Allocazione dinamica della memoria

Andrea Marin

Università Ca' Foscari Venezia
Laurea in Informatica
Corso di Programmazione part-time

a.a. 2011/2012

Tipi di memoria dati

Nella macchina astratta C esistono tre tipi di memoria per allocare le variabili:

1. Memoria statica
2. Stack
3. Memoria dinamica (heap)

Memoria statica

- Nella memoria statica vengono allocate le variabili globali

```
int a;
int vect[100];
int main() {
    int c;
}
```

- `a, vect` sono allocati nella memoria statica

Lo stack

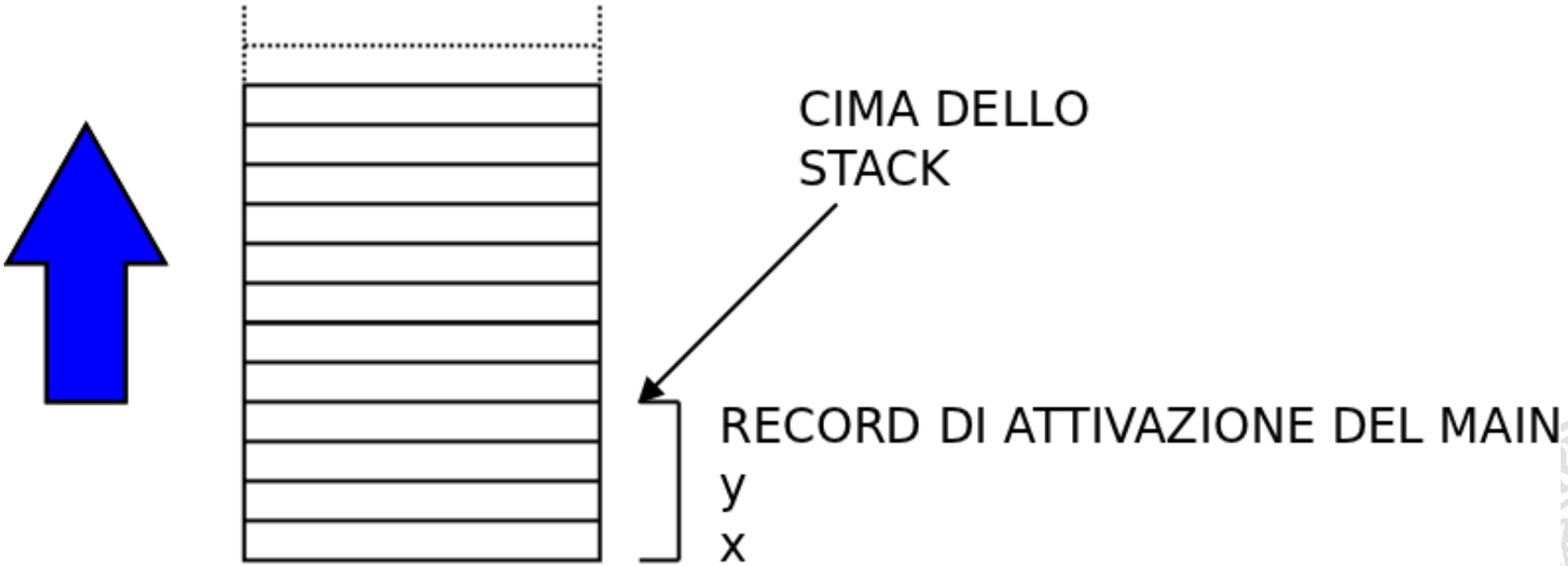
- La memoria stack è usata per memorizzare i record di attivazione delle funzioni
- Il record di attivazione è costituito da:
  - L'indirizzo di codice dell'istruzione successiva a quella che ha invocato la funzione (indirizzo di rientro)
  - I parametri della funzione
  - Le variabili locali alla funzione

Esempio: codice

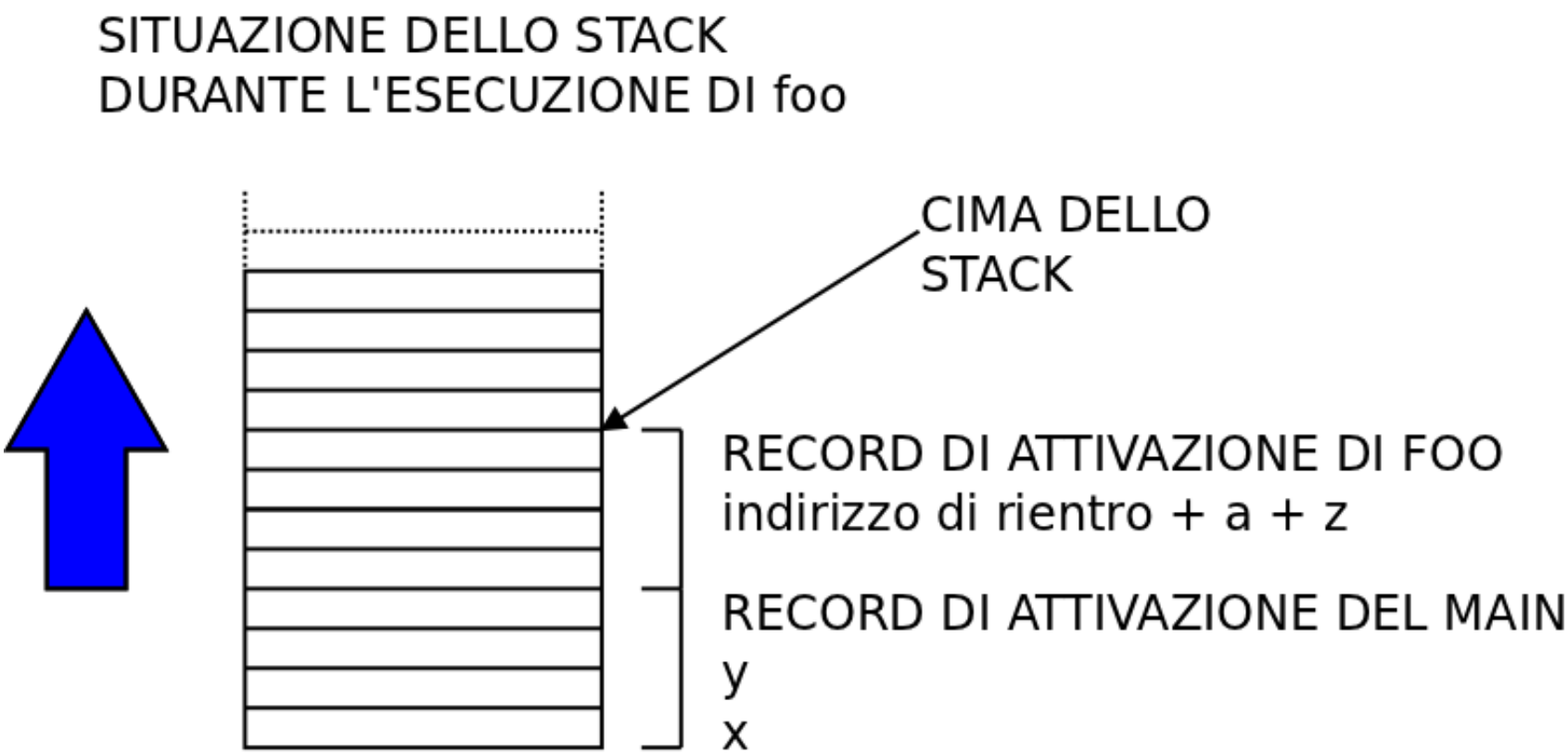
```
int foo(int z) {
    int a;
    a=z+1;
    return a;
}

int main() {
    int y =15;
    int x;
    x = foo(, 13);
    printf(``%d\n'', x);
    return 0;
}
```

Esempio /1: Stack

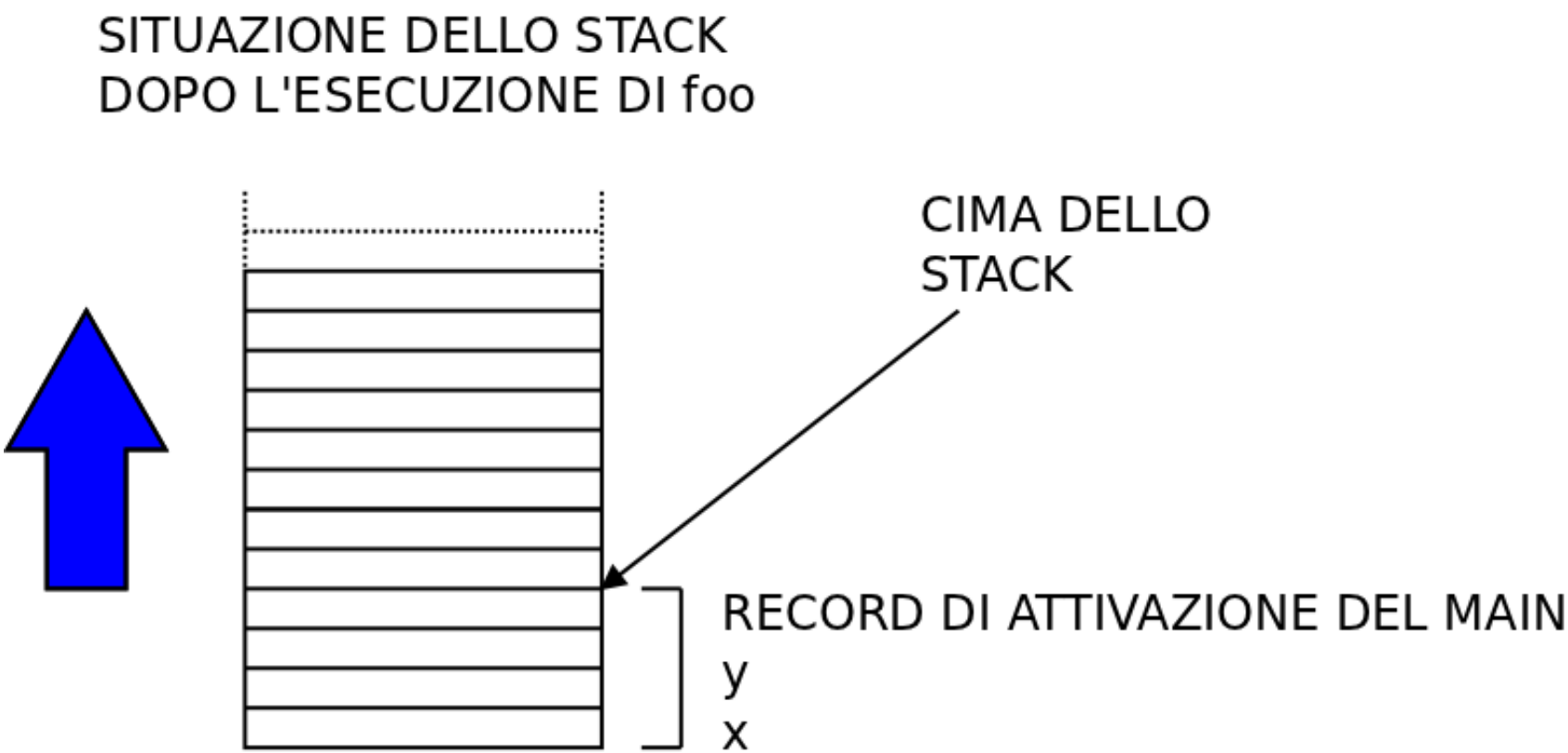
SITUAZIONE DELLO STACK PRIMA
DELLA CHIAMATA A foo



Esempio /2: Stack



Esempio /3: Stack



Uso dello stack e ricorsione

- L'uso dello stack consente ad una funzione di invocare un'altra funzione
- Questo causa la creazione di un nuovo record di attivazione sulla cima dello stack
- In questo modo le funzioni possono richiamare se stesse dando origine alla ricorsione

La memoria Heap

- Notiamo che sebbene -in generale- non sappiamo quanti record di attivazione saranno istanziati nello stack durante l'esecuzione del programma, *la dimensione di un record di attivazione si conosce dall'analisi del codice*
- La memoria dinamica consente di allocare spazi di memoria la cui dimensione si conosce solo in fase di esecuzione
  - Esempio: scrivi un programma che dopo aver acquisito il numero di rilevazioni disponibili di una sonda di temperatura, le memorizzi in un vettore
  - Quanto grande deve essere il vettore? È impossibile determinarlo in fase dall'analisi del codice perchè dipende da un valore in input

Uso della memoria Heap

- In C, l'allocazione nella memoria heap e la sua liberazione sono espliciti, cioè si usano apposite chiamate a funzioni
- La funzione `malloc` della libreria `stdlib.h` alloca uno spazio di memoria contiguo nell'heap e restituisce un puntatore al primo byte allocato
  - In case di fallimento restituisce il valore `NULL` che corrisponde all'indirizzo riservato `0`
- La funzione `free` della librerira `stdlib.h` libera lo spazio di memoria precendemente allocato con la `malloc`

malloc

- prototipo:

```
void* malloc(size_t size);
```

- `size_t` è un tipo equivalente ad `unsigned int`. Il parametro specifica la quantità di byte da allocare
- Il tipo del valore restituito è `void*`, che denota un valore di tipo indirizzo.
  - `void*` specifica che il tipo dell'oggetto indicizzato è sconosciuto
  - Il valore è l'indirizzo del primo byte allocato
  - Se l'allocazione non ha successo viene resituito l'indirizzo `NULL`

Calcolo dello spazio richiesto: l'operatore `sizeof`

- Per conoscere quanti byte occupa un tipo ti dato si può usare l'operatore `sizeof`
  - `sizeof(int)`: restituisce il numero di byte occupati da un `int`
  - `sizeof(float)`: restituisce il numero di byte occupati da un `float`
  - `...`

Esempio di allocazione vettore di float

```
int main() {
    float* ptr;
    int dim;
    scanf(``%d'', &dim); /*lettura della dimensione*/
    int i;
    ptr = malloc(sizeof(float)*dim);
    if (ptr) { /*spazio allocato*/
        for (i=0; i<dim; i++)
            scanf(``%f'', &ptr[i]); /*scanf(``%f'', ptr+i);
    }
    ..
}
```

Allocazione delle variabili

Eliminazione dello spazio allocato: `free`

- Prototipo:

```
void free(void* ptr)
```

- Il puntatore `ptr` deve contenere l'indirizzo del primo byte di memoria del blocco da liberare
- La memoria allocata va sempre liberata una volta terminato il suo uso
  - I blocchi di memoria non liberati causano *garbage*

Acquisizione di una stringa: problemi

Dal manuale GNU C

The getline function is the preferred method for reading lines of text from a stream, including standard input. The other standard functions, including gets, fgets, and scanf, are too unreliable. (Doubtless, in some programs you will see code that uses these unreliable functions, and at times you will come across compilers that cannot handle the safer getline function. As a professional, you should avoid unreliable functions and any compiler that requires you to be unsafe.)

- Cosa c'è di così insicuro nel seguente codice?

```
char stringa[100];
scanf(``%s'', stringa);
```

getline

- Le stringhe vanno acquisite con la funzione `getline` della libreria `stdlib`
- prototipo:

  size_t getline(char** pstr, size_t* pbytes, FILE* in);
- `pstr` è un puntatore al puntatore al primo byte della stringa.
- `pbytes` è un puntatore ad un intero che contiene il numero di bytes allocati per la stringa
- `in` è lo stream in input (tipicamente `stdin`)
- Lo spazio allocato per la stringa potrebbe non essere sufficiente a contenere l'input. In questo caso la `getline` modifica l'allocazione dell'heap in modo da contenerlo. In questo case modifica anche il contenuto della variabile puntata da `pbytes`

Esempio

```
#include <stdio.h>

int main()
{
   int bytes_read;
   int nbytes = 9;
   char *my_string;
   printf (``%s'',``Please enter a line of text.'');
   my_string = malloc (nbytes + 1);
   bytes_read = getline (&my_string, &nbytes, stdin);
   if (bytes_read == -1)
        printf (``ERROR!'');
   else {
        printf (``You typed: %s'', my_string);
        free(my_string);
   }
   return 0;
}
```

Situazione in memoria