = 10; //a tempo di esecuzione nello stack, ma la sua presenza aumenta la dimensione del codice binario  
    return 0;  
}
```

```
gcc memory-layout.c -o memory-layout  
size -m memoryLayout
```

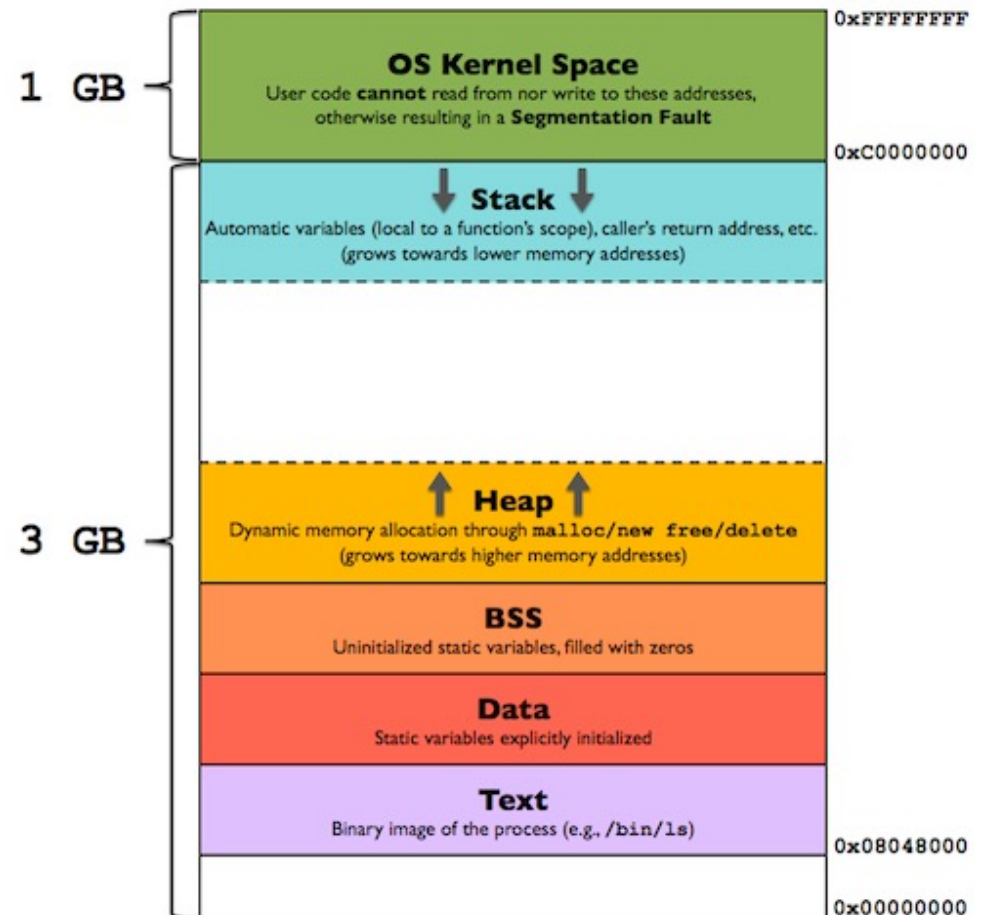
```
segment __PAGEZERO: 4294967296 (zero fill)  
segment __TEXT: 16384  
    Section __text: 28  
    Section __unwind_info: 00  
    total 116  
segment __DATA: 16384  
    Section __data: 16  
    Section __common: 4 (zerofill)  
    total 20  
segment __LINKEDIT: 16384
```


Memory Layout di un processo – parte dinamica

- ...ovvero com'è organizzata la memoria assegnata dal Sistema Operativo al processo in esecuzione

Heap

Il segmento Heap può essere considerato come un'estensione dei segmenti contenenti i dati. In questo segmento avviene l'allocazione dinamica della memoria, cioè l'allocazione delle variabili la cui dimensione può essere conosciuta solo a run-time e non può essere determinata statisticamente dal compilatore prima dell'esecuzione del programma. Il segmento Heap inizia subito dopo il segmento BSS e cresce verso gli indirizzi alti di memoria. Viene gestita tramite le istruzioni malloc/new e free/delete



Programmi .c e comando size

- Una buona descrizione si trova qui <https://www.scaler.com/topics/c/memory-layout-in-c/> e contiene anche un esempio che spiega la differenza tra stringhe dichiarate con array e stringhe dichiarate con puntatori

Stack & Heap

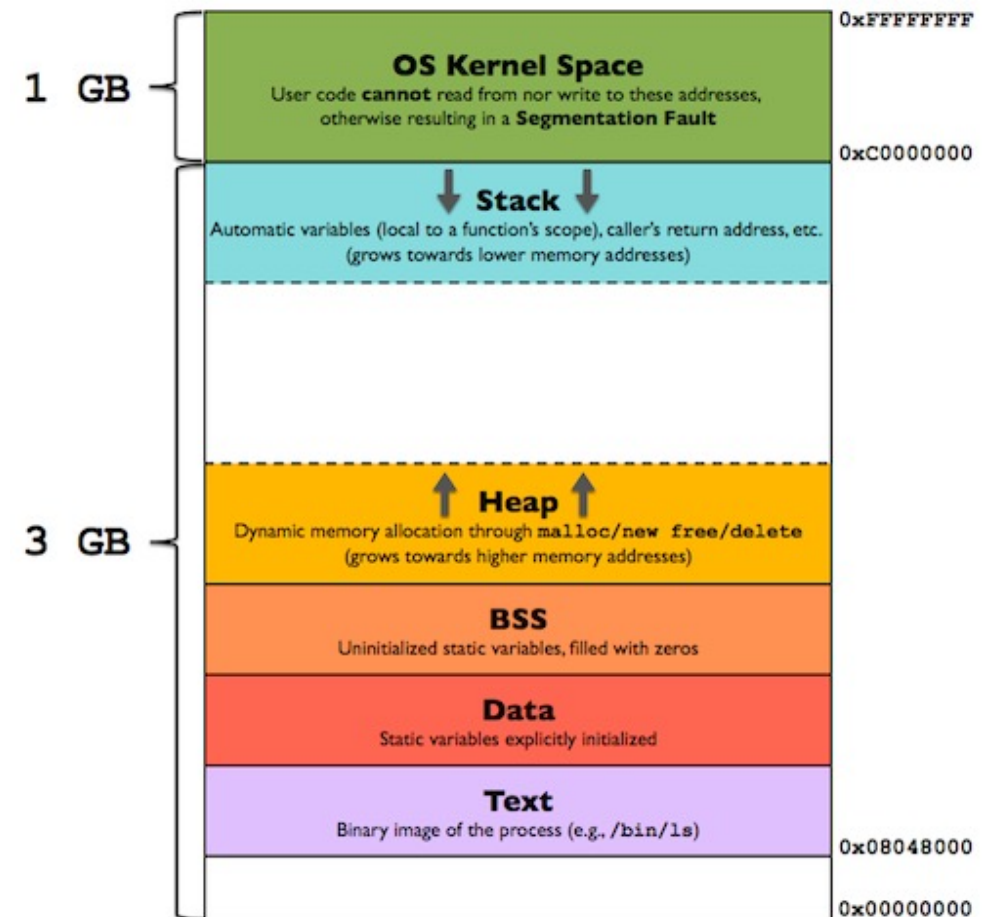
Parte dinamica della memoria RAM assegnata ad un processo da SO

Stack & Heap

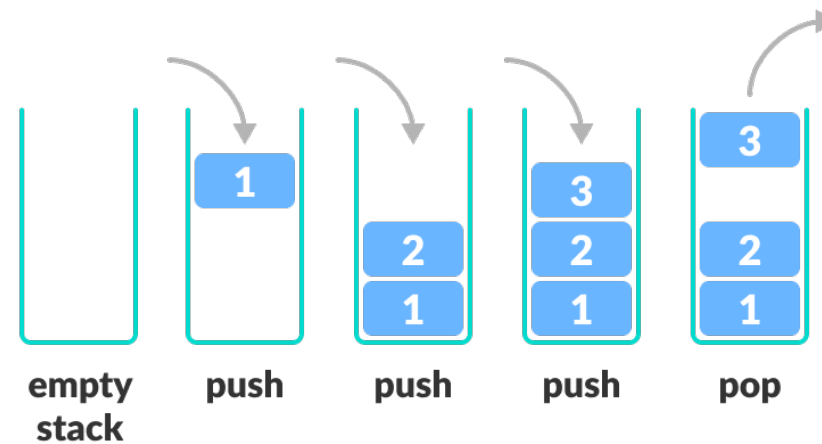
Parte dinamica della memoria RAM assegnata ad un processo da SO

Stack

- Lo stack tiene traccia della chiamata a funzioni. Tutte le volte che si effettua una chiamata ad una funzione è qui che viene salvato l'indirizzo di ritorno e le informazioni dello stato del chiamante con i parametri e le variabili locali.
- Lo stack cresce automaticamente quando una funzione chiama un'altra funzione e si riduce quando le funzioni ritornano.
- Errore di stack overflow: quando lo stack raggiunge il massimo consentito



Stack struttura dati LIFO



LAST IN FIRST OUT

Esempio record attivazione

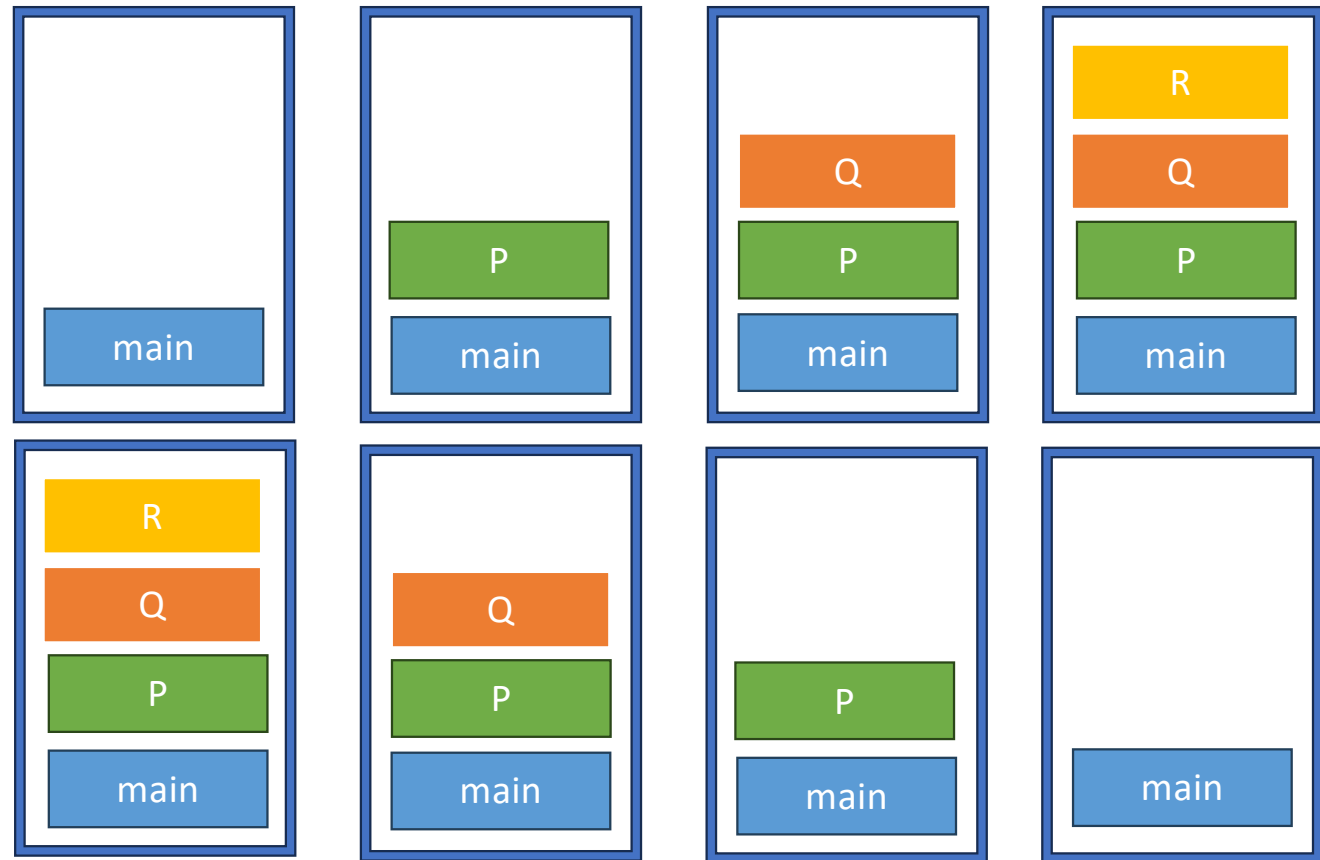
```
#include <stdio.h>

int R(int a) {
    return a + 1;
}

int Q(int x) {
    int b = x;
    return R(b);
}

int P(void) {
    int a = 10;
    return Q(a);
}

int main() {
    int x = P();
    printf("%d", x);
}
```

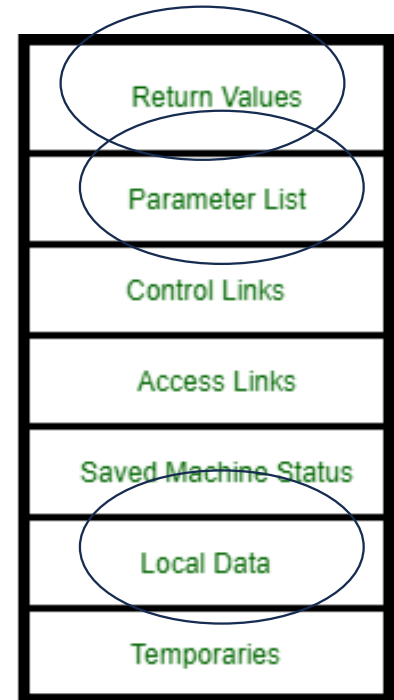


Lo stack tiene traccia della chiamata a funzioni.

Sequenza chiamate main → P() → Q() → R()

Stack – record di attivazione

- Un record di attivazione (o frame di stack) è una struttura di dati che memorizza tutte le informazioni necessarie per eseguire una chiamata di funzione.
- Questo include parametri, variabili locali, indirizzi di ritorno e altri dati di controllo.



Lo stack è utilizzato per memorizzare le variabili locali, i parametri delle funzioni e tenere traccia dei record di attivazione delle funzioni (chiamate funzioni)

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

```
int R(int a) {  
    return a + 1;  
}
```

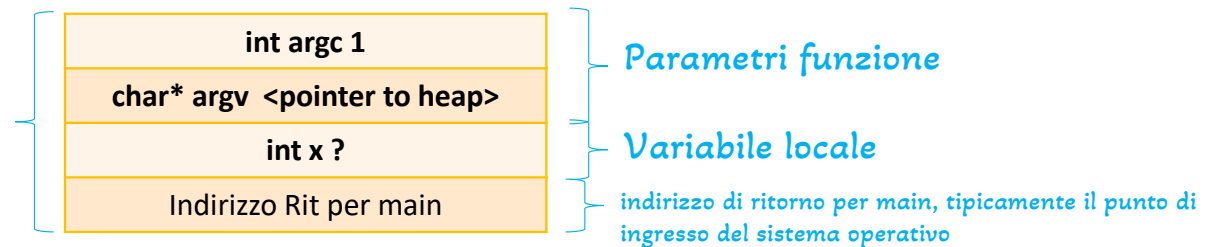
```
int Q(int x) {  
    int b = x;  
    return R(b);  
}
```

```
int P(void) {  
    int a = 10;  
    return Q(a);  
}
```

```
int main() {  
    int x = P();  
    printf("%d", x);  
}
```



Main stack frame
push




il record di attivazione di una funzione o metodo contiene anche altre informazioni e altri dati oltre ai parametri e alle variabili locali ma noi vediamo una versione semplificata

Lo stack è utilizzato per memorizzare le variabili locali, i parametri delle funzioni e tenere traccia dei record di attivazione delle funzioni (chiamate funzioni)

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

```
int R(int a) {  
    return a + 1;  
}
```

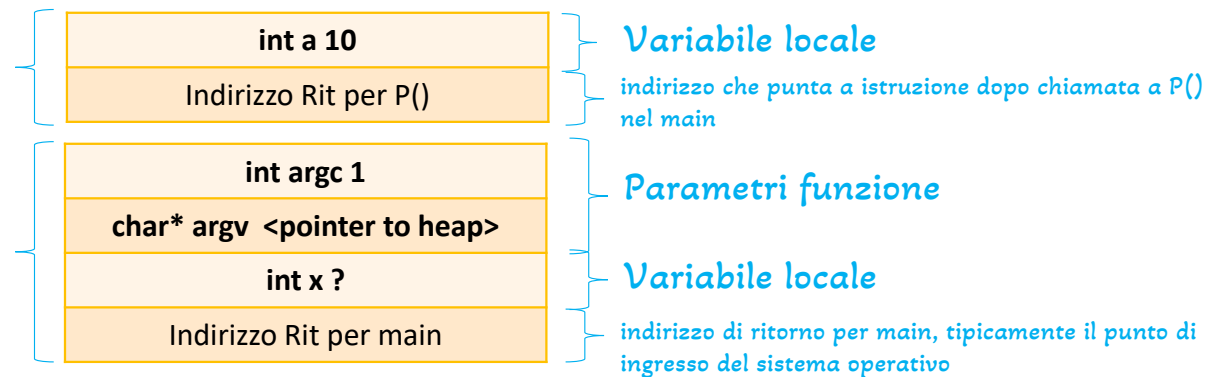
```
int Q(int x) {  
    int b = x;  
    return R(b);  
}
```

```
int P(void) {  
    int a = 10;   
    return Q(a);  
}
```

```
int main() {  
    int x = P();  
    printf("%d", x);  
}
```

P stack frame push


Main stack frame



Lo stack è utilizzato per memorizzare le variabili locali, i parametri delle funzioni e tenere traccia dei record di attivazione delle funzioni (chiamate funzioni)

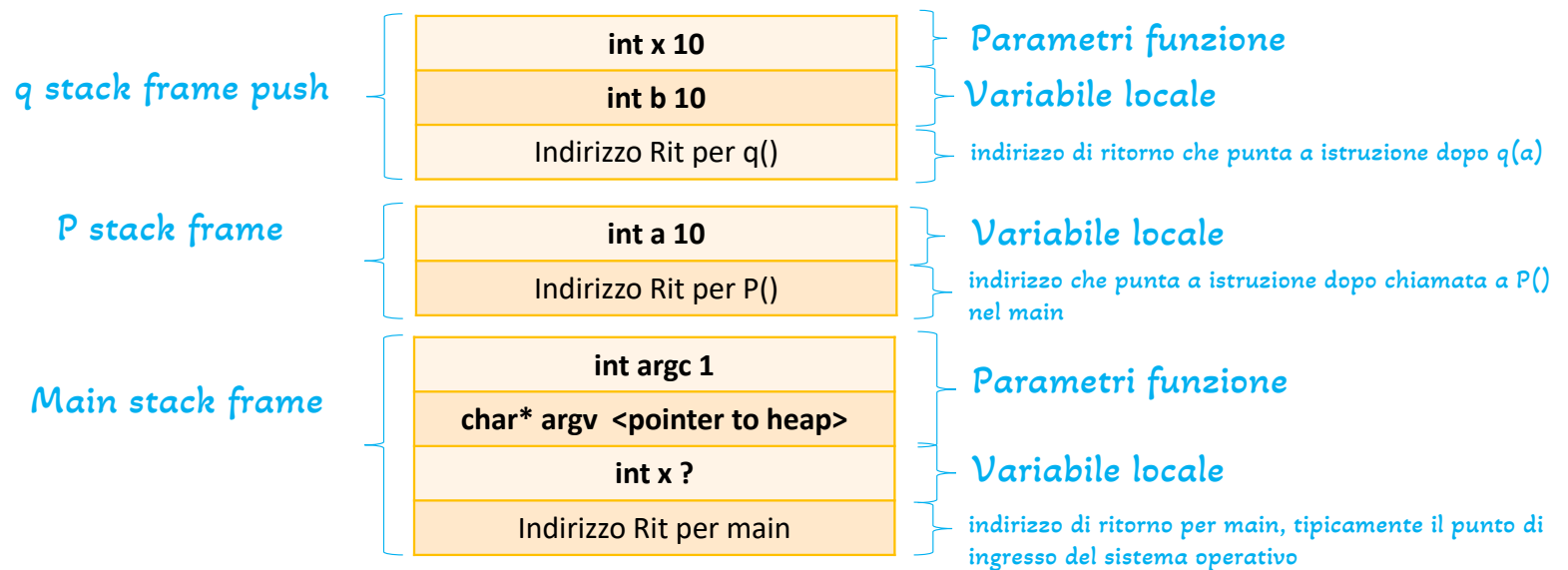
```
#include <stdio.h>
```

```
int R(int a) {  
    return a + 1;  
}
```

```
int Q(int x) {  
    int b = x;   
    return R(b);  
}
```

```
int P(void) {  
    int a = 10;  
    return Q(a);  
}
```

```
int main() {  
    int x = P();  
    printf("%d", x);  
}
```



Lo stack è utilizzato per memorizzare le variabili locali, i parametri delle funzioni e tenere traccia dei record di attivazione delle funzioni (chiamate funzioni)

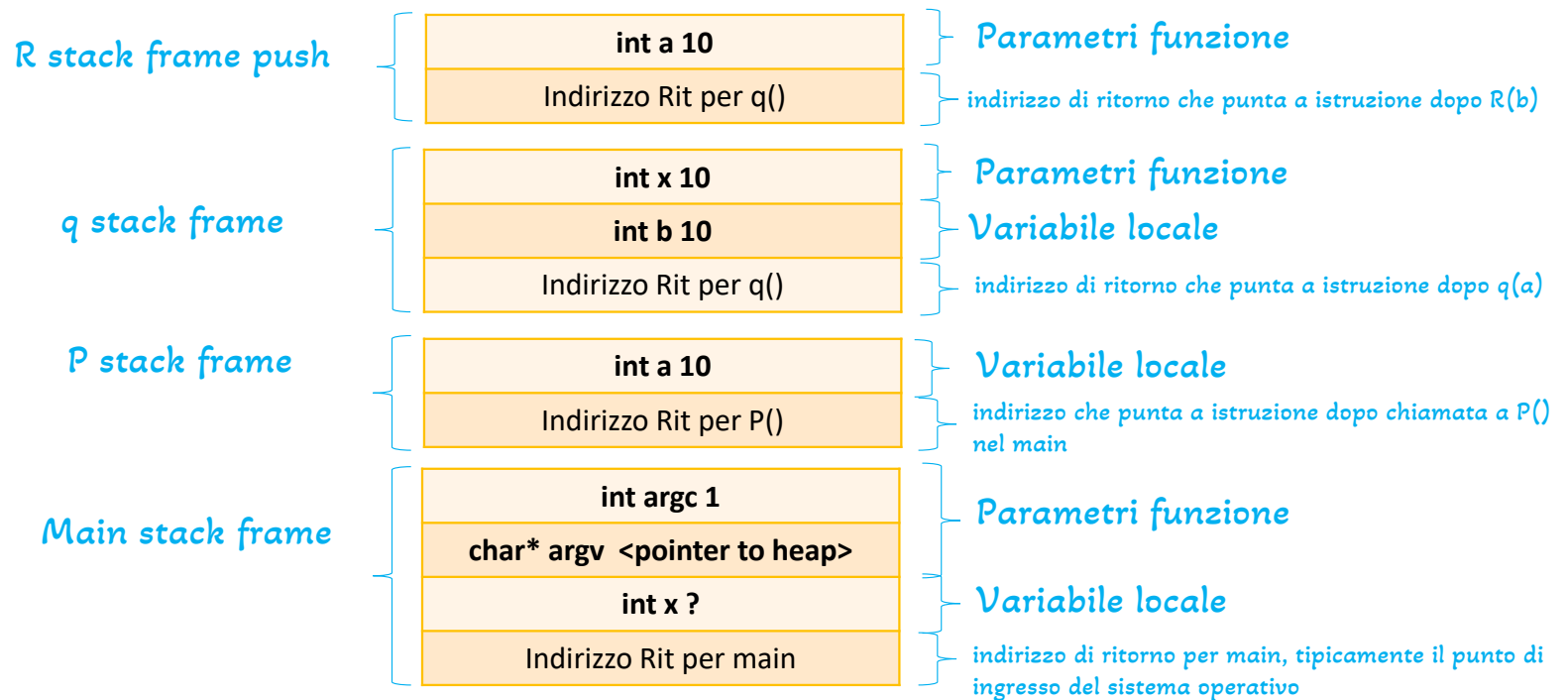
```
#include <stdio.h>
```

```
int R(int a) {  
    return a + 1; ←  
}
```

```
int Q(int x) {  
    int b = x;  
    return R(b);  
}
```

```
int P(void) {  
    int a = 10;  
    return Q(a);  
}
```

```
int main() {  
    int x = P();  
    printf("%d", x);  
}
```



Lo stack è utilizzato per memorizzare le variabili locali, i parametri delle funzioni e tenere traccia dei record di attivazione delle funzioni (chiamate funzioni)

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

```
int R(int a) {  
    return a + 1;  
}
```

```
int Q(int x) {  
    int b = x;  
    return R(b); ←
```

```
int P(void) {  
    int a = 10;  
    return Q(a);  
}
```

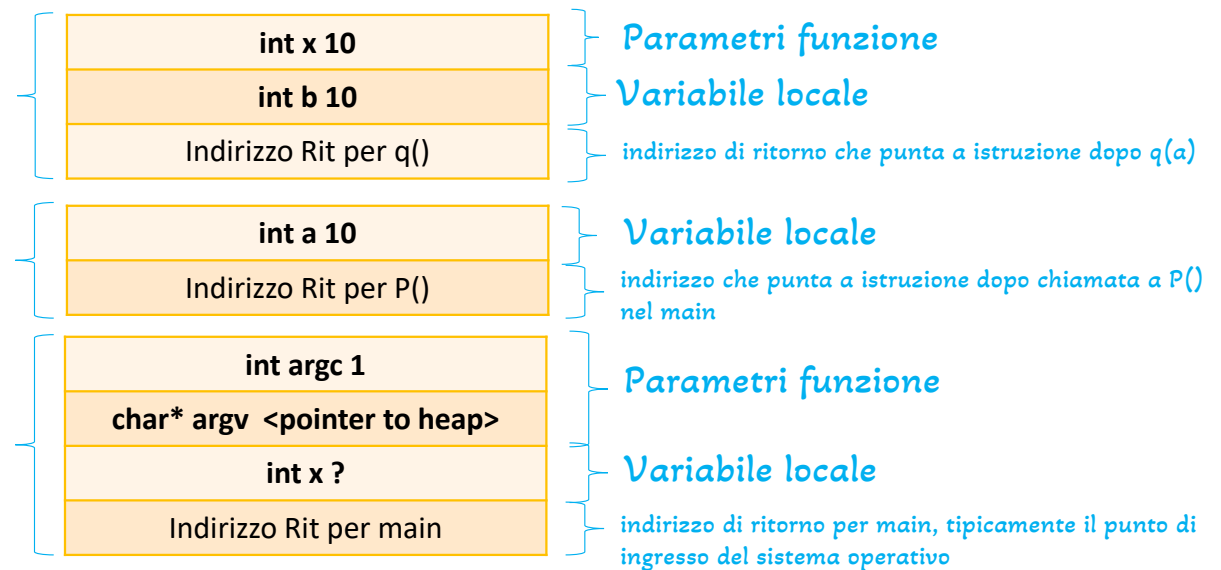
```
int main() {  
    int x = P();  
    printf("%d", x);  
}
```

R stack frame pop

q stack frame

P stack frame

Main stack frame



Lo stack è utilizzato per memorizzare le variabili locali, i parametri delle funzioni e tenere traccia dei record di attivazione delle funzioni (chiamate funzioni)

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

```
int R(int a) {  
    return a + 1;  
}
```

```
int Q(int x) {  
    int b = x;  
    return R(b);  
}
```

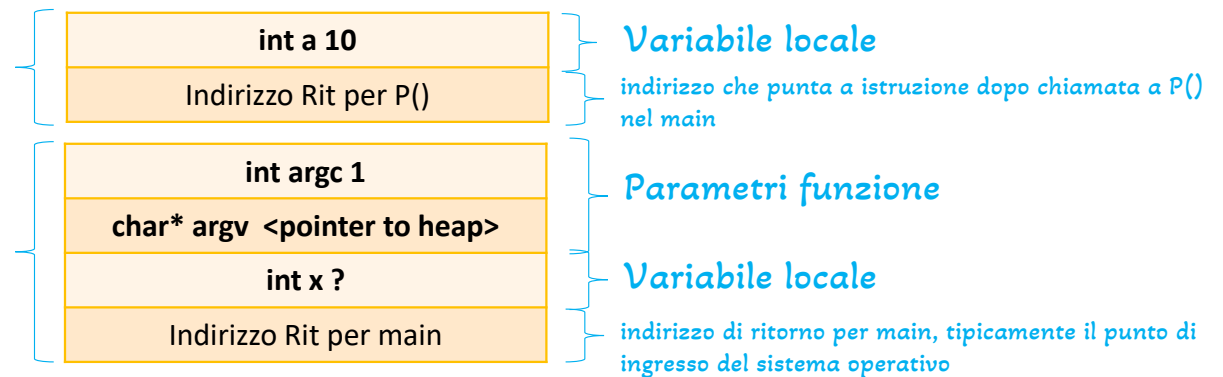
q stack frame pop

```
int P(void) {  
    int a = 10;  
    return Q(a); ←
```

```
int main() {  
    int x = P();  
    printf("%d", x);  
}
```

P stack frame

Main stack frame



Lo stack è utilizzato per memorizzare le variabili locali, i parametri delle funzioni e tenere traccia dei record di attivazione delle funzioni (chiamate funzioni)

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

```
int R(int a) {  
    return a + 1;  
}
```

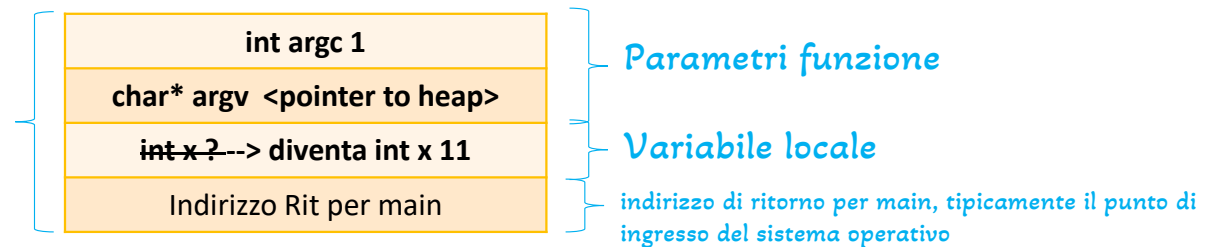
```
int Q(int x) {  
    int b = x;  
    return R(b);  
}
```

```
int P(void) {  
    int a = 10;  
    return Q(a);  
}
```

```
int main() {  
    int x = P();  
    printf("%d", x);  
}
```

P stack frame pop

Main stack frame



Lo stack è utilizzato per memorizzare le variabili locali, i parametri delle funzioni e tenere traccia dei record di attivazione delle funzioni (chiamate funzioni)

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

```
int R(int a) {  
    return a + 1;  
}
```

```
int Q(int x) {  
    int b = x;  
    return R(b);  
}
```

```
int P(void) {  
    int a = 10;  
    return Q(a);  
}
```

```
int main() {  
    int x = P();  
    printf("%d", x);  
}
```

Main stack frame
pop



FINE ESECUZIONE!

Link utili

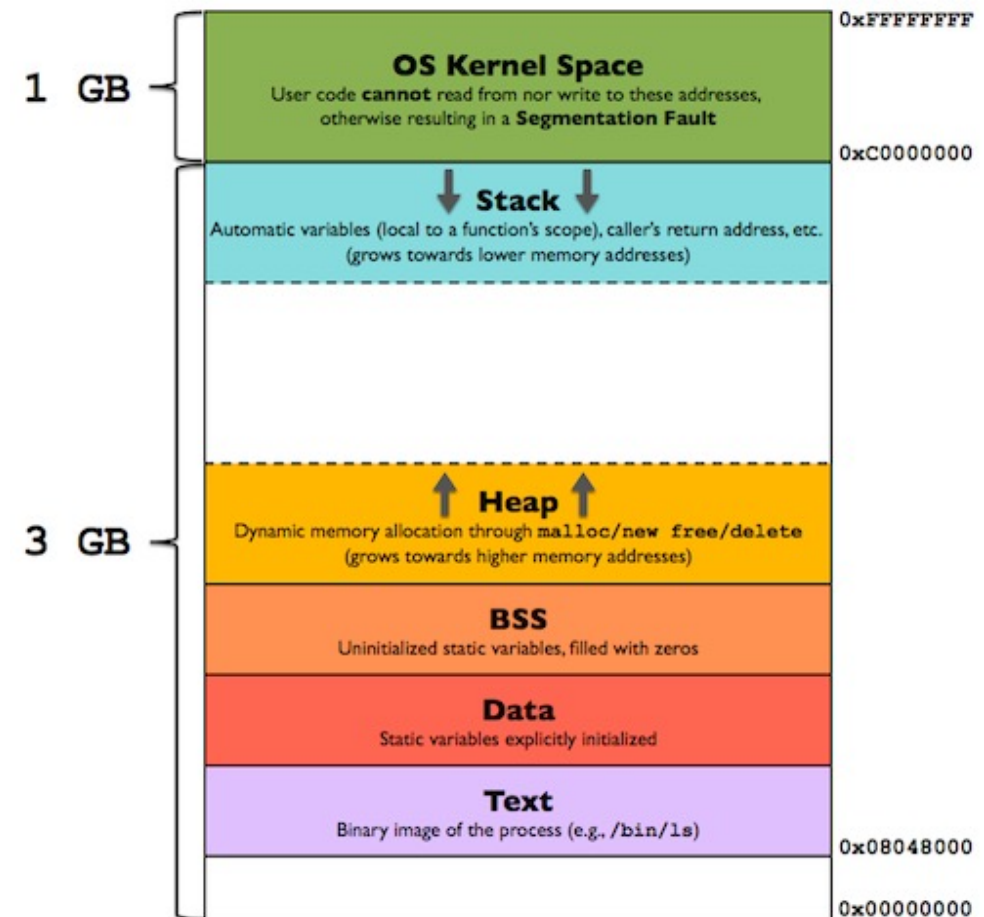
- Una lezione interattiva su stack e heap
 - <https://courses.engr.illinois.edu/cs225/fa2022/resources/stack-heap/>
- Tool online che simula stack e heap per diversi linguaggi
 - <https://pythontutor.com/render.html#mode=display>

Stack & Heap

Parte dinamica della memoria RAM assegnata ad un processo da SO

Heap

- L'heap è utilizzato per l'allocazione dinamica della memoria.
- Quando nel programma vengono utilizzate funzioni come `malloc()` in C o `new` in C++ o Java, viene allocata memoria nell'heap.
- La dimensione dell'heap può crescere o diminuire a seconda delle esigenze del programma. L'allocazione e la deallocazione della memoria nell'heap sono gestite esplicitamente dal programma in C, dal garbage collector in Java
- Ci servono i puntatori per fare un esempio...



Esempio puntatori

In C, i simboli & e * sono strettamente legati al concetto di puntatori e indirizzi di memoria.

L'operatore & viene utilizzato per ottenere l'indirizzo di memoria di una variabile.

Ad esempio, se hai una variabile `int y = 5;`, scrivendo `&y` otterrai l'indirizzo di memoria di `y`.

L'operatore * viene utilizzato per accedere al valore memorizzato all'indirizzo a cui punta un puntatore.

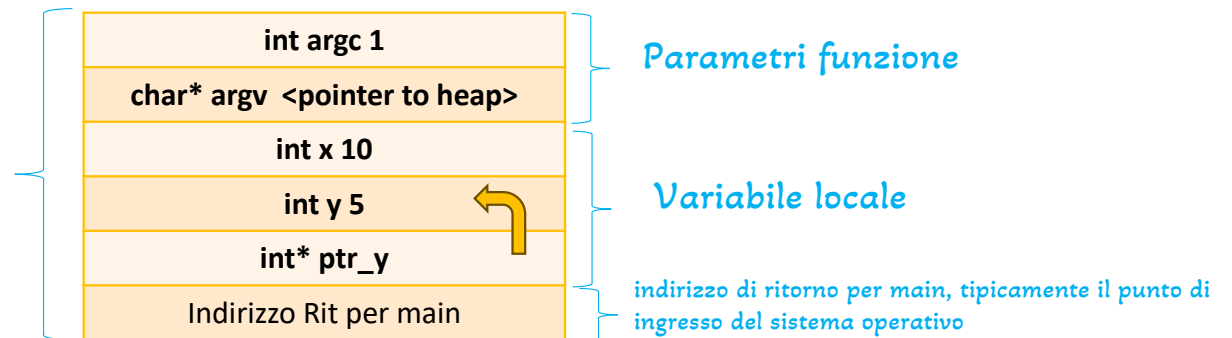
Ad esempio, se `int* ptr` è un puntatore a intero e punta all'indirizzo di `y`, scrivendo `*ptr` otterrai il valore memorizzato in `y` (cioè 5).

```
int main() {  
    int x = 10;  
    int y = 5;  
    int *ptr_y = &y;  
    printf("%d", x);  
    printf("%d", y);  
    printf("%d", ptr_y);  
    printf("%d", &ptr_y);  
    printf("%d", *ptr_y);  
}
```

Output

```
/Users/cristina/GIT_AV0/Avo/C/misc/stack  
10  
5  
1808249768  
1808249760  
5
```

Main stack frame
push



Esempio puntatori

In C, i simboli & e * sono strettamente legati al concetto di puntatori e indirizzi di memoria.

L'operatore & viene utilizzato per ottenere l'indirizzo di memoria di una variabile.

Ad esempio, se hai una variabile `int y = 5;`, scrivendo `&y` otterrai l'indirizzo di memoria di `y`.

L'operatore * viene utilizzato per accedere al valore memorizzato all'indirizzo a cui punta un puntatore.

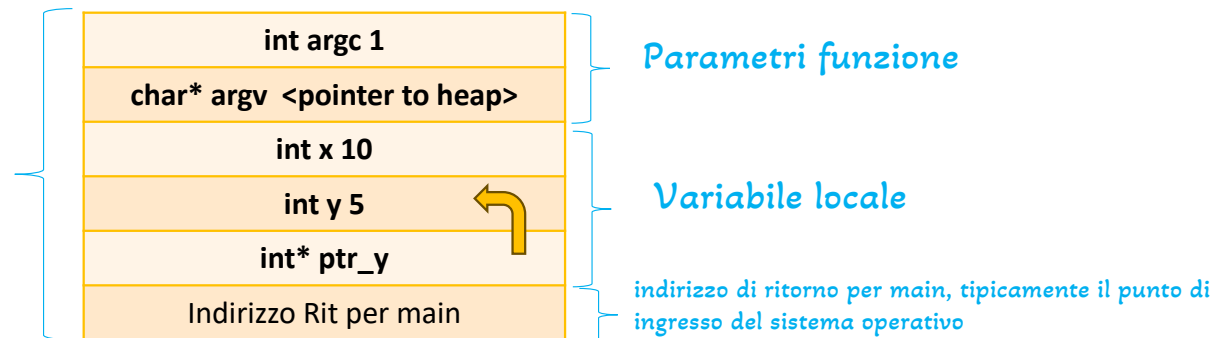
Ad esempio, se `int* ptr` è un puntatore a intero e punta all'indirizzo di `y`, scrivendo `*ptr` otterrai il valore memorizzato in `y` (cioè 5).

```
int main() {  
    int x = 10;  
    int y = 5;  
    int *ptr_y = &y;  
    printf("%d", x);  
    printf("%d", y);  
    printf("%d", ptr_y);  
    printf("%d", &ptr_y);  
    printf("%d", *ptr_y);  
}
```

Output

```
/Users/cristina/GIT_AV0/Avo/C/misc/stack  
10  
5  
1808249768  
1808249760  
5
```

Main stack frame
push



E lo heap???? Ci serve la malloc...

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

```
int main() {
    // Allocazione nell'heap
    int *ptr = (int*)malloc(sizeof(int)); ←
    if (ptr == NULL) {
        printf("Allocazione di memoria fallita.\n");
        return 1;
    }

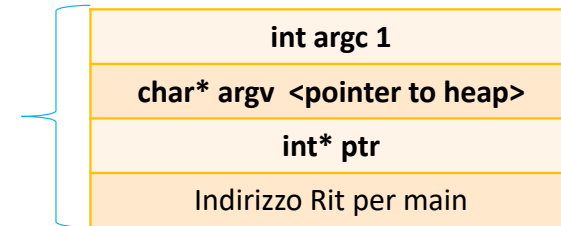
    *ptr = 5; // Assegnazione di un valore

    printf("Valore dell'intero nell'heap: %d\n", *ptr);

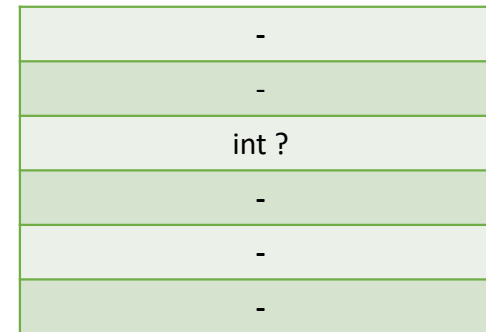
    free(ptr); // Rilascio della memoria

    return 0;
}
```

Main stack frame
push



Heap



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

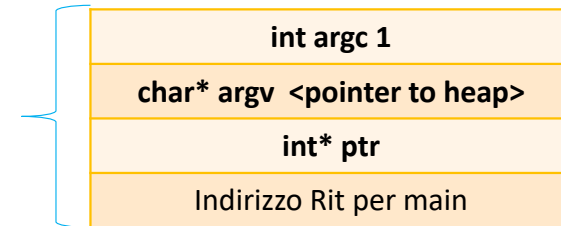
```
int main() {
    // Allocazione nell'heap
    int *ptr = (int*)malloc(sizeof(int));
    if (ptr == NULL) {
        printf("Allocazione di memoria fallita.\n");
        return 1;
    }

    *ptr = 5; // Assegnazione di un valore ←
    printf("Valore dell'intero nell'heap: %d\n", *ptr);

    free(ptr); // Rilascio della memoria

    return 0;
}
```

Main stack frame
push



Heap



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

```
int main() {
    // Allocazione nell'heap
    int *ptr = (int*)malloc(sizeof(int));
    if (ptr == NULL) {
        printf("Allocazione di memoria fallita.\n");
        return 1;
    }

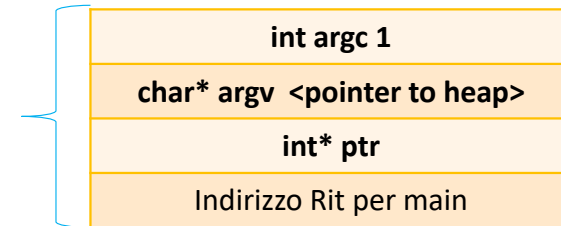
    *ptr = 5; // Assegnazione di un valore

    printf("Valore dell'intero nell'heap: %d\n", *ptr);

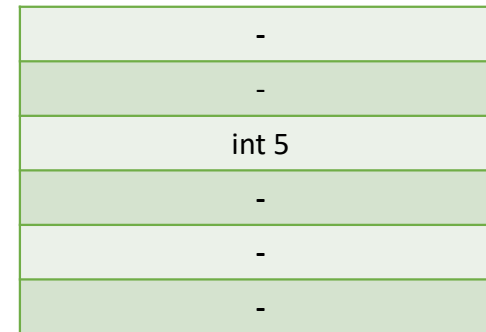
    free(ptr); // Rilascio della memoria

    return 0;
}
```

Main stack frame
push



Heap



L'operatore * viene utilizzato per accedere al valore memorizzato all'indirizzo a cui punta un puntatore.


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

```
int main() {

    int *array;
    int n = 10; // Dimensione dell'array

    // Allocazione di memoria per 10 interi
    array = (int*)malloc(n * sizeof(int));

    // Verifica se l'allocazione di memoria è riuscita
    if (array == NULL) {
        printf("Allocazione di memoria fallita.\n");
        return 1; // o exit(EXIT_FAILURE);
    }

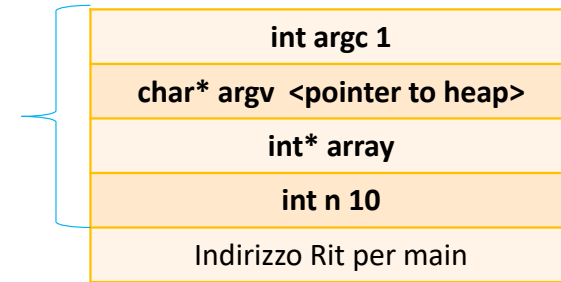
    // Inizializzazione dell'array
    for (int i = 0; i < n; i++) {
        array[i] = i; // Assegna alcuni valori (es. 0, 1, 2, ..., 9)
    }

    // Stampa dell'array
    for (int i = 0; i < n; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");

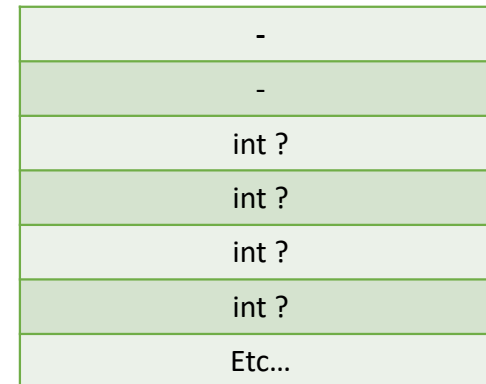
    // Rilascio della memoria allocata
    free(array);

    return 0;
}
```

Main stack frame
push



Heap



Esempio stack Java

```
public class Main {  
  
    1 usage  
    public static void metodoConParametro(int parametro) {  
        int variabileLocale = 5;  
        System.out.println("Parametro: " + parametro + ", Variabile Locale: " + variabileLocale);  
    }  
  
    public static void main(String[] args) {  
        int variabileInMain = 3;  
        metodoConParametro(variabileInMain);  
    }  
}
```

Main stack frame
push

String[] args
int variabileInMain 3
Indirizzo Rit per main

Esempio stack Java

```
public class Main {  
  
    1 usage  
    public static void metodoConParametro(int parametro) {  
        int variabileLocale = 5;  
        System.out.println("Parametro: " + parametro + ", Variabile Locale: " + variabileLocale);  
    }  
  
    public static void main(String[] args) {  
        int variabileInMain = 3;  
        metodoConParametro(variabileInMain);  
    }  
}
```

metodoConParametro
o stack frame push

Main stack frame

int parametro 3
int variabileLocale 5
Indirizzo Rit per metodoConParametro

String[] args
int variabileInMain 3
Indirizzo Rit per main

Esempio stack Java

```
public class Main {  
  
    1 usage  
    public static void metodoConParametro(int parametro) {  
        int variabileLocale = 5;  
        System.out.println("Parametro: " + parametro + ", Variabile Locale: " + variabileLocale);  
    }  
  
    public static void main(String[] args) {  
        int variabileInMain = 3;  
        metodoConParametro(variabileInMain);  
    }  
}
```

metodoConParametro
o stack frame pop

Main stack frame

String[] args
int variabileInMain 3
Indirizzo Rit per main

Esempio stack Java

```
public class Main {  
  
    1 usage  
    public static void metodoConParametro(int parametro) {  
        int variabileLocale = 5;  
        System.out.println("Parametro: " + parametro + ", Variabile Locale: " + variabileLocale);  
    }  
  
    public static void main(String[] args) {  
        int variabileInMain = 3;  
        metodoConParametro(variabileInMain);  
    }  
}
```

```
/Library/Java/JavaVirtualMachines/jdk  
Parametro: 3, Variabile Locale: 5  
  
Process finished with exit code 0
```

Main stack frame
pop

FINE ESECUZIONE!

Java & heap

- In C la memoria dello heap viene allocata dinamicamente dal programmatore usando sistem calls apposite come malloc()
- In Java, gli oggetti sono creati nello heap quando il programmatore crea l'oggetto con **new**
 - Quando usi la parola chiave new per creare un oggetto, Java alloca memoria nello heap per quell'oggetto e restituisce un riferimento a quella memoria.
 - Questo è diverso dalle variabili primitive (come int, double, char, ecc.), che possono essere allocate nello stack quando vengono dichiarate come variabili locali all'interno di un metodo.

Esempio stack & heap

```
public class Esempio {  
    2 usages  
    String variabileIstanza = "Sono una var istanza";  
    1 usage  
    public void metodoConParametro(String str) {  
        this.variabileIstanza = str;  
    }  
  
    1 usage  
    public String getVariabileIstanza() {  
        return this.variabileIstanza;  
    }  
  
    public static void main(String[] args) {  
        int variabileInMain = 3;  
        Esempio es = new Esempio();  
        es.metodoConParametro(str: "nuovo valore");  
        System.out.println("Ora var istanza vale: "+es.getVariabileIstanza());  
    }  
}
```

Main stack frame

String[] args
int variabileInMain 3
Esempio es
Indirizzo Rit per main

Heap

variabileIstanza

"sono una var istanza"

Esempio stack & heap

```
public class Esempio {  
    2 usages  
    String variabileIstanza = "Sono una var istanza";  
    1 usage  
    public void metodoConParametro(String str) {  
        this.variabileIstanza = str;  
    }  
  
    1 usage  
    public String getVariabileIstanza() {  
        return this.variabileIstanza;  
    }  
  
    public static void main(String[] args) {  
        int variabileInMain = 3;  
        Esempio es = new Esempio();  
        es.metodoConParametro( str: "nuovo valore");  
        System.out.println("Ora var istanza vale: "+es.getVariabileIstanza());  
    }  
}
```

metodoConParametro
stack frame push

this
String str "nuovo valore"
Indirizzo Rit per metodoConParametro

Main stack frame

String[] args
int variabileInMain 3
Esempio es
Indirizzo Rit per main

Heap

variabileIstanza	"sono una var istanza"
------------------	---------------------------

Esempio stack & heap

```
public class Esempio {  
    2 usages  
    String variabileIstanza = "Sono una var istanza";  
    1 usage  
    public void metodoConParametro(String str) {  
        this.variabileIstanza = str;  
    }  
  
    1 usage  
    public String getVariabileIstanza() {  
        return this.variabileIstanza;  
    }  
  
    public static void main(String[] args) {  
        int variabileInMain = 3;  
        Esempio es = new Esempio();  
        es.metodoConParametro( str: "nuovo valore");  
        System.out.println("Ora var istanza vale: "+es.getVariabileIstanza());  
    }  
}
```

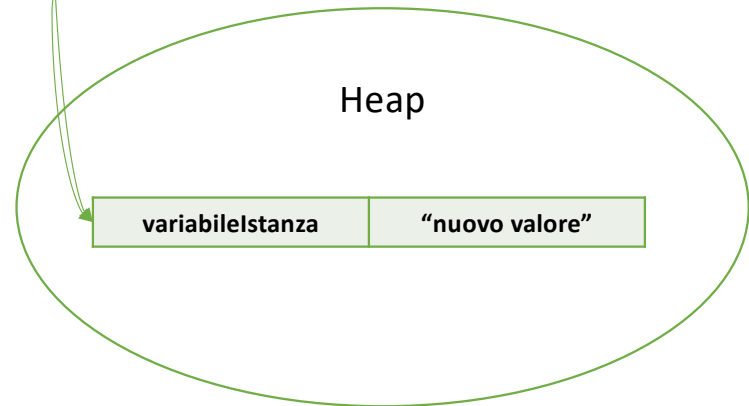
metodoConParametro
stack frame push

this
String str "nuovo valore"
Indirizzo Rit per metodoConParametro

Main stack frame

String[] args
int variabileInMain 3
Esempio es
Indirizzo Rit per main

Heap



Esempio stack & heap

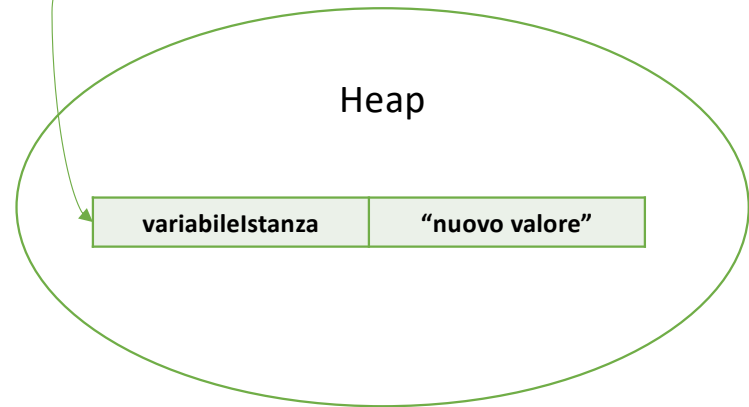
```
public class Esempio {  
    2 usages  
    String variabileIstanza = "Sono una var istanza";  
    1 usage  
    public void metodoConParametro(String str) {  
        this.variabileIstanza = str;  
    }  
  
    1 usage  
    public String getVariabileIstanza() {  
        return this.variabileIstanza;  
    }  
  
    public static void main(String[] args) {  
        int variabileInMain = 3;  
        Esempio es = new Esempio();  
        es.metodoConParametro( str: "nuovo valore");  
        System.out.println("Ora var istanza vale: "+es.getVariabileIstanza());  
    }  
}
```

metodoConParametro
stack frame pop

Main stack frame

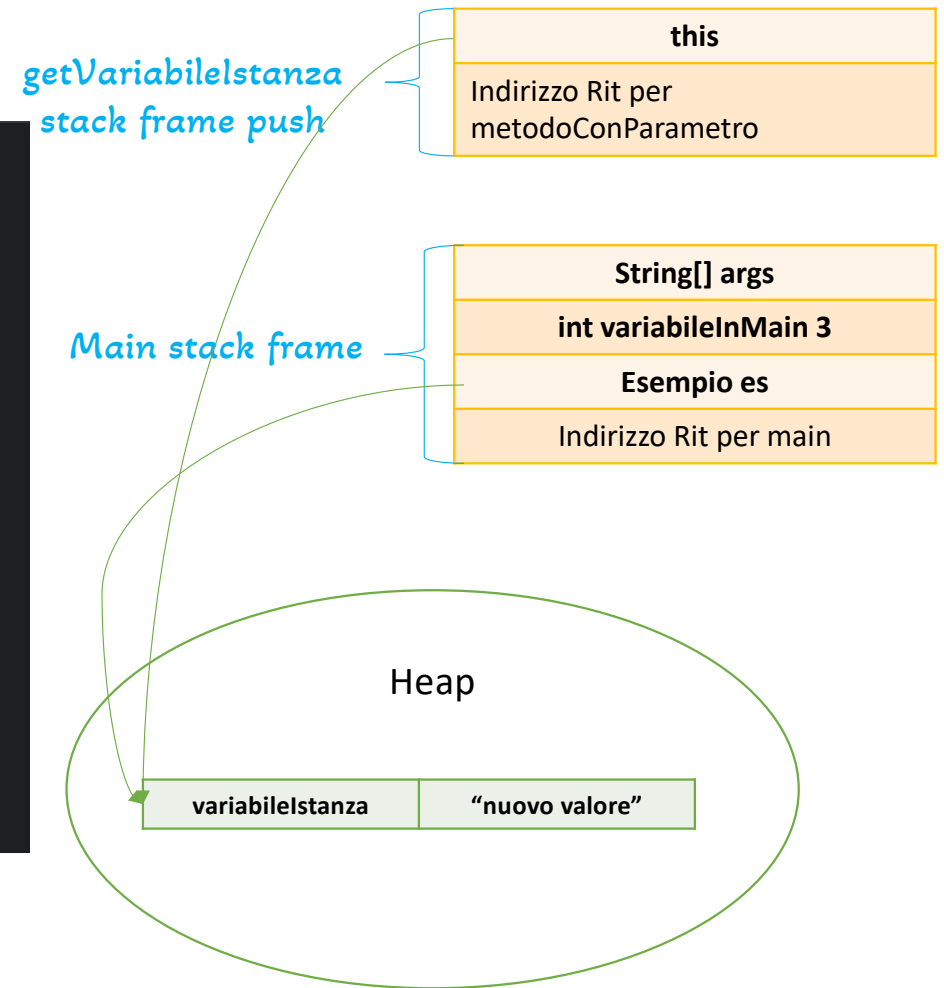
String[] args
int variabileInMain 3
Esempio es
Indirizzo Rit per main

Heap



Esempio stack & heap

```
public class Esempio {  
    2 usages  
    String variabileIstanza = "Sono una var istanza";  
    1 usage  
    public void metodoConParametro(String str) {  
        this.variabileIstanza = str;  
    }  
  
    1 usage  
    public String getVariabileIstanza() {  
        return this.variabileIstanza;  
    }  
  
    public static void main(String[] args) {  
        int variabileInMain = 3;  
        Esempio es = new Esempio();  
        es.metodoConParametro( str: "nuovo valore");  
        System.out.println("Ora var istanza vale: "+es.getVariabileIstanza());  
    }  
}
```



Esempio stack & heap

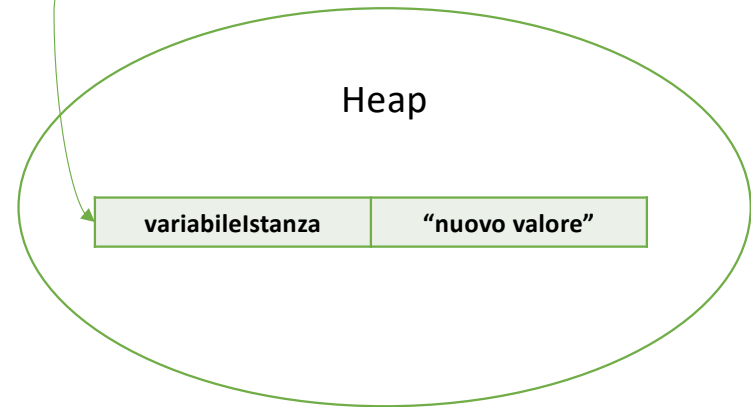
```
public class Esempio {  
    2 usages  
    String variabileIstanza = "Sono una var istanza";  
    1 usage  
    public void metodoConParametro(String str) {  
        this.variabileIstanza = str;  
    }  
  
    1 usage  
    public String getVariabileIstanza() {  
        return this.variabileIstanza;  
    }  
  
    public static void main(String[] args) {  
        int variabileInMain = 3;  
        Esempio es = new Esempio();  
        es.metodoConParametro( str: "nuovo valore");  
        System.out.println("Ora var istanza vale: "+es.getVariabileIstanza());  
    }  
}
```

getVariabileIstanza
stack frame pop

Main stack frame

String[] args
int variabileInMain 3
Esempio es
Indirizzo Rit per main

Heap



Esempio stack & heap

```
public class Esempio {  
    2 usages  
    String variabileIstanza = "Sono una var istanza";  
    1 usage  
    public void metodoConParametro(String str) {  
        this.variabileIstanza = str;  
    }  
  
    1 usage  
    public String getVariabileIstanza(){  
        return this.variabileIstanza;  
    }  
  
    public static void main(String[] args) {  
        int variabileInMain = 3;  
        Esempio es = new Esempio();  
        es.metodoConParametro( str: "nuovo valore");  
        System.out.println("Ora var istanza vale: "+es.getVariabileIstanza());  
    }  
}
```

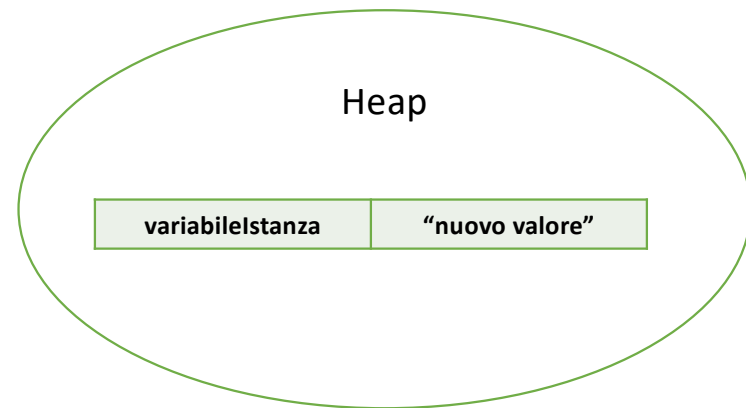
```
/Library/Java/JavaVirtualMachines/jdk-1  
Ora var istanza vale: nuovo valore
```

getVariabileIstanza
stack frame pop

Main stack frame

FINE ESECUZIONE!

Il Garbage Collector (GC) in Java si occupa di liberare la memoria occupata da oggetti che non sono più raggiungibili, ovvero oggetti a cui non esistono più riferimenti attivi nel programma.



C: passaggio per valore versus per indirizzo

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

void addValue(int a, int b){
    a += 1;
    b += 1;
}

void addRef(int *a, int *b){
    *a +=1;
    *b +=1;
}

int main(int argc, char *argv[]) {
    int x = 4;
    int y = 5;
    printf("%d \n", x);
    printf("%d \n", y);
    addValue(x,y);
    printf("dopo passaggio per valore x e y valgono\n");
    printf("%d \n", x);
    printf("%d \n", y);
    addRef(&x, &y);
    printf("dopo passaggio per riferimento x e y valgono\n");
    printf("%d \n", x);
    printf("%d \n", y);
}
```

Passaggio per valore e riferimento

In programmazione, il passaggio di parametri a una funzione può avvenire principalmente in due modi: passaggio per valore e passaggio per riferimento. La scelta tra questi due metodi dipende dal linguaggio di programmazione e dal comportamento desiderato durante la chiamata della funzione.

1. Passaggio per Valore:

1. Nel passaggio per valore, viene passata una copia del valore effettivo del parametro alla funzione.
2. Le modifiche apportate al parametro all'interno della funzione non influenzano la variabile originale nel contesto chiamante.
3. I tipi primitivi in molti linguaggi, come Java e C, vengono passati per valore.

2. Passaggio per Riferimento:

1. Nel passaggio per riferimento, viene passato un riferimento (o un puntatore) alla variabile effettiva, piuttosto che una copia del suo valore.
2. Le modifiche apportate al parametro all'interno della funzione influenzano direttamente la variabile originale nel contesto chiamante.
3. Questo metodo è comune in linguaggi come C++ (con puntatori e riferimenti) e Python/Java (dove gli oggetti, come liste e dizionari, sono passati per riferimento).

```
public class Esempio {
```

```
    private static class InnerClass{
```

non ha bisogno di accedere ai membri di Esempio quindi classe statica

```
        private int number;
```

```
        public InnerClass(int number){
```

```
            this.number = number;
```

```
        }
```

```
        public int getNumber(){
```

```
            return this.number;
```

```
        }
```

```
        public void multiply() {
```

```
            this.number *= 2;
```

```
        }
```

```
    }
```

```
    public static int multiply(int firstParameter){
```

```
        firstParameter *= 2;
```

```
        return firstParameter;
```

```
    }
```

Metodo statico può essere chiamato senza creare un'istanza della classe

```
    public static void main(String[] args){
```

```
        int num = 10;
```

```
        InnerClass ex = new InnerClass(num);
```

```
        System.out.println("Valore -> before num int local main var " + num);
```

```
        // qui dovrei fare num = Esempio.multiply(num);
```

```
        Esempio.multiply(num);
```

```
        System.out.println("Valore -> after num int local main var " + num);
```

```
        System.out.println("passaggio per riferimento " + num);
```

```
        System.out.println("Rif -> before num int local main var " + ex.getNumber());
```

```
        ex.multiply();
```

```
        System.out.println("Rif -> after num int local main var " + ex.getNumber());
```

```
    }
```

```
}
```

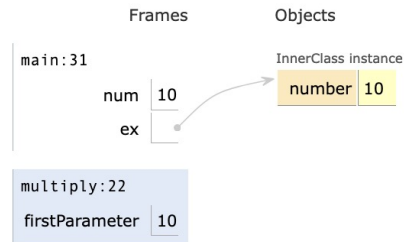
Nel caso del passaggio per valore, il valore originale di **num** non cambia dopo la chiamata al metodo **multiply**, mentre nel caso del passaggio per riferimento, il valore all'interno dell'oggetto **ex** cambia dopo la chiamata al metodo **multiply**.

1

Java
[known limitations](#)

Print output (drag lower right corner to resize)

Valore -> before num int local main var 10

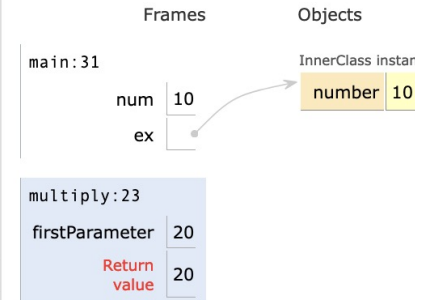


2

Java
[known limitations](#)

Print output (drag lower right corner to resize)

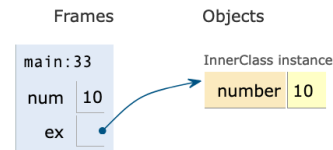
Valore -> before num int local main va



3

Java
[known limitations](#)

Print output (drag lower right corner to resize)

Valore -> before num int local main var 10
Valore -> after num int local main var 10

Passaggio parametri attuali per valore

Java: tipi primitivi passati per valore, viene passata una copia. La variabile originale non si modifica.

1

```

4      private int number;
5
6      public InnerClass(int number){
7          this.number = number;
8      }
9
10     public int getNumber(){
11         return this.number;
12     }
13
14     public void multiply() {
15         this.number *= 2;
16     }
17
18 }
19
20
21 public static int multiply(int firstParameter){
22     firstParameter *= 2;
23     return firstParameter;
24 }

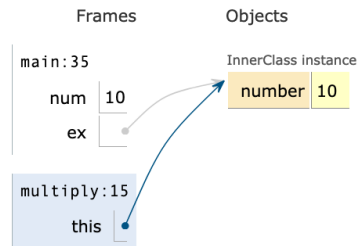
```

Print output (drag lower right corner to resize)

```

Valore -> before num int local main var 10
Valore -> after num int local main var 10
passaggio per riferimento 10
Rif -> before num int local main var 10

```



Passaggio parametri attuali per riferimento

Java: oggetti passati per riferimento, viene passato un riferimento a oggetto nello Heap. L'oggetto si modifica.

2

```

19
20
21 public static int multiply(int firstParameter){
22     firstParameter *= 2;
23     return firstParameter;
24 }
25
26
27 public static void main(String[] args){
28     int num = 10;
29     InnerClass ex = new InnerClass(num);
30     System.out.print("Valore -> before num int local m
31     Esempio.multiply(num);
32     System.out.print("Valore -> after num int local ma
33     System.out.print("passaggio per riferimento " + n
34     System.out.print("Rif -> before num int local main
35     ex.multiply();
36     System.out.print("Rif -> after num int local main
37
38 }
39 }

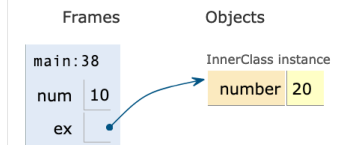
```

Print output (drag lower right corner to resize)

```

Valore -> before num int local main var 10
Valore -> after num int local main var 10
passaggio per riferimento 10
Rif -> before num int local main var 10
Rif -> after num int local main var 20

```



Approfondimenti

Curiosare con la shell la memoria di un processo

```
//file sleep.c
#include <stdio.h>
#include <unistd.h>
int global_var = -40;
int main() {
    printf("Il programma inizia...\n");
    int a = 10;
    int b = 20;
    int c = a + b;
    // Dormi per 10 secondi
    sleep(300);
```

```
    printf("Il programma è stato svegliato dopo 10 secondi.\n");
    printf("Somma %d\n", c);

    return 0;
}
```

```
ps aux | grep sleep
```

```
sudo vmmap 4946 > vmmap.txt
```

Comandi linux per ispezionare la memoria

Vm_stat: info su gestione memoria

Vmmap <id processo>

provare vmmap eseguendo un
programma che dorme per 300 secondi

ps aux | grep sleep

prendo PID

pagesize

File sleep.c

```
#include <stdio.h>
#include <unistd.h>
int global_var = -40;
int main() {
    printf("Il programma inizia...\n");
    int a = 10;
    int b = 20;
    int c = a + b;
    // Dormi per 300 secondi
    sleep(300);

    printf("Il programma è stato svegliato dopo 10 secondi.\n");
    printf("Somma %d\n", c);

    return 0;
}
```

Java e bytecode (1)

- Java utilizza una macchina virtuale (Java Virtual Machine, JVM) per l'esecuzione del codice
- il codice sorgente viene prima compilato in bytecode anziché in codice macchina native
- Il bytecode Java viene eseguito dalla JVM. La JVM funge da intermediario tra il bytecode e l'hardware/sistema operativo, permettendo a Java di essere un linguaggio "scrivi una volta, esegui ovunque" (write once, run anywhere - **WORA**).
- La gestione del segmento di testo è fortemente influenzata dalla presenza della JVM e dal suo modo di eseguire il bytecode. Questo approccio fornisce vantaggi in termini di portabilità, sicurezza e ottimizzazione, ma introduce anche un livello di astrazione aggiuntivo rispetto all'esecuzione diretta del codice macchina nativo.

Java e bytecode (2)

- Quando il bytecode viene eseguito, la JVM lo carica nella sua area di memoria. Questa area funge da segmento di testo per il codice Java in esecuzione.
- Se il JIT (Just In Time compilazione) è attivo, parti del bytecode vengono compilate in codice macchina nativo. Questo codice compilato viene memorizzato in una cache JIT, che può essere considerata parte del segmento di testo, poiché contiene codice eseguibile.
- **Isolamento e Sicurezza:** La JVM fornisce un livello di isolamento tra il codice eseguibile e il sistema operativo. Questo aiuta a migliorare la sicurezza, poiché il codice Java non ha accesso diretto alla memoria del sistema o ad altre risorse a basso livello. Inoltre, la JVM può imporre restrizioni di sicurezza sul codice Java, come la sandbox in cui vengono eseguite alcune applicazioni Java.
- **Ottimizzazioni:** La JVM può eseguire varie ottimizzazioni durante l'esecuzione del bytecode, come l'inlining di metodi, l'ottimizzazione dei loop e altre trasformazioni del codice, che possono modificare dinamicamente il contenuto del segmento di testo (nella cache JIT).

Java e Memory Layout

```
public class Esempio{  
  
    // Variabile di istanza  
    int var = 10;  
  
    public static void main(String[] args) {  
  
        System.out.println("Il valore della  
variabile è: " + var);  
    }  
}
```

Compilation failed due to following error(s). Main.java:8: error:
non-static variable var cannot be referenced from a static
context

- In Java l'organizzazione dell'area di memoria di un processo è più complicata perché la JVM fornisce un livello di astrazione ulteriore da "smontare"
- In Java, una variabile di istanza (non statica) non può essere acceduta direttamente da un metodo statico, come il metodo main.

Java e Memory Layout

```
public class Esempio{  
  
    // Variabile di istanza  
    static int var = 10;  
  
    public static void main(String[] args) {  
  
        System.out.println("Il valore della  
variabile è: " + var);  
    }  
}
```

- O rendo la variabile statica e allora ...
- ...var è memorizzata in un'area di memoria dedicata alla gestione delle classi (Metaspace o PermGen, a seconda della versione della JVM), che è diversa dallo heap utilizzato per le istanze degli oggetti e dallo stack utilizzato per le variabili locali e i record di attivazione delle funzioni.

Java e Memory Layout

```
public class Esempio{  
  
    // Variabile di istanza  
    int var = 10;  
  
    public static void main(String[] args) {  
  
        Esempio myEs = new Esempio();  
  
        System.out.println("Il valore della  
variabile di istanza è: " + myEs.var);  
    }  
}
```

- O rendo la variabile una variabile di istanza ma...
- ... questa volta creo nel main l'oggetto per accedere alla variabile di istanza dell'oggetto appena creato

Le variabili di istanza in Java sono memorizzate nello heap. Quando crei un'istanza di un oggetto (ad esempio, con `new Esempio()`), la JVM alloca memoria nello heap per quell'oggetto, inclusa la memoria per le sue variabili di

Java Heap memory error

- La dimensione dell'heap di Java è controllata da parametri di configurazione della JVM, come -Xms per la dimensione iniziale dell'heap e -Xmx per la dimensione massima. Se il tuo programma tenta di allocare più memoria di quella consentita dal limite -Xmx, si verificherà un OutOfMemoryError.
- `java -Xmx100m HeapOverflow`
 - Questo comando esegue il programma con un massimo di 100 MB di memoria heap. La dimensione specifica per -Xmx può essere regolata in base alle esigenze e alle capacità del tuo ambiente di esecuzione.

```
import java.util.ArrayList;
import java.util.List;

public class HeapOverflow {
    public static void main(String[] args) {
        List<Object> objects = new ArrayList<>();

        try {
            while (true) {
                objects.add(new Object()); // Continua ad
                aggiungere oggetti all'elenco
            }
        } catch (OutOfMemoryError e) {
            System.out.println("Errore: memoria heap
esaurita!");
            e.printStackTrace();
        }
    }
}
```

Java Stack Memory Error

- In Java, un overflow dello stack si verifica tipicamente a causa di una ricorsione eccessiva, dove una funzione si chiama ripetutamente se stessa senza un caso base adeguato per terminare la ricorsione. Questo porta a un eccessivo utilizzo dello stack, poiché ogni chiamata di funzione aggiunge un frame allo stack fino a quando non si esaurisce lo spazio disponibile, causando un `StackOverflowError`.
- In questo esempio, la funzione `recursiveMethod` si chiama se stessa indefinitamente senza una condizione di terminazione. Ogni chiamata ricorsiva aggiunge un nuovo frame allo stack finché non si esaurisce lo spazio, causando un `StackOverflowError`.
- `java -Xss1m NomeDellaClasse`
 - la dimensione dello stack per ogni thread è impostata su 1 megabyte. Puoi regolare il valore 1m in base alle tue esigenze specifiche.

```
public class StackOverflowExample {  
    public static void recursiveMethod(int i) {  
        System.out.println("Chiamata numero: " + i);  
        recursiveMethod(i + 1); // Chiamata ricorsiva  
        // senza condizione di terminazione  
    }  
  
    public static void main(String[] args) {  
        try {  
            recursiveMethod(1);  
        } catch (StackOverflowError e) {  
            System.out.println("Errore: stack overflow!");  
            e.printStackTrace();  
        }  
    }  
}
```