Corso di Laboratorio 2 – Programmazione C++

Silvia Arcelli

13 Ottobre 2014

Array Statici

- Una sequenza ordinata di variabili dello stesso tipo è un array. Un array è una variabile di tipo strutturato, al contrario dei tipi scalari predefiniti che sono detti tipi elementari o "atomici ".

- Gli elementi sono indicizzati partendo da zero fino a n-1, dove n è la dimensione dell'array, e sono memorizzati in sequenza nella parte di memoria denominata "stack". Si ha accesso diretto al singolo elemento tramite l'indice dell'array. non c'è protezione sugli indici (attenzione a non superare n-1!).

  Type name[size]= {init_list};

- Type è un qualunque tipo predefinito, o creato dall'utente. Si riferisce al tipo degli elementi contenuti nell'array; size è la dimensione dell'array, e deve essere una espressione intera costante.

- La lista di inizializzazione è opzionale e contiene una lista di valori separati da virgole che corrispondono o convertono a Type, il cui numero non deve superare la dimensione dell'array (obbligatoria se l'array è dichiarato const) .

Array Statici

- Esempi di dichiarazioni di array:

  int numeri[100];//ok

  char string[10]; //ok

  int labels[]; // illegale, size non specificata

  char st[] ={'c','i','a','o','\0'}; //ok, size è data dalla init_list

  char st[] ="ciao"; //equivalente

  int labels[5] ={1,5,3}; // ok, solo 3 elementi inizializzati

Si considerino le seguenti righe di codice:

Test esame 14/07/2014

```
int n = 10;
int  array[n];
```

Quale delle seguenti affermazioni è vera?

A) Le righe di codice produrranno un errore in fase di compilazione/linking
B) Le righe di codice produrranno un errore in fase di esecuzione
C) Le righe di codice non produrranno nessun errore e la dimensione dell' array è 10
D) Le righe di codice non produrranno nessun errore e la dimensione dell'array varierà con n all'interno del codice

Array Statici

- La dimensione dell'array deve essere valutabile al compile time:

```
const int len=12; // compile time, const integer
int buffer[len]; //ok
```

ma:

```
int val=12; // non const integer
const int len=val; // const int, run time
int buffer[len]; // illegale!!!!
```

NB: dimensione non nota al compile time è consentita in estensioni gcc e nel nuovo standard C++ 11 (non raccomandato).

Puntatori

- Un puntatore è una variabile il cui valore corrisponde ad un indirizzo di memoria.
- Possono essere di qualunque tipo (predefinito e di ogni tipo nuovo creato dall'utente), incluso il puntatore a void (utile per puntare a dati di tipo diverso).

```
Type *ptr= init;
```

- Permettono l'accesso indiretto a locazioni di memoria. Type si riferisce al tipo della variabile il cui valore è contenuto nella locazione di memoria puntata. Possono puntare allo stesso indirizzo puntato da un altro puntatore.

Puntatori

- Si possono definire array di puntatori:

Type *ptr[size1][size2][sizeN]= {init_list}; //array di puntatori

- I puntatori quando utilizzati devono sempre corrispondere ad un indirizzo di memoria valido

- Buona norma inizializzarli a 0 (o NULL) (che non è mai un indirizzo di memoria valido, e fare un controllo sul valore del puntatore), altrimenti possono verificarsi comportamenti indefiniti

- La loro dimensione in byte dipende dalla piattaforma (in genere è quella di un intero)

Puntatori

Date le seguenti linee di codice:

A) char *a; a = new char[10];
B) char a[10];
C) char *a;
D) char *a,b; a = &b;
in quale caso una successiva linea "*a = 'c';" generera' un errore?

Dato il seguente segmento di codice:

```
double n=5.6;
int *ptr= &n;
cout << *ptr <<  endl;
```

Test esame 10/06/2014

Il codice:

A) compila ed esegue correttamente, stampando su standard output il valore 5.2
B) compila ed esegue correttamente, operando un troncamento e stampando su s.o. il valore 5
C) produce un errore in fase di esecuzione
D) produce un errore in fase di compilazione

Puntatori

- Per inizializzare un puntatore si può utilizzare l'operatore di indirizzo &:

```
int i=3;
double n=5;
int *ptr=&i; //ptr è inizializzato all'indirizzo della variabile i
int *ptr2= i; //illegale, i non è un indirizzo
int *ptr3= &n; //illegale n non è dello stesso tipo di ptr3
```

- Per agire sul contenuto della cella indirizzata da un puntatore si usa l'operatore di indirection(o dereferenziazione), *:

```
int j=*&i; //j è inizializzata a 3
(*ptr)=9;  //sostituisce 9 a 3 nel valore di i
```

- Puntatori di puntatori:

```
int i=3;
int *ptr=&i; //ptr è inizializzato all'indirizzo della variabile i
int **pptr=&ptr; //puntatore di puntatore
```

Puntatori

I puntatori possono essere const, puntare a tipi const, o entrambe le cose:

Puntatori a costanti:

```
const char pippo =‘A’; // pippo è un const char
const char pluto=‘B’; // anche pluto
char minnie=‘C’;
const char *ptr=&pippo// OK
ptr=&pluto//OK, ptr non è const!
ptr=&minnie//illegale, ptr è un puntatore a un const char
```

Puntatori costanti:

```
char minnie=‘C’;
char minnie2=‘D’;
char * const ptr=&minnie// OK
ptr=&minnie2// illegale, riassegnazione di ptr che è un puntatore const
```

Puntatori

Regole di conversione per i puntatori:

- Ogni genere di puntatore può essere convertito in un puntatore a void

```
char a='A'; char* ch=&a;  //puntatore a char;
void *generic_p; // puntatore a void;
generic_p=ch;//ok, cast automatico da *char a void
```

- Un puntatore generico a void non è compatibile con un puntatore di tipo arbitrario se non attraverso un cast esplicito:

```
ch=generic_p;// cast da *void a *char, illegale
ch=(char*)generic_p;// cast esplicito, ok
```

Puntatori

- Incrementare di 1 un puntatore a int significa aggiungere sizeof(int) al valore del puntatore (algebra dei puntatori).

- In generale, dato un puntatore a Type questa operazione implica aggiungere sizeof(Type) al puntatore.

- Accesso alla memoria e equivalenza fra Array e Puntatori: il nome di un array è un puntatore associato all'indirizzo della prima cella di memoria occupata dall'array:

```
int buffer[100];
int *ptr=buffer;
buffer[0]=0; // equivale a *ptr=0 o ptr[0]=0;
*(ptr+2) =1; // equivale a buffer[2]=1 o ptr[2]=1;
```

a[i]  *(a+i)

Reference

- Per accedere al contenuto dei dati creati dal programma, abbiamo visto che si utilizzano o i nomi delle variabili o puntatori con indirection (*).

- Un altro modo è il reference (introdotto in C++), che permette l'accesso simbolico a locazioni di memoria già esistenti.

Type & alias_name=name; // dove name è una variabile già dichiarata

- Richiedono definizione con inizializzazione, e Type non può essere void.

```
int & pippo;//illegale, manca inizializzazione
void test;
void &pluto =test;//illegale
```

- Non allocano nuova memoria. posso definire reference a variabili const, ma reference const non hanno senso (la reference è necessariamente legata alla locazione di memoria specificata in inizializzazione, l'indirizzo è fissato)

Reference

- Sono sostanzialmente degli alias, e consentono di accedere alla memoria con gli stessi vantaggi dei puntatori ma con notazione più lineare.

```
int a=2;
int & p= a;
p++;
cout << a =" << a <<endl;
```

```
int a=2;
int*p=&a;
(*p)++;
cout << "a =" << a <<endl;
```

a=3

- L'uso più comune dei reference è nel passaggio per indirizzo di argomenti di una funzione, o nel suo valore di ritorno (qui bisogna però stare attenti che l'indirizzo che si restituisce sia ancora valido quando la funzione "ritorna").

  di fatto, nella ridefinizione degli operatori di tipi composti, è l'unica scelta.

```
enum  case{bad,good,verygood};
case operator++(case &c) {++c; return c;}
case a=bad;
++a;
```

```
enum  case{bad,good,verygood};
case *operator++(case *c) {++*c; return c;}
case a=bad;
++&a;
```

"innaturale", e non compila!

Funzioni

- Una funzione è un modulo di programma dedicato ad un compito definito (utilizzo dello stesso codice più volte nel programma)

- Una funzione in C++ necessita di una definizione e di una dichiarazione. Il compilatore C++ controlla che ogni chiamata alla funzione sia coerente con la sua dichiarazione e definizione

- La dichiarazione ha il compito di far conoscere il simbolo al modulo chiamante e ha la forma:

  Type funzione(Type par1, Type par2,...);

  cioè specifica il cosiddetto *prototipo* della funzione:
  - nome della funzione (in genere, max 32 caratteri),
  - tipo della funzione (tipo del valore restituito da return),
  - tipo, numero e ordine dei parametri in ingresso (*signature*).

Funzioni

- La definizione (che implementa il codice corrispondente al compito della funzione)ha invece il compito di definire cosa fa la funzione ed ha la seguente sintassi:

```
Type  funzione(Type par1, Type par2,...) {
//serie di statement (corpo della funzione)
return expr;
}
```

la dichiarazione deve concordare con la dichiarazione. L'istruzione return è essenziale se la funzione ritorna un valore (n.b. non deve essere necessariamente l'ultima istruzione). Nel caso di funzioni di tipo void (senza valore di ritorno, *procedure*) il return non è strettamente obbligatorio.

- In generale occorre che siano presenti sia dichiarazione che definizione della funzione. Si può omettere la dichiarazione (è sufficiente la definizione) se la funzione è chiamata dopo aver definito la funzione nel codice sorgente.

Funzioni

Esempio:

```
int mymax(int a,int b){
return a>b? a:b;  }
int main(){....
int n1=1,n2=3;
int nmax=mymax(n1,n2);
..... }
```

OK

```
int main(){....
int n1=1,n2=3;
int nmax=mymax(n1,n2);
..... }
int mymax(int a,int b){
return a>b? a:b;  }
```

NO

- Notare che occorre in qualche modo definire mymax prima del suo utilizzo (altrimenti il simbolo non è riconosciuto in fase di compilazione).

```
int mymax(int ,int );
int main(){
int n1=1,n2=3;
int nmax=mymax(n1,n2);
.... }
int mymax(int a,int b){
return a>b ? a:b;  }
```

OK

•In alternativa, occorre inserire prima la dichiarazione (il prototipo).

Funzioni-parametri di ingresso

- I parametri specificati nella definizione (e opzionalmente nella dichiarazione, in cui è richiesto solo di specificare il tipo) vengono detti parametri formali (nel caso precedente sono a e b). Quelli che vengono effettivamente passati nel corso del programma sono detti parametri attuali.

```
int n1=5, n2=3;
int val=max(n1,n2); // n1, n2  sono parametri attuali
```

- Nel C++ i parametri attuali sono passati per valore e ne viene fatta una copia locale nello stack (nello spazio di memoria riservato alla funzione). Al termine dell'esecuzione della funzione questa copia viene cancellata.

- Come conseguenza, nel C++ le function non modificano i valori dei parametri attuali nel programma chiamante.

Funzioni-parametri di ingresso

Se si vuole modificare il valore di parametri attuali (ad esempio, incrementare una variabile di 1), occorre passare i parametri per indirizzo:

- dichiarazione e definizione:

```
void increment(int *); //incrementa di 1
void increment (int *a){ ( *a)++;}
```

(per inciso, le parentesi in (*a) sono necessarie?)

- nell'unità di programma chiamante:

```
int a=1; int *p=&a;
increment(&a); // a=2
increment(p); // a=3
```

Funzioni-parametri di ingresso

- Il passaggio per indirizzo consente di avere argomenti “di output” oltre al valore di ritorno della funzione:

```
void increment(int *a, int *b){
  (*a)++;  *b+=*a; }

int main() {
  int n=5, sum=0;
  for(int i=0;i<n;)increment(&i,&sum);
  cout << "sum=" << sum << endl;
  int i=0;sum=0;
  while(sum<1.e2) increment(&i,&sum);
cout << “i=" << i << endl;
  return 0;
}
```

Calcola la somma dei primi 5 interi, sia i che sum sono aggiornati quando tornano nel main

Calcola la somma degli interi consecutivi fino a quando è maggiore di 2000

Funzioni-parametri di ingresso

Un modo alternativo, che consente di non utilizzare la notazione dei puntatori è usare i reference:

- dichiarazione/definizione (notare l'"&"):

```
void increment(int &); //incrementa di 1
void increment (int &a){a++;}
```

- Nell'unità di programma chiamante, la chiamata è identica alla chiamata per valore, ma in realtà si passano gli indirizzi attraverso l'alias definito dal reference:

```
int a=1; int *p=&a;
increment(a); // a==2
increment(*p); // a==3
```

Funzioni-parametri di ingresso

- Anche l'uso del reference permette alla funzione di modificare i parametri attuali, essendo equivalente a una chiamata per indirizzo. Ad esempio, se si vogliono scambiare i valori di due variabili:

```
void swap(int &, int &);
void swap (int &a, int & b){
 int temp=a;
 a=b;
 b=temp;
 return;}
```

Funzioni-parametri di ingresso

- L'esempio visto in precedenza utilizzando i reference:

```
void increment( int &a,  int &b){
  a++;  b+=a;
}

int main() {
  int n=5, sum=0;
  for(int i=0;i<n;)increment(i,sum);
  cout << "sum=" << sum << endl;
  return 0;
}
```

```
void increment(int *a, int *b){
  (*a)++;  *b+=*a;
}

int main() {
  int n=5, sum=0;
  for(int i=0;i<n;)increment(&i,&sum);
  cout << "sum=" << k << endl;
  return 0;
}
```

Notazione + lineare rispetto alla sintassi con puntatori.

Funzioni-parametri di ingresso

- Utilizzando il passaggio per indirizzo (sia attraverso puntatori che reference) non viene fatta una copia dei parametri attuali nello stack e si limita l'overhead della funzione. Rilevante quando i parametri di ingresso della funzione hanno una dimensione considerevole in memoria (oggetti complessi).

- Il passaggio per indirizzo consente di modificare il valore dei parametri attuali (i parametri di ingresso possono anche diventare parametri di output)

- Se non si vuole che questo avvenga, ma si vuole continuare ad usare la chiamata per reference per ragioni di efficienza, si può utilizzare la keyword const (ovviamente ogni tentativo di modificare il parametro nella funzione produce poi un errore di compilazione...)

Funzioni-Overloading

Nella chiamata ad una funzione, il compilatore controlla in primo luogo se il tipo del valore di ritorno è corretto . Quindi verifica la corrispondenza fra parametri attuali e formali secondo il seguente ordine di priorità:

- **Ricerca la corrispondenza esatta**: se esiste una versione della funzione che richiede esattamente quel tipo di parametri (o conversioni triviali..const, a->&a)
- **Promozione** degli argomenti: si utilizza (se esiste) una versione della funzione che richieda al piu` promozioni di tipo (char->int, float->double,..)
- **Conversioni standard di tipo**: si utilizza (se esiste) una versione della funzione che richieda al più conversioni di tipo standard (int ↔ double,..).
- **Conversioni definite dall'utente**: si tenta un matching con una definizione (se esiste), cercando di utilizzare conversioni di tipo definite dal programmatore (nel caso di oggetti);

Chiamata a funzione con ellissi

- Se tutti gli altri tentativi di trovare una corrispondenza falliscono, il compilatore cerca una **Corrispondenza con l'ellipses (...):** si utilizza (se esiste) una versione della funzione che accetti un qualsiasi numero e tipo di parametri (cioe` funzioni nel cui prototipo e` stato utilizzato il simbolo **...**);

- In C++ è consentito l'uso dell'ellissi senza alcun argomento, e indica una funzione che accetta un numero arbitrario di parametri:

  int fun(...)

- Invece la signature del tipo: **int fun( )** n C++ è equivalente a: **int fun(void)**

- Se nessuna di queste regole puo` essere applicata, si genera un errore in compilazione (funzione non definita!).

Funzioni-Overloading

- Il fatto che l'identità di una funzione sia specificata non solo dal suo nome e dal suo valore di ritorno, ma anche dalla sua *signature* apre la possibilità, nel linguaggio C++, al cosiddetto meccanismo di ***overloading*:**

- Il termine overloading (*sovraccaricamento*) nel contesto del C++ indica la possibilita` di attribuire allo stesso nome di funzione piu` implementazioni distinte. Il sovraccaricamento fa in modo che lo stesso nome sia contemporaneamente utilizzato per piu` funzioni internamente differenti , semplicemente distinguendole per *signature*.

```
int sum(int a, int b); // per somma due interi,
float sum(float a, float b); // per somma due float,
float sum(float a, int b); //  somma di un
float sum(int a, float b); // float e un intero.
```

Sono funzioni distinte!

Funzioni-Overloading

```
#include < iostream >
using namespace std;
/* .....Dichiarazione ed implementazione delle varie sum .....*/
int main(  )
{ int a = 5; int y = 10; float f = 9.5; float r = 0.5;
cout << "Sum(int, int):       "  << sum(a, y) << endl;
cout << "Sum(float, float): " << sum(f, r) << endl;
cout << "Sum(int, float):    "  << sum(a, f) << endl;
cout << "Sum(float, int):    " << sum(r, a) << endl;
return 0; }
```

- le varie versioni devono differire necessariamente nei tipi dei parametri , o che questi siano forniti in un ordine diverso.
- Il compilatore decide quale funzione chiamare in base ai parametri forniti.

Funzioni-Overloading

- il tipo di ritorno invece non è influente (non fa parte della signature!):

```
float sum(int a, float f);
 int   sum(int a, float f); // Errore, come dichiarazione ripetuta!
 int   sum(float f, int a); // scambio l'ordine, Ok!
```

- La seconda dichiarazione è errata perché, per scegliere tra la prima e la seconda versione della funzione, il compilatore si basa unicamente sui tipi dei parametri in ingresso che in questo caso coincidono. Non si può fare overloading cambiando solo il tipo di ritorno, in sostanza.

- la terza dichiarazione è OK, ora il compilatore è in grado di distinguere perche` il primo parametro anziche` essere un int e` un float.

Funzioni-Overloading

- Il compilatore segnala anche l'occorrenza di casi ambigui:

```
#include < iostream >
using namespace std;
void print(double);
void print(long);
int main( )
{
print(1L); // ok, chiama print(long)
print(1.0); // ok, chiama print(double)
print(1); // ambiguo, non sa se promuovere a long o a double

return 0; }
```

Funzioni-Argomenti di Default

- Ad ogni parametro di una funzione può essere attribuito un valore di default (si può chiamare la funzione tralasciando quei parametri per cui il default è appropriato).
- E' possibile assegnare un default solo i parametri più a destra della calling sequence (altrimenti la signature è ambigua). L'assegnazione può essere fatta sia nella definizione che nella dichiarazione della funzione*.

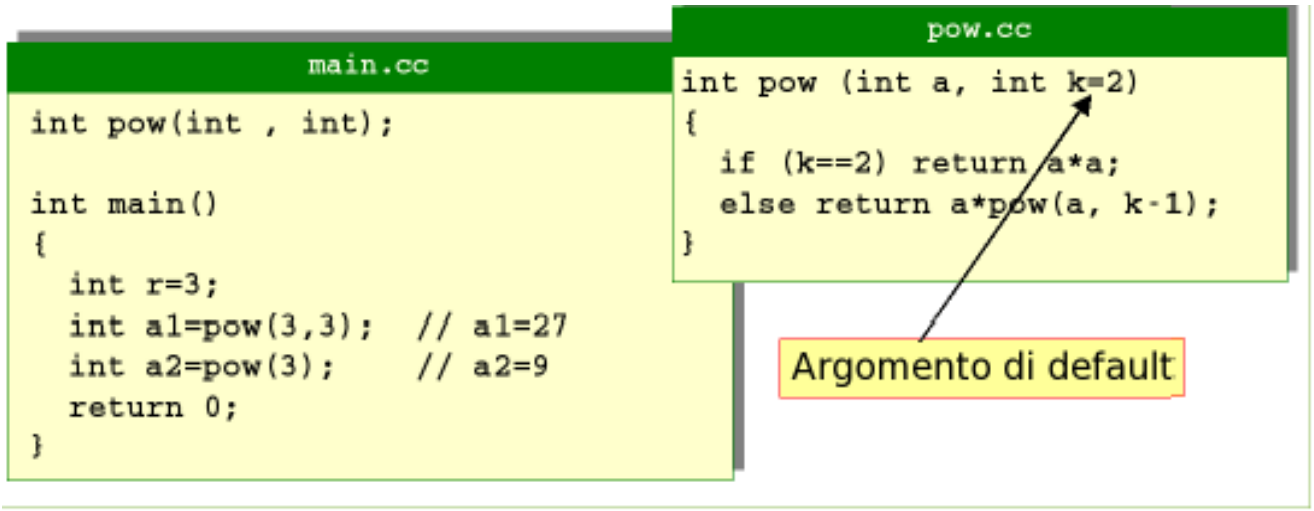
main.cc

```
int pow(int , int);

int main()
{
  int r=3;
  int a1=pow(3,3);   // a1=27
  int a2=pow(3);     // a2=9
  return 0;
}
```

pow.cc

```
int pow (int a, int k=2)
{
  if (k==2) return a*a;
  else return a*pow(a, k-1);
}
```



*(in questo esempio è fatta nella definizione perchè nella dichiarazione non sono stati nominati i parametri).

Funzioni-Argomenti di Default

All'interno di un programma e' definita la seguente funzione:

double foo(int a, double b=3; double c){ return a*b*c;}

Test esame 10/06/2014

la funzione viene cosi' invocata all'interno del programma:

double result = foo(1,2);
cout << " result = " << result << endl;

quale dei seguenti casi si verifica:
A) il programma esegue e stampa su standard output il valore 6
B)il programma esegue e stampa un risultato indeterminato
C) il programma fallisce in fase di esecuzione
D)il programma fallisce in fase di compilazione

Funzioni-Argomenti di Default

- Così come non sono ammesse funzioni con overload che differiscano solo per il tipo del valore di ritorno, non sono ammesse funzioni che differiscano solo per argomenti di default. Ad esempio la presenza simultanea di due definizioni del tipo:

fun(int){....} e fun(int, double=0.0){....};

non sono accettate, in quanto generano ambiguità: in una chiamata tipo fun(n), il programma non saprebbe se trasferirsi alla prima funzione (un solo argomento), oppure alla seconda (due argomenti, ma il secondo può essere omesso per *default*).

In una classe, un costruttore parametrico dove per tutti gli argomenti sia stato specificato un default è il costruttore di default.

Funzioni inline

- Semplici funzioni come increment possono essere definite inline:

```
inline void increment (int *a){ ( *a)++;}
```

- la keyword inline va premessa nella definizione (il compilatore necessita di conoscere il codice contenuto nel corpo nella funzione!). Se la funzione è un metodo di una classe, l'implementazione del metodo va nella dichiarazione di classe e anche se non si specifica inline viene interpretata come inline.

- In questo modo si suggerisce al compilatore di sostituire il codice eseguibile della funzione in tutti i punti del programma in cui è chiamato. Il compilatore decide autonomamente se convertire la funzione in inline (in generale, funzioni di max 4-5 statement).

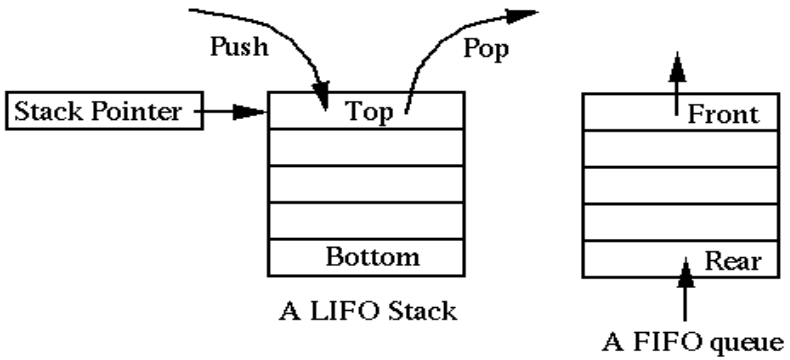
Funzioni inline

- Come in una macro in C, anche in questo caso non si ha l'overhead tipico di una chiamata a funzione, non viene fatta una copia locale dei parametri attuali e si aumenta l'efficienza del programma (anche se la sostituzione ne aumenta la dimensione, fare attenzione...). **Ideale nel caso di funzioni chiamate in cicli.**
- Preferibile rispetto alle MACRO (type checking e si evitano errori dovuti alla sostituzione testuale).

```
#include<iostream>
#define MAX(a,b) (a>b? a :b)
using namespace std;
inline int max(int a,int b){ return a>b ? a: b;}
int main(void){
 int i=8, k=5;
//cout <<max(i++,k) <<endl; //stampa 8
cout <<MAX(i++,k) <<endl;//stampa 9!!!!
return 0;}
```

Funzioni recursive

- In C/C++ possono essere definite funzioni ricorsive utilizzando il meccanismo di trasmissione dei parametri fra programma chiamante e funzione (stack di tipo LIFO):



- quando una funzione A chiama una funzione B, sistema in un'area di memoria, detta appunto stack della funzione, un *pacchetto* di dati, comprendenti:
  - le variabili automatiche di B;
  - gli argomenti di B in cui copia i valori trasmessi da A;
  - l'indirizzo di rientro in A

  } *Record di Attivazione*
- la funzione B utilizza tale *pacchetto* e, se a sua volta chiama un'altra funzione C (che può essere se stessa), sistema nell'area stack un altro *pacchetto*, "impilato" sopra il precedente

Funzioni ricorsive

- Esempi di funzioni ricorsive : fattoriale, elevamento a potenza, massimo comun divisore,...

```
int fattoriale (int n){
if (n<=0 ) return 1;
return n*fattoriale(n-1);
}
```

```
int mcd (int x, int y)
{if (y==0)return x;
return mcd(y,x%y);}
```

- Notare che deve essere sempre presente una condizione che faccia terminare la sequenza di chiamate annidate (evitare lo *stack overflow*). Nel caso del fattoriale, n==0
- Occorre evitare valori di ingresso che rendano impossibili la condizione di terminazione. Nel caso del fattoriale, n<0

Funzioni ricorsive

- Alternativamente si può scrivere in maniera iterativa:

Versione iterativa

```
int fattoriale (int n){
int k; for(k=1;n>0; n--)k*=n;
return k;
}
```

Versione ricorsiva

```
int fattoriale (int n){
if (n<=0) return 1;
return n*fattoriale(n-1);
}
```

- Anche se in questo caso non si apprezza, un algoritmo scritto in forma ricorsiva è più compatto ed elegante rispetto alla forma iterativa, ma comporta in generale un maggior dispendio di memoria e un tempo di esecuzione più lungo. Se le prestazioni del programma sono prioritarie è meglio usare la versione iterativa.