1 Programmazione mista C - Assembly

1.1.1 Assembly inline

Molti compilatori C hanno al loro interno anche un "piccolo" Assembler, detto Assembler "**inline**". Utilizzando questo Assembler è possibile scrivere istruzioni Assembly in qualsiasi punto del sorgente C o C++ e si possono utilizzare direttamente le variabili e le costanti definite in C. Frequentemente l'Assembler inline non è così potente come un Assembler completo, per cui sarà soggetto ad alcune limitazioni.
Ogni compilatore C ha il suo modo per indicare la presenza di una parte di codice scritta in Assembly, per cui sarà necessario consultare la sua documentazione per apprezzarne i dettagli.

Caratteristiche dell'Assembly inline del Turbo C++

In questo paragrafo illustreremo il funzionamento dell'Assembler inline del compilatore Turbo C++ (o Borland C++).
In Turbo C++ è molto semplice far capire al compilatore che un blocco di istruzioni è scritto in Assembly e non in C. Basta far precedere la parentesi graffa di inizio del blocco dalla parola chiave "asm". Tutto quello che c'è fra "asm {" e la parentesi graffa di fine blocco verrà considerato Assembly e compilato come tale.
Come esempio si prenda il seguente codice:

```
#include <iostream.h>
unsigned int A;
int B;
main()
{
A = 65534;       // il numero massimo a 16 bit senza segno
    B = 0x7FFF;      // il numero massimo a 16 bit con segno

  asm {  // ! asm e la parentesi graffa devono stare nella stessa linea !
    ADD word ptr [A], 1     // somma a 16 bit, A diviene 65535
    ADD byte ptr [B], 1     /* operazione SBAGLIATA a 8 bit, produce
        un carry = 1. Nella variabile B finisce il numero 8000h e non
        viene segnalato NESSUN ERRORE */
      }
    cout << " A = " << A << " B = " << B;
    A++;  /* in A c'era 65534 + 1 (per la ADD). Se aggiungo ancora 1
          si fa un errore. Di solito programma compilato in C non si accorge
          di questo errore. E' peraltro possibile che il compilatore
          abbia un'opzione per generare un programma che fa il controllo
          degli errori aritmetici a tempo di esecuzione.
          Qualora questa opzione sia abilitata il C SI PUO' ACCORGERE DELL'ERRORE
          e il programma può essere bloccato!
          Nella parte ASM l'errore non può essere MAI rilevato automaticamente */
    cout << " A dopo " << A;
  return 0;
}
```

Le variabili del C corrispondono ad indirizzi e possono essere usate come tali anche nella parte Assembly inline. Si noti comunque che nella parte Assembly non si fa il controllo dei tipi per cui possono essere eseguite operazioni che in C sarebbero rifiutate perché pericolose od errate.

Per evitare problemi è meglio indicare sempre il parallelismo delle operazioni (word o byte ptr), perché il compilatore C fa assunzioni non sempre prevedibili.
La limitazione principale dell'Assembler inline del Turbo C++ è che non può accettare label. Le uniche label cui può puntare un'istruzione Assembly inline sono quelle del C. Anche se non sono molto "pubblicizzate" il C ammette le label e l'istruzione goto, che è come una jump incondizionata. La scrittura di un'etichetta ha in C la stessa sintassi dell'Assembly: un nome simbolico seguito dal duepunti.
Vediamo un di programma con una loop, che necessita di un'etichetta:

```
#include <iostream.h>;
unsigned int A[2];
main()
{
  A[0] = 0;  A[1] = 1;  A[2] = 2;
  // aumenta di uno tutti gli elementi del vettore:
  asm {
        MOV CX, 3
        MOV BX, 0
      }
```

```
UnaLabelDelC:
  asm {
        INC WORD PTR [A + BX]
        ADD BX , 2      // vado al prossimo elemento del vettore
        LOOP UnaLabelDelC
      }
  cout << " A[0] = " << A[0] << " A[1] = " << A[1] << " A[2] = " << A[2];
  return 0;
}
```

Altre regole e limitazioni dell'Assembler inline del Turbo C sono le seguenti:
- Dall'ambiente Turbo C++ non è possibile ottenere un file eseguibile. Si può ottenere un eseguibile solo compilando dalla linea di comando con TCC (vedi oltre)
- Nei blocchi asm non si possono usare direttive che allocano la memoria (DB, DW, ..)
- Dato che il segno ";" in C conclude l'istruzione, non si può usare come segno di commento, anche nei blocchi asm {. Per i commenti si dovrà utilizzare ovunque la sintassi dei commenti del C o del C++, anche nella parte Assembly. Ciò era già stato evidenziato dal programma precedente.
- Alla fine di ogni blocco Assembly inline i registri BP, SP, CS, DS e SS devono contenere gli stessi valori che avevano all'inizio del blocco. Gli altri registri, ES compreso, possono essere usati liberamente.
- L'ottimizzazione della parte C può essere influenzata negativamente dalla presenza di blocchi asm {.

Queste limitazioni dell'Assembly inline lo rendono scomodo da usare per programmi solo minimamente complessi. Quando la parte in Assembly è impegnativa è meglio confinarla in un modulo separato, che conterrà procedure da chiamare da C e che verrà collegato al programma C tramite il linker.

Compilazione Turbo C dalla linea di comando

Il programma TCC (Turbo C Compiler) compila il programma richiesto, file di include compresi, ed usa il linker per collegare gli .OBJ ottenuti con le librerie del C. Il processo di linking è svolto in modo automatico dal programma TCC, per cui si ottiene direttamente un file .EXE.
Per esempio, per ottenere l'eseguibile dal sorgente programm.cpp, si deve scrivere:

```
C:> tcc -v programm.cpp
```

nell'esempio si è indicata l'opzione –v, che fa includere nell'eseguibile le informazioni simboliche per il debugger. Se –v è abilitata sarà possibile fare il debugging direttamente dal sorgente, nel modo consueto:

```
C:> td programm
```

Un'opzione interessante del Turbo C++ permette di compilare in Assembly invece che in linguaggio macchina; l'esecuzione di:

```
C:> tcc -S programm.cpp
La S deve essere maiuscola
```

produce il file programm.asm. Esso è un sorgente Assembly che mostra come il C ha tradotto le sue istruzioni in linguaggio di basso livello. Può essere modificato "a mano" e compilato con TASM.
E' interessante studiare come il C traduce le istruzioni, per ampliare la propria cultura e per capire come poter ottimizzare i programmi che il C produce.

Caratteristiche dell'Assembler inline del MS Visual C++

Anche Visual C++ ammette l'uso dell'Assembly inline, con minori limitazioni rispetto a Turbo C++.
La più importante differenza è che Visual C++ è un compilatore per CPU X86 a 32 bit, dall'80386 in poi, e perciò funziona con la CPU in "protected mode"(*), ed utilizza i nuovi registri e le nuove istruzioni a 32 bit.
Il fatto che la CPU sia a 32 bit si riflette anche nella dimensione dei tipi delle variabili C, come illustrato nella seguente tabella:

Tipo	Turbo C e C++ Per X86 16 bit	Visual C e C++ per X86 32 bit	GNU C (gcc) per Linux (ed altre)
char	Intero **8** bit con segno	Intero **8** bit con segno	Intero **8** bit con segno
unsigned char	Intero 8 bit senza segno	Intero 8 bit senza segno	Intero 8 bit senza segno
short int	Intero **16** bit con segno	Intero **16** bit con segno	Intero **16** bit con segno
unsigned short int	Intero 16 bit senza segno	Intero 16 bit senza segno	Intero 16 bit senza segno
int	Intero **16** bit con segno	Intero **32** bit con segno	Intero **32** bit con segno

long int	Intero **32** bit con segno	Intero **32** bit con segno	Intero **32** bit con segno
float	Virgola mobile 32 bit	Virgola mobile 32 bit	Virgola mobile 32 bit
double	Virgola mobile **64** bit	Virgola mobile **64** bit	Virgola mobile **64** bit
long double	Virgola mobile **96** bit	Virgola mobile **64** bit	Virgola mobile **96** bit

Tabella 1 Dimensioni dei tipi in Turbo C++ (16 bit) e Visual C++ (32 bit)

(*) per i dettagli sul funzionamento in protected mode vedere in altro luogo di questo libro.

Il fatto che il C abbia variabili di dimensioni diverse in dipendenza del compilatore, anche se hanno lo stesso tipo, è visto da alcuni come una caratteristica che ne migliora la portabilità.
- I tipi numerici del C hanno numero di bit diversi in dipendenza dall'architettura del computer su cui sono usati. Questo "costringe" i programmatori a scrivere applicazioni che sono indipendenti dalla rappresentazione dei numeri, dato che non ci si può fidare di come sono implementati.
Quando invece si lavora a basso livello, come accade spesso nella programmazione per la grafica, può essere importante sapere con quanti bit sono rappresentati i numeri. In questo caso sarebbe meglio avere tipi di dati che mantengano costante il loro numero di bit cambiando piattaforma.
Questo problema si risolve facilmente definendo tipi appositi e non usando quelli tipici del C. Infatti applicazioni od ambienti di sviluppo hanno librerie che includono la definizione in linguaggio C di tipi numerici "portabili", che hanno lo stesso numero di bit in tutte le piattaforme. (*)

(*) Come esempio si può citare la libreria GTK, usata nella programmazione grafica in Linux; essa ha il tipo "gint" intero con segno a 16 bit in ogni piattaforma e il tipo "gint32" che è sempre di 32 bit.

L'Assembly inline del Visual C++ ha le seguenti caratteristiche:
- Accetta tutte le espressioni che sarebbero legali in MASM
- Il blocco asm ha inizio con la stringa "__asm {" (due segni di sottolineatura, asm ed una graffa aperta) e si conclude con una graffa chiusa
- Si possono usare label all'interno del blocco asm, ma bisogna fare molta attenzione a che queste etichette non abbiano il nome di funzioni del C, perché altrimenti il programma potrebbe saltare a quelle funzioni
- Da un blocco asm si può saltare a qualsiasi etichetta, sia C che Assembly
- I numeri esadecimali possono essere scritti sia con la notazione del C (es. 0xFFFF), sia con la notazione dell'Assembly (es. 0FFFFh)
- Non si possono usare direttive di allocazione della memoria (define)
- I commenti dentro un blocco asm possono essere sia "stile C" che "stile Assembly"
- I registri EAX, ECX, EDX possono essere usati liberamente dentro un blocco asm, senza alcuna limitazione
- Se si usano dentro un blocco asm i registri EBX, ESI o EDI, il C li salverà prima di entrarci e li ripristinerà all'uscita; per questo la velocità di esecuzione del programma sarà leggermente penalizzata
- EBP viene usato dal C, per cui se lo dovessimo usare dentro un blocco asm dovremmo salvarlo e ripristinarlo prima di chiudere il blocco
- Tutti gli altri registri non devono esser modificati all'uscita di un blocco asm. Se li si vuole usare è indispensabile salvarli all'inizio e ripristinarli prima della fine del blocco.
- Per fare procedure con l'Assembly inline bisogna scrivere in C tutta la parte relativa alla definizione del prototipo ed al passaggio dei parametri alla procedura; poi all'interno della procedura di aprirà un blocco asm, nel quale i parametri saranno usati simbolicamente.
Per esempio:

```
; POWER.ASM
; Compute the power of an integer
;
       PUBLIC _power2
_TEXT SEGMENT WORD PUBLIC 'CODE'
_power2 PROC

        push ebp        ; Save EBP
        mov ebp, esp    ; Move ESP into EBP so we can refer
                        ;   to arguments on the stack
        mov eax, [ebp+4] ; Get first argument
```

```
        mov ecx, [ebp+6] ; Get second argument
        shl eax, cl      ; EAX = EAX * ( 2 ^ CL )
        pop ebp          ; Restore EBP
        ret              ; Return with sum in EAX

_power2 ENDP
_TEXT   ENDS
        END
```

Diventa:

```
/* POWER2.C */
#include <stdio.h>
int power2( int num, int power );
void main( void ) {
  printf( "3 times 2 to the power of 5 is %d\n", \ power2( 3, 5) );
 }
int power2( int num, int power ) {
  __asm { mov eax, num ; Get first argument
          mov ecx, power ; Get second argument
          shl eax, cl ; EAX = EAX * ( 2 to the power of CL )
   } /* Return with result in EAX */
}
```

1.1.2 Chiamata dal C di procedure scritte in Assembly

Quando si devono scrivere applicazioni C che facciano uso di parti in Assembly complesse è possibile ricorrere al collegamento di moduli separati, compilati con linguaggi diversi.
In genere in questo caso il programma Assembly sarà costituito da una collezione di procedure, senza un programma principale.
Le procedure Assembly che devono essere utilizzate in un programma C si devono comportare esattamente come se fossero scritte in C.
In particolare il passaggio dei parametri, sia in ingresso che in uscita, ed i nomi simbolici utilizzati dovranno rispettare con precisione le specifiche del compilatore C utilizzato.

L'impostazione tipica dei programmi multilinguaggio in C e Assembly consiste nello scrivere il programma principale in C e collegarlo a moduli scritti in Assembly chiamati dal C attraverso funzioni.
Le procedure scritte in Assembly si dovranno comportare come se fossero state scritte in C, per cui è importante capire bene come il C chiama le sue funzioni, come passa ad esse i parametri e come ne utilizza i risultati.
Il linguaggio C non possiede istruzioni diverse per funzioni e procedure, come succede in Pascal e BASIC, ma ha solo funzioni, anche se quelle definite come "void" sono di fatto procedure, non producendo valori in ritorno.

Il C passa i parametri alle procedure sempre e solo attraverso lo stack e riceve il risultato delle funzioni sempre e solo attraverso registri.

Il risultato della funzione viene ritornato nel registro AL se è della lunghezza di un byte (p.es char), nel registro AX se è di 16 bit (int), in DX e AX se è di 32 bit (long), con la parte bassa in AX e l'alta in DX.
Nella maggioranza dei compilatori C e C++ i nomi simbolici delle funzioni e delle variabili globali vengono salvati nell'OBJ mantenendo le maiuscole e minuscole ed anteponendo al simbolo il carattere "_" (sottolineatura, underscore).

Questo non è sempre vero perché p.es. le specifiche ELF per i file OBJ prevedono che i nomi rimangano immutati (il formato ELF è usato dai compilatori freeware GNU, nel S.O. Linux).

Una funzione restituisce un valore del tipo dichiarato nel suo "**prototipo**". Nel prototipo della funzione sono indicati anche il numero ed il tipo dei parametri di ingresso della funzione. In generale il prototipo di una funzione in Turbo C++ si esprime in questa forma:

```
[extern] ["C" | "Pascal"] <tipo del valore di ritorno> [near | far] <nome della funzione> ( <lista dei parametri> );

<lista dei parametri> := <tipo del parametro> [far] [*]<nome del parametro>, ..
```

Il segno .. significa che ciò che lo precede può essere ripetuto a piacimento.

La clausola extern, opzionale, indica che la funzione non è definita all'interno del progetto C corrente, ma che deve essere collegata dal linker. extern si dovrà usare se la funzione è realizzata in un modulo Assembly separato.
La clausola far, opzionale, indica che il riferimento deve essere a 20 bit segmentato, composto da 16 bit di offset e 16 bit di segmento. Ciò vale sia per i riferimenti al codice, per i quali il C genererà una call far, che per i riferimenti a locazioni di dati, per le quali il C passerà sia l'indirizzo di offset che quello di segmento.
La presenza, opzionale, dell'asterisco prima del nome del parametro significa che deve essere passato l'indirizzo invece che il valore del parametro. Al posto di <nome del parametro> ci può anche essere una costante.

Un programma in C passa i suoi parametri in due modi: per valore o per indirizzo.
Il **passaggio per valore** è quello standard, che si usa sempre perché più "sicuro". In questo caso il programma C copia nello stack il valore corrente di tutti i parametri indicati, poi fa la chiamata alla procedura. Dunque il C, prima di fare una "CALL" all'indirizzo della procedura, farà una serie di "PUSH".
L'ordine con il quale i parametri vengono messi nello stack è dall'ultimo della lista al primo. Usando questo ordine, che può sembrare strano, è più facile realizzare funzioni che abbiano un numero variabile di parametri.

Infatti, dato che lo stack funziona in modo LIFO, il parametro che "emerge" dallo stack prima della chiamata è il primo della lista del prototipo, cioè l'ultimo che è stato messo nello stack. Se al momento della chiamata manca qualcuno dei parametri lo stack sarà "più vuoto", ma i parametri che ci sono avranno sempre la stessa posizione relativa. Ciò non accadrebbe se i parametri fossero passati nello stack in ordine dal primo.

In Turbo C è possibile obbligare il compilatore a scrivere i parametri nello stack nell'ordine con cui sono scritti nella lista dei parametri formali. Per ottenere questo effetto basta scrivere "pascal" (fra virgolette) nel prototipo della funzione, prima della dichiarazione del suo tipo
Esempio:

```
extern "pascal" int ProceduraScrittaInPascal (int LunicoParametro) ;)
```

Vediamo ora un esempio che illustra come il C passa i parametri per valore. Mostreremo un programma in C e la sua traduzione in Assembly, ottenuta usando l'opzione -S del compilatore:

Programma in C:

```
int prova(int nparam, char CercaQuesto, long int DaElaborare);
int P, R;
main()
{
    /* passa a prova il valore di una variabile di 16 bit, il carattere A
       ed una costante di 32 bit, scritta con la notazione del C per i numeri
       esadecimali */
    R = prova(P, 'A', 0xA0B0C0D0);
}
```

come già visto nel primo volume in Turbo C++, compilatore per X86 da 16 bit, il tipo int ha una lunghezza di 2 byte, char è da 8 bit e long da 32. Date queste "dimensioni" per i parametri si spiegano le push che compaiono nel codice seguente:

Traduzione in Assembly del brano di programma C:

```
    mov ax, 41136    ; 16 bit alti del numero di 32 bit (A0C0h)
    mov dx, 49360    ; 16 bit bassi del numero di 32 bit (C0D0h)
    push ax          ; parte alta nello stack
    push dx          ; parte bassa nello stack
    mov al, 65       ; 65 è il codice ASCII di "A"
    push ax          ; push di 'A', in AL (anche push di AH!)
    push word ptr [_P]       ; push del valore corrente di P
    call near ptr _prova     ; chiamata alla procedura
    add sp,8                 ; sistemazione dello stack dopo il ritorno
    mov word ptr [_R], ax    ; assegnazione del risultato a R
/* (nota: il codice prodotto dal compilatore è stato leggermente modificato,
    per renderlo più comprensibile) */
```

Si noti come i riferimenti alle variabili P e R siano cambiati in _P e _R.

Le prime righe mettono nello stack la costante 0xA0B0C0D0, numero di 32 bit espresso in esadecimale.
La push AX, che viene dopo aver messo in AL il codice ASCII di A, passa 'A' alla procedura. Si noti che viene messo nello stack anche AH (push AX), perché con un 8086 le PUSH possono essere solo a 16 bit. Peraltro AH assumerà un valore non determinato, dato che non viene modificato dal codice che prepara i parametri per la procedura.
La PUSH successiva mette nello stack il valore corrente della variabile P prelevandolo dalla locazione di memoria che gli è stata riservata (push word ptr [_P]).
La situazione dello stack quando la procedura _prova prende il controllo è dunque la seguente:

IP di ritorno		SP	
CS di ritorno		SP + 2	
Valore corrente di P		SP + 4	variabile P
Non usato	'A'	SP + 8	costante char
C0h	D0h	SP + 10	costante
A0h	B0h	SP + 12	di 32 bit

(memoria rappresentata a word, parte alta della word a sinistra)

Dopo il ritorno della procedura sono due le cose importanti da notare, che si evidenziano nelle due istruzioni dopo la CALL.
La prima istruzione (add sp,8) ci spiega come il C aggiusta lo stack dopo averlo usato per passare i parametri alla procedura: lo rimette a posto "gettando via" tutti i dati che aveva passato. Dopo che la procedura è ritornata aggiunge allo stack pointer il numero di byte usati come parametri, nell'esempio erano otto. Ciò toglie effettivamente i parametri dallo stack, anche se non li cancella dalla memoria. Infatti i vecchi valori non potranno essere più letti con pop perché queste istruzioni lavoreranno al nuovo indirizzo puntato da SP e verranno fisicamente sovrascritti con le prossime PUSH.

Esaminiamo ora il **passaggio per indirizzo**. Esso si usa quando si vuole che la procedura lasci delle modifiche permanenti alle variabili che vengono passate come parametri, modificando il contenuto delle locazioni di memoria corrispondenti. Si usa il passaggio per indirizzo anche quando il parametro è troppo "ingombrante" per essere passato attraverso lo stack, come è il caso degli array e delle stringhe.
La sintassi per ottenere in C un passaggio per indirizzo consiste nel dichiarare un puntatore nella lista dei parametri del prototipo, anteponendo un asterisco (star) al suo nome: ***<nome del parametro formale>.
Nella chiamata effettiva bisogna invece utilizzare l'indirizzo della variabile che si vuole passare, anteponendo al suo nome una e commerciale (ampersand): **&**<nome del parametro effettivo>.
Vediamo un esempio di dichiarazione ed utilizzazione di una funzione che passa un parametro per indirizzo:

```
// Nome del file che contiene questo programma: WIPE&LEN.CPP
int far WipeANDlen(char far *stringa);
char LaDevoCancellare[] = "12345678";
  main()
  {
     cout << LaDevoCancellare << ' ';
        /*  WipeANDlen conta il numero dei caratteri della stringa che
            viene passata e mentre conta la cancella, riempiendola
             di caratteri blank (" " ASCII = 32) */
     cout << WipeANDlen(LaDevoCancellare); (*)
     cout << LaDevoCancellare;
     return 0;
  }

(*) nella chiamata non si mette & prima di LaDevoCancellare perché essa è una stringa,
che viene passata comunque per indirizzo.
```

Si noti che la definizione del prototipo comprende la clausola "far", che fa compilare una CALL FAR.
La dichiarazione del puntatore è far (char far *stringa), ciò significa che nello stack viene memorizzato sia l'offset che il segmento dell'indirizzo del primo carattere della stringa.

Vediamo come viene tradotto questo programma:

```
    push    ds                  ; passaggio del segmento della stringa (è in DS)
    ; passaggio dell'offset della stringa:
    mov     ax, offset [_LaDevoCancellare]
    push    ax
```

```
    call    far ptr _WipeANDlen  ; chiamata della funzione
    pop     cx                   ; pulizia di una word dallo stack
    pop     cx                   ; pulizia dell'altra word dallo stack
    mov     word ptr [bp-2],ax   ; salvataggio del valore di _WipeANDlen
                                 ; nell'area di memoria delle variabili automatiche
```

Anche in questo caso i nomi simbolici sono diventati _WipeANDlen e _LaDevoCancellare, avendo acquistato una sottolineatura all'inizio.
La situazione dello stack quando la procedura _WipeANDlen prende il controllo è dunque la seguente:

IP di ritorno	SP	
CS di ritorno	SP + 2	
offset di _LaDevoCancellare	SP + 4	variabile passata per
segmento di _LaDevoCancellare	SP + 6	Indirizzo

(memoria rappresentata a word, parte alta della word a sinistra)

Collegamento al C di procedure scritte in Assembly

Le regole stabilite dal compilatore C per chiamare le sue procedure devono essere rispettate da un programma Assembly che deve essere usato dal C, per cui i nomi usati nel modulo Assembly dovranno essere uguali a quelli del C, anche come maiuscole e minuscole, con l'unica differenza che inizieranno con una sottolineatura perché, come visto, il C l'inserisce all'inizio di tutti gli identificatori.

Ora che sappiamo come il C chiama le sue stesse funzioni vediamo come si possono scrivere procedure Assembly che si comportano come se fossero scritte in C.

Prendiamo come esempio la procedura WipeANDlen, definita nell'ultimo esempio. Essa deve cancellare completamente la stringa che le viene passata, sostituendone tutti i caratteri e contemporaneamente restituire come integer la lunghezza della stringa stessa.

I registri AX, BX, CX, e DX possono essere usati liberamente all'interno della procedura Assembly perché se il Turbo C ne ha bisogno li salva prima di chiamare. Gli altri registri devono essere salvati e ripristinati prima della RET.

Per prima cosa è necessaria una modifica al prototipo della funzione in C. Visto che il codice della funzione non è più presente in forma di sorgente C si dovrà dichiarare al compilatore che esso è esterno, con una direttiva "extern" (da notare la e sottolineata, che non è presente nella analoga direttiva del TASM).
Se WipeANDlen viene dichiarata extern il compilatore non segnalerà errori quando non la troverà ed essa potrà essere collegata in un secondo tempo dal linker.
Dunque il prototipo della funzione WipeANDlen diverrà il seguente:

```
// Nome del file che contiene questo programma: WIPE&LEN.CPP
..
extern "C" int far WipeANDlen(char far *stringa);
..
```

Tutto il resto del programma in C rimane uguale.

Segue il testo del modulo Assembly, che dovrà essere compilato e collegato a quello in C:

```
; Nome del file che contiene questa procedura: STRINGFN.ASM

PUBLIC _WipeANDlen PROC FAR
  ; prologo della procedura: salvataggio del contesto
  PUSH BP       ; salva BP 'che punta all'heap del C
  PUSH DS       ; salva DS perché il puntatore che viene passato è far
                ; e non è detto che la stringa che mi viene passata sia
                ; nello stesso segmento del DS corrente
  MOV BP, SP
  ; Carica il puntatore alla stringa, che è di tipo FAR:
  MOV BX, [BP + 8]      ; l'offset
    ; il registro di segmento:
  MOV AX, [BP + 10]
  MOV DS, AX
  ; cancella il conteggio dei caratteri in AX:
  XOR AX, AX
  ; mette spazi in tutta la stringa, fino a che non trova il null,
  ; contemporaneamente conta con AX:
PerTuttaLaStringa:
  MOV byte ptr [BX], " "        ; mette un blank
  INC BX                        ; incrementa il puntatore nella stringa
  INC AX                        ; incrementa il contatore di caratteri della stringa
  CMP byte ptr [BX], 0
```

```
  JNZ PerTuttaLaStringa
  ; quando esce di qui in AX c'è il numero di caratteri
  ; (è già a posto perché la funzione lo deve restituire in AX)
  ; al posto del loop precedente si poteva usare una istruzione di stringa,
  ; quale la LODSB
  ; epilogo della procedura: ripristino di quanto salvato inizialmente:
  POP DS
  POP BP                ; ripristina il BP del C
  RET
_WipeANDlen ENDP

;Nota: la procedura non è ottimizzata, per farlo si potrebbero
;      usare le funzioni di stringa
```

Si noti che il nome che si deve utilizzare per la procedura è _WipeANDlen.
Si deve inoltre fare in modo che essa sia visibile dall'esterno, come specificato dalla direttiva PUBLIC, in modo che il linker possa collegare questo programma Assembly al programma principale in C.
La procedura salva subito DS e BP nello stack, per cui la situazione dello stack cambia rispetto a quanto visto prima: sopra ad IP di ritorno ci sono ora BP e DS, come illustrato dal seguente schema:

BP del chiamante	SP	
DS del chiamante	SP + 2	
IP di ritorno	SP + 4	
CS di ritorno	SP + 6	
Offset di _LaDevoCancellare	SP + 8	Variabile passata per
Segmento di _LaDevoCancellare	SP + 10	Indirizzo

Questa procedura è un esempio di quanto possa essere utile collegare al C procedure Assembly. Con poco sforzo abbiamo ottenuto una procedura che svolge "contemporaneamente" due funzioni, cancella una stringa e ne conta anche il numero di caratteri. Usando le librerie standard del C o del C++ non è possibile fare entrambe le istruzioni in una sola volta e bisogna chiamare due funzioni diverse, con la perdita di efficienza del programma che ne consegue.
Una volta terminati i programmi per compilarli e collegarli si usa TCC:

```
C:> tcc WIPE&LEN STRINGFN.ASM
```

Lanciando TCC con questi parametri entrambi i programmi vengono compilati, uno dal C compiler e l'altro da TASM; poi viene automaticamente chiamato il linker che collega i due OBJ appena generati e gli eventuali altri OBJ di libreria. Il risultato è il file eseguibile WIPE&LEN.EXE.
Se si vuole effettuare un debugging simbolico si può compilare con l'opzione -v prima del primo nome di file e poi usare il Turbo Debugger (TD WIPE&LEN).
Anche in questo caso si possono compilare i moduli C in modo che vengano resi in Assembly, usando l'opzione -S.

Curiosità

una canzone da "The Embedded Muse", di Jack Ganssle, da cantare sul motivo di "Let it be" dei Beatles:

Write in C (Sing to the Beatle's tune "Let it Be")

When I find my code in tons of trouble,
Friends and colleagues come to me,
Speaking words of wisdom:
"Write in C."

As the deadline fast approaches,
And bugs are all that I can see,
Somewhere, someone whispers:
"Write in C."

Write in C, Write in C,
Write in C, oh, Write in C.
LOGO's dead and buried,
Write in C.

I used to write a lot of FORTRAN,

For science it worked flawlessly.
Try using it for graphics!
Write in C.

If you've just spent nearly 30 hours,
Debugging some assembly,
Soon you will be glad to
Write in C.

Write in C, Write in C,
Write in C, yeah, Write in C.
BASIC's not the answer.
Write in C.

Write in C, Write in C
Write in C, oh, Write in C.
Pascal won't quite cut it.
Write in C.