

Vediamo come allocare dinamicamente un vettore di double:

```
int dim;  
double *vect;  
  
...  
cout << "Dimmi la dimensione del vettore : ";  
cin >> dim;  
  
...  
vect = new double[dim];  
if (vect != NULL) {  
    int i;  
    for (i=0; i<dim; i++) {  
        vect[i] = ...  
    }  
    ...  
    delete[] vect;  
}
```

Allocazione statica di matrici

```
// Matrice allocata staticamente
double x[2][3]={ 1.0, 2.0, 3.0,
                4.0, 5.0, 6.0 };

cout << " Valori della matrice: " << endl;
for (int i=0; i < 2; i++){
    for (int j=0; j < 3; j++)
        cout << x[i][j] << " ";
    cout <<endl;
}
cout << endl;
```

numero righe

numero colonne

Output del programma

Valori della matrice

1	2	3
4	5	6

Allocazione dinamica di matrici

```
// Matrice allocata dinamicamente
int nrow,ncol,i,j;
double **matrix;
cout << "numero di righe e di colonne: ";
cin >> nrow >> ncol;
matrix = new double*[nrow];
for ( i=0; i < nrow; i++){
    matrix[i] = new double[ncol];
}
for ( i=0; i < nrow; i++){
    for ( j=0; j < ncol; j++)
        matrix[i][j] = i*ncol+j;
}
for ( i=0; i < nrow; i++){
    for ( j=0; j < ncol; j++)
        cout << matrix[i][j]<< " ";
    cout << endl;
}
cout << endl;
```

Output del
programma

```
numero di righe e di colonne: 2 5
0 1 2 3 4
5 6 7 8 9
```

Allocazione dinamica di matrici: libero la memoria

```
// Matrice allocata dinamicamente
int nrow,ncol,i,j;
double **matrix;
cout << "numero di righe e di colonne: ";
cin >> nrow >> ncol;
matrix = new double*[nrow];
for ( i=0; i < nrow; i++){
    matrix[i] = new double[ncol];
}
.....

for ( i=0; i < nrow; i++)
    delete[] matrix[i];

delete[] matrix;
```

Controllo della dimensione

È importante ricordare che il C++ non esegue alcun controllo sulla corrispondenza tra la dimensione dei vettori (sia statici che dinamici) ed il loro effettivo utilizzo. Questo significa che nulla vieta di accedere alla posizione 100000 di un vettore definito con `int v[10]`; il compilatore calcolerà l'indirizzo `v+100000*sizeof(int)` e vi farà accedere a quella locazione, qualunque cosa essa contenga. La cosa funziona perfettamente anche se `v` è un puntatore non inizializzato. Naturalmente i risultati sono imprevedibili, e specie in scrittura, si possono combinare notevoli disastri. Quindi, si deve fare attenzione, specie se si usa l'allocazione dinamica.

Tipi composti

E' possibile definire **gruppi** di variabili dei **tipi predefiniti** in modo che, in determinati contesti, possano venire utilizzati **come una variabile singola**. Tali gruppi di dicono tipi composti o strutture.

```
struct studente {  
    string nome;  
    int matricola;  
    float media;  
};
```

Viene definita una struttura di nome **studente** con tre **campi**: una **stringa** per il nome, un **intero** per il numero di matricola e un **float** per la media.

La definizione **non** riserva alcuno **spazio in memoria**, ma serve a definire i campi che compongono la struttura. Per creare una variabile **st1** di tipo "studente" ed un vettore **vstud** a 100 componenti useremo:

```
struct studente st1, vstud[100];
```

Si possono utilizzare **vettori o matrici** dentro le strutture:

```
struct studente {  
    string nome;  
    int matricola;  
    float media;  
    float voti[50];  
};
```

Come pure si possono usare **altre strutture**

```
struct corso {  
    string nome;  
    int numero esami;  
    struct studente iscritti[1000];  
};
```

Accesso ai campi di una struttura

Per accedere ai **singoli elementi** di una struttura si usa la sintassi:

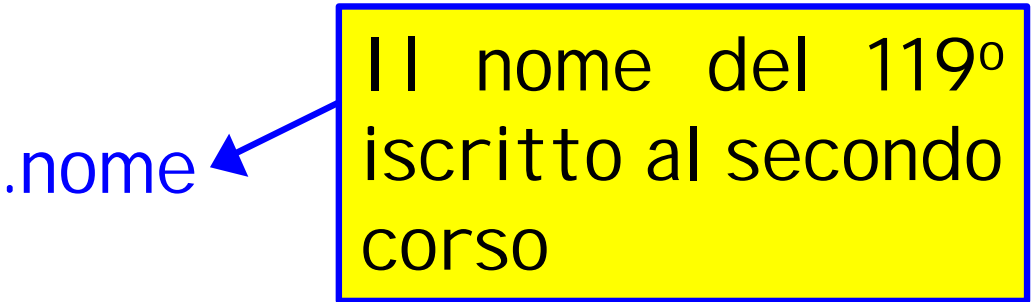
st1.matricola

La matricola di st1



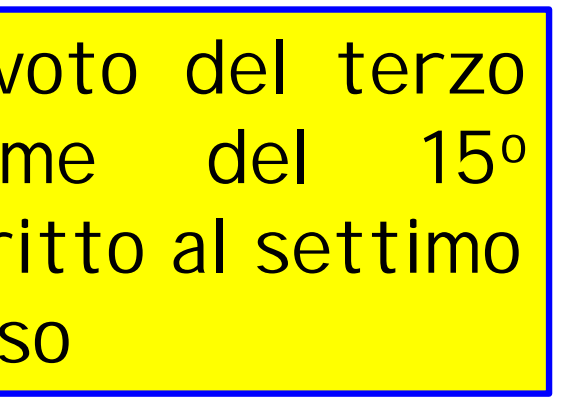
cl[1].iscritti[120].nome

Il nome del 119°
iscritto al secondo
corso



cl[6].iscritti[14].voti[2]

Il voto del terzo
esame del 15°
iscritto al settimo
corso



Puntatori a tipi composti

Si possono creare **puntatori a tipi composti**:

```
struct aa {  
    int i1;  
    float r1;  
};  
struct aa *ptr;
```

Per accedere ai **campi** si può usare `(*ptr).r1` (l'operatore `.` ha priorità maggiore di `*`) oppure

```
ptr->r1
```

Si possono definire **vettori di puntatori**:

```
float *rptr[10]; /* un vettore di 10 puntatori a float */  
*rptr[2]        /* il target della 3ª componente di rptr */  
rptr[3][4]     /* la 5ª componente del vettore puntato  
                dalla 4ª componente di rptr */
```

Subroutines

Una **subroutine** (come abbiamo visto parlando di linguaggio macchina) è un **blocco di istruzioni** che può essere “**chiamato**” da diversi punti di un programma e che, al termine, **provoca il ritorno dell'esecuzione al punto di chiamata**. In C++ una subroutine (in C++ si chiamano **functions**) si definisce così :

```
tipo nome(definizione parametri) {  
    istruzioni;  
    return;  
}
```

E si chiama con la sintassi

```
nome(parametri);
```

main() è una particolare function (da cui **comincia l'esecuzione del programma**) Ogni function può essere chiamata da **main()** o da un'altra function (inclusa se stessa).

Il return value

Ogni function può restituire il valore di una **variabile** al programma chiamante. Il **tipo** di tale variabile è quello **specificato prima del nome della function**. Il programma chiamante può utilizzare il **valore di ritorno** della function inserendo la chiamata in una **espressione al posto di una comune variabile**:

```
float a, val;  
...  
val = 3*cube(x) + x;
```

```
float cube(float x) {  
    float x3;  
    x3 = x*x*x;  
    return x3;  
}
```

Viene sostituito il valore di x^3

Variabili locali e globali

Tutte le variabili definite all'interno di una subroutine sono **locali**, ovvero **visibili a quella sola function**; sono allocate in modo **automatico**, a meno che l'opzione **static** non venga specificata prima del tipo.

Le variabili definite in un file .cpp **fuori da ogni function** sono invece **variabili globali**: sono visibili **a tutte le functions contenute in quel file sorgente** e sono allocate dal compilatore in modo **statico**.

Come anche per le functions, vale la regola che **in un dato punto del sorgente** possono essere utilizzati **soltanto oggetti** (functions, variabili globali, definizioni di strutture...) **che siano stati definiti nelle righe precedenti** (compilazione a passo singolo).

Esempio di variabili locali e globali

```
#include <iostream>
```

```
double in, out;
```

```
void quadrato() {
```

```
    out = in*in;
```

```
}
```

```
main() {
```

```
    cin >> in;
```

```
    quadrato();
```

```
    cout << out;
```

```
}
```

in e out sono **variabili globali**, ovvero visibili da tutte le functions del programma. Sono allocate in modo statico.

quadrato() è una function che agisce sulle variabili locali in ed out.

main() è una function che legge da tastiera il valore di in, chiama la function quadrato() e scrive su terminale il valore di out.

I parametri delle functions

Sono **variabili locali** della function allocate in modo **automatico** il cui **valore iniziale** è ottenuto **copiando quello della variabile specificata nella posizione omologa della riga di chiamata**. Siccome la function riceve **una copia** della variabile specificata dal programma chiamante, **non gli è possibile modificarne il valore** permanentemente (passaggio "**by value**").

```
void test(double a, int b) {
```

```
...  
}
```

a vale 0.123
b vale 45

```
double xx = 0.123;  
test(xx, 45);
```

Passaggio “by reference”

Se volete che una function **modifichi il valore** di una **variabile del programma chiamante**, dovete passare **un puntatore** a quella variabile. Il puntatore verrà **copiato**, ma le due copie punteranno comunque allo stesso target; la cosa viene fatta **implicitamente** per i vettori (il nome del vettore corrisponde al puntatore) o **esplicitamente** per gli scalari (mettendo una & davanti al nome della variabile nella definizione della function).

```
void exa(int &a, float b[], int *c) {
```

```
...
```

```
b[3] = 11.33;
```

```
}
```

viene modificato
cx[3]

```
int v1, v2[10];  
float cx[15];
```

```
...
```

```
exa(v1,cx,v2);
```

Passaggio di strutture

In generale anche le **strutture** passate come parametri vengono **copiate**; questo significa che i **campi vengono copiati uno a uno**. Naturalmente, se tra i campi ci sono **vettori o puntatori**, le corrispondenti variabili sono comunque **passate "by reference"**; ad esempio:

```
struct lista {  
    int index;  
    float key;  
    double dat[6];  
    lista* next;  
};  
search(struct lista first) {  
    ...  
}
```

index e **key** verranno passati a **search "by value"**; il target del puntatore **next** e il vettore **dat "by reference"**.

Il main program

Ogni programma eseguibile deve contenere uno ed un solo main program, ovvero una function da cui inizia l'elaborazione. In C il main program è una function che si chiama main. Oltre al formato main(), esiste il formato che riceve dall'esterno (dalla riga di comandi) un intero ed un vettore di stringhe:

```
main(int argc, char** argv) {  
    ...  
}
```

Se si attiva il programma con la linea di comandi

```
>./test par1 par2
```

si avrà che:

```
argc = 3  argv[0] = "./test"  argv[1] = "par1"  argv[2] = "par2"
```

Puntatori a functions

E' possibile fabbricare **puntatori a functions**; la cosa è utile quando si possiede una collezione di funzioni simili e si vuole **scegliere dinamicamente** quale function utilizzare.

La sintassi è la seguente:

```
int func(float *x, int n) {  
.....  
}  
main() {  
    int (*pf)(float *x, int n);  
    pf = func;  
    pf(a,b);  
}
```

Programmazione strutturata

Con **programmazione strutturata** si intende la **scomposizione** di un programma complicato in blocchi più semplici **opportunamente interconnessi** tra loro. Si tratta quindi di una **caratteristica del progetto** più che del linguaggio, e risulta particolarmente utile in **progetti complessi** o quando **diversi programmatori** devono lavorare sullo stesso codice.

Ci sono alcuni linguaggi di programmazione che forniscono **ampio supporto** alla strutturazione del codice e che, in qualche modo, **aiutano il programmatore a scomporre in modo corretto il programma**; tra questi possiamo menzionare **C++, Java, Fortan 90, C, e Pascal**. Altri linguaggi, come ad esempio il **Basic**, **non forniscono adeguato supporto** alla programmazione strutturata.

Struttura all'interno delle functions

Sebbene le **functions** debbano eseguire solo compiti specifici e debbano **essere mantenute piuttosto corte** (**al massimo 15-20 righe**) è comunque possibile **realizzare una struttura anche al loro interno**: ogni gruppo di istruzioni racchiuso tra parentesi graffe rappresenta un **blocco**, e può definire **variabili locali che non sono accessibili al resto della function**.

Anche una corretta **indentazione** del codice è indice di buona strutturazione, in quanto **aiuta ad identificare in modo rapido i blocchi** e a capire il flusso del programma.

L'approccio object oriented

Nell'approccio **object oriented** la programmazione si basa sulla **definizione di oggetti**, che sono evoluzioni sia del concetto di **struttura** che del concetto di **modulo**. Si tratta di **gruppi di variabili e functions** che implementano **funzionalità specifiche** (oggetti). Nei linguaggi "OO" è possibile:

- Realizzare **molte copie (funzionali)** di un particolare oggetto.
- Ridefinire gli **operatori standard (+, -, ...)** in modo che agiscano correttamente sugli oggetti definiti
- Definire oggetti che **ereditano le caratteristiche da altri oggetti definiti in precedenza** e ridefiniscono solo le caratteristiche nuove.

Il linguaggio C++ fornisce un ampio supporto alla programmazione object oriented. Nel seguito faremo **solo alcuni accenni** a queste funzionalità.

Le classi

In C++ il singolo **oggetto** (inteso come **l'insieme dei dati e delle funzioni che su di essi operano**) è rappresentato dalla **classe**. Si tratta di una evoluzione del concetto di struttura, alla quale è stata **aggiunta la possibilità di definire funzioni** che accedono in modo privilegiato ai dati contenuti nella classe (**metodi**).

Per visualizzare il concetto di classe possiamo pensare ai numeri complessi. È possibile definire una struct che contenga parte reale e immaginaria in due double. Una classe **consente però di aggiungere a tale definizione anche le funzioni che calcolano il modulo, la fase o una ridefinizione dell'operatore di somma**.

Si deve ricordare che, come una struct, una class **definisce un nuovo tipo di variabili**. È poi possibile **istanziare diverse variabili di quel tipo**.

struct complex

Cominciamo con il creare una struttura che contiene parte reale e immaginaria in due double.

```
struct complex {  
    double real;  
    double imaginary;  
};
```

```
int main() {  
    complex c;  
    double mod;  
    ...  
    c.real = 1.0;  
    c.imaginary = -2.0;  
    ...  
    mod = sqrt(c.real*c.real +  
               c.imaginary*c.imaginary);  
}
```

I dati contenuti nella struttura sono **pubblici**, nel senso che **tutti gli utilizzatori della struttura possono accedervi direttamente.**

class complex

Il passaggio ad una classe ha come conseguenza che **in assenza di indicazioni esplicite i dati sono privati**, cioè visibili solo dai metodi della classe.

In generale si raccomanda di mantenere i **dati privati** e di creare **funzioni nella classe** (metodi) **che consentano di leggerli**.

```
class complex {  
public:  
    double Re() { return real; }  
    double Im() { return imaginary; }  
private:  
    double real;  
    double imaginary;  
};
```

```
int main() {  
    complex c;  
    double mod;  
    ...  
    mod = sqrt( c.Re()*c.Re() +  
                c.Im()*c.Im() );  
    ...  
}
```


Overloading

Naturalmente i dati nella classe vanno anche scritti, e quindi si possono fabbricare delle funzioni apposite. I nomi di tali funzioni possono coincidere con quelli delle functions di lettura (**overloading**), visto che il compilatore può scegliere la funzione da usare in base al tipo dei parametri specificati nella chiamata.

```
class complex {  
public:  
    void Re(double r) { real = r; }  
    double Re() { return real; }  
    void Im(double i) { imaginary = i; }  
    double Im() { return imaginary; }  
private:  
    double real;  
    double imaginary;  
};
```

```
int main() {  
    complex c;  
    double mod;  
    ...  
    c.Re(1.0);  
    c.Im(-2.0);  
    ...  
    mod = sqrt( c.Re()*c.Re() +  
                c.Im()*c.Im() );  
    ...  
}
```

Operatore di creazione

L'operatore di creazione è un metodo che viene chiamato nel momento in cui un oggetto appartenente alla classe viene creato. Il nome del metodo coincide con quello della classe. Si possono usare parametri che vengono utilizzati per inizializzare l'oggetto. È possibile l'overloading.

```
class complex {  
public:
```

```
    complex() { real = 0.0; imaginary = 0.0; }
```

```
    complex(double r, double i) { real = r; imaginary = i; }
```

```
    void Re(double r) { real = r; }
```

```
    double Re() { return real; }
```

```
    void Im(double i) { imaginary = i; }
```

```
    double Im() { return imaginary; }
```

```
private:
```

```
    double real;
```

```
    double imaginary;
```

```
};
```

```
int main() {  
    complex c(1.0, -2.0);  
    double mod;  
  
    ...  
    mod = sqrt( c.Re()*c.Re() +  
                c.Im()*c.Im() );  
  
    ...  
}
```

Metodi accessori

Si possono aggiungere metodi che semplificano la manipolazione degli oggetti.

```
class complex {  
public:
```

```
    complex() { real = 0.0; imaginary = 0.0; }
```

```
    complex(double r, double i) { real = r; imaginary = i; }
```

```
    double Mod() { return sqrt(real*real + imaginary*imaginary); }
```

```
    double Phase() { return atan2(imaginary,real); }
```

```
    void Re(double r) { real = r; }
```

```
    double Re() { return real; }
```

```
    void Im(double i) { imaginary = i; }
```

```
    double Im() { return imaginary; }
```

```
private:
```

```
    double real;
```

```
    double imaginary;
```

```
};
```

```
int main() {  
    complex c(1.0, -2.0);  
    double mod;  
  
    ...  
    mod = c.Mod();  
  
    ...  
}
```

Ridefinizione di operatori

Il C++ consente di (ri)definire il modo in cui gli operatori (+, -, *, /, ...) agiscono sugli elementi di una classe.

Se a e b sono oggetti di una classe, allora per definire il comportamento di

$a + b$

si deve pensare che l'operazione sia scritta come

$a.operator+(b)$

e definire il metodo **operator+** in modo acconcio.

Si capisce che, nel caso di operatori binari, spetta sempre alla classe che compare a sinistra dell'operatore di eseguire la definizione. Si deve anche ricordare che un operatore è una funzione che restituisce un valore (nel caso del + la somma dei due addendi).

class complex

```
class complex {
public:
    complex() { real = 0.0; imaginary = 0.0; }
    complex(double r, double i) { real = r; imaginary = i; }
    double Mod() { return sqrt(real*real + imaginary*imaginary); }
    double Phase() { return atan2(imaginary,real); }
    double Mod() { return sqrt(real*real + imaginary*imaginary); }
    double Phase() { return atan2(imaginary,real); }
    double Re() { return real; }
    double Im() { return imaginary; }
    complex operator+(const complex &b) {
        complex c;
        c.real = real + b.real;
        c.imaginary = imaginary + b.imaginary;
        return c;
    }
    bool operator==(const complex &b) {
        if (real == b.real && imaginary==b.imaginary) {
            return true;
        }
        return false;
    }
private:
    double real;
    double imaginary;
};
```

```
int main() {
    complex a;
    complex b(1.0, -2.0);
    complex c(-1.0, 7.0);
    ...
    a = b + c;
    ...
    if (a == c) {
        ...
    }
    ...
}
```

Osservazioni sulla strategia OO

- Come abbiamo osservato nell'evoluzione da struct a class, c'è una tendenza a semplificare sempre più il main scaricando la complessità nella definizione delle classi. La programmazione a oggetti prevede che tutto il codice sia incluso in una classe.
- Noi ci siamo soffermati su uno degli aspetti importanti delle classi, ovvero la possibilità di estendere le capacità di base del linguaggio includendo nuovi tipi di variabili (o oggetti) e nuove definizioni degli operatori. Esiste un altro aspetto importante: una classe può derivare le sue funzionalità di base da una classe preesistente, aggiungendo o modificando solo alcuni metodi specifici. Questa possibilità rende molto facile adattare programmi esistenti a nuove necessità.
- Alcuni tipi di variabili che abbiamo usato (string e fstream) sono classi appartenenti alla cosiddetta "C++ Standard Library" o STL.
- Malgrado il fatto che la programmazione OO sia molto efficace e molto di moda, non è vero che tutti i problemi si prestano ad essere risolti in questo modo. Un buon progetto software non cerca di "oggettizzare" a tutti i costi, ma cerca di sfruttare al meglio le tecniche OO dove queste sono in grado di fornire i migliori risultati.