Corso di Linguaggi di Programmazione

Lezione 15

Alberto Ceselli
alberto.ceselli@unimi.it

Dipartimento di Informatica
Università degli Studi di Milano

30 Aprile 2013

Università degli Studi di Milano

Classificazione di un Type System

Riassumendo:

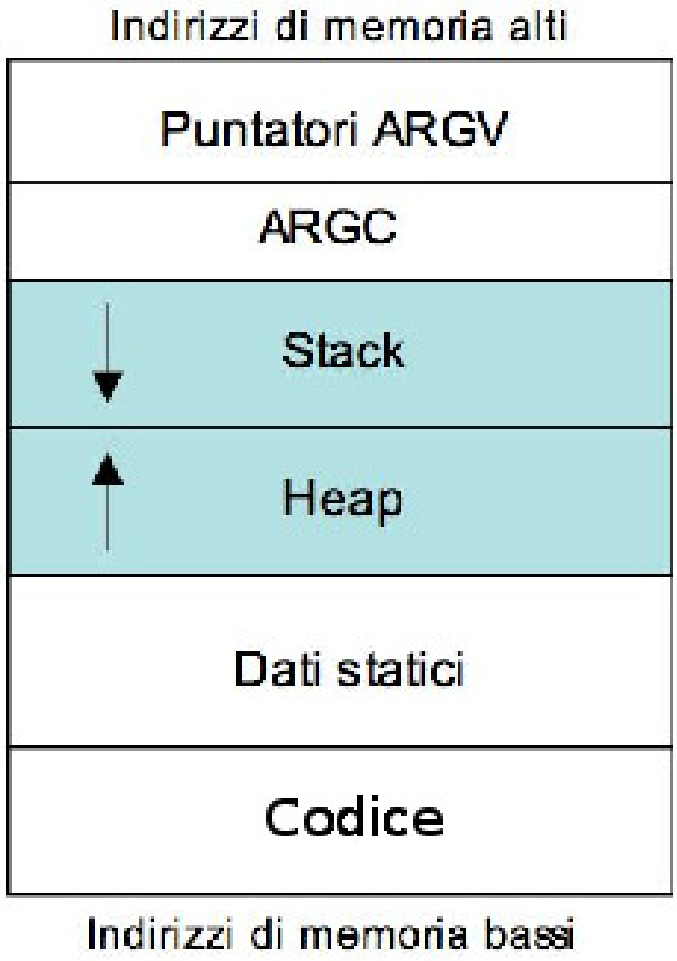
F	statico VS dinamico	il tipo può cambiare *durante* l'esecuzione del programma?
F	esplicito VS implicito	il tipo è assegnato *dal programmatore* tramite opportuni costrutti?
Q	forte VS debole	il *compilatore* è (staticamente) in grado di individuare *tutti* gli errori di tipo?
Q	safe VS unsafe	ho garanzia che *tutti* gli errori di tipo siano catturati (staticamente o a run-time)?

Organizzazione di un processo in memoria (1)

Le aree di memoria associate ad un *programma* sono:

- la **parte codice** che contiene il codice oggetto (in sola lettura);
- la **parte dati** che consiste in:
  - le variabili statiche;
  - l'attivazione dei sottoprogrammi (e.g., funzioni o procedure) organizzata a **stack** (un frame per ciascuna attivazione);
  - le variabili dinamiche organizzate a **heap**;

Organizzazione di un processo in memoria (2)



Variabile di un programma

La **variabile** in un linguaggio imperativo è un'**astrazione** di una o più celle di memoria.
Una variabile è caratterizzata da un insieme di attributi, alcuni dei quali sono essenziali per la conoscenza della semantica dei linguaggi di programmazione (non solo imperativi!).

Attributi di una Variabile (1)

- **Nome**: sequenza di caratteri che identifica una variabile.
  - a seconda del linguaggio di programmazione, permette una *lunghezza massima* o meno (Fortran 90 e ANSI C max 31 caratteri, Ada e Java nessun limite, C++ nessun limite ma gli implementatori ne possono imporre uno);
  - di solito è *case-sensitive* (distingue tra lettere maiuscole e minuscole), quindi permette di avere nomi che appaiono simili con significati diversi (bad?);
  - accetta anche il carattere _;
  - è possibile riutilizzare lo stesso nome in parti diverse del programma per diverse componenti;
  - le parole *speciali* sono considerate parole riservate o parole chiave?

Attributi di una Variabile (2)

- **Indirizzo** di memoria e **valore**
  - il nome di una variabile permette al programmatore di astrarre dall'attuale indirizzo di memoria al quale essa è associata;
  - risolve il problema dell'indirizzamento assoluto;
  - permette *aliasing*, cioè uno stesso oggetto (e uno stesso indirizzo di memoria) con nomi diversi;
  - l'indirizzo rappresenta il *l-value* di una variabile, mentre il valore il *r-value*.
- **Tipo**: definisce l'insieme di valori che la variabile può assumere.
- **Lifetime** (tempo di vita)
- **Scope** (campo di visibilità)

Binding

Definition

Il processo di **binding** associa un attributo ad un'entità, o un operazione ad un simbolo (operatore).
Ci sono diverse classi di binding:

- associazione di nomi di variabile a locazioni di memoria;
- associazione tipo-nome variabile;
- associazione variabile-valore;
- associazione tra parametri formali e argomenti attuali;
- associazione dei parametri formali a locazioni di memoria.

Binding Time

Definition

Il **binding time** è il momento in cui avviene l'associazione tra un'entità del programma ed un suo attributo.
Sono possibili diversi tempi di binding:

1. in fase di progettazione e definizione del linguaggio;
2. in fase di implementazione del linguaggio (quando vengono scritti il compilatore o l'interprete del linguaggio);
3. in fase di compilazione o linking (i.e., staticamente);
4. in fase di loading (quando il programmma viene caricato in memoria);
5. *a run-time (durante l'esecuzione del programma)*.

Esempi di Binding

1. L'operatore + in Fortran, C, Java è legato al suo significato nella **definizione** del linguaggio.
2. Il tipo `int` in C è legato al suo significato nell'**implementazione** del linguaggio.
3. Il binding di una variabile locale al suo tipo avviene in **compilazione**.
3. Il binding di una chiamata di funzione al codice corrispondente, in C avviene in fase di **link**.
4. Il binding di una variabile globale all'indirizzo in memoria avviene in **caricamento**.
5. Il binding di una chiamata di funzione *virtual* al codice corrispondente, nei linguaggi ad oggetti avviene durante l'**esecuzione**.

Esempi di Binding

3,4 Il binding di chiamata al codice e di variabile globale all'indirizzo in memoria in C

```
...
int status; // binding in fase di loading
...

int fact(int n) {
    ...
}

...

int main(int argc, char ** argv) {
    ...
    fact(3); // binding in fase di linking
    ....
}
```

Esempi di Binding

5 Il binding di una chiamata di funzione *virtual* al codice in un linguaggio ad oggetti

```
public class Father {

//virtual
public void display() {
...
}
...
}

public class Son: extends Father {

//virtual
public void display() {
...
}
...
}
```

Esempi di Binding

5 Il binding di una chiamata di funzione *virtual* al codice in un linguaggio ad oggetti

```
int main() {
Father * v;

...
v->display(); // binding a runtime
...

}
```

Tipo di Binding

- **Statico** se avviene prima dell'esecuzione e resta immutato per tutta l'esecuzione del programma.
- **Dinamico** se avviene per la prima volta durante l'esecuzione e può essere cambiato nel corso dell'*esecuzione* del programma.
  - Esempio: in C il *tipo* di una variabile è univocamente (esplicitamente) definito nella dichiarazione e immutabile, mentre in Perl è possibile cambiare il tipo delle variabili a run-time (e definirlo in modo implicito).
  - Esempio precedente: Il binding di chiamate di funzione a codice in C è deciso in fase di link, mentre è deciso a runtime per funzioni *virtual* nei linguaggi ad oggetti (es. Java)
- Il binding dinamico di chiamate a codice nei linguaggi che offrono solo il binding statico può essere *emulato* utilizzando puntatori a sottoprogrammi.

Binding Time e Tipo di Binding

- Perché avere così tante possibilità di scelta?
  - Il trade-off è tra *efficienza* e *flessibilità*.
  - Se si vuole un linguaggio *efficiente*, allora il più possibile viene fatto in fase di *compilazione*. Se si vuole un linguaggio *flessibile*, viene lasciato il più possibile in fase di *esecuzione*.
- Svantaggi del binding dinamico:
  - l'abilità nel trovare gli errori del compilatore è diminuita (no errori di tipo in fase di compilazione);
  - costo in fase di compilazione perchè necessarie strutture dati aggiuntive;
  - costo in fase di esecuzione per fare type-checking (attraente utilizzare un interprete);

Storage Bindings

L'associazione *variabile-cella di memoria*, detta comunemente fase di **allocazione** in memoria della variabile.
Classificazione delle variabili a seconda del tempo di allocazione in memoria:

- **Statiche** (dette anche permanenti o globali).
- **Stack-dynamic** (dette anche automatiche o locali).
- **Heap-dynamic** (dette anche dinamiche), con un ulteriore classificazione in *implicite* o *esplicite* a seconda che vengano allocate a run-time usando comandi espliciti (e.g., `new` in C++) o implicitamente quando viene loro assegnato un valore.

Variabili Statiche

- Sono associate alla stessa cella di memoria da prima che inizi l'esecuzione del programma fino alla fine.
- Es. in C, C++ specificate all'interno di una procedura tramite l'identificatore `static`.
- Vantaggi:
  - efficienti perché permettono indirizzamento diretto;
  - efficienti perché non c'è overhead per l'allocazione/deallocazione.
  - conservano valore nelle varie esecuzioni della stessa procedura (history-sensitive);
- Svantaggi: se ho *solo* var statiche
  - diminuisce la flessibilità.
  - (es. non cè supporto per funzioni ricorsive);

Variabili Automatiche (stack dynamic)

- Vengono allocate (nello *stack*) quando a run-time viene elaborata la loro dichiarazione (mentre il binding con gli altri attributi avviene staticamente)
  - Tipo più comune di variabili nei linguaggi imperativi.
  - In C, C++, Java la dichiarazione di una variabile può apparire in qualsiasi punto di un blocco di istruzioni, ma l'allocazione avviene all'ingresso del blocco per motivi di efficienza.
  - Lo scope di solito è costituito da una funzione, ma in molti linguaggi (es. C) anche granularità più fine (**blocco**).
- Vantaggi:
  - allocazione/deallocazione hanno un basso overhead;
  - indirizzamento indiretto ma efficiente;.
  - Permettono l'implementazione di funzioni ricorsive.

Variabili Dinamiche (heap dynamic)

- Vengono allocate (nello *heap*) quando il programmatore lo richiede, tramite operatori (`new` in C++) o funzioni (`malloc` in C).
  - cella di memoria senza nome indirizzata da un puntatore o da una variabile di riferimento.
- Possono essere deallocate quando il programmatore lo richiede tramite operatori (`delete` in C++) o funzioni (`free` in C), oppure automaticamente (in caso di **garbage collection** come in Java).
- Alcuni linguaggi prevedono anche l'*allocazione* implicita: es Javascript `vals = [23, 75, 21, 55, 12];`

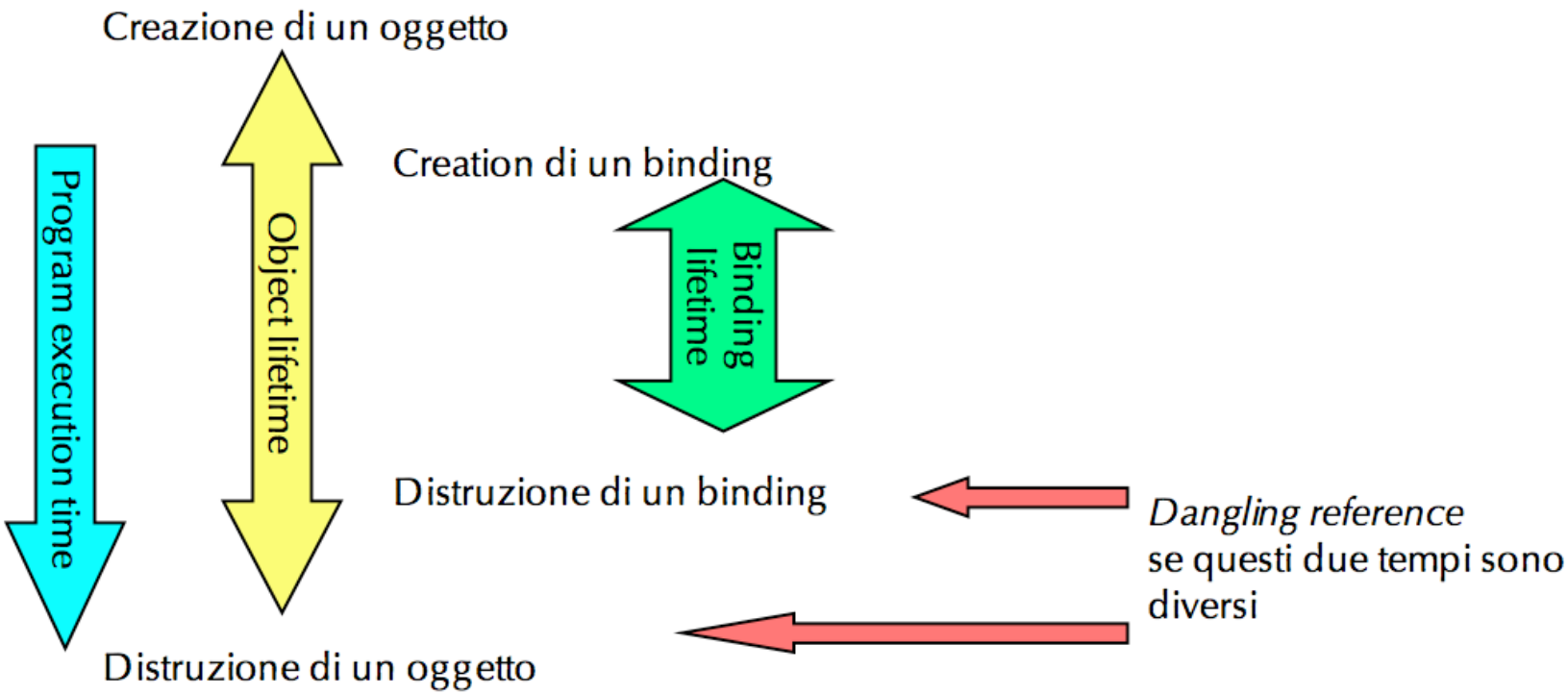
Variabili Dinamiche (heap dynamic)

- Vantaggi:
  - Permettono di implementare strutture dati dinamiche.
  - permettono di avere vettori di dimensione variabile.
- Svantaggi:
  - Accesso inefficiente
  - Allocazione/deallocazione costose (system call).
  - Overhead molto alto per allocazioni / deallocazioni implicite.

Lifetime

Definition

Il **tempo di vita** di una variabile è il tempo che intercorre tra l'allocazione e la deallocazione di una variabile.



Function-Call Frame - Record di Attivazione

Function-Call Frame - Record di Attivazione

- Indirizzo di Ritorno: indirizzo dell'istruzione chiamante
- Link dinamici: puntatore al top del frame (allocazioni stack-dynamic durante l'esecuzione)
- Parametri: argomenti in ingresso e valori restituiti
- Variabili locali: variabili stack-dynamic

Scoping

Definition

Il **campo di visibilità** di una variabile è la parte di codice in cui si può accedere ad essa (in cui la variabile è **visibile**).

- **Statico**: può venire determinato staticamente in fase di compilazione.
  - utilizzato originariamente da Algol 60 e poi in seguito copiato da quasi tutti i linguaggi imperativi.
- **Dinamico**: viene determinato solo in fase di esecuzione.

Scoping Statico

- Gli scope sono *nested* come scatole cinesi:
  - in Ada, Pascal: uno scope per ogni funzione, che a sua volta può contenerne delle altre;
  - in Algol 60, C, C++, Java: uno scope per ogni blocco, che a sua volta può contenerne altri.
- **Funzionamento:** (in fase di riferimento ad una variabile) la variabile viene cercata nello scope locale, poi in quello che lo contiene e così via, sino a trovarne la dichiarazione o a determinare che non è dichiarata.
  - un identificatore ne maschera localmente uno uguale dichiarato in uno scope più esterno;
  - spesso disponibili meccanismi per accedere a identificatori temporaneamente mascherati.
- Svantaggi:
  - In caso di nested scope, variabili necessarie in punti diversi ma non dappertutto devono essere definite globali.

Scoping Dinamico

- Una dichiarazione resta valida sino a quando viene trovata una diversa dichiarazione per lo stesso identificatore.
  - È determinabile solo a run-time.
- **Funzionamento:** un identificatore viene cercato nello scope locale, poi in quello della procedura chiamante e così via , sino a trovarne la dichiarazione o a determinare che non è dichiarato.
- Svantaggi:
  - rende i programmi di difficile leggibilità, quindi è più facile commettere errori;
  - il controllo dei tipi deve per forza essere dinamico;
  - poco efficiente (quindi poco usato).

Referencing environment

Definition

Il **referencing environment** è l'insieme di tutti gli identificatori che sono visibili in un punto del programma.

- Nel caso di linguaggio a *scoping statico*: tutte le variabili definite localmente, più tutte quelle del blocco contenente e così via.
- Nel caso di linguaggio a *scoping dinamico*: tutte le variabili definite localmente, più tutte quelle delle funzioni *attive* (i.e., non ancora terminate).

Passaggio dei parametri (1)

Classi di parametri:

- **Parametri Formali** nella definizione della procedura.
  - nomi di variabili
- **Parametri (o argomenti) Attuali** nella chiamata di una procedura.
  - nomi di variabili o espressioni

Concettualmente esistono 3 modi di passaggio degli argomenti:

- valori che passano dal chiamante alla procedura (modo **In**);
- valori che passano dalla procedura al chiamante (modo **Out**);
- valori che passano dal chiamante alla procedura e ritornano al chiamante modificati (modo **InOut**).

Passaggio dei parametri (2)

- Esistono 5 possibili implementazioni del passaggio di argomenti, che rappresentano la modalità di legame tra argomenti attuali e parametri formali:
  - per *valore*;
  - per *risultato*;
  - per *valore e risultato* (o passaggio per copia).
  - per *riferimento*;
  - per *nome*.
- Con uno qualsiasi di questi metodi, tranne il passaggio per risultato, si possono simulare gli altri (ad esempio usando puntatori).

Passaggio dei Parametri per Valore

- Il valore dell'argomento attuale viene calcolato e copiato nel frame della funzione chiamata, dove equivale ad una variabile locale.
  - nessuna azione della procedura chiamata ha effetti sulle variabili della funzione chiamante (modo **In**);
  - meccanismo semplice, di default in molti linguaggi (Pascal, C, Java, ..);
  - parametro attuale e formale devono avere tipi compatibili;
  - ambiente locale esteso con il binding tra il parametro formale e una nuova variabile che viene inizializzata con il valore dell'argomento attuale.
- Costo:
  - per strutture dati molto grandi, copia dispendiosa;
  - accesso a costo minimo (come le altre variabili dell'ambiente locale)

Passaggio dei Parametri per Risultato

- Una variabile del tipo corretto viene allocata nel frame della procedura chiamata.
  - il valore viene trasferito alla procedura chiamante al momento dell'uscita tramite il parametro attuale che deve essere una variabile (modo **Out**);
  - il valore iniziale del parametro è indefinito, la procedura chiamata assegna solo il valore finale;
  - parametro attuale e formale devono avere tipi compatibili.

Passaggio dei Parametri per Valore e Risultato

- Il valore dell'argomento attuale viene calcolato e copiato nel frame della funzione chiamata, dove equivale ad una variabile locale. Al momento dell'uscita, il valore viene passato alla procedura chiamante.
  - implementa direttamente il modo **InOut**;
  - parametro attuale e formale devono avere tipi compatibili;
  - introdotto nel 1966 dall'Algol W.

Passaggio dei Parametri per Riferimento

- Un riferimento a una variabile viene creato nel frame della procedura chiamata.
  - ogni azione della procedura chiamata sul riferimento ha effetto sulla variabile passata come argomento;
  - implementa direttamente il modo **InOut**;
  - parametro attuale e formale devono avere tipi compatibili;
  - all'uscita distruzione del legame, ma NON della variabile.
- Costo: abbastanza basso sia per la memorizzazione (dell'indirizzo e non dell'intera variabile), sia per l'accesso (costo di un indirizzamento indiretto).

Passaggio dei Parametri per Nome

- Il parametro attuale viene *testualmente sostituito* nel corpo della procedura chiamata a quello formale senza essere valutato.
  - implementa il modo **In**;
  - viene temporaneamente ripristinato l'ambiente del chiamante.
- Poco utilizzato perché l'implementazione è difficile e inefficiente.
  - introdotto dall'Algol 60 è stato adottato solo dal Simula 67 e poi caduto in disuso.
- In Pascal il passaggio di default è *per valore*, ma utilizzando la parola chiave `var` si chiede il passaggio *per riferimento*.

Passaggio dei Parametri
Metodi utilizzati dai principali linguaggi

- In C il passaggio è solo *per valore*
  - il passaggio per riferimento può essere ottenuto tramite l'utilizzo di puntatori (in C++ anche di riferimenti);
- In Java il passaggio di argomenti di tipo semplice è solo *per valore*, mentre gli oggetti sono accessibili solo tramite *riferimenti*.

Passaggio dei Parametri
Esempio (1)

```
void swap1(int a, int b) {
int tmp = a;
a = b;
b = temp;
}
```

- Con chiamata `swap1(c, d);`

Passaggio dei Parametri
Esempio (2)

```
void swap2(int *a, int *b) {
int tmp = *a;
*a = *b;
*b = temp;
}
```

- Con chiamata `swap1(&c, &d);`

Passaggio dei Parametri
Esempio (3)

```
void swap3(int &a, int &b) {
int tmp = a;
a = b;
b = temp;
}
```

- C++, con chiamata `swap1(c, d);`

Esercizi per casa (1)

1 Si consideri il seguente programma C:

```
void scambia(int a, int b) {
   int tmp;
   tmp = a;
   a = b;
   b = tmp;
}
void main() {
   int num1, num2;
   scanf("%d", &num1);
   scanf("%d", &num2);
   scambia(num1, num2);
   printf("num1 = %d, num2 = %d", num1, num2);
}
```

(1) Quali sono i valori stampati dalla `printf` per le variabili `num1` e `num2` nell'ipotesi che i valori acquisiti da tastiera ed assegnati all'inizio del programma a queste due variabili siano rispettivamente 15 e 50?

Continua ...

Esercizi per casa (2)

...1

(2) Che modello semantico di passaggio dei parametri è utilizzato? Supponendo che l'obiettivo del sottoprogramma `scambia` sia quello di scambiare il valore delle variabili `a` e `b`, il modello semantico utilizzato è corretto? In caso negativo, quale sarebbe stato il modello corretto?

(3) In caso di risposta negativa alla domanda del punto precedente, quale sarebbe un'implementazione corretta in C? E in Ada (è sufficiente indicare la sola signature del sottoprogramma)?

Esercizi per casa (3)

2 Si implementi un sottoprogramma in C che riceve come parametro le informazioni relative ad un triangolo, ne calcola l'area e restituisce tale valore al chiamante:

1. utilizzando il meccanismo del valore di ritorno.
2. utilizzando il passaggio di argomento per riferimento.

Si supponga che le informazioni relative ai rettangoli nel programma vengano rappresentate utilizzando il seguente tipo `struct`:

```
typedef struct {
   float base;
   float altezza;
} Triangolo;
```