# 12. Astrazione delle procedure

Il compilatore è responsabile di fornire un'implementazione per ciascun costrutto del linguaggio di programmazione, che si suddivide principalmente in due categorie: le istruzioni individuali e le procedure. Queste ultime costituiscono l'astrazione essenziale che rende possibile la programmazione pratica, facilitando lo sviluppo di sistemi software complessi attraverso principi chiave come l'occultamento delle informazioni, l'uso di spazi di nomi distinti e separabili e la creazione di interfacce uniformi.

Dato che l'hardware offre solo un supporto limitato per tali astrazioni, il compilatore ha anche il compito di implementarle, rendendole poi efficienti grazie all'ottimizzazione del codice.

# Panoramica pratica

Il compilatore deve prendere decisioni specifiche riguardo alla collocazione di ogni valore, sia nominato che anonimo, e sul metodo di calcolo per ciascun risultato, come nel caso delle operazioni tra variabili (ad esempio, come dovrebbe calcolare $x \times y$ o gestire una dichiarazione *case*).

Un'altra sfida fondamentale consiste nell'individuare e accedere a oggetti o valori creati o manipolati da codice esterno, non visibile al compilatore, come altri file o librerie.

Inoltre, le operazioni di caricamento e collegamento dinamico aggiungono ulteriore complessità, richiedendo un controllo attento. Questi aspetti diventano centrali soprattutto nell'implementazione delle procedure, dove il compilatore deve garantire coerenza ed efficienza nel trattamento di ogni elemento.

# Interfaccia tra compilazione e esecuzione

Le questioni fondamentali per i compilatori riguardano vari aspetti, come il comportamento durante la fase di compilazione rispetto a quello di esecuzione, ossia il confronto tra comportamenti *statici* e *dinamici*. Tra i compiti principali del compilatore c'è la gestione della memoria, che include l'assegnazione di spazi di memoria e la mappatura dei nomi verso indirizzi specifici, oltre alla generazione di codice per accedere ai valori.

Il compilatore deve poi occuparsi delle interfacce con altri programmi, linguaggi e con il sistema operativo, assicurando un'implementazione efficiente. Nel processo di generazione

del codice per ciascuna procedura, è essenziale che i singoli blocchi combacino, formando un programma coerente e funzionante.

Ogni procedura eredita un insieme di nomi, che include variabili, valori e procedure, così come la capacità di individuare e accedere agli oggetti in memoria. Inoltre, ogni procedura eredita una "storia di controllo," ovvero la sequenza di chiamate che ne ha determinato l'invocazione, insieme al meccanismo che consente di restituire il controllo al chiamante.

Ogni procedura crea anche uno spazio di nomi indipendente, che può nascondere altri nomi attraverso meccanismi di *scoping*. Infine, ogni procedura accede alle interfacce esterne tramite nomi specifici, che possono comprendere parametri, con supporto per collegamenti e caricamenti dinamici. È fondamentale garantire la sicurezza di entrambe le parti coinvolte nell'interfaccia.

# Tre astrazioni inerenti le procedure

---

Le procedure offrono un'**astrazione del controllo** che include ingressi e uscite ben definiti, oltre a un meccanismo per restituire il controllo al chiamante. Spesso queste strutture prevedono una parametrizzazione, che consente di modulare il comportamento della procedura in base ai dati forniti.

Lo **spazio di nomi pulito** è un'altra caratteristica fondamentale delle procedure: ogni procedura possiede un proprio spazio di nomi, dove i nomi locali sono visibili solo all'interno della procedura stessa e possono oscurare nomi identici presenti in ambiti esterni, restando però invisibili all'esterno.

Infine, l'**interfaccia esterna** facilita l'accesso alla procedura tramite il suo nome e i parametri associati, garantendo protezione sia per chi chiama sia per chi viene chiamato. La procedura invocata può operare indipendentemente dal contesto di chiamata, mantenendo così una separazione funzionale cruciale per la gestione delle responsabilità.

# Costruire sistemi complessi

---

Le procedure sono essenziali per costruire sistemi complessi, poiché richiedono un accordo a livello di sistema che definisca convenzioni precise per la gestione della memoria, la protezione, l'allocazione delle risorse, le sequenze di chiamata e la gestione degli errori. Questi accordi coinvolgono l'architettura della macchina (ISA), il sistema operativo e il compilatore, formando un insieme coerente di regole operative.

Le procedure offrono un accesso condiviso alle funzionalità di sistema, che comprendono gestione della memoria, flusso di controllo, gestione degli interrupt e interfacce per

dispositivi di input/output. Oltre a queste funzioni, mettono a disposizione strutture di protezione, timer, flag di sincronizzazione e contatori. Al contempo, creano un contesto privato per ciascuna invocazione, istanziando una memoria separata e incapsulando informazioni sul flusso di controllo e sulle astrazioni dei dati.

Un vantaggio cruciale delle procedure è la possibilità di compilazione separata, che permette di sviluppare programmi complessi con tempi di compilazione contenuti e agevola la collaborazione tra più programmatori. Questa compilazione modulare richiede procedure indipendenti: senza di essa, la costruzione di sistemi di grandi dimensioni sarebbe praticamente impossibile.

La convenzione di collegamento di una procedura assicura che essa erediti un ambiente di esecuzione valido e che l'ambiente del chiamante venga correttamente ripristinato al termine della procedura. Il compilatore genera codice apposito per rispettare queste convenzioni, mantenendo l'integrità del flusso di esecuzione.

A livello concettuale, una procedura è un'astrazione realizzata attraverso il software. L'hardware di base fornisce solo un supporto parziale, limitato alla gestione di bit, byte, numeri interi, numeri reali e indirizzi. Non comprende, tuttavia, aspetti come i punti di ingresso e uscita, le interfacce o i meccanismi di chiamata e ritorno. Pur supportando il trasferimento del controllo (chiamata e ritorno), l'hardware non gestisce né la sequenza completa di chiamata (come la preservazione del contesto) né lo spazio dei nomi e gli ambiti nidificati. Tali elementi vengono realizzati attraverso una serie di meccanismi appositamente progettati, messi a disposizione dal compilatore, dal sistema di esecuzione, dal linker, dal loader e dal sistema operativo.

### Tempo di esecuzione vs tempo di compilazione

I collegamenti (e il codice del corpo della procedura) vengono eseguiti in tempo di esecuzione, mentre il codice per il collegamento viene generato in tempo di compilazione. Tuttavia, la struttura del collegamento è progettata molto tempo prima di entrambi questi momenti.

# La procedura come astrazione del controllo

---

Le procedure definiscono un flusso di controllo chiaro: la chiamata di una procedura avviene in un punto specifico del codice, accompagnata da un insieme di parametri effettivi, e, dopo l’esecuzione della procedura, il controllo ritorna immediatamente al punto di chiamata.

Inoltre, la maggior parte dei linguaggi moderni consente l’uso della ricorsione, un aspetto fondamentale per l'implementazione delle procedure.

Implementare procedure con questo comportamento richiede del codice che salvi e ripristini l'**indirizzo di ritorno** e che associ i parametri effettivi a quelli formali. È necessario anche

creare uno spazio di memoria per le variabili locali della procedura e, potenzialmente, per i parametri stessi.

Nelle chiamate ricorsive, però, sorge la questione di dove allocare questo spazio per ciascuna attivazione. Qui entra in gioco il compilatore, che genera codice per assicurare che ogni chiamata gestisca correttamente queste allocazioni a tempo di esecuzione.

La vera sfida emerge nella gestione dello **stato di $p$ durante l'esecuzione di $q$** nelle chiamate ricorsive. Per affrontare questo problema, si impiega una strategia che assegna una posizione di memoria unica per ciascuna attivazione della procedura. Nei casi più semplici, un *stack* di blocchi di memoria conserva gli indirizzi di ritorno e la memoria locale. Quando si utilizzano chiusure, l'allocazione avviene nello *heap*, con codice generato dal compilatore per orchestrare queste operazioni a tempo di esecuzione.

In sintesi, il sistema di collegamento delle procedure incapsula il codice specifico di ciascuna procedura, conferendole un'interfaccia uniforme, paragonabile alla struttura regolare di un muro di mattoni, a differenza di una disposizione più caotica come un muro di pietre irregolari.

# Le procedure come spazio di nomi

---

Ogni procedura crea uno spazio di nomi proprio, in cui quasi ogni nome può essere dichiarato localmente. I nomi locali hanno la precedenza su quelli non locali con lo stesso identificatore, restando invisibili all'esterno della procedura stessa. Nelle procedure annidate, queste convenzioni sono applicate secondo regole definite di *scoping lessicale*, che permettono alle procedure interne di accedere alle dichiarazioni delle procedure esterne.

Lo scoping lessicale offre un meccanismo durante la fase di compilazione per associare variabili "libere" e semplifica le regole per la denominazione e la risoluzione dei conflitti. Questo tipo di scoping consente al programmatore di introdurre liberamente nomi locali senza preoccuparsi di conflitti con nomi esterni.

**Attenzione!** Nel punto $p$, a quale dichiarazione di $x$ si riferisce il codice corrente? In fase di esecuzione, dove si trova $x$? E come può il parser gestire l'inclusione o l'esclusione di $x$ entrando e uscendo dagli scope?

Il compilatore risolve queste questioni modellando lo spazio di nomi tramite **tabelle di simboli** con supporto per lo scoping lessicale, che tracciano le variabili e i loro contesti, associando ciascun nome allo scope appropriato.

# Tabelle di simboli con scoping lessicale

---

Il compilatore richiede un record unico per ogni dichiarazione, ma gli scope lessicali annidati possono introdurre dichiarazioni duplicate. Per gestire questi contesti, vengono usati tre principali metodi:

- `insert(nome, livello)` consente di creare un record per il nome a un livello specifico;
- `lookup(nome, livello)` restituisce un puntatore o indice del nome richiesto;
- `delete(livello)` rimuove tutte le dichiarazioni presenti a un certo livello.

Molteplici schemi di implementazione sono stati proposti, ma qui ci concentreremo sul concetto di base, poiché le tabelle hash, pur essendo un metodo complesso e dettagliato, non sono l'unica soluzione possibile. Le tabelle di simboli, dunque, sono strutture utilizzate durante la compilazione per risolvere i riferimenti ai nomi. Più avanti vedremo le strutture corrispondenti usate in fase di esecuzione per l'indirizzamento.

### Concetto generale

Per ogni scope, si crea una nuova tabella e la si collega in una catena per la ricerca, in un'implementazione definita a "fascio di tabelle". La funzione `insert()` crea, se necessario, una nuova tabella e aggiunge il nome al livello corrente, mentre `lookup()` scorre la catena di tabelle per restituire la prima occorrenza del nome. La funzione `delete()` rimuove la tabella del livello superiore nella catena quando viene raggiunto il livello specificato.

Questo sistema è particolarmente pratico quando il compilatore deve mantenere la tabella per scopi come il debugging, e ciascuna tabella viene tipicamente implementata con una struttura hash.

### Implementazione delle tabelle di simboli con scoping lessicale

Nell'implementazione più dettagliata, `insert()` aggiunge ogni nuova voce all'inizio della lista per il nome specifico, mentre `lookup()` può accedere direttamente alla posizione del nome. La funzione `delete()` procede eliminando ciascun elemento dichiarato nel livello da rimuovere, partendo dall'inizio della lista. Per migliorare le prestazioni, un indice sparso può velocizzare le operazioni, e una tabella densa permette di limitare l'uso di memoria.

Un chiaro vantaggio di questa struttura è la velocità di `lookup`, ma il principale svantaggio è che `delete` può richiedere un tempo proporzionale al numero di variabili dichiarate nel livello.

# La procedura come interfaccia esterna

---

La denominazione riveste un ruolo essenziale per permettere l'uso delle chiamate di procedura come interfaccia universale. Il sistema operativo necessita di un punto di avvio

per eseguire un programma, e il programmatore deve definire esplicitamente questo punto di inizio. Nella maggior parte dei linguaggi di programmazione, la procedura *main* assume questa funzione, stabilendo così il punto di ingresso del programma e garantendo un avvio coerente dell'esecuzione.

---

**Esempio.** Quando l'utente invoca “grep” da una linea di comando, il sistema operativo trova l'eseguibile, crea un processo e organizza l'esecuzione di “grep”. Concettualmente, esegue una chiamata `fork()` e una `exec()` dell'eseguibile “grep”. Il codice di “grep” proviene dal compilatore e viene collegato al sistema di runtime, che avvia l’ambiente di esecuzione e chiama `main`. Al termine di `main`, il sistema di runtime viene chiuso e il controllo è restituito al sistema operativo. Quando “grep” necessita di servizi di sistema, effettua una chiamata di sistema, come `fopen()`.

---

# Allocazione delle variabili

---

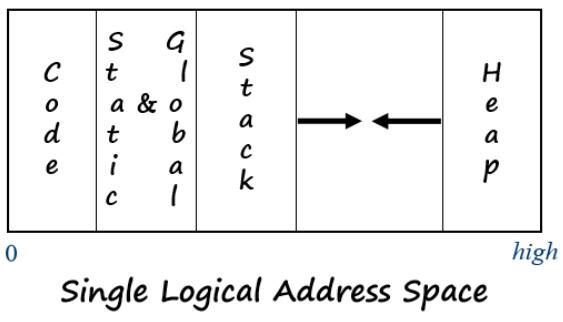
Le variabili locali e automatiche vengono gestite nel record di attivazione della procedura o in un registro. La definizione "automatiche" indica che la loro durata è limitata alla durata della procedura che le contiene.

Invece, le variabili statiche mantengono uno scope che può essere legato a una procedura specifica o a un file intero, assicurando che la loro durata persista per tutta l'esecuzione del programma.

Allo stesso modo, le variabili globali, allocate in aree dati globali dedicate per variabile, file o programma, permangono per l’intero arco di esecuzione.

# Posizionamento delle strutture dati in esecuzione

---

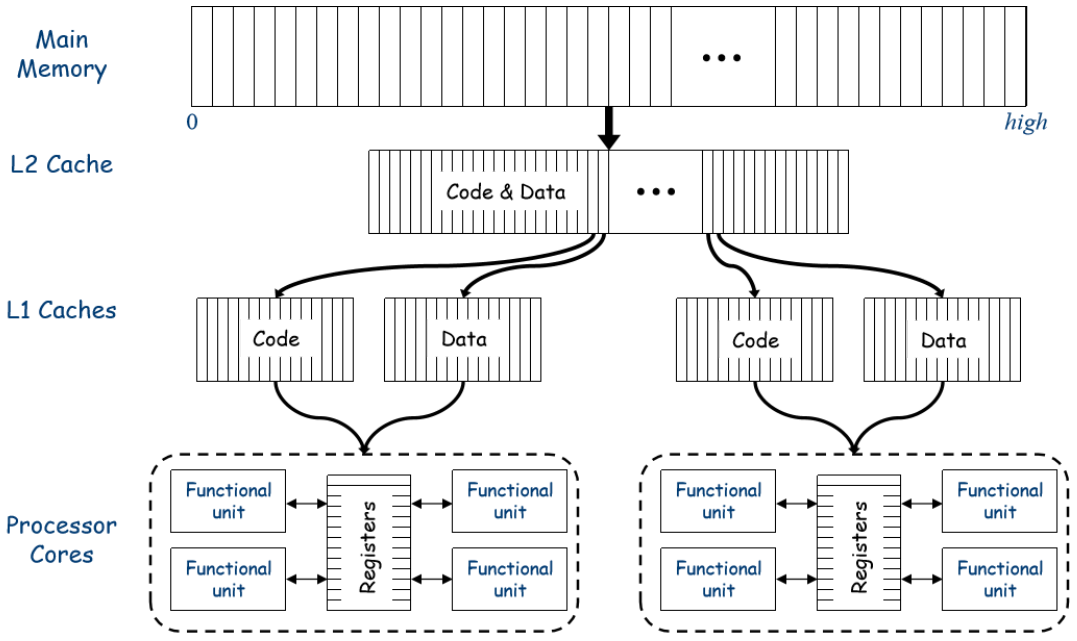


L’organizzazione classica della memoria prevede che il codice, i dati statici e i dati globali abbiano dimensioni note e utilizzino etichette simboliche per la gestione degli indirizzi.

Al contrario, heap e stack crescono e si riducono dinamicamente all'interno di uno spazio di indirizzi virtuali. Questa crescita opposta tra stack e heap è un modello classico per l’ottimizzazione della memoria, come suggerito da Knuth, consentendo un uso efficiente dello spazio. Il codice e i dati, pur condividendo lo spazio degli indirizzi, possono essere fisicamente separati.

# Memorizzazione delle variabili locali

Nel modello più semplice, ogni scope ha un’area di memoria riservata, e per ogni attivazione di uno scope si crea un *activation record* (AR), che può includere anche informazioni di controllo. Nei contesti ricorsivi, ogni nuova attivazione genera un AR separato per la procedura.




In modelli più avanzati, tutte le variabili locali della procedura condividono un unico AR, e la struttura statica definisce in anticipo le relazioni tra scope, permettendo al compilatore di organizzare l'accesso a questi spazi di memoria.

# Traduzione dei nomi locali

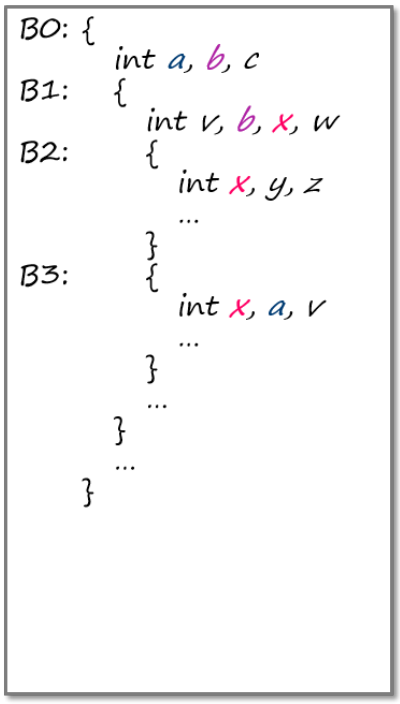
Il compilatore rappresenta un’istanza di una variabile locale come una coordinata statica, espressa dalla coppia `<livello, offset>`, in cui "livello" indica il livello di annidamento lessicale e "offset" un valore univoco per quello scope. Questa coordinata consente al compilatore di generare in modo preciso gli indirizzi di memoria necessari per l'esecuzione,

garantendo un accesso efficiente ai nomi locali tramite la tabella dei simboli, con il livello e l'offset stabiliti in fase di compilazione.

# Memoria per i blocchi all'interno di una singola procedura

---

I dati a lunghezza fissa possono sempre essere posizionati a un offset costante dall'inizio di una procedura.




- La variabile $a$ dichiarata al livello 0 sarà sempre il primo elemento di dati, memorizzato al byte 0 nell'area di dati a lunghezza fissa.
- La variabile $x$ dichiarata al livello 1 sarà sempre il sesto elemento di dati, memorizzato al byte 20 nell'area di dati fissa.
- La variabile $x$ dichiarata al livello 2 sarà sempre l'ottavo elemento di dati, memorizzato al byte 28 nell'area di dati fissa.

Ma cosa succede alla variabile $a$ dichiarata nel secondo blocco al livello 2?

# Dati a lunghezza variabile

---

```
B0: {  int a, b
       ...
       assign value to a
       ...
B1:    {  int v(a), b, x
          ...
B2:       {  int x, y(8)
             ...
          }
       }
    }
```

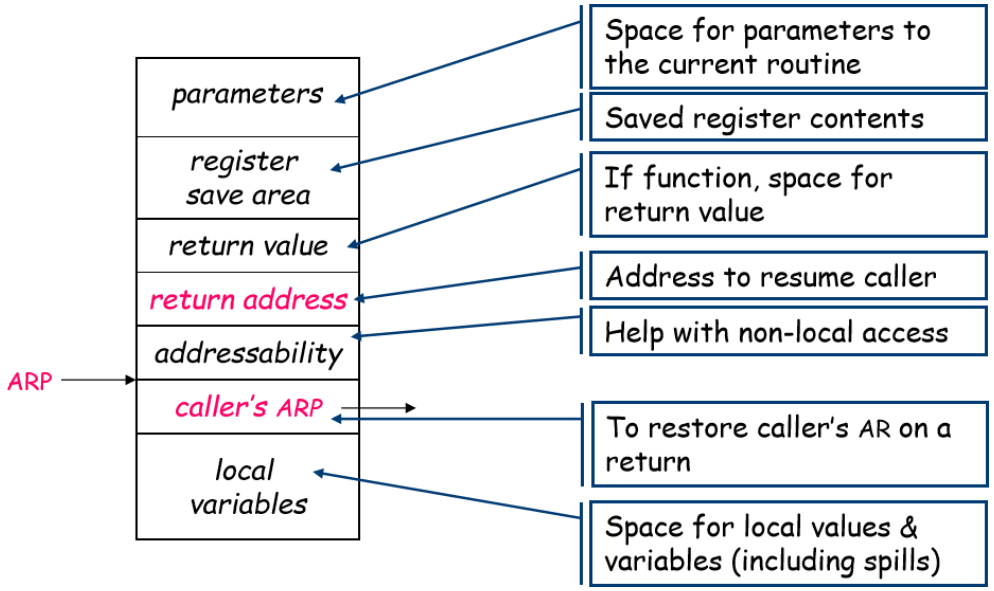
**Array**

- Se la dimensione è fissa in fase di compilazione, l'array viene memorizzato nell'area di dati a lunghezza fissa.
- Se la dimensione è variabile, si memorizza un descrittore nell'area di lunghezza fissa, con un puntatore all'area di dati a lunghezza variabile.

L'area di dati a lunghezza variabile viene assegnata alla fine dell'area di dati a lunghezza fissa per il blocco in cui è allocata (inclusi tutti i blocchi contenuti).

# Introduzione ai record di attivazione (AR)

---




**Come trova il compilatore le variabili?** Le variabili si trovano a offset noti dal puntatore ARP (Activation Record Pointer). La coordinata statica porta a un'operazione `loadAI`. Il livello specifica un ARP, e l'offset è la costante.

## Dati a lunghezza variabile

- Se l'AR può essere esteso, i dati a lunghezza variabile sono posizionati sopra le variabili locali e un puntatore è lasciato a un offset noto dal puntatore ARP.

- Altrimenti, i dati a lunghezza variabile sono posti nell'heap.

### Inizializzazione delle variabili locali

Il compilatore deve generare codice esplicito per memorizzare i valori delle variabili locali, e ciò avviene tra le prime azioni della procedura. Dove vivono i record di attivazione?

- Se la durata dell'AR corrisponde alla durata dell'invocazione e il codice esegue normalmente un "return", i record di attivazione sono mantenuti su uno stack.
- Se una procedura può sopravvivere al proprio chiamante, o se può restituire un oggetto che può fare riferimento al suo stato di esecuzione, gli AR devono essere conservati nell'heap.
- Se una procedura non fa chiamate, l'AR può essere allocato staticamente.

L'efficienza preferisce, in ordine, allocazione statica, stack e heap.

---

---

## Stabilire l'accessibilità delle variabili

---

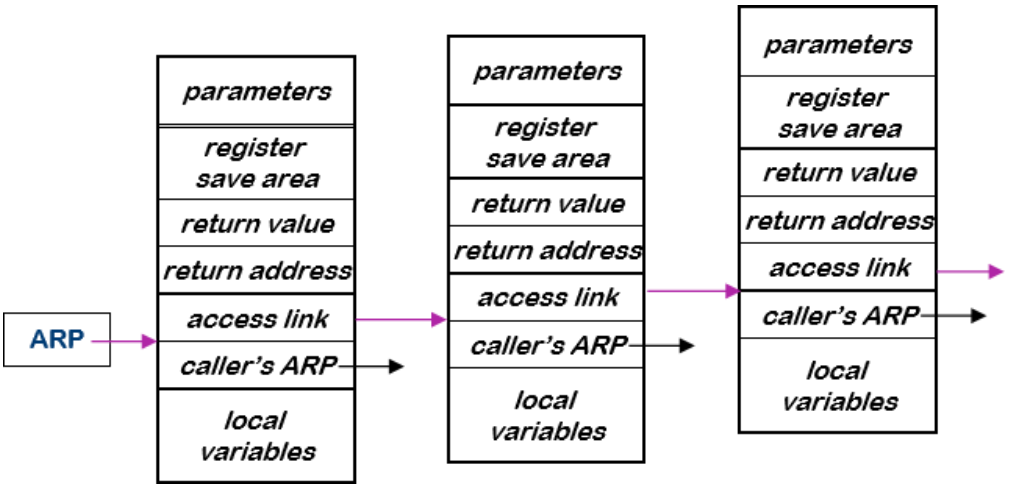
#### Creazione degli indirizzi base

Per garantire l'accessibilità delle variabili, è necessario creare indirizzi base in diversi contesti:

- **Variabili locali:** Vengono convertite in coordinate di dati statici e utilizzano un'ARP (Activation Record Pointer) con un offset appropriato.
- **Variabili globali e statiche:** Si costruiscono etichette attraverso il "mangling" dei nomi, ad esempio, `&_fee`.
- **Variabili locali di altre procedure:** Vengono anch'esse convertite in coordinate statiche. È necessario:
  - Trovare l'ARP appropriato.
  - Accedere all'Activation Record (AR) corretto attraverso link verso AR identificabili.
  - Utilizzare quell'ARP con l'offset richiesto.

#### Uso dei link di accesso per trovare un ARP di una variabile non locale

Ogni Activation Record (AR) contiene un puntatore verso l'AR dell'antenato lessicale. Questo antenato lessicale non deve necessariamente coincidere con il chiamante diretto.




**Meccanismo.**

1. Durante ogni chiamata si configura un certo costo iniziale per impostare il collegamento.
2. Quando si fa riferimento a una variabile `<p,16>`, si risale la catena dei link di accesso (*access link*) fino al record di attivazione della procedura $p$.
3. Il costo dell'accesso è proporzionale alla distanza lessicale tra il chiamante e la variabile.

**Gestione dei Link di Accesso**

Quando una procedura chiama un'altra a un livello immediatamente superiore ($k+1$), il link di accesso è semplicemente il **current ARP** (Activation Record Pointer) della procedura chiamante. Questo perché la procedura chiamata è direttamente annidata all'interno della chiamante.

Se la procedura chiamata si trova a un livello $j \leq k$, il link di accesso deve puntare all'ARP del livello $j-1$. In questo caso, si risale la catena dei link di accesso per individuare il record di attivazione appropriato, e si utilizza quell'ARP come link.

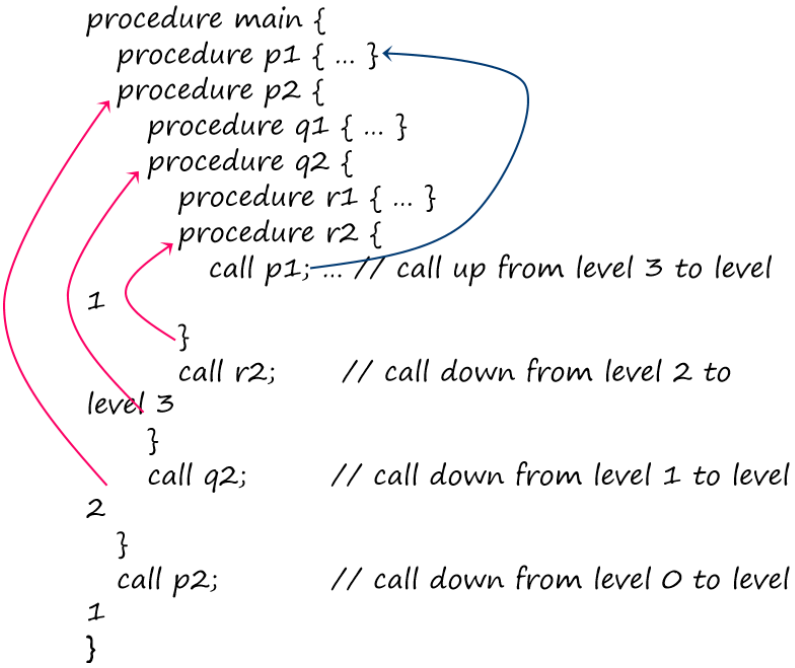
Questo meccanismo garantisce che i link di accesso puntino sempre al contesto lessicale corretto, facilitando il recupero delle variabili non locali.

**Perché Funziona?**

1. Se la chiamata è al livello $k+1$, la procedura chiamata deve essere annidata all'interno della procedura chiamante. In caso contrario, la visibilità non sarebbe garantita.
2. Se la chiamata è al livello $j > k$, la procedura chiamata deve essere annidata nella procedura contenente al livello $j-1$.

Questo approccio consente un'allocazione efficace e sicura delle variabili nei contesti lessicali nidificati.

# Problema: Stabilire l'Accessibilità delle Variabili




Nell'immagine:

- Le frecce rosa rappresentano le chiamate "discendenti" o "ascendenti" tra diversi livelli lessicali.
- Per esempio, la chiamata alla procedura `p1` dal livello 3 richiede di risalire la catena fino al livello 1.
- Con il display, invece di risalire i link, il livello di `p1` è immediatamente accessibile tramite il display, migliorando l'efficienza dell'accesso.

**Utilizzo di un display per trovare un ARP di una variabile non locale**

Un approccio alternativo per gestire l'accessibilità delle variabili non locali è l'uso di un **display**, ossia un array globale di puntatori ai record di attivazione (AR) identificabili.

**Funzionamento del Display**

- Ogni livello lessicale ha una posizione nel display che punta all'ARP corrispondente.
- Per accedere a una variabile non locale, si cerca il record di attivazione (ARP) relativo al livello $p$ nel display e si aggiunge l'offset della variabile, ad esempio `<p,16>`.

**Vantaggi del Display**

1. **Accesso Costante:** Il costo per trovare un'ARP tramite il display è costante, poiché si tratta semplicemente di un accesso array e di un'operazione di somma ($\text{ARP} + \text{offset}$).
2. **Efficienza:** Questo approccio riduce il costo di risalire una catena di link di accesso, rendendo l'accesso alle variabili non locali molto più rapido.

Questo sistema è particolarmente utile nei linguaggi che supportano nidificazioni profonde di procedure.

## Uso di un Display

| SC | Generated Code | |
|---|---|---|
| <2,8> | loadAI $r_0$,8 | $\Rightarrow r_{10}$ |
| <1,12> | loadI _disp | $\Rightarrow r_1$ |
| | loadAI $r_1$,4 | $\Rightarrow r_1$ |
| | loadAI $r_1$,12 | $\Rightarrow r_{10}$ |
| <0,16> | loadI _disp | $\Rightarrow r_1$ |
| | loadAI $r_1$,0 | $\Rightarrow r_1$ |
| | loadAI $r_1$,16 | $\Rightarrow r_{10}$ |

Nell'immagine:

- Il livello lessicale corrente è **2**.
- Il display è memorizzato all'etichetta `_disp`.
- Per `<2,8>`, l'accesso è diretto all'ARP corrente, con un offset di `8`.
- Per `<1,12>`, l'accesso richiede prima di recuperare l'ARP del livello 1 (`loadAI r1,4`), poi di sommare l'offset `12`.
- Per `<0,16>`, il recupero si basa sull'ARP del livello 0 e sull'offset `16`.

Il codice generato segue queste regole per accedere alle variabili non locali utilizzando il display:

1. **Accesso al Display:** Si carica il display globale memorizzato in `_disp` (es. `loadAI _disp`).
2. **Recupero del Puntatore al Record di Attivazione (ARP):**
   - Si accede al livello desiderato tramite l'indice del display.
   - Ad esempio, `loadAI r1,4` accede all'ARP del livello 1.
3. **Accesso alla Variabile:** Una volta ottenuto l'ARP, si somma l'offset associato alla variabile. Per esempio, `loadAI r1,12` accede all'indirizzo della variabile.

**Meccanismo per il Mantenimento del Display.** Per ogni livello lessicale:

1. **All'Entrata del Livello $j$:**
   - Si salva l'ingresso attuale del livello $j$ nel record di attivazione (campo `saved ptr`).
   - Si memorizza l'ARP del livello corrente nello slot corrispondente nel display.
2. **All'Uscita dal Livello $j$:**
   - Si ripristina l'ingresso precedente del livello $j$ dal campo salvato nel record di attivazione.

**Costi di Accesso e Manutenzione**

- **Costi Fissi:** Il costo di accesso e di manutenzione è costante, grazie all'uso del display come array diretto.
- **Uso dei Registri:** L'indirizzo del display potrebbe consumare un registro, il che potrebbe influire sull'efficienza nei casi di disponibilità limitata di registri.

# Access Links Versus Display: Un Confronto

---

## Access Links

**Costi di Overhead:**

- I costi variano in base al livello lessicale della variabile di riferimento.
- L'overhead è sostenuto solo durante le chiamate e i riferimenti alle variabili non locali.
- **Persistenza dei Record di Attivazione (AR):**
  - Gli access links funzionano anche se i record di attivazione sopravvivono alla procedura (ad esempio, per procedure annidate o chiamate successive).
- **Efficienza:**
  - Più efficiente se le variabili non locali vengono utilizzate raramente rispetto al numero di chiamate, poiché l'accesso è limitato a un costo proporzionale alla distanza lessicale.

## Display

- **Costi di Overhead:**
  - I costi sono **fissi** per tutte le referenze, indipendentemente dal livello lessicale della variabile.
  - Ogni chiamata e riferimento deve caricare l'indirizzo del display, che può richiedere un registro per l'archiviazione temporanea (rematerializzazione).
- **Efficienza:**
  - Più vantaggioso quando ci sono numerose referenze a variabili non locali, poiché l'accesso è costante ($\mathrm{ARP} + \mathrm{offset}$).
  - Tuttavia, l'uso di un registro aggiuntivo può influire sulla velocità complessiva, specialmente in sistemi con registri limitati.

**Considerazioni Pratiche**

**Fattori da Valutare:**

- La scelta tra access links e display dipende dal rapporto tra **accessi a variabili non locali** e **chiamate a procedure**.

- Un registro aggiuntivo nel display può migliorare la velocità complessiva, ma potrebbe ridurre la disponibilità di registri per altri calcoli.

**Supporto del Compilatore:**

- Entrambi i meccanismi richiedono che il compilatore inserisca codice specifico in ogni chiamata e ritorno di procedura per gestire i link di accesso o aggiornare il display.

In conclusione:

- Gli access links sono ideali per situazioni in cui gli accessi a variabili non locali sono rari, riducendo i costi delle chiamate.
- Il display è più adatto a scenari con frequenti accessi a variabili non locali, garantendo tempi di accesso costanti.
- La scelta ottimale dipende dal bilanciamento tra overhead e frequenza degli accessi non locali nel programma.

# Creazione e Distruzione dei Record di Attivazione (AR)

---

**Stato nei Record di Attivazione**

Ogni procedura lascia tracce del suo stato nel record di attivazione (AR). Questo stato include:

1. **Indirizzo di ritorno** per tornare al chiamante.
2. **Valori dei parametri** passati alla procedura.
3. **Accesso ad altri ambiti lessicali**, quando necessario.
4. **Dati locali** della procedura, inclusi eventuali spill di registro.

**Creazione di un Record di Attivazione**

La chiamata a una procedura comporta:

1. **Allocazione:** Il chiamante alloca lo spazio necessario per l'AR.
2. **Inizializzazione:** Lo stato del chiamante viene preservato, compreso l'indirizzo di ritorno, i parametri, e le informazioni per accedere agli ambiti lessicali richiesti.

**Distruzione di un Record di Attivazione**

Il ritorno da una procedura richiede:

1. **Smantellamento dell'Ambiente:** Il callee rilascia le risorse allocate per i dati locali, inclusi eventuali registri spillati.

2. **Ripristino dello Stato del Chiamante:** Il chiamante recupera l'indirizzo di ritorno, i parametri e il contesto originale.

**Collaborazione tra Chiamante (caller) e Chiamato (callee)**

Il processo di gestione degli AR richiede una stretta collaborazione tra il chiamante e il chiamato:

**Responsabilità del chiamante.**

- Conoscere l'indirizzo di ritorno, i valori dei parametri e le informazioni necessarie per accedere agli ambiti.
- Allocare lo spazio iniziale per l'AR.

**Responsabilità del chiamato.**

- Conoscere la dimensione dell'area dati locale, incluse le necessità di spill di registro.
- Preparare e gestire i dati locali richiesti durante l'esecuzione della procedura.

**La Convenzione di Collegamento**

Questa collaborazione è implementata attraverso una **linkage convention**, che specifica:

- **Chi** è responsabile di allocare, inizializzare e smantellare il record di attivazione.
- **Come** vengono passati i parametri e restituiti i risultati.
- **Dove** vengono memorizzati i valori intermedi, come i registri salvati e le variabili locali.

La linkage convention garantisce una gestione coerente degli AR e un'interoperabilità affidabile tra caller e callee, indipendentemente dalla complessità delle procedure o del contesto lessicale.

# Collegamenti tra Procedure

---

Le chiamate a procedure sono un meccanismo complesso che garantisce il trasferimento di controllo e dati tra il chiamante e il callee. Tuttavia, alcune caratteristiche del contesto di compilazione rendono la loro implementazione non banale:

- **Incertezza a Tempo di Compilazione:** Il compilatore potrebbe non avere accesso diretto al codice della procedura chiamata (callee), che potrebbe risiedere in un'altra unità di compilazione.
- **Codice Differenziato:** Diverse chiamate potrebbero coinvolgere combinazioni di codice di sistema e codice utente.
- **Protocollo Comune:** Tutte le chiamate devono seguire lo stesso protocollo per garantire il corretto funzionamento, indipendentemente dalla loro origine.

Il compilatore implementa i collegamenti tra procedure attraverso una **sequenza standard di operazioni**, che:

1. **Imposta il Controllo:** Trasferisce il flusso di esecuzione dal chiamante al callee, preservando l'indirizzo di ritorno.
2. **Gestisce i Dati:** Passa parametri dal chiamante al callee e riceve eventuali risultati dal callee al chiamante.
3. **Dividi le Responsabilità:**
    - Il chiamante e il callee sono entrambi coinvolti nella configurazione e gestione del record di attivazione.
    - Ogni parte è responsabile di una porzione del processo, come definito dalla convenzione di collegamento.

Una convenzione di collegamento è un **accordo a livello di sistema** che definisce le regole per:

- **Passaggio dei Parametri:** Specifica dove e come i parametri sono trasferiti (es. tramite registri o stack).
- **Allocazione dello Stack:** Determina chi alloca e dealloca il record di attivazione (caller o callee).
- **Gestione dell'Indirizzo di Ritorno:** Stabilisce dove viene memorizzato l'indirizzo per riprendere l'esecuzione nel chiamante.
- **Interoperabilità:** Consente la cooperazione tra moduli compilati separatamente, indipendentemente dal loro contesto di compilazione.

Un protocollo uniforme garantisce che:

- **Compatibilità tra Moduli:** Chiamante e callee, anche se compilati separatamente o scritti in linguaggi diversi, possano interagire correttamente.
- **Astrazione:** I dettagli di implementazione di una procedura rimangono nascosti al chiamante, rispettando i principi di incapsulamento.
- **Affidabilità:** Il comportamento delle chiamate è prevedibile e coerente a livello di sistema.

In sintesi, la gestione dei collegamenti tra procedure richiede l'adozione di una convenzione standard che sia supportata sia dal compilatore sia dal sistema operativo per garantire la portabilità e l'interoperabilità.

# Salvataggio dei registri: caller o callee?

---

La gestione dei registri durante una chiamata a procedura è una questione cruciale per garantire l'integrità del programma. Chi deve salvare i registri? **Caller** o **callee**? Entrambe le

parti hanno argomenti validi per il proprio coinvolgimento.

**Argomenti a Favore del Salvataggio da Parte del Caller**

- **Conoscenza dei Valori LIVE:**
  Il caller sa quali valori devono rimanere validi dopo la chiamata e può quindi salvare solo ciò che è necessario.
- **Efficienza:**
  Il caller evita di salvare registri non critici, riducendo il sovraccarico.

**Argomenti a Favore del Salvataggio da Parte del Callee**

- **Conoscenza dei Registri Utilizzati:**
  Il callee sa quali registri intende utilizzare e può limitarsi a salvare e ripristinare solo quelli, minimizzando il lavoro.
- **Riduzione del Carico sul Caller:**
  Il callee si assume la responsabilità, semplificando la logica del caller.

**Suddivisione Convenzionale: Tre Tipi di Registri**

1. **Caller-Saved Registers:**
   - Il caller salva questi registri prima di effettuare la chiamata.
   - Sono registri che il callee può utilizzare liberamente senza preoccuparsi di preservarne il valore.
   - Tipicamente contengono valori che non sono **LIVE** attraverso la chiamata.
2. **Callee-Saved Registers:**
   - Il callee salva questi registri se ha bisogno di usarli.
   - Li ripristina prima di tornare al caller.
   - Sono usati solo dopo aver riempito i registri riservati per il salvataggio da parte del caller.
3. **Registri Riservati per la Convenzione di Collegamento:**
   - Registri specifici riservati per scopi come:
     - Il puntatore al record di attivazione (ARP).
     - L'indirizzo di ritorno (se memorizzato in un registro).
   - Questi sono gestiti in base alle regole della convenzione di collegamento.

I registri salvati, sia dal caller che dal callee, vengono memorizzati in uno dei **record di attivazione (AR)**:

- **Caller-Saved:** Possono essere memorizzati nello stack del caller.
- **Callee-Saved:** Vengono salvati nello stack del callee all'inizio della procedura e ripristinati prima del ritorno.

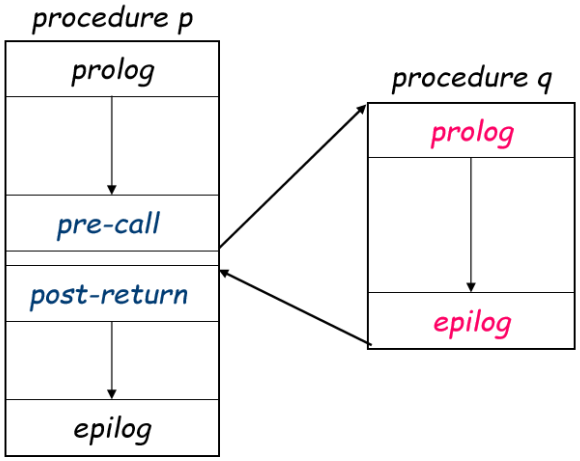
La scelta di chi salva i registri dipende dalla suddivisione tra caller e callee definita dalla **convenzione di collegamento**. Questa suddivisione ottimizza il bilanciamento del carico:

- Il caller salva i valori non LIVE attraverso la chiamata.
- Il callee salva i registri necessari per il proprio utilizzo.
- Registri dedicati come ARP e indirizzi di ritorno seguono regole specifiche della convenzione.
  Questo approccio garantisce un'interazione efficiente e coerente tra le procedure.

# Standard Procedure Linkage

---

Un **collegamento standard** per le procedure segue una sequenza ben definita e prevedibile che coinvolge diverse fasi per gestire il passaggio di controllo e dati tra il chiamante e il callee.




## Pre-call Sequence

**Obiettivo.** Configurare il record di attivazione (AR) del callee e preservare l'ambiente del chiamante.

## Dettagli:

1. **Allocazione AR:** Allocare spazio per il record di attivazione del callee, esclusa l'area per le variabili locali.
2. **Valutazione dei Parametri:**
   - Valutare ogni parametro.
   - Memorizzare il valore o l'indirizzo nel record di attivazione del callee.
3. **Salvataggio dell'Indirizzo di Ritorno:**
   - Memorizzare l'indirizzo di ritorno e l'ARP del chiamante nel record di attivazione del callee.
4. **Access Links (se usati):**

- Identificare l'antenato lessicale appropriato.
- Copiare il puntatore all'interno del record di attivazione del callee.

5. **Registri Caller-Save:**
   - Salvare i registri necessari nello spazio riservato nell'AR del chiamante.
6. **Salto al Callee:**
   - Effettuare un salto all'indirizzo del codice del prologo del callee.

## Post-return Sequence

**Obiettivo.** Ripristinare l'ambiente del chiamante e gestire i valori di ritorno.

## Dettagli:

1. **Valore di Ritorno:**
   - Copiare il valore di ritorno dal record di attivazione del callee, se necessario.
2. **Deallocazione AR:** Liberare lo spazio occupato dal record di attivazione del callee.
3. **Ripristino dei Registri Caller-Save:**
   - Ricaricare i registri salvati dal chiamante.
4. **Parametri Call-by-Reference o Call-by-Value/Result:**
   - Ripristinare i parametri passati per riferimento nei registri, se necessario.
   - Copiare indietro i parametri call-by-value/result.
5. **Continuazione dell'Esecuzione:** Riprendere l'esecuzione del chiamante subito dopo la chiamata.

## Prolog Code

**Obiettivo.** Completare la configurazione dell'ambiente del callee e preservare parti dell'ambiente del chiamante che potrebbero essere alterate.

## Dettagli:

1. **Registri Callee-Save:** Salvare i registri che saranno utilizzati dal callee.
2. **Gestione del Display (se usato):**
   - Salvare l'ingresso del display per il livello lessicale corrente.
   - Memorizzare l'ARP corrente nel display per il livello lessicale corrente.
3. **Allocazione Spazio per Dati Locali:**
   - Allocare spazio per le variabili locali, estendendo il record di attivazione.
4. **Dati Statici e Inizializzazioni:**
   - Individuare eventuali aree di dati statici referenziate dal callee.
   - Inizializzare le variabili locali.

## Epilog Code

**Obiettivo.** Concludere l'esecuzione del callee e iniziare a ripristinare l'ambiente del chiamante.

## Dettagli:

1. **Valore di Ritorno:**
   - Memorizzare il valore di ritorno nel record di attivazione del chiamante (se non già fatto nella dichiarazione `return`).
2. **Ripristino dei Registri Callee-Save:** Ricaricare i registri salvati all'inizio del prologo.
3. **Deallocazione dei Dati Locali:**
   - Liberare spazio per le variabili locali (es. se allocate nell'heap).
4. **Ripristino dell'Ambiente del Chiamante:**
   - Caricare l'indirizzo di ritorno dal record di attivazione.
   - Ripristinare l'ARP del chiamante.
5. **Ritorno al Chiamante:**
   - Effettuare un salto all'indirizzo di ritorno memorizzato.

In conclusione, il collegamento standard suddivide il lavoro tra il caller e il callee:

- Il **pre-call** configura l'ambiente del callee e preserva quello del chiamante.
- Il **post-return** ripristina l'ambiente del caller e gestisce i valori di ritorno.
- Il **prologo** e l'**epilogo** gestiscono la configurazione e la pulizia all'interno del callee.
  Questa struttura garantisce prevedibilità e interoperabilità, semplificando la gestione dei collegamenti tra procedure in un sistema complesso.
  1

# Gestione dei Record di Attivazione: Stack vs Heap

---

# Record di Attivazione sullo Stack

## Vantaggi:

1. **Facilità di Espansione:** L'estensione del record di attivazione è semplice, basta spostare il puntatore superiore dello stack (top of stack pointer).
2. **Collaborazione tra Caller e Callee:** Entrambe le parti condividono la responsabilità di configurare e utilizzare i record di attivazione.

## Dettagli di Implementazione:

- **Caller:**

    - Spinge nello stack:
        - I parametri della procedura chiamata.
        - Lo spazio per i registri salvati.
        - Lo slot per il valore di ritorno.
        - L'indirizzo di ritorno.
        - Le informazioni per l'accessibilità.
        - Il proprio puntatore al record di attivazione (ARP).
- **Callee:**
    - Spinge nello stack:
        - Lo spazio necessario per le variabili locali, sia di dimensione fissa che variabile.

### Limiti:

- Lo stack funziona bene con procedure nidificate e ricorsione, ma potrebbe essere meno efficiente quando si lavora con dati di grandi dimensioni o strutture complesse.

---

# Record di Attivazione nello Heap

### Vantaggi:

1. **Supporto per Allocazioni di Dimensione Variabile:** Ideale per record di attivazione che necessitano di flessibilità dimensionale.
2. **Adatto a Modelli Non Ricorsivi:** Se la ricorsione non è necessaria, i record di attivazione possono essere gestiti staticamente.

### Limiti:

1. **Difficoltà di Estensione:** A differenza dello stack, estendere i record di attivazione nello heap è più complesso e richiede un'allocazione esplicita.
2. **Maggiore Overhead:** Il callee deve allocare spazio esplicitamente e riempire il record di attivazione.

### Opzioni di Implementazione:

- **Passaggio di Tutto nei Registri:**
    - Il caller passa tutti i dati (parametri, indirizzo di ritorno, ecc.) nei registri.
    - Il callee alloca il record di attivazione nello heap e lo riempie con le informazioni ricevute.
- **Uso del Record di Attivazione del Caller:**

    - Il caller memorizza parametri, indirizzo di ritorno, e altre informazioni nel proprio record di attivazione.
    - Il callee fa riferimento a questi dati.
- **Dimensione del Record Predefinita:**
    - Il compilatore utilizza una costante statica per definire la dimensione del record di attivazione del callee.
    - Questo riduce la flessibilità ma semplifica la gestione.

---

# Record di Attivazione Statici

- **Senza Ricorsione:** Se non è presente ricorsione, i record di attivazione possono essere staticamente preallocati.
- **Vantaggi:**
    - Nessun overhead per l'allocazione dinamica.
    - Riduzione della complessità nella gestione della memoria.
- **Limiti:**
    - Non supporta la ricorsione o la concorrenza, poiché i record di attivazione non possono essere duplicati.

---

# Confronto: Stack vs Heap

| Caratteristica | Stack | Heap |
|---|---|---|
| **Facilità di Allocazione** | Molto semplice (movimento del puntatore) | Complessa (allocazione esplicita) |
| **Supporto per Ricorsione** | Sì | Limitato o non supportato |
| **Overhead** | Basso | Alto |
| **Flessibilità Dimensionale** | Limitata | Alta |
| **Efficienza** | Alta per accesso rapido | Minore, dipendente dalla gestione heap |

---

# Conclusione

- **Stack:** Ideale per la maggior parte dei programmi, soprattutto con procedure nidificate e ricorsione.
- **Heap:** Adatto per scenari con requisiti complessi di dimensione o flessibilità, ma con un overhead maggiore.
- **Statici:** Efficienti ma limitati a programmi senza ricorsione o concorrenza.

# Comunicazione tra Procedure

## Meccanismi di Passaggio dei Parametri

La maggior parte dei linguaggi di programmazione offre un meccanismo per il passaggio dei parametri, in cui un'espressione utilizzata nel **punto di chiamata** diventa una variabile nella procedura chiamata (**callee**). I due meccanismi più comuni sono:

---

## 1. Call-by-Reference

- **Descrizione:**
  - Viene passato un **puntatore** al parametro effettivo (l'indirizzo di memoria).
  - La procedura chiamata può modificare direttamente il valore del parametro originale.
- **Dettagli di Implementazione:**
  - Richiede uno slot nel **record di attivazione (AR)** per memorizzare l'indirizzo del parametro.
  - Più nomi possono fare riferimento allo stesso indirizzo di memoria, il che può portare a effetti collaterali inattesi.
- **Vantaggi:**
  - Efficiente per strutture di dati grandi (es. array o oggetti complessi), poiché non è necessario copiare i valori.
- **Svantaggi:**
  - La procedura chiamata può modificare i parametri originali, il che può rendere il comportamento del programma meno prevedibile.

---

## 2. Call-by-Value

- **Descrizione:**
  - Viene passata una **copia del valore** del parametro effettivo al momento della chiamata.
  - La procedura chiamata non può modificare il parametro originale, ma solo la copia.

- **Dettagli di Implementazione:**
  - Richiede uno slot nel **record di attivazione (AR)** per memorizzare la copia del valore.
  - Ogni nome associato a un parametro ha una posizione di memoria unica, anche se inizialmente i valori possono essere gli stessi.
- **Vantaggi:**
  - Previene modifiche ai parametri originali, migliorando l'isolamento tra le procedure.
- **Svantaggi:**
  - Inefficiente per strutture di dati grandi, poiché è necessario copiare il contenuto.

---

# Gestione degli Array

- Gli array sono **tipicamente passati per riferimento**, piuttosto che per valore.
  - Passare un array per valore comporterebbe la copia di tutti gli elementi, il che è inefficiente in termini di memoria e tempo.
- Anche quando passato per riferimento, la procedura chiamata può accedere o modificare direttamente gli elementi dell'array originale.

---

# Uso di Variabili Globali

- Un'alternativa al passaggio dei parametri è l'utilizzo di variabili globali:
  - Le variabili globali sono direttamente accessibili da più procedure.
  - **Vantaggi:** Eliminano la necessità di passare parametri.
  - **Svantaggi:** Possono introdurre dipendenze non esplicite tra procedure, rendendo più difficile il debug e la manutenzione.

---

# Considerazioni Finali

| Caratteristica | Call-by-Reference | Call-by-Value |
|---|---|---|
| **Efficienza** | Alta per strutture grandi | Bassa per strutture grandi |
| **Modifica dei Parametri** | Sì, modifica il parametro originale | No, modifica solo una copia |
| **Isolamento** | Minore, rischio di effetti collaterali | Maggiore, comportamento più prevedibile |

| Caratteristica | Call-by-Reference | Call-by-Value |
|---|---|---|
| **Utilizzo della Memoria** | Meno memoria per strutture grandi | Più memoria per copie |

Entrambi i metodi hanno vantaggi e svantaggi. La scelta dipende dalle esigenze specifiche del programma, come efficienza, sicurezza, e leggibilità del codice.

# Gestione dei Record di Attivazione: Stack vs Heap

## Record di Attivazione sullo Stack

### Vantaggi:

- **Facilità di Espansione:** È sufficiente spostare il **puntatore superiore dello stack** (top of stack pointer) per allocare spazio per nuovi record.
- **Collaborazione tra Caller e Callee:** Entrambi condividono la responsabilità di gestire i record di attivazione:
  - **Caller:**
    - Spinge sullo stack:
      - Parametri per la procedura chiamata.
      - Spazio per registri salvati.
      - Slot per il valore di ritorno.
      - Indirizzo di ritorno.
      - Informazioni di accessibilità.
      - Il proprio puntatore al record di attivazione (ARP).
  - **Callee:**
    - Spinge sullo stack:
      - Spazio per le variabili locali (di dimensione fissa e variabile).

### Limiti:

- Lo stack è meno flessibile per strutture dati di dimensione dinamica o per record di attivazione di grandi dimensioni.
- Non è adatto per ambienti in cui i record di attivazione devono sopravvivere oltre il ritorno della procedura (es. coroutine o closure).

---

## Record di Attivazione nello Heap

### Vantaggi:

- **Flessibilità:** Permette di gestire record di attivazione di dimensione variabile o complessi, non limitati dalla struttura dello stack.
- **Persistenza:** Adatto per situazioni in cui i record devono persistere oltre il ritorno della procedura (es. coroutine).

### Limiti:

- **Difficoltà di Espansione:** Non è semplice estendere un record di attivazione allocato nello heap.
- **Maggiore Overhead:** L'allocazione dinamica è più lenta rispetto allo stack.

### Strategie di Implementazione:

1. **Allocazione Differita al Callee:**
    - Il **caller** passa tutti i dati che può attraverso registri.
    - Il **callee**:
        - Alloca il record di attivazione nello heap.
        - Memorizza i contenuti dei registri nel record.
    - I parametri extra che non possono essere passati nei registri sono memorizzati nel record di attivazione del **caller**.
2. **Allocazione di un'Area Locale per i Dati nello Heap:**
    - Il callee alloca dinamicamente un'area dati locale nello heap per il record.
    - Questo approccio è utile per dati di grandi dimensioni o dimensione variabile.

---

## Record di Attivazione Statici

- Se la ricorsione non è necessaria, i record di attivazione possono essere allocati staticamente.
- **Vantaggi:**
    - È un approccio semplice ed economico, poiché elimina il sovraccarico dell'allocazione dinamica.
- **Limiti:**
    - Non supporta ricorsione o concorrenza.

---

## Confronto: Stack vs Heap

| Caratteristica | Stack | Heap |
|---|---|---|
| **Facilità di Allocazione** | Molto semplice (movimento del puntatore) | Complessa (allocazione esplicita) |
| **Supporto per Ricorsione** | Sì | Limitato o non supportato |
| **Overhead** | Basso | Alto |
| **Flessibilità Dimensionale** | Limitata | Alta |
| **Persistenza dei Record** | No, legata al ciclo di vita della chiamata | Sì, persiste oltre la chiamata |

---

# Conclusione

- **Stack:** Ideale per la maggior parte dei programmi, soprattutto con ricorsione e procedure nidificate.
- **Heap:** Adatto a scenari con strutture di dati dinamiche o necessità di persistenza.
- **Statico:** Utile quando non è richiesta ricorsione o dinamismo, per ridurre al minimo il sovraccarico.