

Il C nel C++: Funzioni

Funzioni (1)

- il concetto -> spezzare il programma in parti (procedure)
- una funzione è un parte di programma dotata di un nome che può essere richiamata in altri punti del programma
- tutte le funzioni hanno un tipo e ritornano un valore del tipo definito

Funzione (2)

■ Dichiarazione (*il prototipo*)

```
float ratio (int , int , float);
```

- il compilatore segnala un errore se in seguito si usano tipi differenti da quelli dichiarati

■ Definizione

```
float ratio (int x, int y, float r) {  
    return (r=x/y);  
}
```

Vantaggi delle Funzioni

- Localizzazione della funzionalità
 - correggibilità
 - manutenibilità
- Riduzione del corpo del codice
 - leggibilità globale
 - verificabilità
 - correttezza semantica dei parametri

Differenze ...

- In C++ una function deve sempre ritornare un valore (no default a int). Si può usare void per indicare una function che non ritorna nessun valore.

Return.cpp

- Attenzione all'uso di empty args list

```
int Fun1(void); // non ha argomenti
int Fun2();    // in C++ è la stessa cosa
              // in C = num argomenti indeterminato
```

I Parametri delle funzioni

In C++ i parametri delle funzione possono essere di due tipi:

- *parametri valore*, quando si passa il valore del parametro
- *parametri riferimento*, quando si passa la locazione di memoria (indirizzo) del parametro
 - (i) per puntatore a (*) [anche C]
 - (ii) per indirizzo di (&) [solo C++]

Il Tipo Puntatore

- Operatore indirizzo: &
&a fornisce l'indirizzo della variabile a
- Operatore di dereferenziazione: *
*p interpreta la variabile p come un puntatore e opera con il valore contenuto nella cella di memoria puntata
- Il tipo di dato "puntatore a"
 - `int *pa;` per i tipi interi
 - `float *pb;` per i tipi reali
 - `float *pb, pc;` attenzione !!

YourPets2.cpp

Passaggio per Valore (1)

```
# include <iostream>
int somma (int , int ); ←
void main() {
    int a, b, res;
    a = 5;
    b = 10;
    res = somma(a,b); ←
    cout << a << b << res;
}

int somma (int x, int y) { ←
    return (x+y);
}
```

dichiarazione

"call by value"

definizione

valore.cpp
PassByValue.cpp

Passaggio per Valore (2)

```
# include <iostream>
int somma (int , int );
void main() {
    int a, b, res;
    a = 5;
    b = 10;
    res = somma(a,b);
    cout << a << b << res;
}
int somma (int x, int y) {
    x++; y++;
    return (x+y);}
```

**&a !=&x
&b !=&y**

Uso non corretto

Passaggio per Riferimento (1)

■ Per “puntatore a” (*): tipico C e C++

```
int somma(int * , int * );
void main(){
    int a, b, res;
    a = 5;
    b = 10;
    res = somma (&a,&b);
}
int somma (int *x, int *y) {
    (*x) = (*x) + (*y);
    return (*x); }
```

“call by reference (1)”

Uso esplicito dei
puntatori e della loro
”aritmetica”

Passaggio per Riferimento (2)

■ Per “indirizzo a” (&): solo C++

```
int somma(int& , int& );
void main() {
    int a, b, res;
    a = 5;
    b = 10;
    res = somma(a,b);
}
int somma (int& x, int& y) {
    x=x+y;
    return (x); }
```

“call by reference (2)”

int & : riferimento all’indirizzo
della variabile di tipo int
Elimina l’uso esplicito dei
puntatori !!

Quanto vale s?

```
int modifica(int s) {
    s++;
    return s;
}
main(void) {
    int s=1;
    modifica(s);
    cout << "s=" << s << endl;
}
```

“variabile locale”

```
int s;
int modifica() {
    s++;
    return s;
}
main(void) {
    s=1;
    modifica();
    cout << "s=" << s << endl;
}
```

“variabile globale”

Variabili globali

Le variabili globali sono "cattive"
(almeno quanto il GOTO)!

- Perché violano il principio della località della informazione (Principio di "Information hiding")
- E' impossibile gestire correttamente progetti "grossi" nei quali si faccia uso di variabili globali.
- Principio del NEED TO KNOW: ciascuno deve avere TUTTE e SOLO le informazioni che servono a svolgere il compito affidato

Principi di Parna

Il committente di una funzione deve dare a chi la implementa tutte le informazioni necessarie a realizzare la funzione,
e NULLA PIÙ

Chi implementa una funzione deve dare a chi la utilizza tutte le informazioni necessarie ad usare la funzione,
e NULLA PIÙ

Scope delle variabili

Variabili globali: Nel seguente esempio `a` e' una variabile globale.
Il suo valore è visibile a tutte le funzioni

ATTENZIONE! EVITARE le variabili globali (Information Hiding)

```
int a=5;
void f() {
    a=a+1;
    cout << "a in f:" << a << endl;
    return;
}
main() {
    cout << "a in main:" << a << endl;
    f();
    cout << "a in main:" << a << endl;
}
```

```
Output:
a in main: 5
a in f: 6
a in main: 6
```

globali.cpp

Scope delle variabili

Variabili locali (automatiche): Nel seguente esempio `a` e' una variabile automatica per la funzione f. Il suo valore è locale ad f.

```
int a=5;
void f() {
    int a=2, b=4;
    cout << "(a,b) in f:" << a << b << endl ;
    return;
}
main() {
    int b=6;
    cout << "(a,b) in main:" << a << b << endl;
    f();
    cout << "(a,b) in main:" << a << b << endl; }
```

locali.cpp

Output

Output:

```
(a,b) in main: (5,6)
(a,b) in f:    (2,4)
(a,b) in main: (5,6)
```

ATTENZIONE! Le variabili automatiche SCHERMANO le variabili globali.

Scope delle variabili

Variabili statiche "interne"

```
void f (int n) {
    int b; static int a;
    if (n == 0) {a = 1 ; b = 1;}
    else {a=a+1; b=b+1;}
    cout << "(a,b) in f:" << a << b << endl);
}

void g(void) {int x=99,y=100,z=101;}

main() {
    int a=5,b=5;
    f(0); g(); f(1);
    cout << "(a,b) in main:" << a << b << endl);}

statiche.cpp
Static.cpp
```

Output

Output:

```
(a,b) in f:    (1,1)
(a,b) in f:    (2,?)
(a,b) in main: (5,5)
```

ATTENZIONE: ? significa che il valore di b alla seconda iterazione NON è in generale) predicibile!

Scope e definizioni

- In C++ si possono definire variabili in un qualsiasi punto di uno scope

```
int main() {
    //.. { // Begin a new scope
    int q = 0; // C requires definitions here
    //..
    // Define at point of use:
    for(int i = 0; i < 100; i++) {
        q++; // q comes from a larger scope
        // Definition at the end of the scope:
        int p = 12;
    }
    int p = 1; // A different p
} // End scope containing q & outer p
```

Variabili statiche e automatiche

- Una variabile in C++ può essere statica o automatica
 - statica: viene allocata la memoria necessaria per il tipo di variabile all'inizio del programma e la variabile esiste per tutta la durata del programma

```
static char ch[100];
```
 - automatica: la memoria è allocata ogni volta che l'esecuzione del programma trova la definizione ed esiste solo all'interno del blocco che contiene la dichiarazione

```
char ch[100];
```
- In entrambi i casi la dimensione deve essere definita esplicitamente nel programma: allocazione statica della memoria

Scope tra più file

- Variabili globali sono visibili tra file diversi (si accede con extern)

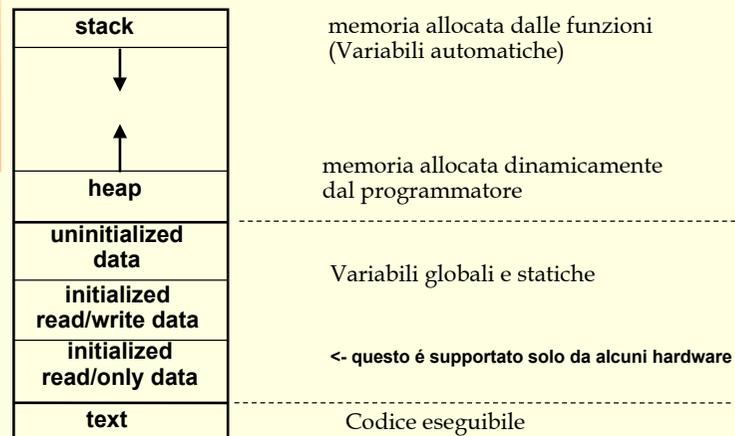
```
Global.cpp  
Global2.cpp
```

- La dichiarazione Static può essere usata per indicare una variabile globale all'interno di un file

```
FileStatic.cpp  
FileStatic2.cpp
```

- External vs Internal linkage

Il modello di memoria



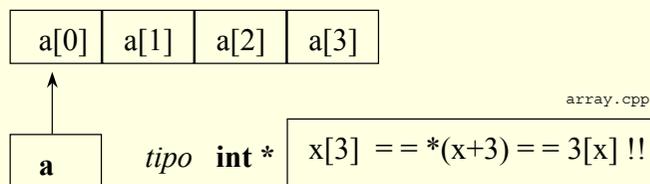
Array: Vettori e Matrici

- “oggetti” che permettono di descrivere un insieme di variabili
- gli array sono composti da:
 - un tipo
 - un nome
 - una dimensione
 - un indice

```
... {  
    int i; float y;  
    float a[100] ;  
    for (i=0; i<10; ,i++) {  
        a[i] = 0.0};  
    y = a[99];  
}
```

Importante !!

- Gli indici vanno da 0 fino alla dimensione (n - 1)
- Gli indici possono essere solo numeri interi
- Un accesso fuori dal range degli indici non genera alcun avvertimento nel corso dell'esecuzione (conseguenze catastrofiche)
- Il nome di un array è in realtà un puntatore alla sua prima cella



Array Multidimensionali

- Allo stesso modo degli array monodimensionali, possono essere creati anche array multidimensionali.
- In questo caso il nome dell'array è un puntatore ad un puntatore ad un ... (n volte) puntatore all'oggetto interessato.
- Il loro uso è però delicato e perciò limitato (analisi numerica..). Per altre applicazioni meglio usare le Struct e le Classi.

Battaglia Navale

- `int a[4][4], i, j;`
- `a[2][3] = 9;`

j ->

	0	1	2	3
0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
2	a[2][0]	a[2][1]	a[2][2]	a[2][3] = 9
3	a[3][0]	a[3][1]	a[3][2]	a[3][3]

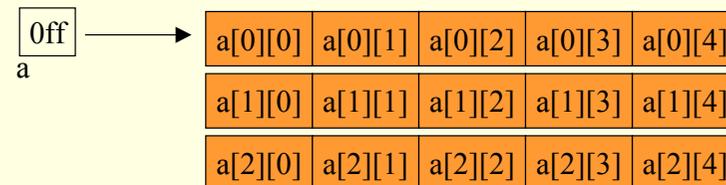
i ↓

Matrici con alloc. statica

```
char a[3][5]; char b;
```

```
...
a[i][j] = b;
```

il compilatore crea del codice macchina che eseguirà la seguente istruzione: "all'indirizzo **a** aggiungi **5** volte **i**, quindi aggiungi **j** e ritorna il valore contenuto nell'indirizzo ottenuto"



Matrici con alloc. Statica (2)

- nelle chiamate a funzione, questo metodo di indirizzamento crea problemi:

```
...
int a[3][5]; int i;
i = calcola (a);
...
int calcola ( int *m)
{ int i; int j; int sum = 0;
  ..
  sum = sum + m[i][j];
  ..
}
```

Il compilatore non sa per quanto deve moltiplicare **i**

Esercizio: Cerca il Min e Max

Scrivere un programma che inserisca N x M numeri in una matrice N x M e trovi il minimo ed il massimo dei valori