3.4 – Chiamate a funzione e memory layout

Libro di testo:

- Capitoli 8.5, 8.6

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Agenda

- Meccanismo di chiamata a funzione
- Layout di memoria di un processo
- Gestione delle variabili in memoria

Chiamata a funzione

- Una chiamata a funzione causa la creazione di una struttura dati chiamata **RA – Record di Attivazione (Function Activation Record)**
- La RA contiene:
  - una copia dei parametri
  - variabili locali
- I RA sono impilati in una zona di memoria chiamata **stack**
  - Stack (pila): struttura LIFO

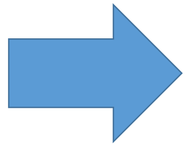
Cosa significa?

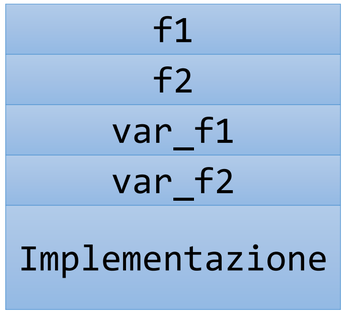
- La dimensione della RA dipende dal numero (e dal tipo) di variabili locali
  - Il tempo necessario per inserire il RA nello stack è costante

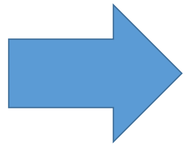
Record di Attivazione | Struttura dati

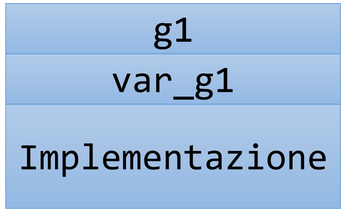
```
int f(int f1, int f2)
{
    int var_f1;
    int var_f2;
    // ...
}
```

```
int g(int g1)
{
    double var_g1;
    // ...
}
```









- Implementazione: informazioni che servono per ritornare al chiamante e fornire un risultato (`return`)
- Parametri e variabili locali sono equivalenti in questo schema
- Ogni chiamata a `f` o `g` ha il proprio RA

Meccanismo di chiamata a funzione

- Stack
- Chiamate in sequenza
  - Chiamata a f
    - f esce
  - Chiamata a g
    - g esce

```
int f(int f1, int f2)
{
    int var_f1;
    int var_f2;
    // ...
}

int g(int g1)
{
    double var_g1;
    // ...
}
```

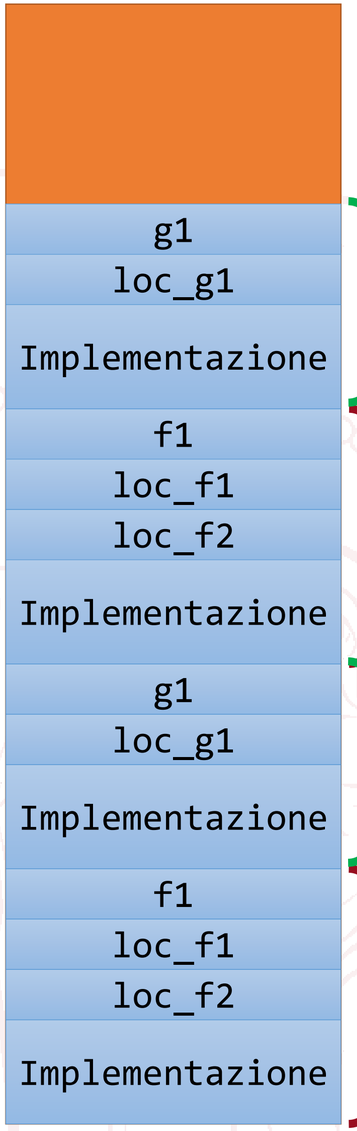
Meccanismo di chiamata a funzione

- Stack
- f chiama g e viceversa
  - Chiamata a f
    - Chiamata a g
      - Chiamata a f (2)
        - Chiamata a g (2)
        - g (2) esce
      - f (2) esce
    - g esce
  - f esce

Il fatto che ci siano RA diversi in chiamate a funzioni ricorrenti, per quale tipo di funzione è molto utile?

```
int f(int f1)
{
    int loc_f1;
    // ...
    int loc_f2 = g(f1);
    // ...
}
```

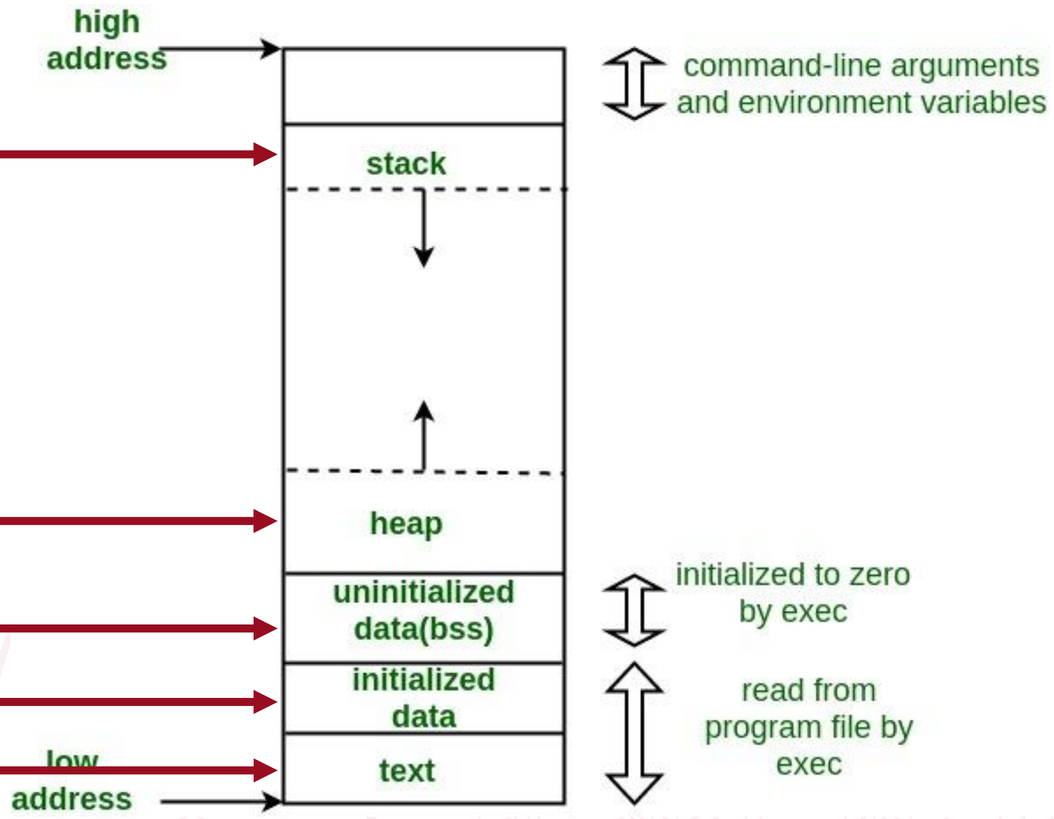
```
int g(int g1)
{
    double loc_g1;
    switch(g1)
    {
        case 0:
            loc_g1 = f(g1);
        case 1:
            loc_g1 = -1;
        default:
            loc_g1 = 0;
    }
    return loc_g1;
}
```



Rappresentazione in memoria di un programma

Layout di memoria

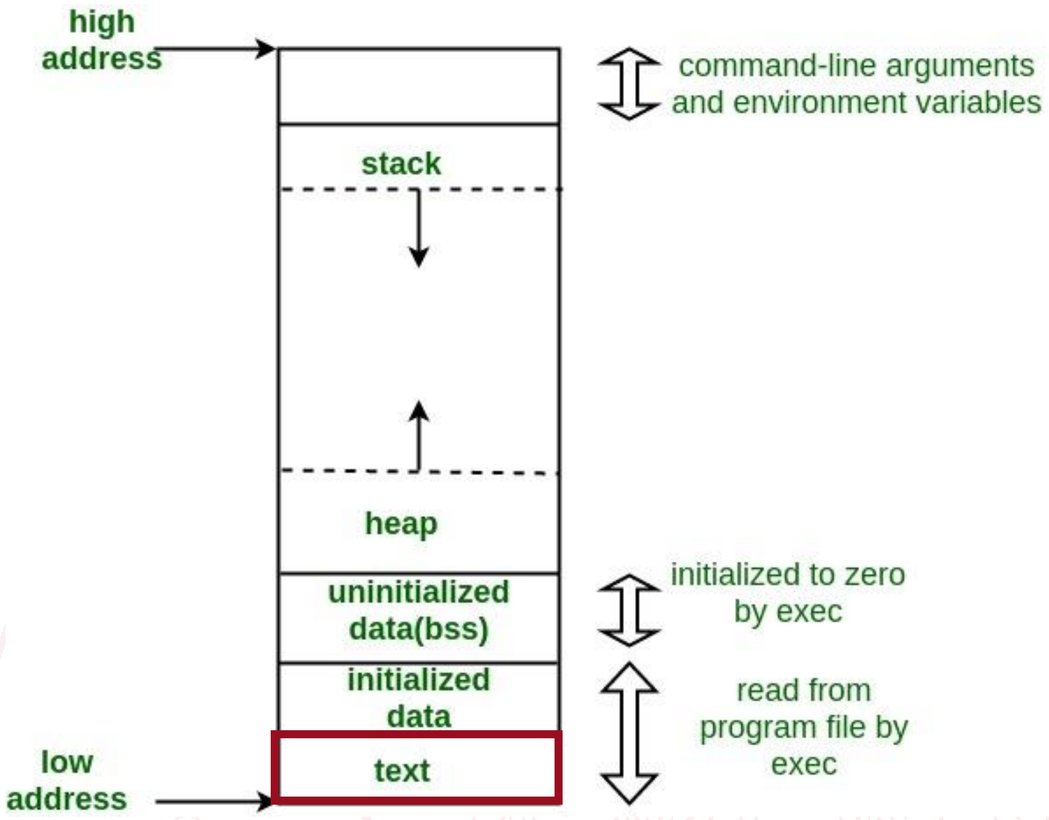
- Text segment
- Initialized data segment
- Uninitialized data segment
- Stack
- Heap



https://www.geeksforgeeks.org/memory-layout-of-c-program/

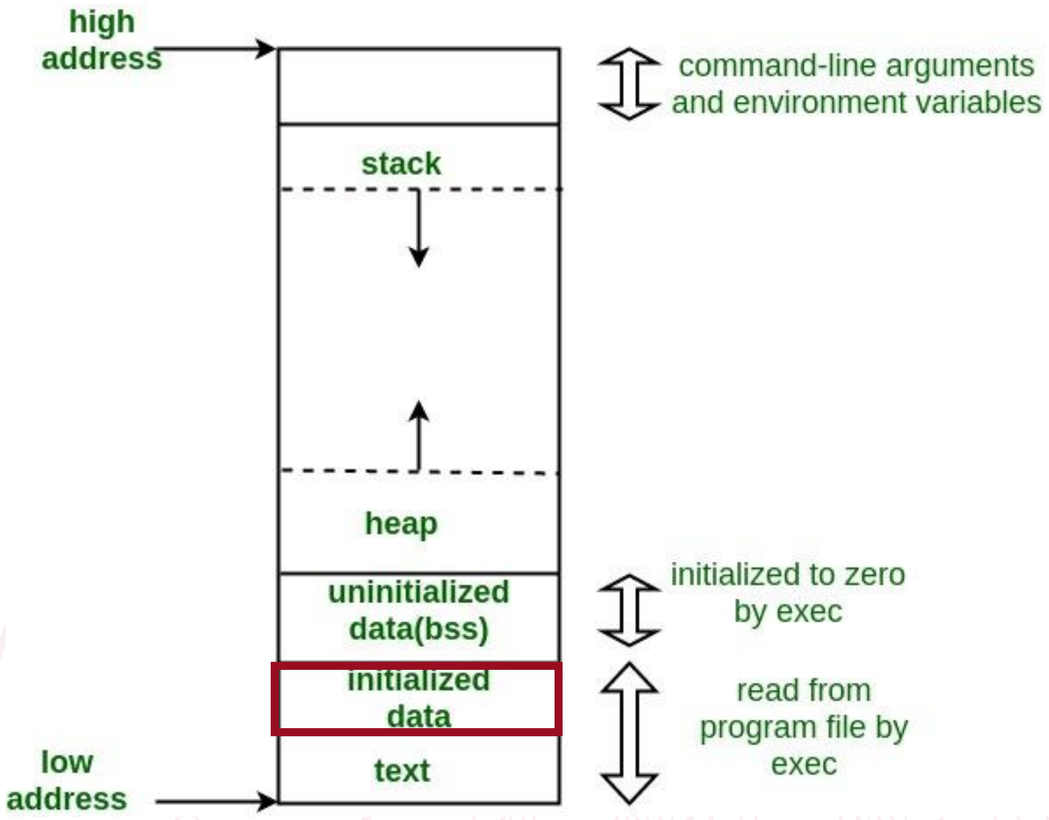
Layout di memoria | Text segment

- **Text segment**
  - AKA code segment / text
  - Copiato in memoria dal file oggetto
  - Contiene le istruzioni eseguibili
  - Spesso read-only
  - Spesso condiviso



Layout di memoria

- **Initialized data segment**
  - AKA data segment
  - Contiene variabili globali e variabili statiche
  - Read/write
    - Può contenere una parte read-only

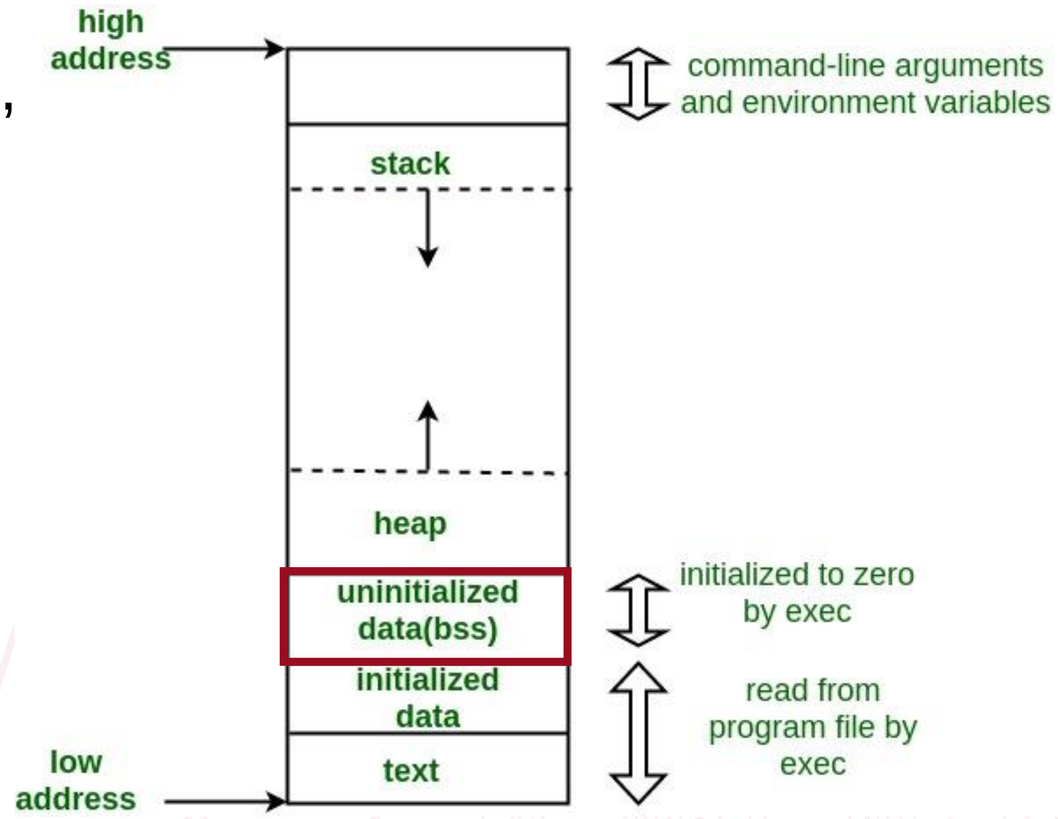



```
int glb_i = 10;                 // read-write
const int cnst_glb_i = 10;      // read-only

int main(void) {
    // …
    return 0;
}
```

Layout di memoria

- **Uninitialized data segment**
  - AKA BSS segment (block started by symbol, ragioni storiche)
  - Inizializzato a 0 dal SO
  - Contiene variabili globali e statiche non inizializzate esplicitamente

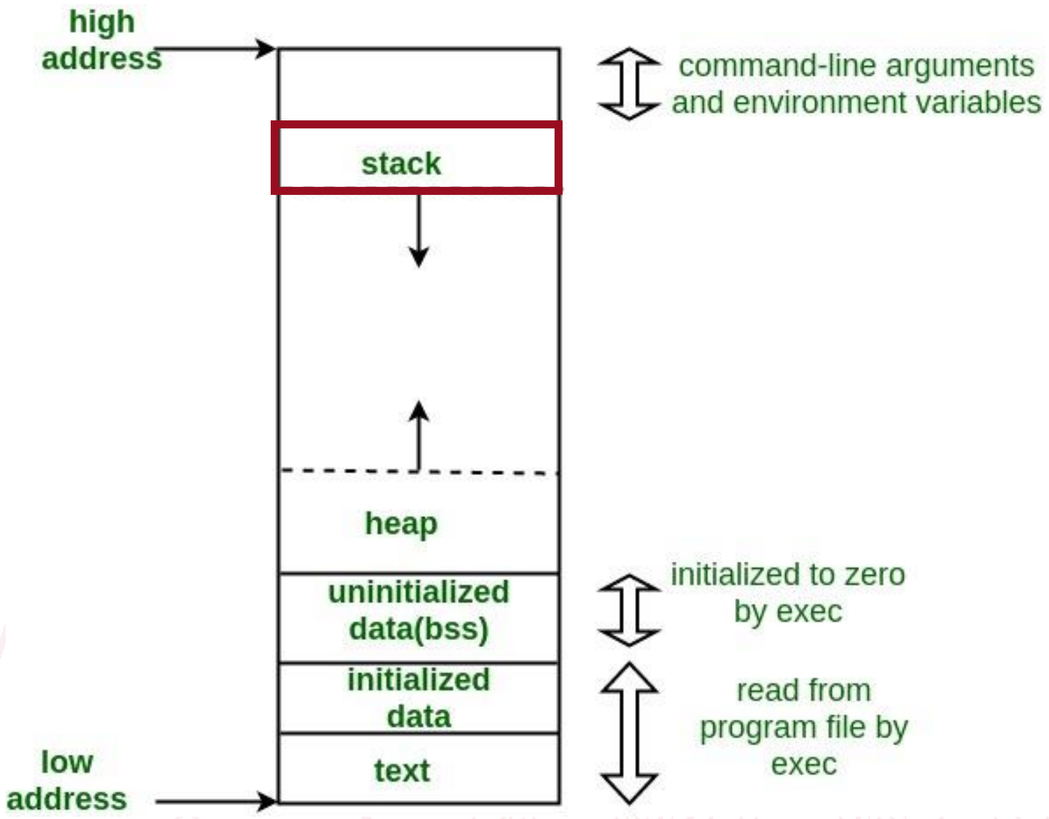



```
int glb_i;              // in BSS

int main(void) {
    // …
    return 0;
}
```

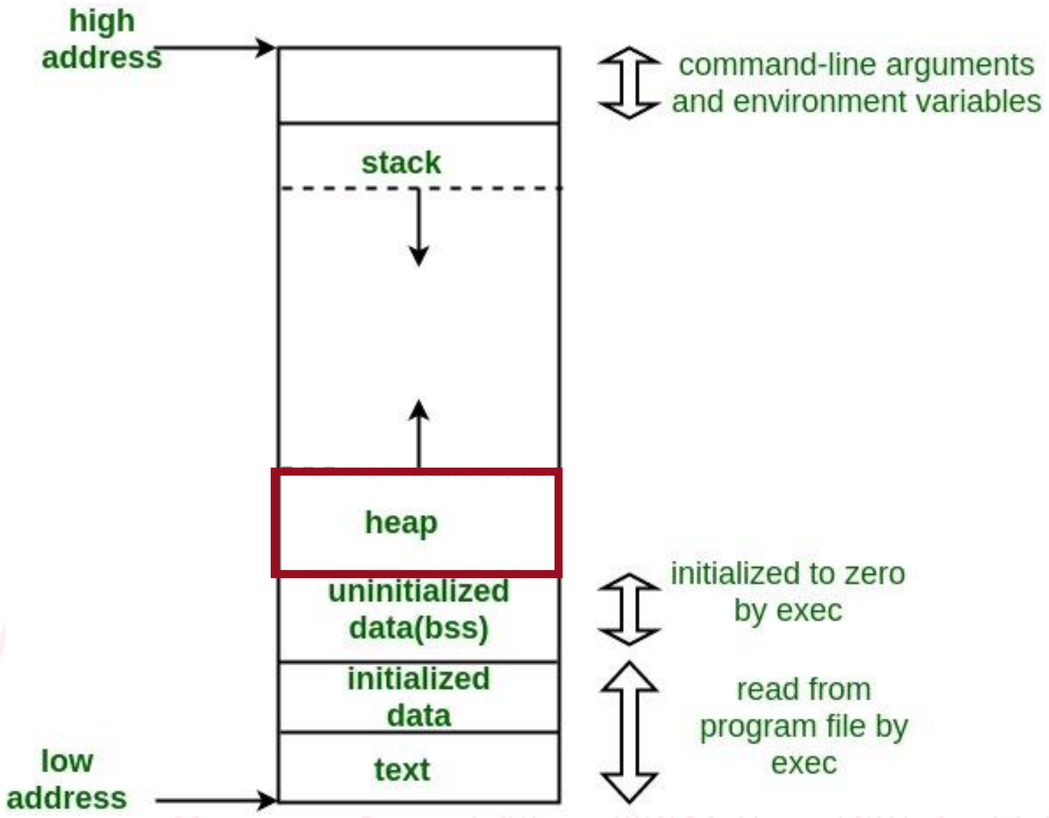
Layout di memoria

- **Stack**
  - Contiene i RA delle funzioni
    - Quindi le variabili locali e i parametri



Layout di memoria

- **Heap**
  - Formalmente è spesso visto come uno spazio che cresce in verso opposto allo stack
  - La sua dimensione può essere modificata con le funzioni di sistema `brk` e `sbrk`
  - Può essere gestito usando blocchi di memoria non contigui (funzione di sistema `mmap`)
  - Gestito tramite `new` e `delete` (allocazione dinamica della memoria)



Verifichiamo il layout della memoria

```
int main(void) {

    return 0;
}
```

```
ltonin@ltonin-laptop$ g++ -c main.cpp
ltonin@ltonin-laptop$ ls
main.cpp main.o
ltonin@ltonin-laptop$ size main.o
 text      data     bss     dec     hex     filename
 103          0       0     103      67     main.o
```

- Cosa succede se aggiungo una variabile globale non inizializzata?
- Cosa succede se aggiungo una variabile globale inizializzata?
- Cosa succede se aggiungo una variabile statica non inizializzata?
- Cosa succeed se aggiungo una variabile statica inizializzata?

Ordine di valutazione di un programma

Ordine di valutazione di un programma

- La valutazione di un programma corrisponde alla sua esecuzione
- Quando l'esecuzione raggiunge il codice che identifica la definizione di una variabile:
  1. La variabile viene «costruita»
  2. La memoria viene assegnata
  3. L'oggetto viene inizializzato
- Quando la variabile va out-of-scope
  - La memoria viene liberata e può essere utilizzata per altro

Valutazione delle variabili

- **Variabili globali**
  - Inizializzate prima della prima istruzione del `main`
  - Esistono fino al termine del programma

- **Variabili automatiche**
  - Di base, tutte le variabili locali sono automatiche
  - Esistono solo all'interno dello scope
  - Vengono ricreate ogni volta (es., se all'interno di una funzione)

- **Variabili statiche (`static`)**
  - Inizializzate solo la prima volta
  - Mantengono il valore anche al di fuori del loro scope
  - Possono essere globali o locali (se dichiarate dentro a una funzione)

Valutazione delle variabili

1. Creazione variabili globali: **`program_name`**, **`v`**
   [vivranno fino al termine del programma]
2. **`s`** viene costruita
   [vivrà fino all'uscita della funzione `f()`]
3. **`s`** viene inizializzata con una stringa vuota
4. Ad ogni iterazione del while-loop, **`stripped`** e **`not_letters`** sono costruite e inizializzate
   [nell'ordine del codice]
   [vivranno fino al termine del blocco]
   [vengono distrutte in ordine inverso]
5. …
6. …

```
std::string program_name = "silly";
std::vector<std::string> v;

void f()
{
    std::string s;                                  // s locale a f
    while (std::cin >> s && s != "quit") {
        std::string stripped;                       // locale nel loop
        std::string not_letters;                    // costruite e
        for (int i = 0; i < s.size(); ++i)          // rimosse ad ogni
            if (isalpha(s[i]))                      // iterazione
                stripped += s[i];
            else
                not_letters += s[i];
        v.push_back(stripped);                      // Aggiunge un
                                                    // valore in coda
                                                    // al vettore
        // ...
    }
    // ...
}
```

Variabili statiche vs. automatiche

```
#include <ostream>

void f(void) {
    int local_auto = 0;
    static int local_static = 0;

    std::cout << "Auto: " << local_auto++ << ", static: "
              << local_static++ << '\n’;
}

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

Qual è l’output di questa funzione?

Variabili statiche vs. automatiche

```
#include <ostream>

void f(void) {
    int local_auto = 0;
    static int local_static = 0;

    std::cout << "Auto: " << local_auto++ << ", static: "
              << local_static++ << '\n’;
}

int main(void) {
    f();                  // "Auto: 1, static: 1"
    f();
    f();
    return 0;
}
```

Variabili statiche vs. automatiche

```
#include <ostream>

void f(void) {
    int local_auto = 0;
    static int local_static = 0;

    std::cout << "Auto: " << local_auto++ << ", static: "
              << local_static++ << '\n’;
}

int main(void) {
    f();                // "Auto: 1, static: 1"
    f();                // "Auto: 1, static: 2"
    f();
    return 0;
}
```

Variabili statiche vs. automatiche

```
#include <ostream>

void f(void) {
    int local_auto = 0;
    static int local_static = 0;

    std::cout << "Auto: " << local_auto++ << ", static: "
              << local_static++ << '\n’;
}

int main(void) {
    f();                // "Auto: 1, static: 1"
    f();                // "Auto: 1, static: 2"
    f();                // "Auto: 1, static: 3"
    return 0;
}
```

Valutazione delle variabili

- I compilatori fanno sì che il codice si comporti come se le azioni descritte prima fossero eseguite
- Varie ottimizzazioni sono possibili
- I compilatori spesso modificano l'implementazione per ottimizzare
  - Ma il comportamento è immutato

Inizializzazione globale

- Le variabili globali sono inizializzate prima dell'esecuzione della prima istruzione del `main`
  - Se le variabili di diverse *translation unit* hanno una dipendenza, non sappiamo in che ordine sono effettuate le inizializzazioni
  - Variabili globali hanno la memoria scritta a 0 prima delle inizializzazioni

Perché?
(hint: layout memoria)

Inizializzazione globale

f1.cpp

```
// file f1.cpp – variabili globali
int x1 = 1;
int y1 = x1 + 2;
```

f2.cpp

```
// file f2.cpp – variabili globali
extern int y1;
int y2 = y1 + 2;
```

main.cpp

```
#include <ostream>
#include "f1.cpp"
#include "f2.cpp"

int main(void) {

    std::cout<< x1 << " – " << y1 << " - "<< y2 << "\n";

    return 0;
}
```

Da evitare!!!

Recap

- Record di attivazione, ruolo nella chiamata a funzione
- Layout della memoria
  - Varie regioni, ciascuna con la sua caratteristica
- Variabili
  - Locali vs globali
  - Statiche vs automatiche
- Inizializzazione globale