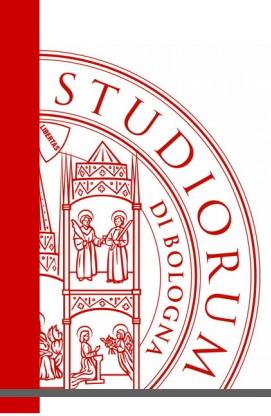


Paradigmi Emergenti di Programmazione:

Le Chiusure

Oggetti di prima classe

Un “oggetto” è di PRIMA CLASSE se può essere usato come un dato qualunque:

- Preso in input / dato in output da una funzione
- Memorizzato in variabili globali/locali
- …

Funzioni (al top-level)

- Chiamiamo FUNZIONI le funzioni definite al top-level
- Le uniche variabili in scope in una funzione sono i parametri della funzione, le variabili locali definite nel corpo della funzione e quelle globali
- Parametri e variabili locali sono allocate nello stack frame della chiamata di funzione e non sopravvivono la chiamata; le variabili globali sopravvivono fino alla fine del programma
- Il codice macchina della funzione risiede in RAM a partire da un indirizzo chiamato entrypoint; il valore di un puntatore a funzione è l'entrypoint della funzione puntata

Funzioni in C

- Il C ammette solo funzioni al toplevel
- I puntatori a funzione sono oggetti di prima classe

```
void qsort(void *base, size_t nitems, size_t size,
           int (*compar)(const void *, const void*));


int comp (const void * elem1, const void * elem2)
{
    int f = *((int*)elem1);
    int s = *((int*)elem2);
    return (f > s ? 1 : f < s ? -1 : 0);
}

int main(int argc, char* argv[])
{
    int x[] = {4,5,2,3,1,0,9,8,6,7};
    int (*mycomp)(const void*, const void*) = comp;
    int res = mycomp(x,x+2);
    qsort (x, sizeof(x)/sizeof(*x), sizeof(*x), mycomp);
}
```

Chiusure

- Una CHIUSURA è una funzione di prima classe definita all'interno di uno o più blocchi
- Il codice di una chiusura può far riferimento a variabili locali definite nei blocchi che la circondano

Il C non ha chiusure

Allocazione sullo stack + funzioni annidate = BOOM!

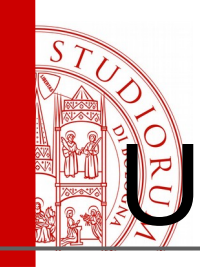
```
int (*mk_closure(int c))(int) {
        int closure(int x) {
                return (c+x);
        }
        return closure;
}

int main(int argc, char* argv[]) {
        int (*closure)(int) = mk_closure(3); // c è deallocata dallo stack!
        return (closure(2)); // invece di tornare 5  accede a c
                             // (dandling stack pointer)
}
```

Il Pascal non ha chiusure

- Il Pascal ha funzioni annidate
- Il Pascal non ha puntatori a funzione
- Durante l'esecuzione della funzione annidata lo stack frame esterno non è ancora stato deallocato

```
function E(x: integer): integer;
    function F(y: integer): integer;
    begin
        if y = 0 then F := x else F := F(y -1)
    end;
begin
    E := F(3)
end;
```

- Come compilereste F in modo da recuperare l'indirizzo di x nello stack? (hint: googlate "catena statica vs catena dinamica pascal")



Un concetto emerso: le chiusure

- Chiusure: nome introdotto da Landin nel 1964 (!!), usate per l'implementazione efficiente del lambda-calcolo
- Costrutto fondamentale della programmazione funzionale: PAL (1970), ML (early '70s), Scheme (1975), Miranda (1985), Erlang (1986), Haskell (1987), ...
- Senza chiusure: C (1969), Pascal (1970), C++(1979-2011?), Java (1995-2014), C# (1999-2007), ...
- In tutti i linguaggi moderni: Scala, Python, Go, Rust, JavaScript, Java (>=2014), ...

Chiusure: implementazione

- Problema 1: le variabili usate da una chiusura devono non essere deallocate fino a quando la chiusura è accessibile
- Soluzione: allocare i dati nello heap invece che sullo stack

```
(int → int) f (int x) {                    // x is heap allocated
        int g (int y) {
                return x + y;              // because g references x
        }
        return g;                          // and g will be used after f returns
}
```

Chiusure: implementazione

- Problema 2: una chiusura non può essere rappresentata solo da un entrypoint (il codice non saprebbe accedere alle variabili non locali!)
- Soluzione:
  chiusura = function pointer (for code) +
  record of pointers to data

```
(int → int) f (int x) {
        int a = 3;
        int g (int y) {
                return x + y +a;
        }
        return g;
}
```

```
(int → int) * (int * int)) f (int x) {
        int* hx = new_heap(x);
        int* ha = new_heap(3);
        return (g, { dx = hx; da = ha });
}
int g (int d, int y) {
        return d->x + y + d->a;
}
```

Chiusure vs Oggetti

- Oggetto
  - record di campi e puntatori a funzioni (metodi)
  - lo scope dei campi si estende solo ai metodi
  - I metodi prendono (implicitamente) in input il record per accedere ai campi e invocare gli altri metodi
- Chiusura
  - record di puntatori a dati (comprese altre chiusure) + un puntatore a funzione
  - Lo scope è determinato dal nesting sintattico
  - Il puntatore a funzione prende in input il record per accedere ai dati (e alle altre chiusure)

Le chiusure sono come metodi non intrappolati in un oggetto.

Chiusure in linguaggi HO

- Un linguaggio è di ordine superiore (Higher Order) quando le funzioni (= chiusure) sono oggetti di prima classe
- Spesso presente una sintassi semplificata per scrivere funzioni anonime (espressioni che definiscono una funzione e ne ritornano la chiusura)
- Nessuna differenza sintattica fra funzioni e chiusure
- Esempio (in Ocaml):

```
let maggiorenni ?(eta=18) =
  List.filter (fun x -> x <= eta)   (* questa è applicazione
                                       parziale di List.filter *)
```

Uso delle chiusure

- Per garantire data-hiding
  Esempio: un counter in Ocaml; il valore può essere acceduto solo tramite una coppia getter/setter

```
let new_counter n =
  let c = ref n in
   (fun () -> !c), (fun x -> c := x) /* here two closures are returned */

let get1,set1 = new_counter 0 in
let get2,set2 = new_counter 17 in
  set1 (get2 ())  /* both counters are now 17 */
```

Uso delle chiusure

- Per implementare strutture di controllo

```
let for_loop start stop step f =
  let rec aux i =
   if i < stop then (f i ; aux (i + step))
  in
   aux start

let print_even a =
  for_loop 0 (Array.length a – 1) 2 (fun i -> a.[i])
```

Quante definizioni di chiusure ci sono nel codice OCaml precedente?

Uso delle chiusure

- Per implementare callback nella programmazione asincrona/ad eventi

```
var http = require('http');
var fs = require('fs');
http.createServer(function (req, res) {
  fs.readFile('demofile1.html', function(err, data) {
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.write(data);
    res.end();
  });
}).listen(8080);
```

Quante definizioni di chiusure ci sono nel codice JavaScript precedente?

Uso delle chiusure

- Per decorare altre funzioni

```
def logging(f):
   def log_f(i):
      print("I am calling", f, "on", i)
      r = f(i)
      print("The result is", r)
      return r
   return log_f

def double(x): return x * 2
doublel = logging(double)
```

Quante definizioni di chiusura ci sono nel codice Python precedente?

Uso delle chiusure

- Per limitare accessibilità

```
let reverse l =
      let rec aux acc =
            function
                  [] → acc
              | hd::tl → aux (hd::acc) tl in
       aux [] l

let f,g =
      let h,i =
            let j x = … in
            let h x = …. in
            let i x y = j x + h x in
            h, i
      in ….                    (* solo h e i in scope qui *)
      …  in …                  (* solo f e g in scope quii *)
```

Uso delle chiusure

- Non possono essere utilizzate per limitare l'accessibilità di costrutti non di prima classe

  Esempi:

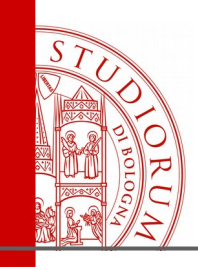
  - posso dichiarare un tipo all'interno di un blocco?
  - e un'eccezione?

Riassunto

- Una chiusura è una funzione che usa variabili non globali definite negli scope più esterni
- Nei linguaggi di alto livello non vi è differenza sintattica fra funzioni e chiusure e le chiusure sono oggetti di prima classe
- Le chiusure hanno molteplici usi
- Il linguaggio può supportare una sintassi per definire funzioni anonime (che spesso sono chiusure)
- A basso livello una chiusura è una coppia puntatore al codice, record puntatori ai dati su cui operare e quindi assomiglia a un oggetto

Funzioni anonime: sintassi

- Python: lambda x, y: x * y
- Ocaml: fun x y -> x * y
- Go: func(x int, y int) int { return x * y }
- Lisp/Scheme: lambda (x y) (* x y)
- JavaScript: function(x, y) { return x * y }
- Scala: (x: Int, y: Int) => x * y
- Erlang: fun (X, Y) -> X * Y
- Haskell: \ x y -> x * y
- Rust: |x: i32, y: i32| x * y
  Ma con molte varianti e complicazioni



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Claudio Sacerdoti Coen

Dipartimento di Informatica: Scienza e Ingegneria (DISI)

claudio.sacerdoticoen@unibo.it

www.unibo.it