Programmazione Avanzata per il Calcolo Scientifico
Lezione N. 2

Luca Formaggia

MOX
Dipartimento di Matematica "F. Brioschi"
Politecnico di Milano

A.A. 2005/2006

Introduzione al C++(parte 2)

- typedef
- Arrays
- Puntatori
- auto_ptr e smart pointers
- vector<T>
- Iteratori
- Sequence
- Organizzazione della memoria
- Puntatori multipli
- Referenze
- Stringhe
- Costanti
- enumeratori
- Funzioni
- Passaggio per valore e per referenza

typedef: parte 1

Il comando `typedef` crea degli *alias* ai tipi

```
typedef unsigned int Uint;
typedef vector<float> Vf; typedef double Real;
typedef double myvect[20];
...
Vf v;// v e‘ un vettore di float
myvect z// z è un array di 20 double
```

Regola d'uso: se togliete typedef avete la dichiarazione di una variabile:
Il typedef semplifica la scrittura (e lettura) di un codice: `typedef vector<vector<double*» Vvdb`
Dichiarare un `typedef` a un array nativo è possibile ma sconsigliato.

Gli array

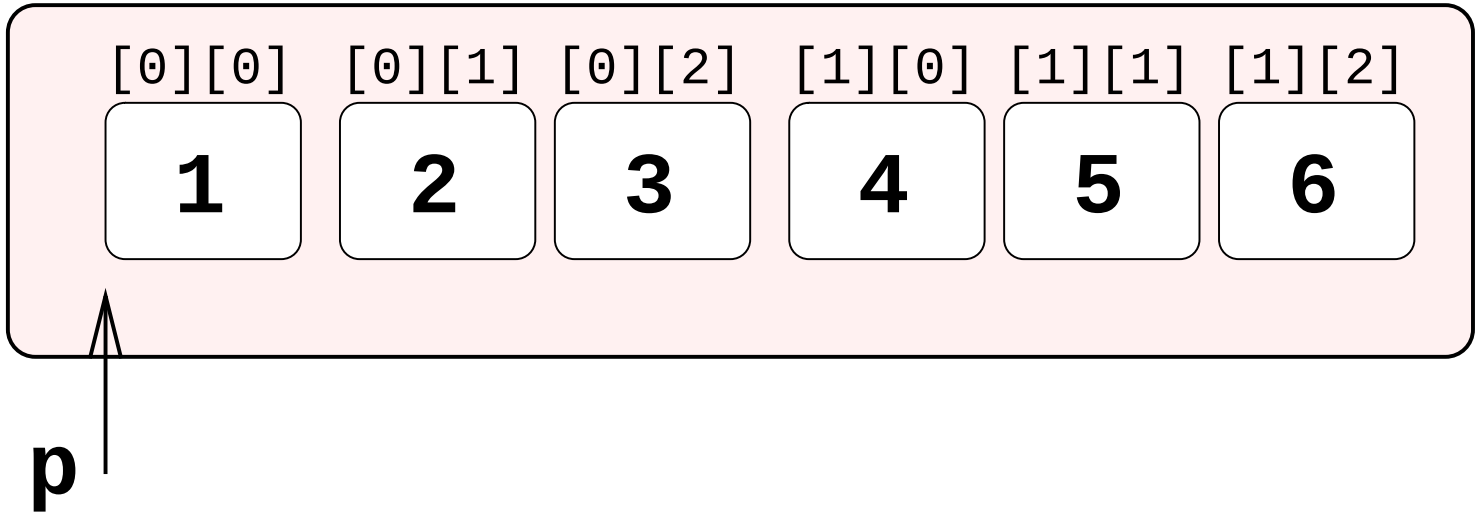
Gli array nativi in C++ sono dichiarati nel modo seguente

```
float a[4]; //un array di 4 elementi float
double b[2][3];// un array 2x3 di double
int c[2][2]={{1,2},{7,8}};// inizializzazione
int a=c[0][1]// accesso a un elemento
float p[]={1.0, 2., 4.5}; //dimensionamento implicito
```

Organizzazione degli array

Questi sono array di dimensione predefinita. Il C++ ordina gli array multidimensionali per riga.

```
int p[2][3]={{1,2,3},{4,5,6}}
```



Gli array visti fin qui sono array a dimensionamento statico. E' possibile avere dimensionamento indefinito o dimensionamento dinamico (run time). Il primo è possibile solo per gli argomenti di una *function*:

```
int myfunc(float a[][2]);
```

Questa funzione ha come argomento un array bidimensionale di `float` la cui seconda dimensione è 2 mentre la prima non è specificata. In generale, solo la prima dimensione può essere omessa.

Il dimensionamento dinamico di array nativi utilizza l'operatore `new []` verrà presentato insieme ai puntatori.

Array automatici

Il compilatore g++ ha come estensione la possibilità di usare *array automatici:*

```
double myfun(int n, ...){
double a[n] // array automatico
...}
```

ATTENZIONE: è una estensione NON PORTABILE. Lo standard C++ NON prevede array automatici

Puntatori

I puntatori sono delle variabili che possono contenere l'*indirizzo* (in memoria) di una altra variabile o funzione. Il loro uso è sopratutto legato a array a dimensionamento variabile, alla implementazione del polimorfismo o alla gestione di funzioni argomento di funzioni.
Il linguaggio C++ si sta evolvendo nella direzione di ridurre l'uso dei puntatori nativi. Ciò è possibile usando *smart pointers* per memorizzare l'indirizzo di una variabile, *contenitori STL* al posto di *array*, *funtori* al posto di puntatori a funzione (come vedremo in seguito).
Tuttavia l'uso dei puntatori è ancora importante.

Esempi di uso di puntatori

```
int* pi(0) ; // puntatore a int inizializzato nullo
char** ppc ; // puntatore a puntatore a carattere
int* ap[15]; // array di 15 puntatori a int
int (*fp)(char*); // puntatore a una funzione
// che prende un char* come argomento e restituisce un int
int f; pi=&f';//ora p punta a f (*pi == f) ;// il risultato
è 'true'
```

In C++ il valore 0 indica il *puntatore nullo*. L'*operatore (unario) di dereferenziamento* (dereferencing operator) * restituisce il contenuto della variabile indirizzata dal puntatore, mentre l'*operatore (unario) d'indirizzamento* (&) estrae l'indirizzo di una variabile.

Puntatori ed array

Puntatori ed array sono concetti collegati

```
double * pa;
pa=new double[10];//ora pa punta a un array di 10 elementi
pa[3]=9.0;
...
delete []pa;
```

L'operatore `new` (nella sua forma standard) richiede memoria al sistema operativo, nelle tre forme

`T* new T`	Puntatore a un elemento di tipo `T`
`T* new T(z)`	Puntatore a un elemento di tipo `T` l'oggetto è inizializzato al valore `z`
`T* new T[m]`	Puntatore a m elementi di tipo `T`

New e delete

L'operatore `delete` applicato a un puntatore restituisce al sistema la memoria dell'elemento "puntato". Se si tratta di un array occorre usare `delete[]`. Altrimenti si restituisce al sistema solo primo elemento dell'array!

Regola d'oro: Ogni `new` deve avere un `delete` e ogni `new []` un `delete []`.

Un'altra buona regola: A meno di casi particolari (che vanno ben documentati) il responsabile della creazione di un oggetto è anche responsabile della sua cancellazione.

Puntatori intelligenti (smart pointers)

La standard library fornisce un classe che permette di implementare il paragigma al massimo un puntatore per oggetto. Si chiama `auto_ptr` e per usarla occorre includere l'header `<memory>`.

La libreria *boost* `www.boost.org` aggunge altri puntatori intelligenti. In particolare `shared_pointer` implementa il paradigma L'ultimo puntatore cancellato cancella l'oggetto

Ne parleremo più avanti nel corso.

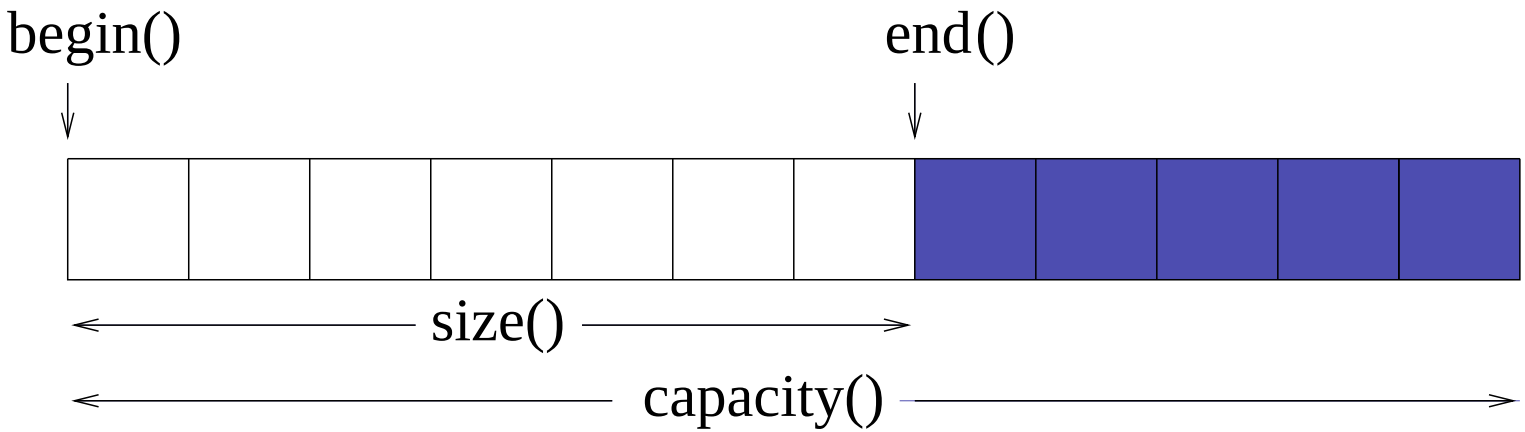
vector<T>

La standard library mette a disposizione dei contenitori generici. Qui presentiamo il più semplice (e tra i più utili), std::vector<T>. È un esempio di class template (di cui parleremo estensivamente più avanti). È sostanzialmente un array monodimensionale con memoria dinamica e possibilità di dimensionamento automatico. Richiede l'header `<vector>`.

Caratteristiche	Accesso random	O(1)
	Aggiunta/canc. alla fine	O(1)[1]
	Aggunta/canc.	O(N)

[1] Se la capacità è adeguata

Organizzazione interna



L'area contenente i dati viene allocata dinamicamente. La *capacità* rappresenta l'ampiezza di tale area. La *dimensione* (size) si riferisce alla porzione effettivamente occupata. In generale, la capacità viene solo *aumentata* in modo automatico.
begin() e end() sono gli *iteratori* al primo e all'ultimo elemento+1.

Esempi

```
vector<float> a;//creo un vettore vuoto
```

Sia *size* che *capacity* è 0.

```
vector<float> a(10);//creo un vettore con 10 elementi
```

Gli elementi sono qui costruiti con il costruttore di default `float()` che istanzia l'oggetto ma non lo inizializza (in genere). `size()` è pari a 10, `capacity()` è ≥ 10 (probabilmente 10).

```
vector<float> a(10,3.14);//vettore inizializzato a 3.14
```

Gli elementi sono costruiti usando `float(3.14)` (costruttore di copia).

push_back(T const & value)

Il metodo `push_back(value)` inserisce `value` alla fine (back) del vettore. L'area di memoria è gestita dall'algoritmo seguente, dove *size* è la dimensione prima dell'inserimento,

- Se *size*+1<*capacity*, aggiunge l'elemento e *size*=*size*+1;
- altrimenti
  a. alloca un area di memoria 2 volte la capacità corrente;
  b. copia gli elementi in tale area, che diverrà la nuova area di memoria;
  c. aggiunge il nuovo elemento alla fine del vettore.

Indirizzamento

Gli elementi di un `vector` possono essere indirizzati usando l'operatore `[]` o il *metodo* `at()`. Il secondo lancia (throw) un'eccezione nel caso di indice fuori dall'intervallo $[0, size()[$ (*range_error*).

```
vector<double> a;
b=a[5]; //Errore
c=a.at(5)//Errore, il programma abortisce
// (a meno che non si catturi l'eccezione)
```

Riservazione, prego

```
vector<float>a;
for (i=0;i<1000,++i)a.push_back(i*i);

vector<float>c;
c.reserve(1000);
for (i=0;i<1000,++i)c.push_back(i*i);

vector<float>d;
d.resize(1000);
for (i=0;i<1000,++i)c[i]=i*i;
```

Quale dei tre schemi è più efficiente?

Quello per il vettore c, in quanto: (1) non si rialloca memoria durante il riempimento (caso di a) e (2) non usa il costruttore di default+ assegnazione (caso di d).

`resize` *costruisce* il vettore della dimensione assegnata usando, quando necessario, il costruttore di default per gli elementi.

Ridurre la capacità di un vettore

Può essere opportuno ridurre la capacità di un vettore fino alla dimensione effettivamente usata. Non c'è un metodo apposito. Ma si può fare cosi

```
vector<double>a;
...//
{// Creo un vettore temporaneo che contiene
// solo gli elementi di a
vector<double> tmp(a);//
// scambio tmp con a
a.swap(tmp);
// tmp e‘ distrutto quando si esce dallo scope
}
```

Si noti le { } per definire uno scope per la variabile temporanea `tmp`. Alla fine a.size() = a.capacity() (o quasi).
La parte tra graffe può semplificare in

```
vector<double>(a).swap(a);
```

Annientare il contenuto di un vettore

```
vector<double>a;
...
//voglio 'cancellare' il contenuto di a
a.clear();//così però la capacità è invariata
vector<double>(a).swap(a);//capacity() ora è 0
```

Iteratori

Gli iteratori sono un modo di accedere in modo *uniforme* ed efficiente a tutti i contenitori della STL.

```
vector<double>a;
typedef vector<double>::iterator ivd;
...
for (ivd i=a.begin(); ivd!=a.end();++i)*i=10.56;
```

`begin()` e `end()` restituiscono un iteratore, che punta rispettivamente al primo e all'ultimo (+1) elemento del vettore.
L'iteratore può essere *deferenziato* con l'operatore `*`, che restituisce l'elemento associato all'iteratore.
Il test `i!=a.end()` assicura che siamo ancora all'interno dell'intervallo di iteratori associati a un elemento del vettore.

Si poteva anche operare in modo più classico:

```
...
for (int i=0; i!=a.size();++i)a[i]=10.56;
```

Usare gli iteratori però:

- È più efficiente: `a[i]` $\longrightarrow$ `*(&a[0] + i)`, mentre l'iteratore accede direttamente l'elemento (ma i metodi `begin()` e `end()` ha un costo) ;
- È usabile anche con altri contenitori: l'operatore `[]` esiste solo per i `vector`.

Sono definiti per gli iteratori gli operatori di somma, sottrazione e confronto: `i+1` è l'iteratore associato all'elemento successivo a quello associato a `i` (purchè `i+1` $<$ `end()`).

Una nota importante

Gli iteratori a un vettore sono invalidati ogni volta che l'area di memoria usata viene modificata, per esempio per inserire un nuovo valore:

```
it=a.begin();
v=a[5];// OK
a[2]=-7.6;// OK
a.push_back(7.8);  l'area di memoria del vettore può essere
cambiata*
c=*it; //NOOO!
```

const_iterator

Un const_iterator (iteratore costante) è un iteratore il cui oggetto può essere acceduto solo in lettura e non puo' quindi essere modificato.

```
vector<float> a;
....
const_iterator b(a.begin());
*b=5.8;// Errore
```

Algebra degli iteratori (il caso di `vector<T>`)

Gli iteratori a `vector<T>` ammettono le operazioni seguenti

- Post/pre incremento/decremento. Permettono di spostarsi all'elemento adicente. Sono valide solo se l'oggetto referenziato esiste o se l'iteratore è pari a `end()` dopo un incremento.
- Addizione e sottrazione di un intero. Ci si sposta nel vettore. Vale la considerazione di cui sopra.
- Confronto. `i1<i2` se `i1` e `i2` sono validi iteratori e se referenziano due elementi del vettore con indice crescente.

Sequence

Una sequenza (*sequence*) è un area di memoria definita da due validi iteratori (o puntatori) `i1` e `i2` tali che `i1<=i2`.
Vi sono molte metodi di contenitori della standard library che operano su sequenze. sono in genere da preferirsi rispetto a operare sulle singole componenti.

Esempio

Dati due vettori `v1` e `v2` si vuole assegnare al primo la seconda metà del secondo (assumiamo per semplicità che `v2` abbia un numero pari di elementi).
Modo consigliato: usare assign

```
v1.assign(v2.begin()+v2.size()/2,v2.end());
```

Modo sconsigliato:

```
int j=0; for(int i=v2.size()/2;i<v2.size();++i,++j)
v1[j]=v2[i];
```

```
int i; double a;
float d[2]={1.,2.};
class Vector;
```

STACK

```
new double[10];
new float[m];
```

HEAP

Nello stack prendono posto tutte le variabili la cui dimensione è nota in fase di compilazione: variabili statiche e variabili automatiche.

Nella heap prendono posto le variabili allocate "run time" attraverso l'operatore `new`.

E se non ci fosse memoria sufficiente?

Nel caso il sistema operativo non fosse in grado di fornire la memoria richiesta `new` restituisce il puntatore nullo.

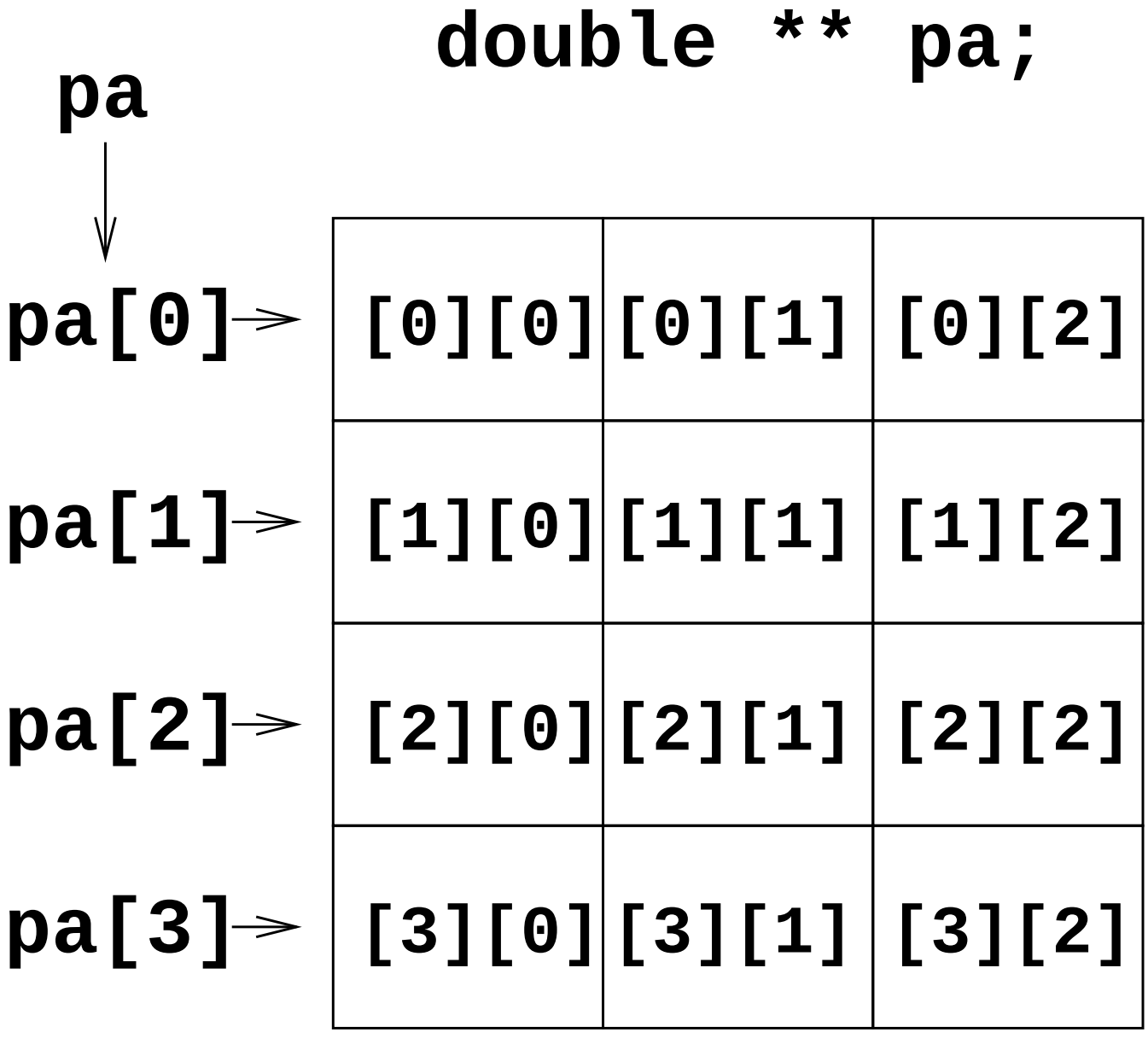
```
double * pa;
if(!(pa=new double[100])) {
cout«”Memoria insufficiente”;
call abort();
}...
```

Il comando abort() interrompe il programma (una scelta un pò brutale, ma talvolta inevitabile).

Puntatori doppi

```
double ** pa;
pa= new double *[4];
for(int i=0;i<4;++i)pa[i]=new double[3](3.0);
// ora posso indirizzare pa come un array double[4][3];
for(int i=0;i<4;++i) for(int j=0;j<3;++j)pa[i][j]=i*j;
..
for(int i=0;i<4;++i) delete[] pa[i];
delete[] pa;
```

Layout della memoria



Errori comuni

```
double *p; double * t;
p=new double[10];
delete p;// delete []!
delete t;// t non è assegnato!
```

Ci sono 2 errori gravi.
Un altro errore comune consiste nel dimenticare il `delete`, causando i cosidetti memory leak.
Regole generale Limitare l'uso dei pointers, e preferire gli "smart pointer" (argomento di una prossima lezione)

Puntatori e vector<T>

Talvolta può essere utile (per esempio per compatibilità con una funzione che usa puntatori) 'interpretare' un `vector<T>` come `T *`.

```
double myf(double * x, int dim);//funzione esterna
...
vector<double> r;
...
y=myf(&r[0],r.size());
```

Attenzione: con questa tecnica si possono eseguire solo operazioni che non modifichino l'allocazione dei dati di `vector<T>`.
Nell'esempio, il trucco non si può usare se `myf` alloca memoria per `x`.

References

In una dichiarazione il simbolo & dopo il nome di un tipo (nativo o definito dagli utenti) indica una referenza (reference) al tipo:
`double b; double & a=b;`
Il simbolo a è qui un alias di b: l'istruzione `a=10.;` modifica anche il contenuto di b.
Le referenze sono utili negli *argomenti di funzione*.
Una referenza deve sempre essere inizializzata

```
double horner(double & x);
double & pippo(int i);//ATTENZIONE
double & c= pippo(3);// OK
double& a=horner(5)//NO!;

double * pz= new double;
double & z=*pz;//OK, per ora
z=5.0;// OK *pz è ora 5
delete pz;// NO z è ora ‘‘dangling’’
z=7;//FORIERO DI DISASTRI
```

Le stringe C

In C++ si può usare la stessa convenzione del C per le stringhe.

```
char * l=”Ciao mondo”;// stringa di 10 caratteri
//+ terminatore
char * ch;// puntatore a carattere
ch=new char[n];//stringa a dim. dinamico
...
delete [] ch;
```

L'header <cstring> richiama una serie di strumenti per manipolare stringhe (ereditati dal C).

Le stringe C++

L'header della STL <string> introduce la classe template string, molto più flessibile dell'equivalente C (che viene però mantenuto per compatibilità). Si riportano le caratteristiche principali (si veda esempio_string per maggiori dettagli).

```
#include<string>
std::string a("this is"); //una stringa può essere iniziali
std::string b(" a string");
std::string c=a+b;// concatenazione
std::cout«c«endl;//operatore «
c.clear();
getline(cin,c)
```

Variabili Costanti

Il C++ permette di definire delle costanti, premettendo la parola chiave `const` nella dichiarazione. Una variabile `const` deve essere assegnata.

```
double const pi=3.14159265358979;
float const e=2.7182818f;
double const Pi=atan(1.0)/4;
const unsigned ndim=3u;
```

La qualifica *const* indica che la variabile non può essere modificata.

Regole di constanza

La parola chiave `const` si associa al tipo alla sua sinistra, a meno che non compaia all'inizio. In tal caso, non essenduci nessuna indicazione di tipo alla sua sinistra, si applica a destra. Quindi `const float a` è identico a `float const a`. Nei casi più complessi si consiglia di leggere *da destra verso sinistra* ed in Inglese:

```
double const * const p
```

significa "p is a constant pointer to a constant double". Cioè nè il puntatore nè il valore "puntato" può essere modificato.

Esempi

`const double * const p`

Identico a prima: p is a constant pointer to a (constant) double.

`double const * & p`

p is a reference to a pointer to a constant double: `*p` non può essere cambiato (ma p si).

`double & const e=z`

Errore! Una referenza è di per sè const: non può essere riassegnata.

```
double const a=30;
double const * pa=&a;
double * pc=&a;//&a e‘ un const double *
double const * & pd=pa;
double const b=89.8;
double const * pb=&b;
pb=pd;
*pb=34; // NO *pb e‘ const
```

Usate const!

Non abbiate paura di `const`: è vostro amico.
Variabili che non vengono modificate vanno sempre dichiarate const:

- Il programma è di più facile lettura e manutenzione.
- Si evita di modificare per errore una variabile che si intende non modificabile.
- Il compilatore può fare molte ottimizzazioni che altrimenti non sarebbe in grado di eseguire.

const_cast<T>

Purtroppo il mondo reale è imperfetto. Potremmo avere la necessità di passare una variabile const come argomento a una funzione dove l'argomento non è stato dichiarato const sebbene la variabile non venga modificata
Occorre usare il comando `const_cast<T>(const T&)`, che restituisce una referenza alla stessa variabile ma con il *flag* const disattivato.

Esempio di uso di const_cast<T>

```
double minmod(float & a, float & b);
...
double fluxlimit(float const& ul,float const& ur)
...
l=minmod(const_cast<float>(ul),const_cast<float>(ur));
...
```

Lvalue e rvalue

È il momento di introdurre una terminologia che appare sovente nei testi (e nei messaggi del compilatore). Ne diamo una definizione sufficientemente precisa per la maggior parte dei casi.
Un lvalue è una espressione che può apparire alla sinistra di un operatore di assegnazione. A un `lvalue` è sempre associato un'area di memoria. Un rvalue può *solo* apparire a destra di una assegnazione.
Per esempio, una *espressione letterale*, p.es. `3.1415`, è un `rvalue`.

Enumeratori

```
enum bctype{Dirichlet,Neumann,Robin};// definizione
...
bctype bc=Neumann;// ..
switch(bc){
case Dirichlet:
...
break;
case Neumann:
...
break;
Default:
...
}
```

Enumeratori

Il tipo *enumerator* definito dal comando `enum` è di fatto un *integral type* (tipo assimilabile a un intero). Infatti possono essere associati dei valori interi.

```
enum bctype{Dirichlet=0,Neumann=2,Robin=4};//
```

Rispetto però all'uso di indicatori di tipo intero permettono un controllo più fine: una variabile di tipo `bccond` può assumere solo i valori dichiarati, cioè `Dirichlet`, `Neumann` o `Robin`.

Dichiarazione di funzione

Una dichiarazione di funzione consiste degli elementi seguenti:

```
return type name (argument type , argument type, ...)
const;
```

Una funzione è caratterizzata da un nome , che la identifica rispetto alle regole di visibilità, dagli argomenti, dal tipo di ritorno (return type) ed *limitatamente a funzioni membro di una classe* dall'eventuale parametro const.

nome+argomenti (+const) formano la cosidetta firma (signature) della funzione, che la identifica univocamente.

Esempio

```
double cyVolume(const double radius,const double length);
double polygArea(const vector<lati> & sides);
```

Nella dichiarazione i nomi delle variabili possono essere omessi:

```
double cylVolume(const double,const double);
double polyArea(const vector<lati> &);
```

Definizione di funzione

```
returnType name (argument, argument) const{
...
corpo (body) della funzione
...
return value
}
```

Il corpo della funzione ne definisce le funzionalità. L'istruzione `return` ritorna un valore di tipo returnType. Essa non è presente se returnType=void.

Chiamata di una funzione

Nel programma chiamante:

```
...
double R, L;
vector<double> sides;
...
double volume=cylinderVolume(R,L);
double area=polygonArea(sides);
```

Gli argomenti `R,L` e `sides` nel programma chiamante sono detti argomenti attuali (actual arguments) della funzione. I corrispondenti argomenti nella definizione della funzione sono detti argomenti formali (o locali) e sono a tutti gli effetti delle variabili locali della funzione.

Passaggio per valore e per referenza

```
int fun1(const int i, float b, double c);
float fun2(int& i, float& b, double const & c);
...
int s=fun1(5, z, r);
float g=fun2(h, z, 3.5);
```

Nella funzione `fun1` si dice che gli argomenti formali `i`, `b` e `c` sono passati per valore. Eventuali loro modifiche in `fun1` non altera il valore degli argomenti attuali corrispondenti.
Si noti come `i` sia stata dichiarata *const*. Questo vuole dire che NON può essere modificata nel corpo di `fun1`.

```
float fun2(int& i, float& b, double const & c);
```

Nel caso di `fun2` gli argomenti formali sono 'passati per referenza'. Quindi sono delle *references* agli argomenti attuali. Quindi una loro modifica in `fun2` si riflette sull'argomento attuale corrispondente.

L'uso delle referenze è un altro modo di passare valori dalla funzione al programma chiamante.

Si noti l'uso del *const*: c non può essere cambiata: essa è quindi una variabile di *input*.

Demistifichiamo il passaggio per referenza

L'espressione 'passaggio per valore' o 'passaggio per referenza' è causa di confusione. In realtà il meccanismo con cui viene gestito il passaggio degli argomenti di una funzione in C++ è sempre lo stesso!
È il tipo associato all'argomento formale che fa cambiare l'interazione della funzione con l'argomento attuale corrispondente.

Vediamo come il C++ (ma non solo!) gestisce la chiamata a una funzione.

- Le variabili locali associate agli argomenti formali della funzione vengono inizializzate con i valori degli argomenti attuali corrispondenti;
- Viene eseguito il corpo della funzione;
- Il valore di ritorno viene reso disponibile al programma chiamante eseguendo l'operazione (assegnamento, inizializzazione) prevista in tale programma.

In generale, e variabili definite nel corpo della funzione (compresi gli argomenti formali) sono *variabili locali* il cui scope è contenuto nel corpo della funzione. Vengono quindi distrutte al momento del ritorno al programma chiamante.
Fanno eccezione le variabili statiche (static variables)

Esempi

Negli esempi che seguono cercheremo di spiegare il meccanismo della chiamata di una funzione scrivendo una sorta di *codice equivalente* che rimpiazza la chiamata.

Le variabili locali della funzione saranno indicate con `nomefunzione::`, per distinguerle dalla variabili del programma chiamante.

NOTA IMPORTANTE Questi esempi vogliono solo illustrare il meccanismo di chiamata a funzione e passaggio di valori. Corrispondono SOLO da un punto di vista concettuale a quello che accade realmente.

```
double fun (const double a, const double b){
a *=b; return a*a }

...
double A(10), B(20);
double C=fun(A,B);
{ }
{ const double fun::a(A); const double fun::b(B);
fun::a *=fun::b;
double C= (fun::a*fun::a);
// nota: C appartiene allo scopo esterno
 }
```

N.B Qui a e b avrebbero dovuto essere dichiarate const!

```
double fun (const  double & a, const double & b){
a *=b; return a*a }

...
double A(10), B(20);
double C=fun(A,B);
{ }
{ const double & fun::a=A; const double & fun::b(B);
fun::a *=fun::b; //ERRORE!!
double C= (fun::a*fun::a);
// nota: C appartiene allo scopo esterno
 }
```

N.B Dopo la chiamata alla funzione `A` sarà pari a 200 in quanto `fun::a` è un *alias* di `A`. Quindi la funzione modifica non solo `C`. Qui `b` avrebbe dovuto essere dichiarata const ma non a

Ovviamente una funzione può prendere in input dei puntatori o degli array, la regola è la stessa

```
void copy (double const  * a, vector<double> & b){
for (int i=0;i<b.size();++i)b[i]=a[i]; }
...
double * A(0), vector<double> B;
...
A=new double[100];
B.resize(100);
copy(A,B);
{ }
{ double * copy::a=A; const vector<double> & copy::b(B);
for (int i=0;i<copy::b.size();++i)copy::b[i]=copy::a[i];
 }
```

Il valore del puntatore `A` viene usato per assegnare la variabile locale `copy::b`, che quindi conterrà quindi l'indirizzo dell'area assegnata ad `A`.
Il puntatore `copy::b` andrebbe dichiarato *pointer to a constant*

Regole generali

- preferire il passaggio per referenza: è più efficiente, sopratutto per dati "grandi". Si evita infatti la copia.
- Dichiarare SEMPRE const & le variabili in "input", cioè quelle che non vengono modificate dalla funzione.
- Una costante letterale (literal), es. 34.2, "abcd", può essere passata a una funzione solo per valore o come referenza costante (`const &`).
- Bisogna fare attenzione al passaggio per referenza nel caso di funzioni ricorsive.

Allocare memoria per un argomento formale

E' possibile allocare memoria per un puntatore passato come parametro di ritorno. Consideriamo una funzione che fa la lista dei nodi con una certa condizione al bordo.

```
int * pList(mesh const & mesh,int const bc){
unsigned count=0;
for(int i=0;i<mesh.numnodes();++i)
if(mesh.nodeBc(i)==bc)++count;
int * list=new int[count];...
return list; }
```

ALTAMENTE SCONSIGLIATO:causa di errori anche gravi e di difficile individuazione (tipicamente leak di memoria): chi ha la responsabilità di cancellare la lista dei nodi `list`?
La gestione dinamica della memoria è meglio farla attraverso le *classi* (cosi come avviene per `vector<T>`).

```
void pList(mesh const & mesh,vector<int> & list, int const
bc);
```

Se fosse necessario avere un allocatore di memoria particolare è

TROVATE L'ERRORE?

```
void pList(mesh const & mesh,int * list, int const bc){
unsigned count=0;
for(int i=0;i<mesh.numnodes();++i)
if(mesh.nodeBc(i)==bc)++count;
int * list=new int[count];
...
}
...
int * mylist; pList(mymesh,mylist,5);
```

vi è un errore grave!.

Overloading di funzioni

Due funzioni con *signature* differente sono di fatto due funzioni distinte. Questo è il meccanismo alla base dell'overloading di funzione.

```
float fun(const float * & a, vector<float> & b);
double fun(const double * & a, vector<double> & b);
void fun(int & a);
...
double * z, vector<double> x, int l;
g=fun(z,x);
//chiama float fun(const float * &, vector<float> & b)
...
fun(l);//chiama fun(int & a)
```

Il compilatore sceglie la *signature* che soddisfa gli argomenti nel modo migliore, tenendo conto anche delle conversioni implicite.

Argomenti di default

Nella definizione di una funzione si possono fornire argomenti di default, che devono però essere sempre quelli più a destra:

```
vector<double> crossProd(vector<double>const &,
vector<double>const &, const int ndim=2);
...
a=crossProd(c,d);//usa ndim=2
...
```

Variabili locali statiche

Le variabili definite nel corpo di una funzione sono *variabili automatiche* che vengono distrutte all'uscita della funzione (hanno uno scope locale).
Il valore di una variabile locale la cui dichiarazione è preceduta della parola chiave *static*, mantiene il valore tra diverse invocazioni della funzione.
Nota Bene: Vengono distrutte le variabili automatiche (non statiche), non una eventuale area di memoria nella heap, richiesta usando `new`. Per tale scopo occorre usare `delete`.

```
int funct(){
static bool first=true;
if(first){//fai qualcosa solo la prima volta
first=false;
}else{
//parte di codice fatta dalla seconda volta in poi
...
}return}
```

Puntatori a funzione

```
double integranda(const double & x)
...
typedef double (*pf)(const double &);
double simpson(const double a, const double b, pf const
f, const int n);
...

integral= simpson(0,3.1415, integranda,150);

//ALTERNATIVA
pf p_int=integranda;
integral= simpson(0,3.1415, p_int,150);
...
```