

Appunti del corso di Linguaggi I

Appunti raccolti durante il corso di *Linguaggi I* tenuto dal professore **Fabio De Luigi** per il *Corso di Laurea in Informatica* presso *l'Università degli Studi di Ferrara*.

Gli appunti raccolti in questo opuscolo sono stati trascritti e riorganizzati da *Farinelli Agnese*.



Assembler INTEL

Esistono diversi tipi di dato:

- Tipo BYTE → 8 bit
- Tipo WORD → 16 bit
- Tipo DWORD → 32 bit

Guardiamo ora alcune istruzioni in linguaggio *assembler*. Le istruzioni di seguito espresse sono tutte dichiarate nel **SEGMENTO dei DATI** (DATA SEGMENT):

ISTRUZIONI	SIGNIFICATO
BYTEVAR DB 7	Creo una variabile di tipo BYTE alla quale assegno il valore 7. Il nome della variabile in questione è BYTEVAR.
BYTEARR DB 1, 2, 3, 4	Creo una sorta di simulazione di array di tipo BYTE nel quale inserisco i valori 1, 2, 3, 4 nell'ordine in cui sono stati esplicitati. Il nome della variabile è BYTEARR.
STRINGA ₁ DB '8','0','8','6' (*)	Non creo propriamente una stringa ma la simulo, come nel caso dell'array. La "stringa" è di tipo BYTE e contiene all'indirizzo STRINGA ₁ la codifica ASCII dei caratteri descritti tra apici ('.'). Posso scrivere la stessa cosa anche nei seguenti modi, tutti equivalenti tra loro: <ul style="list-style-type: none">• STRINGA₂ DB '8086'• STRINGA₃ DB 56, 48, 56, 54 Nell'ultimo esempio, 56 è la codifica ASCII del carattere '8' e la forma di tale istruzione ricalca perfettamente quella dell'array simulato di cui sopra.
TITOLO DB 'TITOLO', ODH, OAH	Creo la stringa TITOLO. Con le diciture ODH comunico che durante la stampa desidero "spostare il carrello della <i>macchina da scrivere</i> " mentre con OAH desidero "spostarmi in basso di una riga".
MOLTIZERI DB 256 DUP o CR EQU ODH LF EQU OAH TITOLO DB 'PIPPO',CR,LF	Desidero inizializzare 256 indirizzi di memoria a o. Questa dicitura equivale alla DEFINE in C. I parametri ODH e OAH sono quindi associati e "definiti" come, rispettivamente, CR e LF che possono essere usati al loro posto in istruzioni successive.

(*) In generale io non posso sapere quanto è lunga la stringa che vado a scrivere:

ASSEMBLER-C	PASCAL
56	4
48	56
56	48
54	56
/o	54

- in C leggo la stringa fino al riconoscimento del carattere designato a chiudere genericamente ogni stringa definita tale. Il carattere in questione è uno zero chiamato NULL-TERMINATED.
- in PASCAL prima di leggere i valori contenuti nella stringa leggo immediatamente il valore corrispondente alla quantità di elementi della stringa, che viene memorizzato sempre all'inizio della stringa stessa.

La sintassi assembler va letta da destra a sinistra. Le istruzioni si trovano nella parte di memoria dove è salvato tutto il resto del codice. Il linguaggio assembler è atto a gestire le funzioni del processore.

Altre istruzioni sono:

ISTRUZIONI	SIGNIFICATO
INT 2iH	Classica chiamata ad una libreria di sistema.
MOV AX, CX	OPERANDO REGISTRO - Muovo da il contenuto di CX in AX.
MOV AX, 100	OPERANDO IMMEDIATO - Salvo in AX il dato immediato 100, quindi carico una costante immediata nel registro AX.
PIPPO DW 15	ASSEGNAZIONE DIRETTA - Al nome PIPPO accosto la costante

MOV AX, PIPPO	15 di tipo WORD. INDIRIZZAMENTO DIRETTO - Muovo nel registro AX il contenuto dell'indirizzo PIPPO dopo averlo cercato, trovato e copiato.
----------------------	--

INDIRIZZAMENTO INDIRETTO MEDIANTE REGISTRO

```
MOV BX, OFFSET PIPPO
MOV AX,[BX]
```

OFFSET è la distanza da PIPPO dell'inizio del segmento.

(1) OFFSET va in BX.

(2) Metto in AX l'oggetto puntato da BX, in questo caso proprio l'OFFSET.

In generale, tra parentesi [...] non possono essere inseriti tutti i registri, ma solamente BX, BP, SI, DI.

INDIRIZZAMENTO INDIRETTO MEDIANTE REGISTRO BASE

```
MOV AX, [BX+4]
```

Sommo una costante al puntatore al contenuto di BX. Accedo così ad elementi strutturati.

In C, troviamo un caso equivalente nel funzionamento delle *strutture*:

```
TYPDEF STRUCT {
    INT GIORNO, MESE, ANNO;
} DATATYPE;
DATATYPE DATA1, DATA2;
```

Infatti, l'indirizzo (DATA1.GIORNO) è memorizzato al posto [DATA1+0]; l'indirizzo (DATA1.MESE) al posto [DATA1+2] e l'indirizzo (DATA1.ANNO) al posto [DATA1+4].

INDIRIZZAMENTO INDIRETTO MEDIANTE REGISTRO INDICE

```
MOV DI, 4
MOV AX, [TABLE+DI]
```

In questo caso posso usare solo i registri DI e SI che si chiamano INDEX come quelli dei vettori.

INDIRIZZAMENTO INDIRETTO MEDIANTE REGISTRO BASE E INDICE

```
MOV AX, [DISP+BX+SI]
```

Ideale per le matrici.

INDIRIZZAMENTO STRINGHE

```
DS:SI
ES:SI
```

Le istruzioni assembler per le *operazioni aritmetiche* sono:

ISTRUZIONI	SIGNIFICATO
DIV BX	DIVISIONE - Succede questo: <ul style="list-style-type: none"> • AX ← AX diviso BX • DX modulo BX
ADD BX, DX	ADDIZIONE - BX ← BX addizione DX

Se può verificarsi un overflow è necessario controllare con un *flag* del tipo **JO LABELi**.

INC DI	INCREMENTO - DI ← DI addizione 1
DEC DI	DECREMENTO - DI ← DI sottrazione 1

Il linguaggio assembler utilizza spesso dei registri che non vengono dichiarati e possiede istruzioni che per funzionare hanno bisogno solo ed esclusivamente di determinati registri... queste sono alcune delle sue imperfezioni. Da questo punto di vista si tratta di un linguaggio poco pulito.

ALTRE COSE...

OR DI, DI → ?

JZ LABEL → Jump on zero label

LEA BX, TABLE equivale a dire **MOV BX, OFFSET TABLE**

Analisi di un frammento di codice

Diamo una prima occhiata a come è organizzato il codice nel segmento **CODE SEGMENT**.

Innanzitutto, come un qualsiasi altro sorgente il programma **.ASM** viene compilato in un file eseguibile **.EXE**. Durante la compilazione vengono caricati i *registri di segmento* che vengono specificati nel codice. Il programma che analizziamo ora è il seguente:

```
;  
;      Program Proval  
;  
;      TITLE   Programma campione  
  
;***** Stack  
MyStack SEGMENT PARA STACK 'STACK'      ; declare stack segment  
        DB      64 DUP (?)                ; reserve 64 bytes  
MyStack ENDS                               ; end of stack segment  
  
;***** Dati  
MyData  SEGMENT PARA PUBLIC 'DATA'       ; declare data segment  
Hello1  DB      'Hello1! $'              ; define string to display  
Hello2  DB      'Hello2! $'              ; define string to display  
MyData  ENDS                               ; end of data segment  
  
;***** Codice  
MyCode  SEGMENT PARA PUBLIC 'CODE'       ; declare code segment  
ASSUME  CS:MyCode, DS:MyData, SS:MyStack  
  
;***** routine che stampa 1 carattere ASCII in dl  
stampa proc  
    push ax  
    push dx  
    mov  ah, 2  
    int  21h  
    pop  dx  
    pop  ax  
    ret  
stampa endp  
  
;***** routine che stampa gli ASCII per il binario in Ax  
asciitobin proc  
  
    push dx  
    push si  
    push ax  
    push di  
  
    mov di, 0 ; di conta i caratteri da stampare  
    mov si, 10
```

```

again:

    ; AX <- (DX:AX) div SI
    ; DX <- (DX:AX) mod SI
    mov dx, 0
    div si

    add dx, '0'
    ;call stampa

    ; metto ASCII (in dl) da stampare sullo stack e lo conto
    push dx
    inc di

    ; verifico se AX==0
    or ax, ax
    jnz again

    ; ora sullo stack ho in ordine inverso di caratteri da stampare
    ; li tolgo uno ad uno e li stampo
loopdistampa:
    or di, di
    jz fineascii
    pop dx
    dec di
    call stampa
    jmp loopdistampa

fineascii:
    ; stampo <CR> e <LF>
    mov dl, 0dh
    call stampa
    mov dl, 0ah
    call stampa

    pop di
    pop ax
    pop si
    pop dx
    ret
    asciitobin endp

Main:
    MOV     AX,MyData
    MOV     DS,AX                ; segment into DS
    LEA    DX>Hello1            ; address of message
    MOV     AH,09h              ; DOS service "output text string"
    INT     21h                 ; DOS service call
    mov ax, 10000
    call asciitobin

    MOV     AX,MyData
    MOV     DS,AX                ; segment into DS
    LEA    DX>Hello2            ; address of message
    MOV     AH,09h              ; DOS service "output text string"
    INT     21h                 ; DOS service call
    mov ax, 20000
    call asciitobin

    mov ah, 4ch
    mov al, 00h
    int 21h

MyCode ENDS                    ; end of code segment

;***** Fine modulo
    END     Main

```

All'interno dell'area CODE SEGMENT individuiamo 2 sottoprogrammi o *funzioni*:

- STAMPA PROC (procedura di stampa)
- ASCIITOBIN PROC

Esiste anche un'altra specie di sottoprogramma costituito dal frammento di codice adibito al MAIN. E' importante notare che anche nel linguaggio C il *main* non è altro un sottoprogramma (o funzione) all'interno del quale si raccolgono le istruzioni per il corretto funzionamento del programma. In C, inoltre, il *main* non può mai mancare risultando quindi una funzione necessaria al funzionamento del programma.

Analizziamo ora in dettaglio il codice:

END MAIN → costituisce una direttiva al compilatore per indicare ad esso l'etichetta alla quale far arrestare il programma.

STAMPA PROC

```
;***** routine che stampa 1 carattere ASCII in dl
  stampa proc
    push ax
    push dx
    mov ah, 2
    int 21h
    pop dx
    pop ax
    ret
  stampa endp
```

MOV AH, 2 → comando che indica la stampa del registro DL (= ovvero la *parte bassa* del registro DX).

INT 21H → chiamata alle librerie del sistema operativo, che a loro volta possono essere intese come sottoprogrammi o funzioni del sistema operativo. I parametri che vengono passati a questa libreria di sistema sono quelli manipolati in precedenza alla sua chiamata, ovvero DL e AH, dall'istruzione subito precedente la chiamata stessa.

E' necessario fare attenzione alle chiamate ricorsive che non sempre sono possibili, poiché si correrebbe il rischio di andare a sovrascrivere l'area dati già utilizzata.

STACK: le istruzioni Push e Pop

All'interno dello *stack*, che non è altro che una struttura dati in forma di pila gestita da una politica LIFO (Last In First Out, ovvero il primo dato ad entrare nella pila corrisponde all'ultimo ad uscirne), le informazioni sono inserite ed estratte tramite due specifiche istruzioni assembler, che sono:

PUSH REGISTRO	Copio il valore contenuto nel registro specificato sullo stack e successivamente decremento il valore dello STACK POINTER (SP) di un unità.
POP REGISTRO	Nel registro specificato viene copiato il valore che si trova in cima allo stack, poi incremento lo SP di una unità.

Lo stack è una struttura dati per l'immagazzinamento temporaneo di informazioni e prende posto all'interno della RAM, dove sono memorizzate anche tutte le altre istruzioni del programma assembler da eseguire.

La semantica di tali istruzioni è:

PUSH → $SP = SP - 2$

Copio il registro dove punta SP

POP → Copio nel registro il dato puntato da SP

$SP = SP + 2$

E' importante ricordare che per ogni operazione PUSH è necessaria una conseguente operazione POP in modo tale da riportare man mano lo stack alle condizioni originarie. Per esempio:

```
PUSH AX
PUSH DX
...
POP DX
POP AX
```

In questo modo prima impilo nello stack i dati che mi interessano e successivamente al loro utilizzo li estraggo nell'ordine inverso in cui li ho inseriti nella pila, in modo tale da lasciare lo stack nella situazione iniziale in cui lo ho trovato prima di utilizzarlo. Se non utilizzassi POP in conseguenza al PUSH non ripristinerei mai l'ordine dei dati.

Facciamo ora un esempio di procedura (*proc. Chiamata*) chiamata da un'altra procedura (*proc. Chiamante*):

<i>Procedura chiamante</i>	<i>Procedura chiamata</i>
...	f proc
1. mov ax,7	5. push bp
2. push ax	6. mov bp,sp
3. mov ax,k	7. mov ax,[bp+4]
push ax	...
4. call f	8. mov bx,[bp+6]
12. add sp,4	...
...	9. pop bp
	10. ret
	11. f endp

Analisi del codice

Innanzitutto:

- Enumero le istruzioni mappandole. La numerazione va inserita tra le istruzioni e non a corrispondenza delle stesse.
- Suppongo di avere uno stack che posso riempire dal basso verso l'alto.

Analizzo poi passaggio per passaggio il mio codice:

1. Lo *stack pointer* contiene l'indirizzo dell'ultima cella dello stack puntata. Ignoro che cosa abbia fatto il programma fino a questo punto, e non ho interesse a saperlo.
2. Nel registro AX ora è registrata la costante 7.
3. Decremento di 2 (una unità) lo stack pointer e registro il valore 7 all'interno dello stack.
4. In AX ora è memorizzato il valore K, che non è specificato (ma non ci interessa). Mi preparo a chiamare la funzione f con l'istruzione CALL F.

CALL PROCEDURA → l'istruzione è ovviamente scritta ad un dato indirizzo. L'istruzione subito successiva alla CALL si trova all'indirizzo seguente (o comunque ad un indirizzo differente). Quando viene invocata la CALL con una PUSH inserisco nello stack l'indirizzo dell'istruzione successiva alla chiamata a funzione, in modo tale da tornare al programma chiamante una volta terminata la chiamata a funzione con una JMP (jump) che mi consente di "riprendere" il programma chiamante da dove l'ho lasciato.

5. Incremento lo SP di 2 e inserisco nello stack l'indirizzo all'istruzione successiva alla chiamata a funzione. Allora sposto BP nello spazio lasciato dal decremento di SP di 2.
6. Ora BP ed SP hanno il medesimo valore.
7. [BP+4] punta alla locazione dello stack corrispondente a SP+4.
8. [BP+6] punta alla locazione dello stack corrispondente a SP+6.
9. Invoco una POP sul dato BP estraendolo così da in cima allo stack. Ora il dato BP possiede il valore che aveva prima della chiamata a funzione.

10. Con RET vado ad estrarre l'indirizzo che avevo precedentemente salvato nello stack relativo all'istruzione seguente alla chiamata a funzione. In questa maniera torno immediatamente alla funzione chiamante.
11. Termino la procedura e torno al programma chiamante.
12. In questo contesto, con ADD faccio sostanzialmente quello che avrei potuto fare con due POP consecutivi: in questa maniera non mi preoccupo di lasciare "pulito" lo stack ma solamente di portare lo SP alla posizione di partenza.

E' importante sottolineare che:

- Il linguaggio C passa i parametri per valore. La funzione chiamata non può alterare il valore dell'indirizzo fornito dal chiamante.
- All'interno della funzione chiamata potrebbe esistere una ricorsione allorché la funzione prenda a richiamare se stessa. In questa maniera nello stack vanno a crearsi dei *RECORD DI ATTIVAZIONE* che, come scatole chiuse, ricordano i dati di ogni chiamata ricorsiva.

Sintassi dei linguaggi di programmazione

Ogni linguaggio di programmazione possiede dei **NOMI** (es: *int, float, ecc...*) e **PAROLE CHIAVE** (es: *main, typedef, ecc...*). Si tratta di oggetti predefiniti all'interno di ogni linguaggio, ovvero decisi a priori ed utilizzabili così come sono dati all'utente. E' vero però che si è soliti creare *nomi* personali per nominare a piacimento gli oggetti già predefiniti all'interno del linguaggio, come per esempio nominare una struttura appena creata per riferirsi a lei con il suo nome in modo da distinguerla da tutte le altre strutture ad essa simili.

Guardiamo ora l'esempio seguente:

```
int succ(i){
    return(i+1)%dim;
}
```

succ, i, dim sono **nomi** definiti dall'utente. Infatti *succ* è il nome arbitrario della procedura di tipo intero, *i* è il nome arbitrario della variabile utilizzata e *dim* è il nome arbitrario di ciò che la funzione vuole restituire. Tutto il resto è già conosciuto a priori dal linguaggio, ed è il linguaggio stesso che mette a disposizione dell'utente la sua sintassi predefinita, come ad esempio {}, (), ; , + , % , *int, return*. E' importante ricordare che i nomi degli oggetti sono costituiti da *stringhe di caratteri*.

Vengono solitamente definiti dall'utente:

- Le variabili
- I parametri formali
- Le procedure (funzioni, sottoprogrammi)
- I tipi definiti dall'utente
- Le etichette (che tuttavia in C non vengono mai utilizzate)
- Le costanti
- I moduli (che in C non esistono, ma esistono in altri linguaggi)
- Le eccezioni (che in C non esistono, ma esistono in altri linguaggi)

Invece sono definiti dal manuale del linguaggio:

- I nomi e le caratteristiche primitive (per esempio, INT)
- Le operazioni primitive
- Le costanti del linguaggio
- Le costanti lessicali

E' importante ricordare che ad ogni nome corrisponde o un valore allocato in una specifica locazione di memoria oppure una vera e propria locazione di memoria.

Nel linguaggio C il *tipo* è determinante solo al momento della scrittura/compilazione.

Ambienti

Si indicano con **SCOPE RULES** le norme che regolano le associazioni di nomi e celle di memoria e le associazioni tra celle di memoria ed il loro contenuto. Nel caso specifico dell'associazione tra nome e cella di memoria, tale legame (o *binding*) è definito **AMBIENTE**.

Nei linguaggi a blocchi le regole di scope (*scope rules*) sono definite dal testo effettivo del programma, ovvero da come viene scritto tipograficamente il codice, e non cambiano a fronte di successive esecuzioni del programma compilato, quindi non dipendono assolutamente da ogni particolare attivazione. *Queste regole sono quindi note al tempo di compilazione*. In caso contrario si avrebbero cambiamenti solo in fase di esecuzione, quindi *in fase di run*, e da un run all'altro le regole suddette potrebbero variare.

Sono dette **UNITA' SINTATTICHE**:

- Le funzioni
- I blocchi

Nel caso del linguaggio C, all'interno sia dei moduli che delle funzioni, posso definire sempre dei binding locali. Lo stesso posso fare con i blocchi, ma dopo essere usciti dal blocco torno alle condizioni di partenza tralasciando quello che è stato manipolato al suo interno.

E' detto **AMBIENTE PREDEFINITO** quell'insieme di associazioni (binding) di nomi e locazioni di memoria definite dal manuale del linguaggio.

Supponiamo:

```
int a;

void P1();

void main(){
  ...
}
```

In questo caso, *int a* è visibile a tutto il programma salvo mascheramenti (ad opera di altre variabili dello stesso nome inserite nel codice). *int a* costituisce quindi l'**AMBIENTE GLOBALE**.

E' detto **AMBIENTE LOCALE** l'insieme di associazioni (binding) tra nomi e locazioni di memoria che si vengono a creare all'attivazione di un sottoprogramma o funzione. All'interno dell'*ambiente locale* troviamo l'attivazione di parametri formali e variabili locali che nascono e muoiono solamente all'interno del sottoprogramma. In altri linguaggi a blocchi è consentita la creazione di *sottoprogrammi locali*, quindi sono permessi annidamenti di funzioni. In questo caso una procedura ne può richiamare un'altra al suo interno. In C però questo non è possibile.

Guardiamo il seguente esempio in linguaggio DELPHI-PASCAL:

```
PROGRAM M (OUTPUT)
VAR A,B,C:REAL;
PROCEDURE SUB1 (A:REAL)
  VAR D:REAL;
PROCEDURE SUB2 (C:REAL)
  VAR D:REAL;
  begin
    C:=C+B;
  end
begin
  SUB2(B);
end
begin
  SUB1(A)
end
```

Nella procedura SUB2:

L'**ambiente locale** è costituito

- Dal parametro formale C
- Dalla variabile locale D

La variabile B si riferisce alla variabile globale omonima.

La variabile globale C è mascherata dalla variabile locale omonima.

La variabile D è mascherata, tanto che se non lo fosse apparterebbe all'ambiente locale di SUB2.

L'**ambiente globale** è costituito da OUTPUT. Nel linguaggio C *l'ambiente globale coincide con l'ambiente non-locale*.

Modelli di valutazione dei parametri

Guardiamo ora il seguente esempio:

```
int a(int par);

main(){
  int j;

  j=2;
  b=a(j);
}
```

int par è un parametro formale noto alla compilazione (quindi alla stesura del codice). Nell'istruzione *b=a(j)*, *j* diventa parametro attuale. Dopotutto conosco già il tipo di *j* (*int*) e lo controllo solo durante la compilazione.

MODELLO NORMALE. La valutazione dei parametri non viene eseguita dal chiamante, così i parametri rimangono non valutati. Vengono passati solo i parametri che servono al chiamante e solamente quelli vengono valutati all'occorrenza. Vengono conservati e perpetrati tutti i binding presenti nell'ambiente precedente alla procedura chiamante, ed assieme a loro sono passati anche i *thunk* (ovvero segmenti di codice che vengono pubblicati solo se serve).

Nell'esempio che segue possiamo vedere come sia possibile evitare il collasso della procedura chiamando solamente il parametro che le serve, ovvero il parametro *a*, senza dover per forza valutare anche tutti gli altri (parametro *b*) che, se valutati assieme, potrebbero bloccare l'esecuzione del sottoprogramma:

```
float try(float a, float b);

void main(){
  int x;
  float c;

  x=00;
  c=try(x,1/x);
}

float try(float a, float b){
  if(a==00)
    return(10);
  else
    return(b);
}
```

Avviene un **passaggio di parametro per nome** solo all'interno del *modello normale*. Tale procedura consente di passare il nome del parametro assieme a tutti i suoi binding e la sua valutazione è eseguita solamente da chi la ha bisogno di operarla.

MODELLO APPLICATIVO. Con questo metodo sostituisco ogni parametro dopo averlo valutato, così che i parametri siano tutti ugualmente noti.

Con questo modello si hanno **passaggi di parametri**:

- **per valore** – il sottoprogramma chiamato riceve il valore del parametro e non l'indirizzo di memoria al quale si trova. Per riferire allora tale valore lo associa ad una variabile locale. L'unica cosa che si può fare in questo caso è passare un qualsiasi oggetto che abbia un valore (per questo motivo non può prelevare indirizzi). E' l'unico passaggio di parametri consentito nel linguaggio C.
- **per indirizzo** – non è il valore ad essere passato ma la vera variabile sulla quale è registrato il valore. In questa maniera si crea un sinonimo (*alias*) del nome del parametro chiamante. Quando passo una variabile per indirizzo non posso chiamare, per esempio, una *costante* dal momento che il compilatore ignora quale cella di memoria associare all'*alias*.
- **per valore risultato** – è un misto dei due tipi di passaggi precedenti. Il codice della funzione chiamata richiede un passaggio dei parametri per valore, mentre l'istruzione *return* richiede un passaggio dei parametri per indirizzo.

Preprocessore

La fase del PREPROCESSORE costituisce la fase antistante quella della compilazione. Nella fase inerente al preprocessore vengono prese in considerazione le istruzioni precedute da # (cancellotto) all'interno del codice.

```
#include <nomefile.h>
```

Con questa istruzione includo nel codice un *modulo* (ovvero un file differente da quello su cui sto lavorando ma che può contenere una parte del mio programma) oppure una *libreria*. Se scrivo il nome del file con la sua relativa estensione all'interno di <...> includo un file che non si trova all'interno della stessa directory dove si trova il codice sul quale sto lavorando. Automaticamente il file incluso viene cercato dove è indicato dalla *variabile d'ambiente* definita dalla stessa istruzione *include*, la quale viene settata in ambiente Microsoft con *set include=C:/comp/c/inlcude*. Se invece il file da includere è indicato tra "..." allora si trova nella stessa cartella del codice da compilare.

```
#define NOME TESTO-DA-SOSTITUIRE
#undefine NOME
```

Con la prima istruzione indico una sostituzione da effettuare tutte le volte che nel codice trovo scritto NOME. La sostituzione avverrà quindi con TESTO-DA-SOSTITUIRE. Per invalidare questa sostituzione prendo in considerazione la seconda istruzione. Entrambe queste istruzioni possono essere collocate ovunque nel codice sia necessario. E' doveroso sottolineare che l'utilizzo di queste istruzioni agevola la lettura del codice, che deve essere sempre il più intuitiva possibile. Ora vedremo altre istruzioni che possono perorare tale causa con l'utilizzo di quello che in linguaggio C viene definito col nome di *macro*.

```
#define add(a,b) ((a)+(b))
```

L'istruzione dell'esempio indica che, ogni qual volta nel codice si trovi la dicitura *add(a,b)*, è necessario operare una somma tra i due valori. Questa macro costituisce uno dei primi modi in cui costruire delle funzioni semplici. Nel linguaggio C, infatti, le prime procedure erano costruite in questa maniera. E' necessario fare attenzione a dichiarare procedure simili, perché se si scrivesse *#define add(a,b) a+b* e si incontrasse nel codice un'istruzione simile a *z=3/add(a,b)* il compilatore potrebbe capire *z=3/a+b* e potrebbe parafrasare l'istruzione ignorando le priorità delle operazioni.

```
boolean and(boolean a,boolean b){
```

```

return(a&&b)
}

#define and(a,b) ((a)&&(b))

```

Le due diciture di cui sopra sono pressoché equivalenti. Tuttavia, *macro* e *funzione* sono due cose differenti. Le istruzioni suggerite all'interno di una procedura vengono eseguite in run time (quindi in tempo di esecuzione del programma) mentre quelle descritte all'interno di una macro agiscono dal codice sul programma a tempo di compilazione. L'operazione booleana AND, inoltre, costituisce uno **SHORT CIRCUIT**, ovvero costituisce un sistema che quando viene utilizzato controlla il primo valore che gli viene passato (nel nostro caso *a*) e valuta se questo è *true*, allora, solo in questo caso, prende in considerazione anche il secondo valore. Lo *short circuit* non “funziona” se uno dei suoi parametri non è confermato.

In sostanza, macro e funzione portano allo stesso risultato ma con due dinamiche differenti. Utilizzare le macro non è sbagliato, ma bisogna farne un utilizzo oculato, evitando quindi ovvietà che possono essere evitate (ESEMPIO. #define UNO 1)

Una **ASSERT** è una macro standard che appartiene ad una specifica libreria ma non è propria del linguaggio di programmazione. Per poter utilizzare una *assert* (o *asserzione*) devo prima includere la sua libreria all'interno del codice (#include <assert.h>). In seguito a ciò in qualsiasi parte del codice posso inserire un'istruzione del tipo `assert(condizione)` per verificare la condizione espressa tra le parentesi (...). Il controllo che viene effettuato dall'assert verifica se l'indirizzo del parametro segnalato sia diverso da NULL. Se la condizione espressa è verificata il programma è autorizzato a proseguire, se invece non lo è viene stampato a schermo tramite uno standard error un messaggio di errore del tipo `assert failed in file nomefile.c line numero-linea`. Un'assert può tornare utile all'interno di programmi molto lunghi e complicati dove è necessario e vitale controllare la veridicità di certe condizioni; in questo modo ovunque esista una condizione non verificata controllata da una asserzione è possibile ricondursi con facilità a dove si trova all'interno del codice. Come per le macro, è importante non abusare delle assert nei propri programmi.

Funzioni con numero di parametri variabile

Come è possibile creare funzioni che, come la `printf`, possano ricevere un numero sempre variabile di parametri? Se analizziamo la funzione `printf` presente dalla `stdio.h`, ci accorgiamo che la sua dichiarazione è costituita generalmente di due parti e, all'interno della prima, è possibile enumerare la quantità di parametri che essa si appresta a manipolare. In generale, il linguaggio C fa largo uso di macro per definire le procedure a numero variabile di parametri.

Guardiamo l'esempio seguente:

```

#include <stdio.h>
#include <stdarg.h> //libreria che include le macro che ci servono

void chiamata(int n, ...);

void main(){

    chiamata(1,5);      //chiamata alla funzione "chiamata"
    chiamata(2,5,7);
    chiamata(3,5,7,9);

}

void chiamata(int n, ...){
    int i, i2;
    va_list puntatore;

    va_start(puntatore,n);
    for(i=1;i<=n;i++){
        i2=va_arg(puntatore,int);
        printf("PARAMETRO N %d-%d\n", i, i2);
    }
}

```

```

}
va_end(puntatore);
}

```

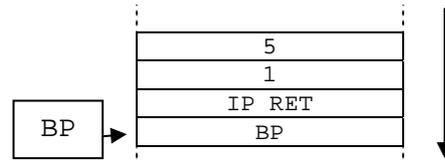
Prendiamo in esempio la chiamata a funzione `chiamata(1,5)`:

E' noto che il linguaggio C passa i parametri da destra a sinistra, allora la creazione di una funzione a numero variabile di parametri è possibile. I linguaggi DELPHI e PASCAL passano i parametri da sinistra a destra e non è possibile mediante il loro utilizzo creare funzioni a numero variabile di parametri. Detto ciò, a livello di codice assembler succede:

```

MOV  AX,5
PUSH AX
MOV  CX,1
PUSH CX
CALL
ADD  SP,4

```



In questo modo anche se non so quanti parametri ho inserito nello stack, mi è dato modo di leggerne la quantità. (?)

Nel caso:

```

int *f(int i);
void useless();

void main(){
  int tre=3;
  int *ris;

  ris=f(tre);
  printf("%d\n",*ris);
  useless();
  printf("%d\n",*ris);
}

int *f(int i){
  int locale;
  locale=1;
  return &locale;
}

void useless(){
  int a, b, c;
  a=1;
  b=2;
  c=3;
}

```

`int locale` è un binding di ambiente locale inerente alla funzione, che quindi nasce e muore all'interno del suo ambiente (ovvero, all'interno della funzione che lo utilizza). Nello stack di riferimento si allocano quindi tutte le variabili di cui fa uso la funzione (quindi tutte quelle che sono invocate nel segmento di codice tra le parentesi graffe della funzione). Alla fine della procedura, però, lo stack viene riportato alla condizione iniziale ma non viene cancellato, viene spostato solo lo stack pointer. In questa maniera, l'istruzione `return &locale` ritorna un indirizzo che non contiene più il dato che si sta cercando.

Analisi di un programma: lo stile di scrittura

Per esercizio, analizziamo un programma che, data una *sequenza* di oggetti, la rovesci. Non a caso parliamo di una generica sequenza e non di una specifica successione di numeri o caratteri, poiché è importante in questo esempio generalizzare ed astrarre il più possibile. Infatti è nostro compito generare un algoritmo e quindi un codice ad esso relativo che funzioni correttamente (per lo meno in teoria) in maniera completamente indifferente al tipo di architettura che poi lo dovrà eseguire. Inoltre, è proprio il concetto di sequenza quello che noi ora vogliamo esaminare.

Per fare un esempio di sequenza, immaginiamo una qualsiasi sequenza di caratteri. La posso strutturare come:

- un array
- una lista
- un file

Esistono quindi molti modi differenti di realizzare una sequenza di caratteri.

Prendiamo ora in considerazione alcune funzioni di base che ci serviranno poi per costruire il nostro algoritmo di rovesciamento della sequenza. Le procedure che elencheremo sotto sono procedure che osservano il loro scopo astraendo dall'architettura dell'hardware sul quale si appoggiano, e che quindi sono valide sempre e si specializzano solo al loro interno. Il programma che verrà quindi scritto è sempre valido in ogni circostanza, perché a cambiare saranno poi il codice e non il comportamento di queste specifiche *funzioni-mattone*.

`FineSeq(Seq S)` → Determina l'ultimo elemento della sequenza S.

`PutElem(Seq S, Elem E)` → Inserisce un elemento E nella sequenza S.

`GetElem(Seq S)` → (?)

`Next(Seq S)` → sposta lo *scansore* (una sorta di segnalibro di posizione) sul prossimo elemento della sequenza S. Lo scansore mi indica quindi l'oggetto corrente della sequenza.

`InitSeq(Seq S, char modo)` → crea una nuova sequenza S in modalità R/W.

E' importante ricordare che l'implementazione delle funzioni sopra elencate dipende in modo vincolante dal modello che si sceglie per realizzare la sequenza in oggetto (c'è quindi differenza tra una sequenza sotto forma di array, di lista o di file).

I codici della funzione *main* e della procedura *reverse* sono riportati di seguito.

```
void main(){
  InitSeq(inseq, 'R');
  InitSeq(outseq, 'W');
  reverse(inseq, outseq);
}

void reverse(seqtype inseq, seqtype outseq){
  elem e;
  if(!FineSeq(inseq)){
    e=GetElem(inseq);
    Next(inseq);
    reverse(inseq, outseq);
    PutElem(outseq, e);
    Next(outseq);
  }
}
```

La sequenza `inseq` dev'essere piena ed avere lo scansore puntato al suo primo elemento. Parimenti, la sequenza `outseq` deve essere vuota ed avere lo scansore che punta dove dovrebbe trovarsi il suo primo elemento. Nella prima parte della procedura l'algoritmo manipola `inseq` e poi viene chiamata ricorsivamente la procedura `reverse` finché è verificata la condizione dell'`if`. Nella seconda parte della funzione viene organizzata la sequenza `outseq` "generata" dalla ricorsione precedente. E' importante notare che, poiché sia `inseq` che `outseq` vengono manipolate e modificate, sarebbe ideale passare entrambe alla funzione per indirizzo. Regola generale vuole *che tutto ciò che viene modificato da una procedura venga passato ad essa per indirizzo*.

Dando un'occhiata al resto del codice, è necessario sottolineare che la procedura `reverse` ed il `main` stesso sono clienti concettuali di "cose" che poi vengono definiti nello specifico nelle singole funzioni-

matteone. Nel `main` vale il discorso introdotto poco prima secondo il quale le sequenze andrebbero passate alle funzioni per indirizzo. I caratteri `W` e `R` costituiscono *flags* per i permessi di scrittura e lettura.

In sintesi, quando creo una sequenza utilizzo:

- una sequenza di oggetti vera e propria
- uno scansore che mi indica l'oggetto utilizzato della sequenza
- un flag `R/W`

Quindi quando creo una sequenza posso organizzare questi tre aspetti in una *struttura* che li raggruppi assieme per definire in maniera univoca la sequenza che voglio utilizzare.

```
struct sequenza {
    char sequenza[];
    int scansore;
    char flag;
} seq;

void InitSeq (seqtype S, char modo){
    if(modo=='R'){
        S->flag='R';
        S->scansore=0;
        S->sequenza[0]='a';
        S->sequenza[1]='b';
        S->sequenza[2]='c';
        S->sequenza[3]='d';
    }
    else
        S->flag='W';
        S->scansore=0;
        S->sequenza[0]='a';
        S->sequenza[1]='b';
        S->sequenza[2]='c';
        S->sequenza[3]='d';
    }
}
```

Sopra è descritta una ipotetica *struttura* sequenza dove la sequenza è definita come un array di caratteri. La funzione `InitSeq` allora inizializza una nuova sequenza utilizzando la struttura definita sopra.

Definizione di funzioni

Il linguaggio C definisce le sue procedure con:

- un nome
- alcuni parametri e tipi
- un tipo ritornato

Tutto ciò è detto **firma** della funzione (in inglese, **signature**).

All'interno della definizione di una procedura sono ammessi tipi base e tipi definiti dall'utente tramite precedenti `#typedef`:

`#typedef void (*TipoN)(int)` → allora poi con `TipoN`, definito a piacere dall'utente, posso descrivere una variabile `x` come `TipoN x`, ed `x` sarà sempre e comunque di tipo `int`.

Posso definire anche **puntatori a funzioni** (ricordiamo che un *puntatore* è una cella di memoria che contiene un indirizzo). Nel nostro caso un puntatore a funzione potrebbe essere una cella di memoria che contiene l'indirizzo della prima istruzione di una funzione.

```
void f1(int a){
    printf("Sono f1 chiamata con %d\n",a);
}

x=&f1;
(*x)(5);

// oppure:
```

```
x=f1;  
(x)(5)
```

Data la funzione f_1 che ha come suo unico scopo quello di stampare a video il messaggio del suo utilizzo attivato dal parametro a , con le due seguenti diciture (completamente equivalenti) posso richiamare la funzione f_1 e utilizzarla con il valore specificato tra (...).

A basso livello, il **puntatore a funzione** si riferisce ad un indirizzo specificato nel *code segment*, contrariamente al *puntatore ad una variabile* che fa riferimento ad un indirizzo reperibile solo nel *data segment*. Nella realizzazione di un linguaggio ad oggetti è fondamentale, non ch  estremamente utile, l'utilizzo dei puntatori a funzione.

Il linguaggio C ammette puntatori a funzioni passati anche come parametri, ma altri linguaggio come il DELPHI o il PASCAL no (per esempio, loro ammettono l'esistenza di 4 ambienti differenti e consentono la programmazione a blocchi che invece   impossibile nei soli due ambienti semplificati del linguaggio C). Bisogna comunque fare attenzione ai contrasti ed ai fraintendimenti che possono venire a crearsi all'interno delle procedure alle quali   stato passato come parametro un puntatore a funzione.

Record di attivazione

Guardiamo il seguente frammento di codice in linguaggio PASCAL:

```
function FN(x:real; y:real):real;  
const max=20;  
var M:array[1..max]of real;  
var N:integer;  
begin  
  // segnalibro: Istante 1  
  N:=2;  
  x:=2*x+M[5];  
  FN=x;  
end;  
  
// segue il comando che chiama la funzione  
  
R:=FN(A,B);
```

Generalmente, sono dette **funzioni** quelle procedure che tornano sempre un risultato di qualsiasi genere, mentre sono dette semplicemente **procedure** quei segmenti di codice che, come le *void* nel C, non ritornano nulla. Il termine *funzione*   stato introdotto per assimilare le procedure del linguaggio atte a manipolare dei dati alle vere e proprie funzioni matematiche.

Il linguaggio PASCAL, contrariamente al C, consente di inizializzare array che incominciano e terminano nelle locazioni definite dall'utente. Come se pu  vedere dal codice sopra, l'array M incomincia alla "celletta" 4 e termina alla 20. In C questa definizione non sarebbe stata possibile.

All'istante 1 (segnalato appositamente nel codice) il compilatore ha gi  compilato il codice ed ottenuto quindi un file seguibile. Abbiamo che:

1.   stata riservata una locazione in memoria rispettivamente per i **parametri** specificati x e y ;
2.   stato creato in memoria lo spazio necessario per il **valore di ritorno** della funzione FN che   tra l'altro di tipo *real*. Tutto ci    sottolineato dall'espressione $FN=x$;
3.   stato riservato dello spazio in memoria per le **variabili locali** (20 *real*, 1 intero);
4.   stato creato lo spazio dove riporre le **costanti** (max , 2, 5, 1);
5. in memoria   stato trascritto il **codice assembler** eseguibile del programma generato dalla sua precedente compilazione.

Gli spazi in memoria specificati ai punti 4. e 5. sono locazioni di memoria di dimensioni note, ovvero note a tempo di compilazione, e sempre a tempo di compilazione   reso noto ancora il loro contenuto che anche in seguito risulta essere immutabile. Ci  che   stato descritto viene collocato nel **segmento del codice** (che, si badi bene,   totalmente differente dal *code segment* di basso livello). Per quanto riguarda i punti 1., 2. e 3., si parla sempre di aree di memorie rese note a tempo di compilazione ma che contengono

altresì dati che possono subire modifiche specialmente in tempo di esecuzione. L'area di memoria che nel suo complesso raccoglie tutti e cinque i tipi di locazioni di memoria è chiamata **RECORD di ATTIVAZIONE**. A basso livello, il codice assembler ad esso relativo si trova nel *data segment*. Guardiamo di seguito la rappresentazione schematica del *record di attivazione*:

CODICE	RISULTATO FN
20	Y
2	X
5	M
1	N
	PUNTO RITORNO + ALTRO
	VARIABILI TEMP.

Record di attivazione

Ovviamente, nel record di attivazione non esistono solo gli oggetti di cui abbiamo parlato sopra ma troviamo anche, per esempio, le **variabili temporanee** che il calcolatore genera automaticamente all'interno del programma anche se non è specificatamente richiesto dal programmatore.

In ogni sistema di calcolo esistono:

- stack
- heap
- code segment
- data segment

Nello specifico, i record di attivazione si allocano sempre sullo stack. **Stack** e **heap** sono aree di memoria. Nello heap trovano posto generalmente tutte le *variabili dinamiche* allocate durante l'esecuzione di un programma. Tali variabili sono invocate da specifiche istruzioni inserite all'interno del codice come la `malloc` per il linguaggio C o la `new` per il DELPHI-PASCAL. Con l'istruzione C `free` invalido una precedente `malloc` e restituisco all'heap l'area di memoria presa precedentemente.

All'interno di un record di attivazione troviamo:

- *return adres*, ovvero l'indirizzo dell'istruzione subito conseguente alla procedura corrente la quale verrà invocata non appena terminerà la funzione in corso
- *static link*
- *dinamic link*

Static link e *dinamic link* costituiscono il PUNTO RITORNO citato nello schema visto sopra. Lo static link è un oggetto caratteristico di un linguaggio a blocchi; costituisce un puntatore al record posizionato superiormente al blocco di codice che contiene la variabile in esame (è così via procedendo di blocco in blocco). Ovviamente esistono intere catene di static link, denominate **catene statiche**. Il dinamic link costituisce un puntatore al record di attivazione della funzione chiamante dal momento che si tratta del record di attivazione al quale è necessario tornare dopo l'esecuzione del record della procedura corrente.

Prendiamo ora in esame l'annidamento proposto come esempio nel seguente codice in linguaggio PASCAL-DELPHI:

```

program main;
procedure a;
begin
  writeln('FINITO');
end;
procedure b(p:integer);
var v1:integer;
  procedure c(i:integer);
  begin
    a;
  end;
begin
  v1=p;
  c(p);

```

```

end;
begin
  b(2);
end;

```

Regole di Scope Statico

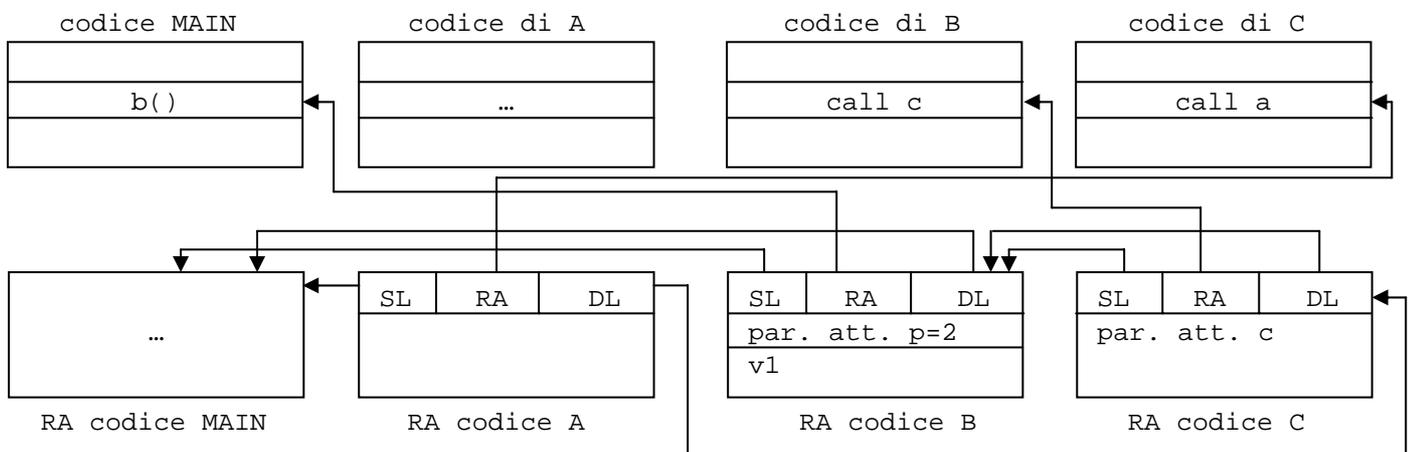
Valgono nello specifico per linguaggi come Ada, PASCAL, Java.

- (1) Le dichiarazioni delle variabili locali di un blocco definiscono l'ambiente locale all'interno del blocco stesso, ma non comprendono le dichiarazioni delle variabili all'interno del blocco.
- (2) Quando cerco un nome, se lo cerco all'interno di un blocco e lo trovo, lo trovo sicuramente nella sua definizione delle variabili globali. Se non esiste alcuna dichiarazione delle variabili globali, allora cerco il nome nel blocco subito precedente all'interno delle associazioni globali (binding globali) altresì conosciuto come l'ambiente predefinito del linguaggio. Se non esiste nemmeno lì, allora c'è un errore.
- (3) Se un blocco ha un nome, allora il nome appartiene sicuramente all'ambiente globale del blocco che lo comprende. Allora possiamo usare quel nome nell'ambiente del blocco esterno a tutti quanti.

Ora possiamo analizzare il codice riportato sopra:

- a definisce un blocco e appartiene all'ambiente globale del programma
- b appartiene all'ambiente globale del programma
- v1 è dichiarato nell'ambiente locale della procedura b, quindi appartiene all'ambiente locale di b
- c è il nome del blocco omonimo ed appartiene all'ambiente locale della funzione b
- i costituisce una dichiarazione locale della procedura c, i allora appartiene all'ambiente locale d della procedura c
- p appartiene all'ambiente locale di b

Di seguito schematizzo le strutture dei record di attivazione e dei link del codice sopra esemplificato.



RA → record di attivazione

SL → static link

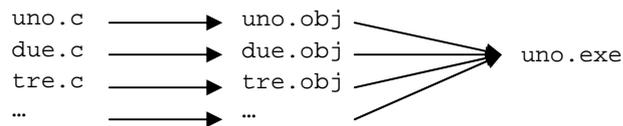
DL → dinamic link

Nel linguaggio C, se una funzione che invoca al suo interno due variabili a e b tali variabili sono allocate sullo stack.

Programmazione multi-modulo

Come già spiegato, è convenzione che da un file sorgente `.c` venga generato durante la compilazione un file oggetto `.obj` e, conseguentemente all'operazione di linking, un file eseguibile `.exe`. Durante il linking, tra l'altro, se il programma compilato consta di più file avvicinati, ovvero è composto di più **moduli** distinti, il conseguente file eseguibile conterrà tutti questi moduli presentandosi quindi come un unico file.

Facciamo un esempio:



E' importante che almeno uno dei moduli che costituiscono il programma nel complesso contenga la funzione *main*. Nel nostro esempio, probabilmente è il file `uno.c` a contenere il *main* del programma, così tutti gli altri moduli si uniscono a lui nel generare il file eseguibile che porta il suo nome.

Solitamente, durante la stesura del codice di un file sorgente, vengono definiti dei **simboli** e fin'ora non ci siamo mai curati troppo della loro visibilità. La visibilità di un simbolo indica la possibilità che quest'ultimo venga visto e usato in altre parti del codice, nel nostro caso in altri moduli, creando potenzialmente dei fraintendimenti o dei conflitti tra simboli nominalmente uguali. Ovviamente, i contrasti possono essere evitati proteggendo accuratamente il codice dei diversi moduli. Vediamo come.

CLASSE DI MEMORIA → ogni oggetto, nome o simbolo visibile nel *main* è altresì visibile a qualsiasi altro modulo venga ricollegato allo stesso *main*. In fase di link è possibile che vengano individuati altri simboli nominalmente uguali nei moduli asserviti al *main*, così viene segnalato un errore che va corretto. Per celare agli altri moduli il nome di alcuni simboli e oggetti si utilizza una particolare dicitura, della *static*. Per esempio:

```
static int pluto;
```

Allora la variabile intera di nome `pluto` è definita statica, ovvero è mascherata a tutti gli altri moduli con cui potrebbe comunicare quello dove la variabile è stata definita. Parimenti, se una data variabile fosse dichiarata in uno specifico modulo ma servisse anche ad altri moduli ad esso affini, con la dicitura *extern* all'interno del modulo "richiedente" utilizzo la stessa variabile all'interno di un diverso file sorgente:

```
extern int pippo;
```

Utilizzando la dicitura *extern* è possibile richiamare nome e tipo della variabile ma non viene allocato nessuno spazio in memoria per lei (contrariamente a quanto avviene per una semplice dichiarazione).

Riassumendo, definisco *static* le variabili che non intendo condividere tra i moduli e segno *extern* quelle che utilizzo all'interno di moduli che non sono quelli dove sono definite. Lo stesso identico discorso vale anche per le funzioni.

Quando creo un file sorgente `.c` collateralmente creo sempre un altro file `.h` avente lo stesso identico nome (ES. `uno.c` → `uno.h`). E' sempre buona abitudine includere nel file sorgente il suo omonimo file header; in questo modo non ho bisogno di dichiarare per esempio i prototipi delle funzioni prima del *main*. Inoltre è possibile lasciare gli oggetti da non esportare all'interno nel file sorgente e immettere quelli da condividere nel file header. Scrivendo:

```
#include "uno.h"
```

importo nel file sorgente tutte le dichiarazioni del file header. È importante ricordare che quando “include” una libreria, in realtà include le dichiarazioni che essa contiene di oggetti (tipo funzioni) che sono già state definite a priori. Inoltre, non si inseriscono mai istruzioni in un file header perché il file header costituisce innanzi tutto un’interfaccia per due o più moduli che dialogano assieme. Importantissimo è che ogni modulo deve includere tutti i file header di cui ha bisogno, ignorando il fatto che, magari, lo stesso file header incluso utilizzi a sua volta un file header che potrebbe servire anche al modulo che include; questo, innanzi tutto, per ragioni di chiarezza e di completezza. I file header sono principalmente elenchi di oggetti definiti `extern`. Vanno assolutamente protetti da eventuali rischi di conflitto con altri file sorgente, così ogni file `.h` incomincia e termina con le seguenti istruzioni:

```
#ifndef_uno
#define_uno
...
#endif
```

In questo modo sono sicuro di proteggermi da qualsiasi conflitto possa nascere tra gli oggetti dichiarati nel file header ed in quelli dichiarati nel file sorgente che utilizza l’header. Inoltre, il compilatore mi proteggerà anche dal pericolo delle *doppie inclusioni* (ho una doppia inclusione quando include un file header all’interno del file `.h` e poi lo richiamo ancora nel file sorgente che utilizza lo stesso file `.h`) scartando lui stesso i file header che ho ripetuto più volte o che non sono utilizzati dal mio file sorgente. Ora guardiamo il seguente esempio di costruzione di un file header:

BOOLEAN.H

```
#ifndef_boolean
#define_boolean

#define FALSE 0
#define TRUE 1
#define boolean int

#endif
```

Ora posso includere questo file header in un qualsiasi file sorgente ed utilizzare in esso il tipo `boolean` qui definito.

ELEM.H

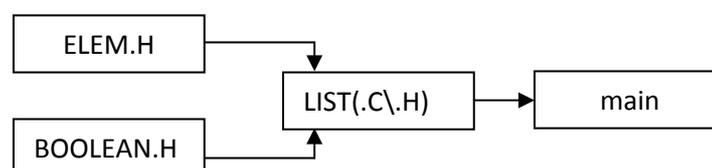
```
#ifndef_elem
#define_elem

typedef int el;

#endif
```

Nel file header `elem.h` ho definito il tipo `el` come `int`.

Ora, seguirò lo schema di un programma costituito da più moduli riportato sotto.



LIST.H

```
#ifndef_list
#define_list

#include "boolean.h"
#include "elem.h"
#include <stdlib.h>

const emptylist=NULL;
typedef struct listelem{el value, struct listelem*next};
typedef listelem*list;
extern boolean vuota(list l);
extern list cons(el e, list l);
extern el car(list l);
extern list cdr(list l);

#endif
```

LIST.C

```
#include "boolean.h"
#include "list.h"
#include "elem.h"
#include <stdlib.h>

boolean vuota (list l){
    return(l==emptylist?TRUE:FALSE);
}

list cons(el e, list l){
    list t;
    t=malloc(sizeof(listelem));
    t->value=e;
    t->next=l;
    return(t);
}

...
```

Classi di memoria

Un parametro o una qualsiasi variabile locale è definibile come **variabile automatica**. A livello assembler queste variabili sono sempre allocate sullo stack ed il suo tempo di vita è pari al tempo effettivo di attivazione della procedura entro la quale la variabile è stata generata. Di solito, una variabile automatica non si cancella alla fine dell'utilizzo della funzione che l'ha creata, ma viene invalidata, quindi resa inutilizzabile. Ricordiamo inoltre che se viene dichiarata una variabile al di fuori di una funzione (quindi se viene creata una variabile globale), questa è resa visibile a tutto il resto del file di codice a meno di particolari mascheramenti.

STATIC → Questo è un modificatore che, se scritto prima della dichiarazione di una variabile o di una funzione, rende la consecutiva variabile o funzione inaccessibili a qualsiasi altro codice ne voglia fare uso. Gli oggetti definiti static rimangono all'interno del file d'origine dove possono essere utilizzati ma non esportati. La definizione static modifica le regole di scope del codice in cui viene utilizzata. Tuttavia, se static viene utilizzato su una variabile automatica, il valore di tale variabile viene registrato nel data segment e non più sullo stack: ciò implica che la variabile viene resa disponibile e nota anche dopo l'interruzione della procedura nella quale è nata.

Se scriviamo `static int alfa` definiamo una variabile di nome `alfa` di tipo `int` per la quale viene allocato uno spazio di memoria pari al suo tipo, quindi atto a contenere un `int`.

EXTERN → modificatore che rende il simbolo che lo segue fruibile da qualsiasi file di codice. Ovviamente in questo caso posso dichiarare il simbolo ma non allocare nessuno spazio di memoria inerente.

REGISTER → è il tipico modificatore da *smanettoni* (=n.d.r.), si tratta di un consiglio dato al compilatore riguardo la variabile che segue la dichiarazione register. In questo caso la variabile che normalmente verrebbe allocata sullo stack viene allocata preferibilmente nel registro della CPU la maggior parte di tempo possibile. Se si tratta di una variabile che viene utilizzata di continuo e per lunghi periodi di tempo, è conveniente che si trovi nel registro rispetto allo stack dal momento che l'accesso al registro è molto più veloce di quello allo stack. La variabile comunque risulta allocata sia sullo stack che nel registro è importante tenere le due variabili costantemente aggiornate.

VOLATILE → è il modificatore contrario al precedente register. Volatile implica che la variabile associata può essere modificata e che quindi non vale la pena di scriverla sul registro per poi aggiornarla i continuazione con quella memorizzata sullo stack.

Oggetti software

Gli oggetti costituiscono un meccanismo per astrarre sui dati, e si tratta principalmente di

- *oggetti di libreria*
- *dati astratti*
- *tipi di dato astratto*

Si tratta di modalità di interazione tra client e server, dove il client è il programma in linguaggio C ed il server è il codice .h e .c. E' il client che vedere il server nei tre modi esplicitati sopra e spiegati di seguito.

Oggetti di libreria. Gli *oggetti di libreria* sono oggetti che presuppongono la precedente esistenza di un tipo di dato astratto che fornirà al client le funzioni primitive. Costruire una libreria significa assemblare un software che da disposizione di nuove primitive a sostegno di quelle già esistenti. La libreria non deve MAI vedere come è fatto il tipo di dato astratto che comunica.

Dati astratti. Supponiamo che il client abbia bisogno di una struttura dati specifica, tipo uno stack. Il client può guardare le funzioni primitive che gli servono per creare lo stack ma non è autorizzato a vedere come queste funzioni primitive sono implementate. Il vantaggio di questo sistema sta nella sua astrazione generale che conserva la compattezza dell'interfaccia confezionata per il client senza dare importanza a come questa è implementata a livello di codice. L'importante infatti è sapere come usare gli strumenti che sono dati e non come questi siano fatti.

Tipi di dato astratto. Un tipo di dato astratto mette a disposizione del client una dichiarazione specifica seguitante un'istruzione di typedef in modo tale che il client possa creare un numero a piacere di variabili di quello specifico tipo. Allegate al dato sono messe a disposizioni anche funzioni che lavorano su quel tipo di dato. Il cliente comunque tratta il tipo di dato astratto esattamente come tratterebbe qualsiasi altro tipo di dato primitivo. Questi tipi di dati equivalgono alle classi in Java.

E' importante ragionare sempre dal punto di vista del client.

Implementazione di uno stack come dato astratto

Di seguito forniremo un esempio concreto dell'implementazione di oggetti software.

```
#include "stackad.h"

void main(){
    int i,j;
    InitStack();
    i=2;
    Push(i);
    i=4;
    Push(4);
    j=Pop();
    j=Pop();
}
```

Le funzioni come Push, Pop e InitStack non sono funzioni primarie del linguaggio C, per questo motivo devono essere costruite e definite da qualche parte. Per questo motivo viene incluso nel programma il file stackad.h.

STACKAD.H

```
#ifndef_stackad
#define_stackad

extern void InitStack();
extern void Push(int);
extern int Pop();
extern void StackList();
```

In questo modo lavoro sull'interfaccia del client.

E' buona norma non utilizzare mai un array per creare delle strutture dati dinamiche, poiché non è dinamico ed è fortemente limitato, sarebbe meglio creare una lista. Ma nel nostro caso, a fine didattico, utilizzeremo per motivi di semplicità un array.

STACKAD.C

```
#include "stackad.h"
#define STACKSIZE 100

static int stack[STACKSIZE];
static int stackptr;

void InitStack(){
    stackptr=0;
}

void Push(int p){
    if(stackptr==STACKSIZE){
        printf("Stack overflow");
    }
    else
        stack[stackptr]=p;
    stackptr++;
}

int Pop(){
    if(stackptr==0){
        printf("Stack underflow");
    }
    else
        stackptr--;
    return(stack[stackptr]);
}

void StackList(){
    int i;
    printf("Contenuto stack\n");
    for(i=stackptr-1;i>=0;i--){
        printf("%d\n",stack[i]);
    }
}
```

Area di memoria condivisa → si tratta di una memoria appartenente al server che il client può vedere ed utilizzare.

Regole generali per i dati astratti

Sono molto importanti.

- (1) Il file .h di un tipo di dato astratto (ADT= Abstract Data Type) definisce sempre dei prototipi di funzioni tra le quali una è una funzione di inizializzazione.
- (2) Il file .h definisce talvolta alcune variabili di interfaccia.
- (3) Il file .h non contiene MAI delle istruzioni typedef.

-
- (4) Il file .c di un ADT comprende sempre una rappresentazione del dato in questione nel suo blocco più esterno ed è sempre dichiarato `STATIC`.
 - (5) Il file .c presenta gli statement di inizializzazione.
 - (6) Il file .c inizializza sempre il dato astratto.
 - (7) Il file .c riporta funzioni dichiarate `STATIC` ad uso locale.
-

xxx