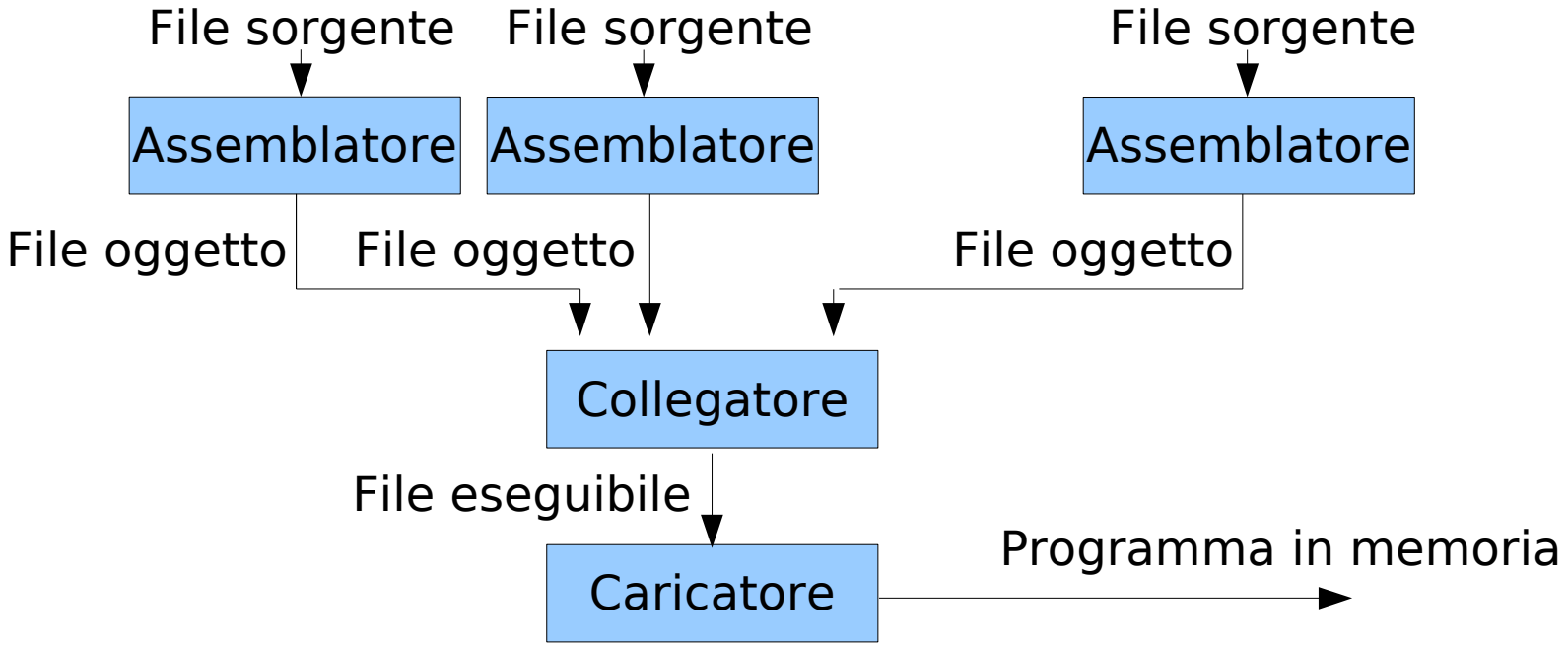
Moduli

- Un programma Assembler può essere costituito da uno o più moduli
- Un modulo è un insieme di entità che si trovano su un unico file detto file sorgente



- Un modulo può usare nomi definiti in altri moduli: tali nomi vanno dichiarati esterni
  - .extern nome
- Alcuni nomi definiti in un modulo possono essere utilizzati da altri moduli: tali nomi vanno dichiarati globali
  - .global nome

Esempio di programma a moduli

```
#------------  file1.s  -----------------
.extern alpha, beta, esamina

.data
kappa:  fill  8, 1
#-----------------------------------------
.text
.include     “servizio.s”
.global  _main

_main:
ancora: call     tastiera
        cmpb     $0x0d, %al
        je       fine
        movb     %al, alpha
        movl     $kappa, beta
        call     esamina
        call     video
        movb     0x20, %al
        call     video
        movl     $0, %esi
ripeti: movb     kappa(%esi), %al
        call     video
        incl     %esi
        cmpl     $8, %esi
        jb       ripeti
        movb     0x0d, %al    # CR
        call     video
        movb     0x0a, %al    # LF
        call     video
        jmp      ancora
fine:   xorl     %eax, %eax
        ret
#---------- fine file1.s ------------------
```

Per compilare e collegare i due moduli:

gcc file1.s file2.s -o codifica

Esempio di programma a moduli

```
#--------------- file2.s ------------------
.data

.global  alpha, beta
alpha:   .byte    0
beta:    .long    0
#----------------------------------------
.text
.global  esamina
esamina:
         pushl    %eax
         pushl    %ebx
         pushl    %esi
         movb     alpha, %al
         movl     beta, %ebx
         movl     $0, %esi
ciclo:   testb    $0x80, %al
         je       zero
         movb     $'1, (%ebx, %esi)
         jmp      avanti
zero:    movb     $'0, (%ebx, %esi)
avanti:  shlb     $1, (%al)
         incl     %esi
         cmpl     $8, %esi
         jb       ciclo
         popl     %esi
         popl     %ebx
         popl     %eax
         ret
#----------------- fine file2.s ----------
```

Il passaggio dei parametri avviene attraverso variabili globali

- i moduli potrebbero usare la pila ...

Programmazione mista

- Un programma organizzato a moduli può essere scritto utilizzando linguaggi di programmazione differenti per i vari moduli
  - i traduttori dei vari linguaggi devono produrre moduli oggetto che hanno tutti la stessa struttura
  - ogni modulo sorgente viene tradotto con il proprio traduttore, quindi tutti i moduli oggetto sono collegati da un unico collegatore
- I vari traduttori devono fare uso della stesso standard per la traduzione degli identificatori, per la rappresentazione dei dati, per la modalità di aggancio dei sottoprogrammi
- Faremo riferimento alla convenzione sui nomi utilizzata dal compilatore C
- Il compilatore C++ opera in modo diverso (con riferimento agli identificatori delle funzioni), ma può essere reso compatibile con le convenzioni del C
  - in un programma C++ si può dichiarare una funzione come *extern "C"*, in questo caso vengono rispettate le convenzioni sugli identificatori del C

Regole di visibilità e collegamento

- Un programma C++ è costituito da un modulo principale, che contiene la definizione della funzione *main()* e da eventuali moduli secondari
  - ogni modulo è contenuto in un file separato
- Una funzione C++ può restituire un valore oppure essere di tipo void
- Con riferimento ad un modulo si chiama blocco tutto ciò che è racchiuso tra '{' e '}'
- Regole di visibilità:
  - gli identificatori dichiarati in un blocco, o identificatori *locali*, sono visibili dal punto in cui sono dichiarati fino alla fine del blocco stesso
  - i parametri formali di un sottoprogramma hanno le stesse regole di visibilità degli identificatori locali dichiarati nel blocco più esterno del sottoprogramma stesso
  - se un identificatore locale viene riutilizzato per dichiarare una nuova entità in un blocco locale più interno, la nuova entità nasconde quella originaria
  - gli identificatori dichiarati al di fuori dei blocchi, o identificatori *condivisi*, sono visibili dal punto in cui sono dichiarati fino alla fine del modulo
  - se un identificatore condiviso viene riutilizzato per dichiarare una entità locale, la nuova entità locale nasconde parzialmente quella condivisa (per default l'entità riferita è quella locale, ma può essere riferita anche quella condivisa con l'operatore di risoluzione di visibilità '::')
- Fra gli identificatori condivisi vi sono anche i nomi dei sottoprogrammi

Regole di visibilità e collegamento

- Collegamento
  - uno stesso tipo deve essere dichiarato in tutti i moduli in cui si usa
  - una variabile o una funzione deve essere dichiarata in tutti i moduli in cui si usa, e deve essere definita in un solo modulo
  - in un modulo si possono riferire variabili e funzioni definite in altri moduli, dichiarando tali entità come esterne (per le funzioni le dichiarazioni sono implicitamente esterne)
  - variabili e funzioni possono essere riferite anche da altri moduli purché siano definite al di fuori dei blocchi (condivise): tali entità sono anche dette globali
- Le regole di visibilità e collegamento si basano solo sulla struttura testuale del programma (sono statiche)

Ambiente di una istanza di sottoprogramma

- Un sottoprogramma viene chiamato (messo in esecuzione) con dei parametri attuali (valori dei parametri formali) che dipendono dalle regole di visibilità del blocco chiamante
- Il sottoprogramma chiamato ha le regole di visibilità del blocco corrispondente, che possono essere diverse da quelle del blocco chiamante

```
extern “C” void primo(int* a) {
    int aa;
    // ...
    aa = 1;
    *a = 2 + *a;
    // ...
}
```

```
extern “C” void secondo(int b) {
    int bb;
    //...
    primo(&b);
    primo(&bb);
    //...
}
```

- Durante l’esecuzione di un programma, un sottoprogramma può essere richiamato più volte
  - ogni volta che un sottoprogramma va in esecuzione si ha una *istanza* diversa di quel sottoprogramma

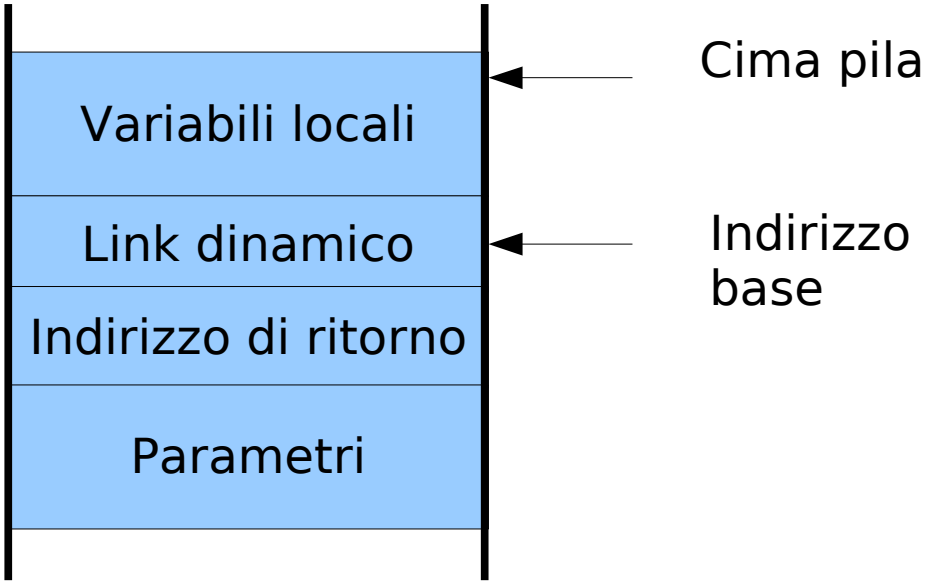
Ambiente di una istanza di sottoprogramma

- Il livello dinamico di una istanza è il numero di istanze non ancora terminate (di quel sottoprogramma o di altri sottoprogrammi), a partire dalla prima istanza del sottoprogramma *main()*
  - vanno in esecuzione con la regola LIFO
- Tempo di vita delle variabili (e costanti):
  - le variabili condivise hanno lo stesso tempo di vita del programma (variabili statiche)
  - le variabili locali esistono per il tempo di esecuzione del sottoprogramma stesso (variabili automatiche): ogni esecuzione del sottoprogramma ha la propria copia delle variabili locali
  - le variabili create e distrutte dinamicamente hanno un tempo di vita che va dalla loro creazione alla loro distruzione (variabili dinamiche): in ogni caso distrutte al termine del programma
- Gli oggetti riferibili da una istanza di sottoprogramma costituiscono l'ambiente di quella istanza
  - per ogni istanza esistono parametri attuali diversi e copie differenti delle variabili locali
  - l'ambiente varia da istanza a istanza, ma la "forma" dell'ambiente è uguale per tutte le istanze di uno stesso sottoprogramma (numero e tipo di argomenti e variabili locali sono sempre gli stessi)
  - l'ambiente di una istanza si compone di
    - un ambiente globale (insieme delle variabili globali) uguale per tutte le istanze
    - un ambiente locale (insieme parametri attuali e variabili locali)

Ambiente e record di attivazione

- Ogni sottoprogramma ha una singola copia di codice (in Ass. nella sezione testo)
- Ogni istanza di un sottoprogramma ha un proprio ambiente in cui è possibile riferire: tutti i sottoprogrammi, l'ambiente globale, l'ambiente locale
  - l'ambiente globale in Assembler occupa la sezione dati
    - un oggetto appartenente all'ambiente globale si riferisce con un indirizzo assoluto
  - l'ambiente locale viene ad occupare una parte della pila
    - un oggetto appartenente all'ambiente locale si riferisce con un indirizzo relativo costituito da
      - un indirizzo base che individua in pila l'ambiente locale attuale
      - da uno spiazzamento che individua l'oggetto all'interno dell'ambiente e che è invariante per tutte le istanze
- Ogni volta che si ha una nuova istanza di sottoprogramma, viene costruito sulla pila un record di attivazione che contiene
  - i parametri
  - lo spazio per le variabili locali
  - l'indirizzo di ritorno al sottoprogramma chiamante
  - link dinamico al record di attivazione del livello dinamicamente precedente

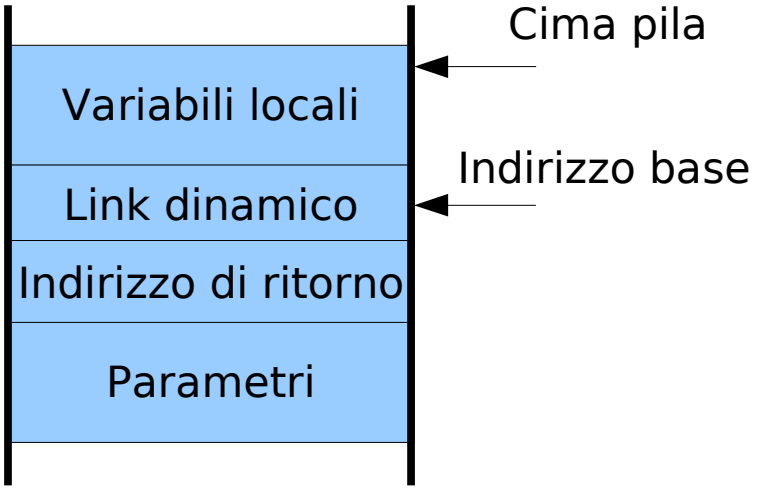
Record di attivazione



- Un record di attivazione utilizza un registro base che contiene l'indirizzo base del record di attivazione e che consente di individuare l'ambiente locale
- Ogni volta che viene creato un nuovo record di attivazione, il valore del registro base viene salvato in pila e successivamente ripristinato quando il record di attivazione costruito viene distrutto
  - il link dinamico è il vecchio valore del registro base, pari all'indirizzo base del record di attivazione dinamicamente precedente

Record di attivazione

- La costruzione del record di attivazione viene iniziata dal programma chiamante e completata dal sottoprogramma chiamato, mediante un apposito prologo
  - il chiamante inserisce in pila:
    - i parametri attuali
    - l'indirizzo di ritorno (istruzione CALL)
  - il chiamato inserisce in pila
    - il link dinamico
    - lo spazio per le variabili locali



- I parametri attuali vengono riferiti con spiazzamenti positivi e le variabili locali con spiazzamenti negativi (rispetto all'indirizzo base)
- La distruzione del record di attivazione viene iniziata dal chiamato, mediante un apposito epilogo, e completata dal chiamante
  - il chiamato rimuove dalla pila
    - lo spazio per le variabili locali
    - il link dinamico
    - l'indirizzo di ritorno (istruzione RET)
  - il chiamante rimuove dalla pila
    - lo spazio per i parametri attuali

Rappresentazione dei dati e modalità di trasmissione dei parametri

- Rappresentazione dei dati:
  - tipo *char* (o *unsigned char*): 8 bit
  - tipo *bool*: 8 bit (l'intero 0 per la costante false, 1 per true)
  - tipo *short int* (o *unsigned short int*): 16 bit
  - tipo *int* (o *unsigned int*): 32 bit
  - tipo *long int* (o *unsigned long int*): 32 bit
  - tipo enumerazione: un intero su 32 bit, il valore dipende dal numero d'ordine dell'enumeratore
  - tipo puntatore: un indirizzo lineare su 32 bit
  - tipo array: valori degli elementi ordinati per righe
  - tipo struttura o unione: valori degli elementi nell'orine specificato
- Il tipo di un parametro formale determina il tipo di valore del corrispondente argomento attuale. Tale valore può rappresentare un indirizzo, per esempio, quando il parametro formale è un puntatore
- Poiché un array corrisponde all'indirizzo del primo elemento, un parametro formale di tipo array è un puntatore, e il corrispondente parametro attuale è l'indirizzo del primo elemento
- I parametri possono essere trasmessi dal chiamante al chiamato in due modalità: per *valore* e per *indirizzo*
  - un puntatore viene trasmesso per valore, mentre l'oggetto puntato viene trasmesso per indirizzo
  - un array viene sempre trasmesso per indirizzo, quello del primo elemento

Modalità di trasmissione dei parametri (cont.)

- Un parametro di tipo struttura viene trasmesso per valore
  - se una struttura è costituita da un array, si ha come conseguenza la trasmissione per valore dell'array
- Una funzione non void che restituisce un valore lascia il risultato in
  - AL se il valore è di tipo *char*, *unsigned char*, o *bool*
  - AX se il valore è di tipo *short int* o *unsigned short int*
  - EAX se il valore è di tipo *int*, *unsigned int*, *long int*, *unsigned long int*, di un tipo enumerato o di un tipo puntatore

Istruzioni di interfaccia del chiamante e del chiamato

- Azioni eseguite dal chiamante (interfaccia del chiamante):
  - immettere in pila i parametri attuali (valori o indirizzi) in ordine inverso rispetto a quello di chiamata (allineati alla parola lunga)
  - eseguire l'istruzione di chiamata
  - dopo l'istruzione di chiamata, rimuovere dalla pila lo spazio occupato dai parametri attuali (numero intero di parole lunghe)
- Azioni eseguite dal chiamato (interfaccia del chiamato). Nel calcolatore PC, il registro base destinato a contenere l'indirizzo base di un record di attivazione è EBP.
  - prologo
    - salvare in pila il contenuto del registro EBP (costruzione del link dinamico)
    - ricopiare nel registro EBP il valore attuale del registro ESP, determinando in tal modo l'indirizzo base del record di attivazione attuale
    - lasciare in pila lo spazio per le variabili locali
  - epilogo
    - liberare nella pila lo spazio relativo alle variabili locali
    - ripristinare il contenuto del registro EBP
    - effettuare il ritorno al chiamante (RET)

Interfaccia del chiamato

- Prologo

```
pushl %ebp          #link dinamico
movl %esp, %ebp     #indirizzo base del record di attivazione attuale
subl $v, %esp       #spazio per le variabili locali (v byte)
```

- Epilogo

```
movl %ebp, %esp     # eliminazione spazio variabili locali
popl %ebp           # ripristino vecchio indirizzo base
ret                 # ritorno al chiamante
oppure
leave               # equivalente alle prime due istruzioni
ret
```

- I parametri vengono riferiti con

par_i = 8, 12, ..., per esempio

```
movl par_i(%ebp), %eax

movl par_i(%ebp), %ebx
movl (%ebx), %eax

movl par_i(%ebp), %ebx
movl %eax, (%ebx)
```

- Le variabili locali vengono riferite con var_j = -4, -8, ..., per esempio

```
movl %eax, var_j(%ebp)
```

Interfaccia del chiamante

- Esempi
  - i parametri attuali del chiamato sono variabili globali

```
pushl aa                    #valore di variabile globale aa intera

leal bb, %eax               #indirizzo di variabile globale bb intera
pushl %eax
ovvero
pushl $bb

call sott                   # chiamata di sottoprogramma
addl $n, %esp               # eliminazione spazio parametri (n byte)
```

  - i parametri attuali del chiamato sono variabili locali del chiamante

```
pushl par_i(%ebp)           # parametro valore intero trasmesso per valore

leal par_i(%ebp), %eax      # parametro valore intero trasmesso per indirizzo
pushl %eax

movl par_i(%ebp), %eax      # parametro indirizzo di intero
pushl (%eax)                # intero trasmesso per valore

pushl par_i(%ebp)           # parametro indirizzo di intero

pushl var_j(%ebp)           # variabile locale intera trasmessa per valore

leal var_j(%ebp), %eax      # variabile locale intera trasmessa per indirizzo
pushl %eax
```

Esempio 1

```
int alfa = 1, beta = 2, gamma = 5;
extern “C” int fai (int a, int b, int* c) {
        int d;
        // ...
        d = a + 1;
        *c = a + b;
        // ...
        return d;
}
int main(){
        int d;
        // ...
        d = fai(alfa, beta, &gamma);
        // ...
}
```

Esempio 1

```
#-------------------------------------------------
.data
alfa:           .long    1
beta:           .long    2
gamma:          .long    5
#-------------------------------------------------
.text
.global  fai
fai:        # a vale 8(%ebp)
            # b vale 12(%ebp)
            # c vale 16(%ebp)
            # d vale -4(%ebp)

            pushl    %ebp          # prologo
            movl     %esp, %ebp
            subl     $4, %esp

            pushl    %ebx
            # ...
            movl     8(%ebp), %eax
            incl     %eax
            movl     %eax, -4(%ebp)

                     movl     16(%ebp), %eax
                     movl     8(%ebp), %ebx
                     addl     12(%ebp), %ebx
                     movl     %ebx, (%eax)
                     # ...
                     movl     -4(%ebp), %eax
                     popl          %ebx

                     leave    # epilogo
                     ret
#-----------------------------------------
```

Esempio 1

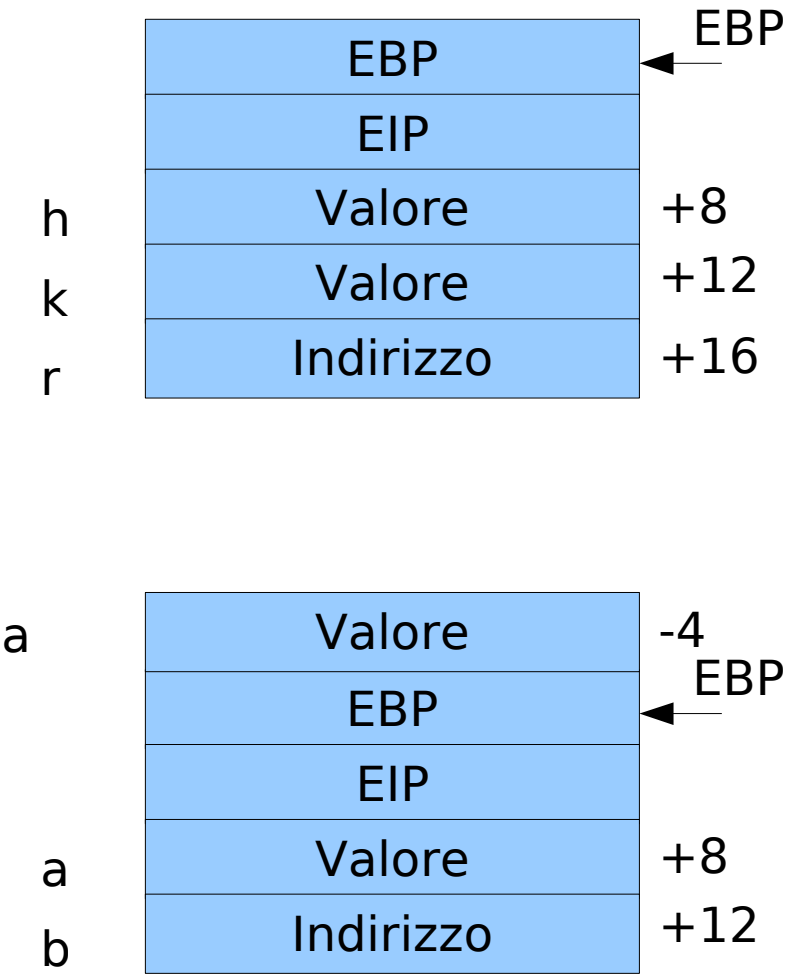
```
.global _main
_main:
        pushl   %ebp            # prologo
        movl    %esp, %ebp
        subl        $4, %esp

        # ...
        pushl   $gamma
        pushl   beta
        pushl   alfa
        call    fai
        addl    $12, %esp
        movl    %eax, -4(%ebp)
        # ...

        xorl    %eax, %eax
        leave                   # epilogo
        ret
#-----------------------------------------------
```

Esempio 2

```
int alfa = 5, beta = 6;
extern “C” void uno(int h, int k, int* r) {
        // ...
        *r = h + k;
        // ...
}
extern “C” void due(int a, int* b) {
        int aa;
        // ...
        *b = 10;
        uno(a, *b, &a);
        uno(a, *b, &aa);
        uno(a, aa, b);
        // ...
}
int main(){
        // ...
        due(alfa, &beta);
        // ...
}
```



Esempio 2

```
#-------------------------------------------------
.data
alfa:     .long    5
beta:     .long    6
#-------------------------------------------------
.text
.global  uno
uno:      # h vale 8(%ebp)
          # k vale 12(%ebp)
          # r vale 16(%ebp)

          pushl    %ebp          # prologo
          movl     %esp, %ebp

          pushl    %eax
          pushl    %ebx
          # ...
          movl     16(%ebp), %eax
          movl     8(%ebp), %ebx
          addl     12(%ebp), %ebx
          movl     %ebx, (%eax)
                                          # ...
                                          popl     %ebx
                                          popl     %eax

                                          leave    # epilogo
                                          ret
                                    #-----------------------------------------
```

Esempio 2

```
#-----------------------------------------------
.global  due
due:
        # a vale 8(%ebp)
        # b vale 12(%ebp)
        # aa vale -4(%ebp)

        pushl   %ebp
        movl    %esp, %ebp
        subl    $4, %esp

        pushl   %eax
        # ...
        movl    12(%ebp), %eax
        movl    $10, (%eax)

        leal        8(%ebp), %eax
        pushl   %eax
        movl    12(%ebp), %eax
        pushl   (%eax)
        pushl   8(%ebp)
        call    uno
        addl    $12, %esp

        leal    -4(%ebp), %eax
        pushl   %eax
        movl    12(%ebp), %eax
        pushl   (%eax)
        pushl   8(%ebp)
        call    uno
        addl    $12, %esp

        pushl   12(%ebp)
        pushl   -4(%ebp)
        pushl   8(%ebp)
        call    uno
        addl    $12, %esp
        # ...
        popl    %eax

        leave
        ret
```

Esempio 2

```
#---------------------------------------
.global _main
_main:  pushl   %ebp
        movl    %esp, %ebp

        pushl   $beta
        pushl   alfa
        call    due
        addl    $8, %esp

        xorl    %eax, %eax
        leave
        ret
#---------------------------------------
```