



Linguaggio C++

Sottoprogrammi 2^a parte

(passaggio di parametri per riferimento e conclusioni)



Funzione scambia() C e C++

```
int main(){  
    int m,n;  
        //lettura  
        scambia(m,n);  
        //stampa  
}
```

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

(vedi disegno Stack alla lavagna)



Osservazioni C e C++

- Utilizzando il passaggio di parametri per valore non siamo riusciti a realizzare il compito della funzione scambia().
- Soluzione:
 - Utilizzare il **passaggio di parametri per riferimento** (caratteristica solo del C++).



Passaggio di parametri per valore C e C++

- I parametri formali sono delle locazioni di memoria nelle quali vengono copiati i contenuti dei parametri attuali.
- Nel passaggio di parametri per valore i parametri formali e parametri attuali fanno riferimento ad aree distinte di memoria.



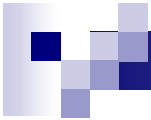
Variabili di tipo reference C++

- In C++ è possibile avere un nome alternativo per la stessa variabile attraverso la dichiarazione di un “riferimento” (reference).

```
int a;
```


```
int &b=a;    //⇒ b e a sono due nomi per una stessa area di memoria
```

- Una “reference” non è una copia della variabile a cui si riferisce bensì è la stessa variabile sotto un altro nome (alias).



Passaggio di parameri per riferimento C++

- Nel passaggio di parametri “by reference” i parametri formali sono alias dei parametri attuali.
- Alias = sono identificatori che denotano una variabile già definita.



Passaggio di parameri per riferimento C++

```
#include <iostream>
using namespace std;
```

```
void scambia (int &a, int &b);
```

```
int main(){
int m, n;
```

```
.....
```

```
    scambia(m,n);
```

```
.....
```

```
}
```

```
void scambia (int &a, int &b){
int temp;
```

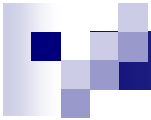
```
    temp=a;
```

```
    a=b;
```

```
    b=temp;
```

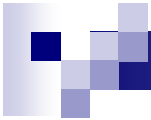
```
}
```

Disegno dello stack alla lavagna.



Passaggio di parameri per riferimento C++

- Con i riferimenti tutto il lavoro viene svolto dal compilatore che forza il passaggio dell'indirizzo della variabile alla funzione (anche se l'utente non passa esplicitamente l'indirizzo).
- Il chiamante non distingue dalla chiamata se il passaggio è per valore o per riferimento. Si deve guardare il prototipo della funzione.



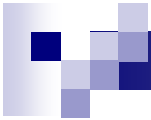
Passaggio di parametri per riferimento C++

- Nel passaggio di parametri per riferimento parametri formali e parametri attuali fanno riferimento a stesse aree di memoria.



Esercizio

- Scrivere un pgm che, immessi il numeratore e il denominatore di una frazione, verifichi se è ridotta ai minimi termini, facendo uso di una funzione C++ che calcoli il massimo comun divisore.



```
int main(){
int num, den, mcd;

//leggi frazione
mcd = maxComDiv(num,den);
if(mcd == 1)
    cout<<"La frazione è ridotta ai minimi termini";
else{
    num /= mcd;
    den /= mcd;
    cout<<"La frazione equivalente è "<<num<<"/"<<den;
}
}
```



```
int maxComDiv (int a, int b){  
    int resto;  
    do{  
        resto = a%b;  
        a = b;  
        b = resto;  
    }while(resto != 0);  
  
    return a;  
}
```

(vedi disegno alla lavagna Stack)



Osservazione

- Mostrare che utilizzando il passaggio di parametri per riferimento non è possibile poi stampare frazione equivalente, perché cambiano i valori di num e den (a meno di salvarli nel main()).

- Anziché:

`int maxComDiv(int a, int b); // ← migliore! (è una “vera” funzione)`

potevo usare:

`□ void maxComDiv(int a, int a, int & mcd); //brutto!!!!`



Versione senza **return**

```
void maxComDiv(int a, int b, int & mcd){  
    int resto;  
    do{  
        resto = a%b;  
        a = b;  
        b = resto;  
    }while(resto != 0);  
    mcd = a;  
}
```

Chiamata: maxComDiv(num,den,mcd);

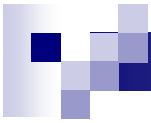


Esercizio

- Scrivi una funzione C++ che ritorni sia la somma che la differenza di due argomenti.
- Nota: E' una procedura!



```
int main(){  
    int a, b;  
    int somma, diff;  
  
        //lettura di a e b  
        sommaDiff(a, b, somma, diff);  
        .....  
}
```

```
void sommaDiff(int x, int y, int &sum, int &diff){  
    sum = x + y;  
    diff = x - y;  
}
```

(vedi disegno Stack alla lavagna)



Regole di visibilità (scope rules)

- Un pgm C/C++, come abbiamo più volte ricordato, è costituito da un insieme di funzioni.
- Ogni funzione definisce e utilizza internamente delle variabili proprie: una variabile definita internamente ad una funzione è detta variabile locale.



Regole di visibilità (scope rules)

- Esiste la possibilità di definire anche variabili globali: queste sono accessibili a tutte le funzioni del pgm (ti ricordo che noi al momento abbiamo visto pgm scritti su un unico file).



Regole di visibilità (scope rules)

- L'allocazione di una variabile è il processo di generazione della variabile mediante attribuzione di una porzione di memoria, a cui viene associato l'identificatore della variabile stessa. Tale allocazione avviene in modo temporaneo per le variabili locali e permanente per quelle globali.



Regole di visibilità (scope rules)

- La visibilità (scope) di una variabile è la porzione di pgm all'interno della quale la variabile è conosciuta e accessibile.
- Per le variabili valgono le seguenti regole di allocazione e di visibilità:
 - le variabili definite all'interno di una funzione (cioè le variabili locali alla funzione) sono allocate soltanto durante l'esecuzione della funzione stessa (cioè 'nascono' quando questa viene attivata e 'muoiono' quando il flusso ritorna al chiamante) e sono visibili soltanto all'interno di essa. Ti ricordo che per la funzione main() valgono le stesse regole delle funzioni che hai implementato.
 - le variabili definite all'esterno (di tutte le funzioni, compresa la main()) sono allocate in modo permanente per tutta la durata di esecuzione del pgm e sono visibili in tutte le funzioni dichiarate successivamente nello stesso file sorgente.



Regole di visibilità (scope rules)

- Variabili locali aventi lo stesso nome ma definite in funzioni diverse non sono correlate e rappresentano entità diverse.
- I parametri formali delle funzioni devono essere considerati come variabili locali.
- Ti ricordo infine la possibilità di definire variabili locali all'interno di un blocco.
- Un blocco è un insieme di dichiarazioni, definizioni e istruzioni racchiuso tra parentesi graffe oppure una funzione formata da due elementi indivisibili: l'intestazione ed il corpo.



Regole di visibilità (scope rules)

```
void change();


int x = 4;           //var globale

int main(){
    int x;           //var locale
    x = 0;
    change();
    cout<<x;
    return 0;
}
```

```
void change(){
    int x;           //var locale
    x=1;
}
```

Che cosa stampa in output
questo pgm?

- Concetto di mascheramento delle variabili.



Regola pratica per stabilire la direzionalità di un parametro.

- Porsi idealmente all'interno del sottopgm e chiedersi:
 - ☐ Il valore del parametro viene fornito dall'esterno? (**IN**)
 - ☐ Il valore del parametro viene fornito all'interno? (**OUT**)
 - ☐ Il valore del parametro viene modificato all'interno del sottopgm? (**IN/OUT**)



Legame tra direzionalità e passaggio di parametri (**parziale!!!**)

- Si passano per valore tutti i parametri di solo ingresso.
- Si passano per riferimento tutti i parametri di O e di I/O.



Nota

- Ad un parametro (attuale/formale) vengono associati:
 - Un nome
 - Un tipo
 - Una posizione d'ordine
 - Una direzionalità (I, O, I/O)



Alcuni buoni motivi per usare i sottopgm (vantaggi della metodologia Top-Down)

- **TEMPO** risparmiato per la scrittura e la verifica dei pgm: se una parte di lavoro è già stata svolta è chiaro che si risparmiano tempo e prove.
- **SPAZIO** occupato dai pgm: ciò si verifica quando uno stesso sottopgm è chiamato (in generale con parametri diversi) in più punti di uno stesso pgm.
- **LEGGIBILITA'** dei pgm: una chiamata ad un sottopgm occupa una sola riga invece di tutte quelle occupate dal corpo dello stesso sottopgm che dovremmo inserire nell'approccio senza sottopgm.



Alcuni buoni motivi per usare i sottopgm

(vantaggi della metodologia Top-Down)

- La possibilità di definire e di richiamare dei sottopgm costituisce uno strumento di **ESTENSIONE** del linguaggio
- Distribuzione del lavoro di analisi e di codifica.
- Semplifica la successiva codifica in un linguaggio ad alto livello.
- Facilita il controllo (test) del modulo isolato.
- Favorisce la manutenzione del s/w.
- Favorisce la riusabilità di alcuni moduli.
- I sottopgm costituiscono il primo fondamentale strumento che i programmatori incontrano per non disperdere il loro lavoro



Alcuni buoni motivi per usare i sottopgm (vantaggi della metodologia Top-Down)

- I sottogm consentono il ricorso all'**ASTRAZIONE PROCEDURALE/FUNZIONALE**. Con questa espressione indichiamo il punto di vista del programmatore nel momento in cui richiama una procedura. In quel momento egli astrae (si disinteressa) dai dettagli risolutivi insiti nel corpo della procedura: egli chiede semplicemente che in quel punto venga risolto il suo sottoproblema (non gli interessa il "come" ma solo il "cosa"). (v. concetto di black box)



NOTA

- Una delle critiche più forti , che oggi viene fatta alla metodologia Top-Down, è che essa difficilmente porta a costruire s/w riutilizzabile.



RICORDA

- In C++ esistono due modalità di passaggio di parametri
 - **per valore**
 - **by reference** (per riferimento)



RICORDA (2)

- In C esistono due modalità di passaggio di parametri
 - **per valore**
 - **per indirizzo** (alla funzione viene passato l'indirizzo di memoria della variabile. Utilizzo dei puntatori = variabili che possono contenere indirizzi di aree di memoria)



RICORDA (3)

- Se vogliamo essere precisi, e noi lo vogliamo, in C l'unica modalità di passaggio di parametri è quella **per valore**.
- Sconvolti?!!! MAI!!!



Array (mono e bidimensionali) come parametri di funzioni

- Un vettore (array monodimensionale) si passa in C/C++ **per indirizzo**.
- Più precisamente viene passato l'indirizzo dell'elemento di indice 0 dell'array stesso.



Vettori come parametri di funzioni

Esempio:

```
const int DIM = 10;
```

```
int main(){  
int vett[DIM];  
int numElem;
```

```
    //insert  
    stampa(vett, numElem);  
}
```



Vettori come parametri di funzioni

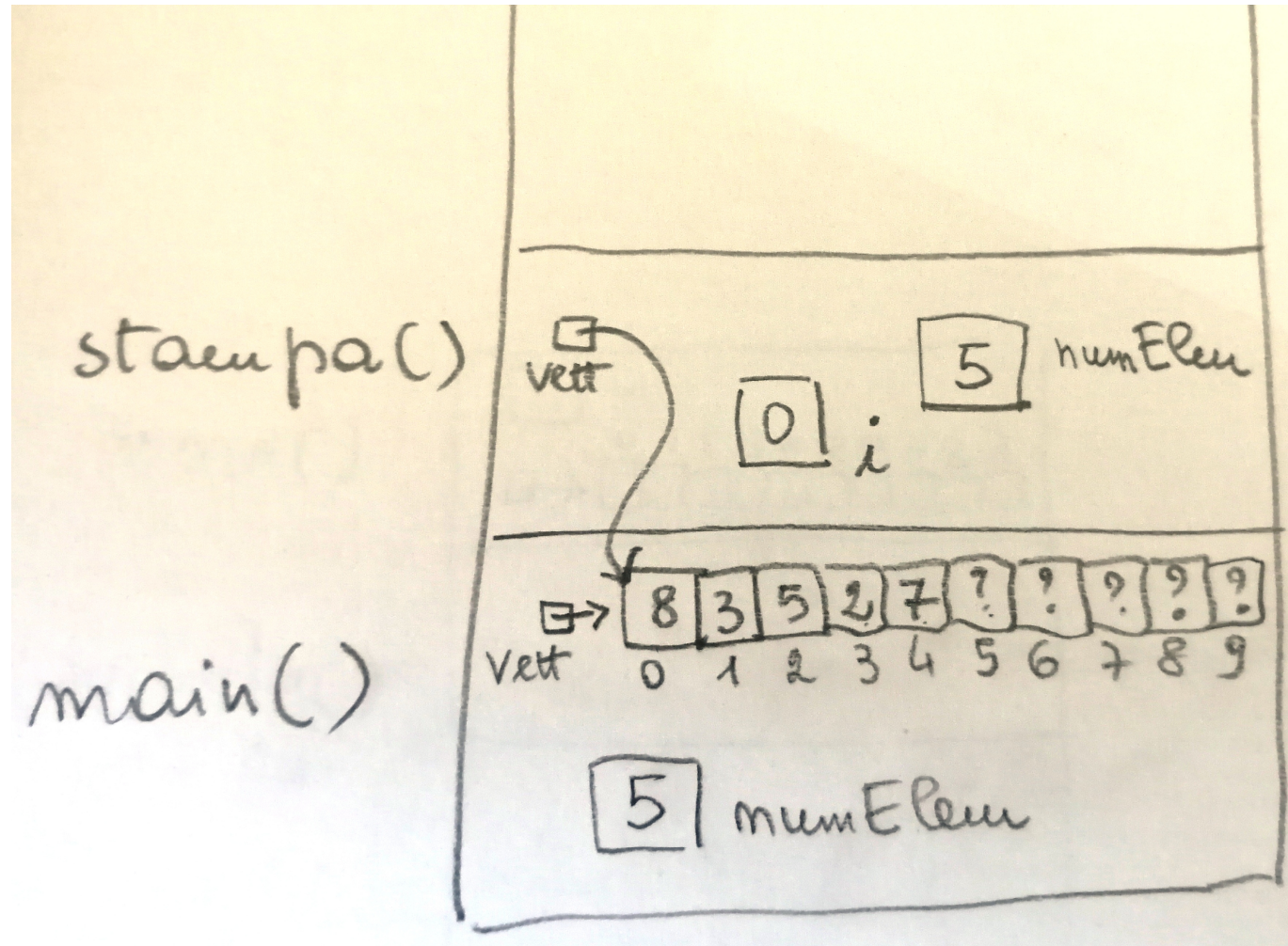
■ Prototipi equivalenti:

- ☐ **void stampa (int vett[], int numElem);**
- ☐ **void stampa (int vett[DIM], int numElem);**
- ☐ **void stampa (int [], int);**
- ☐ **void stampa (int * vett, int numElem);**



Funzione stampa()

```
void stampa (int vett[], int numElem){  
    int i;  
  
    for(i=0; i<numElem; i++)  
        cout<<vett[i]<<endl;  
}
```





ATTENZIONE!!!

- Considerato che un array in C/C++ può essere passato solo per indirizzo, è possibile all'interno del sottopgm, attraverso il parametro, modificare l'array (allocato nel record di attivazione del chiamante). Vedremo come è possibile evitarlo.
- Il parametro è infatti una variabile puntatore che punta alla prima componente dell'array (allocato nel record di attivazione del chiamante).
- Vedi disegno, alla lavagna, stack funzione stampa().



Funzione insert()

```
const int DIM = 10;
```

```
int main(){
```

```
int vett[DIM];
```

```
int numElem;
```

```
    insert(vett, numElem);
```

```
    stampa(vett, numElem);
```

```
    return 0;
```

```
}
```



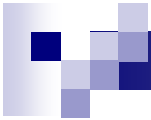

Funzione insert()

```
void insert (int vett[ ], int &numElem){           //E' una procedura!  
    int i;  
    do{  
        cout<<"Quanti elementi vuoi inserire?\n";  
        cin>>numElem;  
    }while(numElem < 1 || numElem > DIM);  
  
    for(i=0; i < numElem; i++){  
        cout<<"Inserisci l'elemento n. "<<i+1<<endl;  
        cin>>vett[i];  
    }  
  
}
```




Nota

- insert() essendo una procedura, preferisco non usare “return”.
- Quindi scrivo:
 - **void insert (int vett[], int &numElem);**
 - anziché
 - **int insert (int vett []);**



Versione con **return**

```
int insert (int vett [ ]){  
    int i, numElem;  
  
    do{  
        cout<<"Quanti elementi vuoi inserire?\n";  
        cin>>numElem;  
    }while(numElem < 1 || numElem > DIM);  
  
    for(i=0; i < numElem; i++){  
        cout<<"Inserisci l'elemento n. "<<i+1<<endl;  
        cin>>vett[i];  
    }  
    return numElem;  
}
```



Passaggio di una matrice (array bidimensionale) ad una funzione

- Una matrice (array bidimensionale) si passa in C/C++ **per indirizzo**.

```
#define MAX_R 3  
#define MAX_C 4
```

```
int main(){  
    int matrix [MAX_R][MAX_C];  
    int numRighe, int numColonne;
```

```
        insert(matrix, numRighe, numColonne);  
        stampa(matrix, numRighe, numColonne);  
}
```



Funzione stampa()

```
void stampa(int matrix[ ][MAX_C], int numRighe, int numColonne){
    int i, j;

    for(i=0; i<numRighe; i++){
        for(j=0; j<numColonne; j++){
            cout<<matrix[i][j]<<" ";
            cout<<endl;
        }
    }
}
```



NOTA

- Come vedi nel prototipo

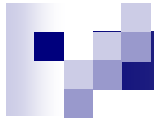
```
void stampa(int matrix[ ][MAX_C], int numRighe, int numColonne);
```

È obbligatorio indicare la dimensione delle colonne (vedi implementazione di una matrice in memoria centrale).



Esercizio

- Scrivi la funzione C++ `insert()` che inserisce i dati in una matrice.



Esercizio

- Scrivi una funzione C++ che, dato un vettore di interi, calcola la media aritmetica.



Soluzione

```
double mediaAritmetica(int vett[], int numElem){  
    int i, somma=0;  
  
    for(i=0; i<numElem; i++)  
        somma += vett[i];  
    return (double)somma/numElem;  
}
```



NOTA

■ Chiamata:

```
media = mediaAritmetica(vett, numElem);
```

Si può fare anche così:

```
void mediaAritmetica(int vett[ ], int numElem, double &media);
```

ma, essendo una “vera” funzione preferisco l’uso del “return”.



Esercizio

- Scrivi una funzione C++ che, dato un vettore di interi, restituisce il massimo e la sua posizione.
- E' una funzione o una procedura?
Perché?



Legame tra direzionalità e passaggio di parametri (completo!!!)

- Si passano per valore tutti i parametri di solo ingresso purché non siano array o dati che occupano molta memoria (record).
- Si passano per riferimento (by reference) tutti i parametri di O o di I/O, i dati che occupano molta memoria, non gli array.
- Gli array si passano solo per indirizzo, indipendentemente dalla direzionalità.



Osservazione

- L'uso di **const** nella dichiarazione dei parametri impedisce al codice della funzione di modificare l'oggetto puntato dal parametro.
- Esempio:
 - `void stampa (const int vett[], int numElem);`