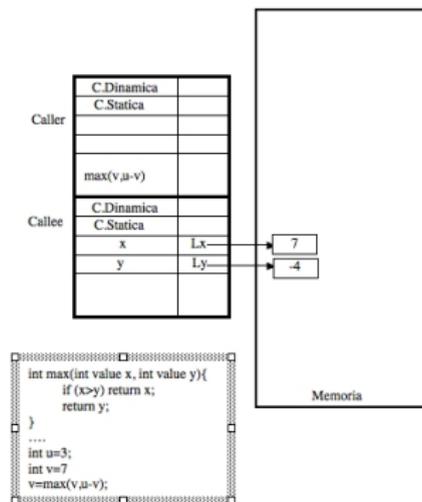


Sommario: 28 marzo, 2018

- Astrazioni: Procedure, Funzioni e Unità di Programmazione
- Trasmissione per Valore, Costante
- Trasmissione per Riferimento, Risultato, Valore-Risultato
- Trasmissione per Nome
- Trasmissione e Ritorno di funzioni come valori
- Il meccanismo delle eccezioni
- Esercizi

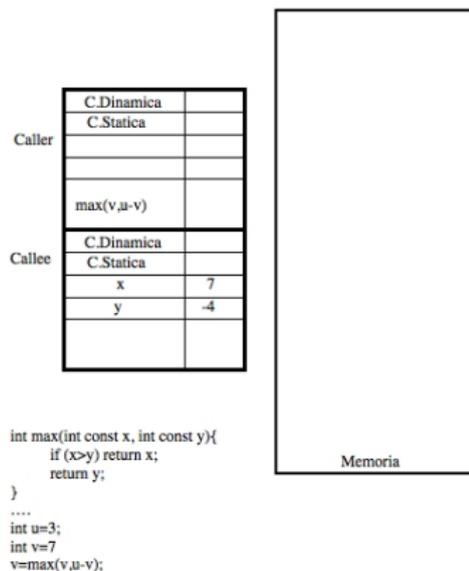
Trasmissione per Valore

- I parametri **attuali** sono (completamente) valutati, nel chiamante, come espressioni che devono calcolare un v. memorizzabile.
- I parametri **formali** hanno come denotazione valori modificabili inizializzati al valore del corrispondente parametro attuale
- Trasmissione one-way: Invocato riceve solo valori per i parametri
- Presente in tutti i linguaggi Imperativi/a stato. Assente in linguaggi dichiarativi.



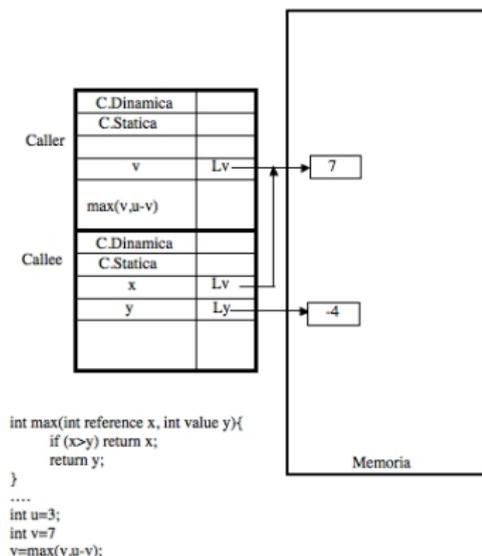
Trasmissione per Costante

- I p. **attuali** sono (completamente) valutati, nel chiamante, come espressioni che devono calcolare denotazioni di v . non modificabili, ovvero costanti
- I parametri **formali** hanno come denotazione tali valori
- Presente in tutti i linguaggi dichiarativi. Nei Ling. Imperativi Pascal, ADA.
- Trasmissione one-way: Invocato riceve solo valori per i parametri



Trasmissione per Reference

- I p. **attuali** sono (completamente) valutati, nel chiamante, come espressioni che devono calcolare denotazioni di v. modificabili.
- I parametri **formali** hanno come denotazione tali valori
- Presente, o emulabile, in tutti i linguaggi Imperativi
- Trasmissione a memoria condivisa: Chiamante e Invocato condividono una regione di memoria



Trasmissione per Risultato e Valore-Risultato

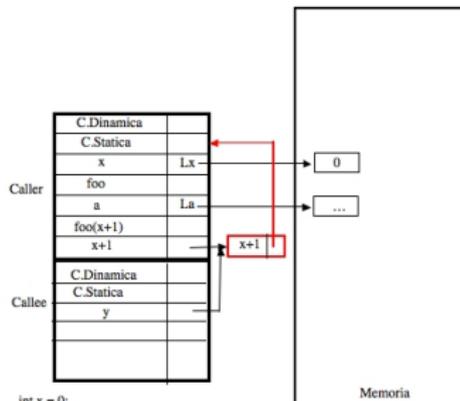
- **Risultato.** Come il reference ma il parametro **formale** è in sola scrittura. Quindi one way in scrittura.
- **Valore-Risultato.** Come il reference ma il parametro **formale** è in lettura locale durante la valutazione del chiamato e in scrittura quando il chiamante termina. Quindi è un two-way ma senza condivisione di memoria.
- Talvolta il Valore-Risultato può essere emulato con la trasmissione per Riferimento, ma non sempre come si vede sotto

```
int foo(int reference x,  
        int reference y,  
        int reference z){  
    y = 2;  
    x = 4;  
    if (x==y) z=1;  
}  
....  
int a=3;  
int b=0;  
foo(a,a,b);  
/* qui a vale 4, b vale 1 */
```

```
int foo(int value-result x,  
        int value-result y,  
        int value-result z){  
    y = 2;  
    x = 4;  
    if (x==y) z=1;  
}  
....  
int a=3;  
int b=0;  
foo(a,a,b);  
/* qui a vale 4, b vale 0 */
```

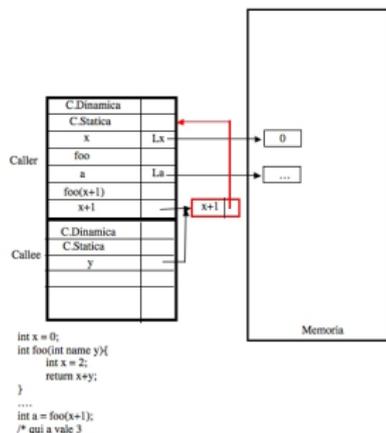
Trasmissione per Nome: Creazioni di Chiusure

- I p. **attuali** sono un codice (espressione o comando). Il codice è chiuso con il suo ambiente in una struttura che si chiama *chiusura*.
 - **Chiusura** Valore/Struttura formato da una coppia (codice,ambiente).
 - È utilizzata per chiudere un codice con i legami per i non locali che occorrono in esso.
- I p. **formali** hanno come denotazione chiusure
- Quando un formale è valutato, il codice della chiusura è valutato nell'ambiente della chiusura



```
int x = 0;
int foo(int name y){
  int x = 2;
  return x+y;
}
....
int a = foo(x+1);
/* qui a vale 3
```

Trasmissione per Nome, oggi



- Presente in Algol60, Simula e nei Linguaggi Funzionali non ML e derivati (incluso OCaml), per esprimere funzioni parzialmente non strette. Presente anche in varianti, quali:
 - **by need** (Linguaggi Miranda) Il formale è valutato solo la prima volta e il valore ottenuto rimpiazza la chiusura (solo per codice espressione).
 - **lazy** (Linguaggi Haskell) Ad ogni valutazione il codice è rimpiazzato dalla valutazione parziale ottenuta (solo per codice espressione che definisce valori strutturati)
 - **procedure/function** In quasi tutti i linguaggi, Algol68, Pascal, Lisp, Modula, ADA, Miranda, ML, Ocaml, C#, Java. Prossime slides.

Trasmissione di Funzione (o Procedura)

- Il p. **attuale** è un'espressione che calcola un identificatore di Funzione.
- Il p. **formale** ha come denotazione la chiusura ottenuta in accordo al *Binding del Linguaggio*:

```
{ int x = 1;
  int y = 7;
  int f(){
    return x+y;
  }
  void g (int h){
    int x = 2;
    return h() + x;
  }
  ...
  { int x = 4;
    int z = g(f);
  }
  ...
}
```

Binding del Linguaggio. L'identificatore calcolato dal parametro attuale è:

- **Deep Binding.** Valutato nel chiamante, individuando subito, la Funzione definita e il suo ambiente non locale che sono inseriti nella chiusura.
- **Shallow Binding.** Inserito in una chiusura che sarà completata con l'ambiente non locale individuato se e quando un'invocazione del p. formale è effettivamente richiesta.

Trasmissione di Funzione (o Procedura -2)

- Il p. **attuale** è un'espressione che calcola un identificatore di Funzione.
- Il p. **formale** ha come denotazione la chiusura ottenuta in accordo al Binding del Linguaggio:

```
{ int x = 1;
  int y = 7;
  int f(){
    return x+y;
  }
  void g (int h){
    int x = 2;
    return h() + x;
  }
  ...
  { int x = 4;
    int z = g(f);
  }
  ...
}
```

- Quando il p. formale è invocato, la Funzione invocata dipende dal Binding:
 - **Deep Binding.** Funzione e Ambiente non locale sono già nella chiusura.
 - **Shallow Binding.** Funzione e Ambiente non locale sono ottenuti valutando nel corrente ambiente, l'identificatore presente nella chiusura: I componenti della chiusura sono rimpiazzati da Funzione e Ambiente non locale ottenuti, e utilizzati nelle successive invocazioni.

Higher Order: Funzione come Valori Calcolati

- **Higher Order:** Le funzioni sono valori che possono essere argomenti o valori calcolati di un invocazione di funzione
- Chi sono le non locali di una funzione ottenuta come risultato di un invocazione?

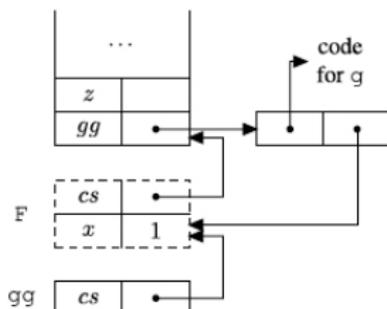
```
{int x = 1,  
void->int F () {  
    int g () {  
        return x+1,  
    }  
    return g,  
}  
void->int gg = F(),  
int z = gg(),  
}
```

- gg è solo un nome per la funzione g definita nel corpo di F
- In un Linguaggio con Scope Statico:
 - gg ha come valore denotabile una chiusura contenente tale funzione g e il suo ambiente dei non locali
- In un Linguaggio con Scope Dinamico:
 - gg ha come valore denotabile una chiusura contenente tale funzione g ma l'ambiente dei non locali è determinato dagli ambienti nei quali gg è invocata

Higher Order: Funzione come Valori Calcolati /2

- In un Linguaggio con Scope Statico, l'ambiente non locale di una funzione calcolata g potrebbe non essere più accessibile dopo l'invocazione della funzione che l'ha calcolata.

```
void->int F () {  
    int x = 1;  
    int g () {  
        return x+1;  
    }  
    return g;  
}  
void->int gg = F();  
int z = gg();
```



- AR Retention** Retention di un Activation Record

Astrazioni di Controllo: Eccezioni

- Eccezione = Eventi che si verificano durante l'esecuzione del programma e che non devono essere gestiti con il normale flusso di controllo.
- Possono essere generate dal sistema (ad es. Heap/stack in segmentation fault, divisione per 0,...).
- Possono essere generate dall'utente (ad es. uso di interi negativi in invocazione di fattoriale).
- I linguaggi recenti forniscono meccanismi per programmare con situazioni di eccezione, prevedendo meccanismi per la definizione di:
 - Eccezioni che il programma intende gestire
 - Condizioni che nei costrutti usati nel programma possono interrompere la normale esecuzione e sollevare un'eccezione.
 - Gestione che deve essere avviata per riportare il programma in uno stato da dove riavviare la normale esecuzione

Astrazioni di Controllo: Eccezioni - Un esempio

```
int fact(int n){//it throws an exception when n<0
  if (n<0) throw Illegal_Argument("fact",n);
  if (n == 0) return 1;
  return n * fact(n-1);
}
int main (int argc, char *argv[]){
  //maximum computable factorial on the current
  //platform of a C-like language with exception mechanism.
  int num = 1;
  int factNum = 1;
  try{
    while (true) factNum = fact(num++);
  }
  catch(Integer_Overflow){
    printf("The maximum signed integer is %d. Then:\n",maxInteger);
    printf("The maximum computable factorial is %2d!=%d\n",num-1,factNum);
  }
}
```

Esercizio: T. per Nome e L. Estendibili con Nuovi Costrutti

- Si supponga di disporre di un linguaggio C^N ottenuto estendendo C con trasmissione per Nome.
 - È possibile in C^N sostituire ogni occorrenza di "`while E do C;`", per qualsivoglia espressione E e qualsivoglia comando C , con l'invocazione di un'opportuna procedura avente comportamento equivalente?
 - Si mostri la definizione C^N di una tale procedura
 - Si confrontino gli stack di AR ottenuti in un semplice caso concreto scelto opportunamente
 - Si discutano i risultati di tale confronto.

Esercizio: Confronto tra T. di Funzione e T. per Nome

```
{ int x = 1;
  int y = 7;
  int f(){
    return x+y;
  }
  void g (int h){
    int x = 2;
    return h() + x;
  }
  ...
  { int x = 4;
    int z = g(f);
  }
  ...
}
```

- Ha lo stesso comportamento che avremmo avuto utilizzando by name e sostituendo come sotto?
 - **int** h() con **int name h** nel formale di g.
 - f con x+y nell'attuale di g.
- La risposta dipende dal Linguaggio usato, in particolare da:
 - meccanismo di Scope degli identificatori.
 - meccanismo di Binding della trasmissione.

Esercizio: T. Funzione e I non locali

- Il programma precedente in un linguaggio con Scope statico avrebbe avuto stesso comportamento con entrambi i tipi di Binding. Ma consideriamo il programma sotto in due linguaggi con Scope statico ma:
 - (a) Deep Binding,
 - (b) Shallow Binding
- Osserva. Ad ogni invocazione ricorsiva `foo` introduce una nuova funzione `fie`.

```
{void foo (int f(), int x){
    int fie(){
        return x;
    }
    int z;
    if (x==0) z=f();
    else foo(fie,0);
}
int g(){
    return 1;
}
foo(g,1);
}
```