

Linguaggi di programmazione

Appunti

Giovanni Palma e Alex Basta

CONTENTS

CHAPTER	NOMI E AMBIENTE	PAGE
1.1	Nomi e Oggetti denotabili	
1.2	Ambienti e Blocchi Blocchi — • Tipi di Ambiente — • Operazioni sull'ambiente — • Vita di un oggetto —	
1.3	Regole di scope Scope statico — • Scope dinamico —	
1.4	determinare l'ambiente	
CHAPTER	GESTIONE MEMORIA	PAGE
2.1	Allocazione Statica	
2.2	Allocazione Dinamica Allocazione Dinamica con Pila — • Allocazione Dinamica con Heap —	
2.3	Implementazione delle Regole di Scope Scope Statico —	
CHAPTER	ESERCITAZIONI	PAGE
3.1	Scope Esercizio 1 — • Esercizio 2 — • Esercizio 3 — • Esercizio 4 — • Esercizio 5 — • Esercizio 6 — • Esercizio 7 —	

Chapter 1

Nomi e Ambiente

Nell'evoluzione dei linguaggi di programmazione, i *nomi* hanno avuto un ruolo fondamentale nella sempre maggiore astrazione rispetto al linguaggio macchina.

Definition 1.0.1: Nome

I nomi sono solo una sequenza (significativa o meno) di caratteri che sono usati per rappresentare un oggetto, che può essere uno spazio di memoria se vogliamo etichettare dei dati, o un insieme di comandi nel caso di una funzione.

1.1 Nomi e Oggetti denotabili

Spesso, i nomi sono *identificatori*, ovvero token alfanumerici, ma possono essere usati anche simboli (+,-,...). E' importante ricordare che il nome e l'oggetto denotato non sono la stessa cosa, infatti un oggetto può avere diversi nomi (*aliasing*) e lo stesso nome può essere attribuito a diversi oggetti in momenti diversi (*attivazione* e *deattivazione*).

Definition 1.1.1: Oggetti denotabili

Sono gli oggetti a cui è possibile attribuire un nome.

Note:

Non centra con la programmazione ad oggetti

Possono essere:

- Predefiniti: tipi e operazioni primitivi, ...
- Definibili dall'utente: variabili, procedure, ...

Quindi il legame fra nome e oggetto (chiamato **binding**) può avvenire in momenti diversi:

- Statico: prima dell'esecuzione del programma
- Dinamico: durante l'esecuzione del programma

1.2 Ambienti e Blocchi

Non tutti i legami fra nomi e oggetti vengono creati all'inizio del programma restando immutati fino alla fine. Per capire come i binding si comportano, occorre introdurre il concetto di *ambiente*:

Definition 1.2.1: Ambiente

Insieme di associazioni nome/oggetto denotabile che esistono a runtime in un punto specifico del programma ad un momento specifico durante l'esecuzione.

Solitamente nell'ambiente non vengono considerati i legami predefiniti dal linguaggio, ma solo quelli creati dal programmatore utilizzando le *dichiarazioni*, costrutti che permettono di aggiungere un nuovo binding nell'ambiente corrente.

Notare che e' possibile che nomi diversi possano denotare lo stesso oggetto. Questo fenomeno e' detto *aliasing* e succede spesso quando si lavora con puntatori.

1.2.1 Blocchi

Tutti i linguaggi di programmazione importanti al giorno d'oggi utilizzano i *blocchi*, strutture introdotte da ALGOL 60 che servono per strutturare e organizzare l'ambiente:

Definition 1.2.2: Blocco

Pezzo contiguo del programma delimitato da un inizio e una fine che puo' contenere dichiarazioni **locali** a quella regione.

Puo' essere:

- In-line (o anonimo): puo' apparire in generale in qualunque punto nel programma e non corrisponde a una procedura.
- Associato a una procedura

Permettono di strutturare e riutilizzare il codice, oltre a ottimizzare l'occupazione di memoria e rendere possibile la ricorsione.

1.2.2 Tipi di Ambiente

Un'altro meccanismo importante che forniscono i blocchi e' il loro *annidamento*, ovvero l'inclusione di un blocco all'interno di un altro (non la sovrapposizione parziale). In questo caso, se i nomi locali del blocco esterno sono presenti nell'ambiente del blocco interno, si dice che i nomi sono *visibili*. Le regole che determinano se un nome e' visibile o meno a un blocco si chiamano *regole di visibilita'* e sono in generale:

- Un nome locale di un blocco e' visibile a esso e a tutti i blocchi annidati.
- Se in un blocco annidato viene creata una nuova dichiarazione con lo stesso nome, questa ridefinizione *nasconde* quella precedente.

Definition 1.2.3: Ambiente associato a un blocco

L'ambiente di un blocco e' diviso in:

- **locale**: associazioni create all'ingresso nel blocco:
 - variabili locali
 - parametri formali (nel caso di un blocco associato a una procedura)
- **non locale**: associazioni ereditate da altri blocchi (senza considerare il blocco globale), che quindi non sono state dichiarate nel blocco corrente
- **globale**: associazioni definite nel blocco globale (visibile a tutti gli altri blocchi)

1.2.3 Operazioni sull'ambiente

- Creazione: dichiarazione locale, in cui introduco nell'ambiente locale una nuova associazione
- Riferimento: uso di un nome di un oggetto denotato
- Disattivazione/Riattivazione: quando viene ridefinito un certo nome, all'interno del blocco viene disattivato. Quando esco dal blocco riattivo la definizione originale
- Distruzione: le associazioni locali del blocco dal quale si esce vengono distrutte

Note:

Creazione e distruzione di un *oggetto denotato* non coincide necessariamente con la creazione o distruzione sull'associazione tra il nome e l'oggetto stesso, per essere più precisi nemmeno la vita dell'oggetto e del legame è la stessa. Verrà quindi mostrato nel dettaglio

1.2.4 Vita di un oggetto

Definition 1.2.4: Vita

Si definisce **tempo di vita** o **lifetime** di un oggetto o legame il tempo che intercorre tra la sua creazione e la sua distruzione

Per comprendere meglio questo concetto, i seguenti notino gli *eventi fondamentali*

- | Creazione di un oggetto
- | Creazione di un legame per l'oggetto
- | Riferimento all'oggetto, tramite il legame
- | Disattivazione di un legame
- | Riattivazione di un legame
- | Distruzione di un legame
- | Distruzione di un oggetto

Dal punto 1 e 7 è *la vita dell'oggetto*, mentre dall'evento 2 al 6 è *la vita dell'associazione*

Note:

È pertanto vero, quindi, che la vita di un oggetto non coincide con la vita dei legami per quell'oggetto

Esistono 2 modi per categorizzare il tempo di vita di un legame/associazione:

- Vita dell'oggetto più **lunga** di quella del legame

Si consideri questo codice

```

1      program ExampleCode;
2
3      procedure P(var X: integer); begin {...} end;
4      {...}
5      var A: integer;
6      {...}
7
8      P(A); {chiamata a P con A}
```

Nel codice dato, inizialmente il nome A viene associato a un oggetto (un valore intero). Quando si chiama la procedura P(A), l'argomento A viene passato per riferimento, il che significa che all'interno della procedura non viene creato un nuovo oggetto, ma semplicemente un nuovo nome per lo stesso oggetto: X.

Durante l'esecuzione della procedura, X e A sono quindi due nomi che fanno riferimento allo stesso valore in memoria. Qualsiasi modifica apportata a X all'interno della procedura si riflette direttamente su A.

Una volta terminata l'esecuzione della procedura, il legame tra X e l'oggetto viene distrutto, mentre A continua a riferirsi allo stesso valore, eventualmente modificato dalla procedura. Questo è un classico esempio in cui la durata del legame tra un nome (X) e un oggetto è più breve della vita dell'oggetto stesso

- Vita dell'oggetto più **breve** di quella del legame

Si consideri questo codice, piuttosto nasty in C:

```

1      #include <stdio.h>
2      #include <stdlib.h>
3
4      int main() {
5          int *X, *Y;
6          X = (int *) malloc(sizeof(int));
7          Y = X;
8          free(X);
9          X = NULL;
10         return 0;
11     }
12

```

Nel codice illustrato vengono creati due puntatori. L'oggetto puntato da **X**, attraverso il comando **malloc**, punta a un'area di memoria allocata dinamicamente. Di conseguenza, assegnando **Y = X**, anche **Y** farà riferimento allo stesso oggetto puntato da **X**.

Col comando **free(X)**, l'oggetto alla fine della catena viene deallocato, ovvero la memoria precedentemente allocata viene liberata. Successivamente, l'istruzione **X = NULL** imposta **X** a **NULL**, indicando che non punta più a un'area valida di memoria.

Tuttavia, il puntatore **Y** continua a riferirsi all'oggetto che è stato deallocato. Questo crea un *dangling pointer* (puntatore pendente), poiché il legame tra **Y** e l'oggetto non esiste più in modo sicuro. Accedere a **Y** dopo la deallocazione può portare a comportamenti indefiniti e **DA EVITARE CAZZO**

1.3 Regole di scope

Innanzitutto fornirò la definizione di scope

Definition 1.3.1: Scope

Lo **scope** (o **ambito**) è un concetto semantico che determina in quali porzioni di un programma una variabile o un nome è visibile e utilizzabile. Le regole di scope stabiliscono come i riferimenti ai nomi vengono risolti all'interno di un determinato contesto di esecuzione, garantendo che l'uso delle variabili sia coerente e prevedibile.

Come detto in precedenza una dichiarazione locale ad un blocco è visibile in quel blocco e in tutti i blocchi in esso annidati, a meno che intervenga in tali blocchi una nuova dichiarazione dello stesso nome che nasconderà quello precedente (shadowing)

Occorre tuttavia determinare come interpretare le regole di visibilità di una variabile in presenza di porzioni di blocchi eseguiti in posizioni diverse dalle loro definizioni e in presenza di ambienti non locali... nasty vero?

Vi sono due filosofie principali

- **Statico:** Basato sul testo del programma
- **Dinamico:** Basato sul flow di esecuzione

Prima di andare avanti, si noti la seguente annotazione

Note:

Entrambe gli approcci differiscono solo in presenza congiunta di ambiente non locale e non globale e procedura

Vabbuò, è normale non capirci un cazzo solo a parole, si consideri il seguente testo:

```

1      A:{int x = 0;
2          void pippo(int n){
3              x = n+1;
4          }
5          pippo(3);
6          write(x);
7      }

```

```

8      int x = 0;
9      pippo(3);
10     write(x);
11 }
12 write(x);
13 }

```

Che cosa caspita scriveremo a riga 10? Ebbene dipenderà dal tipo di regola di scope, o statica o dinamica. Di seguito sono riportati nel dettaglio

1.3.1 Scope statico

Definition 1.3.2: Scope statico

La regola dello **scope statico** (o **regola dello scope annidato più vicino**) si basa sui seguenti principi:

1. **Ambiente locale di un blocco:** Le dichiarazioni all'interno di un blocco definiscono il suo ambiente locale. Questo include solo le dichiarazioni presenti direttamente nel blocco stesso e non quelle eventualmente presenti nei blocchi annidati al suo interno.
2. **Ricerca delle associazioni di un nome:** Se un nome viene utilizzato all'interno di un blocco, si segue questa gerarchia per determinare quale dichiarazione è valida:
 - Se esiste una dichiarazione del nome nel **blocco locale**, questa è quella valida.
 - Se il nome non è dichiarato nel blocco locale, si cerca nel **blocco immediatamente contenitore**.
 - Se il nome non è ancora trovato, si continua a risalire nei blocchi contenitori fino al più esterno.
 - Se il nome non è dichiarato nemmeno nel blocco più esterno, si cerca nell'**ambiente predefinito del linguaggio**.
 - Se il nome non è presente neanche nell'ambiente predefinito, si genera un errore.
3. **Blocchi con nome:** Un blocco può essere **assegnato a un nome**, e in questo caso tale nome diventa parte dell'ambiente locale del **blocco immediatamente contenitore**. Questo vale anche per i blocchi associati a procedure o funzioni.

Molto più semplicemente si può dire che

Note:

Un nome non locale è risolto nel blocco che *testualmente* lo racchiude

Pertanto nel codice d'esempio nel primo `write(x)` verrà stampato 4, nel secondo 0 e nel terzo 4, in quanto la `x` che la funzione `pippo(3)` modifica è quella dichiarata all'interno del blocco che lo racchiude, in questo caso A. Nel blocco B non verrà modificata la `x` racchiusa nello stesso quindi si stamperà, quindi 0.

Si ha quindi una forte indipendenza dalla posizione della posizione da parte dei nomi. Ad esempio se si dichiara una funzione all'interno di un blocco, il corpo della procedura si riferirà sempre alle regole di scope medesime del blocco in cui è stata dichiarata, pertanto dovunque la funzione verrà chiamata lo scope a cui riferisce sarà sempre lo stesso.

Tra i vantaggi dello scope statico troviamo una maggiore comprensione per il programmatore, ogni nome può essere collegato alla sua dichiarazione semplicemente analizzando la struttura del codice, senza dover simulare l'esecuzione e la facilità di analisi del programma da parte del compilatore che può determinare tutte le occorrenze di un nome e fare controlli di correttezza sui tipi di dati ed eseguire ottimizzazioni del codice prima dell'esecuzione.

1.3.2 Scope dinamico

Definition 1.3.3: Regole di scope dinamico

Secondo le regole di scope dinamico, l'associazione valida per un nome x ad un punto P del programma è la più recente (in senso temporale) associazione creata per x ancora attiva appena il controllo di flusso arriva a P

In pratica occorre andare indietro *nell'esecuzione* per cercare l'occorrenza d'interesse (è l'ultima che è stata introdotta) blocco attivato per ultimo (che deve essere ancora attivo), come riassunto in questa nota quindi:

Note:

Un nome non locale è risolto nel blocco attivato più di recente e non ancora disattivato

Quando un nome non è dichiarato localmente in un blocco, viene cercato nel blocco attivato più recentemente che lo contiene. Questo significa che la risoluzione dei nomi segue una logica *LIFO* (*Last In First Out*), cioè una gestione a stack basata sull'ordine di chiamata delle funzioni. Questo approccio è più semplice da gestire a runtime perché si basa solo sulla pila di attivazione, senza necessità di altre strutture dati

1.4 determinare l'ambiente

L'ambiente è quindi determinato da:

- Regole di scope (statico o dinamico)
- Regole di visibilità
- Regole di binding (intervengono quando una procedura P è passata come parametro ad un'altra procedura mediante il formale X)
- Regole per il passaggio di parametri

Chapter 2

Gestione Memoria

Prima di discutere su i diversi modi in cui la memoria (RAM e HD solo se necessario) puo' essere gestita dal compilatore di un linguaggio, e' importante identificare cosa esattamente deve essere contenuto all'interno di essa. Sicuramente, ogni *dato* che deve essere salvato durante l'esecuzione del programma dovra' avere un posto nella memoria, come ad esempio le variabili, ma ci sono anche informazioni per il *controllo dell'esecuzione* che necessitano di essere memorizzati.

La vita di un oggetto corrisponde con tre meccanismi di allocazione di memoria:

- **Allocazione statica:** l'oggetto viene allocato una volta sola, prima dell'inizio dell'esecuzione del programma, e deallocato alla fine dell'esecuzione,
pertanto è una memoria allocata a tempo di compilazione
- **Allocazione automatica:** l'oggetto viene allocato all'entrata di un blocco (tipicamente una funzione) e deallocato all'uscita del blocco.
- **Allocazione dinamica:** l'oggetto viene allocato e deallocato esplicitamente dal programmatore tramite chiamate a funzioni di allocazione e deallocazione (ad esempio, `malloc` e `free` in C). *pertanto è una memoria allocata a tempo di esecuzione*

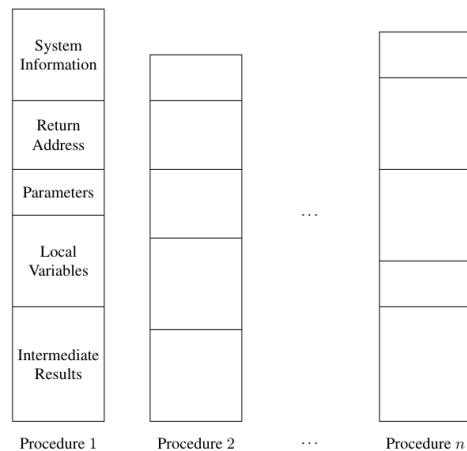
Questo tipo di allocazione di serve di due aree di memoria:

- pila (stack): gli oggetti sono allocati con una politica LIFO, utilizzato per le variabili locali e i parametri formali delle funzioni
- heap: gli oggetti sono allocati e deallocati in qualsiasi ordine, utilizzato per gli oggetti dinamici (puntatori)

2.1 Allocazione Statica

Non sarebbe possibile utilizzare solo una memoria con allocazione statica? Sicuramente per le variabili *globali*, *costanti* che non dipendono da altri valori non noti inizialmente e anche per le *tabelle* che utilizza il compiler, l'allocazione statica funziona benissimo.

Per quanto riguarda i sottoprogrammi, si puo' pensare di allocare per ognuna di esse tutto lo spazio necessario per i parametri e le variabili locali (e altre informazioni necessarie), dato che possiamo determinare tutto cio' prima di eseguire il programma:



Questo metodo funziona solo se siamo sicuri che, se una procedura e' attiva, allora non puo' chiamare la stessa procedura ricorsivamente. Questo e' perche' esiste solo un'istanza per ogni procedura allocata in memoria e non e' possibile sapere quante volte una funzione verra' chiamata ricorsivamente. Quindi, se vogliamo implementare la ricorsione, serve l'allocazione *dinamica* (stack di chiamate):

2.2 Allocazione Dinamica

2.2.1 Allocazione Dinamica con Pila

Ogni istanza di sottoprogramma viene memorizzata con un *frame* (o *record di attivazione*) che contiene tutte le informazioni necessarie. Quando un'istanza viene attivata, il relativo frame viene messo in cima a una **pila**, la struttura dati naturale in quanto se una procedura A chiama una procedura B, allora siamo sicuri che B deve terminare prima che A possa continuare l'esecuzione e terminare anch'esso.

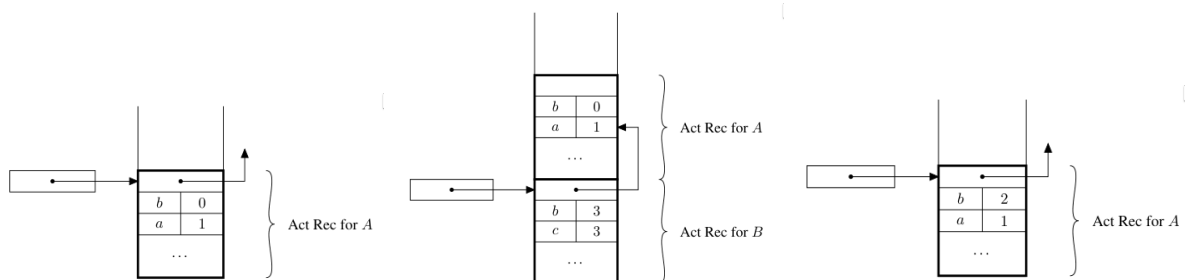
Note:

L'allocazione dinamica e' utile anche quando non c'e' ricorsione come meccanismo per risparmiare memoria.

Vediamo un esempio:

```

1 A:{
2   int a = 1;
3   int b = 0;
4
5   B:{
6     int c = 3;
7     int b = 3;
8   }
9   b = a + 1;
10 }
```



Vediamo piu' in dettaglio cosa viene memorizzato nei record di attivazione:

Record di attivazione

Per un semplice blocco anonimo, il corrispondente frame ha tale forma:

Dynamic chain pointer
Local variables
Intermediate results

Note:

Nella realta' la maggior parte dei linguaggi usa l'allocazione statica per blocchi anonimi per maggiore efficienza di calcolo (sacrificando pero' l'efficienza di memoria).

Mentre per le procedure e' un po' piu' complesso:

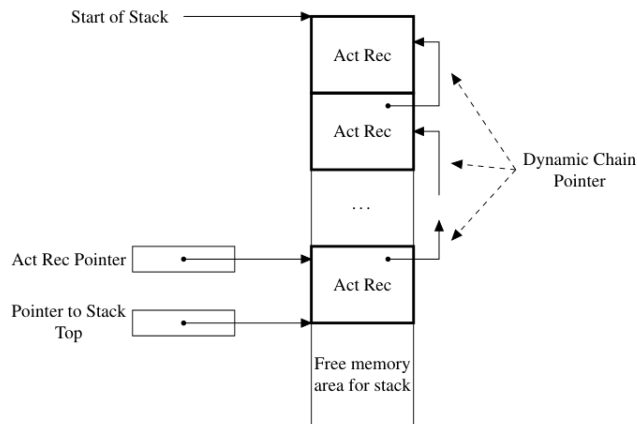
Dynamic Chain Pointer
Static Chain Pointer
Return Address
Address for Result
Parameters
Local Variables
Intermediate Results

Vediamo in dettaglio cosa sono tutti sti dati:

- **Intermediate Results:** serve per memorizzare risultati intermedi di equazioni complicate e per risultati di chiamate ricorsive.
- **Local Variables** e **Parameters:** per le variabili locali e parametri.
- **Dynamic Chain Pointer:** puntatore all'ultimo RdA creato sulla stack, l'insieme di tutti i puntatori dinamici e' chiamata *catena dinamica*.
- **Static Chain Pointer:** informazione necessaria per implementare lo scope statico, vedremo piu' avanti.
- **Return Address:** indirizzo di memoria della prima istruzione da eseguire quando termina la procedura corrente.
- **Address for Result:** indirizzo dove puo' essere salvato il valore di ritorno della funzione (sara' un indirizzo interno al frame della funzione chiamante).

Gestione della Pila

Vediamo la struttura di un sistema a pila (discendente):



Come puo' notare dall'immagine, ci sono due puntatori esterni che puntano a posti specifici sull'ultimo record di attivazione:

- **Pointer al RdA:** e' un puntatore ad un luogo predeterminato all'interno del frame usato come base da cui si puo' calcolare l'offset per accedere alle variabili locali. Questo offset e' determinabile staticamente dal compilatore (ad eccezione del caso di variabili di dimensione variabile).
- **Stack Top Pointer:** come si puo' indovinare, e' il puntatore al primo indirizzo libero di memoria dopo l'ultimo RdA. Puo' essere omesso se e' possibile calcolare lo stesso indirizzo partendo dal pointer al RdA.

Il funzionamento corretto della pila di RdA e' dato dalla collaborazione fra il chiamante e il blocco chiamato, che eseguono dei blocchi di codice inseriti dal compilatore (o interprete) prima e dopo chiamate a procedure e blocchi anonimi:

- **Sequenza di Chiamata:** eseguito dal chiamante subito prima della chiamata
- **Prologo:** eseguito immediatamente all'inizio del blocco chiamato
- **Epilogo:** eseguito alla fine del blocco
- **Sequenza di Ritorno:** eseguito dal chiamante immediatamente dopo la chiamata

Al momento della chiamata, la Sequenza di Chiamata e il Prologo devono:

- **Modificare PC**
- **Allocare spazio sulla pila**
- **Modifica pointer RdA**
- **Passaggio parametri**
- **Memorizzare registri**

Quando il blocco o processo chiamato termina, l'Epilogo e la Sequenza di Ritorno devono:

- **Modificare PC**
- **Ritornare Valori**
- **Recuperare Registri**
- **Deallocare spazio sulla pila**

Note:

Sono stati omessi meccanismi per l'implementazione delle regole di scope. Vedremo queste piu' avanti.

2.2.2 Allocazione Dinamica con Heap

Nel caso in cui vogliamo dare la possibilità a chi usa il linguaggio di allocare esplicitamente memoria a run-time o di usare oggetti di dimensioni variabili sorge il seguente problema: la vita degli oggetti non e' per forza LIFO, ovvero un oggetto creato prima di un altro puo' essere rimosso dalla memoria prima di un'altro oggetto creato dopo, come nel seguente blocco di codice:

```
1  int *p, *q;  
2  p = malloc(sizeof(int));  
3  q = malloc(sizeof(int));  
4  *p = 0;  
5  *q = 1;  
6  free(p);  
7  free(q);
```

Dobbiamo usare quindi la *heap*, ovvero una regione di memoria i cui blocchi possono essere allocati/deallocati in momenti arbitrari.

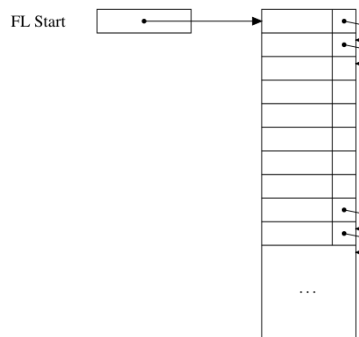
Note:

La heap che abbiamo appena definito non centra niente con la struttura dati usata per la "heap sort".

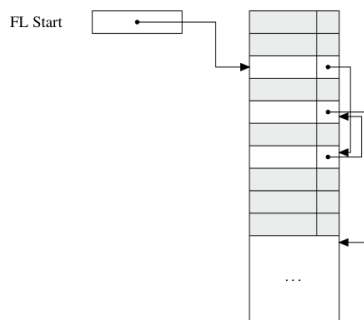
Esistono due categorie principali di metodi di gestione della heap a seconda della lunghezza dei blocchi che memorizza, che possono essere di *dimensione fissa* o *variabile*.

Dimensione fissa

La heap viene suddivisa in blocchi di dimensione fissa abbastanza limitata che sono inizialmente collegati tutti assieme nella *lista libera*:



A run-time, quando viene richiesto un blocco di memoria, il primo elemento della lista libera viene rimosso e restituito al processo che ha richiesto la memoria, mentre la testa della lista libera si sposta al prossimo elemento della lista. Vediamo un esempio di heap a dimensione fissa dopo alcune operazioni di allocazione/deallocazione (i blocchi grigi sono in uso):

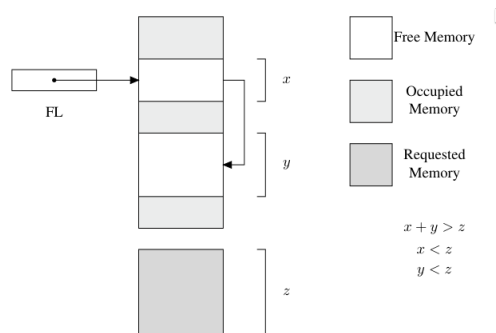


Dimensione variabile

Nel caso volessimo poter allocare array la cui dimensione e' determinata solo a runtime, la soluzione a dimensione fissa non sarebbe adeguata, dato che l'array puo' essere di dimensioni maggiori rispetto ai blocchi fissi e non si possono allocare piu' blocchi perche' la memoria deve essere per forza contigua.

In questi casi e' necessario poter richiedere un blocco di dimensione arbitraria. In un sistema di questo tipo, bisogna prestare attenzione ad usare *operazioni efficienti* e a limitare lo *spreco di memoria* che puo' avvenire in due situazioni:

- **Frammentazione interna:** viene richiesto un blocco di dimensione n ma ne viene restituito uno di dimensione $k > n$, quindi $k - n$ parole si perdono.
- **Frammentazione esterna:** lo spazio totale nella memoria sarebbe abbastanza per soddisfare una richiesta di dimensione n , ma i blocchi liberi sono separati o non contigui.



Esistono due meccanismi diversi per gestire blocchi di dimensione variabile:

- **Unica lista:**

Inizialmente, la lista libera e' composta da un solo blocco che occupa tutta la heap. Quando viene fatta la richiesta per un blocco di dimensione n , le prime n parole dell'heap vengono allocate e l'inizio della lista si sposta di n . Si va avanti cosi' finche' la memoria rimasta dall'inizio della lista libera alla fine della heap non e' abbastanza per soddisfare la richiesta. In questo caso dobbiamo riutilizzare memoria deallocata, che puo' essere fatto in due modi:

- **Uso diretto della lista libera:** si scorre lungo la lista finche' si trova un blocco di dimensioni $k > n$. Se la differenza fra la grandezza del blocco e della memoria effettivamente usata e' maggiore di una tolleranza, allora il blocco viene diviso ed il blocco di dimensione $k - n$ viene reinserito nella lista. Possono essere usate due politiche di ricerca diverse per scegliere quale blocco prendere: *first fit* e *best fit*. Quando un blocco viene deallocato, si guarda se blocchi adiacenti sono anch'essi liberi e in caso affermativo si fondono per formare un blocco piu' grande (*compattazione parziale*).
- **Compattazione della memoria libera:** tutti i blocchi attivi vengono spostati alla fine della heap, molto efficiente ma funziona solo quando possiamo spostare la memoria.

- **Liste libere Multiple:**

Per ridurre il costo operativo di cercare un blocco di dimensione arbitraria, e' possibile utilizzare diverse liste per diverse dimensioni di blocchi. Quando viene richiesto un blocco di dimensione n , si scorrono le liste finche' una che contiene blocchi di dimensioni adeguate non e' vuota. Anche in questo caso e' possibile ridurre la dimensione dei blocchi per ridurre la frammentazione interna, esistono due metodi:

- **Buddy system:** la dimensione dei blocchi aumenta per potenze di 2. Si calcola l'intero minore k tale che $2^k \geq n$ e si controlla se la relativa lista ha blocchi liberi. Altrimenti, si va a cercare nella lista $k + 1$ e si divide il blocco in due blocchi da 2^k (la dimensione che volevamo), uno viene allocato e l'altro viene spostato nella lista corretta. Quando viene deallocato, la meta' cerca il suo compagno nella lista libera e se lo trova si uniscono e tornano nella lista originale.
- **Fibonacci:** funzionamento equivalente ma le dimensioni dei blocchi seguono la sequenza di Fibonacci, che sale piu' lentamente quindi porta a meno frammentazione ma piu' tempo.

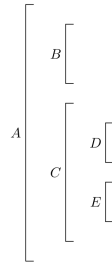
TODO: disegni esplicativi

2.3 Implementazione delle Regole di Scope

Ora che abbiamo capito come viene gestita la memoria di un programma dal compilatore, soprattutto per quanto riguarda i RdA, vediamo come possiamo implementare le regole di scope quando un blocco deve accedere a variabili non-locali.

2.3.1 Scope Statico

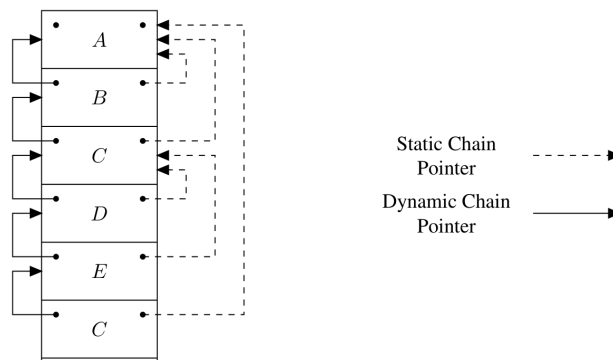
Se vengono implementate le regole di scope statico, allora l'ordine da seguire quando stiamo cercando il valore di una variabile non-locale non e' necessariamente quello dettato dalla catena dinamica, ovvero l'ordine delle chiamate. Infatti, l'RdA corretto e' determinato dalla struttura statica (testuale) del programma, seguendo quindi l'ordine di annidamento dei blocchi. Vediamo un esempio:



Data la struttura soprastante, immaginiamo di chiamare in ordine A, B, C, D, E, C e che rimangano tutte attive:

- Aggiungiamo inizialmente l'RdA di A allo stack di sistema, aggiornando lo SP e l'RdA pointer. Dato che e' il primo sulla pila, non ha nessun link.
- Man mano che aggiungo i RdA di B, C, \dots aggiorno sempre SP e RdA pointer, ma e' anche necessario determinare il link dinamico e statico.

Quindi con tutte le chiamate aperte la situazione e' questa:



Come al solito, il puntatore di catena dinamica punta all'RdA *temporalmente precedente* (quello appena sotto nella pila), mentre il puntatore di catena **statica** indica l'RdA del blocco che *contiene strutturalmente* quello in cui siamo.

Note:

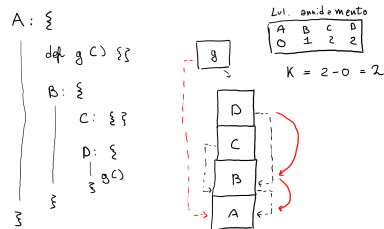
Se un sottoprogramma e' annidato a livello k , allora la catena statica sara' lunga k .

Supponiamo di essere in E e di voler accedere alla variabile x in modo statico dichiarata in A :

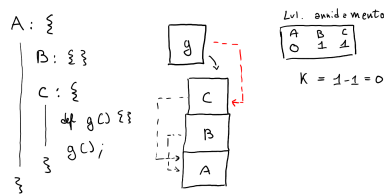
- Seguendo la catena statica, controllo se x e' dichiarata in C .
- Non c'e', quindi continuo a seguire i link statici e arrivo in A , dove trovo il valore cercato.

Il supporto a run-time della catena statica e' compito della sequenza di chiamata, prologo e l'epilogo che abbiamo visto prima per le chiamate. L'approccio piu' comune e' quello dove il chiamante calcola il puntatore a catena statica che passa poi al chiamato, e' abbastanza semplice e puo' essere diviso in due casi:

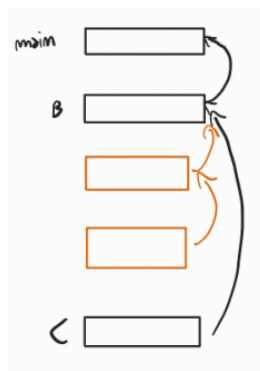
- **Il chiamato e' esterno al chiamante:** secondo le regole di visibilita', questa situazione e' possibile solo se il chiamante e' annidato internamente rispetto al blocco del chiamato. Cio' significa che tale blocco e' ancora attivo sulla pila, come tutti i blocchi annidati fino a quello del chiamante. Dato che sappiamo il livello di annidamento di ciascun blocco (puo' essere calcolato staticamente), basta trovare la differenza fra il livello del chiamante meno il chiamato e fare questo numero di salti sulla catena statica del chiamante (che e' in cima alla stack). In questo modo ci troviamo nel RdA del blocco contenente la definizione del chiamato, e ci basta solo passare l'indirizzo di tale RdA al chiamato che lo imposta come link statico.



- **Il chiamato e' interno al chiamante:** tale situazione puo' succedere solo se il chiamato e' definito nello stesso blocco dove viene chiamato. Siamo quindi in un caso speciale del punto precedente dove la distanza di annidamento e' 0, quindi basta passare come link statico al chiamato il puntatore RdA del record contenente la chiamata (che sara' quello in cima).



Il numero di "salti" da fare per raggiungere il RdA corretto e' calcolabile staticamente dal compiler usando la *tabella dei simboli*, che memorizza anche il livello di annidamento al quale e' stata dichiarata la variabile cercata. Pero' non e' possibile stabilire staticamente la locazione esatta di memoria dove si trovera' tale variabile non-locale, dato che, pur sapendo il numero di "salti", non sappiamo quanti e quali RdA si trovano fra ogni salto a run-time. Per questo motivo ci serve la catena statica.



Il Display

E' un po' uno schifo dover seguire sta catena statica, dato che se siamo a un livello k di annidamento, dobbiamo per forza eseguire k accessi a memoria per arrivare finalmente al frame giusto. Anche se nella pratica generalmente non e' un problema, possiamo ottimizzarlo quindi lo facciamo.

Chapter 3

Esercitazioni

3.1 Scope

3.1.1 Esercizio 1

Testo

```
1      {
2          int x =2;
3          int func ( int y){
4              x = x+y;
5              write(x);
6          }
7          {
8              int x = 5;
9              func(x);
10             write(x);
11         }
12         write(x);
13     }
```

- Si descriva il comportamento del programma assumendo uno scope statico.
- Si descriva il comportamento del programma assumendo uno scope dinamico.

Soluzione

- **Scope statico:** il programma stampa 7, 5, 7
- **Scope dinamico:** il programma stampa 10, 10, 2

3.1.2 Esercizio 2

Testo

```
1      {
2          int x = 0;
3          int next () {
4              x = x+1;
5              write(x);
6          }
7          int exec(){
8              int x = 3;
9              next();
10             write(x);
11         }
```

```

11     }
12     exec();
13     write(x);
14 }

```

- Si descriva il comportamento del programma assumendo uno scope statico
- Si descriva il comportamento del programma assumendo uno scope dinamico

Soluzione

- **Scope statico:** il programma stampa 1, 3, 1
- **Scope dinamico:** il programma stampa 4 infatti `next` prenderà l'istanza di `x` dell'ultimo ambiente non disattivato, ovvero `exec`, modifico poi il valore di `x` nell'ambiente `exec`, 4 dato che l'ambiente di `exec` è stato modificato, 0 dato che `exec` è stato disattivato si ha che `x = 0`

3.1.3 Esercizio 3

Testo

```

1  {
2      int x = 0;
3      void pippo(value int y, rif int z){
4          z= x+y+z;
5      }
6      {
7          int x = 1;
8          int y = 100;
9          int z = 30;
10         pippo(x++,x); // ricordarsi che il x++ prima passa il valore di x nudo e
                        // crudo poi come side-effect modifica x staticamente in questo caso,
                        // pertanto il secondo parametro sara' 2: pippo(1, 2) -> 3 = 0+ 1 +2
11         pippo(x++, x); // pippo(3, 4) -> 7 = 0 + 3 + 4
12         write(x); // 7
13     }
14     write(x); // 0 perche' il blocco e' finito, terminato e riprende la variabile
                        // x dichiarata all'inizio
15 }

```

Si descriva il comportamento del programma assumendo uno scope statico

Soluzione

Il programma stampa 7, 0

3.1.4 Esercizio 4

Testo

```

1  {
2      void pippo(value int y, value int z){
3          x=y+z;
4      }
5      int x = 100;
6      pippo(x++, x); // pippo(100, 101) -> 201, quindi x = 201
7      pippo(x++, x); // pippo(201, 202) -> 403, quindi x = 403
8      write(x); // 403
9  }

```

Si descriva il comportamento del programma assumendo uno scope dinamico

Soluzione

Il programma stampa 403

3.1.5 Esercizio 5

Testo

```
1 {
2     int f(value int k){ //1. 2, 1. 1
3         int g (value int n){
4             return n+y //
5         };
6         int x=10;
7         int y=10;
8         if k=1 return g(x) + g(y); // 40
9         else {
10            int x = 30;
11            int y= 30;
12            return f(k-1);
13        }
14    }
15    int x =50;
16    int y=50;
17    x= f(2); // x = 40
18    write(x); // 40
19 }
```

si risolva utilizzando uno scope dinamico

Soluzione

Il programma stampa 40

3.1.6 Esercizio 6

Testo

```
1 {
2     int x =10;
3     int v =5;
4     void B(){
5         write(x);
6     }
7     void A(z){
8         int x = z * v;
9         B();
10    }
11    A(??) // si ha che qualsiasi valore di al posto di ?? si scrivera' 10
12 }
```

Fornire una chiamata alla funzione A di modo che il prgramma, usando scope statico stami il valore 10

```
1 {
2     int x =10;
3     int v =5;
4     void B(){
5         write(x);
6     }
7     void A(z){
8         int x = z * v;
9         B();
```

```

10     }
11     A(??) // si scrive 2 cazzo
12 }

```

fornire una chiamata per stampare il valore 10, ma usando uno scope dinamico

Soluzione

PEr la prima parte si stampera sempre 10, per la seconda parte si passa il parametro 2

3.1.7 Esercizio 7

Testo

```

1 {
2     int x =1;
3     int y=2;
4     void A(){
5         int x = 2;
6         int k = 3;
7     }
8     void B(){
9         int x = 5; // 2
10        A();
11        // ** 1**
12        x=2;
13        C():
14    }
15    void C(){
16        int z =5;
17        // ** 2 **
18    }
19    B();
20 }

```

SI descriva qual è il contenuto delle variabili attive al punto ****1**** e ****2****, utilizzando scope statico e dinamico

Soluzione

- **Scope statico:**

- **Punto 1:** x = 5, y = 2
- **Punto 2:** x = 1, y = 2, z = 5

- **Scope dinamico:**

- **Punto 1:** x = 5: nel momento in cui si attiva A() lo shadowing impone che che x in memoria è 2, ma appena A termina si "disattiva" e si riprende x=5 definito nel punto B, y = 2
- **Punto 2:** x = 2, y = 2, z = 5

zio pera