Programmazione B
Ingegneria e Scienze Informatiche - Cesena
A.A. 2021-2022

Spazio di indirizzamento virtuale

Catia Prandi - catia.prandi2@unibo.it

Credit: Pietro Di Lena

```c
// The function name says it all
int stack_overflow(){
  return stack_overflow();
}
```

- Le funzioni sono uno strumento molto importante dei linguaggi di programmazione,
- L'uso di funzioni nei linguaggi di alto livello è talmente diffuso da richiederne una gestione estremamente efficiente, allo scopo di mantenere bassi i tempi di esecuzione dei programmi.
- Molte operazioni per la gestione della chiamata a funzione sono infatti supportate direttamente dall'hardware del calcolatore.

- Le funzioni sono uno strumento molto importante dei linguaggi di programmazione,
- L'uso di funzioni nei linguaggi di alto livello è talmente diffuso da richiederne una gestione estremamente efficiente, allo scopo di mantenere bassi i tempi di esecuzione dei programmi.
- Molte operazioni per la gestione della chiamata a funzione sono infatti supportate direttamente dall'hardware del calcolatore.
- E' importante conoscere le caratteristiche di basso livello della chiamata a funzione per poter essere consapevoli di quali siano le accortezze da tenere in considerazione:
  - per implementare programmi efficienti in termini di tempo di calcolo;
  - per evitare di creare falle di sicurezza sul calcolatore.

Spazio di indirizzamento virtuale

- Lo **spazio di indirizzamento virtuale** (in inglese, **virtual address space**) è lo spazio di memoria che il sistema operativo mette a disposizione di un processo (programma in esecuzione).

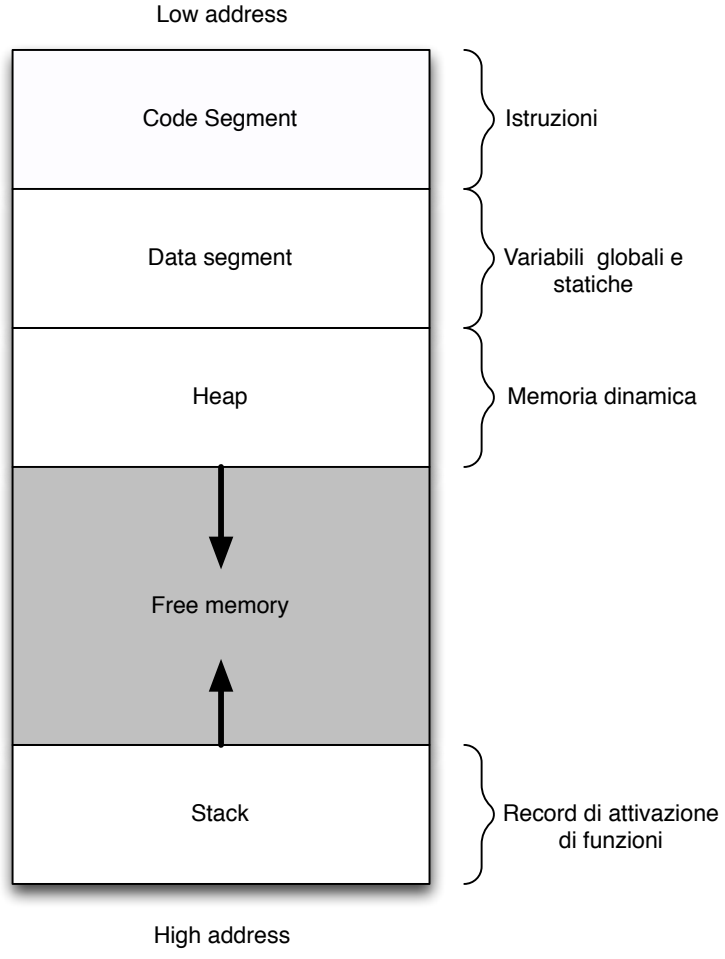
Spazio di indirizzamento virtuale

- Lo **spazio di indirizzamento virtuale** (in inglese, **virtual address space**) è lo spazio di memoria che il sistema operativo mette a disposizione di un processo (programma in esecuzione).
- Parliamo di *indirizzamento virtuale* poiché il sistema operativo *maschera* al processo gli effettivi indirizzi di memoria fisica sulla RAM.
  - I processi operano in uno spazio di indirizzi *virtuale*: la loro visibilità è limitata agli indirizzi di memoria che il sistema operativo fornisce al programma.
  - In questo modo, un programma in esecuzione non è a conoscenza delle effettive risorse e opera come se avesse a disposizione tutta la memoria disponibile sul calcolatore.
  - Il sistema operativo e la CPU si occupano di mappare gli indirizzi *virtuali* in indirizzi *fisici* in modo *trasparente* al programma in esecuzione.

Spazio di indirizzamento virtuale

- Lo **spazio di indirizzamento virtuale** (in inglese, **virtual address space**) è lo spazio di memoria che il sistema operativo mette a disposizione di un processo (programma in esecuzione).
- Parliamo di *indirizzamento virtuale* poiché il sistema operativo *maschera* al processo gli effettivi indirizzi di memoria fisica sulla RAM.
  - I processi operano in uno spazio di indirizzi *virtuale*: la loro visibilità è limitata agli indirizzi di memoria che il sistema operativo fornisce al programma.
  - In questo modo, un programma in esecuzione non è a conoscenza delle effettive risorse e opera come se avesse a disposizione tutta la memoria disponibile sul calcolatore.
  - Il sistema operativo e la CPU si occupano di mappare gli indirizzi *virtuali* in indirizzi *fisici* in modo *trasparente* al programma in esecuzione.
- Lo spazio di indirizzamento contiene tutte le informazioni necessarie all'esecuzione del processo, come
  - il codice eseguibile;
  - i dati;
  - l'area riservata per l'allocazione dinamica della memoria;
  - l'area riservata per l'esecuzione delle funzioni.

Spazio di indirizzamento virtuale: esempio di organizzazione



Struttura semplificata dello spazio di indirizzamento virtuale.

Spazio di indirizzamento virtuale: code segment

- Il **code segment**, chiamato anche **text segment**, contiene le istruzioni eseguibili del programma.
- Quando il compilatore genera un file oggetto, una parte di tale file contiene il set di istruzioni eseguibili del programma.
- Al momento dell'esecuzione, il set di istruzioni del programma è caricato nella sezione code segment dello spazio di indirizzamento virtuale assegnato al processo.
- Il code segment è *read-only* e ha una dimensione fissa, definita al momento della compilazione.
- Ogni istruzione nel code segment corrisponde ad un preciso indirizzo di memoria virtuale.

Spazio di indirizzamento virtuale: data segment

- Il **data segment** è una sezione dedicata alle variabili globali e variabili locali *statiche* (`static`).
- Contiene una sezione, generalmente chiamata **BSS** (Block Started by Symbol), dedicata alle *variabili non inizializzate*.
  - Il file oggetto contiene unicamente la dimensione della sezione BSS (non è necessario mantenere informazioni sul valore iniziale delle variabili).
  - A run-time viene allocata memoria sufficiente a contenere il BSS nel data segment.
- Contiene una sezione *read-only* dedicata alle variabili il cui contenuto non può essere modificato durante l'esecuzione del programma.
- Come per il code segment, la dimensione del data segment è fissa.
- Tranne che per la sezione dedicata alle variabili read-only, il contenuto del code segment può essere modificato a run-time.

Spazio di indirizzamento virtuale: heap

- La sezione **heap** è utilizzata per la gestione dinamica della memoria (non ancora vista).
- Può essere visto come un *magazzino*, che viene utilizzato per *immagazzinare* dati temporanei durante l'esecuzione del programma.
- A differenza della variabili globali e statiche, la cui dimensione totale è nota al momento della compilazione, la dimensione della memoria richiesta dinamicamente è nota solo durante l'esecuzione del programma.
- La sezione heap è generalmente posizionata dopo il data segment e *cresce* verso indirizzi di memoria più alti.

Spazio di indirizzamento virtuale: stack

- La sezione **stack** (o **call stack**, *stack delle invocazioni a funzione*) è utilizzata per mantenere informazioni relative alle chiamate di funzione durante l'esecuzione del programma.
- E' generalmente posizionata a partire dagli indirizzi di memoria più alti nello spazio di indirizzamento virtuale e *cresce* verso il basso, in direzione dell'heap.

Spazio di indirizzamento virtuale: stack

- La sezione **stack** (o **call stack**, *stack delle invocazioni a funzione*) è utilizzata per mantenere informazioni relative alle chiamate di funzione durante l'esecuzione del programma.
- E' generalmente posizionata a partire dagli indirizzi di memoria più alti nello spazio di indirizzamento virtuale e *cresce* verso il basso, in direzione dell'heap.
- Una chiamata a funzione provoca l'allocazione sullo stack di un **record di attivazione** relativo alla funzione chiamata.
- Il record di attivazione di una funzione contiene informazioni quali:
  - indirizzo di ritorno (l'indirizzo dell'istruzione da eseguire non appena la funzione termina l'esecuzione);
  - i parametri attuali passati alla funzione;
  - variabili locali (o automatiche) della funzione.

Spazio di indirizzamento virtuale: stack

- La sezione **stack** (o **call stack**, *stack delle invocazioni a funzione*) è utilizzata per mantenere informazioni relative alle chiamate di funzione durante l'esecuzione del programma.
- E' generalmente posizionata a partire dagli indirizzi di memoria più alti nello spazio di indirizzamento virtuale e *cresce* verso il basso, in direzione dell'heap.
- Una chiamata a funzione provoca l'allocazione sullo stack di un **record di attivazione** relativo alla funzione chiamata.
- Il record di attivazione di una funzione contiene informazioni quali:
  - indirizzo di ritorno (l'indirizzo dell'istruzione da eseguire non appena la funzione termina l'esecuzione);
  - i parametri attuali passati alla funzione;
  - variabili locali (o automatiche) della funzione.
- Lo stack viene gestito tramite una modalità **LIFO** (Last In First Out):
  1. una chiamata a funzione provoca l'allocazione del relativo record di attivazione, che viene posizionato in cima al record di attivazione più recente sullo stack;
  2. quando l'esecuzione della funzione termina, il relativo record di attivazione viene *liberato* e viene *riattivato* il record di attivazione immediatamente precedente;
  3. l'esecuzione del programma procede con le istruzioni della funzione associata al record di attivazione attualmente in cima allo stack.

Record di attivazione

- Il *record di attivazione* (o anche **activation record**, **stack frame**) contiene le informazioni necessarie all'esecuzione della funzione e necessarie a ripristinare lo stato del processo al termine dell'esecuzione della funzione.

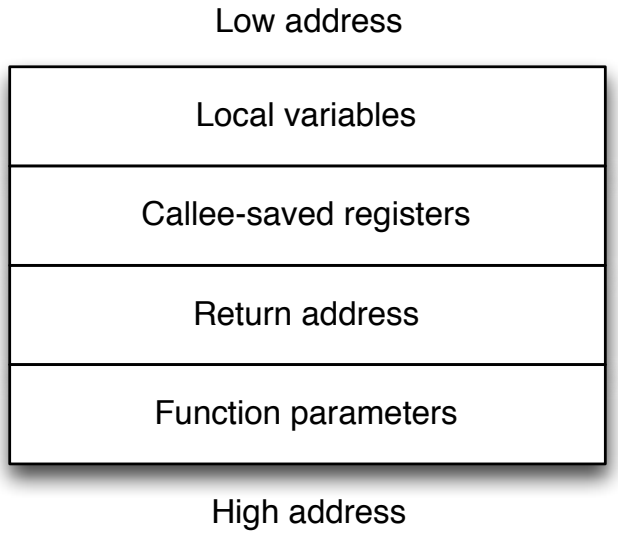
Record di attivazione

- Il *record di attivazione* (o anche **activation record**, **stack frame**) contiene le informazioni necessarie all'esecuzione della funzione e necessarie a ripristinare lo stato del processo al termine dell'esecuzione della funzione.
- La composizione del record di attivazione dipende da diversi fattori, tra cui:
  - architettura del calcolatore,
  - linguaggio di programmazione,
  - versione del compilatore.

Record di attivazione

- Il *record di attivazione* (o anche **activation record**, **stack frame**) contiene le informazioni necessarie all'esecuzione della funzione e necessarie a ripristinare lo stato del processo al termine dell'esecuzione della funzione.
- La composizione del record di attivazione dipende da diversi fattori, tra cui:
  - architettura del calcolatore,
  - linguaggio di programmazione,
  - versione del compilatore.
- Le direttive che definiscono la composizione del record di attivazione e le modalità di attivazione sul call stack sono indicate col nome di **calling conventions**.
- Esistono numerosissime **calling conventions** differenti, che fissano questioni quali:
  - Come appaiono lo stack e i registri della CPU quando avviene una chiamata di funzione?
  - Come appaiono lo stack e i registri della CPU quando la chiamata termina?
  - Dov'è salvato il valore di ritorno della funzione?
- La *cdecl calling convention* è quella comunemente associata al linguaggio C.

Record di attivazione: esempio di organizzazione



Struttura (enormemente) semplificata del record di attivazione.

Record di attivazione: calling conventions

- Indipendentemente dalla convenzione adottata, una chiamata a funzione comporta approssimativamente le seguenti operazioni:
  1. il chiamante, **caller**, posiziona sullo stack o in specifici registri della CPU gli argomenti da passare alla funzione chiamata, **callee**;

Record di attivazione: calling conventions

- Indipendentemente dalla convenzione adottata, una chiamata a funzione comporta approssimativamente le seguenti operazioni:
  1. il chiamante, **caller**, posiziona sullo stack o in specifici registri della CPU gli argomenti da passare alla funzione chiamata, **callee**;
  2. il caller salva l'indirizzo di ritorno sullo stack, i.e. l'indirizzo dell'istruzione da eseguire al termine dell'esecuzione del callee;

Record di attivazione: calling conventions

- Indipendentemente dalla convenzione adottata, una chiamata a funzione comporta approssimativamente le seguenti operazioni:
  1. il chiamante, **caller**, posiziona sullo stack o in specifici registri della CPU gli argomenti da passare alla funzione chiamata, **callee**;
  2. il caller salva l'indirizzo di ritorno sullo stack, i.e. l'indirizzo dell'istruzione da eseguire al termine dell'esecuzione del callee;
  3. il caller passa il controllo al callee;

Record di attivazione: calling conventions

- Indipendentemente dalla convenzione adottata, una chiamata a funzione comporta approssimativamente le seguenti operazioni:
  1. il chiamante, **caller**, posiziona sullo stack o in specifici registri della CPU gli argomenti da passare alla funzione chiamata, **callee**;
  2. il caller salva l'indirizzo di ritorno sullo stack, i.e. l'indirizzo dell'istruzione da eseguire al termine dell'esecuzione del callee;
  3. il caller passa il controllo al callee;
  4. il callee salva sullo stack le variabili locali;

Record di attivazione: calling conventions

- Indipendentemente dalla convenzione adottata, una chiamata a funzione comporta approssimativamente le seguenti operazioni:
  1. il chiamante, **caller**, posiziona sullo stack o in specifici registri della CPU gli argomenti da passare alla funzione chiamata, **callee**;
  2. il caller salva l'indirizzo di ritorno sullo stack, i.e. l'indirizzo dell'istruzione da eseguire al termine dell'esecuzione del callee;
  3. il caller passa il controllo al callee;
  4. il callee salva sullo stack le variabili locali;
  5. il callee salva sullo stack l'attuale contenuto di alcuni registri della CPU, in modo da poterli ripristinare al termine della chiamata;

Record di attivazione: calling conventions

- Indipendentemente dalla convenzione adottata, una chiamata a funzione comporta approssimativamente le seguenti operazioni:
  1. il chiamante, **caller**, posiziona sullo stack o in specifici registri della CPU gli argomenti da passare alla funzione chiamata, **callee**;
  2. il caller salva l'indirizzo di ritorno sullo stack, i.e. l'indirizzo dell'istruzione da eseguire al termine dell'esecuzione del callee;
  3. il caller passa il controllo al callee;
  4. il callee salva sullo stack le variabili locali;
  5. il callee salva sullo stack l'attuale contenuto di alcuni registri della CPU, in modo da poterli ripristinare al termine della chiamata;
  6. il callee esegue le proprie istruzioni.

Record di attivazione: calling conventions

- Indipendentemente dalla convenzione adottata, una chiamata a funzione comporta approssimativamente le seguenti operazioni:
  1. il chiamante, **caller**, posiziona sullo stack o in specifici registri della CPU gli argomenti da passare alla funzione chiamata, **callee**;
  2. il caller salva l'indirizzo di ritorno sullo stack, i.e. l'indirizzo dell'istruzione da eseguire al termine dell'esecuzione del callee;
  3. il caller passa il controllo al callee;
  4. il callee salva sullo stack le variabili locali;
  5. il callee salva sullo stack l'attuale contenuto di alcuni registri della CPU, in modo da poterli ripristinare al termine della chiamata;
  6. il callee esegue le proprie istruzioni.
- Al termine delle esecuzione:
  1. il callee posiziona il valore di ritorno in specifici registri del processore o in altre locazioni di memoria, note al caller.

Record di attivazione: calling conventions

- Indipendentemente dalla convenzione adottata, una chiamata a funzione comporta approssimativamente le seguenti operazioni:
  1. il chiamante, **caller**, posiziona sullo stack o in specifici registri della CPU gli argomenti da passare alla funzione chiamata, **callee**;
  2. il caller salva l'indirizzo di ritorno sullo stack, i.e. l'indirizzo dell'istruzione da eseguire al termine dell'esecuzione del callee;
  3. il caller passa il controllo al callee;
  4. il callee salva sullo stack le variabili locali;
  5. il callee salva sullo stack l'attuale contenuto di alcuni registri della CPU, in modo da poterli ripristinare al termine della chiamata;
  6. il callee esegue le proprie istruzioni.
- Al termine delle esecuzione:
  1. il callee posiziona il valore di ritorno in specifici registri del processore o in altre locazioni di memoria, note al caller.
  2. il callee ripristina lo stato dei registri salvati sullo stack;

Record di attivazione: calling conventions

- Indipendentemente dalla convenzione adottata, una chiamata a funzione comporta approssimativamente le seguenti operazioni:
  1. il chiamante, **caller**, posiziona sullo stack o in specifici registri della CPU gli argomenti da passare alla funzione chiamata, **callee**;
  2. il caller salva l'indirizzo di ritorno sullo stack, i.e. l'indirizzo dell'istruzione da eseguire al termine dell'esecuzione del callee;
  3. il caller passa il controllo al callee;
  4. il callee salva sullo stack le variabili locali;
  5. il callee salva sullo stack l'attuale contenuto di alcuni registri della CPU, in modo da poterli ripristinare al termine della chiamata;
  6. il callee esegue le proprie istruzioni.
- Al termine delle esecuzione:
  1. il callee posiziona il valore di ritorno in specifici registri del processore o in altre locazioni di memoria, note al caller.
  2. il callee ripristina lo stato dei registri salvati sullo stack;
  3. gli argomenti sono rimossi dallo stack e viene aggiornato lo stack pointer (questo compito è assegnato al caller o al callee, a seconda della convenzione adottata);

Record di attivazione: calling conventions

- Indipendentemente dalla convenzione adottata, una chiamata a funzione comporta approssimativamente le seguenti operazioni:
  1. il chiamante, **caller**, posiziona sullo stack o in specifici registri della CPU gli argomenti da passare alla funzione chiamata, **callee**;
  2. il caller salva l'indirizzo di ritorno sullo stack, i.e. l'indirizzo dell'istruzione da eseguire al termine dell'esecuzione del callee;
  3. il caller passa il controllo al callee;
  4. il callee salva sullo stack le variabili locali;
  5. il callee salva sullo stack l'attuale contenuto di alcuni registri della CPU, in modo da poterli ripristinare al termine della chiamata;
  6. il callee esegue le proprie istruzioni.
- Al termine delle esecuzione:
  1. il callee posiziona il valore di ritorno in specifici registri del processore o in altre locazioni di memoria, note al caller.
  2. il callee ripristina lo stato dei registri salvati sullo stack;
  3. gli argomenti sono rimossi dallo stack e viene aggiornato lo stack pointer (questo compito è assegnato al caller o al callee, a seconda della convenzione adottata);
  4. il controllo passa al caller.

Alcune caratteristiche del cdecl calling conventions

- Gli argomenti della funzione sono passati da destra verso sinistra:
  - con questa convenzione, il primo argomento della funzione è in cima allo stack;
  - permette la gestione di *funzioni variadiche*, che hanno un numero variabile di argomenti (ad esempio, la `printf()`).

Alcune caratteristiche del cdecl calling conventions

- Gli argomenti della funzione sono passati da destra verso sinistra:
  - con questa convenzione, il primo argomento della funzione è in cima allo stack;
  - permette la gestione di *funzioni variadiche*, che hanno un numero variabile di argomenti (ad esempio, la `printf()`).
- La funzione chiamante si occupa di ripristinare lo stack dopo la chiamata:
  - questa convenzione aumenta la dimensione dell'eseguibile, dato che il codice per ripristinare lo stack deve essere replicato per ogni chiamata a funzione e non per ogni funzione;
  - permette di gestire la chiamata a funzioni variadiche, in quanto per questi tipi di funzioni solo il caller è a conoscenza del numero di argomenti da passare alla funzione mentre il callee potrebbe non esserne a conoscenza.

Problemi legati alla gestione dello stack

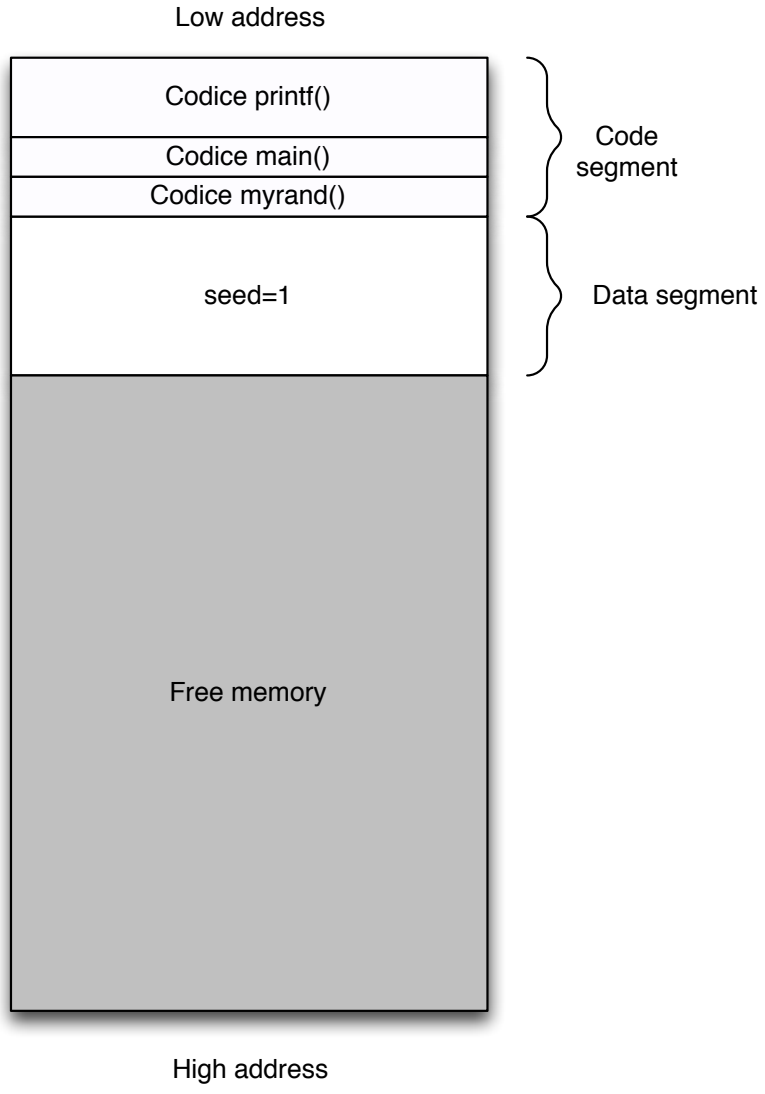
- **Stack overflow**
  - La dimensione della memoria condivisa per il segmento heap e il segmento stack dello spazio di indirizzamento è fissata all'avvio del programma.
  - Quando si verifica una richiesta troppo elevata di memoria nello stack, avviene uno **stack overflow**, che tipicamente causa il crash del programma.
  - Le cause più comuni dello stack overflow sono:
    1. una serie troppo *profonda* o infinita di chiamate ricorsive,
    2. chiamate a funzioni che fanno uso di parametri o variabili *ingombranti*.

Problemi legati alla gestione dello stack

- **Stack overflow**
  - La dimensione della memoria condivisa per il segmento heap e il segmento stack dello spazio di indirizzamento è fissata all'avvio del programma.
  - Quando si verifica una richiesta troppo elevata di memoria nello stack, avviene uno **stack overflow**, che tipicamente causa il crash del programma.
  - Le cause più comuni dello stack overflow sono:
    1. una serie troppo *profonda* o infinita di chiamate ricorsive,
    2. chiamate a funzioni che fanno uso di parametri o variabili *ingombranti*.
- **Stack buffer overflow**
  - Uno **stack buffer overflow** avviene quando un programma scrive dati sul record di attivazione, al di fuori dello spazio riservato per tali dati.
  - Nel linguaggio C sono principalmente causati da *distrazioni* da parte del programmatore e/o dall'uso di funzioni (di libreria) che non eseguono i dovuti controlli di sicurezza sulla dimensione dei dati.
  - Queste falle nelle funzioni del linguaggio C sono state e sono spesso sfruttate per eseguire *attacchi* di tipo informatico.
  - Una tecnica molto comune è, ad esempio, quella di sovrascrivere l'indirizzo di ritorno nel record di attivazione in modo da attivare l'esecuzione di codice malevolo sulla macchina.

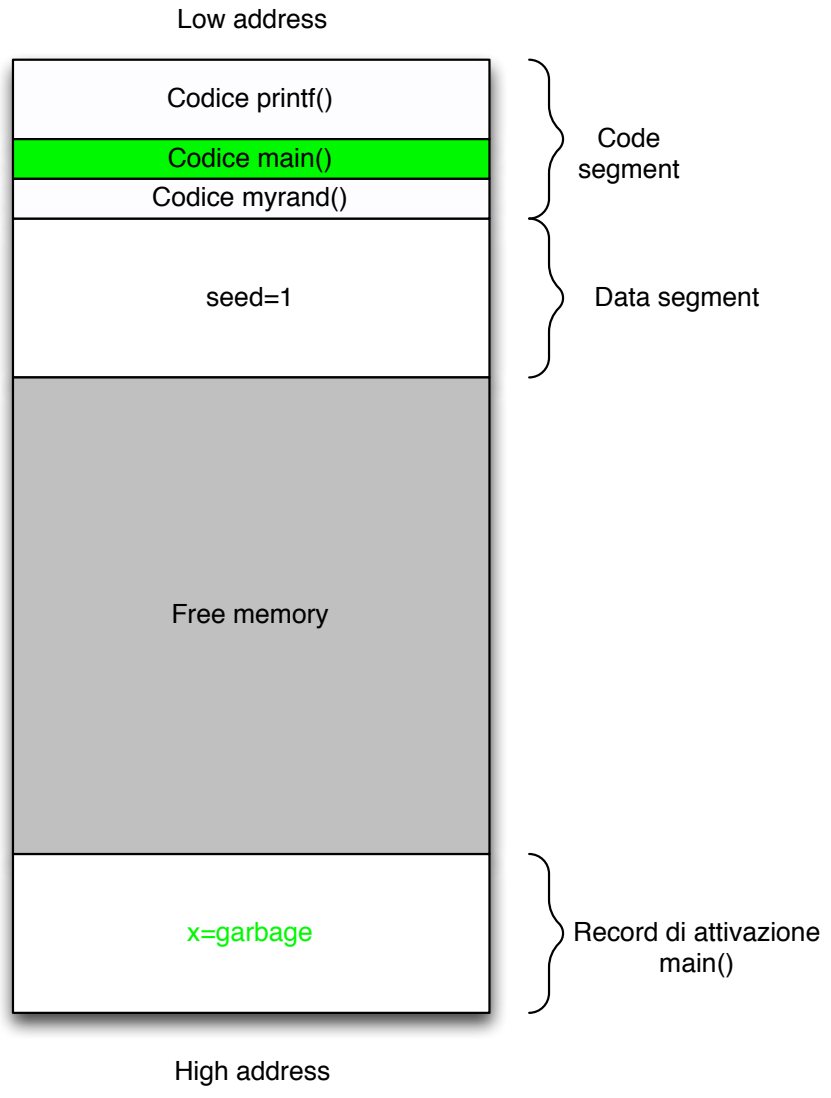
Esempio di chiamata a funzione

```
1  #include <stdio.h>
2
3  int myrand(void) {
4    static int seed = 1;
5    return seed = (5* seed +3)%16;
6  }
7
8  int main() {
9    int x;
10
11   x = myrand();
12   printf("%d\n",x);
13   x = myrand();
14   printf("%d\n",x);
15
16   return 0;
17 }
```



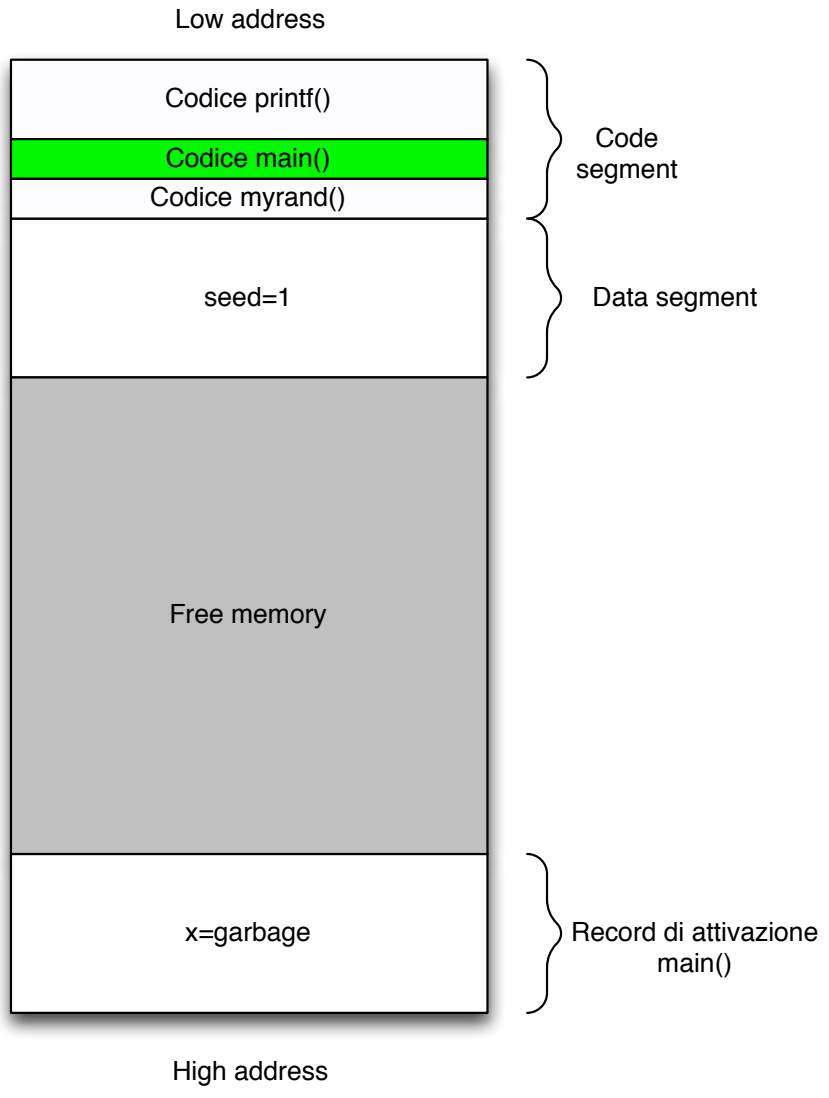
Esempio di chiamata a funzione

```
1  #include <stdio.h>
2
3  int myrand(void) {
4    static int seed = 1;
5    return seed = (5* seed +3)%16;
6  }
7
8  int main() {
9    int x;
10
11   x = myrand();
12   printf("%d\n",x);
13   x = myrand();
14   printf("%d\n",x);
15
16   return 0;
17 }
```



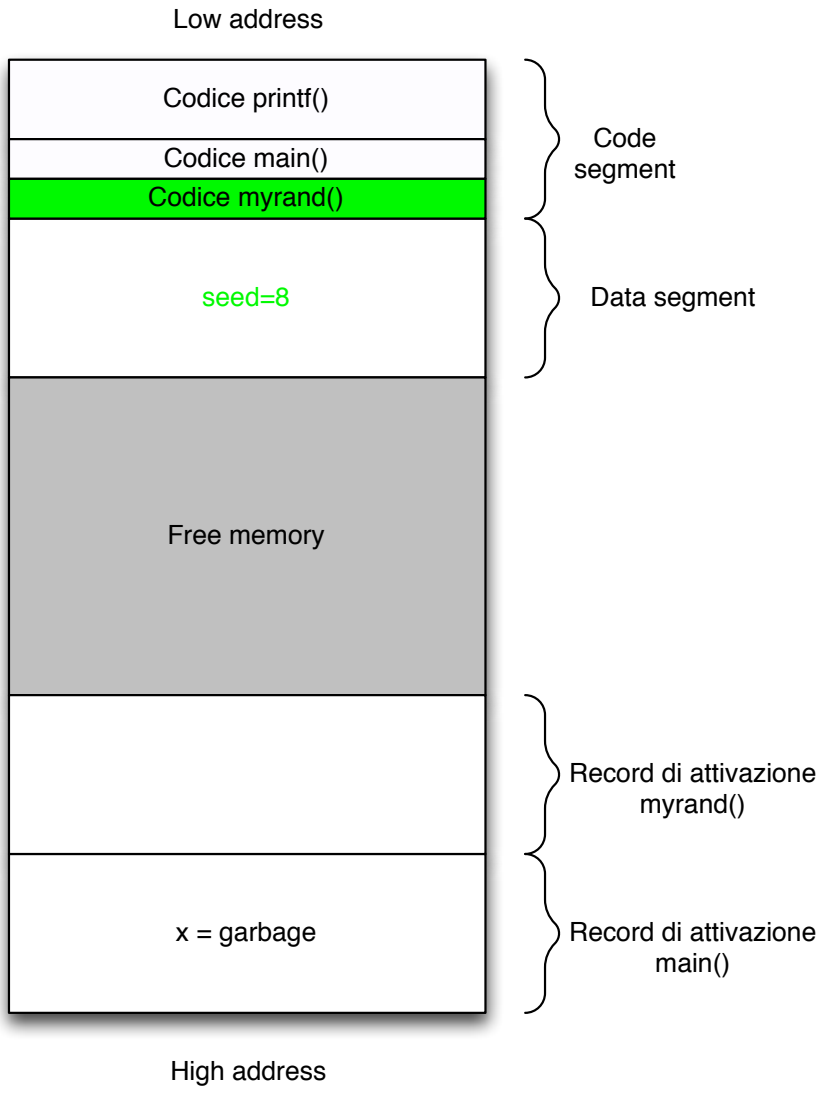
Esempio di chiamata a funzione

```
1  #include <stdio.h>
2
3  int myrand(void) {
4    static int seed = 1;
5    return seed = (5* seed +3)%16;
6  }
7
8  int main() {
9    int x;
10
11   x = myrand();
12   printf("%d\n",x);
13   x = myrand();
14   printf("%d\n",x);
15
16   return 0;
17 }
```



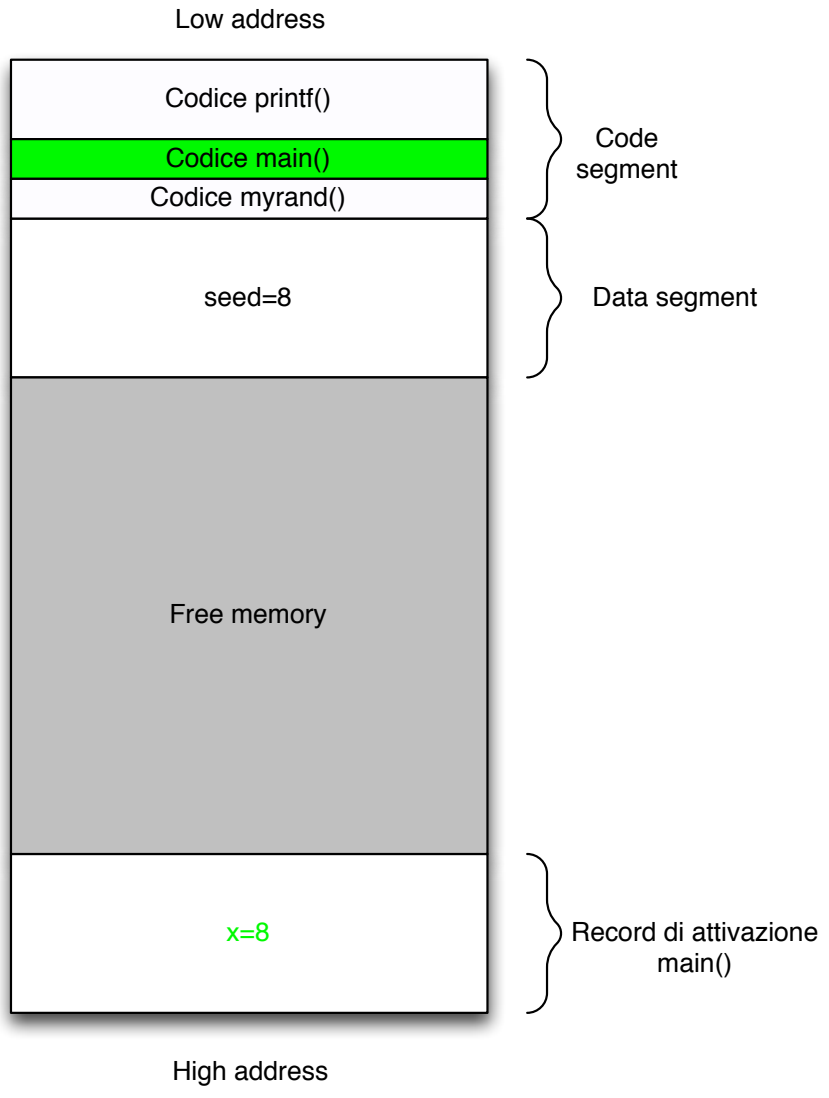
Esempio di chiamata a funzione

```
1  #include <stdio.h>
2
3  int myrand(void) {
4    static int seed = 1;
5    return seed = (5* seed +3)%16;
6  }
7
8  int main() {
9    int x;
10
11   x = myrand();
12   printf("%d\n",x);
13   x = myrand();
14   printf("%d\n",x);
15
16   return 0;
17 }
```



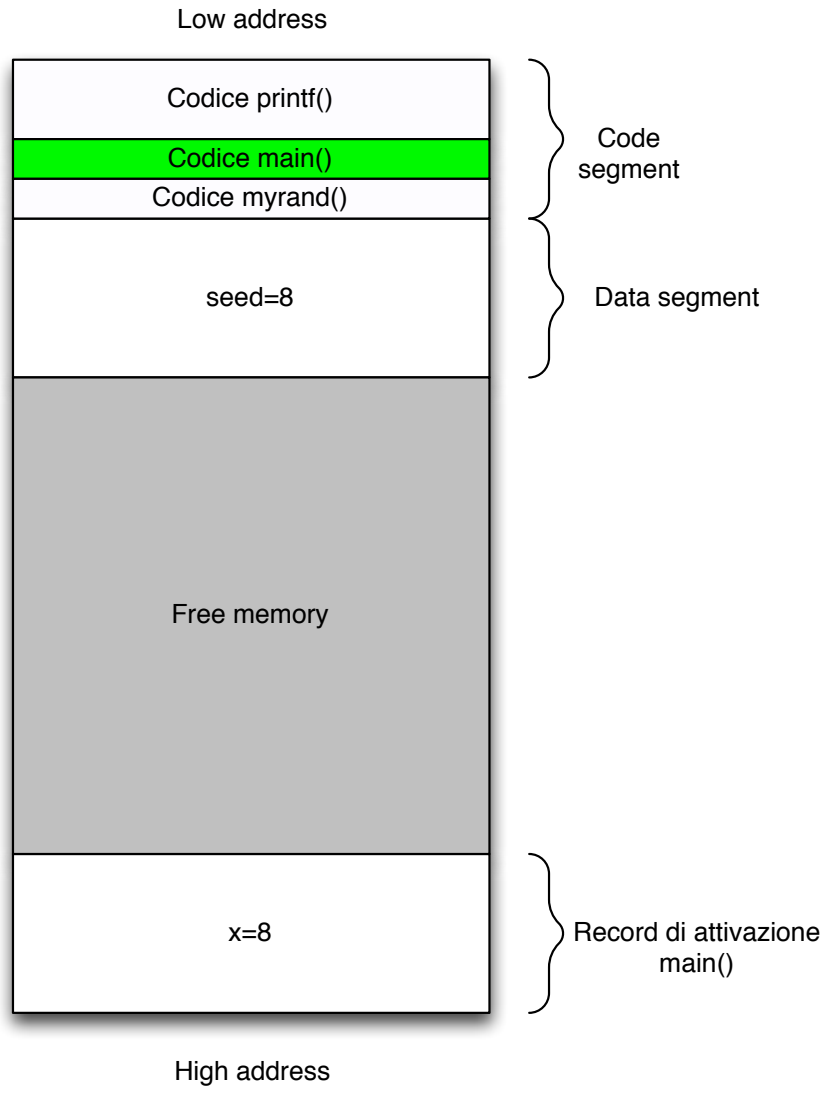
Esempio di chiamata a funzione

```
1  #include <stdio.h>
2
3  int myrand(void) {
4    static int seed = 1;
5    return seed = (5* seed +3)%16;
6  }
7
8  int main() {
9    int x;
10
11   x = myrand();
12   printf("%d\n",x);
13   x = myrand();
14   printf("%d\n",x);
15
16   return 0;
17 }
```



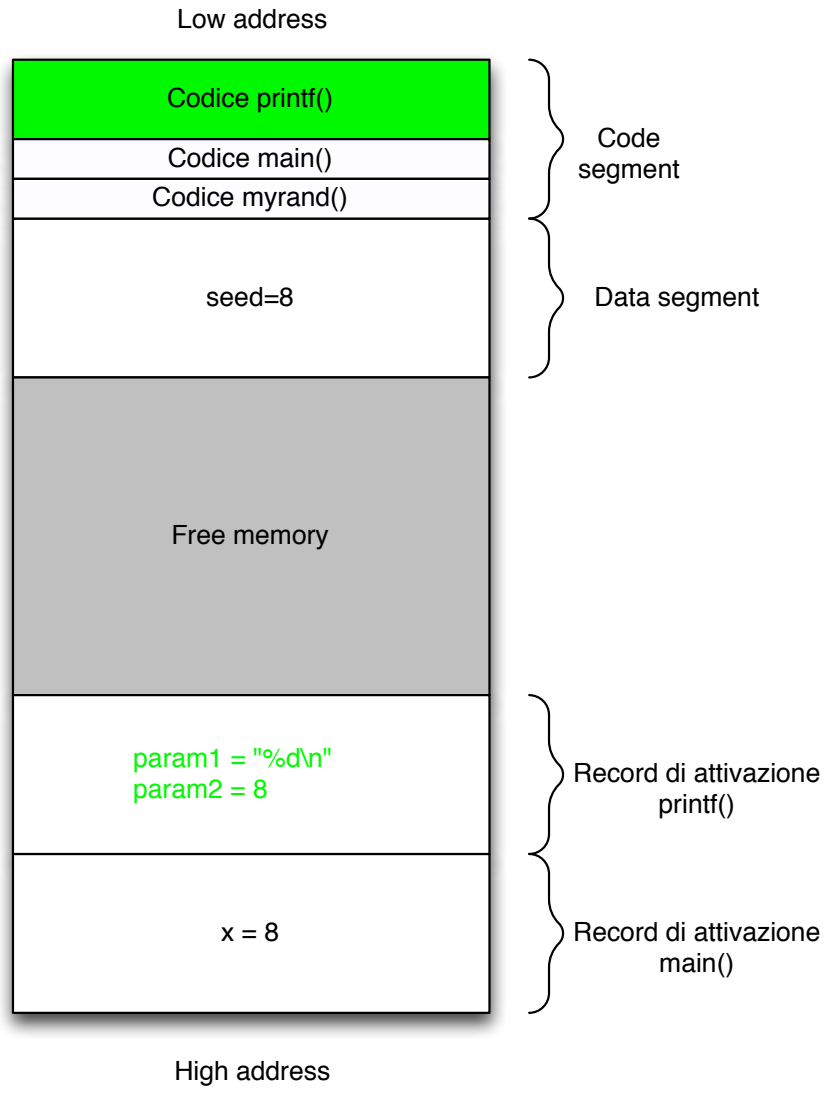
Esempio di chiamata a funzione

```
1  #include <stdio.h>
2
3  int myrand(void) {
4    static int seed = 1;
5    return seed = (5* seed +3)%16;
6  }
7
8  int main() {
9    int x;
10
11   x = myrand();
12   printf("%d\n",x);
13   x = myrand();
14   printf("%d\n",x);
15
16   return 0;
17 }
```



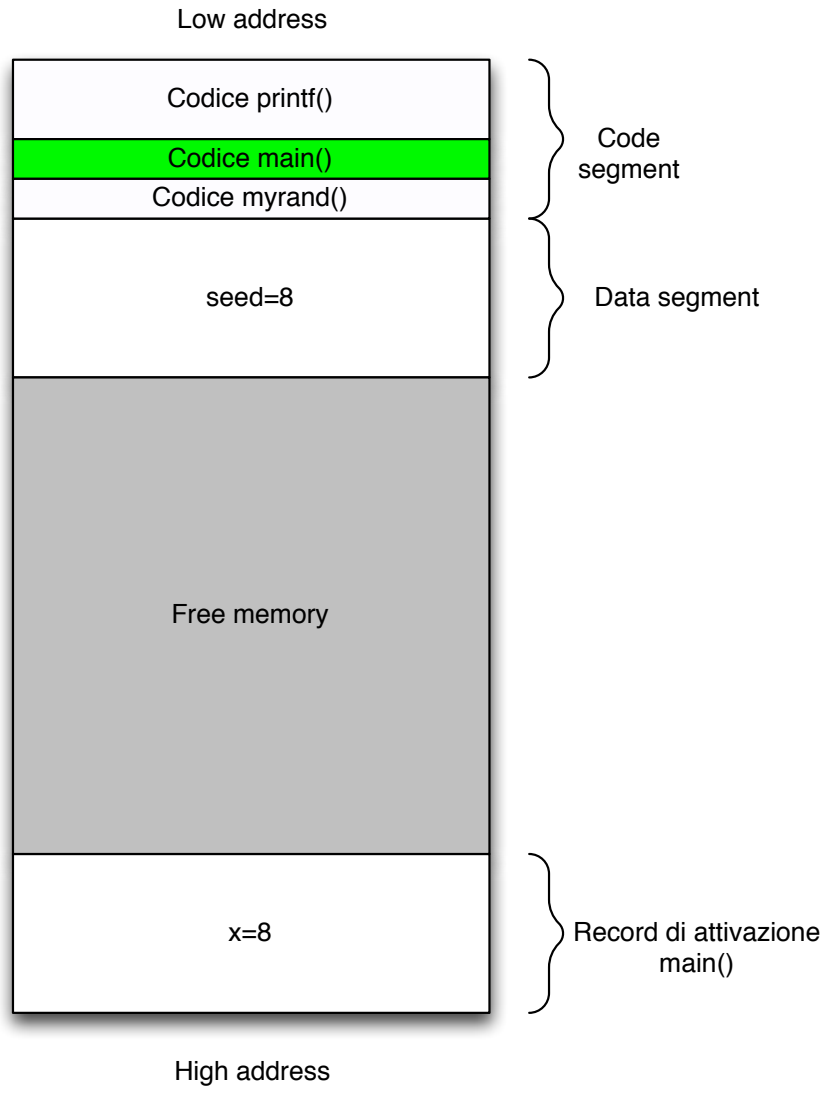
Esempio di chiamata a funzione

```
1  #include <stdio.h>
2
3  int myrand(void) {
4    static int seed = 1;
5    return seed = (5* seed +3)%16;
6  }
7
8  int main() {
9    int x;
10
11   x = myrand();
12   printf("%d\n",x);
13   x = myrand();
14   printf("%d\n",x);
15
16   return 0;
17 }
```



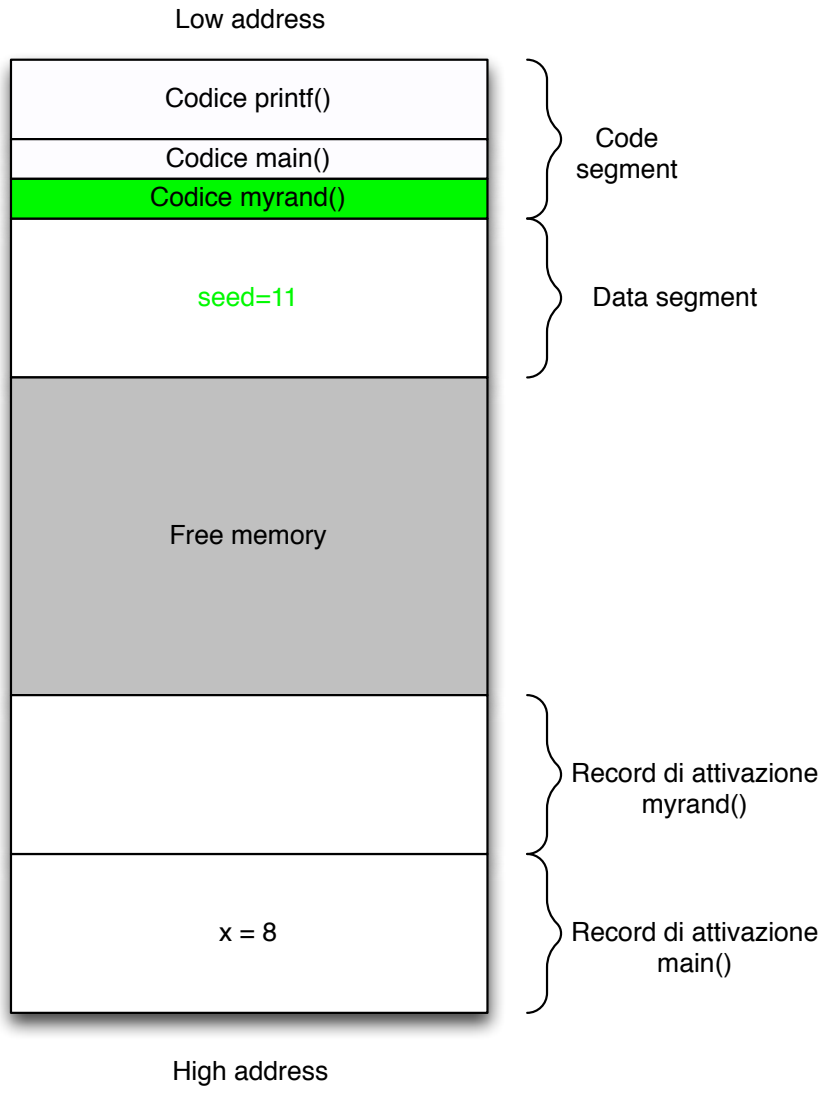
Esempio di chiamata a funzione

```
1  #include <stdio.h>
2
3  int myrand(void) {
4    static int seed = 1;
5    return seed = (5* seed +3)%16;
6  }
7
8  int main() {
9    int x;
10
11   x = myrand();
12   printf("%d\n",x);
13   x = myrand();
14   printf("%d\n",x);
15
16   return 0;
17 }
```



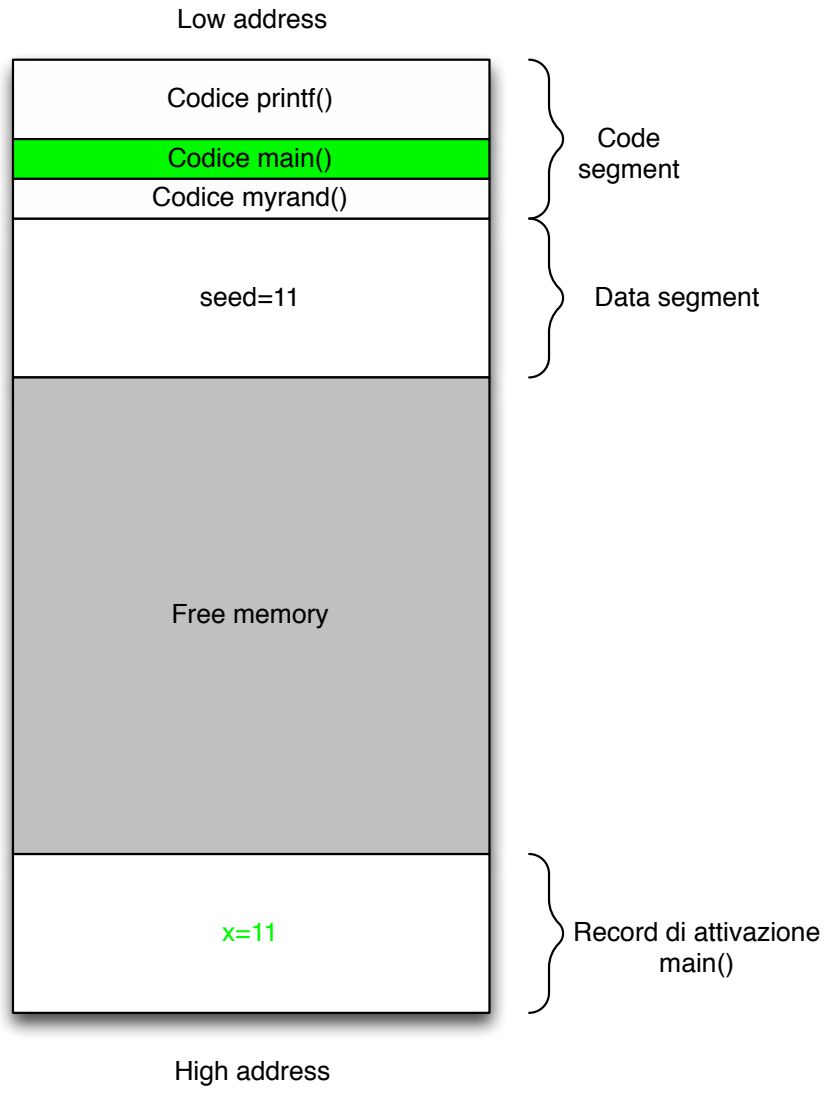
Esempio di chiamata a funzione

```
1  #include <stdio.h>
2
3  int myrand(void) {
4    static int seed = 1;
5    return seed = (5* seed +3)%16;
6  }
7
8  int main() {
9    int x;
10
11   x = myrand();
12   printf("%d\n",x);
13   x = myrand();
14   printf("%d\n",x);
15
16   return 0;
17 }
```



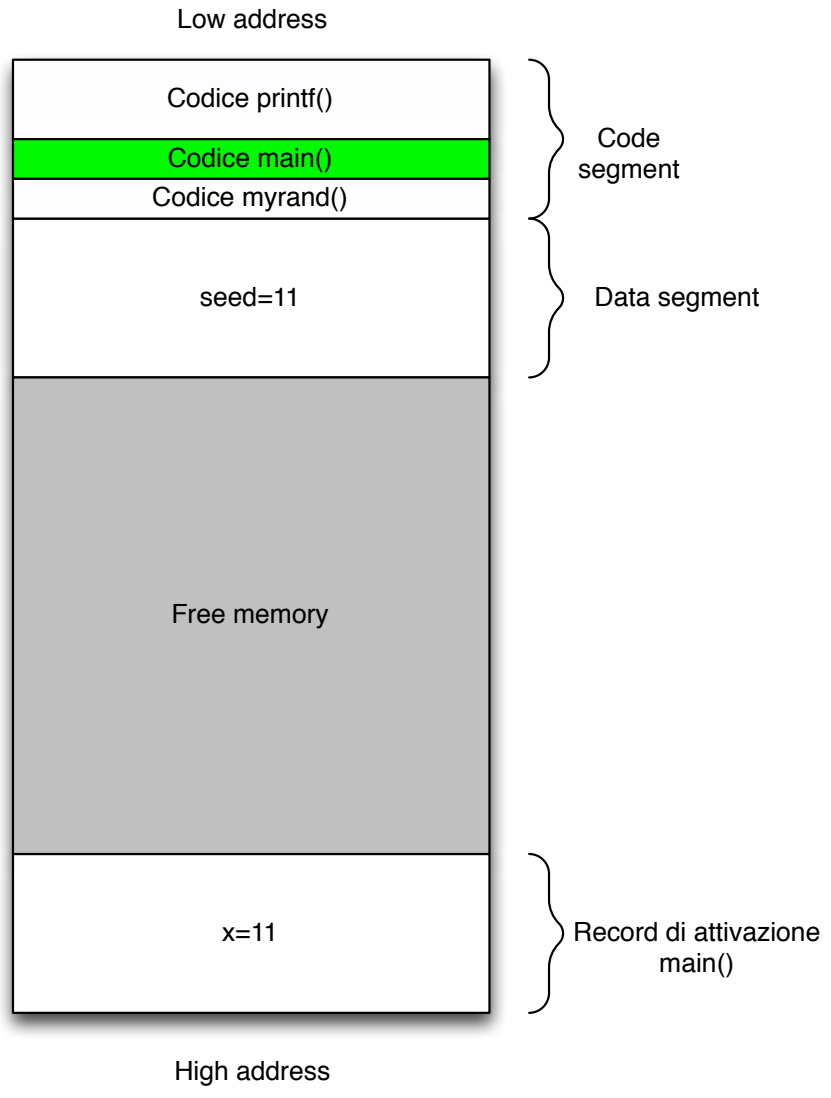
Esempio di chiamata a funzione

```
1  #include <stdio.h>
2
3  int myrand(void) {
4    static int seed = 1;
5    return seed = (5* seed +3)%16;
6  }
7
8  int main() {
9    int x;
10
11   x = myrand();
12   printf("%d\n",x);
13   x = myrand();
14   printf("%d\n",x);
15
16   return 0;
17 }
```



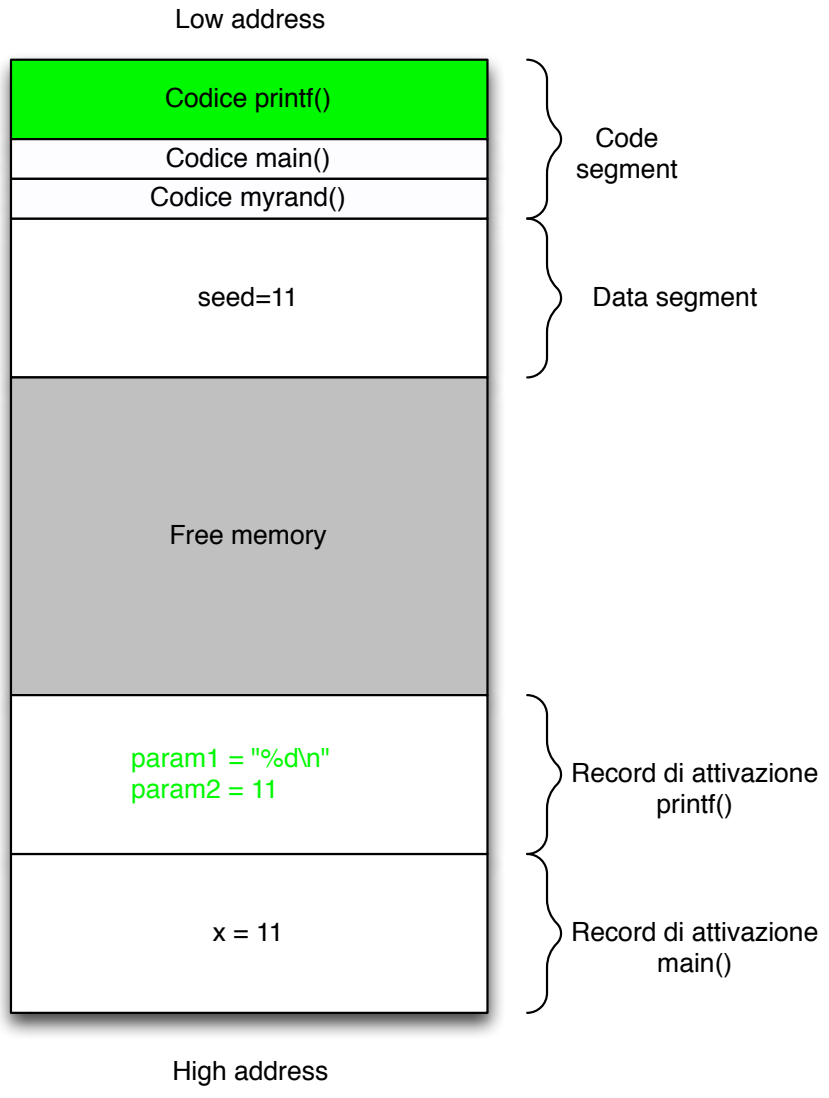
Esempio di chiamata a funzione

```
1  #include <stdio.h>
2
3  int myrand(void) {
4    static int seed = 1;
5    return seed = (5* seed +3)%16;
6  }
7
8  int main() {
9    int x;
10
11   x = myrand();
12   printf("%d\n",x);
13   x = myrand();
14   printf("%d\n",x);
15
16   return 0;
17 }
```



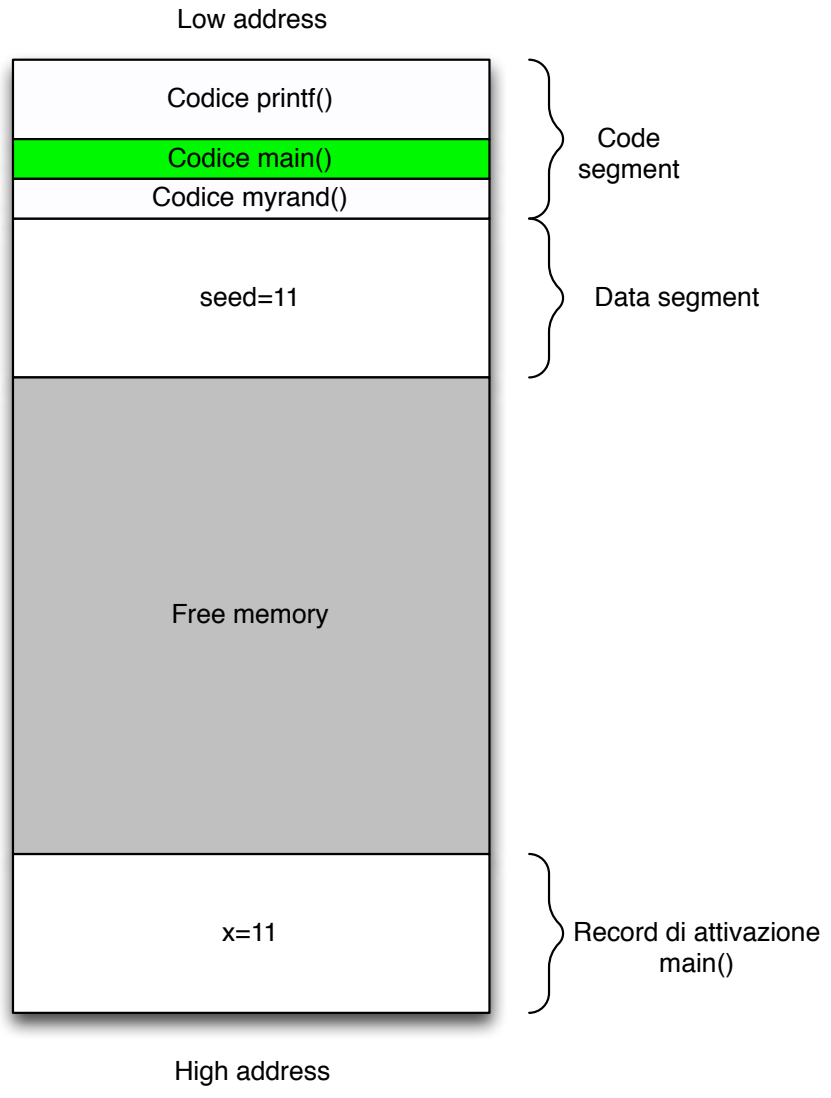
Esempio di chiamata a funzione

```
1  #include <stdio.h>
2
3  int myrand(void) {
4    static int seed = 1;
5    return seed = (5* seed +3)%16;
6  }
7
8  int main() {
9    int x;
10
11   x = myrand();
12   printf("%d\n",x);
13   x = myrand();
14   printf("%d\n",x);
15
16   return 0;
17 }
```



Esempio di chiamata a funzione

```
1  #include <stdio.h>
2
3  int myrand(void) {
4    static int seed = 1;
5    return seed = (5* seed +3)%16;
6  }
7
8  int main() {
9    int x;
10
11   x = myrand();
12   printf("%d\n",x);
13   x = myrand();
14   printf("%d\n",x);
15
16   return 0;
17 }
```



Esempio di chiamata a funzione

```c
#include <stdio.h>

int myrand(void) {
  static int seed = 1;
  return seed = (5* seed +3)%16;
}

int main() {
  int x;

  x = myrand();
  printf("%d\n",x);
  x = myrand();
  printf("%d\n",x);

  return 0;
}
```