INGEGNERIA DEL SOFTWARE

LINGUAGGI

Avvertenza: gli appunti si basano sul corso di Ingegneria del Software tenuto dal prof. Picco della facoltà di Ingegneria del Politecnico di Milano (che ringrazio per aver acconsentito alla pubblicazione). Essendo stati integrati con appunti presi a lezione, il suddetto docente non ha alcuna responsabilità su eventuali errori, che vi sarei grato mi segnalaste in modo da poterli correggere.

e-mail: webmaster@morpheusweb.it

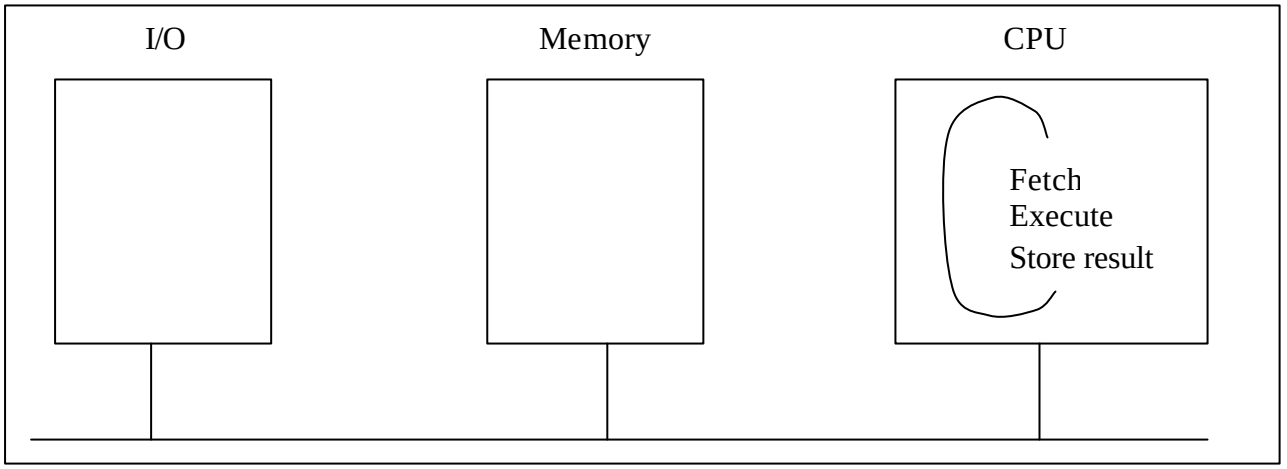
LINGUAGGI

PARADIGMI

- **PROGRAMMAZIONE PROCEDURALE**
    - I programmi sono scomposti in passi di computazione.
    - Le routines (o procedure) sono usate come unità di decomposizione.
- **PROGRAMMAZIONE FUNZIONALE**
    - Computazione dei valori tramite espressioni e funzioni che processano dati di input e forniscono output. Nella sua accezione più pura non ha il concetto di stato.
    - Le funzioni sono i blocchi principali
- **ABSTRACT DATA TYPE PROGRAMMING**
    - Tipi di dato astratto come unità di modularizzazione
- **PROGRAMMAZIONE BASATA SU MODULI**
    - Modularizzazione = gruppi di entità (variabili, procedure, tipi etc…) + export interface
      C'è un concetto di modulo che è un gruppo di entità che possono essere variabili, procedure… è un contenitore di unità elementari
- **PROGRAMMAZIONE AD OGGETTI**
    - Modularizzazione tramite la definizione di classi (simile all'ADT)
      Mi baso sul concetto di classificazione.
    - Istanze create durante l'esecuzione del programma
- **PROGRAMMAZIONE GENERICA**
    - Moduli generici che vengono instanziati
      Il concetto di modulo è variato, consentendo di lasciare alcune parti dei moduli variabili ed istanziarle poi a run-time (ad esempio in c++ posso definire uno stack a prescindere dal tipo di valori che conterrà)
- **PROGRAMMAZIONE DICHIARATIVA**
    - Problema dichiarativo, non ci sono algoritmi
      Specifico i vincoli del problema (ambiente) ed un insieme di regole.

LINGUAGGI ED ARCHITETTURA

Vediamo l'architettura di Von Neumann

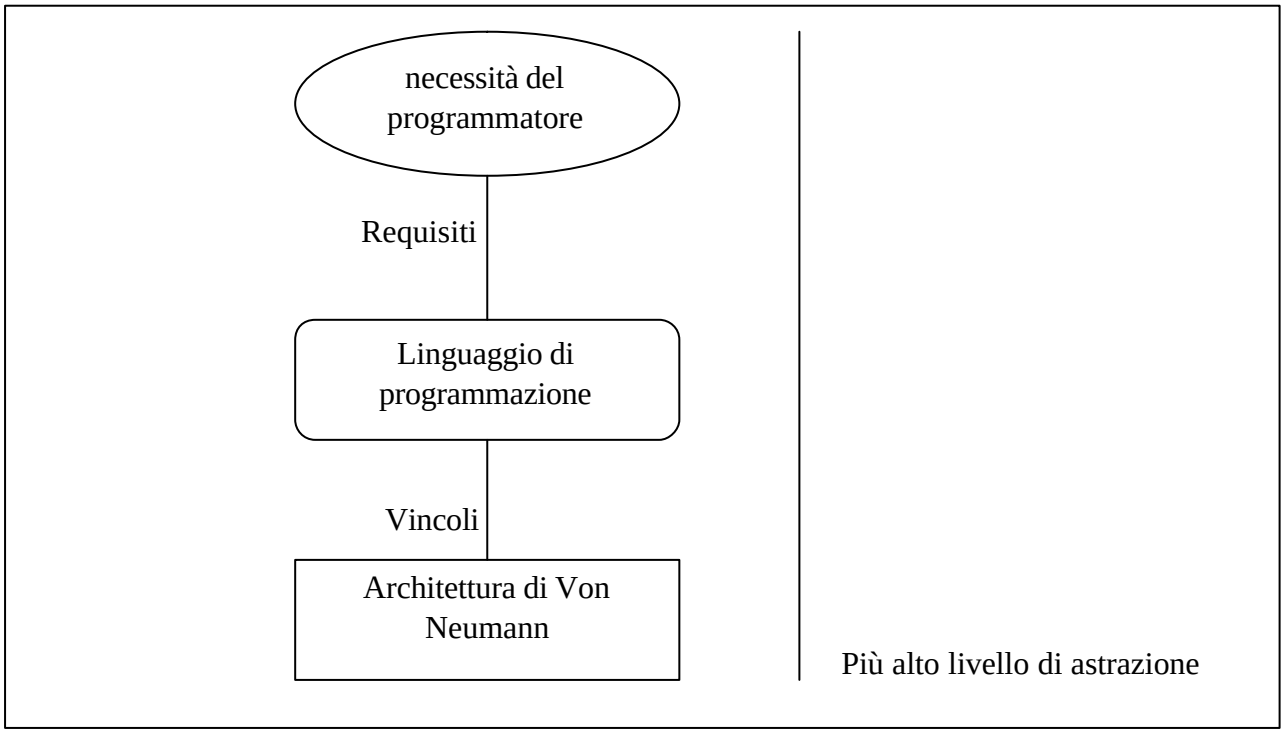


I linguaggi di Von Neumann sono chiamati anche IMPERATIVI, DICHIARATIVI.

Si basano sul concetto di ISTRUZIONE, ed hanno il concetto di STATO (derivato dall'architettura)

Nella macchina di Von Neumann la computazione avviene per passaggi da uno stato all'altro.
I linguaggi funzionali invece non seguono questo paradigma, anche se poi vengono comunque eseguiti su macchine di Von Neumann.

REQUISITI E VINCOLI DI UN LINGUAGGIO



QUALITA' DEI LINGUAGGI

Un software deve essere:

- **AFFIDABILE**

  leggibile e scrivibile

  semplice

  sicuro (ad esempio non deve essere possibile fare accesso ad aree di memoria protette)

  robusto (devo avere dei costrutti che permettono al linguaggio di prevedere possibili malfunzionamenti, come ad esempio con la gestione delle eccezioni)

- **MANUTENIBILE**

  modulare (per permettere di poter modificare solo alcune parti senza dover ridisegnare il software)

- **EFFICIENTE**

  Anche se l'efficienza della programmazione è più importante

CENNI STORICI

- PRIMI PROCESSI: PROGRAMMAZIONE SEMPLICE

  Fine anni 50 – Primi anni 60 (approccio di tipo matematico)

  FORTRAN: compilazione separata, formule
  ALGOL60: struttura a blocchi, ricorsione, strutture di dati
  COBOL: concepito per l'EDP, si concentra sulla gestione dell'I/O ed introduce il concetto di file

- PRIMI LINGUAGGI NON CONVENZIONALI

  LISP: liste, operazioni semplici
  APL: array, molte operazioni
  SNOBOL4: pone l'enfasi sull'elaborazione del testo. (stringhe, pattern-matching, backtracking)

- FINE ANNI 60

  ALGOL68: purity, ortogonalità. Si vuole ottenere la purezza del linguaggio con un insieme minimo istruzioni. L'enfasi è su come le istruzioni sono composte all'interno dei programmi. E' il primo linguaggio specificato in maniera formale.
  SIMULA76: simulazione (antesignano dei linguaggi OO)
  PASCAL: semplicità (usato per scopi didattici)
  BASIC: interattività

- ANNI 70

  Esperimenti con i paradigmi
  SMALLTALK, EIFFEL: programmazione ad oggetti
  PROLOG: logica
  MESA, MODULA2: programmazione concorrente, modularità
  EUCLID, GIPSY: sicurezza

- ANNI 80

  C++, ADA, EIFFEL: programmazione ad oggetti

- ANNI 90

  VISUAL BASIC: interfacce visuali
  FORTRAN90: parallelismo
  JAVA: network programming

SINTASSI E SEMANTICA

SINTASSI: forma del linguaggio, sequenza valide di parole, regole per combinare le parole in maniera legale

REGOLE LESSICALI: come comporre i caratteri per formare parole valide.

SEMANTICA: significato delle frasi

EBNF

Extended BNF
Abbiamo regole sintattiche e lessicali e diagrammi sintattici

REGOLE SINTATTICHE

<PROGRAMMA> ::= {<ESPRESSIONE>*}

<ESPRESSIONE> ::= <ASSEGNAZIONE> | <CONDIZIONALE> | <CICLO>

<ASSEGNAZIONE> ::= <IDENTIFICATORE> = <EXPR>

<CONDIZION.> ::= IF <EXPR> {< ESPRESS.>+} | IF <EXPR> {< ESPRESSIONE>+} ELSE <EXPR> {< ESPRESSIONE >+}

<LOOP> ::= WHILE <EXPR> {< ESPRESSIONE>+}

<EXPR> ::= <IDENTIFICATORE> | <NUMERO> | (<EXPR>) <EXPR><OPERATORE><EXPR>

REGOLE LESSICALI

<OPERATORE> ::= + | - | * | / | = | /= | < | > | <= | >=

<IDENTIFICATORE>::= <LETTERA><ID>*

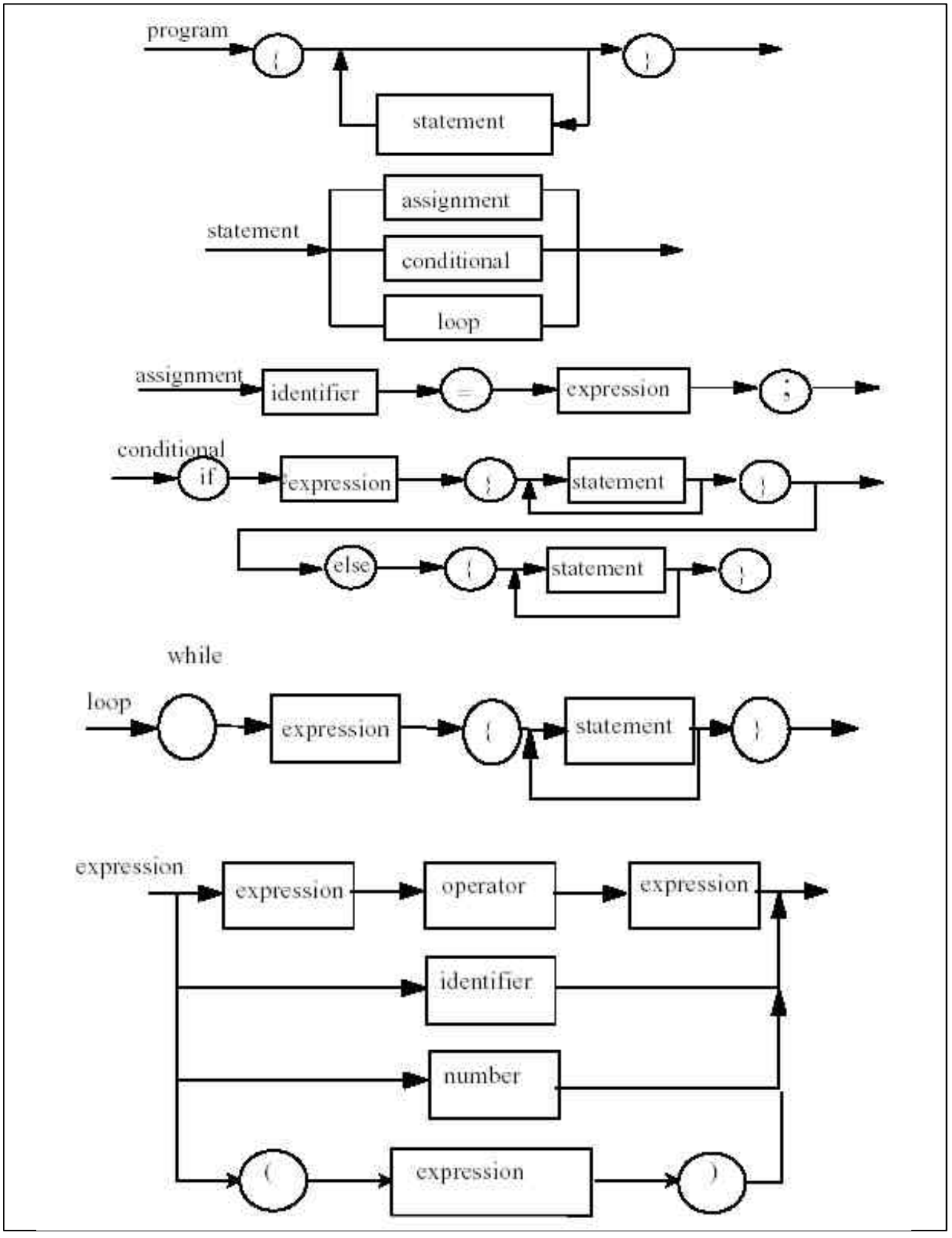
<ID> ::= <CIFRA><NUMERO>

<NUMERO> ::= <CIFRA>+

<LETTERA> ::= A | B | … | Z

<CIFRA> ::= 0 | 1 | … | 9

DIAGRAMMI SINTATTICI



C’è una perfetta equivalenza tra diagrammi sintattici e regole sintattiche.

SINTASSI ASTRATTA

Scrivo la stessa istruzione in C e PASCAL.

In C:

```
while (x!=y) {
…
};
```

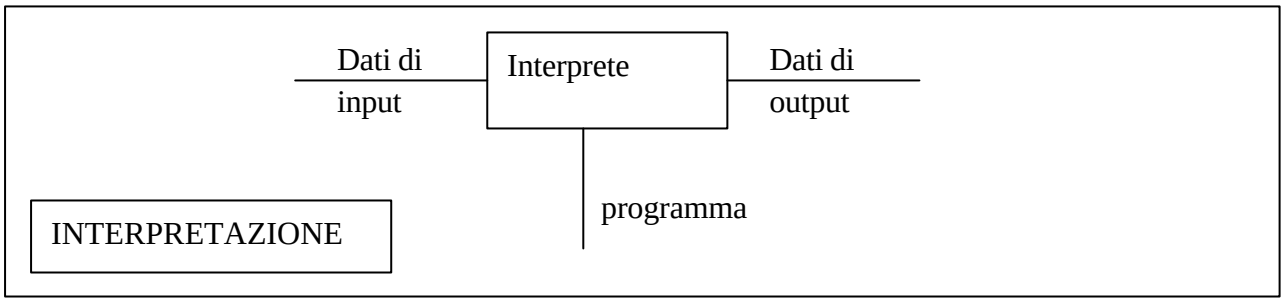
In Pascal

```
While x<>y do
begin
…
end
```

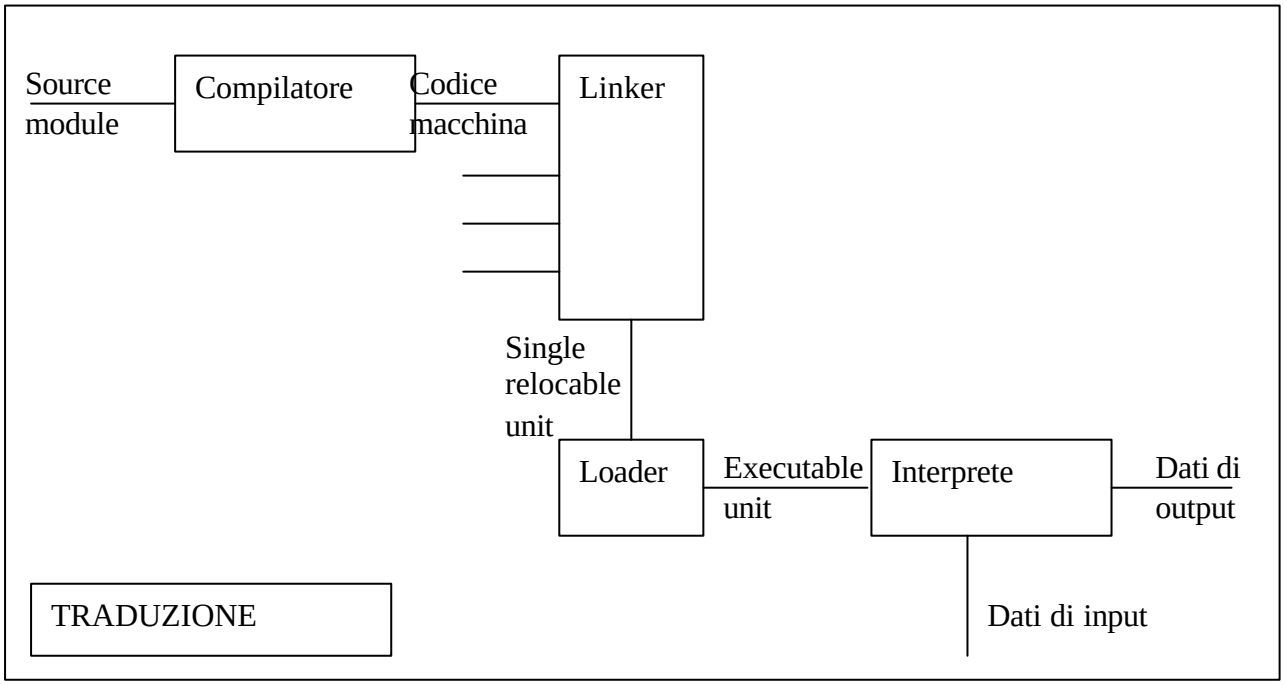
Hanno la stessa sintassi astratta ma non la stessa sintassi concreta
Differiscono nelle parole utilizzate e nel modo in cui sono combinate.

Le scelte sulle regole di un linguaggio hanno un forte impatto sulla produttività del programmatore e sulla possibilità di commettere errori.

INTERPRETAZIONE E TRADUZIONE



L'interprete prende un istruzione, la trasforma secondo le proprie regole (sintattiche e semantiche) e produce l'output.



Il compilatore effettua la traduzione e produce un primo codice macchina. In genere è rilocabile, ovvero non fa riferimento a locazioni di memoria.
C'è poi il linker che a partire dai singoli moduli , produce un unico modulo ancora rilocabile che viene prelevato dal loader il quale risolve gli indirizzi parametrici in fisici e consente il caricamento in memoria del programma.

I linguaggi interpretati sono utili in fase di sviluppo in quanto consentono di tagliare i tempi di compilazione di un programma

Con i linguaggi compilati elimino il tempo di compilazione a run-time poiché il programma eseguibile è già stato compilato. Sono quindi più veloci

Spesso per un linguaggio sono dati sia il compilatore (per generere l'eseguibile) che l'interprete (per il debugging)

Ci sono anche ibridi come ad esempio Java. Quando compiliamo generiamo un byte code che poi viene interpretato dalla Java Virtual Machine. Così facendo ho una maggiore portabilità.

BINDING

Un linguaggio di programmazione è un insieme di elementi quali:

- **ENTITA'**: variabili, routines (procedure), espressioni

- **ATTRIBUTI**:
  - variabili: nome, tipo, area di memorizzazione
  - routines: nome, tipo dei parametri, convenzioni per il passaggio dei valori

il **BINDING** descrive il legame tra entità ed attributi (**to bind = legare**)

DESCRITTORI: memorizzano le informazioni sugli attributi

I linguaggi di programmazione differiscono in:
- Numero delle entità
- Numero degli attributi da legare alle entità
- Tempo di binding (quando una certa entità viene legata ad un suo attributo)

STABILITA': un binding può essere modificato?

STATIC BINDING: non può essere modificato (altrimenti si dice dinamico)

ESEMPI

Gli interi in FORTRAN, ADA, C, C++ sono legati ad un set di valori alla definizione/implementazione ➔ *static binding*

In PASCAL all'atto della traduzione ➔ *dynamic binding*

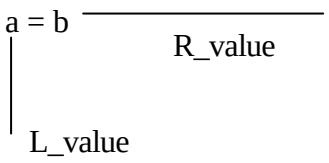
VARIABILI

Le variabili di un linguaggio convenzionale sono delle astrazioni della memoria.
Rappresentano lo stato.

Una variabile può essere rappresentata da 5 attributi:

VARIABILE = <nome, scope, tipo, L_value, R_value>

R_VALUE ED L_VALUE



R_VALUE: E' l'area di memoria in cui è memorizzata una variabile

L_VALUE: E' il valore memorizzato nella locazione di memoria

Memoria principale: celle identificate da un indirizzo (Locazione → L_value)
Espressione di assegnazione: astrazione della modifica di una cella di memoria (valore → R_value)

NOME E SCOPE

NOME: introdotto da una dichiarazione esplicita

SCOPE: Intervallo di istruzioni in cui è visibile la variabile (**scope = portata**)

Una variabile è visibile attraverso il nome nel suo ambito di visibilità.

Il binding dello scope può essere statico oppure dinamico.

> **Static scope**: Lo scope è definito dalla struttura lessicale (lo capisco quando guardo la struttura a blocchi del programma)
>
> **Dynamic scope**: Lo scope è definito in termini di esecuzione del programma. L'effetto di una dichiarazione si estende a tutte le istruzioni fino ad una nuova dichiarazione della stessa variabile.

VANTAGGI E SVANTAGGI

- Le regole di tipo dinamico sono semplici e piuttosto facili da implementare

- Ci sono svantaggi in termini di disciplina di programmazione ed efficienza dell'implementazione

- I programmi sono difficili da leggere

Esempio:

```
main {

     int x;

     {
          /*block A */
          int x;
          ...
     }
     {
          /*block B */
          int x;
          ...
     }
     {
          /*block C */
          x=5;
          ...
     }

}
```

Con lo scoping dinamico se eseguo B e poi C, in C la x che modifico è quella di B (non del main), poiché è sempre l'ultima ad essere stata acceduta)

Con lo scoping statico invece C fa riferimento alla x globale del main.

TIPI

Un tipo è dato da:
- un set di valori
- operatori per creare, accedere, modificare i valori

I tipi proteggono da operazioni che non hanno senso per un determinato tipo.

Una variabile è un'istanza di un tipo.

Un linguaggio può essere:
- Non tipizzato
- Tipizzato dinamicamente
- Tipizzato staticamente

Per altri linguaggi non c'è bisogno di specificare il tipo, ma tramite un procedimento di type inference viene dedotto automaticamente dal programma.

LINGUAGGI TIPIZZATI

- Ho dei tipi built-in (tipi incorporati)

- Dichiarazione di tipi:
  Ho la possibilità di definire i miei tipi usando i tipi built-in
  (es c++: typedef in vector [10])

    - binding tra nome del tipo ed implementazione (al momento della traduzione)

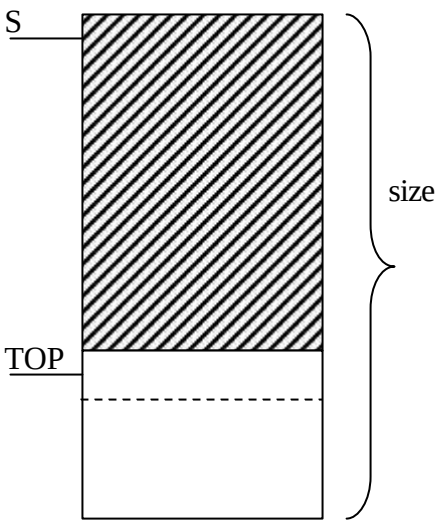
    - i nuovi tipi ereditano tutte le operazioni della struttura dati

- Tipi di dato astratto
  Si associano i nuovi tipi con le operazioni da usare nelle istanze

TIPI DI DATO ASTRATTO IN C++

ESEMPIO: Stack of chars

```
Class stack_of_char {
private:
      int size;
      char* top;
      char* s;
public
      stack_of_char(int sz) {
            top = s = new char [size = sz];
      }
      ~stack_of_char(){
            delete[] s;
      }
      void push (char c) {
            *top++ = c;
      }
      char pop(){
            return *--top;
      }
      int lenght(){
            return top –s;
      }
};
```



Il **COSTRUTTORE** è:

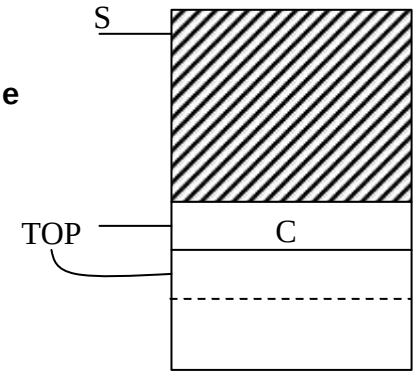
```
      stack_of_char(int sz) {
            top = s = new char [size = sz];
      }
```

Il **DISTRUTTORE** invece (cancello s):

```
      ~stack_of_char(){
            delete[] s;
      }
```

PUSH:

```
      void push (char c) {
            *top++ = c;   _____  deferenziazione
      }
```



equivale a:

```
      void push (char c) {
            *top = c;
            e poi sposto top di uno
      }
```

POP: Torna l'elemento precedente e sposta top.

TYPE CHECKING

Verifica il corretto uso delle variabili

Per linguaggi tipizzati staticamente: le variabli sono fissate ad un tipo prima del runtime

Per i linguaggi tipizzati dinamicamente: le variabili sono polimorfe

Il type checking può essere statico o dinamico

- Per linguaggi staticamente tipizzati, le variabili sono legate ai tipi prima dell'esecuzione del programma. Così facendo siamo sicuri che quando mandiamo in esecuzione il programma non ci saranno errori di tipo.

- Per alcune categorie di linguaggi tipizzati dinamicamente (es. linguaggi OO) possono assumere valori diversi a runtime. Vantaggio è la flessibilità

L_VALUE

Area di allocazione associata ad una variabile.

Ha particolare importanza il tempo di vita all'interno del quale il legame tra area di allocazione e variabile esiste.

Il meccanismo dell'allocazione di memoria può essere:

- **statico** (requisiti di memoria conosciuti prima dell'esecuzione) vs **dinamico**

- **automatico** (non richiede sforzo da parte del programmatore; alloco quando entro nel blocco in cui la variabile è definita e dealloco quando esco) vs **esplicito**

R_VALUE

E' il contenuto della zona di allocazione.

Le istruzioni attribuiscono l'r_value quando si ha l'assegnamento.
Il binding tipicamente è dinamico, eccezioni sono le costanti di tipo simbolico

> Esistono linguaggi che permettono di definire una costante inserendo nella definizione una variabile.
>
> esempio:

```
const float pi=3,1416
const circonf=2*pi*radius
```

INIZIALIZZAZIONE

Se al momento della dichiarazione la variabile non è inizializzata, ci chiediamo qual è il valore di una certa variabile quando creiamo il binding tra r_value ed l_value.

Ci sono tre possibilità:

- IGNORE = si lascia il valore che c'era prima nella locazione di memoria
- SYSTEM DEFINED INITIALIZATION = zero, blank… (in Java è così)
- UNDEFINED = assegna un valore "non inizializzata". Il problema (pur essendo la soluzione più pulita) è il costo a run-time per controllare che l'accesso sia fatto solo quando quella variabile viene definita.

PUNTATORI E RIFERIMENTI

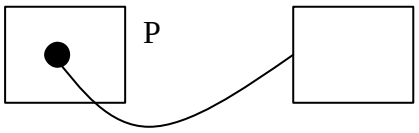
Alle variabili si può accedere in maniera indiretta, mediante l'r_value di un'altra variabile

Si dice che una variabile ha un riferimento (o un puntatore) ad un'altra variabile.

REFERENZIAZIONE

ESEMPIO PASCAL

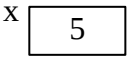
```
P:^integer; new(P)
```



Questa variabile non ha un nome ma è un intero accessibile mediante P

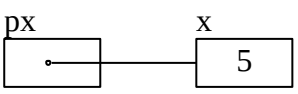
ESEMPIO C++

```
int x = 5;
```




```
int *px;
```


```
px = &x;
```



con **&** ottengo l'indirizzo della variabile

Genera un oggetto il cui r_value è 5; accessibile direttamente tramite x ; accessibile indirettamente tramite px.

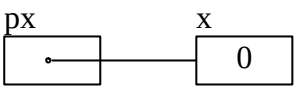
Ho due strade per accedere all'r_value

DEFERENZIAZIONE

E’ l’accesso mediante un puntatore

ESEMPIO C++

```
*px = 0;
```



ROUTINES

Sono unità in cui può essere scomposto un programma.
Permettono di delimitare una parte del problema e di gestire in maniera più efficiente lo sviluppo (dando da sviluppare parti diverse a diversi programmatori del team)

Distinguiamo in **funzioni** (ritornano un valore) e **procedure** (non ritornano un valore)

OGGETTI DELLE ROUTINES

Le routines hanno: nome, scope, l_value ed r_value.

L'attivazione è ottenuta tramite una chiamata di routine.

Possono accedere ad oggetti locali, non locali e globali.

Hanno un **HEADER** che definisce il tipo di routine tramite la SIGNATURE (tipo dei parametri e risultato) ed un **BODY**

Esempio:

```
int sum (int n)              //header
{                            //body
     int i, s; s = 0;
     for (i = 1; i <= n ; ++i)
     s+= i;
     return s;
}
```

Una routine viene ATTIVATA mediante l'invocazione (chiamata a sottoprogramma)
La FIRMA della routine è il numero dei parametri e i loro tipi + il tipo del parametro di ritorno..
Le chiamate di routine devono essere conformi al tipo di routine.

Lo scope segue le stesse regole viste per le variabili, e in più le routine possono accedere a variabili locali, non locali e globali a seconda delle regole di scope per il linguaggio.

L_VALUE ED R_VALUE

L_value: locazione dov'è memorizzato il corpo della routine

R_value: il corpo correntemente legato alla routine

In genere il binding è statico, e viene stabilito durante la traduzione.

Ci sono linguaggi che supportano:

variabili di tipo routine
Può essere assegnato un valore alla routine

Variabili di tipo "puntatore ad una routine"

```
int(*ps)(int);          //ps è un puntatore ad una routine
ps = &sum;              //sum è stata definita prima
int i = (*ps)(5);       //invoca la somma
```

In questi casi si dice che le routine sono "oggetti di prima classe", sono trattate come variabili

DICHIARAZIONE E DEFINIZIONE

La **DICHIARAZIONE** introduce l'header senza specificare il corpo della routine, la **DEFINIZIONE** specifica l'header ed il body.

ESEMPIO

```
int A(int x, int y);  //dichiarazione

float B(int z) {      //definizione
     int w, u;
     w = A(z, u);     //A è visibile in questo punto
}
```

Utile quando ci sono routine che si richiamano a vicenda. Senza il meccanismo della dichiarazione non saremmo in grado di conoscere il tipo.

RAPPRESENTAZIONE RUNTIME DELLE ROUTINES

E' costituita da un segmento di codice, mentre lo stato è dato dall'activation record.

CODICE: istruzioni (contenuto generalmente fisso)

RECORD DI ATTIVAZIONE: Contenuto che può cambiare

In particolare associato all'activation record c'è un ambiente di riferimento che contiene anche le variabili non locali.

ROUTINE RICORSIVE

Tutte le istanze sono composte da:

- stesso codice
- diversi record di attivazione

Occorre un legame dinamico tra il record di attivazione ed il suo segmento di codice

PARAMETRI

Servono a gestire il flusso di informazione tra sottoprogrammi diversi (parametri ingresso/uscita)

Distinguo in:

- **FORMALI**: sono i parametri del momento (indipendenti dal valore particolare che verrà passato)
- **ATTUALI**: reali, quelli della funzione (il valore specifico legato al parametro formale)

La corrispondenza avviene tramite:

METODO POSIZIONALE

```
routine S (F1, F2, … Fn);
call S (A1, A2, … An);
```

NAMED ASSOCIATION

```
procedure Example (A: T1; B: T2 := B1; C: T3);
Example (X, Y, Z);
Example(X, C=>Z);
Example(C=>Z, A=>X, B=>Y);
```

il parametro Z va al posto C mentre B resta non assegnato

ROUTINE GENERICHE

Per rendere generico il modulo sottoprogramma. Nell'esempio in basso specifichiamo in modo parametrico i parametri d'ingresso.
Ci sarà un binding dinamico in seguito (in precompilazione)

La parola chiave per definire una routine generica è: TEMPLATE

ESEMPIO

```
template <class T> void swap(T&a, T&b)

/* La funzione non ritorna alcun valore, è generica rispetto al tipo T
a e b fanno riferimento alla stessa locazione come i parametri attuali;
swap scambia i due valori */
{
      T temp = a;
      a = b;
      b = temp;
}
```

Per effettuare una chiamata il template deve essere inizializzato (è implicito nel C++)

OVERLOADING

Diamo lo stesso nome ad entità diverse.

ESEMPIO 1

```
int i, j, k;
float a, b, c;
i = j + k;
a = b + c;
```

L'operatore + è overloaded (addizione di interi oppure addizione di float)

ESEMPIO 2

```
a = b + c + b()        //b denota sia una variabile che una routine
a = b() + c + b(i)     //chiamata a due routine diverse
```

Nel secondo caso sappiamo qual è la routine da chiamare in base ai parametri

ALIASING

E' l'opposto dell'overloading. Due o più nomi denotano la stessa entità.

ESEMPIO 1

```
int i;
int fun(int &a);
{
     a = a +1;
     printf(i);
     …
}
main()
{
     x = fun(i);
}
```

i ed a denotano lo stesso oggetto

ESEMPIO 2

```
int x = 0;
int *i = &x;
int *j = &x;
```

**i ed *j sono alias. x è accessibile direttamente oppure tramite i due puntatori che sono alias.*

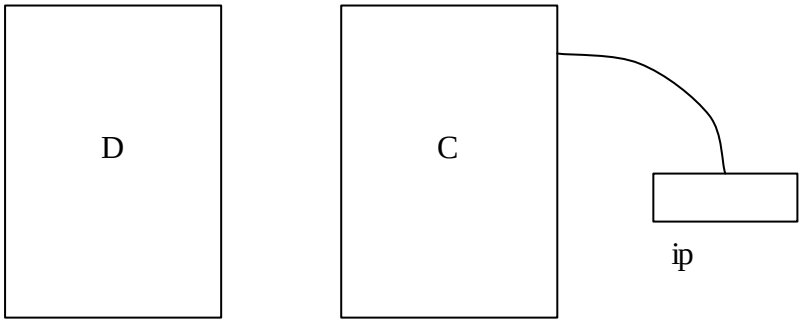
E' passibile di errori e complica la leggibilità del programma.

SIMPLESEM

Vediamo nel dettaglio gli aspetti semantici dei linguaggi di programmazione.
Diamo una semantica di tipo operazionale, descriviamo la semantica facendo riferimento da una macchina astratta di cui conosciamo il funzionamento.

La macchina imita le macchine reali basate sul modello di Von Neumann.

E' un processore semantico astratto.



Abbiamo due memorie:

- C → Code memory
- D → Data memory

viste come un array di celle.

C viene acceduta tramite un registro detto Ip (Instruction pointer) che da il riferimento all'istruzione corrente.

INTERPRETATION CYCLE

1. Ottiene l'istruzione corrente C[ip]
2. Incrementa ip
3. Esegue l'istruzione corrente

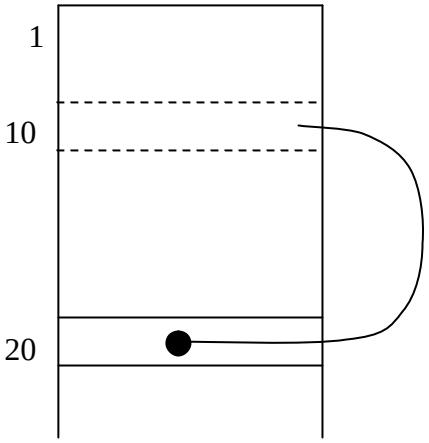
NOTAZIONE

D[X], C[X] sono i valori memorizzati nella cella X

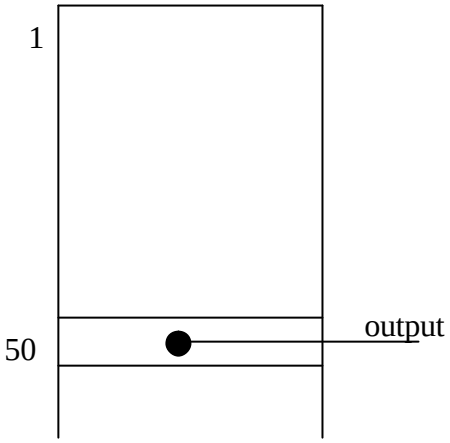
X → l_value
D[X] → r_value

ESEMPI

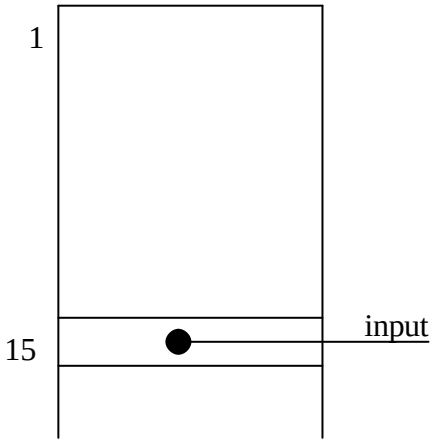
set 10, D[20] → Legge il valore memorizzato nella locazione 20 di D e lo scrive nella locazione 10



set write, D[50] → l'output è il valore memorizzato nella locazione 50



set 15, read → il valore letto viene memorizzato nella locazione 15



Sono valide anche espressioni complesse come ad esempio:

set 99, D[15]+D[33]*D[4]

ESEMPI: CONTROLLO DI FLUSSO

jump 47 → la prossima istruzione sarà memorizzata nel posto 47

jumpt 47, D[3]>D[8] → il salto avviene sono se è verificata la condizione

ESEMPI: INDIRECT ADDRESSING

set D[10], D[20]

jump D[13]

CLASSIFICAZIONE DEI LINGUAGGI IN BASE ALLA LORO STRUTTURA RUN-TIME

LINGUAGGI STATICI

- La quantità di memoria necessaria è conosciuta prima dell'esecuzione
- La memoria può essere allocata prima dell'esecuzione
- Non c'è ricorsione

ESEMPI: FORTRAN, COBOL

LINGUAGGI STACK-BASED

- La quantità di memoria necessaria è sconosciuta al momento della compilazione, ma l'uso è predicibile e segue una disciplina last-in first-out (tipo pila)
- E' possibile usare una politica predefinita per allocazione e rilascio della memoria

ESEMPI: ALGOL60

LINGUAGGI COMPLETAMENTE DINAMICI

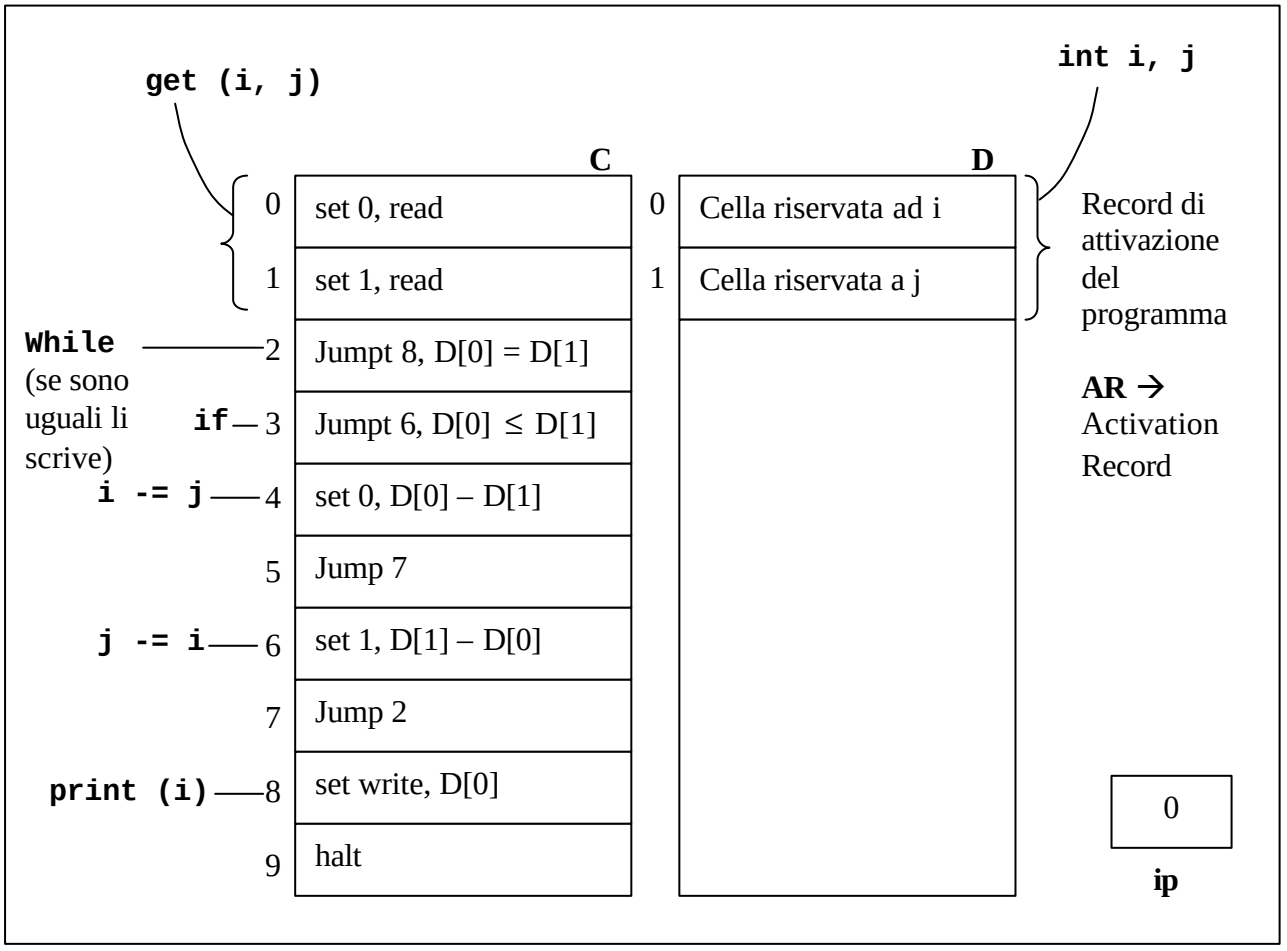
- Uso della memoria impredicibile
- Allocazione dinamica
- D è trattato come un **HEAP** (mucchio)
  La memoria è prelevata in questo mucchio in base alle necessità

C1: LINGUAGGIO CON ESPRESSIONI SEMPLICI

Consideriamo un sottoinsieme del C con tipi semplici ed espressioni (senza funzioni)

```
main(){
     int i, j;
     get(i,j);
     while(i != j)
          if (i > j)
               i -= j;          //i = i – j
          else
               j -= i;
     print(i);
}
```

STATO DEL SIMPLESEM



Le prime due istruzioni sono dichiarazioni di variabili, le metto nella memoria dei dati.
Nella cella 0 metto i, nella cella 1 metto j.

C'è poi l'istruzione in imput get(i,j)
Con "set 0, read" e "set 1, read", metto i valori di i e j nelle celle 0 e 1.

Il while viene espresso tramite un condizionale, poi ho l'if. Concludo con la stampa del valore.

C2: C1 + ROUTINES

Le routine possono accedere a:

- dati locali
- dati globali non dichiarati internamente

Non usiamo ricorsione, valori di ritorno e parametri in ingresso.

```
int i = 1, j = 2, k = 3;

alpha(){
     int i = 4, l = 5;
     ...
     i += k + l;
     ... };

beta(){
     int k = 6;
     ...
     i = j + k;
     alpha();
     ... };

main (){
     ...
     beta();
     ... };
```

Il main è la routine principale che invoca beta(), la quale a sua volta invoca alpha()

SIMPLESEM

Vediamo la gestione della memoria e lo stato del simplesem.

La dimensione dell'activation record (per la routine) può essere determinata a tempo di traduzione.

Posso anche allocare gli AR prima dell'esecuzione

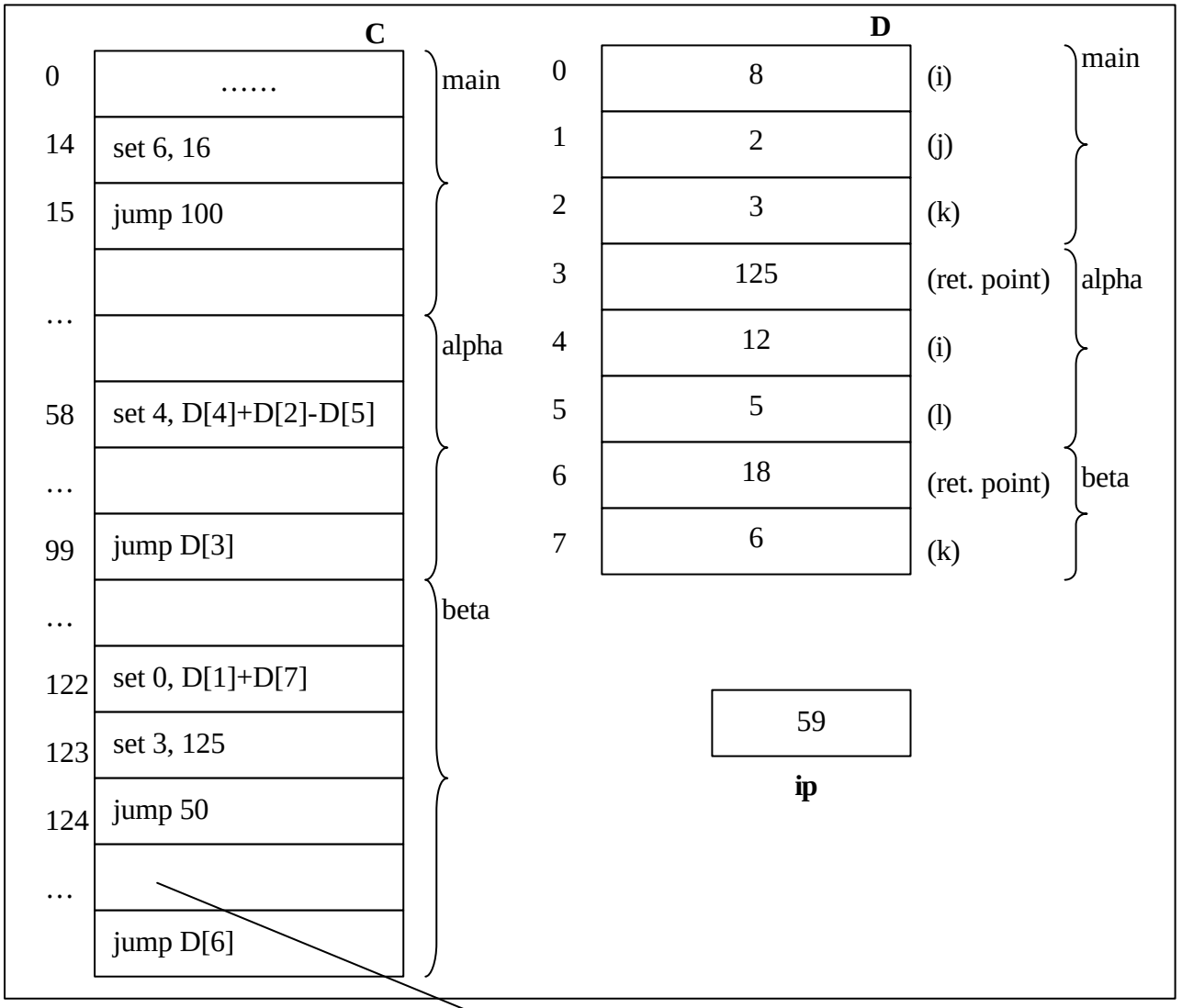
Ho un vantaggio in termini di velocità, non avendo overhead per l'allocazione.
Lo svantaggio è in termini di spreco di memoria in quanto devo allocarla per tutte le routine anche se alcune di esse non verranno mai eseguite.

Per i dati globali ho 3 celle (0, 1 e 2)

Il main non ha bisogno di informazioni di controllo poiché non è invocato da nessuno e non ha variabili locali.

Per alpha() ho bisogno di due variabili locali.
Ho anche bisogno del "return pointer" che contiene a run-time l'indirizzo di C a cui deve tornare il programma quando termina la routine.

Di seguito uno snapshot della macchina quando si è eseguita l'istruzione i += k + l;



Il return pointer di alpha() si trova dentro beta() ; le altre due celle servono per le variabili locali i e j

beta() ha due celle: una per il return pointer che mi fa saltare all'istruzione successiva all'invocazione di beta() che è la 15 ; ed una per la variabile k.

Vediamo com'è tradotta l'invocazione di una procedura.

- Per prima cosa devo settare il return pointer → set 6, 16

- Poi devo terminarla: è il chiamante a dire dove tornare tramite il return point.
  jump D[3], equivale a saltare in 125 (non potevo mettere direttamente jump 125 poiché tale valore è stato inserito in modo dinamico.

GESTIONE DELLA MEMORIA

- La dimensione degli AR (activation record) è determinata durante la traduzione
- Gli AR possono essere allocati prima dell'esecuzione
- Le variabili sono legate alla memoria D prima dell'esecuzione
- L'allocazione statica non è obbligatoria

  Vantaggio dell'allocazione statica: non c'è overhead a run-time
  Svantaggio: lo spazio inutilizzato non viene allocato

COMPILAZIONE SEPARATA PER C2

Una modifica che si può fare è il supporto per la compilazione separata delle varie unità.
Posso spezzare il file in più parti ciascuno dei quali contiene un modulo.

Ho un file con le variabili globali e due con le due routine alpha() e beta()

FILE 1

```
int i = 1, j = 2, k = 3;
extern beta ( );
main ( ) {
      . . .
     beta ( );
      . . .
}
```

FILE 2

```
extern int k;
alpha ( ) {
     . . .
}
```

FILE 3

```
extern int i, j;
extern alpha ( );
beta ( ) {
     . . .
     alpha ( );
     . . .
}
```

Nel primo file ho il richiamo a beta() e devo inserire la parola extern per dire che la definizione di beta() sta in un altro file.

In compilation time:

Non posso stabilire il contenuto della memoria poiché compilo uno solo dei file alla volta.
Quando ad esempio compilo alpha(), non so ancora qual è l'activation frame poiché non so quando verrà chiamata.
So quanto è grande alpha() e quindi l'offset di una variabile a partire dalla base dell'activation frame.
Analogamente le chiamate alla routine in fase di compilazione non possono essere legate alla memoria codice, non so quante unità ho.
Devo lasciare il codice rilocabile.

Link time

Tutte le cose lasciate in aria vengono risolte dal linker
In particolare: decidere dove mettere gli AR e risolvere gli indirizzi di jump nei return pointer.

RICORSIONE

Non ho più una gestione statica della memoria bensì dinamica.

Lo scoping e la tipizzazione sono statici e i dati sono allocati staticamente (non abbiamo le new e le malloc) e non ci sono parametri d'ingresso.

```
int n;                          //variabile globale
int fact()
     {
     int loc;
     if(n > 1)
          {
          loc = n--;
          return loc * fact();
          }
     else
          return 1;
     }
main()
     {
     getint(n);
     if (n >= 0)                //il main legge n, se è ≥ 0 chiama la funzione fact()
          printf(fact());
     else
          printf("input error");
     }
```

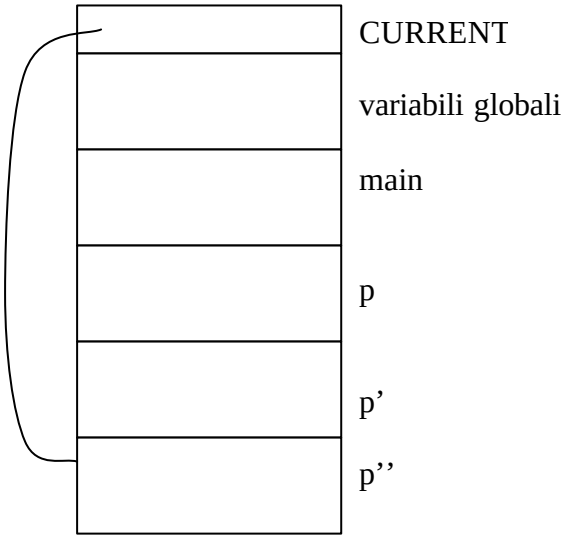
CONSEGUENZE DELLA RICORSIONE

Attivazioni diverse hanno:

- lo stesso segmento di codice
- diversi record di attivazione

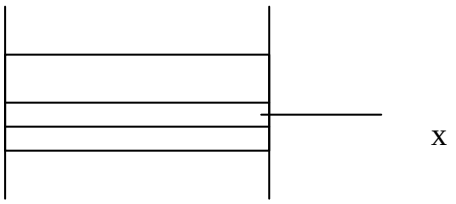
Le variabili sono legate all'offset al momento della traduzione, posso legare le variabili al loro offset ma non ad una cella di memoria (non so dove verrà usata); saprò ad esempio che è la seconda cella, ma non so a partire da quale offset.

Il passo finale viene fatto in esecuzione
La cella 0 di D è usata per l'indirizzo base del record di attivazione (AR) corrente

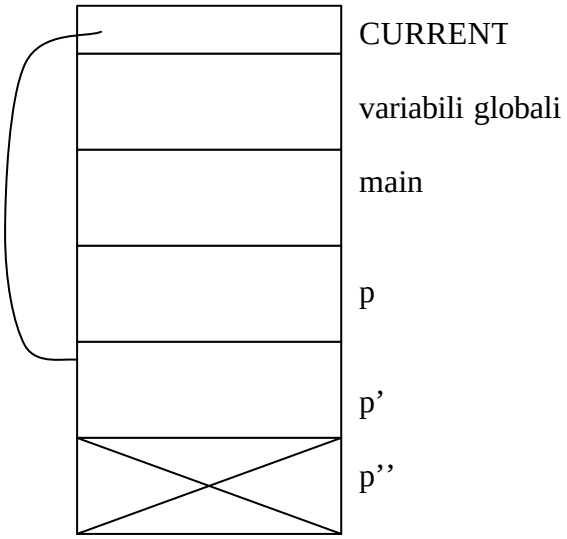



```
main {
      p();
}

p(){          //ricorsiva
      p();
}
```

L'unica cosa che posso sapere è, dato l'activation frame, dove sta una certa variabile x.

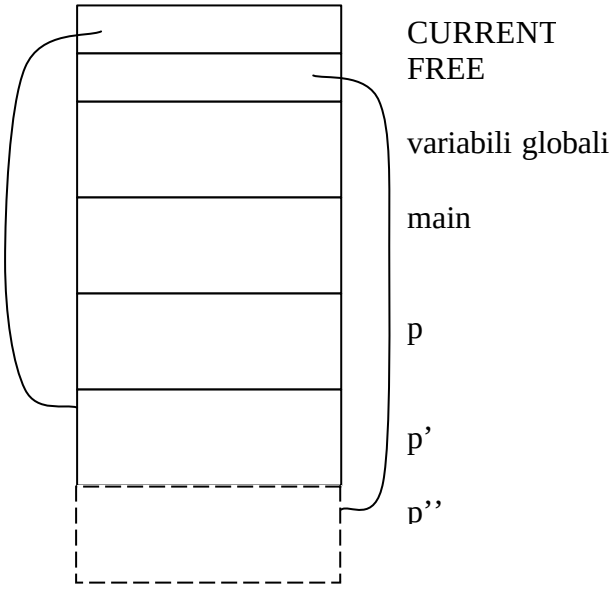


A run-time devo sapere qual è l'AF corrente poiché in p, p' e p'' avrò valori diversi per x.
Questa informazione è memorizzata nella cella zero detta CURRENT.
Quando p'' termina, può essere deallocato l'AR e il valore di current cambia.



Risolve il problema della rimozione.

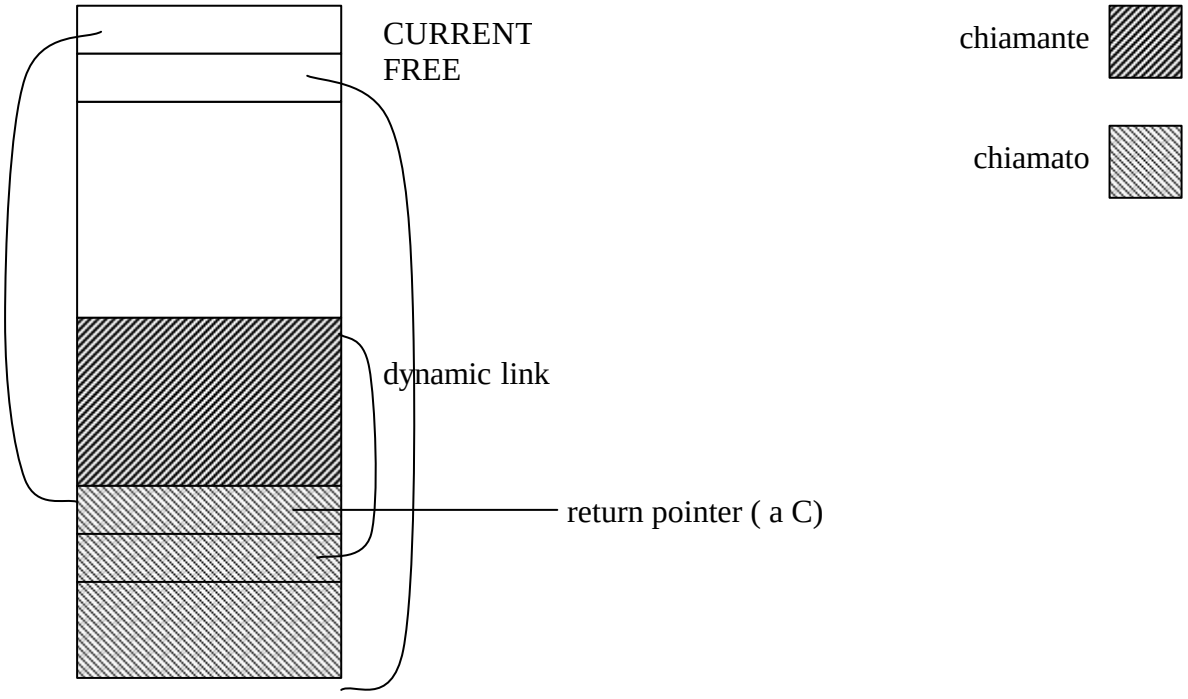
E quando alloco?
Devo avere un altro puntatore FREE che dice qual è la prima cella libera in memoria a partire da cui posso inserire un nuovo AR



Quando aggiungo il nuovo AR:

- **current** assume il valore vecchio di free
- **free** va a "nuovo current + dimensione di AR"

Quando termina la routine:
devo sapere dove portare current, mi serve un ulteriore informazione di controllo che il puntatore all'AR precedente (si chiama DYNAMIC LINK), memorizzato nella seconda cella. Setterò il valore di current al Dynamic Link.
Nella prima cella c'è il RETURN POINTER



Vediamo in dettaglio cosa succede.

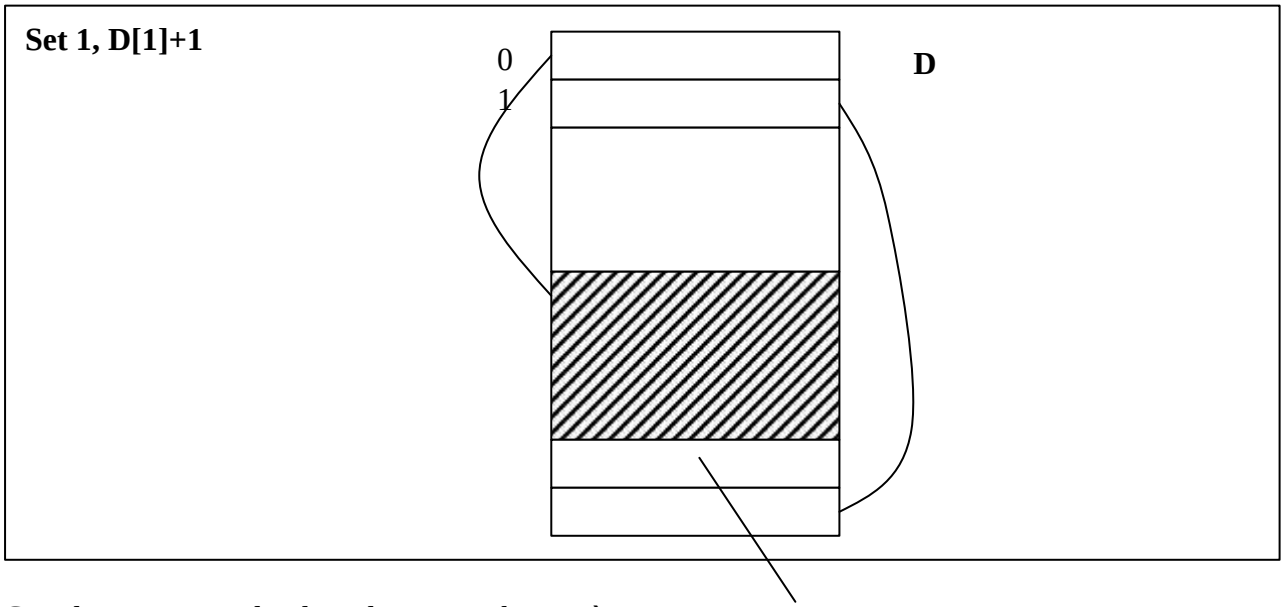
Chiamata di routine

Set 1, D[1]+1	Setta il valore di ritorno
Set D[1], ip+4	Setta il punto di ritorno
Set D[1]+1, D[0]	Setta il link dinamico
Set 0, D[1]	Setta CURRENT
Set 1, D[1]+AR	Setta FREE
Jump start_addr	

Ritorno dalla routine

Set 1, D[0]	Setta FREE
Set 0, D[D[0]+1]	Setta CURRENT
Jump D[D[1]]	torna al punto di ritorno memorizzato

Passo per passo:



Creo lo spazio per il valore di ritorno che verrà memorizzato qui.
Lo faccio incrementando FREE di una unità.

Set 1, D[1]+1

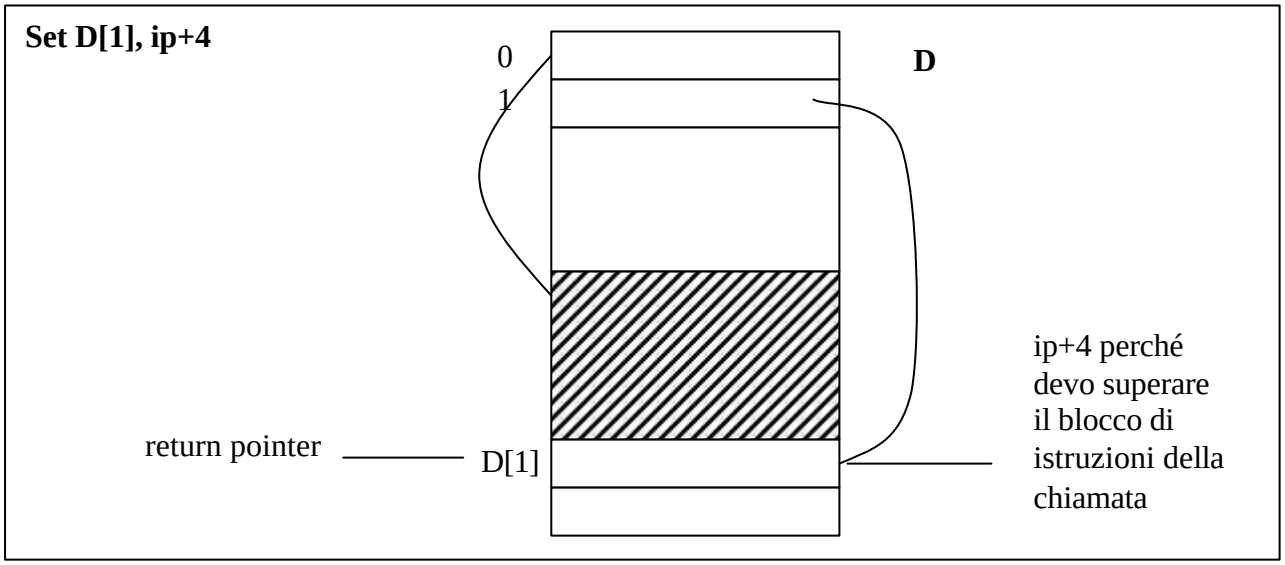
valore corrente di FREE

Poi devo costruire l'AR del nuovo frame

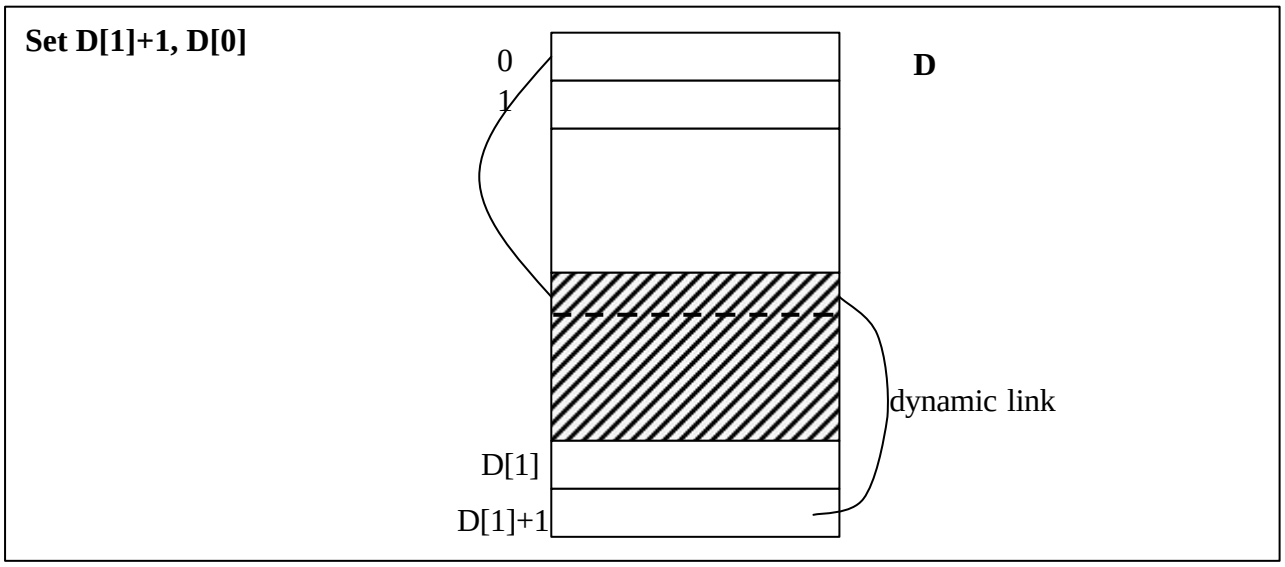
Riservo la cella per il return pointer

Set D[1], ip+4

le quattro istruzioni che servono per gestire l’



Devo poi mettere il dynamic link.

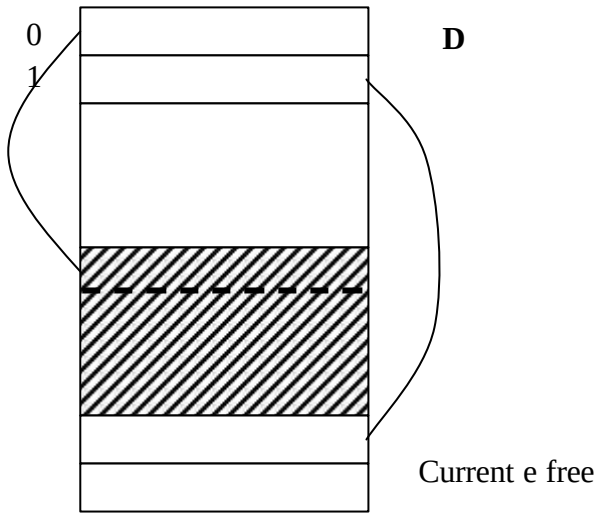


Setto la prima cella libera +1 a D[0]

Infine devo settare i nuovi valori di current e free

Set 0, D[1]
(il nuovo valore di D[0])

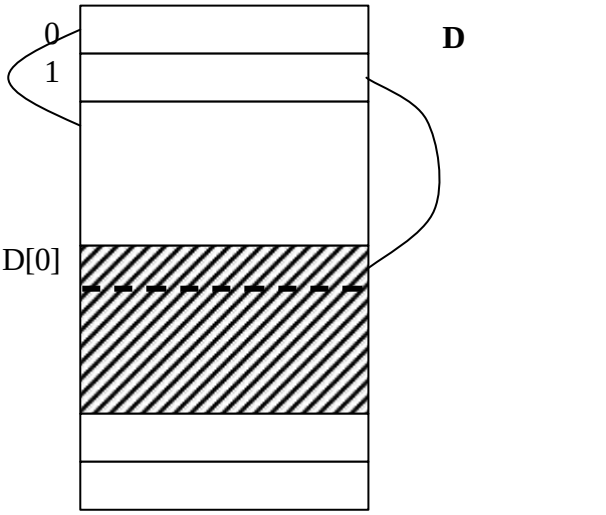
Set 1, D[1]
(il nuovo valore di D[1])



Per il ritorno da una routine: occorre cambiare i valori di D[1] e D[0]

Set 1, D[0]
(FREE)

Set 0, D[D[0] +1]



FREE deve puntare al vecchio CURRENT

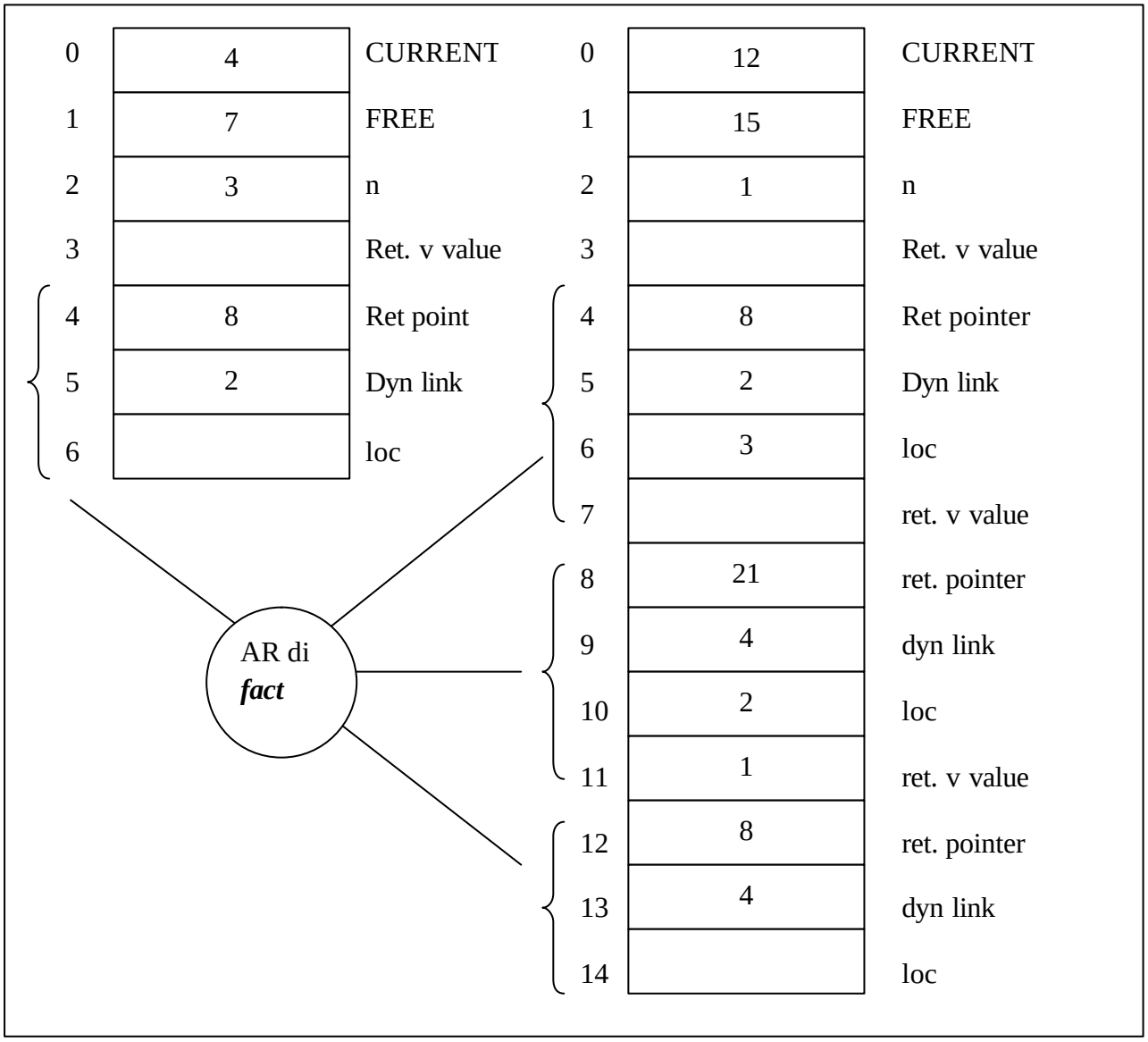
e CURRENT devo portarlo indietro grazie al DYNAMIC LINK salvato precedentemente.

Poi salgo al punto di ritorno (al segmento di codice) con Jump D[D[1]]

ESEMPIO: FATTORIALE

0	set 2, read	Legge il valore di n
1	jumpt 10, D[2]<0	testa il valore di n
2	set 1, D[1]+1	chiamata funz. “fact”, spazio x i ris. salvato
3	set D[1], ip+4	setta il punto di ritorno
4	set D[1]+1, D[0]	setta il link dinamico
5	set 0, D[1]	setta CURRENT
6	set 1, D[1]+3	setta FREE
7	jump 12	la riga 12 è l’inizio del codice per il fattoriale
8	set write, D[D[1]-1]	scrive il risultato della chiamata
9	jump 13	fine della chiamata
10	set write, “imput error”	
11	halt	fine del main
12	jumpt 23, D[2]<=1	testa il valore di n
13	set D[0]+2, D[2]	assegna n a *loc*
14	set 2, D[2]-1	decrementa n
15	set 1, D[1]+1	chiamata della funz. “fact”, x i ris.
16	set D[1], ip+4	
17	set D[1]+1, D[0]	
18	set 0, D[1]	
19	set 1, D[1]+3	3 è la dimensione dell’AR di fact
20	jump 12	
21	set D[0]-1, D[D[0]+2]*D[D[1]-1]	
22	jump 24	
23	set D[0]-1, 1	memorizzo il valore di ritorno
24	set 1, D[0]	
25	set 0, D[D[0]+1]	
26	jump D[D[1]]	

SNAPSHOT DELLA MACCHINA



BLOCCHI NIDIFICATI

Introduciamo la possibilità di inserire dei blocchi nel programma.

Abbiamo una funzione f() che dichiara delle variabili, all'interno dei blocchi possiamo dichiarare delle variabili locali che mascherano quelle più esterne.

```
int f();
{
   int x,y,w;                    //1
   while (…)
   {   int x,z;                  //2
      while (…
      { int y;                   //3
       }
        if (…)
        {  int x,w;              //4
         }
     }
   if (…)
   {       int a,b,c;            //5
   }
}
```

Abbiamo due possibilità:

- visibilità statica: definiamo un AR speciale che tiene conto di tutte le variabili (più semplice, risparmiamo tempo di esecuzione)

- visibilità dinamica: estendiamo le dimensioni dell'AR allocando memoria man mano che si entra nel blocco e le deallochiamo quando usciamo (migliore uso della memoria)

REGOLE DI VISIBILITÀ STATICHE:

- Se all'interno di un blocco dichiaro variabili, esse non sono visibili all'esterno

- Se una variabile dichiarata in un blocco esterno è ridichiarata all'interno, essa non è più visibile

  Nell'esempio: x non è visibile all'interno del primo while, y è visibile all'interno del primo while ma non del secondo.

AR OVERLAYED

Per minimizzare lo spreco di memoria usiamo gli overlayed AR.
Le variabili nei blocchi non sono tutte vive allo stesso momento. Ad esempio quando eseguo //5, mi interessano solo "a, b, c, d" e non lo stato del blocco while e di quelli in esso contenuti.

Usiamo lo stesso insieme di celle per memorizzare variabili tra di loro in mutua esclusione.

Return pointer
Dynamic link
x in //1
y in //1
w in //1
x in //2 – a in //5
z in //2 – b in //5
y in //3 – x in //4 – c in //5
w in //4 – d in //5

a, b, c, d possono essere messe al posto di x, y, z, w (del ciclo while interno che non servono più)
➔ a, b, c, d sono OVERLAYED

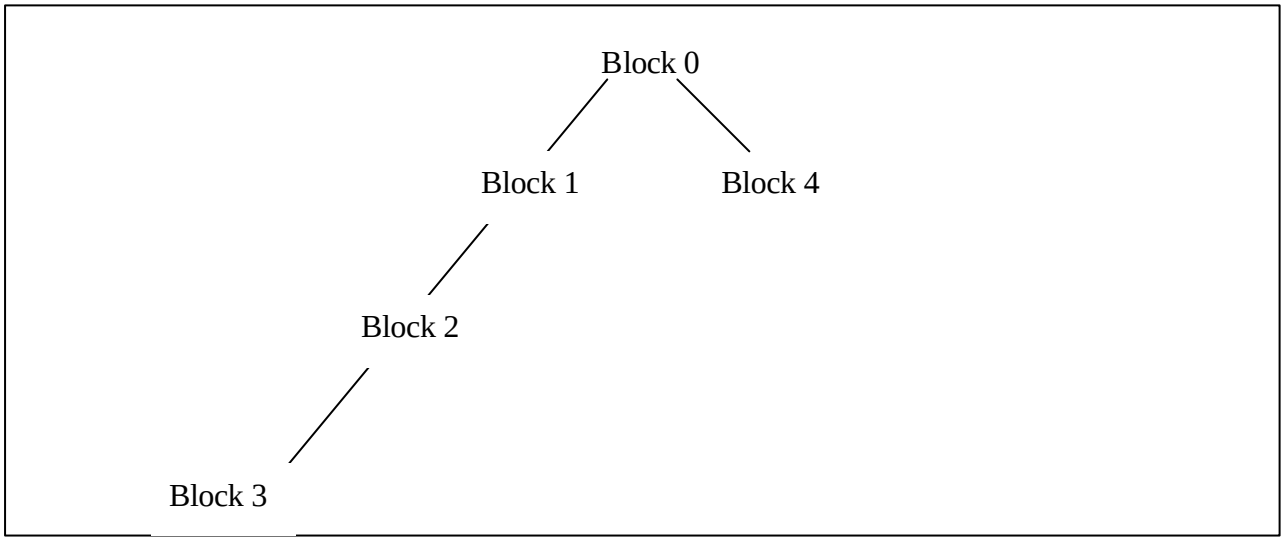
ROUTINE NIDIFICATE

Possiamo avere dichiarazioni di routine dentro altre routine (non si può fare in C, C++ e Java)
f1 dichiara due variabili ed una routine f2 che è localmente visibile in f1 (il main non la vede)
analogamente f2 dichiara f3

```
int x,y,z;
f1()                              //f1
{
      int t,u;
      f2()                        //f2
      {
            int x,w;
            f3()                  //f3
            {
                  int y,w,t;
            }
            x=y+t+w+z;
      }
}
main();
{
      intz,t;
}
```

E’ possibile descrivere la struttura tramite un albero detto STATIC NESTING TREE.

L’albero è utile perché è il modo in cui posso vedere come sono annidate le routine e mi permette quindi di capire qual è la variabile corretta da usare.



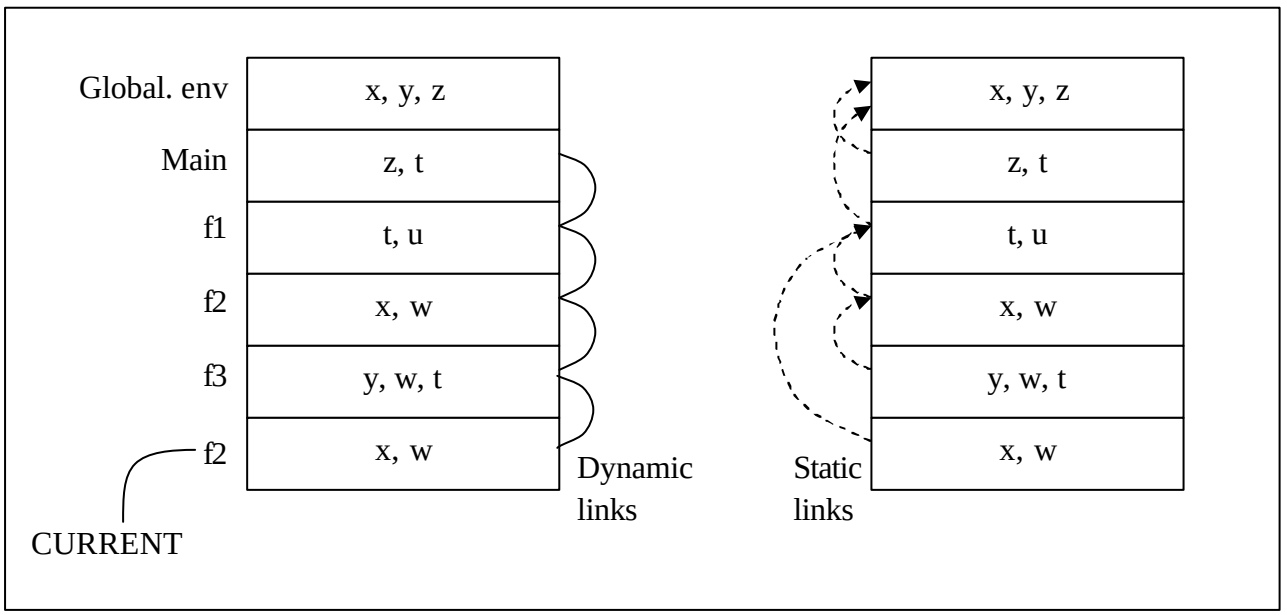
COME ACCEDERE AGLI AMBIENTI NON LOCALI

Vediamo la struttura a pila.
Abbiamo l'istruzione corrente e la catena dinamica per tornare agli AR precedenti.

Il main chiama f1 la quale chiama f2. f2 chiama f3 la quale ha una chiamata ricorsiva ad f2

Consideriamo l'operazione: x=y+t+w+z;
Con la catena dinamica non riusciamo a capire qual è la y a cui dobbiamo accadere.
Se seguiamo la catena dinamica troviamo la y di f3 che non è quella a cui dobbiamo accedere (sarebbe corretto con regole di scoping dinamico e non statico)

Bisogna complicare il run-time del linguaggio introducendo delle informazioni di tipo statico.



Sono i link ststici, che permettono di avere informazioni di tipo statico.

REFERENZIARE LE VARIABILI NON LOCALI

Vediamo come si può modellare.

Inserisco i link statici e poi ripercorro la catena statica all'indietro.

Posso calcolare a compile-time la distanza lungo la catena dei link statici dove trovo la variabile che mi interessa (*distance*). L'altra informazione che mi serve è l'*offset*.

Le referenze alle variabili possono essere legate staticamente a

(distance, offset)

Indirizzo relativo nell'AR

Numero di passi lungo l'SC (punta ad un AR)

Possiamo modellarlo con una funzione **Function Frame Pointer (fp)** che produce un puntatore ad AR che è a "d" passi da "current"

```
fp(d) {

      if d=0 then D[0]          //accedo al frame corrente

      else D[fp(d-1)+2]         //accedo referenziando il contenuto della cella
                                decrementando il frame pointer di 1 e poi aggiungendo
                                2 (perché lo static link sta in posizione 2)
}
```

Accedo alla variabile: `D[fp(d) + o]`

distanza offset

Vediamo come viene gestito a livello di simplesem

```
      Set 1, D[1]+1          Setta il valore di ritorno
      Set D[1], ip+4         Setta il punto di ritorno
      Set D[1]+1, D[0]       Setta il link dinamico
→     Set D[1]+2, fp(d)      Setta il link statico
      Set 0, D[1]            Setta CURRENT
      Set 1, D[1]+AR         Setta FREE
      Jump start_addr
```

Aggiungiamo un istruzione per installare il link statico.

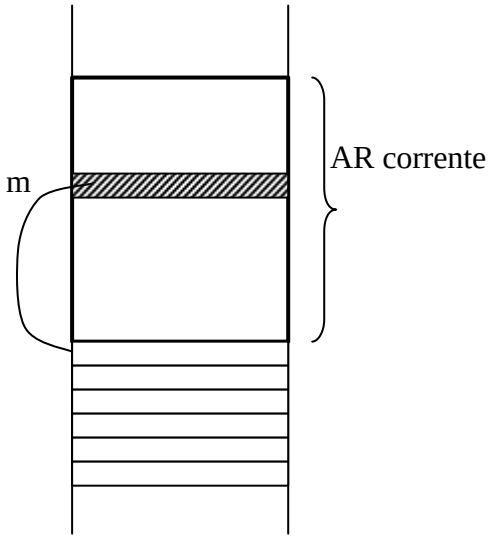
dice dov'è il record base

ARRAY DINAMICI

```
type VECTOR is array (INTEGER range <>);
A:VECTOR(1..N);
B:VECTOR(1..M);
```
//definisce un array con indici liberi.

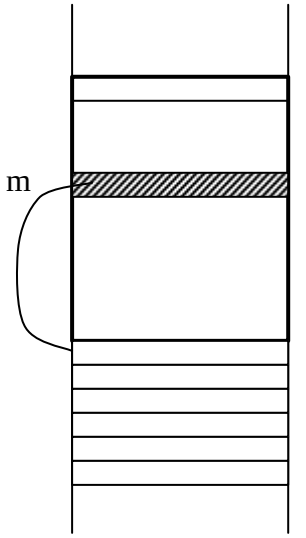
N ed M devono essere fissate ad un valore intero quando le dichiarazioni sono elaborate a run-time

In SIMPLESEM ho un puntatore riservato per ogni array dinamico invece dello spazio per le variabili in quanto non sappiamo qual è la lunghezza dell'array.



Gli oggetti array sono allocati in cima agli ultimi AR allocati (a run-time)

L'accesso agli array viene eseguito indirettamente tramite puntatore



devo sommarci il valore che ottengo mettendo al frame corrente lo spiazzamento s D[D[0]+s]]

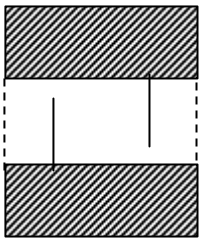
A[I] = 0 set [D[D[0]+m]+D[D[0]+s]], 0

Indirizzo base di A Valore di I

PUNTATORI

Consentono una gestionepiù flessibile dello stato del programma e di estendere il tempo di vita delle variabili.

```
struct node {
      int info;
      node *left;
      node *right;
};
node *n = new node;
```



I nodi non possono essere allocati nello stack, vengono allocati nell'heap

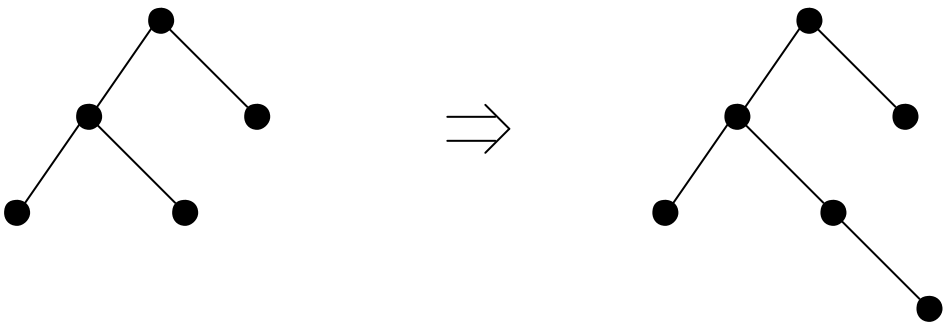
> **NOTA**: definizione di HEAP
> L'heap viene usato per allocare e liberare oggetti in modo dinamico da far utilizzare al programma.
> Le operazioni sull'heap vengono richieste quando:
> 1. Il numero e le dimensioni degli oggetti necessari al programma non si conoscono in anticipo.
> 2. Un oggetto è troppo grande per essere allocato nello stack.
>
> L'heap usa parti di memoria all'esterno di ciò che è allocato per il codice e lo stack durante l'esecuzione

Vediamo perché è necessario allocare i nodi nello heap.

Consideriamo la seguente procedura:

```
m(i) {
      int i;
      n->left = new node;
      node.info = i;
};
```

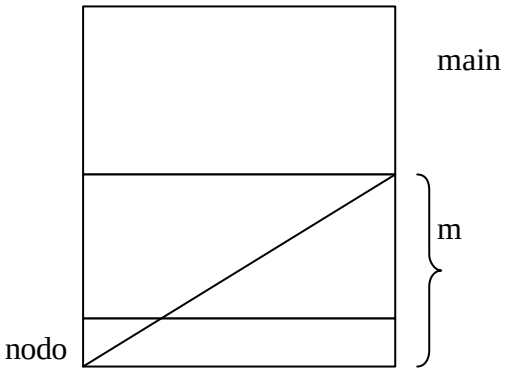
Crea un nuovo nodo.



Ecco il main del programma:

```
main() {
      ...
      m(i);
      ...
};
```

Allocando nello stack avrei dei problemi perché quando deallochiamo la routine alla terminazione della procedura, l'AR di m() è deallocato.



il nodo viene buttato via.

Nel nostro caso vogliamo far si che il nuovo nodo *viva* anche dopo la terminazione della procedura.

Viene fatto usando lo heap.

Gestiamo lo heap prendendo gli indirizzi al contrario rispetto allo stack a partire dall'indirizzo massimo della memoria con indirizzi decrescenti.

DYNAMIC TYPING E SCOPING

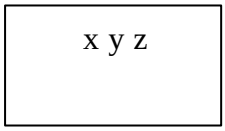
DYNAMIC TYPING: una variabile nell'AR è rappresentata da un puntatore all'oggetto nell'heap (la dimensione può cambiare dinamicamente)

DYNAMIC SCOPING: il vincolo dinamico supporta l'accesso ad oggetti non locali

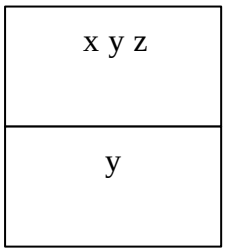
Consideriamo il seguente esempio:

```
sub2 ( )
{
declare x;
. . . x . . .;
. . . y . . .;
. . .
}

sub1 ( ) {
     declare y;
      . . . x . . .;
     . . . y . . .;
     sub2 ( );
     . . .
}

main ( ) {
     declare x, y, z;
```

```
      z = 0;
      x = 5;
      y = 7;
      sub1;
      sub2;
      . . .
}
```



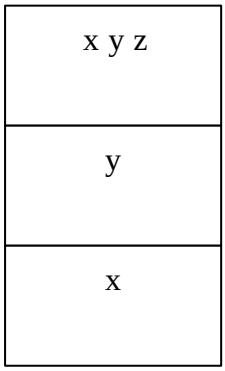
main



main

sub1

x la trovo seguendo la catena dinamica (lo scoping è dinamico)



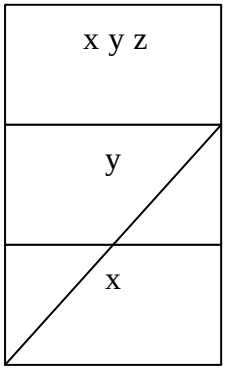
main

sub1

sub2

la y è quella di sub1 (la più recente nella catena dinamica)

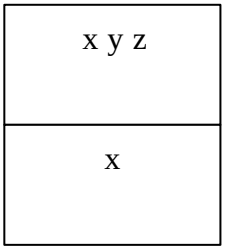
poi il main chiama di nuovo sub2



main

sub1

sub2



main

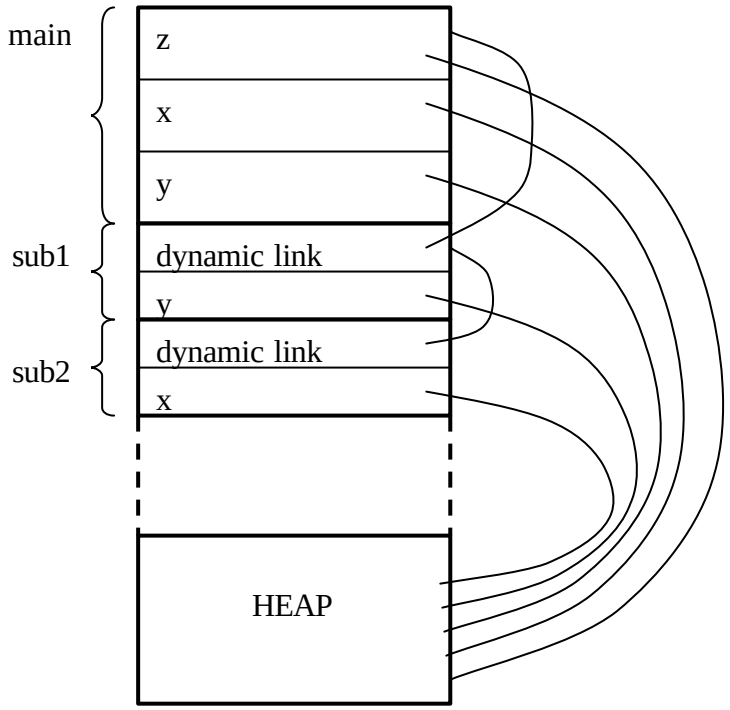
sub2

In questo caso, uso la x locale e la y del main (non più quella di sub1)

TIPIZZAZIONE DINAMICA

Non siamo in grado di determinare a priori la dimensione dell'AR poiché diversi tipi hanno dimensioni diverse

Possiamo memorizzare le variabili nello heap.



Mantengo l'AR di dimensione fissa memorizzando il puntatore alla variabile nello heap in cui c'è memorizzato lo stato (il valore) associato alla variabile.

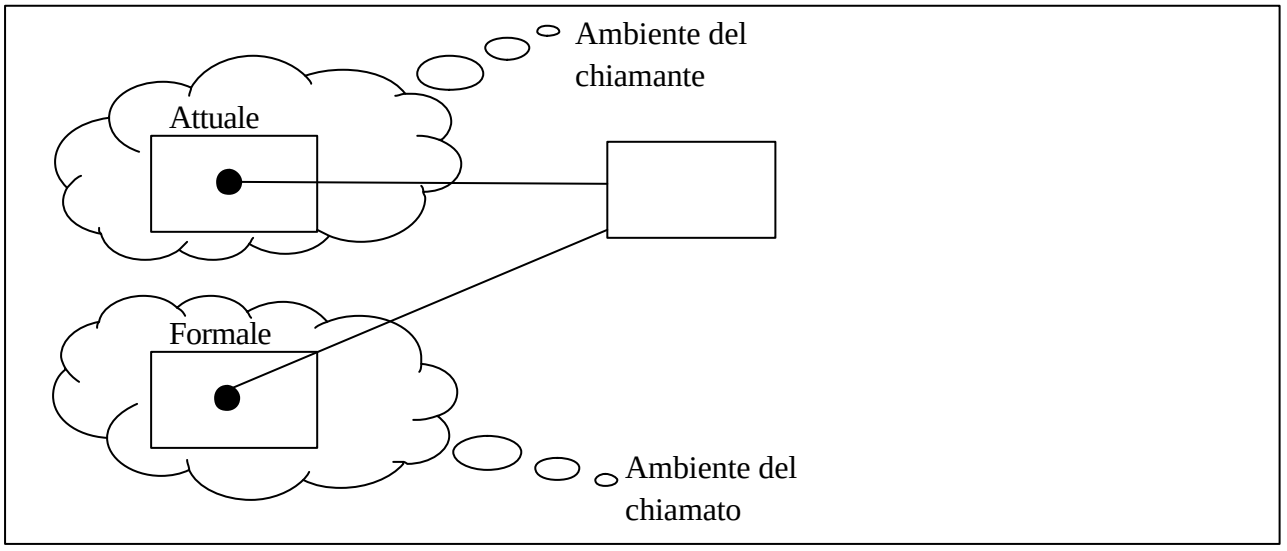
PASSAGGIO DEI PARAMETRI

La comunicazione fra Programma Chiamante e Programma Chiamato può avvenire oltre che attraverso l'uso delle variabili Globali (vedi righe precedenti) anche con altre modalità utilizzando i Parametri.
I Parametri sono delle variabili rispettivamente del Programma Chiamante (Parametri Attuali) e del Programma Chiamato (Parametri Formali).
I Parametri Attuali sono rappresentati da una lista di variabili già definite nel Programma Chiamante. Questa lista viene indicata nell'istruzione di chiamata del sottoprogramma nel Programma Chiamante.
I Parametri Formali sono una lista di definizione di variabili collocata nella prima istruzione del sottoprogramma. Il primo Parametro Attuale deve essere dello stesso tipo del primo Parametro Formale il secondo parametro attuale dello stesso tipo del secondo parametro formale ... . Al momento della chiamata del sottoprogramma il primo Parametro Attuale viene messo in relazione con il primo Parametro Formale, il secondo Parametro Attuale viene messo in relazione con il secondo Parametro Formale .... .

- By REFERENCE (per indirizzo)
- By COPY
- By NAME

CALL BY REFERENCE

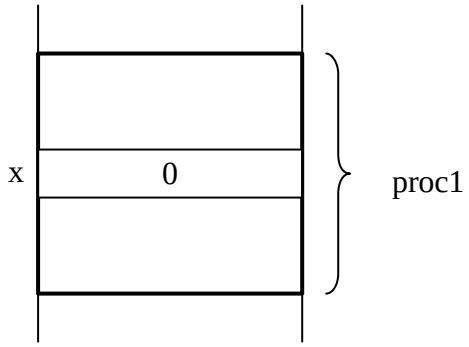
Detto anche call by sharing
Passaggio per Indirizzo anche detto per Referenza: il Parametro Formale assume l'indirizzo del Parametro Attuale e quindi va a rappresentare la stessa area di memoria.
Il chiamante passa l'indirizzo del parametro attuale. Il riferimento al parametro formale è trattato come una referenza indiretta.



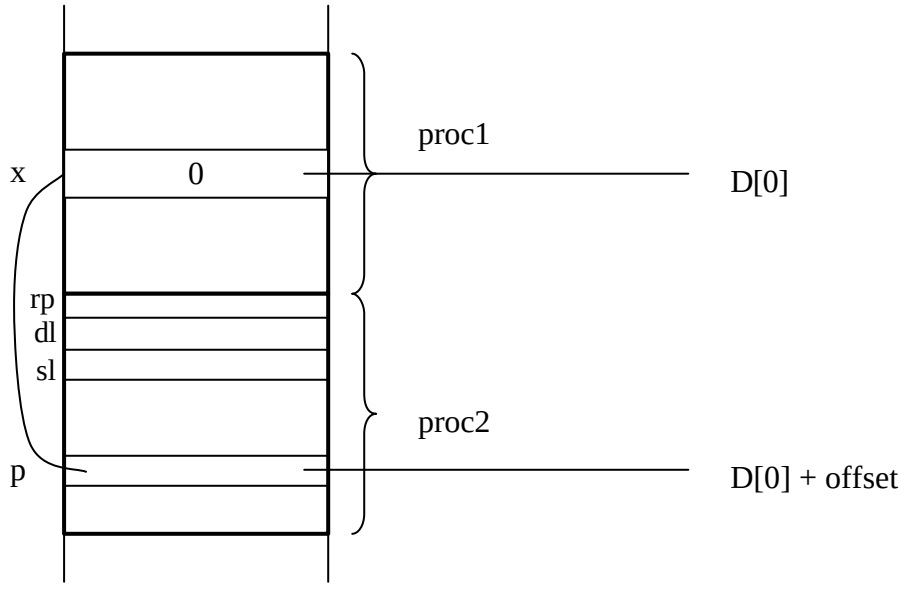
Ho l'ambiente del chiamante con il parametro attuale e quello dei chiamato con il parametro formale che punta al parametro attuale.

Esempio:

```
proc2(int & p){
     p++;                          //p parametro formale
};

proc1(){
     int x = 0;
     proc2(x);                     //x parametro attuale
};
```



Quando chiamiamo proc2( )



In p memorizzo un puntatore del parametro attuale

Il chiamante mette in “D[0] + offset” il risultato di “fp(d) + o”

indirizzo del frame che contiene il valore effettivo offset

set D[0] + off, fp(d) + o

Consideriamo un caso più complesso:

```
proc3(int & p){
     p++;
};

proc1(){
     int x = 0;
     proc2(x);
};

proc2(int & p){
     proc3(p);
};
```

Ho un passaggio indiretto.

proc2( ) invece di operare direttamente sul parametro formale invoca proc3( ).

Devo aggiungere una indirettezza in più.

set D[D[0] + off], fp(d) + o

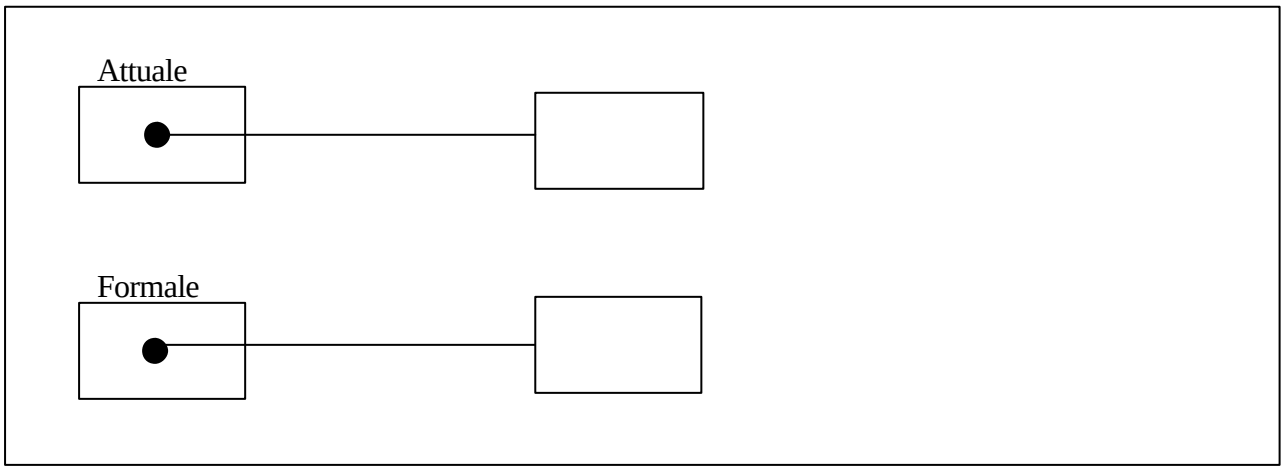
Una volta fatto questo l'accesso alla variabile avviene così:

supponiamo di voler fare l'assegnamento x = 0;

set D[D[0] + off], 0

CALL BY COPY

Non c'è condivisione, i parametri sono variabili locali



I parametri sono effettivamente locali poiché sono copiati dall'ambiente del chiamante a quello del chiamato.

E' più semplice, ma abbiamo sprechi di memoria in quanto duplichiamo le variabili.

Distinguiamo tre casi:

CALL BY VALUE

Passaggio per Valore: il valore del Parametro Attuale viene copiato nel Parametro Formale.
La copia avviene solo dal chiamante al chiamato.
In pratica, serve per comunicare dei valori al sottoprogramma, valori che vengono memorizzati nei parametri formali e che quindi potranno essere utilizzati dal sottoprogramma

Il chiamante valuta i parametri attuali. Non c'è flusso di informazioni tra chiamante e chiamato

CALL BY RESULT

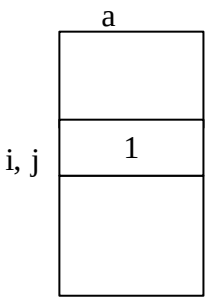
I valori di ritorno dei formali vengono copiati negli attuali. Non c'è flusso di informazioni tra chiamante e chiamato

CALL BY VALUE-RESULT

Entrambi copiati alla chiamata ed al ritorno. C'è flusso di informazioni tra chiamante e chiamato

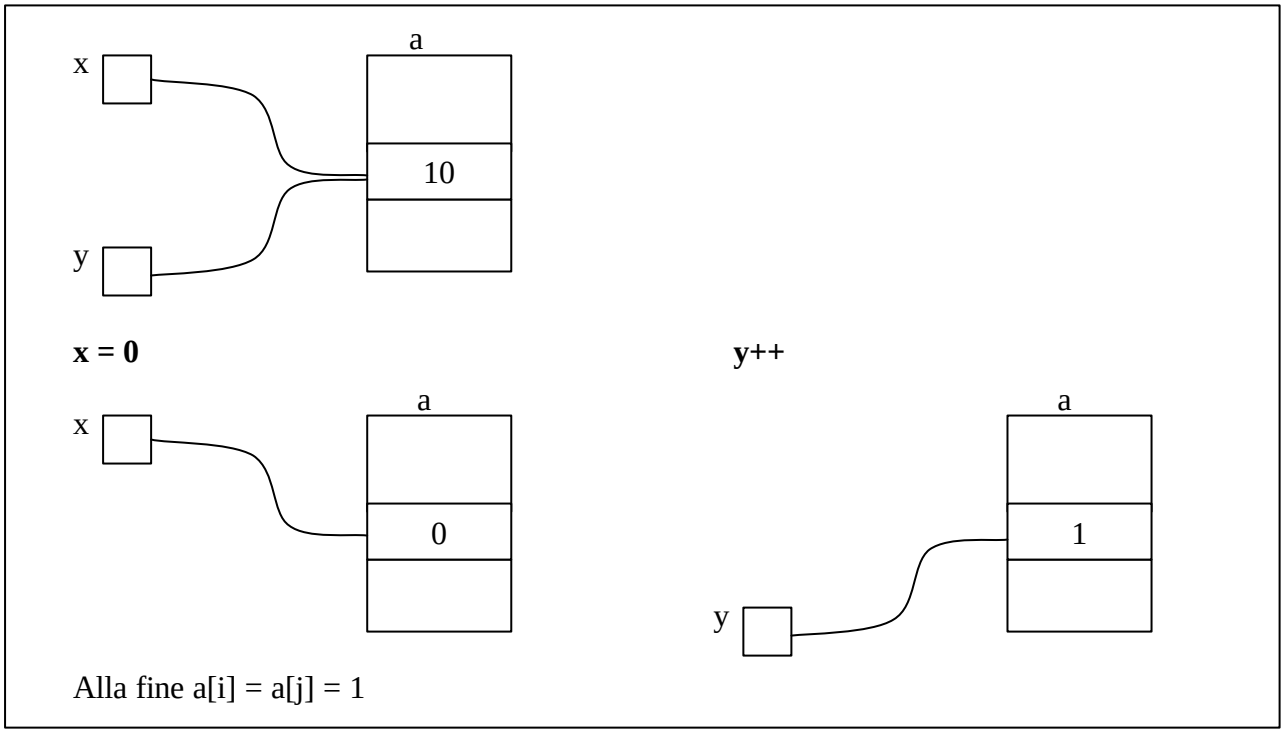
ESEMPIO 1

```
i = j;
a[i] = 10;
foo(a[i], a[j];
…
foo (x,y) {
      …
      x = 0;
      y++;
}
```

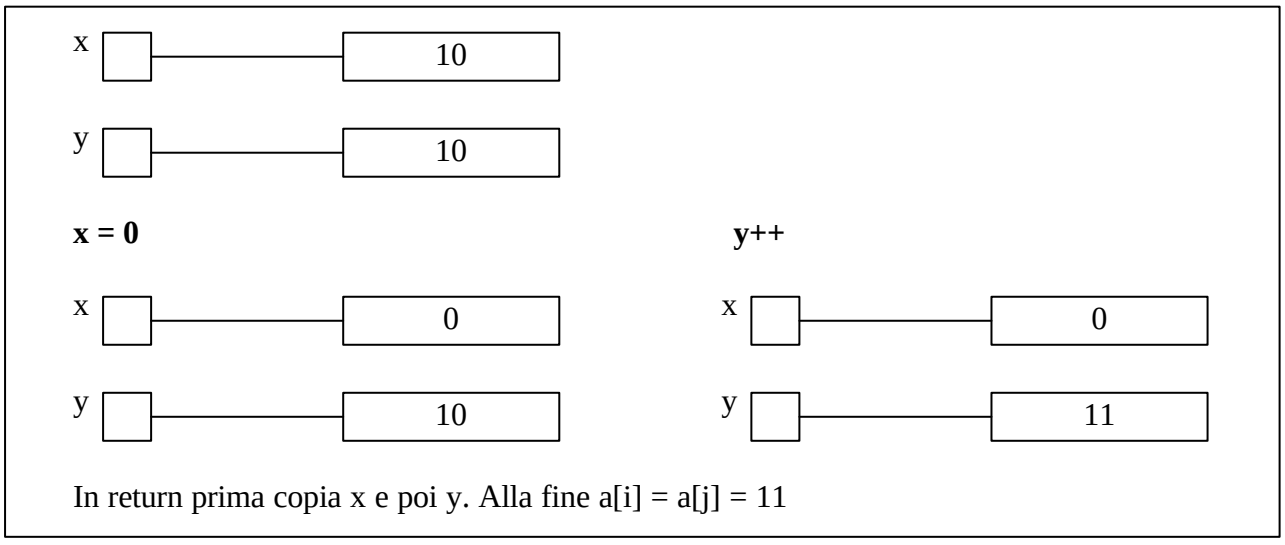


Vediamo cosa succede con i diversi modi:

BY REFERENCE



BY COPY



Quando faccio x = 0, la copia è modificata e non la variabile originale. quindi quando accedo a y ritrovo il valore 10 che avevo prima.

ESEMPIO2

```
i = j;
a [i] = 10;

foo (a[i], a[j]);
 . . .
```

```
a = 10;
 . . . goo (a);

foo (x, y){
      . . .
      x = 0;
      y ++;
}

goo (x){
      . . .
      a = 1;                          //a is nonlocal and visibile
      x = x + a;
}
```

Per riferimento:

a = 10;
x contiene il riferimento di a.

quando eseguo goo()
a = 1;
x = x + a;

ottengo x = 1 +1 = 2

per valore-risultato:

x = 10;

setto il valore di a =1 (ma x resta 10)

x = x + a = 11

alla fine nella variabile non locale ho 11

CALL BY NAME

Definita per sostituzione del nome. Ci sono dei problemi nell'uso

Consideriamo questo esempio:

```
swap (int a, b);
int temp;
{
      temp = a;
      a = b;
      b = temp;
};


swap (i, a[i]);
```

vediamo cosa succede:

```
temp = i;
i = a[i];
a[i] = temp;
```

Sostituisco i nomi.

sia i = 3 ed a[3] = 4

ci aspettiamo di avere alla fine dello swap: i = 4; a[3] = 3

ma vediamo che succede:

```
temp = 3;
i = a[4];
a[4] = 3;
```

Non è quello che ci aspettavamo.

PASSAGGIO DI ROUTINE

Le routine in questo caso sono viste come first class objects.

```
int u, v;

a ( ){
     int y;
     . . .
};

b (routine x){                    // b accetta come parametro una routine.
     int u, v, y;
     c ( ){
          ...
          y =
            . . .;
     . . .
     };
     x ( );
     b (c); . .
     .
};

main ( ) {
     b(a);
};
```

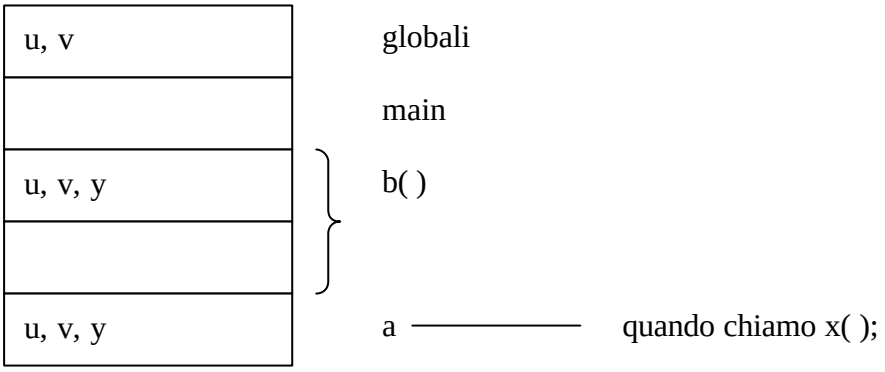
Il main chiama b e gli da come parametro la routine a

Devo passare informazioni di tipo statico, devo dare all'AR in pila informazioni sufficienti per agganciarsi alla catena statica in maniera opportuna.

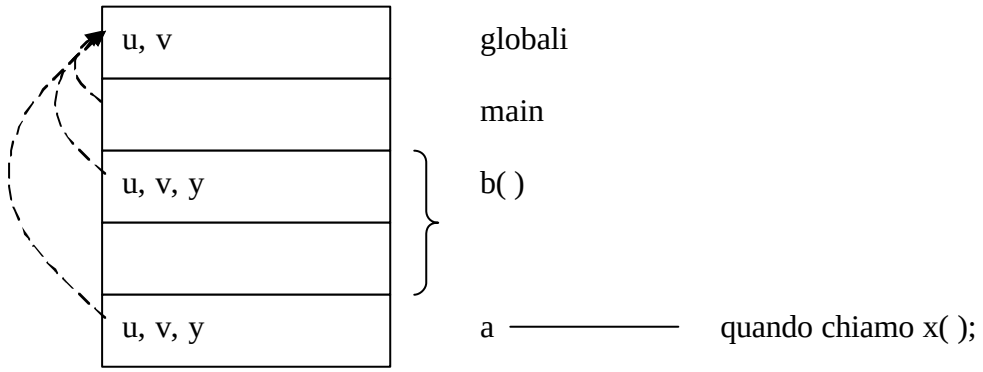
Ho un passo intermedio rispetto a prima.

L'AR che mi interessa (quello della variabile a) viene caricato solo quando la procedura viene invocata.

Quando nel main invoco b(a):



Non posso risolverlo all'atto dell'invocazione perché non so quando *a* viene invocata.
So determinare qual è il link statico.



anche *a* deve puntare alle variabili globali.

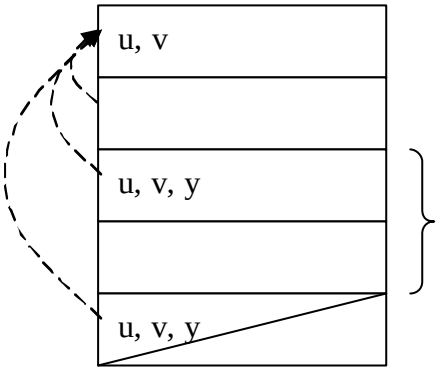
Vediamo come viene fatto:

Ci sono due casi:

Caso a: il parametro attuale passato è nello scope del chiamante (so a che distanza sta) posso passare l'informazione dal chiamante (main) all'AR (b)
➔
Lo static link è il puntatore che è *d* passi all'indietro.

Caso b: il parametro attuale non è visibile dal chiamante ma lo sto usando nella routine corrente
➔
SL è quello che è stato passato dal chiamante.

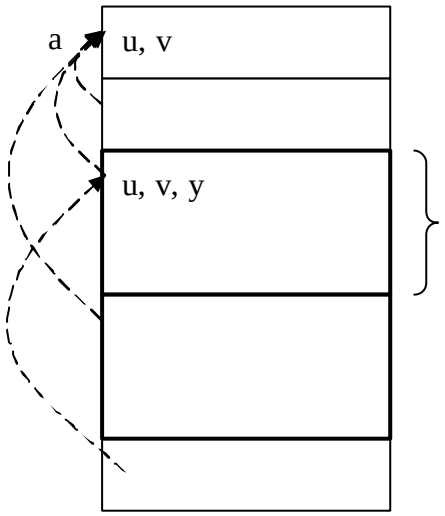
Vediamo cosa succede quando c'è la chiamata di b(c)



globali

main

b( )



globali

main

b( )

b

c ——— **devo andare alla prima invocazione di b**

I TIPI DI DATO

DEFINIZIONE: Un set di valori + un set di operazioni (che possono essere effettuate)

Le entità nei programmi sono INSTANZE dei tipi. Sono le variabili che utilizziamo.
A run-time la variabile assumerà dei valori associati al tipo.

Due domande:

> Cosa sono i valori? Appartengono ad un tipo?
>
> E gli oggetti appartengono ad un tipo?

TIPI PRIMITIVI E COMPOSTI

I tipi PRIMITIVI sono dei tipi atomici (i valori sono generalmente definiti dal linguaggio o dalla macchina)

I tipi COMPOSTI hanno componenti (i valori sono formati da componenti)
Esempi sono le struct, gli array…

I tipi BUILT-IN (incorporati) sono forniti da ciascun linguaggio indipendentemente dal fatto che siano primitivi oppure composti.

I COSTRUTTORI DI TIPI (type constructors) vengono usati per definire nuovi tipi

SCOPO DEI TIPI

Di natura espressiva. Senza i tipi tutti i dati hanno la stessa forma e sta al programma stabilire su cosa sta lavorando.
I tipi definiscono l'insieme dei valori rilevanti per un'applicazione; dichiariamo come useremo le variabili.

Classificare i dati manipolati nel programma (set di valori)

```
c: centigradi;
f: fahrenheit;
```

Proteggere da operazioni illecite (set di operazioni)

```
f := c and f    (non dovrebbe essere permesso; darebbe risultati non predicibili dal
                 punto di vista semantico)
```

VANTAGGI

- Nascondere la rappresentazione sottostante (**Information Hiding**)
  Nascondo la rappresentazione interna, così da non dover accedere ad un'area di memoria ed interpretarla per estrarre l'informazione necessaria.

  - Nome VS Struttura
    Elevo il livello di astrazione, non ho bisogno di conoscere la struttura dei dati ma mi riferisco al nome

  - Specifica VS Implementazione
    Non mi interessa come il tipo di dato è implementato

  - Indipendenza del client (chi usa il tipo) dal server (chi definisce il tipo)
    Non devo sapere com'è fatto il dato per potervi accedere

- Il corretto utilizzo delle variabili può essere controllato in compilazione (se il tipo è conosciuto)
  Cattura solo un insieme di errori, ma non errori dovuti a determinati valori delle variabili che non sono compatibili con l'operazione.
  (Esempio: x = i/j può essere controllata per la correttezza del tipo ma no se $j \neq 0$)

  Forniscono il contesto necessario per la risoluzione degli operatori overloaded in compile-time (se il tipo è conosciuto)
  Il contesto per rimuovere l'ambiguità viene in genere fornito dalla definizione di tipo.

DEFINIRE I "VALORI" DI UN NUOVO TIPO

ENUMERAZIONE
Definisco elementi che appartengono all'insieme dei valori ammissibili per il determinato tipo (c++ fa una rimappazione sugli interi)
Type color = (whit, yellow, red, green)

OPERAZIONI
Sono fornite dal linguaggio; sono quelle di ordinazione degli insiemi e di confronto.
Succ, pred, < (elementi ordinati, per posizione e valore)

AGRREGATI (TIPI COMPOSTI)

- Set di valori, un nome.
  Ogni elemento è a sua volta costituito da un insieme di elementi

- Aggregati omogenei/eterogenei
  omogenei (es. array); eterogenei (es. struct c++)

- Costruttori per valori e tipi composti
  Per inizializzare le singole componenti

- Operazioni su aggregati
  A volte è possibile effettuare operazioni sull'intero aggregato e non solo sulle componenti

- Possibilità di selezione dei componenti
  (ed operare solo su essi)

COSTRUTTORI DI AGGREGATI

PRODOTTO CARTESIANO

- Definito da un insieme di domini A1 X A2 X … An (sono tipi)

- Tuple ordinate (a1, a2, .. an)
  Una singola istanza è una tupla ordinata in cui viene assegnato un valore ammissibile per ogni componente)

- Esempi: numeri complessi; point(x,y)

- Record, struct

Il modo con cui accediamo alle singole componenti è la dot notation.

Esempio

```
record T          //definisco T
     x: int;
     y: float;
end

r: T;             //dichiaro r di tipo T

r.x = 0;          //accedo ad x
```

FINITE MAPPING

L'aggregato è dato da una funzione che associa valori che appartengono ad un certo tipo, al condominio, che appartiene ad un altro tipo.

- map DT→RT

- Array in molti linguaggi
  Dominio: interi (l'indice dell'array);
  Range: il tipo dei valori

Mediante la funzione di mapping ottengo un elemento.

In C++ il DT è fissato a compile time, quando definisco un array devo dire quanti elementi ha. In altri linguaggi posso estendere a run-time DT (il numero di elementi)
In genere l'indice è un intero, in alcuni linguaggi posso fare il mapping su un tipo qualsiasi

UNIONE

E' un costrutto non safe ed alcuni linguaggi come Java non lo supportano.

- A1 U A2 U A3

- Disgiunzione di valori
  Contiene un solo tra gli elementi di A1, A2…

  Esempio C++

  ```
  union address {
        short int offset;            //memorizzo gli indirizzi relativi
        long unsigned absolute;      //indirizzi assoluti
  };
  ```

  Per memorizzare un indirizzo assoluto ho bisogno di più spazio (per l'offset e per l'indirizzo base)

- Qual è il tipo a run-time?
  Non posso saperlo a priori

- Richiede type checking a run-time

Tramite la DISCRIMINATED UNION associo un campo tag (etichetta), che mi dice qual è il tipo attuale della union

Esempio C++

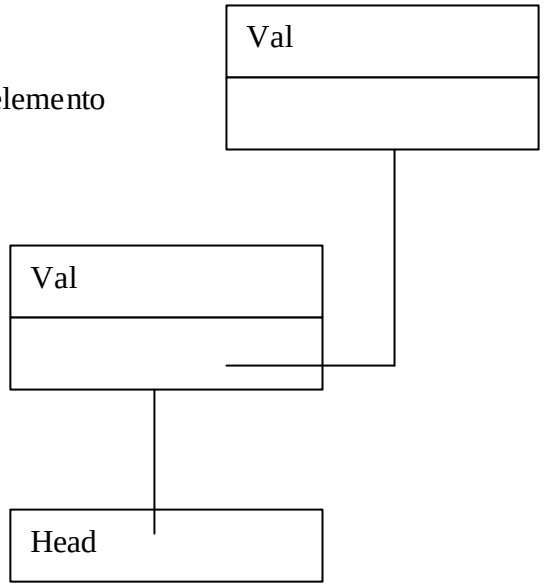
```
union address {
      short int offset;            //memorizzo gli indirizzi relativi
      long unsigned absolute;      //indirizzi assoluti
};

Typedef srtuct  {
      address location;
      descriptor kind;
}safe_address;

descriptor = {relative, absolute};
```

Posso poi fare cose tipo: **`if (descriptor = relative)…`**

STRUTTURE RICORSIVE

La dimensione non è limitata , non è determinata a priori.
Vengono definite mediante puntatori.

```
struct int_list {
      int val;
      int_list* next;  //puntatore al prossimo elemento
}
int_list* head;        //testa della lista
```



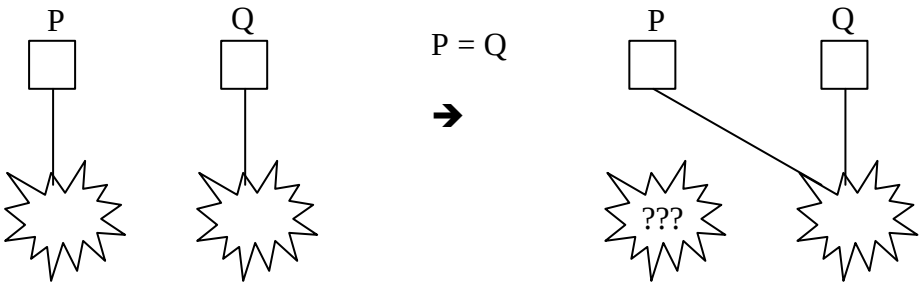
Devo fare allocazione dinamica della memoria.

PROBLEMI CON I PUNTATORI

Sono spesso usati per implementare strutture ricorsive.

Stabiliscono una dipendenza tra due oggetti. E' importante che l'oggetto esista finchè il puntatore punta ad esso.

MEMORY LEAKS (leak = perdita): l'oggetto esiste, ma non può essere raggiunto.



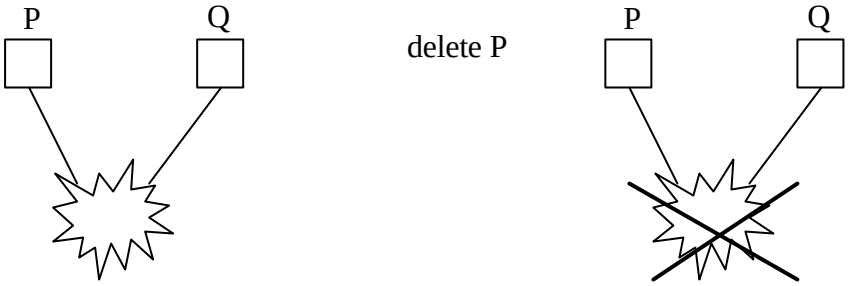
L'oggetto resta allocato in memoria ma non è più raggiungibile. Prima dovrei deallocare la memoria con FREE. E' un problema per linguaggi con la gestione esplicita della memoria

Ci sono linguaggi (come Java) che forniscono nel run-time support un garbage collector che dealloca gli oggetti che non sono più raggiungibili.

Lo svantaggio è che può non essere fatto in modo efficiente.
Linguaggi con il garbage collecor non vengono usati per applicazioni di tipo real time.

DANGLING POINTERS (dangle = dondolare): puntatore ad un'area deallocata

E' il problema opposto.



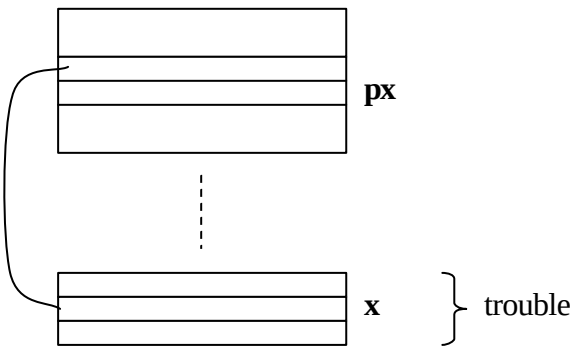
Se P dealloca la memoria, Q non punta più a dati che hanno un senso.

Esempio:

Consideriamo un'area di memoria sullo stack, che viene deallocata

```
int* px;
void trouble (int* t);
{
     int x;
     ...
     px = &x;
     ...
     return;
};
main(){
trouble(p);
     ...
};
```

Ecco cosa succede:



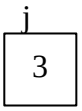
px è nell'area dell'union, x in trouble. Quando l'esecuzione di trouble termina, l'AR è buttato via e con esso si perde il valore di x.

px continua ad esistere ma punta ad una zona non allocata.

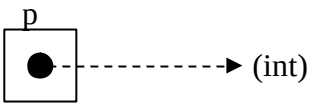
Il problema nasce dal fatto che lo scope del puntatore è più grande dell'oggetto puntato. Ci sono linguaggi che controllano che ciò non sia possibile.

ESEMPI DI PUNTATORI IN C++

```
int j = 3;
```

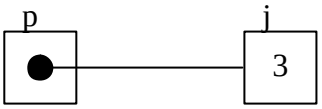



```
int* p;
```

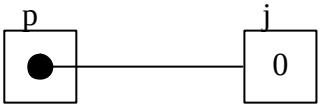



```
int* p = &j;
```

definisco un puntatore passando l'indirizzo della variabile j




```
* p = 0;
```



Puntatore a costante:

```
const int k = 99;
const  *q = &k;
```

Non posso fare *q = 10; poiché il valore è costante
Posso però cambiare il valore del puntatore, q = &k;

Se invece ho:

```
const int *const pc = &j;
```

non posso fare neanche pc = &k; poiché il puntatore è costante.

Questa forma è spesso usata nel passaggio dei paramentri quando non voglio che il loro valore sia modificato.

REFERENZIAZIONE IN C++

Il C++ fornisce una forma di chiamata per indirizzo che è persino più semplice da utilizzare rispetto ai puntatori: il tipo *reference*. Allo stesso modo di una variabile puntatore, il tipo reference fa riferimento alla locazione di memoria di un'altra variabile, ma come una comune variabile, non richiede nessun operatore specifico di deindirizzamento. La sintassi della variabile reference è la seguente:

```
int risultato = 6;
int& ref_risultato = risultato;
```

L'esempio precedente definisce la variabile reference ref_risultato e le assegna la variabile rusultato. Adesso è possibile fare riferimento alla stessa locazione in due modi: tramite risultato e tramite rif_risultato. **Poiché entrambe le variabili puntano alla stessa locazione di memoria, esse rappresentano dunque la stessa variabile. E' questo il motivo per cui una variabile reference viene anche chiamata variabile alias.** Quindi, ogni assegnamento fatto su ref_risultato si rifletterà anche su risultato e viceversa.

Attenzione. E' sempre necessario inizializzare una variabile di tipo reference. Ad esempio non si sarebbe potuto scrivere semplicemente:

```
int& ref_risultato;     // Errore!
```

Il tipo reference, infatti, ha una restrizione che lo distingue dalle variabili puntatore: bisogna sempre definire il valore del tipo refererence al momento della dichiarazione e tale associazione non può essere più modificata durante tutta l'esecuzione del programma.

Si tenga inoltre presente che non è possibile assegnare ad una variabile reference il valore NULL, utilizzato invece per i puntatori

Il vantaggio, facilmente visibile, del tipo reference rispetto ai puntatori è rappresentato dal fatto che *una variabile reference, dopo la sua definizione, va trattata esattamente allo stesso modo di una variabile normale e non necessita degli operatori di indirizzamento e deindirizzamento utilizzati dai puntatori.*

- Una variabile dichiarata per essere “T&” (referenza del tipo T) deve essere inizializzata da un oggetto di tipo T (o convertibile ad un tipo T)

  ```
  int i;

  int& r = i;     //i ed r condividono lo stesso oggetto (l_value uguale)

  r = 1;          //cambia anche il valore di i ad 1 (modifico l’r_value associato a i)

  int& rr= r;     //rr, r ed i condividono lo stesso oggetto
  ```

- Una referenza non può essere cambiata dopo l’inizializzazione

PASSAGGIO DEI PARAMETRI IN C++

Si utilizzano le reference

Al chiamato passo un alias dell’oggetto originale che punta alla stessa area del parametro attuale.

```
void f(int val, int& ref){
     val++;
     ref++;
}
```

- val è per valore (solo la copia locale viene incrementata)
- ref è per riferimento (il parametro attuale viene modificato)
- un argomento di tipo T[] è convertito a T* quando è passato (gli array non possono essere passati per copia per ragioni di efficienza)

TIPI DEFINITI DALL'UTENTE

Definisco la struttura di un tipo e gli do un nome.

```
struct complex {
     float real_part, imaginary_part;
}
…
complex a, b, y;        //dichiarazione = a quella dei tipi normali
```

- Variabili possono essere dichiarate ed istanziate
- Non c'è protezione in questo modo, gli attributi sono public
  Vorrei nascondere la struttura interna dei dati per proteggermi da operazioni che non hanno senso per quel tipo.
  Il modo fornito dai linguaggi di programmazione è la nozione di tipo di dato astratto.

TIPI DI DATO ASTRATTO

Un tipo astratto di dato è un oggetto matematico costituito da due componenti:

- Una collezione di domini, uno dei quali, chiamato di interesse riveste particolare importanza, perché è il dominio del tipo, cioè rappresenta tutti i valori del tipo.

- Un insieme di funzioni, ciascuna delle quali ha come dominio di definizione il prodotto cartesiano di n domini che appartengono alla collezione di cui al punto 1, e come codominio un dominio che appartenga alla collezione stessa.

La nozione di Tipo Astratto consente di concettualizzare:

- I tipi di dato utilizzati nella progettazione di algoritmi e nella realizzazione dei corrispondenti programmi.

- Classi di oggetti qualsiasi, che si ritengono importanti in una applicazione.

E' evidente che la possibilità di usare non solo tipi di dato predefiniti, ma anche tipi astratti che noi come progettisti ideiamo e definiamo evidenzia maggiormente l'importanza di indagare su come realizzare un tipo astratto nella fase di concettualizzazione.

```
Class point {
      float x,y;
public:
      point (float a, floa b) {x = a; y = b;}
      void x_move (float a) {x += a;}            //x = x + a
      void y_move (float b) {y += b;}
      void reset( ) {x = y = 0;}
};
```

Vediamo alcuni esempi

```
point p1 (1.3, 3.7);

point p2 (53.3, 0.0);

point * p3 = new point (0.0, 0.0);
```

La primitiva **new** alloca gli oggetti nell'heap

```
p1.x_move (9.3);
```

Per manipolare gli oggetti
x_move deve essere un meodo pubblico
Invoco il metodo x_move sull'oggetto p1
In genere il metodo è invocato sul "current object", nell'esempio è p1

```
p3 = p2;
```

invocazione metodo

```
point p4 = point (1.3, 3.7);
```

copy constructor: l'oggetto è costruito ed inizializzato per copia di un altro oggetto.
equivale a:
point p4 (point (1.3, 3.7));
point (1.3, 3.7) è l'oggetto di cui devo fare la copia

FIRST CLASS TYPES (tipi di prima classe)

Una **classe** è un costrutto che permette di definire nuovi tipi
Un **tipo** è **di prima classe** se vi posso fare tutte le operazioni definite sui tipi

Point è un tipo di prima classe?
Posso passarlo come argomento?
Posso scrivere una funzione che lo ritorna come risultato?

Esempio:

```
point copy (point p) {
        point temp;
        temp = p;
        return temp;
};
```

CLASSI IN C++

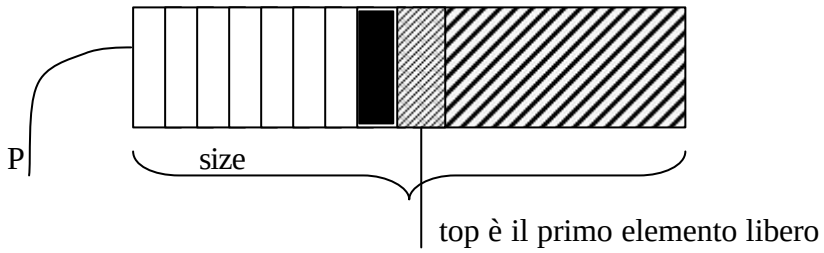
In c++ si possono scrivere e compilare separatamente DEFINIZIONE ed IMPLEMENTAZIONE

DEFINIZIONE: tutto ciò che fa parte della classe tranne l'implementazione. Non è l'interfaccia

Esempio: STACK

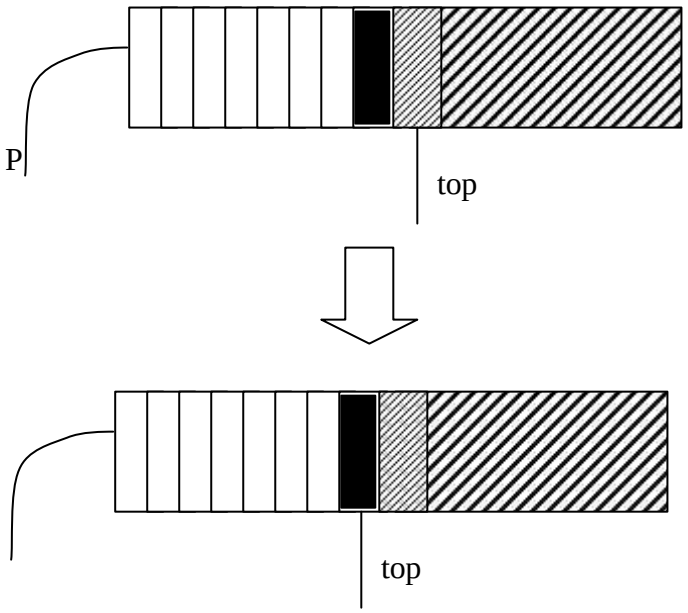
```
class stack {
      int* p;                  //puntatore alla pila
      int* top;                //elemento in cima
      int size;                //dimensioni della pila
public:
      stack (int n) {top = p = new int [size = n];}
      ~stack ( ) {delete [ ] p;}
      void push (int  i) {top++=i;}    //memorizza l'elemento e poi  incrementa top
      int pop( ) {return *--top;}      //prima decremento il valore di  top e poi
                                         restituisco il valore
      ...
}
```

Definisco uno stack (LIFO) di interi.
Lo definisco mediante un array



push → inserisco un elemento nello stack

pop → tolgo un elemento dallo stack



pop decrementa di 1 top e ritorna il valore che si trova nella posizione -1

~stack → distruttore dello stack

COSTRUTTORE

```
stack (int n) {top = p = new int [size = n];}
```

Alloca un array di dimensione n. top e p puntano alla prima cella dell'array

DISTRUTTORE

Se non ci fosse, dall'heap cancellerei solo p top e size, ma resterebbe l'array che non sarebbe più raggiungibile.

TEMPO DI VITA DEGLI OGGETTI

ALLOCAZIONE
L'oggetto è allocato, quindi gli attribuo sono inizializzati come spacificato dai costruttori.
L'allocazione può essere automatica o esplicaita (con il new)

DISTRUZIONE
Gli attributi vengono puliti e poi l'oggetto deallocato. Può essere automatica oppure esplicita (con il delete)

COSTRUTTORE

Ha lo stesso nome del tipo definito.

Ce ne possono essere diversi con SIGNATURE diverse, permettono infatto l'overload in base ai parametri passati.

Ne viene creato uo di default se non se ne definiscono

COPY CONSTRUCTOR

E' un tipo speciale di costruttore che differisce da un'assegnazione di valore

Differisce per l'assegnazione (l'oggetto non esiste ancora)

Costruisce un oggetto da uno esistente

Quello di default fa una "shallow copy", una copia superficiale
Nell'esempio dello stack verrebbero copiati solo p top e size e non il contenuto dell'array

NOTAZIONE

```
point p1 = p2;          oppure
point p1 (p2);
```

DISTRUTTORE

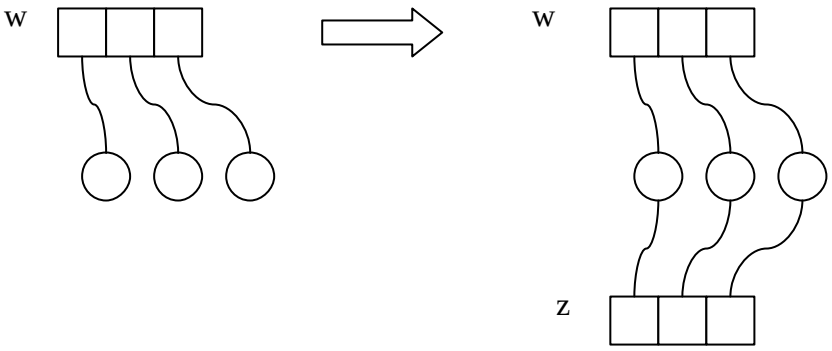
Nome della classe preceduto da ~
Effettua un'azione di pulizia dopo l'ultimo utilizzo di un oggetto
Ci può essere soltanto un distruttore per una classe.
Se non si dichiara, ne esiste uno di default

ASSEGNAZIONE PER LA CLASSI

Il C++ usa una copia membro a membro degli attributi. Di default la copia è "shallow"

Supponiamo di avere una classe X e due oggetti z e w di X.
Possiamo fare cose tipo: z=w !!!

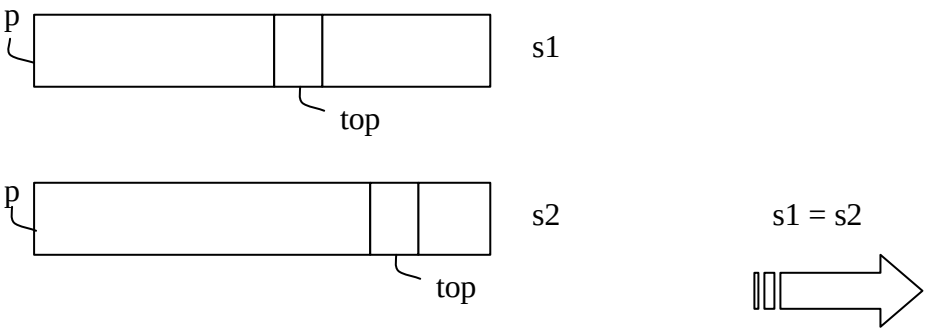
La copia di default è shallow

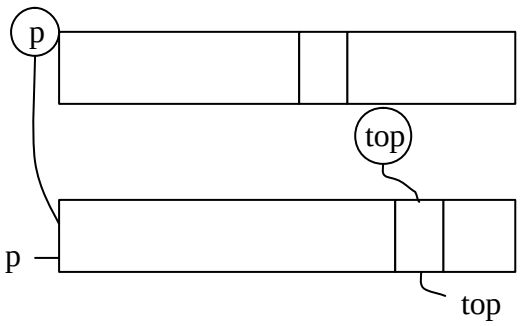


Se voglio una copia più profonda devo ridefinire il metodo.

COPIA MEMBRO A MEMBRO

```
void f( )
{
      stack s1(20), s2(30);
      s1 = s2;                  //l'array referenziato da s1 si perde, non è però cancellato
};
```





Non ho una cpoia ma p e top puntano da un'altra parte e perdo il contenuto dell'array s1.

Devo ridefinire il metodo =

```
class stack {
      int* p;
      int* top;
      int size;
public:
      stack (int n) {top = p = new int [size = n];}
      ~stack ( ) {delete [ ] p;}
      void push (int  i) {top++=i;}

      ?    stack & operator = (const stack &);

      int pop( ) {return *--top;}
      ...
}
```

```
stack & operator = (const stack &);
```

riceve per indirizzo uno stack e torna per riferimento uno stack

```
stack & stack :: operator = (const stack & s)
{
      if (this != &s) {      // per coprire il caso s = s
            delete [] p;
            p = s.p;
            top = s.top;
            size = s.size;
      }
      return * this;
}
```

```
stack :: operator =
```

Quello che definisco sotto è l'implementazione di = per la classe stack

this ? qual è l'oggetto corrente
se l'oggetto corrente è diverso dall'oggetto riferito da s, faccio l'assegnazione.

si cancella l'oggetto p (cancello l'array puntato da p) con **`delete [] p;`**

```
p = s.p;
top = s.top;
size = s.size;
```

costruisco p. non è una deep copy, ma almeno ho eliminato l'array puntato da p che la shallow copy non consentirebbe

DEEP COPY (clonazione o copia profonda)

```
stack & stack :: operator = (const stack & s)
{
      if (this != &s) {      // per coprire il caso s = s
            delete [] p;
            p = new int[size = s.size];
            top = s.top;
            for (int i=0; i<(top-p); ++i) { p[i] = s.p[i]; } //ricopio i valori
      }
      return * this;
}
```

size = s.size
viene calocolato s.size e poi si assegna il valore a size

COPY CONSTRUCOTOR per uno stack

```
stack :: stack (const stack & s)
{
      p = new int[size = s.size];
      top = p + s.top – s.p;
      for (int i=0; i<(top-p); ++i) { p[i] = s.p[i]; }
}
```

fa una copia profonda dell'array nello stack

FUNZIONI FRIEND IN C++

E' una funzione che può acedere ai segreti di una o più classi.
Supponiamo di avere due classi: matrix e vector; voglio eseguire il prodotto tra una matrice e un vettore.
L'operazione appartiene alla classe matrix oppure alla classe vector?
Se l'operazione appartiene alla classe matrice, posso accedere solo alla classe matrix
Se l'operazione appartiene alla classe vector, posso accedere solo alla classe vector

La funzione è definita fuori dalle classi, ma può accedere ai segreti delle classi.

```
class matrix {
      …
      friend vector mult (const matrix &, const vector &);
      // può stare sia nella parte public che in quella privata
}

class matrix {
      …
      friend vector mult (const matrix &, const vector &);
      // può stare sia nella parte public che in quella privata
}
```

```
vector mult (const matrix & m, const vector & v);
{  //assume 0…3 matrix and vector
      vector r;
      for (int i = ; i < 3; i++) {
            r.elem(i) = 0;
for  (int j = 0; j < 3; j++) {
      r.elem(i) += m.elem(i,j)*v.elem(j);
      }
      return r;
}
```

TIPI DI DATO ASTRATTO GENERICI

Il costrutto è TEMPLATE
La classe è parametrica rispetto ad una classe T

```
template <class T> class stack {
public:
      stack(int sz) {top = s = new T[size = sz];}
      ~stack() {delete []s;}
      void push(T el) {*top ++= el;}
      T pop( ) {return *--top}
int lenght( ) {return top – s;}
private:
      int size;
      T *top;
      T* s;
};
```

USO DEGLI ADT GENERICI

```
void foo( ) {
      stack <int> int_st(30);      //posso istanziare interi
      stack <item> item_st(100); //o classi come item, che è stata definita in precedenza
      stack <stack
      ...
      int_st.push(9);
      ...
}
```

FUNZIONI IN C++

Il C++ non ha solo classi ma anche funzioni generiche (mentre in Java non ci sono)

Una generica funzione in C++ ha la forma seguente:

```
tmeplate <class T>
void swap(T&a, T&b)
{
     T temp = a;
     a = b;
     b = temp;
}
int i, j;
char x, y;
pair<int, string> p1, p2;
...
swap(i, j);            //scambia interi
swap(x, y);            //scabmia caratteri
swap(p1, p2);          //scambia pairs
```

TYPE SYSTEMS

E' un set di regole usato dal linguaggio per strutturare ed organizzare i propri tipi

Gli oggetti sono istanze dei tipi

L'uso scorretto dei tipi da un errore di tipo

un programma è "type safe" se non ha errori di tipo

TYPE CHECKING

STATICO: verifia prima dell'esecuzione
DINAMICO: verifica in run-time

TYPE SYSTEM STRONG
Se garantisce la "type safety" (non esistono errori di tipo nel programma)

TYPE SYSTEM STATICO
Permette di sapere per ogni esecuzione qual è il suo tipo

STATICO➔STRONG

Posso verificare che non ci siano errori di tipo prima dell'esecuzione

TYPE SYSTEM DINAMICO
Il tipo può variare durante l'esecuzione

Esempio: linguaggio in cui è definita l'unione T = Q∪W. T non so a priori di che tipo sia. (il linguaggio si dice POLIMORFO)

COMPATIBILITA'

T1 è compatibile con T2 rispetto ad un operazione O, se posso usare oggetti di un tipo o dell'altro nel contesto di una certa operazione

Ho un'operazione che può ricevere oggetti di diverso tipo (l'operazione invece è la stessa)
E' diverso dall'overloading in cui l'operazione è diversa in base al tipo.

REGOLE DI COMPATIBILITA'

COMPATIBILITA' PER NOME: possono apparire solo oggetti di un tipo che mi aspetto e che garantisce la compatibilità
COMPATIBILITA' PER STRUTTURA: tipi con nomi diversi ma con la stessa struttura

Esempio: numeri complessi e coordiante sono compatibili per struttura ma non per nome.

CONVERSIONI DI TIPI

Si trasformano oggetti di un tipo in oggetti di un altro tipo.

Esempio:

fun: T1 → R1
x: T2; y: R2;

è possibile chiamare y = fun (x) ?

O si trasforma il tipo degli oggetti (madianta FUNZIONI DI CASTING)

t21: T2 → T1
r12: R1 → R2
y = r12(fun(t21(x2))

Oppure è fatta automaticamente dal compilatore

Esempio:

```
x: int;
y: float;
x = y;          //trasforma y in un intero
```

ESERCIZI SIMPLESEM

ESERCIZIO

1. Sia dato il seguente frammento di codice:

```
int a, b, c;
p1() {
     int a, d;
     p2() {
     int b, e;
          p3() {
          int x, y;
          ...
          while(...) { int x; ... ; x++; }
          c = a + b + c + d + e;                    (**)
          while(...) { int y; ... ; y++; }
          }
          p3();
     }
     q();
     p2();
}
q() {
...
a = a + b + c;                                      (*)
}
```

Si assuma che venga invocata p1(). Rispondere alleseguenti domande:

a) Mostrare lo stato della memoria dati dell'interprete SIMPLESEM nel punto indicato da (*) e poi in quello indicato da (**);

b) Dare distanza e o set di tutte le variabili coinvolte nell'assegnamento (**).

c) Si guardi ora al frammento di codice assumendo regole di scoping dinamico.
Cambia qualcosa nell'esecuzione dell'istruzione (*)? Perché?

SOLUZIONE

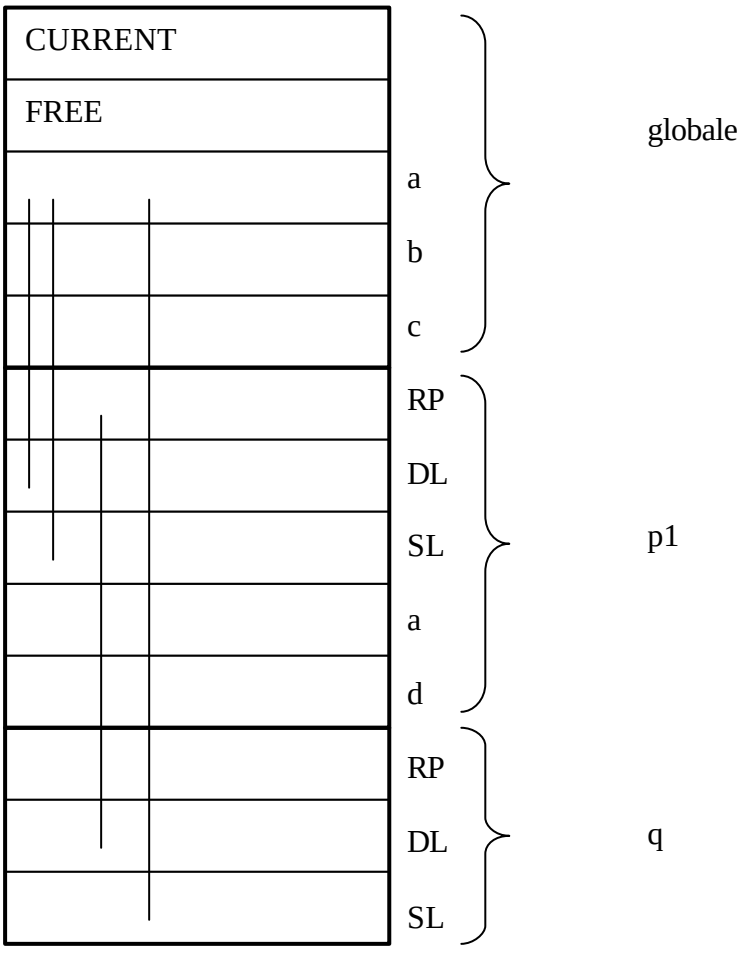
a) Mostrare lo stato della memoria dati dell'interprete SIMPLESEM nel punto indicato da (*) e poi in quello indicato da (**);

p1 (il main) e q sono allo stesso livello
in p3 ci sono due blocchi con variabili locali

(*)

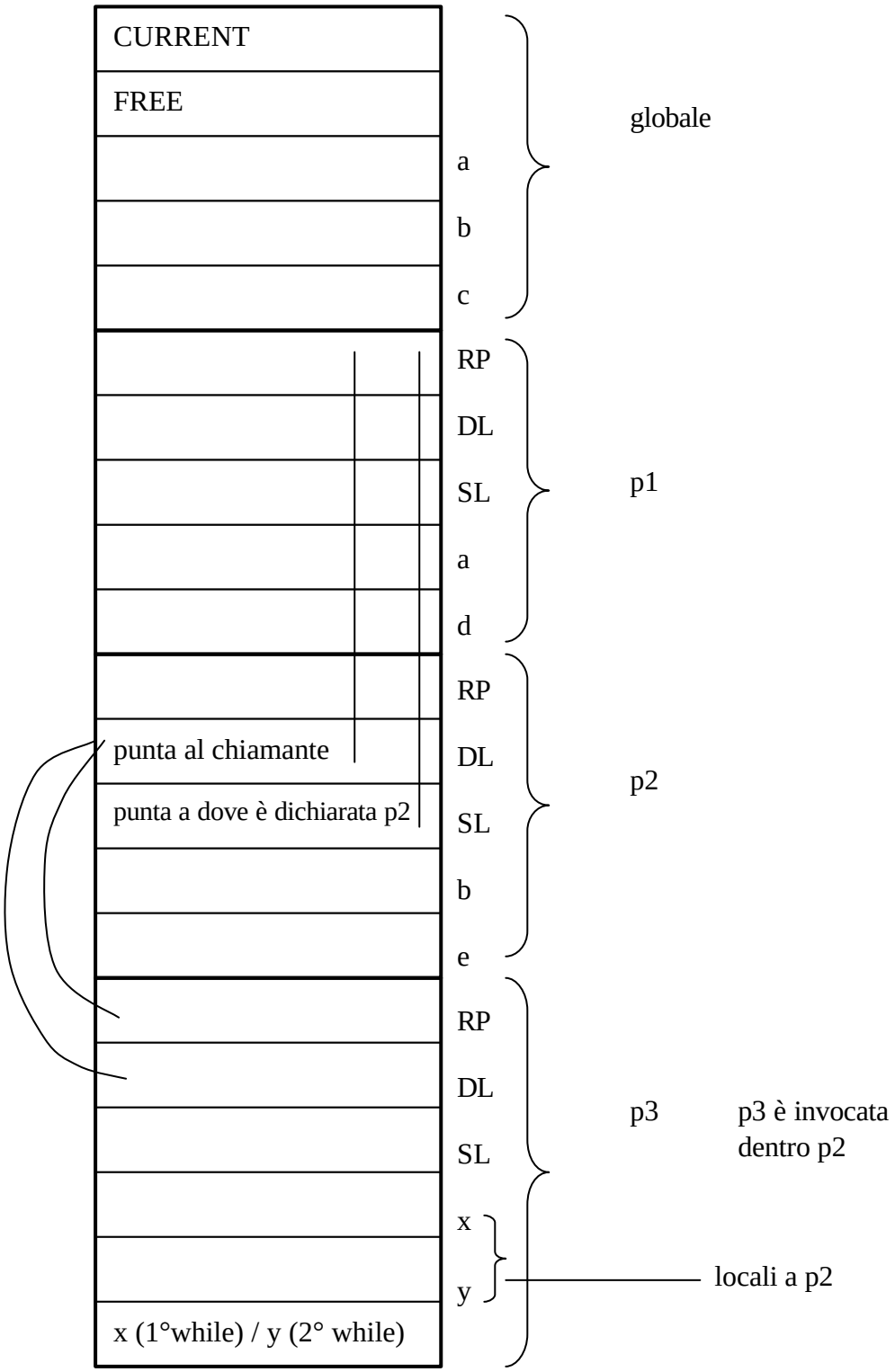
Ho come prima cosa l'invocazione a q

```
q() {
...
a = a + b + c;
}
```



DL → PUNTA AL CHIAMANTE
SL → PUNTA A DOVE È DICHIARATA P2

Esecuzione di q terminata.



I BLOCCHI WHILE DICHIARANO VARIABILI LOCALI ➔ OVERLAY

b) Dare distanza e o set di tutte le variabili coinvolte nell'assegnamento (**).

c = a + b + c + d + e;

Distanza (distanza tra il punto in cui si fa riferimento e in cui è dichiarata):

c: distanza 3 (globale)

a: devo prendere quella più vicina -> distanza 2

b: distanza 1 (dichiarata in p1)

d: uguale a quella di a -> distanza 2

e: uguale a quella di b -> distanza 1

Offset:

Dipende da dove ho messo le variabili

c:	offset 2	(a=0; b=1; c=2)	SIAMO NEL MAIN!
b globale:	offset 1		
b locale:	offset 3	(quella in p1)	DEVO CONSIDERARE RP, SL, DL!

c) Si guardi ora al frammento di codice assumendo regole di scoping dinamico.
Cambia qualcosa nell'esecuzione dell'istruzione (*)? Perché?

Ad esempio in (*) con lo scoping statico *a* è quella dichiarata globalmente, con lo scoping dinamico quella usata più di recente.

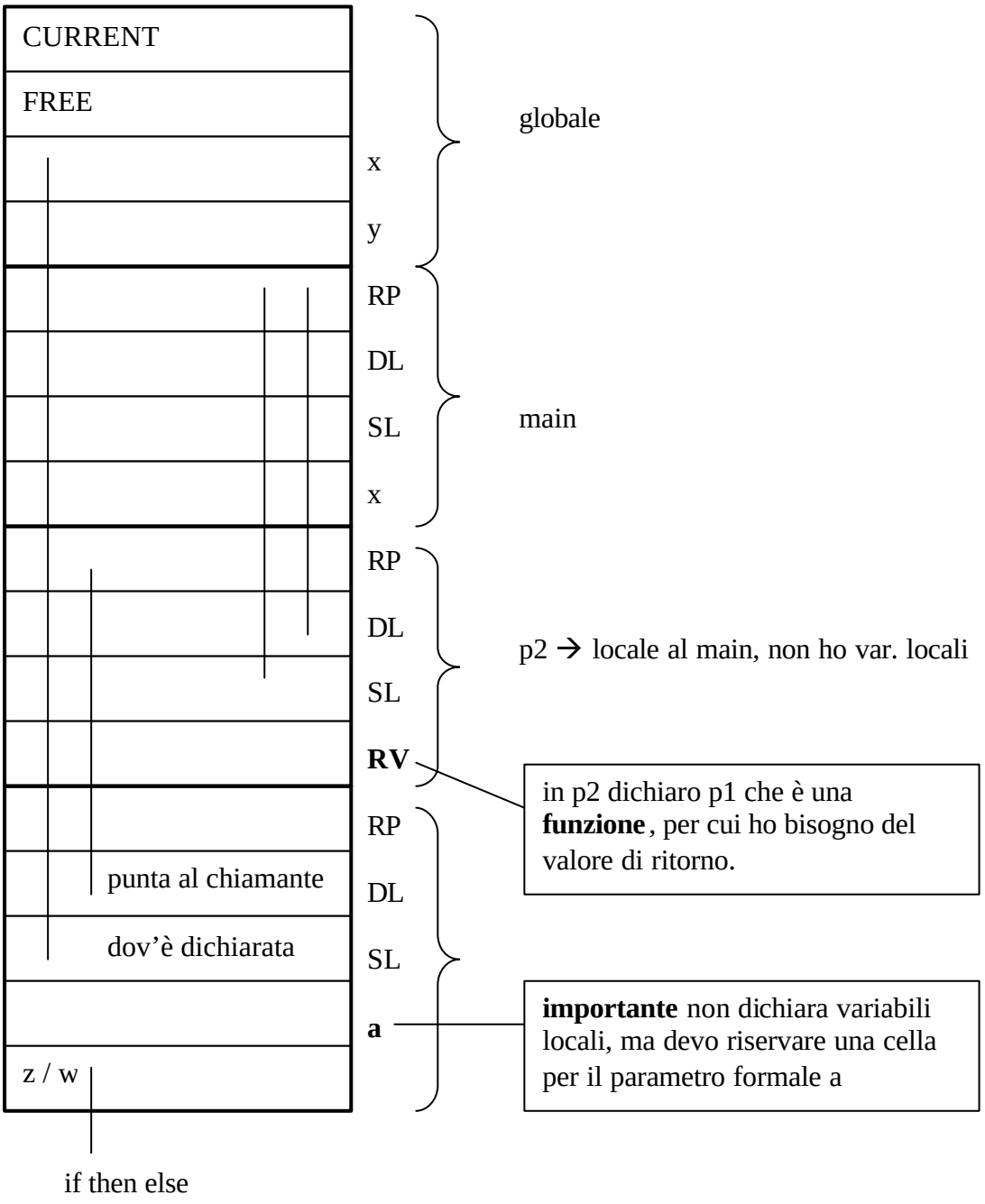
ESERCIZIO

Si consideri il seguente frammento in un ipotetico linguaggio di programmazione:

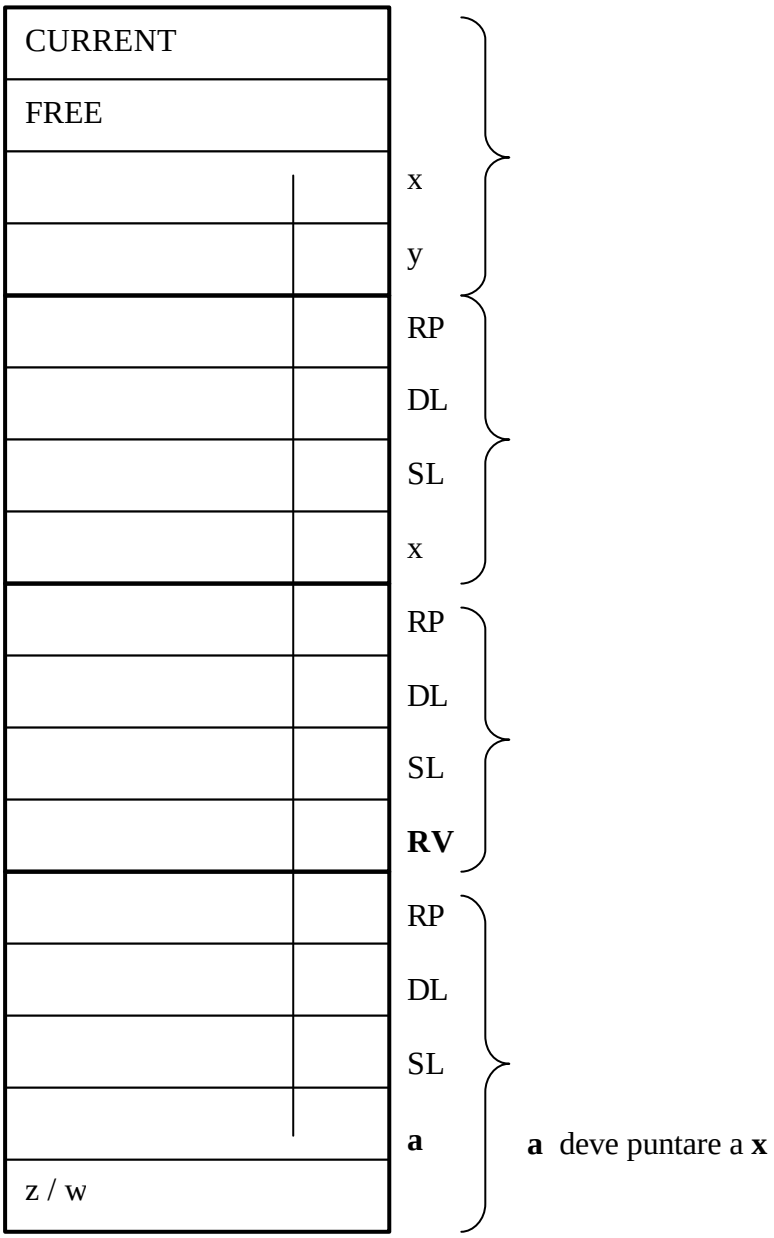
```
int x = 0;

int p1(int a) {
     if (a == 1) {
          int z = 4;
          return z*x;
     } else {
          int w = 3;            (*)
          return x+w;
     }
     p3();
}

int y = 1;

main() {
     int x = 4;
     p2() {
          x++;
          x = p1(x);
     }
     p2();
}

p3() {
     if (y != 0) {
          p1(y);
          y = 0;
     }
}
```

Mostrare lo stato dell'interprete SIMPLESEM immediatamente dopo l'esecuzione dell'istruzione (*). In particolare, mostrare i valori delle variabili e i valori delle informazioni di controllo (es. catene statiche e dinamiche). Si assuma passaggio di parametri per valore.

SOLUZIONE


CURRENT
FREE
x
y
RP
DL
SL
x
RP
DL
SL
RV
RP
DL
SL
a
punta al chiamante
dov'è dichiarata
z / w
if then else
globale
main
p2 → locale al main, non ho var. locali
in p2 dichiaro p1 che è una **funzione**, per cui ho bisogno del valore di ritorno.
importante non dichiara variabili locali, ma devo riservare una cella per il parametro formale a

Nota: passaggio per riferimento:



NOTA:

Nel MAIN mettere tre celle:

RP→torna al sistema operativo
DL→eventuali chiamate ricorsive.

ESERCIZIO

Lo sviluppo di applicazioni distribuite per Internet è caratterizzato dalla necessità di ottenere prodotti competitivi in tempi rapidi, spesso con requisiti estremamente mutevoli (in quanto le necessità del mercato cambiano rapidamente) e con tecnologie che vengono modificate e migliorate continuamente, con *release* che si susseguono a distanza di pochi mesi.

Si commentino pregi e difetti del tradizionale modello a cascata in questo contesto.

SOLUZIONE

Il modello a cascata tradizionale (senza ricicli) non `e adatto a questo tipo di scenario, in quanto troppo rigido. Ben difficilmente le fasi dello sviluppo verranno svolte come previsto, a causa del dinamismo del dominio applicativo considerato.

ESERCIZIO

1. Si consideri il seguente frammento di codice:

```
int x = 1;
int y = 2;
int z = 3;
main() {
      int x = 3;
      int z = 4;

      int p1(int t) {
            t++;
            return t + x*y + p2(z);
      }

      x = x + p1(y);
      print(x);
}

int p2(int u) {
      int p3(int v) { return ++v; }

      int w = u + z;
      return p3(w);
}
```

Rispondere alle seguenti domande:

(a) Assumendo passaggio parametri per valore, mostrare il valore ritornato dall'istruzione di stampa a video print(x) nel caso in cui il linguaggio adotti regole di scoping, rispettivamente:

i. statiche

ii. dinamiche.

(b) Assumendo regole di scoping statiche, mostrare il valore ritornato dall'istruzione di stampa a video print(x) nel caso in cui il passaggio parametri sia invece per riferimento.

SOLUZIONE

(a)

i. 20
ii. 21

(b)

23

ESERCIZIO

Sia dato il seguente frammento di codice:

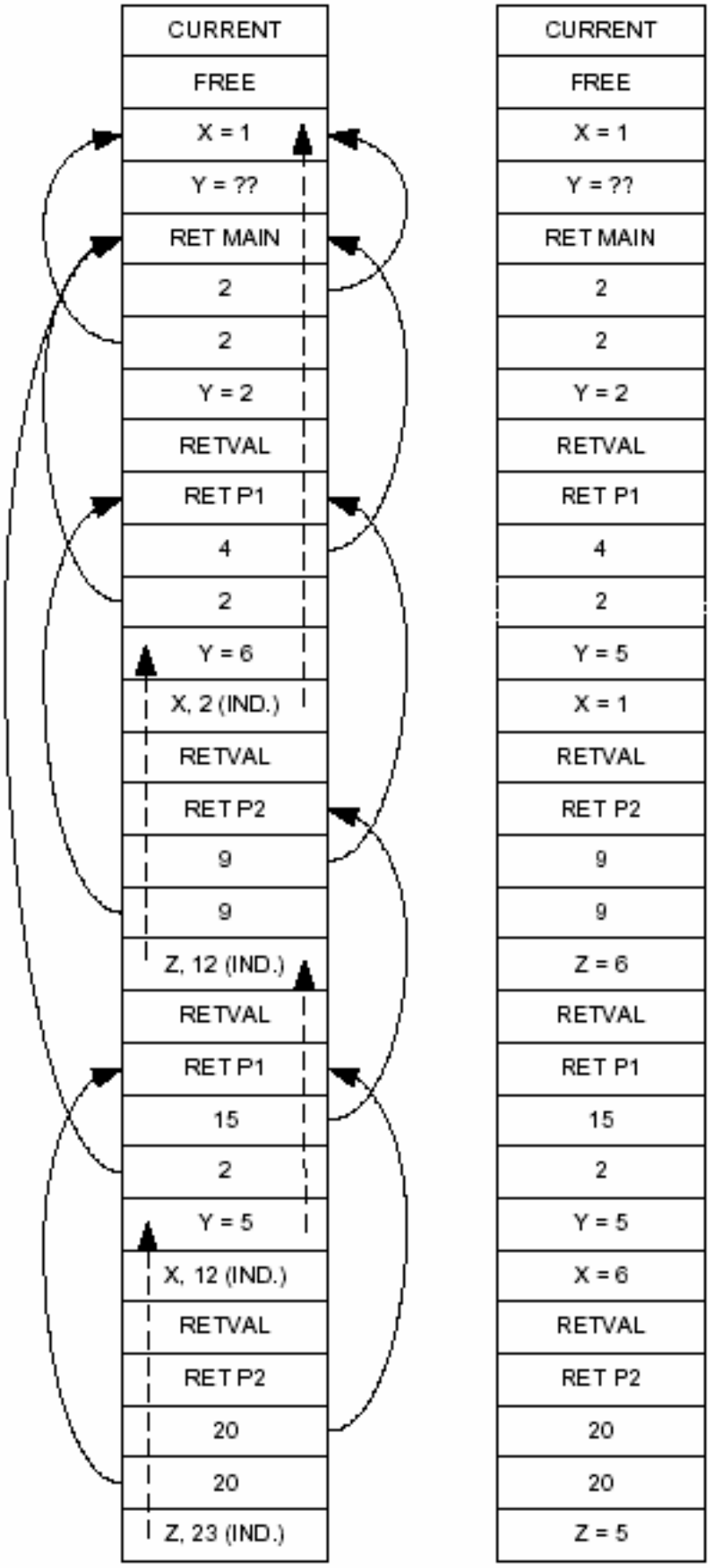
```
int x = 1;
int y;
main() {
     int y = 2;
     ...
     y = p1(x);
     ...
}
int p1(int x) {
     int y = 5;
     int p2(int z) {
          z++;
          z = p1(z);
     }
     x = p2(y); (*)
}
```

(a) Mostrare lo stato della memoria dati dell'interprete SIMPLESEM immediatamente dopo l'invocazione nel punto (*), assumendo che la procedura p2 sia gi`a stata invocata una volta precedentemente per via della ricorsione. Si mostrino i valori delle celle che contengono informazioni di controllo e i valori delle variabili.

Nota bene: Si assuma una semantica di passaggio parametri per riferimento, e si tenga conto di ciò nella rappresentazione della memoria dati.

(b) Si mostri lo stato della memoria dati nel caso in cui invece il passaggio parametri sia per valore anziché per riferimento.

SOLUZIONE


CURRENT
FREE
X = 1
Y = ??
RET MAIN
2
2
Y = 2
RETVAL
RET P1
4
2
Y = 6
X, 2 (IND.)
RETVAL
RET P2
9
9
Z, 12 (IND.)
RETVAL
RET P1
15
2
Y = 5
X, 12 (IND.)
RETVAL
RET P2
20
20
Z, 23 (IND.)
CURRENT
FREE
X = 1
Y = ??
RET MAIN
2
2
Y = 2
RETVAL
RET P1
4
2
Y = 5
X = 1
RETVAL
RET P2
9
9
Z = 6
RETVAL
RET P1
15
2
Y = 5
X = 6
RETVAL
RET P2
20
20
Z = 5

ESERCIZIO

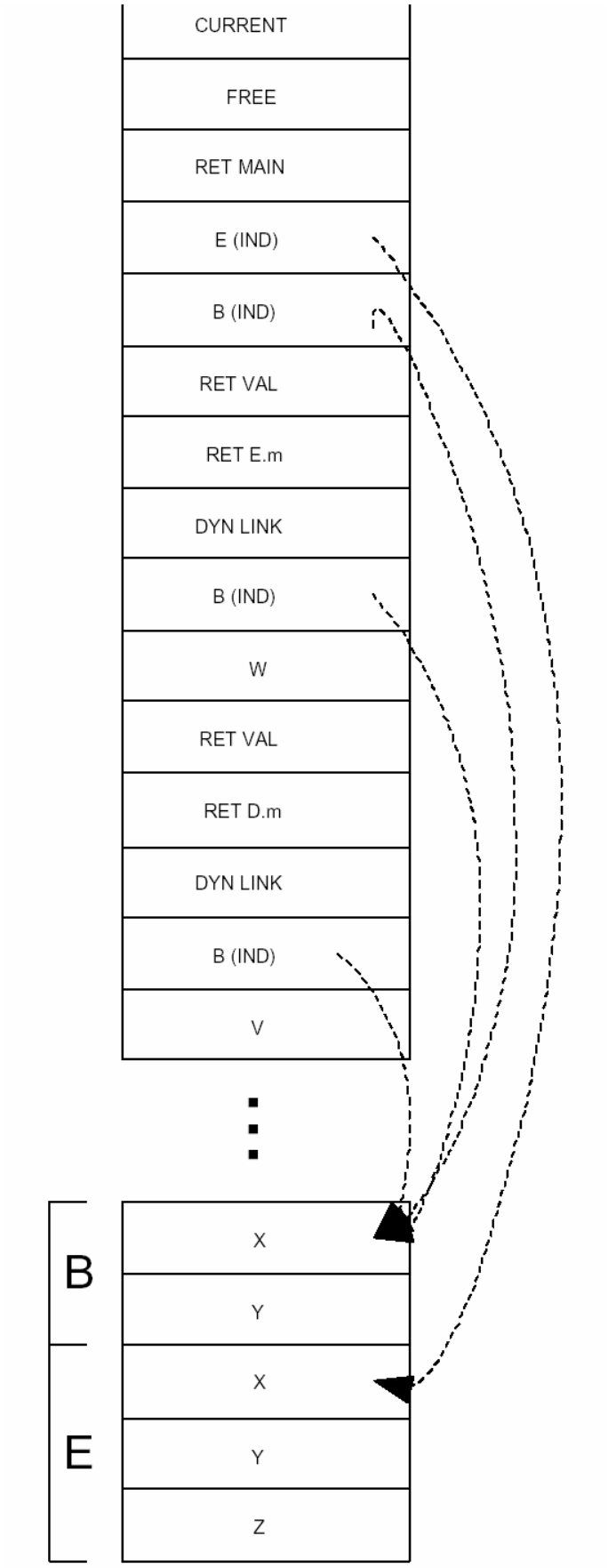
Sia dato il seguente frammento di codice Java:

```
class B {
  int x;
  int y = 1;
}

abstract class C {
  protected int x;
  C(int x) { this.x = x; }
  abstract int m(B b);
}

class D extends C {
  private int y = 1;
  D(int x) { super(x); }
  int m(B b) {
    b.x++;
    int v = x + y;
    return v;    //  (*)
  }
}

public class E extends D {
  private int z = 2;
  E(int x) { super(x); }
  public int m(B b) {
    int w = super.m(b);
    b.y--;
    return w + z;
  }

  public static void main(String[] args) {
    E e = new E(5);
    B b = new B();
    System.out.println(e.m(b));
    System.out.println(b.x + ", " + b.y);
  }
}
```

(a) Mostrare l'output prodotto.

(b) Mostrare lo stato della memoria dati dell'interprete SIMPLESEM appena prima dell'invocazione dell'istruzione di return nel punto (*). Si omettano i link statici.
(Suggerimento: si ricordi la differenza, in Java, nella gestione dell'allocazione della memoria per le variabili rispetto a quella degli oggetti.)

SOLUZIONE

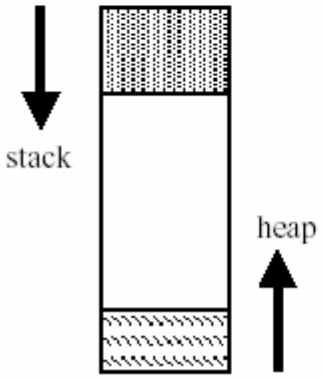

CURRENT
FREE
RET MAIN
E (IND)
B (IND)
RET VAL
RET E.m
DYN LINK
B (IND)
W
RET VAL
RET D.m
DYN LINK
B (IND)
V
B
X
Y
E
X
Y
Z

ESERCIZIO

Si consideri la macchina SIMPLESEM definita durante il corso e descritta nel testo Ghezzi/Jazayeri Programming Language Concepts e si supponga di volerla arricchire per la rappresentazione a runtime dei linguaggi Object Oriented (ad esempio, Java). Si supponga che, come discusso a lezione e nel testo, la gestione della memoria in SIMPLESEM avvenga assegnando al programma una quantità fissa di memoria, allocando lo stack a partire dagli indirizzi bassi e lo heap dagli indirizzi alti (vedi figura). Per semplicità, si supponga che non venga fatto garbage collection. Si ipotizzi che le celle della memoria D della macchina SIMPLESEM di indirizzo 0 e 1 contengano, come usuale, i valori di CURRENT e FREE per la gestione dello stack, mentre la cella di indirizzo 2 venga utilizzata per memorizzare l'indirizzo della prima cella libera dello heap.
Descrivere le operazioni della macchina SIMPLESEM in corrispondenza dell'operazione

x = new X;

dove x è un riferimento a un oggetto di classe X.



SOLUZIONE

Siano (d, o) i valori di distanza, offset associati alla variabile x.
Le istruzioni SIMPLESEM devono effettuare le seguenti operazioni:

- set 2, D[2]- size(X)
- se D[2]<D[1] genera un errore a runtime
- assegna alla variabile a distanza d e offset o (vedi libro per esprimere ciò in SIMPLESEM) il valore D[2]-size(X)