Lezione 8 – Call and Return

Programmazione

Modulo 4 – Sottoprogrammi

Unità didattica 1-5 – Programmazione Imperativa in C

Marco Anisetti

Università degli Studi di Milano - SSRI

Sottoprogrammi e astrazione

- Il concetto di astrazione è un elemento fondamentale per risolvere i problemi
- E' assodato che il cervello umano non riesce a parallelizzare il pensiero in modo indefinito
- La soluzione di un problema passa quindi obbligatoriamente da delle astrazioni che limitano il numero di problemi che vengono gestiti contemporaneamente dal nostro cervello (sottoproblemi).
- Questo approccio è alla base della progettazione Top Down e dell'organizzazione procedurale dei programmi
- **L'astrazione procedurale equivale al raggruppamento di più problemi dettagliati in un unico macro-problema**

Sottoprogrammi

- Sono uno strumento potente per scomporre il problema in sottoproblemi più gestibili
- Non serve solo per abbreviare il lavoro di codifica ma in modo essenziale per articolare, suddividere e strutturare un programma in componenti fra loro coerenti
- La struttura è determinante per la comprensibilità del programma
    - Migliora la leggibilità e la verificabilità del codice

- L'estrazione di una parte di programma in un sottoprogramma permette di
    - mettere in risalto le variabili da essa influenzate
    - di porre in risalto le condizioni da soddisfare per l'ottenimento del risultato intermedio

Sottoprogrammi

- Mettono in evidenza i campi di influenza delle variabili contenute

- La parola ci aiuta a pensarli come se fossero di programmi a parte che comunicano con il programma chiamante attraverso parametri o tornando valori.
    - **ambiti di esecuzione separati**

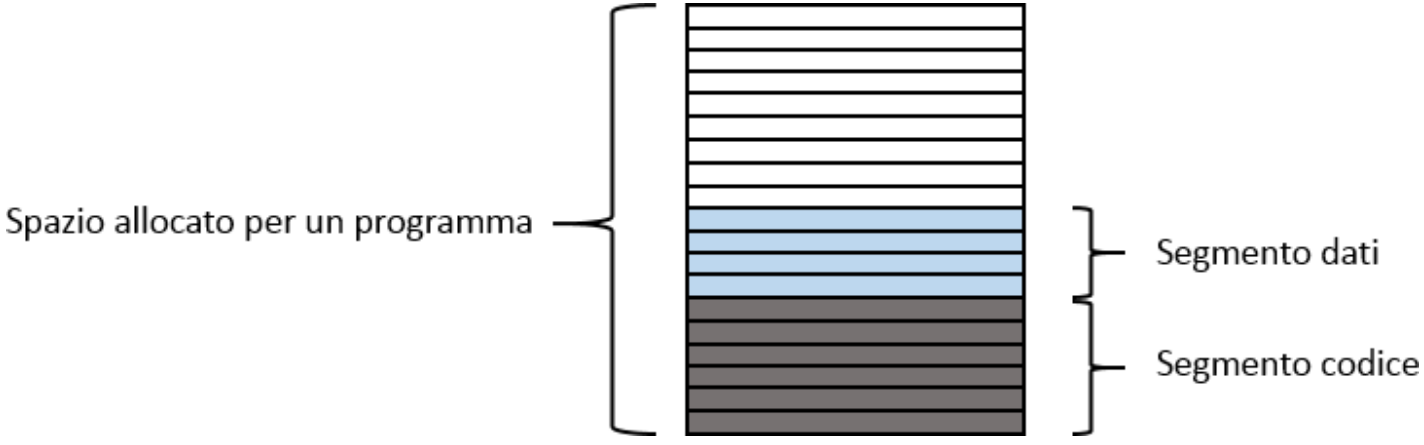
- Accento sulla loro **interfaccia** **ovvero cosa prendono in input e cosa danno in output**

Segmentazione della memoria(1)

- E' importante per un programmatore comprendere cosa succede al programma che scrive nel momento in cui viene eseguito
- Per prima cosa quando viene eseguito il programma è stato trasformato in un codice macchina direttamente interpretabile dall'elaboratore
- In generale i riferimenti ad indirizzi di memoria locali al processo (variabili) sono espressi in forma relativa (dipende dal traduttore usato per passare al linguaggio macchina)
- La gestione della memoria e del caricamento del programma in memoria è a carico del Sistema Operativo (definisce l'indirizzo di base)
  - Approfondimenti nel corso di Sistemi Operativi

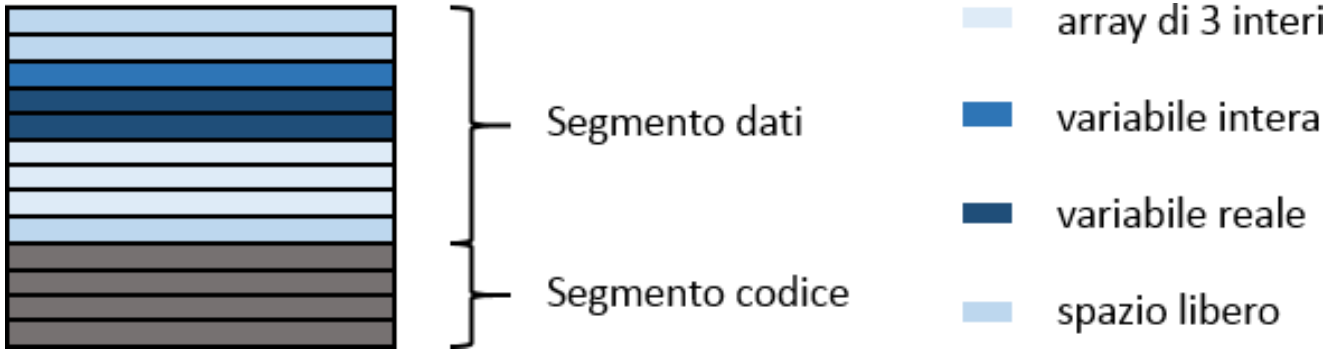
Segmentazione della memoria(2)

- Con una visione semplicistica possiamo dire:
- **Segmento codice**: Contiene il codice del programma (ed in alcuni casi le costanti). *Normalmente viene condiviso fra tutti i processi che eseguono lo stesso programma.* Viene marcato in sola lettura per evitare sovrascritture accidentali (o maliziose)
- **Il segmento dei dati**: Contiene le variabili (a volte diviso in due parti, variabili inizializzate e non inizializzate)



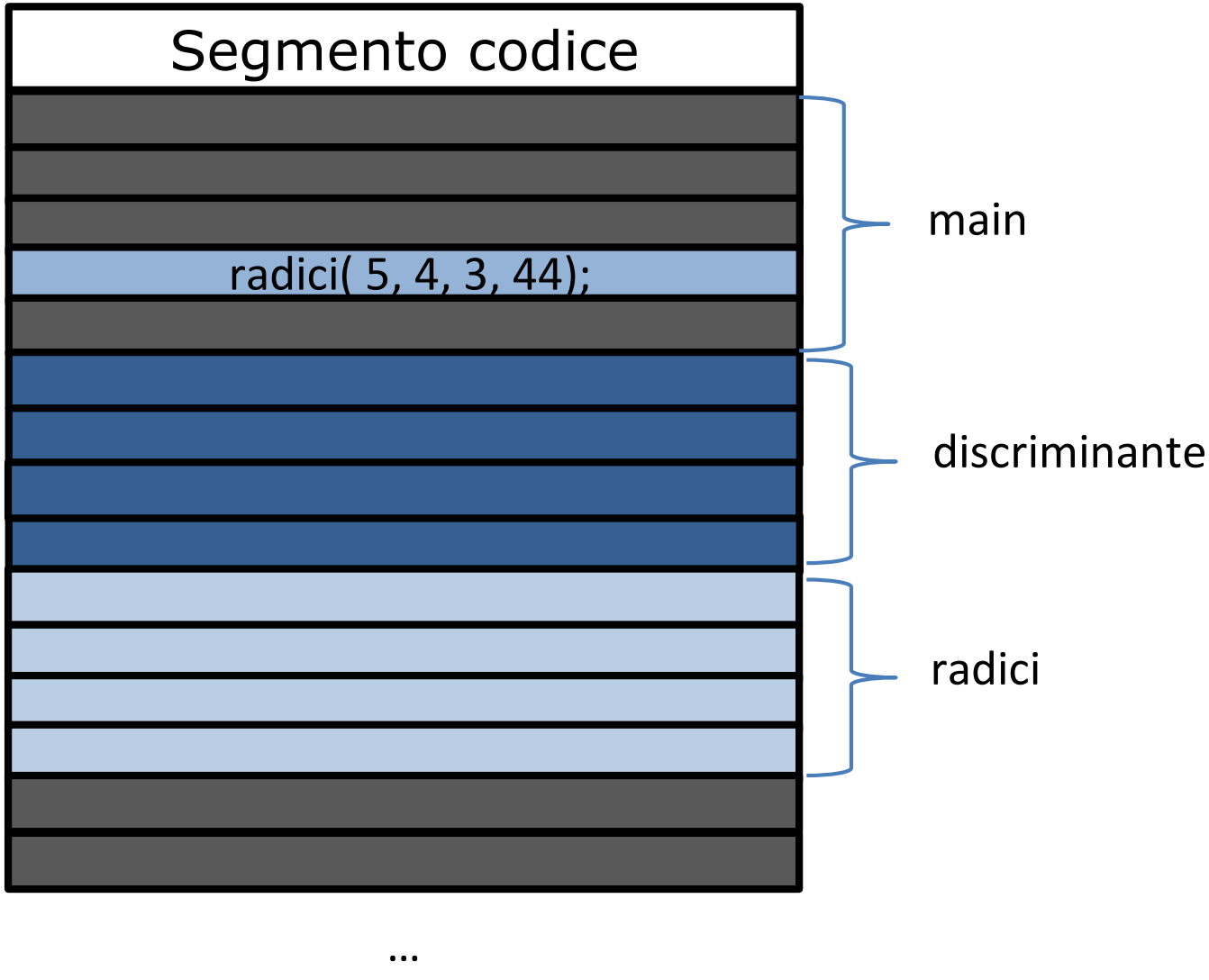
Dati strutturati in memoria

- Vediamo un esempio di come dovrebbero essere allocate le variabili strutturate in memoria



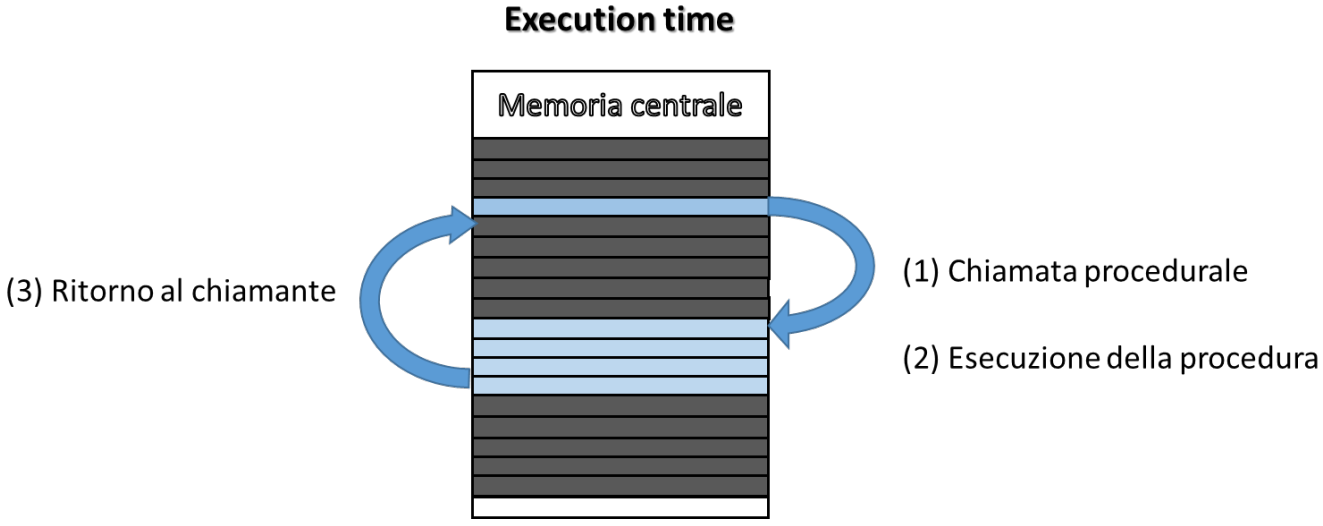
Segmentazione del segmento codice

Nel segmento codice avrò le varie zone dove trovo i sottoprogrammi



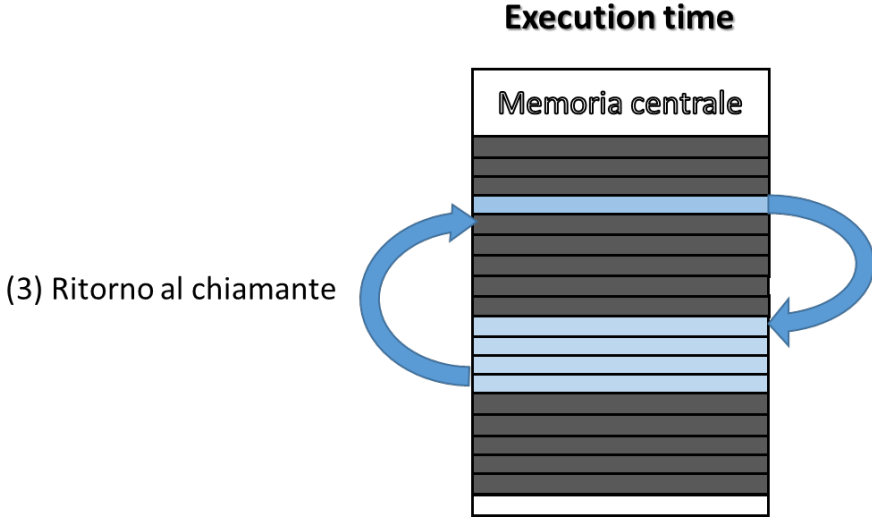
Call and return(1)

- Essendo ambiti di esecuzione separati le procedure o funzioni vengono effettivamente chiamate dal programma che ne fa uso e questa operazione scatena una serie di eventi
- Come cosa fondamentale un salto all'esecuzione di codice della procedura/funzione, la sospensione dell'esecuzione del chiamante e il relativa salto indietro quando termina



Call and return(1)

- Essendo ambiti di esecuzione separati le procedure o funzioni vengono effettivamente chiamate dal programma che ne fa uso e questa operazione scatena una serie di eventi
- Come cosa fondamentale un salto all'esecuzione di codice della procedura/funzione, la sospensione dell'esecuzione del chiamante e il relativa salto indietro quando termina



Valuta i parametri attuali

Alloca memoria per la funzione da chiamare

Assegna i parametri attuali ai formali

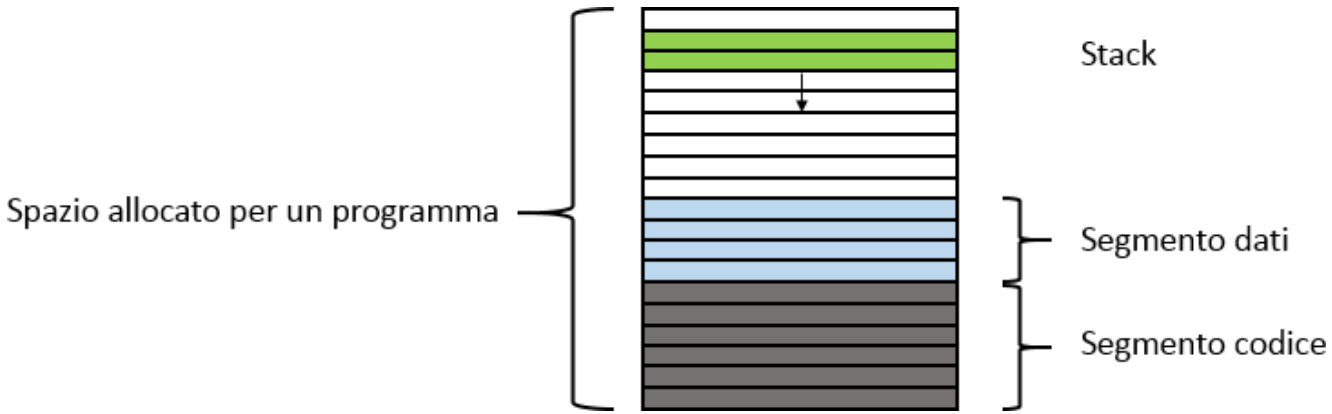
Call and return: Stack(2)

- Esiste una struttura di supporto per memorizzare l'indirizzo di ritorno dopo la chiamata, i parametri passati e utilizzati da procedure o funzioni
- Tale struttura è un segmento di memoria del processo chiamata stack
- E' una struttura dati astratta che agisce come una pila LIFO (Last In First Out)
- Per primo viene inserito l'indirizzo di ritorno (l'indirizzo dell'istruzione successiva alla chiamata)
- In seguito i parametri
- Poi le eventuali dichiarazioni locali alla procedura o funzione

Memoria del processo

- Con la gestione dei sottoprogrammi la struttura della memoria si arricchisce di un segmento stack

- Nello stack viene allocata la memoria per gestire la calla and return



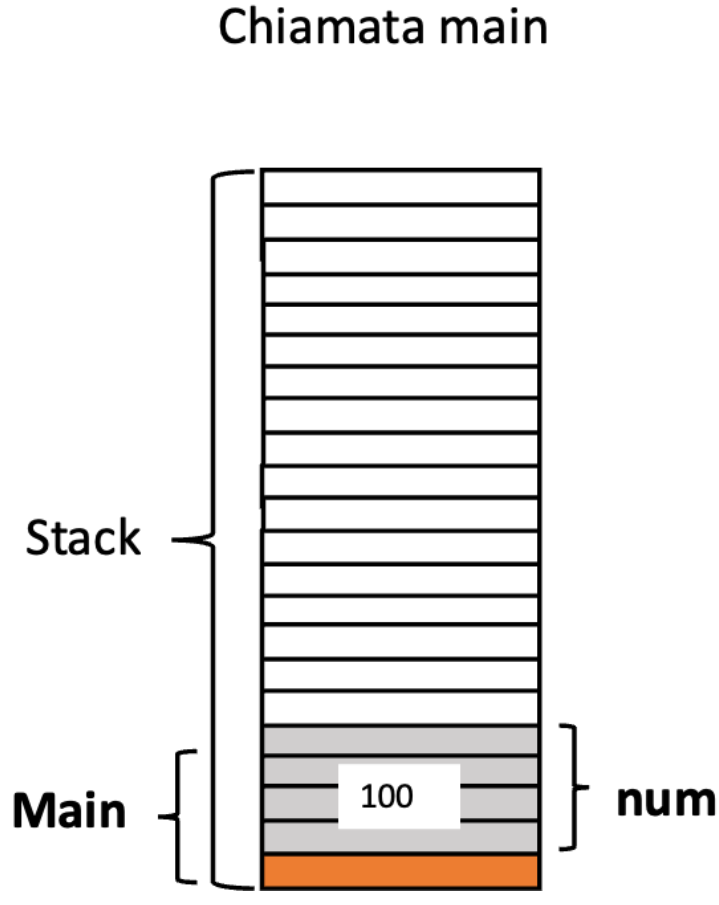
Record di attivazione(1)

- La visibilità della procedura o funzione è determinata dallo stack
- L'insieme di oggetti che vengono caricati nello stack ad ogni chiamata si chiama record di attivazione
- Ogni esecuzione di sottoprogramma ha il suo record di attivazione
- **Anche il programma principale main ha un record di attivazione nello stack**

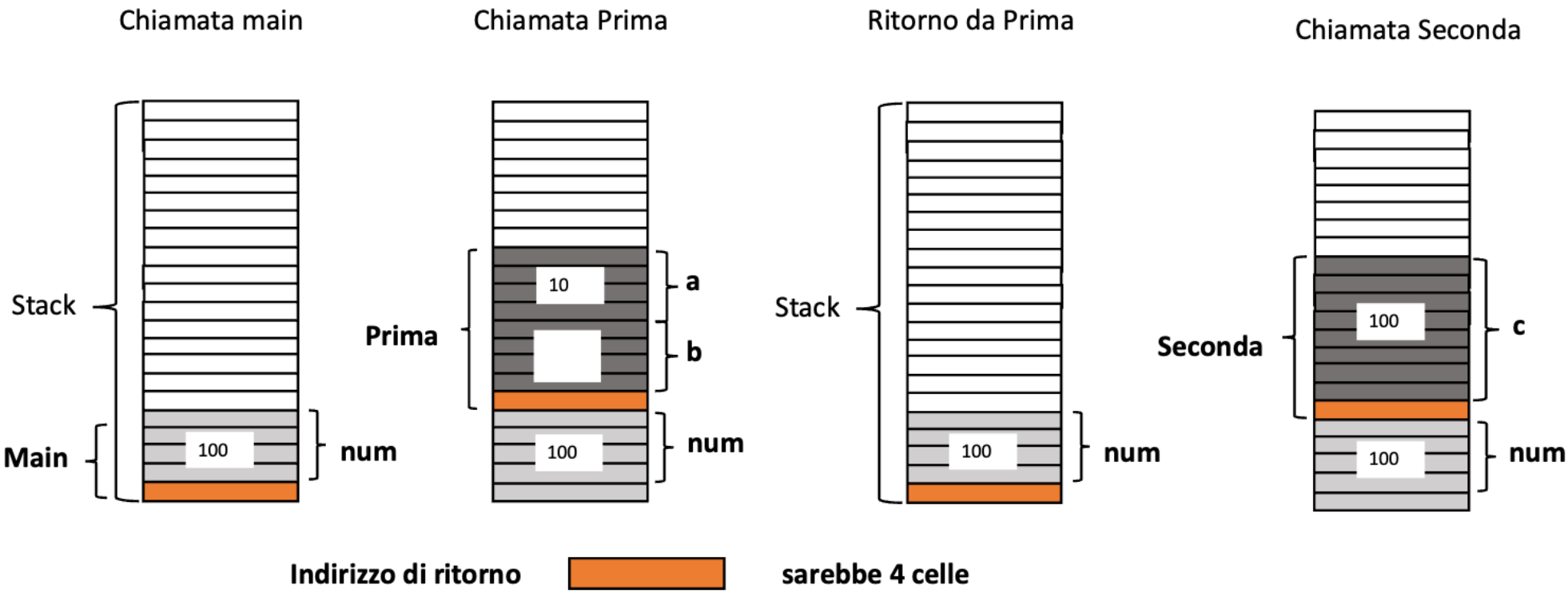
Record di attivazione(2)

[Esempio in linguaggio C]

```
void Prima(int a){
   int b;
}
void Seconda(double c){
}
int main(){
   int num=100;
   Prima(10);
   Seconda(num);
}
```



Record di attivazione(3)



N.B. il parametro attuale di c viene promosso a double quando chiamo Seconda

Stack ricapitolo

- Struttura dati contigua, ma NON ad accesso indicizzato come un array, L’accesso è LIFO e contenente dati eterogenei.

- **LIFO:** Si accede solo dall’alto inserendo o togliendo come se fosse una pila di oggetti e non ne puoi sfilare uno a metà.

Stack ricapitolo

- La sua gestione nella chiamata a funzione prevede allocazione nelle cima della pila di:
    - Indirizzo di ritorno
    - Parametri formali
    - Variabili locali
    - Risultato della funzione (omesso negli esempi di prima)

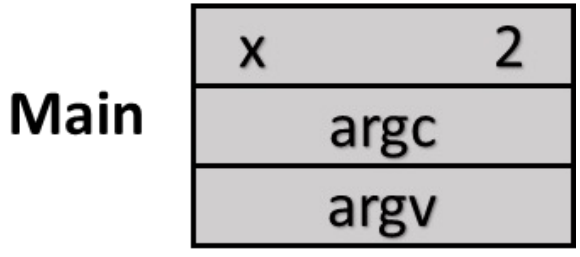
- Quando la funzione termina, lo spazio viene deallocato e il valore di ritorno ritornato. L'indirizzo di ritorno viene usato per tornare al chiamante.

Stack ricapitolo

- Lo stack contiene quindi tutte **le strutture dati statiche** di un programma
- E’ rilevante nella determinazione degli scope soprattutto nelle chiamate a funzione
- **Lo stack è implicito e il programmatore non lo può manipolare direttamente**

Gestione Stack: esempio

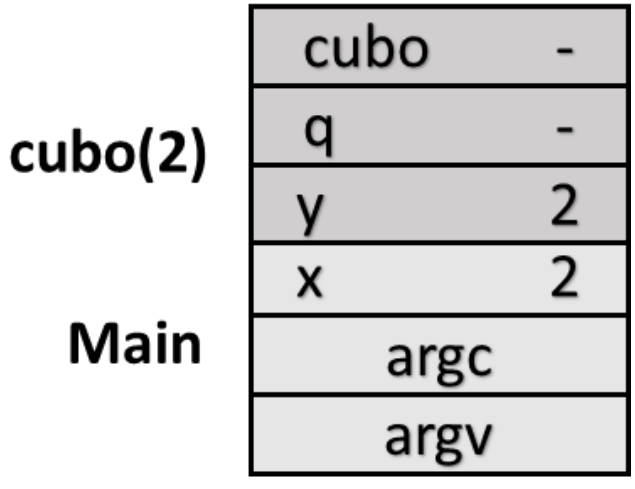
- Considerare il programma del calcolo quadrati e cubi
- Il main esegue l'istruzione cubo(x);



Gestione Stack: esempio

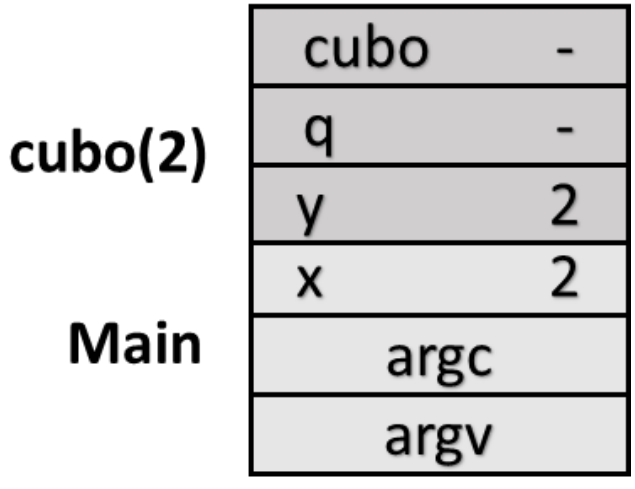
- Considerare il programma del calcolo quadrati e cubi
- Il main esegue l'istruzione cubo(x);

1. il parametro attuale è x e vale 2
2. si alloca sullo stack spazio per y, q e il risultato di cubo
3. si assegna il valore 2 a y



Gestione Stack: esempio

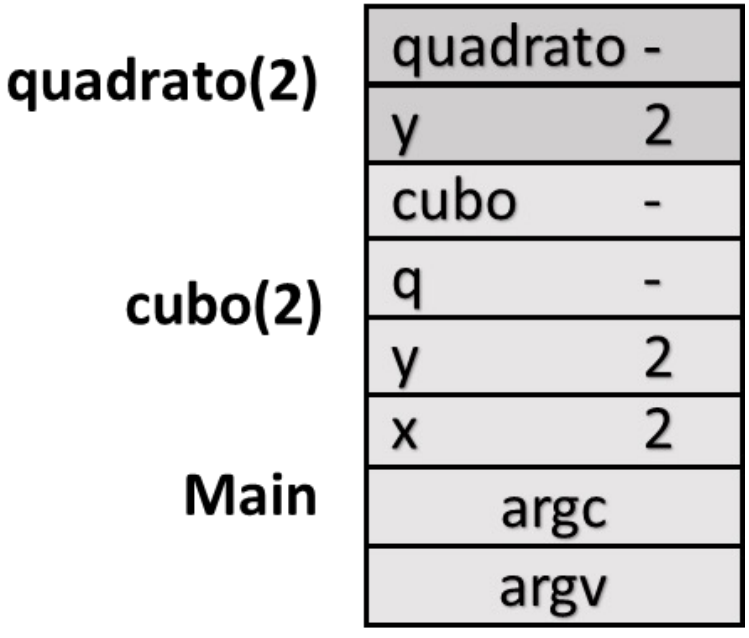
- La funzione cubo esegue la funzione quadrato(y);



Gestione Stack: esempio

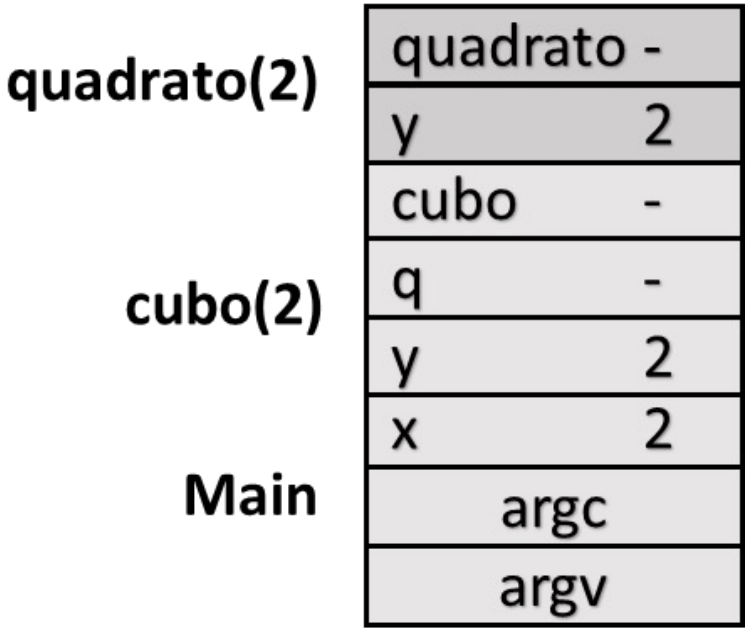
- La funzione cubo esegue la funzione quadrato(y);

1. il parametro attuale è y e vale 2
2. si alloca sullo stack spazio per y e il risultato di quadrato
3. si assegna il valore 2 a y



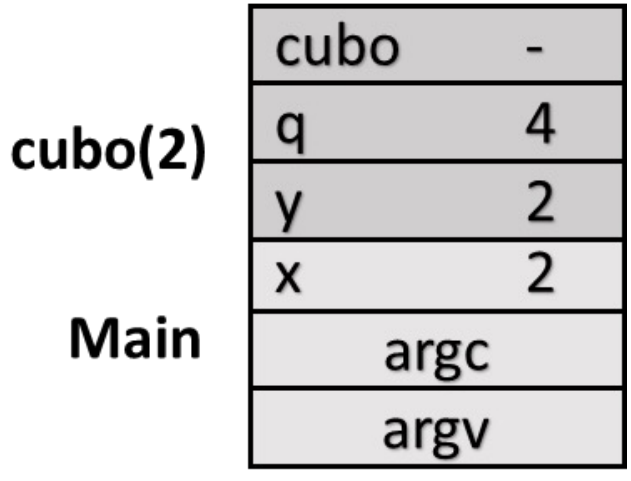
Gestione Stack: esempio

- Al termine della funzione:
    - Valuta il l'espressione dell'istruzione di return
    - Dealloca il record di attivazione
    - Restituisce il risultato
- la funzione quadrato esegue l'istruzione return y * y;



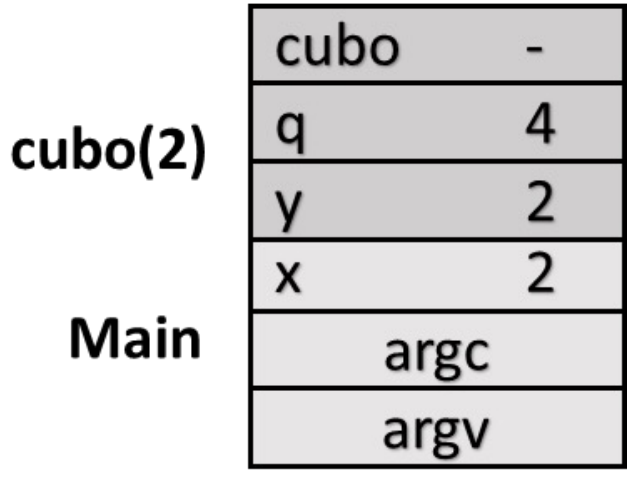
Gestione Stack: esempio

1. il risultato y * y vale 4
2. si dealloca lo spazio per y e per il risultato (salvandolo a parte)
3. si restituisce 4 alla funzione cubo, che lo assegna a q



Gestione Stack: esempio

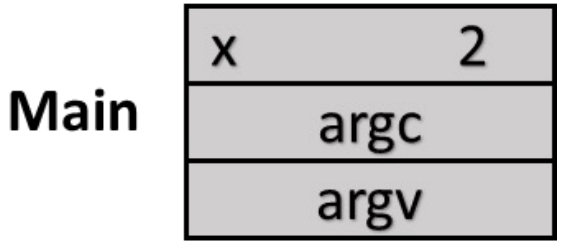
- la funzione cubo esegue l'istruzione return q * y;



Gestione Stack: esempio

- la funzione cubo esegue l'istruzione return q * y;

1. il risultato q * y vale 8
2. si dealloca lo spazio per y, q e per il risultato (salvandolo a parte)
3. si restituisce 8 al main, che lo passa alla funzione printf



Conversioni di tipo nelle chiamate a funzione

- Il passaggio dei dati e il recupero del risultato obbediscono alle regole implicite per le conversioni di tipo
- Se passo un double a quadrato che vorrebbe in ingresso un int che succede?
- È come assegnare un double ad un int avviene un casting implicito e si perde la parte dopo la virgola

Controlli sui tipi

- I tipi associati a delle variabili si estendono alle espressioni (*type system* del linguaggio)

  Overload: significati differenti a seconda del contesto

  Coercion: promozioni automatiche a tipi differenti per risolvere conflitti

  Polimorfismo: una funzione polimorfica ha un tipo parametrico o generico

Controlli sui tipi

- A seconda del linguaggio il tipo di una variabile può essere fisso o meno
  - Valore dinamico ma tipo fisso (binding statico)
  - Tipo associato run-time (binding dinamico)
- Equivalenza tra tipi
- Controllo statico o dinamico
  - Errore sul tipo avviene quando una funzione (o un operatore) si aspetta un tipo come argomento e ne trova un altro
  - Controlli sul codice prima che venga tradotto
  - Controlli sui tipi run-time (controllo sforamento indici di un array)

Tipizzazione

- **Tipizzazione**: tipo di regole che un linguaggio impone (livello sintattico e semantico) circa la tipizzazione dei dati e al loro uso in relazione del tipo
- **Fortemente tipizzato**: specificare il tipo per ogni elemento sintattico che ha un valore. Il linguaggio garantisce che il valore sia usato in coerenza al tipo.
    - Tipizzazione statica
    - Type safety: controlli esaustivi (compile o execution time) sull'uso dei valori in relazione al tipo
    - Impossibilità di convertire tipi
- **Debolmente tipizzato**: viola qualche cosa del fortemente tipizzato

Tipizzazione del C

- Il C è debolmente tipizzato (non tanto debolmente come un assembler)
    - Permette il castring
    - Permette puntatori void che hanno conversioni implicite (ne parliamo tra poco dei puntatori)
    - Presenza del costrutto union, che permette di interpretare una collezione di dati secondo attribuzioni di tipi differenti (dobbiamo ancora vederle)

Stack e scope

- Ogni funzione ha un suo spazio riservato sullo stack e può agire solo su parametri e variabili locali
    - non accede alle variabili delle funzioni chiamate
    - non accede alle variabili della funzione chiamante

- Le variabili globali dichiarate fuori dal main sono visibili in ogni funzione del file

- ***Le variabili globali sono di norma inutili e sconsigliabili***

- nascondono il fatto che una funzione usa o modifica una certa variabile

- **Interfaccia di una funzione deve essere completamente definita**

Passaggio parametri

Come abbiamo visto:

Ad ogni chiamata i parametri attuali vengono valutati e sostituiscono i parametri formali nella funzione chiamata

In generale vi sono due modi di passare i parametri in C

- **per valore (call by value):** la funzione accede a copie dei dati (valori attuali copiati nei parametri formali) e le eventuali modifiche interne sui formali non si riflettono sugli attuali

- **per indirizzo(call by reference):** la funzione accede ai dati originali (vengono copiati i riferimenti ai parametri attuali nei formali) e le eventuali modiche interne sui formali agiscono anche sugli attuali

Passaggio per valore

Nessun effetto collaterale

Si possono usare i parametri formali come variabili locali, modificandoli per produrre il risultato senza sporcare i parametri attuali

```
int main ()
{
int x, q;
x = 10;
q = quadrato(x);
/* x vale ancora 10 */
return 0;
}
```

```
int quadrato (int y)
{
y = y * y;
/* qua y vale 100 */
return y;
}
```

Passaggio per indirizzo

- Il C non supporta davvero il passaggio per indirizzo (come il var del Pascal): Per simularlo si passa il valore dell'indirizzo di memoria del parametro

- Così da dentro la funzione modifico una variabile fuori dal suo record di attivazione

- **Genera effetti collaterali**

- *Ovvero alterazioni del contenuto delle variabili in questo caso fuori dallo scope*

Passaggio per indirizzo

- Siccome non è **supportato** dal linguaggio è tutto a carico del programmatore essere in grado di gestirlo bene e in modo esplicito

- Abbiamo visto come estrarre il valore dell'indirizzo (&<variabile>), ma non ne sappiamo ancora cosa possiamo farci e come manipolarlo

- In C l'indirizzo di una variabile viene gestito come **un tipo** particolare che si chiama **puntatore**

Ritorni di funzioni

- Quando uso una funzione passando per valore i parametri posso solo ottenere un valore di ritorno e uno solo.

- Non posso sperare di tronare un vettore di valori visto che poi non posso assegnarlo a qualche cosa del chiamante.

- Al momento se devo tornare più cose:

- **Lo devo fare attraverso i parametri passati per indirizzo** che posso quindi modificare da dentro e ottenere le modifiche anche fuori.
- Quindi posso pensare a dei parametri che vengono passati in ingresso con il solo obiettivo di venir modificati dalla funzione e fungere da parametri di ritorno in pratica

- Vedremo altri modi più avanti

Esistono altre forme di passaggio parametri?

In generale esistono altre tecniche NON considerate dal linguaggio C:

call by result: copiare formali in attuali all'uscita dalla funzione (out-mode semantics)

call by value-result: copia il valore in chiamata e lo copia ancora in uscita dalla funzione

call by name: passo il nome simbolico dei parametri permettendo sia accesso che modifica. Come se copiassi dentro il corpo della funzione il parametrico attuale in ogni posto dove compare il relativo formale. **Se l'attuale è una espressione, implica che bisogna valutare ad ogni uso (es passo a[i], ogni volta rivaluto e se i cambia allora cambia il valore di a[i] ovviamente)**