SML

Gurpreet S. Surinder

20/06/2018

Il primo **costrutto if** è stato introdotto nel linguaggio **Algol 60**.
Programmazione in the large: sviluppo di software di grandi dimensioni.
Comando sequenziale: è un comando che indica che due operazioni devono essere eseguite in sequenza ";".
Go-to: il codice tende a diventare complesso da seguire, difficile da implementare in tutte le sue casistiche (ad esempio quando si salta in un blocco diverso).

Macchina astratta (Macchina virtuale)

Si denota con il simbolo $\mathcal{M}$
Una macchina astratta è un insieme di algoritmi e strutture dati che permettono di eseguire e memorizzare programmi attraverso una macchina ospite ($\mathcal{M}_{\mathcal{O}}$).
$\mathcal{M}_{\mathcal{O}_{\mathcal{L}_{\mathcal{O}}}}$ denota una macchina ospite che capisce il linguaggio $\mathcal{L}_{\mathcal{O}}$
$\mathcal{M}_{\mathcal{L}}$ identifica una macchina astratta che capisce ed esegue il linguaggio $\mathcal{L}$. $\mathcal{L}$ è il linguaggio macchina di $\mathcal{M}_{\mathcal{L}}$, un programma non è altro che una sequenza di istruzioni in $\mathcal{L}$, $\mathcal{P}^{\mathcal{L}}$ è un programma scritto in linguaggio $\mathcal{L}$.

Macchina astratta ad implementazione compilativa

La tecnica compilativa si basa sull' idea di tradurre una volta per tutte l'intero programma in un programma funzionalmente equivalente scritto nel linguaggio della macchina ospite. Il compito di eseguire tale traduzione e' eseguito da un programma detto **compilatore**.
Questa implemetazione presenta:

- **Vantaggi**: implementazione efficiente;
- **Svantaggi**:
  - non (molto) portabile;
  - perdita di informazione sulla struttura del programma sorgente.

Compilatore

Si denota con il simbolo $\mathcal{C}$
Traduce il programma $\mathcal{L}_1$ in un altro linguaggio $\mathcal{L}_2$.
$\mathcal{C}^{\mathcal{L}_{\mathcal{A}}}_{\mathcal{L}\mathcal{L}_{\mathcal{O}}}$identifica un compilatore $\mathcal{C}$ da linguaggio $\mathcal{L}_{\mathcal{O}}$ in linguaggio $\mathcal{L}$ scritto in linguaggio $\mathcal{L}_{\mathcal{A}}$.
Un compilatore può anche trasformare una ricorsione in coda in modo da farla diventare una iterazione.
Ricorsione: è il processo che si verifica quando una routine richiama se stessa, di norma su porzioni di dati di volta in volta sempre più piccole, se è una ricorsione incontrollata si può incorrere in un **stack overflow**.
Esiste una variante chiamata **ricorsione in coda**, la quale consiste nel chiamare la funzione che ritorna il valore senza fare ulteriori computazioni.
Iterazione: si realizza quando si eseguono più righe di codice al verificarsi di una determinata condizione. Esistono due tipi di iterazione:

1. **Iterazione determinata:** il numero di volte che il ciclo è eseguito è **stabilito a priori** e non può essere cambiato
2. **Iterazione indeterminata:** il ciclo viene ripetuto **fino a** quando una qualche espressione diventa **falsa**

Macchina astratta ad implementazione interpretativa

La tecnica interpretativa consiste nel realizzare tutte le componenti della macchina (cioe' tutti gli algoritmi e le strutture dati che li definiscono) mediante programmi e strutture dati del linguaggio della macchina ospite.
Questa implementazione presenta:

- **Vantaggi:**
  - flessibile e portabile;
  - debugging semplificato.
- **Svantaggi**: poco efficiente.

Interprete

Si denota con il simbolo $\mathcal{I}$.
$\mathcal{I}$ capisce il programma $\mathcal{L}$ senza aver bisogno di trasformarlo ed implementa il ciclo fetch/decode/load/execute/save. (Esistono delle CPU microprogrammate delle quali è possibile aggiornare il firmware, presentano quindi un microinterprete di microistruzioni che esegue il ciclo fetch/decode/load/execute/save.).
$\forall i \epsilon \mathcal{D}, \mathcal{I}^{\mathcal{L}_{\mathcal{O}}}_{\mathcal{L}}(\mathcal{P}^{\mathcal{L}}, i) = \mathcal{P}^{\mathcal{L}}(i)$
Per qualsiasi input i, l'interprete applicato a $\mathcal{P}^{\mathcal{L}}$ ed i ritorna lo stesso risultato ritornato da $\mathcal{P}^{\mathcal{L}}$ applicato ad i
$\mathcal{I}^{\mathcal{L}_{\mathcal{O}}}_{\mathcal{L}}$identifica un interprete $\mathcal{I}$ del linguaggio $\mathcal{L}$ scritto nel linguaggio $\mathcal{L}_{\mathcal{O}}$

Espressione & Comandi

Ambiente e sottodivisioni

Un **ambiente** è un insieme di associazioni fra nomi ed oggetti denotabili esistenti ad uno specifico punto dell'esecuzione del programma.
Esistono tre tipi diversi di ambiente, che definiscono le varie associazioni:

- **Ambiente locale:** insieme delle associazioni create nel blocco (inclusi i parametri)
- **Ambiente non locale:** associazioni ereditate da altri blocchi
- **Ambiente globale:** associazioni comuni a tutti i blocchi

Quando una funzione **produce e/o modifica** un valore o uno stato al di fuori del proprio **scope locale** viene definito **effetto collaterale**.
Una **dichiarazione** è un meccanismo con il quale si crea un'associazione nell'ambiente.
Il **binding** è il nome dato alla creazione di un'associazione tra nome ed un oggetto denotabile.
Quando due diversi nomi denotano lo stesso oggetto si sta verificando un effetto chiamato **aliasing**.
Un **identificatore** è una sequenza di caratteri usata per denotare qualcosa.
Il **blocco** è una regione testuale di un programma avente un inizio e una fine, esso può contenere dichiarazioni locali alla regione.

Espressione

Un'espressione è un'entità sintattica la cui valutazione produce un valore (oppure non termina). Essa presenta due elementi

1. **l-value:** espressioni che possono stare a sinistra di un'operazione di assegnamento (in generale indicano una locazione di memoria)
2. **r-value:** espressioni che possono stare a destra di un'operazione di assegnamento e che quindi forniscono valori assegnabili a variabili

Un espressione può essere valutata(eseguita) in due momenti diversi i quali prendono il nome di **valutazione:**

1. **Lazy:** tipo di valutazione in cui il calcolo viene ritardato fin quando non viene efettivamente richiesto. Questo tipo di valutazione è più difficile da implementare rispetto al prossimo
2. **Eager:** l'espressione viene valutata appena viene legata ad una variabile.

Di una espressione si può creare un **albero sintattico**, cioè un albero che rappresenta la struttura sintattica di una stringa in accordo a determinate forme grammaticali. [1]

[1] Per approfondimenti sull'albero sintattico: Sintassi e Semantica - Alfonso Miola, slide 27 in poi

Le espressioni sono le principali cause di **overflow** e/o di **underflow**, dovuti alle operazioni matematiche che come risultato presentano un numero non codificabile in binario nei bit a disposizione della macchina.
Chiusura: una coppia di ambiente e espressione in cui questa dev'essere valutata. L'ambiente deve comprendere (almeno) tutte le variabili libere dell'espressione.

Passaggio di parametri

Esistono vari modi per passare un parametro ad una funzione:

- **Passaggio di parametri per nome:** il parametro viene sintatticamente ricopiato all'interno del corpo della funzione. La valutazione dell'ambiente del parametro dipende da due casi:
  - **Per nome con cattura:** il valore della variabile passata ad una funzione viene ricercato all'interno della funzione.

    ```
    int x=0;
    int foo(name int y){
            int x=2;
            return x+y;
    }
    ...
    int a = foo(x+1);
    ```

    In questo caso y non ha valore 0+1 ma diventa x+1, cioè la riga 4 diventa

    ```
    ...
    return x+x+1;
    ...
    ```

  - **Per nome senza cattura:** il parametro viene valutato nell'ambiente del chiamante, non in quello della funzione chiamata. Insieme al parametro viene passato anche il suo ambiente di valutazione
- **Passaggio di parametri per valore-risultato:** il passaggio avviene per valore, al termine della procedura il valore del parametro attuale viene ricopiato nell'espressione con **l-value** passata
- **Passaggio di parametri per valore:** viene valutato il **r-value** del parametro ed il valore viene passato alla funzione. Nel caso i parametri siano delle strutture il passaggio per valore crea una copia campo-campo di quella struttura (costo notevole per strutture di grandi dimensioni).
- **Passaggio di parametri per costante:** avviene lo stesso procedimento del **passaggio per valore** ma il valore non può essere modificato. Nel caso i parametri siano delle strutture il passaggio per costante passa un riferimento alla struttura e non permette che questa possa in qualche modo essere modificata.

- **Passaggio di parametri per riferimento:** viene passato alla funzione il **l-value** del parametro.

Si possono passare anche le funzioni come parametro, durante questa valutazione i **binding** non locali possono essere cercati in due modi:

- **Deep binding:** nel blocco in cui la funzione passata come parametro è stata dichiarata (Java)
- **Shallow binding:** nell'ambiente della funzione passata (linguaggio con scope dinamico)

Notazione

- Notazione infissa: consiste nel mettere l'operatore tra i due elementi ($a+b$). Questo tipo di notazione è difficile da realizzare rispetto alle altre notazioni visto che necessita di conoscere l'ordine di precedenza degli operandi.
- Notazione prefissa: consiste nel mettere l'operando prima dei due elementi ($+a \quad b$).
- Notazione postfissa: consiste nel mettere l'operando dopo i due elementi dell'espressione ($a \quad b+$).

Chiamata e ritorno da una procedura

Sequenza di operazioni che avvengono per realizzare una chiamata ad una procedura(funzione):

1. Modifica del programma counter
2. Allocazione dello spazio sulla pila
3. Modifica del puntatore del record di attivazione
4. Passaggio dei parametri
5. Salvataggio dei registri
6. Esecuzione del codice per l'inizializzazione (se presente)

Sequenza di operazioni che avvengono al ritorno da una chiamata ad una procedura:

1. Ripristino del program counter
2. Restituzione dei valori di ritorno
3. Ripristino dei registri
4. Esecuzione codice per la finalizzazione (se esiste)
5. Deallocazione dello spazio sulla pila

Comando

Un comando è un'entità sintattica la cui valutazione non restituisce valore ma può comunque avere effetti collaterali.
Si dice che il comando **void** ritorni "un unico valore" invece che "nessun valore". Essendo le funzioni dei linguaggi di programmazione derivate da funzioni matematiche, esse non possono non avere un valore, grazie al fatto che non esistono funzioni matematiche con codominio vuoto, quindi tutte le funzioni **void** ritornano un unico valore (non ci è dato sapere quale).

Variabili

Il concetto di variabile che comunemente si ha, ha due "proprietà":

- sono entità **denotabili**, ovvero entità a cui possiamo associare un nome
- possono contenere un valore (locazioni di memoria)

Le variabili hanno diverse proprietà in base al tipo di linguaggio in cui si trovano:

- **Linguaggi iperativi**
  Questo tipo di linguaggio presenta delle variabili modificabili, quindi le aree di memoria associate ad un nome possono contenere un valore modificabile. I linguaggi iperativi sono gli unici che presentano il **doppio mapping** cioè il nome è legato alla variabile la quale a sua volta è legata al valore memorizzato in essa.
- **Linguaggi orientati agli oggetti**
  I linguaggi orientati agli oggetti presentato delle variabili con **modello a riferimento**, di conseguenza gli l-value di un'espressione possono fare riferimento ad uno spazio nello heap. Ciò permette anche la modifica dell'assegnamento di uno spazio di memoria.
- **Linguaggi funzionali:** non presentano variabili modificabili.

Tipi (Type)

Un tipo di dato è una **collezione di valori omogenei e rappresentabili**, dotata di un insieme di operazioni che manipolano tali valori.

- Un linguaggio di programmazione $\mathcal{L}$ è **insicuro ai tipi** se nel programma $\mathcal{P}^{\mathcal{L}}$ è possibile violare vincoli di tipo senza che la macchina astratta $\mathcal{M}_{\mathcal{L}}$ se ne accorga.
- Due tipi sono **strutturalmente equivalenti** quando:
  - un tipo equivale a se stesso
  - un **alias** (variante del nome) di un tipo è equivalente a quel tipo
  - Se due tipi sono costruiti applicando lo stesso costruttore di tipo a tipi equivalenti, allora essi sono equivalenti.

- Due tipi sono **equivalenti per nome** solo se hanno lo stesso nome (un tipo è equivalente per nome solo a se stesso)
- Le conversioni tra due tipi vengono dette **unchecked** quando non viene data la possibilità al compilatore di verificare se il **cast** (conversione da un tipo ad un'altro) è corretto.
- Il tipo di dato che fa parte di tipi di dato in cui è definito un ordine specifico è detto **ordinale**.
- Due tipi sono **compatibili** quando il tipo1 è ammesso in qualsiasi contesto in cui sarebbe richiesto il tipo2.
- Si ha **poliformismo universale parametrico** quando un valore ha un'infinità di tipi diversi che si ottengono da un unico schema di tipo generale.
- Si ha **inferenza di tipo** quando il tipo di dato viene rilevato ed automaticamente riconosciuto dal **type checker** (ML presenta inferenza di tipo).
- Un linguaggio ha **tipizzazione**
  - **statica:** se al tempo di compilazione è possibile determinare tutti i vincoli dei tipi;
  - **dinamica:** se i controlli avvengono a tempo di esecuzione
- **Record union:** due campi che condividono la stessa memoria e solo una di queste può essere assegnata alla volta

```
... union{
        int ultimoanno;
        struct{
                int inpari;
                int anno;
        }stud_in_corso;
}
```

  ultimoanno e stud_in_corso condividono la stessa memoria (di grandezza determinata dal campo più grande)

- **Conversione implicita (coercizione):** la conversione da un tipo all'altro viene fatta dalla macchina astratta senza che questa sia specificata esplicitamente

```
int a=3;
char b='3';
a=b;
```

  è possibile eguagliare due tipi diversi grazie alla conversione implicita

- Un sistema di tipi viene detto **polimorfo** quando uno stesso oggetto può avere più di un tipo
- Un valore esibisce **polimorfismo di sottotipo** (o **limitato**) quando ha un'infinità di sottotipi diversi, che si ottengono da uno schema di tipo generale, sostituendo ad un opportuno parametro i sottotipi di un tipo assegnato (**extends** in Java)
- Un sistema di tipi viene detto **monomorfo** quando ogni oggetto del linguaggio ha un unico tipo
- I tipi di dato di cui i valori non sono costituiti da altri valori sono detti **scalari**

Array

Le dimensioni di un array possono essere fissate in momenti diversi:

- **forma statica:** le dimensioni sono note a tempo di compilazione, viene memorizzato interamente nel **RDA**
- **forma fissata al momento dell'elaborazione:** le dimensioni vengono risolte al tempo di esecuzione. In questo caso l'array viene memorizzato nello **stack**, il **RDA** viene diviso in due parti: una a lunghezza fissa ed una a lunghezza variabile. Nella parte a lunghezza fissa viene memorizzato un puntatore all'array, il quale invece è allocato nella parte a lunghezza variabile.
- **forma dinamica:** non vengono mai fissate, le dimensioni sono variabili a tempo di esecuzione, per questo motivo l'array viene allocato nello **heap**

Quando un **array** viene allocato viene creato anche il suo **dope vector**. Il **dope vecor** contiene:

- un puntatore alla prima locazione nella quale è memorizzato l'array (sempre presente)
- **Range:** le dimensioni
- **Extent:** intervalli degli indici
- **Stride:** la distanza in memoria tra due elementi consecutivi

Un array può essere memorizzato in due modi: in **ordine di riga**, l'array viene memorizzato in modo contiguo, e in **ordine di colonna**, in questo caso l'array è memorizzato nei primi/secondi/terzi... elementi delle colone. La memorizzazione di un **array** per ordine di **riga** è più diffusa rispetto a quella per ordine di **colonna** perchè essa rende semplice lo **slicing**[2] dell'array.
Si possono calcolare gli indirizzi di memoria dei valori di un array in base a quale metodo è stato usato per memorizzare l'array:

[2]Suddivisione di un **array** in grandezze determinate

- **Colonna:**

  1. int a[A][B] è un array multidimensionale di interi (si assuma che la dimensione di un intero sia D byte) con a[0][0] che ha indirizzo INDIRIZZO_BASE, qual è l'indirizzo di a[Y][Z]?
     Posizione in memoria = INDIRIZZO_BASE + (D*A*Z) + (D*Y)
  2. int a[A][B][C] è un array multidimensionale di interi (si assuma che la dimensione di un intero sia D byte) con a[0][0] che ha indirizzo INDIRIZZO_BASE, qual è l'indirizzo di a[X][Y][Z]?
     Posizione in memoria = INDIRIZZO_BASE + (D*A*B*Z) + (D*A*Y) + (D*X);

- **Righe:**

  1. int a[A][B] è un array multidimensionale di interi (si assuma che la dimensione di un intero sia D byte) con a[0][0] che ha indirizzo INDIRIZZO_BASE, qual è l'indirizzo di a[Y][Z]?
     Posizione in memoria = INDIRIZZO_BASE + (D*B*Y) + (D*Z);
  2. Se gli array sono memorizzati per righe ed int a[A][B][C] è un array multidimensionale di interi (si assuma che la dimensione di un intero sia D byte) con a[0][0] che ha indirizzo INDIRIZZO_BASE, qual è l'indirizzo di a[X][Y][Z]?
     Posizione in memoria = INDIRIZZO_BASE + (D*B*C*X) + (D*C*Y) + (D*Z);

Scope statico e dinamico

Scope statico
Lo **scope statico** per risolvere una variabile esamina partendo dal blocco interno fino ad arrivare a quello esterno, è più efficiente visto che le associazioni sono note a tempo di esecuzione ma è molto più complesso da implementare.
Presenta **il puntatore a catena statica**, il quale punta al blocco immediatamente esterno a quello chiamato, così facilita la ricerca dei riferimenti ai **binding** non locali. Viene creato anche il **display** cioè un vettore che memorizza un record di attivazione per ciascun blocco di annidamento; ciò serve a ridurre il costo di scansione della catena statica (quando ci si riferisce ad un oggetto non locale, il suo puntatore viene reperito dal vettore, così non serve cercare in tutta la **catena statica** risparmiando in questo modo le risorse della macchina).
Scope dinamico
Lo **scope dinamico,** per risolvere una variabile cerca il più recente legame ancora attivo stabilito a tempo di esecuzione, questo scope è più semplice da implementare ma è meno efficiente e meno leggibile.

Memoria[3]

La memoria può essere allocata nello stack, nello heap o nel **Data Segment**[4] di un programma, il momento di allocazione può cadere o durante il **run-time** o il **compile-time.**

Allocazione di memoria

Il termine allocazione viene utilizzato per indicare l'assegnazione di un blocco di memoria RAM ad un programma.

Allocazione statica

Si ha allocazione statica quando viene definita una variabile statica (interna o esterna con la clausola **static**), essa non è modificabile e non può nemmeno essere rilasciata. Questa variabile viene definita durante il **compile-time** ed allocata nel **Data-segment**. Un oggetto allocato con questa modalità mantiene lo stesso indirizzo di memoria per tutta l'esecuzione del programma.

Allocazione dinamica

Si ha **allocazione dinamica** durante il **run-time**, cioè tutte le varibili istanziate durante una funzione o una ricorsione. Viene usato principalmente lo **heap** per questo tipo di allocazione, quindi è modificabile e rilasciabile ma solo con chiamate opportune da parte del programmatore.

Stack & Heap

Nella scelta della memoria tra **stack** e **heap** dipende sia dal compilatore sia dal programmatore, per esempio i risultati intermedi di un'espressione non vengono sempre memorizzati nello stack ma il copilatore può decidere dove allocare lo spazio.

Stack

> Function parameters and local variables are generally allocated on a stack. A stack, in computer terms, is basically a linear structure in memory where information (values for variables, for example) are stored sequentially with the most recently added value at the "top" or the end of the memory structure from which values are also read and removed. This allows a function to call itself recursively, and

[3] Riferimenti e integrazione: pdf_2010_Claudio_Fornaro

[4] In computing, a data segment (often denoted .data) is a portion of an object file or the corresponding virtual address space of a program that contains initialized static variables, that is, global variables and static local variables. The size of this segment is determined by the size of the values in the program's source code, and does not change at run time.
`https://en.wikipedia.org/wiki/Data_segment`

> have separate copies of all its parameters and local variables retained separately for each call. The variable values are all "pushed" onto the stack, and then execution begins with a new set of values in the new call. When the function returns, the variable values are restored by "popping" the values from the "top" of the stack in reverse order, thus restoring the previous values of all the variables.[5]

RDA (stack frame): lo **stack frame** memorizza nello stack tutte le informazioni relative all'istanza di un sottoprogramma.

> A stack frame is a memory management technique used in some programming languages for generating and eliminating temporary variables. In other words, it can be considered the collection of all information on the stack pertaining to a subprogram call. Stack frames are only existent during the runtime process. Stack frames help programming languages in supporting recursive functionality for subroutines. A stack frame also known as an activation frame or activation record.[6]

Durante una funzione nello **stack frame** vengono memorizzati:

- i risultati intermedi
- le variabili locali
- il puntatore di catena dinamica: punta al record di attivazione precedente; dal momento che i record di attivazione hanno tutti una dimensione diversa, bisogna tenere traccia di dove inizia lo **stack frame precedente**, in modo da poterlo ripristinare in seguito.
- il puntatore di catena statica (se viene implementato scope statico)
- l'indirizzo di ritorno
- l'indirizzo del risultato
- e i parametri
- nel caso di un metodo anche un riferimento all'oggetto su cui il metodo viene chiamato

Invece nel caso di **stack frame con blocchi anonimi** vengono memorizzati:

- il puntatore di catena dinamica
- le variabili locali
- i risultati intermedi delle espressioni

[5] Descrizione presa da: What is the difference between the stack and the heap?
[6] Origine e approfondimenti: Stack Frame

Heap

> A heap is generally used for larger data structures or memory that the programmer wants to explicitly manage. Whenever "new" or "malloc" is used in C++ or C respectively, the memory for that request is allocated on a heap, and then it's up to the coder to make sure that is gets freed at the appropriate time with "delete" or "free". Whereas stack data is usually easy to manage because the compiler allocates and frees the space, heap data is easy to mess up because it often requires manually written code to determine when it gets allocated and freed. But it's appropriate if you want to manage large long-lived blocks of memory independently of the scope of a function. And although all memory in a computer is linear at some level, a heap is less linear in the way it's managed because it maintains pointers to different areas of memory making it easier to allocate and free blocks of memory in an arbitrary sequence instead of strictly in reverse order.[7]

Esistono due modi per gestire lo **heap**:

1. **Heap a blocchi variabili:** lo heap in questo caso è costituito da un unico blocco di memoria che viene suddiviso, creando blocchi di grandezza utile durante il runtime.

   - (**Lista libera**) Quando viene richiesta della memoria si restituisce un blocco di dimensione più appropriata e lo si rimuove dalla lista. Quando invece la memoria viene liberata, essa viene riaggiunta alla lista e fusa con i blocchi liberi adiacenti in modo da formare un unico blocco di memoria libera.
   - (**Compattazione della memoria libera**) Lo heap ogni volta che viene richiesta memoria viene suddiviso continuando dalla allocazione precedente senza liberare lo spazio prima (anche in caso di deallocazione dell'oggetto), ma quando finisce la memoria di heap libera, questo metodo "**compatta**" i blocchi allocati, creando una zona continua di spazio. Alla fine aggiorna il puntatore dello heap allo spazio così liberato.

2. **Heap a blocchi di dimensione fissa:** lo **heap** è costituito da una lista di $\boldsymbol{n}$ blocchi di dimensione fissa. Quando viene richiesta della memoria, il primo elemento libero viene rimosso dalla lista e restituito al richiedente, mentre il puntatore viene spostato sul blocco sucessivo. Alla deallocazione il blocco viene reinserito nella lista.

   - (**Buddy system**) Il **buddy system** suddivide i blocchi in potenze di 2, quando viene richiesto un blocco di dimensione n si cerca il più piccolo blocco libero tale che $2^k \geqslant n$

[7] Descrizione presa da: What is the difference between the stack and the heap?

- (**Heap di Fibonacci**) In questo caso la suddivisione è uguale al **buddy system** ma la dimensione dei blocchi allocabili segue la sequenza di Fibonacci e quindi crescono di dimensione più lentamente.

Questo tipo di heap ha due modalità per restituire il blocco di memoria richiesto:

- (**Best fit**) La strategia **best fit** consiste nello scorrere tutta la lista alla ricerca del blocco che combacia meglio con la dimensione del blocco richiesto
- (**First fit**) La strategia **first fit** consiste nel assegnare una soglia s alla dimensione d del blocco richiesto, restituendo il primo blocco che ha dimensione compresa tra d e $d + s$

Questi due tipi di gestione dello **heap** (blocchi variabili o a dimensione fissa) possono essere soggetti a **due** tipi di **frammentazione:**

1. **interna**, ovvero il blocco di memoria che viene dato alla funzione (e/o variabile....) è leggermente più grande di quello richiesto (la differenza di memoria è sprecata)
2. **esterna**, ovvero non è possibile trovare n byte contigui da restituire al programma anche se in totale sono presenti, ma sono sparsi per tutto lo heap.

Garbage collector

> Garbage collection is the process of automatically determining what memory a program is no longer using, and making it available for other use. It is performed by a subsystem called a garbage collector, which can be part of the runtime system of a particular programming language, or an add-on library. Garbage collection can, but does not have to, be assisted by the compiler, the hardware, or the OS.[8]

Esistono varie tecniche di implementazione di un **garbage collector:**

- **Con contatore dei riferimenti:** ad ogni oggetto allocato viene assegnato un contatore pari a 1, quando un oggetto comincia a puntare a quell'oggetto il suo contatore viene incrementato di 1. Quando la variabile che punta all'oggetto viene riassegnata ad un altro oggetto, il contatore dell'oggetto originale viene decrementato di 1, quindi gli oggetti possono essere deallocati quando hanno contatore uguale a 1
  Svantaggi: non è in grado di rimuovere strutture circolari (A punta B e B punta A)

[8] Definizione presa da: The Very Basics of Garbage Collection

- **Stop and copy:** lo heap viene diviso in due parti di uguale dimensione. Solo una delle parti è attiva alla volta, quando la prima parte sta per esaurirsi, viene fatta una visita di tutti gli oggetti allocati nello heap e quelli validi vengono copiati nella seconda parte
- **Mark and sweep:** si attraversa l'albero composto da tutti gli oggetti memorizzati nello heap, marcando quelli in uso. Una volta fatto, vengono liberati tutti i blochi non marcati.
  Svantaggi:
  - causa frammentazione esterna
  - richiede tempo proporzionale alla dimensione dello heap, indipendentemente da quale sia la percentuale effettivamente utilizzata
  - le posizioni degli oggetti deallocati vengono spesso riciclate, così facendo gli oggetti vecchi si trovano vicino a quelli nuovi, creando un possibile problema per la cache
- **Mark and compact:** funziona allo stesso modo del **mark and sweep**, solo che nella fase di **sweep** gli oggetti in memoria vengono riallocati in maniera che siano tutti contigui, in questo modo viene superato il problema delle posizioni riciclate del **mark and sweep**

Info generali (stack-heap-allocazione dinamica-allocazione statica)

In un linguaggio a sola allocazione statica la memoria dedicata alle variabili viene assegnata prima dell'esecuzione, questo fa si che la memoria dedicata ai sottoprogrammi sia già definita prima dell'esecuzione. La ricorsione molte volte viene implementata durante il **run-time** per questo non è possibile definire la memoria usata prima dell'esecuzione, ciò non permette la ricorsione con un allocazione statica.
La **allocazione statica** alloca sempre tutta la memoria che potrebbe essere necessaria, per cui potrebbe essere allocata memoria anche per le procedure che non verranno richiamate durante l'esecuzione del programma. Invece nel caso di utilizzo di uno stack si alloca solo ciò che è necessario, non avendo uno spreco della memoria.

Programmazione ad oggetti

Problema della classe fragile: Quando una classe, molto in alto nella gerarchia viene modificata

- è possibile che qualche sottoclasse smetta di funzionare correttamente perchè si ha apportato una modifica alla logica della classe sulla quale si basavano le classi figlie

- devono essere ricompilate tutte le classi figlie in quanto devono essere aggiornati gli **offset**[9] ai quali trovare le variabili

Ereditarietà multipla: una classe può ereditare funzionalità e caratteristiche da più di una classe base. Presenta però un problema, è molto difficile determinare il metodo giusto da chiamare quando si estendono due classi che hanno un metodo con la stessa firma.
Ereditarietà multipla con replicazione: una classe eredita da classi che hanno la stessa classe base

```
B extends A
C extends A
D extends B,C
```

vTable: in un linguaggio ad oggetti con tipi statici una **vtable** è una struttura dati che tiene traccia di tutti i metodi definiti, ridefiniti o ereditati di una classe.
Problemi del ADT che il paradigma orientato agli oggetti risolve:

- è più semplice estendere dei tipi di dato già esistenti
- non c'è bisogno di andare a ridefinire tutte le operazioni già esistenti sui tipi di dati derivati
- possibilità di avere array con tipi di dato diversi ma compatibili

References

[1] Appunti basati sul lavoro di Slavetto: `https://slavetto.github.io/primo-anno/` codice git: `https://github.com/slavetto/slavetto.github.io`

[2] Integrazione con le slide di Luca Albeni: SL_Slide_Luca_Albeni.zip

[3] Corso di programmazione in C di Claudio Fornaro: /Riferimenti/18-DynAlloc.pdf

[4] Linguaggi Sentassi e Semantica di Alfonso Miola: /Riferimenti/Linguaggi_Sintassi_e_Semantica.pdf

[5] Integrazione con: `http://www.dmi.unict.it/~barba/Architetture.html/MATERIALE-IN-RETE/MA/implementazione.htm`

[9] offset è un numero intero che indica la distanza tra due elementi all'interno di un gruppo di elementi dello stesso tipo. L'unità di misura in cui si esprimono gli offset è normalmente il numero di elementi.