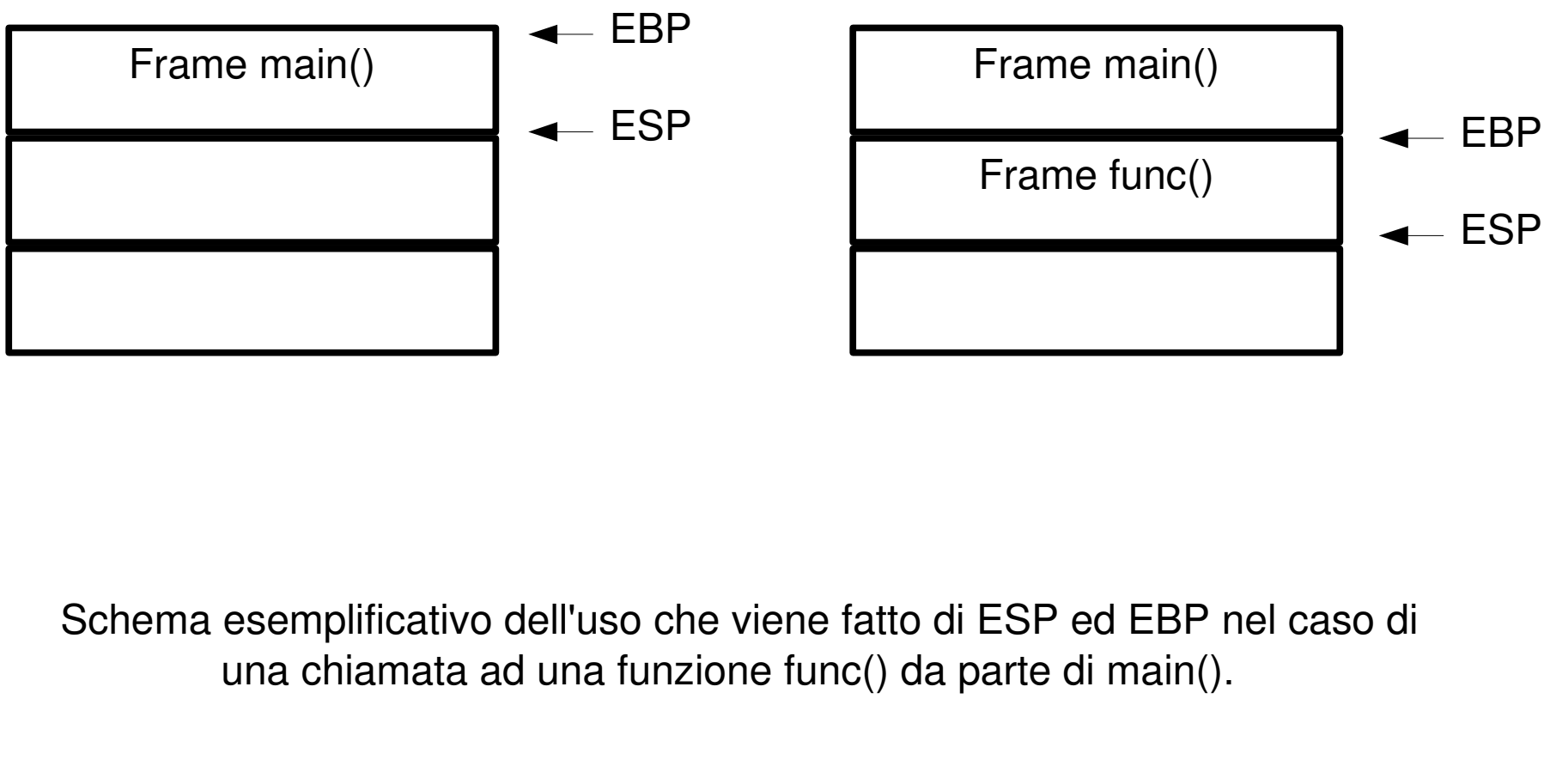
Lo Stack

- Lo stack e' una porzione dinamica di memoria.
- Lo stack come detto cresce verso il basso, cioe' verso indirizzi via via minori.
- La sua politica di gestione interna e' di tipo LIFO.
- Come sapete le uniche due operazioni possibili sono **PUSH** (aggiunge un valore in cima allo stack) e **POP** (recupera dalla cima dello stack l'ultimo valore aggiunto).
- Nello stack vengono memorizzati: parametri passati alla funzione, variabili locali (allocazione automatica), dati necessari a gestire la chiamata a funzione.
- Ogni volta che avviene una chiamata a funzione viene ad essa riservata una zona dello stack, chiamata “frame”.
- Ci sono due registri riservati alla gestione dello stack: **ESP** contiene l'indirizzo della “cima” dello stack, **EBP** contiene l'indirizzo della base del frame corrente

ESP/EBP



Schema esemplificativo dell'uso che viene fatto di ESP ed EBP nel caso di una chiamata ad una funzione func() da parte di main().

Lo Stack

Lo stack e' una zona della memoria cosi' importante che a parte i registri visti in precedenza, il processore prevede anche delle istruzioni specifiche per la sua gestione:

- **PUSH** che inserisce 1'elemento nello stack
- **POP** che preleva l'elemento che si trova nella cima dello stack
- Anche le istruzioni **CALL**, **LEAVE** e **RET**, sono relate strettamente al'uso dello stack come vedremo.

Call Sequence C

Chiamata a funzione in C (tutto quello che riporteremo e' valido per l'architettura **x86** e **gcc version 4.1.1**):

- Memorizzare nello stack l'indirizzo di ritorno. Cioe' l'indirizzo da dove riprendere l'esecuzione una volta completata la funzione.
- Oltre questo in generale e' necessario memorizzare anche altre informazioni (i.e. **EBP**).
- Passare I parametri alla funzione chiamata. A tale scopo come vedremo viene usato lo stack.
- Allocare lo spazio necessario all'allocazione “automatica” relativa alla funzione chiamata.
- Sara' poi necessario trovare qualche meccanismo utile a recuperare gli eventuali valori ritornati dalla funzione chiamata.

Call Sequence C

```
int func1 (int x, int y, int z, int a)
{
   int t = 8;
   return (x+t);
}

int main()
{
   int b = 0;
   b = func1(1, 2, 3, 4);
   return(0);
}
```

Step by Step

Step 1: viene fatto il push dei parametri nello stack (da destra verso sinistra)

```
8048377:        83 ec 20                    sub    $0x20,%esp
804837a:        c7 45 f8 00 00 00 00        movl   $0x0,0xfffffff8(%ebp)
8048381:        c7 44 24 0c 04 00 00        movl   $0x4,0xc(%esp)
8048388:        00
8048389:        c7 44 24 08 03 00 00        movl   $0x3,0x8(%esp)
8048390:        00
8048391:        c7 44 24 04 02 00 00        movl   $0x2,0x4(%esp)
8048398:        00
8048399:        c7 04 24 01 00 00 00        movl   $0x1,(%esp)

80483a0:        e8 af ff ff ff              call   8048354 <func1>
80483a5:        89 45 f8                    mov    %eax,0xfffffff8(%ebp)
```

Step 2: viene fatta la call prima del jump vero e proprio viene fatto il push dell'indirizzo di ritorno

Step by Step

Step 3: EBP contiene il frame pointer della funzione chiamante. Questo sara' necessario al momento della return

```
08048354 <func1>:
 8048354:       55                            push   %ebp

 8048355:       89 e5                         mov    %esp,%ebp

 8048357:       83 ec 10                      sub    $0x10,%esp

 804835a:       c7 45 fc 08 00 00 00          movl   $0x8,0xfffffffc(%ebp)
 8048361:       8b 45 fc                      mov    0xfffffffc(%ebp),%eax
 8048364:       03 45 08                      add    0x8(%ebp),%eax
```

Step 6: dopo aver "computato" il valore di riorno viene messo nel registro EAX

Step 4: Il nuovo frame Pointer sara' dato dll'auttale "testa" dello stack

Step 5: Allocazione automatica nello stack

Call Sequence C

```
8048367: c9    leave
8048368: c3    ret
```

Step 7: Copia l'attuale valore di EBP (Frame Pointer) in ESP. Questo chiaramente riduce la dimesione dello stack. A questo punto nella “cima” dello stack si trovera' il “vecchio” Frame Pointer (in altre parole quello relativo alla funzione chiamente)

Step 8: Esegue il POP del vecchio Frame Pointer in EBP.(undoing Step 3)

Step 9: L'istruzione ret semplicemente fa un pop dell'indirizzo di ritorno nell'Instruction Pointer (memorizzato durante la call) (come se fosse un pop %eip, ma i programmi user non hanno accesso diretto ad EIP)

Just for Fun

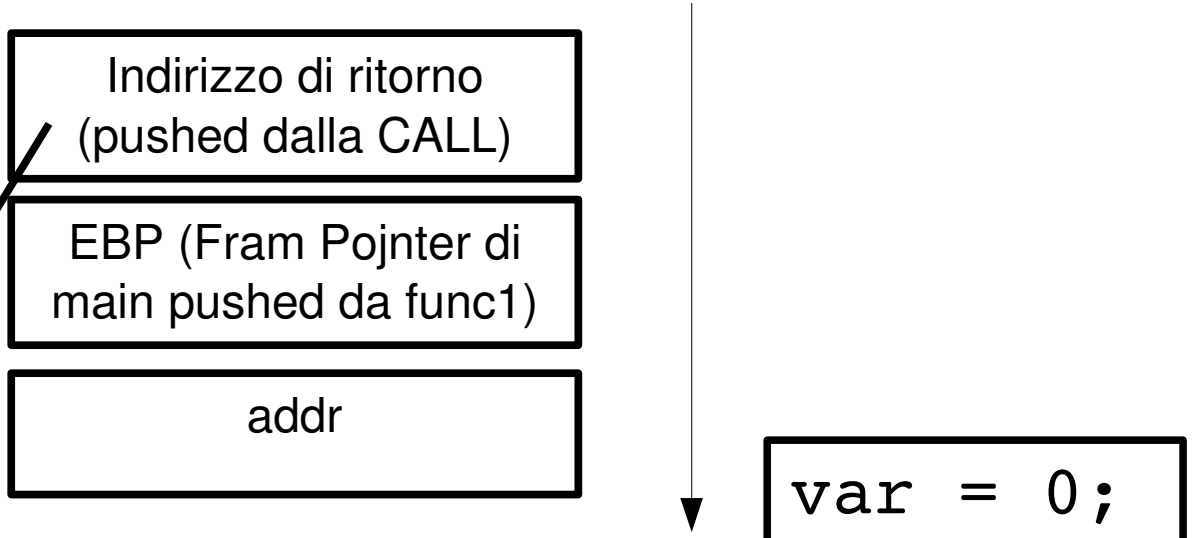
```
#include <stdio.h>

void func ()
{
  int *addr;
  addr = (int *) &addr + 2*((sizeof (void *)/sizeof (int)));
  (*addr) += 7;
}

int main ()
{
  int var = 1;
  func();
  var = 0;
  printf ("%d \n", var);
  return 0;
}
```

```
$ gcc -o stack1 stack1.c
$ ./stack1
1
```

Just for Fun



objdump -D stack1

....

```
4004c8:     b8 00 00 00 00          mov    $0x0,%eax
4004cd:     e8 c6 ff ff ff          callq  400498 <func>

4004d2:     c7 45 fc 00 00 00 00    movl   $0x0,0xfffffffffffffffc(%rbp)

4004d9:     8b 75 fc                mov    0xfffffffffffffffc(%rbp),%esi
4004dc:     bf f8 05 40 00          mov    $0x4005f8,%edi
4004e1:     b8 00 00 00 00          mov    $0x0,%eax
4004e6:     e8 ad fe ff ff          callq  400398 <printf@plt>
```

...

?

Cosa stampera' il seguente programma ?

File ritorna.c:

```
#include <stdio.h>
void ritorna (double a)
{
   printf ("%f\n", a);
   return;
}
```

File error.c

```
#include <stdio.h>
int main ()
{
  double a = 25.0;
  ritorna (a, 2, 3, 4, 5, 6, 7);
  printf ("%f\n", a);
  return 0;
}
```

```
$ gcc -o error error.c ritorna.c
$ ./error
25.000000
25.000000
```

Numero variabile di argomenti

Vediamo come e' possibili definire ed usare una funzione che riceva in input un numero variabile di argomenti. (Possiamo adesso facilmente intuire come sia possibile “implementare” tale “meccanismo”)

```
int test (int val, ...) {
  va_list ap;
  double b;
  int a;

  va_start (ap, val);

  a = va_arg (ap, int);
  b = va_arg (ap, double);

  DBGI(a);
  DBGF(b);

  va_end (ap);

  return 0;
}
```

Esercizio

Problema: Scrivere una funzione, che dato in input una stringa di formato ed un numero arbitrario di numeri (ai con i=1,...,n) o 'double' o 'int', stampi i valori di (ai con i=1,...,n).

Il formato puo' ad esempio essere dato da una sequenza di n lettere 'd' ed 'f'. Dove 'd' stara' ad indicare 1 intero ed 'f' un double. (soluzione1.c)

getopt

Il parsing delle opzioni da linea di comando puo' essere affrontato usando una funzione parte della libc Linux "getopt" o la "getopt_long".

```
int getopt(int argc, char *const argv[], const char *optstring)
```

Esegue il parsing degli argomenti passati da linea di comando riconoscendo le possibili opzioni segnalate con optstring.

Ritorna il carattere che segue l'opzione, ':' se manca un parametro all'opzione, '?' se l'opzione è sconosciuta, e -1 se non esistono altre opzioni.

getopt

Quando la funzione trova un'opzione essa ritorna il valore numerico del carattere, in questo modo si possono eseguire azioni specifiche usando uno switch; getopt inoltre inizializza alcune variabili globali:

- **char *optarg** contiene il puntatore alla stringa parametro dell'opzione.
- **int optind** alla fine della scansione restituisce l'indice del primo elemento di argv che non è un'opzione.
- **int opterr** previene, se posto a zero, la stampa di un messaggio di errore in caso di riconoscimento di opzioni non definite.
- **int optopt** contiene il carattere dell'opzione non riconosciuta.

getopt_long

```
int getopt_long(int argc, char * const argv[], const char
*optstring, const struct option *longopts, int *longindex);
```

longopts e' un puntatore al primo elemento di un'array di **struct option:**

```
struct option {
   const char *name;
   int has_arg;
   int *flag;
   int val;
};
```

name: e' il nome dell'opzione lunga
has_arg: serve a stabilire se l'opzione ha argomento oppure no
flag: nel nostro caso sara' sempre NULL (siete curiosi, “man getopt”)
val: valore che ritornera' la getopt_long (nel nostro caso la corrispettiva opzione corta)

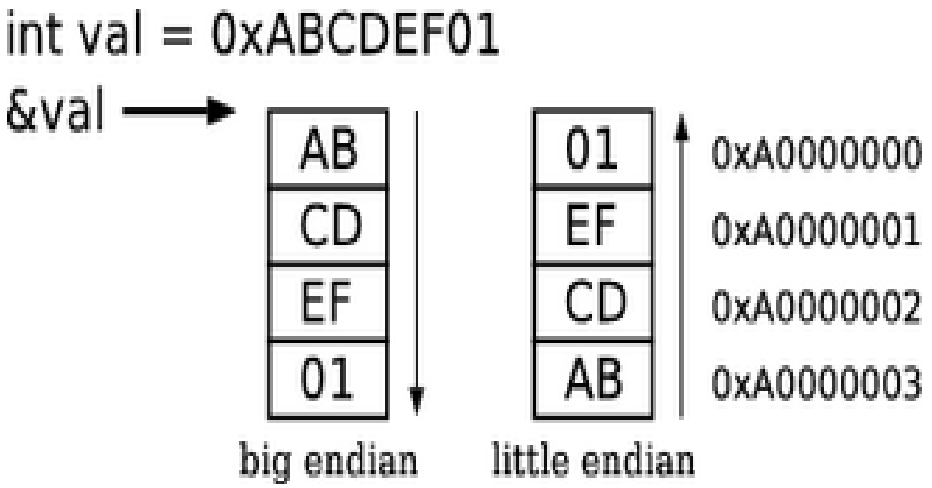
Vediamo 1 esempio

```
 while (1) {
  int c;
  static struct option long_options[] = {
    {"help", no_argument, NULL, 'h'},
    {"val", required_argument, NULL, 'v'},
    {"other-val", required_argument, NULL, 'o'},
    {0,0,0,0}
  };
  c = getopt_long (argc, argv, "hv:o:", long_options, NULL);
  if (c == -1)
    break;
  switch (c) {
    case 'v':
      val = atof (optarg);
      break;
    case 'o':
      othrval = atof (optarg);
      break;
    case 'h':
      usage (argv[0]);
    default:
      usage (argv[0]);
      break;
  }
}
if (optind >= argc)
  usage (argv[0]);
```

Esercizio

A partire dal programma che alloca e riempie una matrice di dimensione N, scriverne uno che usi "getopt_long" per gestire dimensioni delle matrici e nome file dove "eventualmente" scrivere la matrice (se il nome del file non e' specificato la matrice sara' scritta nello stdout). (soluzione2.c)

endianness



Un numero binario in un computer puo' essere in due modi, chiamati “big endian” e “little endian” a seconda di come i singoli bit vengono aggregati per formare le variabili intere (ed in genere in diretta corrispondenza a come sono poi in realtà cablati sui bus interni del computer).

- Big Endian: il byte che contiene i bit piu' significativi si trova all'indirizzo &val e i byte con i bit meno significativi ad indirizzi successivi
- Little Endian: caso opposto, in cui si parte dal bit meno significativo.

endianness

Tramite un semplice esempio vediamo come e' possibile determinare l'endianess del proprio computer, e come eventualmente operare il **Marshalling**