Programmazione Avanzata per il Calcolo Scientifico
Lezione N. 1

Luca Formaggia

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

A.A. 2005/2006

Aule e orari

Il corso consiste di lezioni ex catedra che si tengono il MERCOLEDI dalle 8.30 alle 11.30 nella sala seminari MOX e di laboratori didattici che si tengono il GIOVEDI dalle 8.30 alle 10.30 nel Laboratorio Didattico del Dipartimento (quarto piano).

Docenti

- Docente: Luca Formaggia (luca.formaggia@polimi.it)
- Esercitatore: Tiziano Passerini (tiziano.passerini@mate.polimi.it)

Orario di ricevimento: previo appuntamento.

Modalità di esame

- Progetto concordato con il docente
- Prova orale

Non sono previste prove in itinere.

Supporto informatico

Il portale del corso è

`www2.mate.polimi.it:8080/CN/PACS05`

L'iscrizione al portale è necessaria per accedere ai tutti i servizi.

Il sistema operativo di riferimento è Linux, i compilatori di riferimento sono i compilatori gnu. Utilizzeremo il softare Eclipse come strumento di sviluppo.
Sul portale si trovano informazioni su Linux e su Windows.

Una nota linguistica

Il corso viene dato in lingua Italiana. Ciononostante, spesso la terminologia in lingua Inglese è di uso comune (e talvolta più precisa).
Si farà quindi spesso uso di quest'ultima (cercando comunque di fornire anche il corrispondente termine in Italiano).

Introduzione

- Il calcolo scientifico
- Linguaggi di programmazione

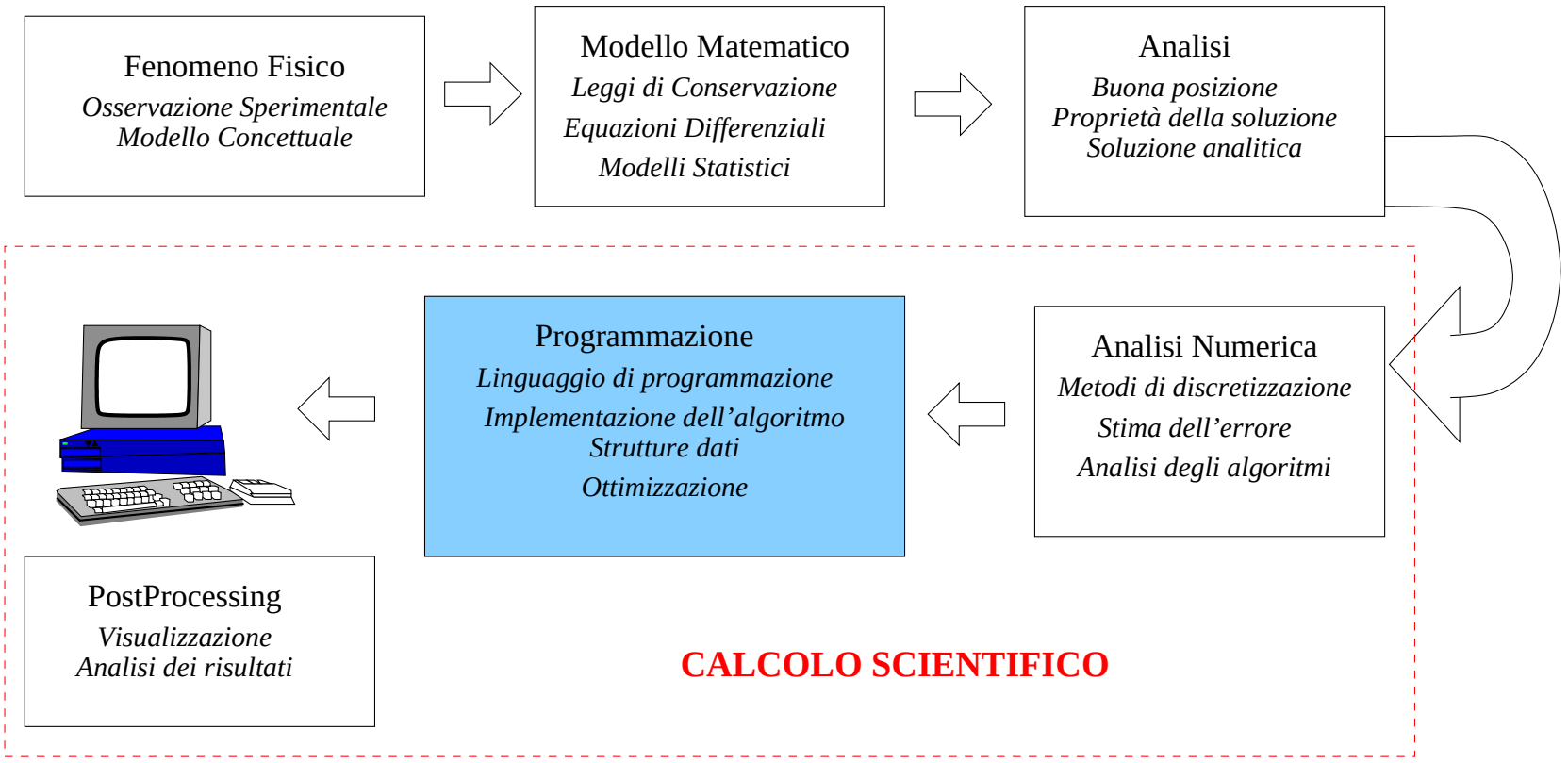
Primi elementi di C++

- Un esempio di programma
- Variabili, operatori, istruzioni di controllo
- Dichiarazione , definizione...
- Il ciclo for più in dettaglio
- Conversione implicita e esplicita
- Identificatori e parole chiave
- I commenti
- Ambito di visibilità
- Gli array

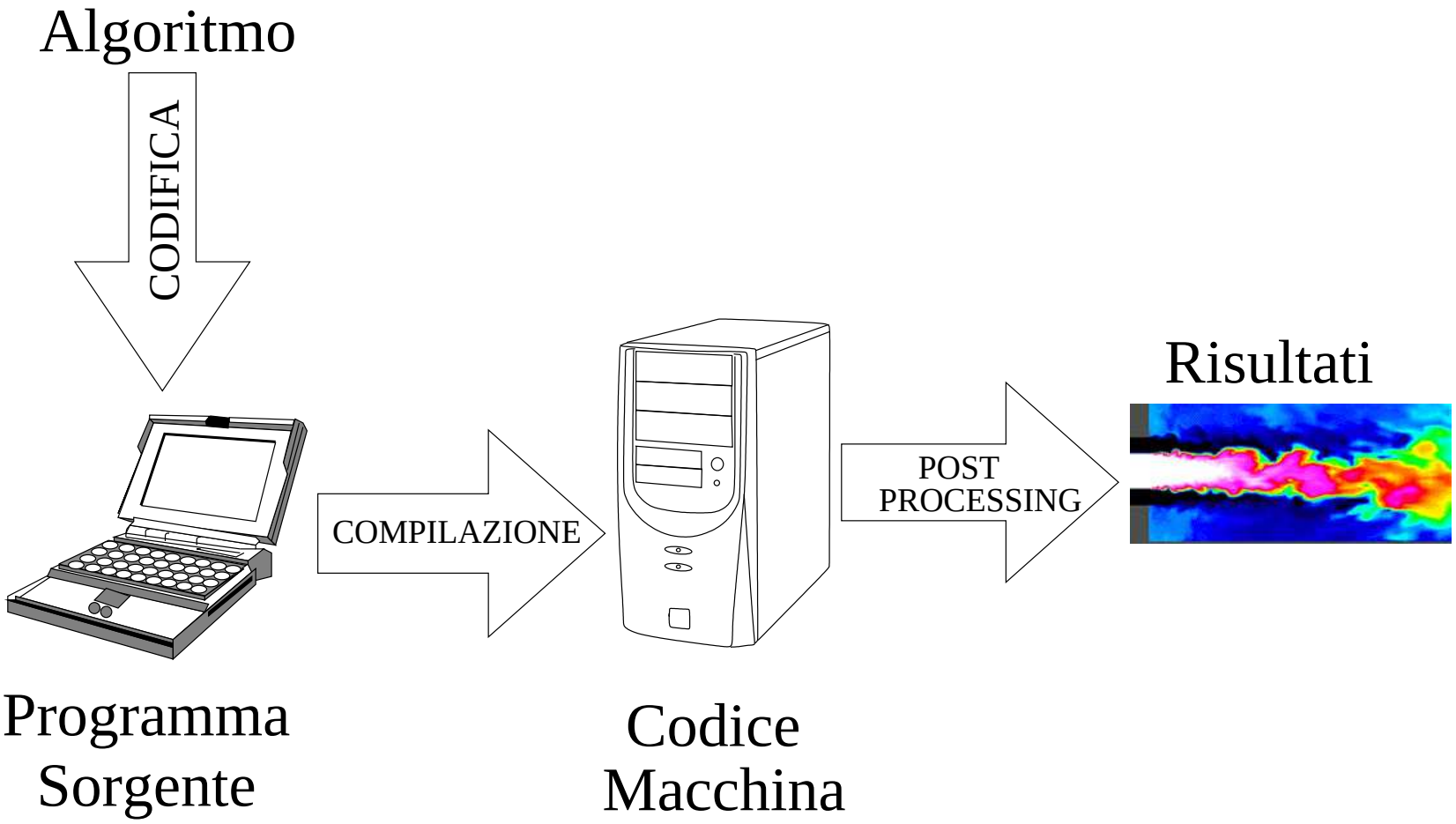
Una possibile definizione

Il calcolo scientifico è la disciplina che permette di riprodurre su un calcolatore un fenomeno o processo descritto da un opportuno modello matematico.

Dalla realtà al computer



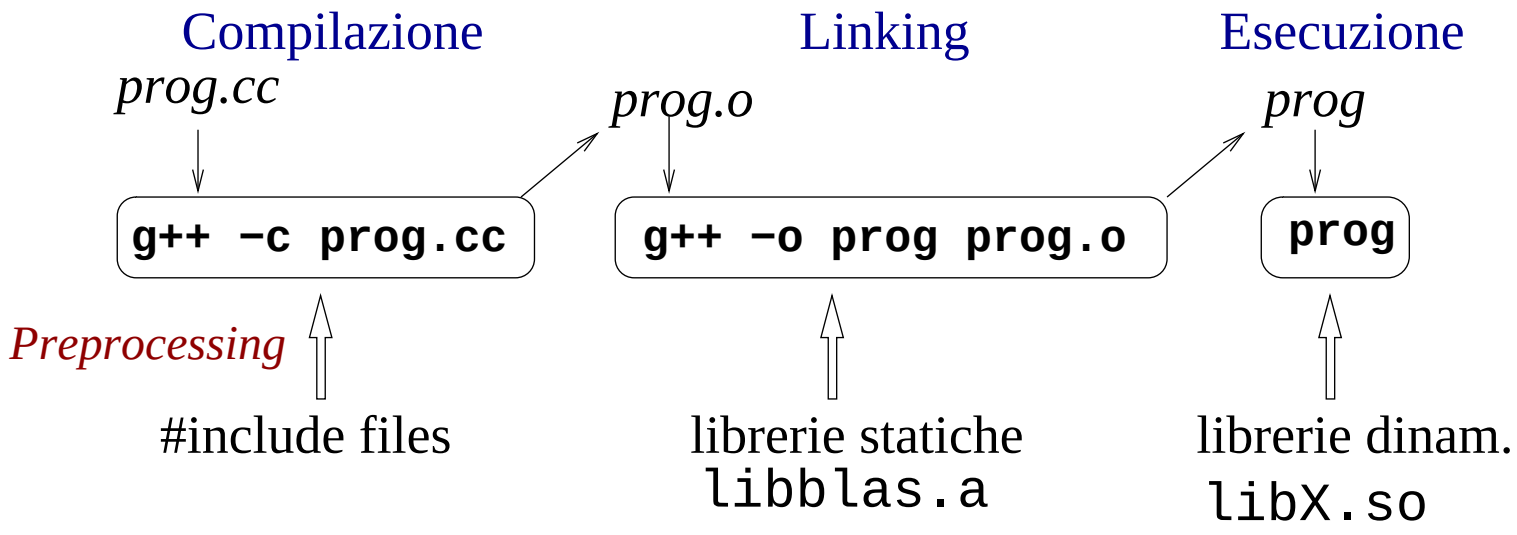
Dall'algoritmo al risultato



Implementazione di un linguaggio

- Interpretato Le istruzioni (comandi) vengono processate 'sequenzialmente' e 'tradotte'in codice macchina. Esempio: BASIC, MATLAB, Pyhthon.*Semplicità di programmazione e debugging*
- Compilato L'intero codice sorgente viene tradotto in codice macchina, creando un *eseguibile*. Esempi: C++, FORTRAN. *Migliori prestazioni*. Più difficile il "debugging".
- Semicompilato Il codice sorgente o parte di esso viene tradotto in un codice intermedio (byte-code) indipendende dalla architettura. Quest'ultimo viene poi interpretato "run-time" da una "macchina virtuale" (virtual machine). Esempio: Java.

Il processo di compilazione in C(++)



Il comando `g++ -o prog prog.cc` esegue i passi di *compilazione* e *linking* in sequenza.

Tipi di programmazione

Programmazione Procedurale.

I dati vengono elaborati da *procedure* (comandi o funzioni) che ne modificano lo stato operando (tipicamente) in modo sequenziale.

Programmazione ad oggetti.

I dati sono encapsulati in strutture apposite (*classi*). Si opera sui dati tramite *metodi* della classe. L'accesso diretto ai dati è *normalmente proibito*.

Programmazione generica.

Si opera su *tipi di dati* differenti (ma che soddisfano *pre-requisiti* opportuni) usando la stessa *interfaccia*.

Programmazione procedurale

Esempio in MATLAB

```
A=gallery('poisson',100); b=ones(100,1);
[L,U]=lu(A) y=L\b;
x=U\y;
```

Programmazione ad oggetti

Esempio in C++

```
class Matrix{public:
Matrix(string filename);
Vector solve(Vector const & b);
private:
double * dati;}
...

Matrix A("file.dat");Vector b;
Vector x;
x=A.solve(b);
..
```

Non si può accedere direttamente a `dati`.

Programmazione generica

Esempio in C++

```
template<class T> T sqrt(T& x);
...
int main(){
float x; int y; complex z;
flot rx; int ry, complex rz;
rx=sqrt(x); // usa sqrt<float>
ry=sqrt(y); // usa sqrt<int>
rz=sqrt(z); // usa sqrt<complex>
...}
```

La funzione `sqrt<T>(T &)` si applica a ogni tipo `T` purchè esso soddisfi opportuni prerequisiti stabiliti dal programmatrore (p. es. $x > 0$ per dati float e int).

Programmazione generica

Un secondo Esempio

```
template<class T> class Vector {...}

class triangle {...}

int main(){
Vector<float> a;// Un vettore di float Vector<triangle>
b;// Un vettore di triangle ...
triangle t=b[2]; ...}
```

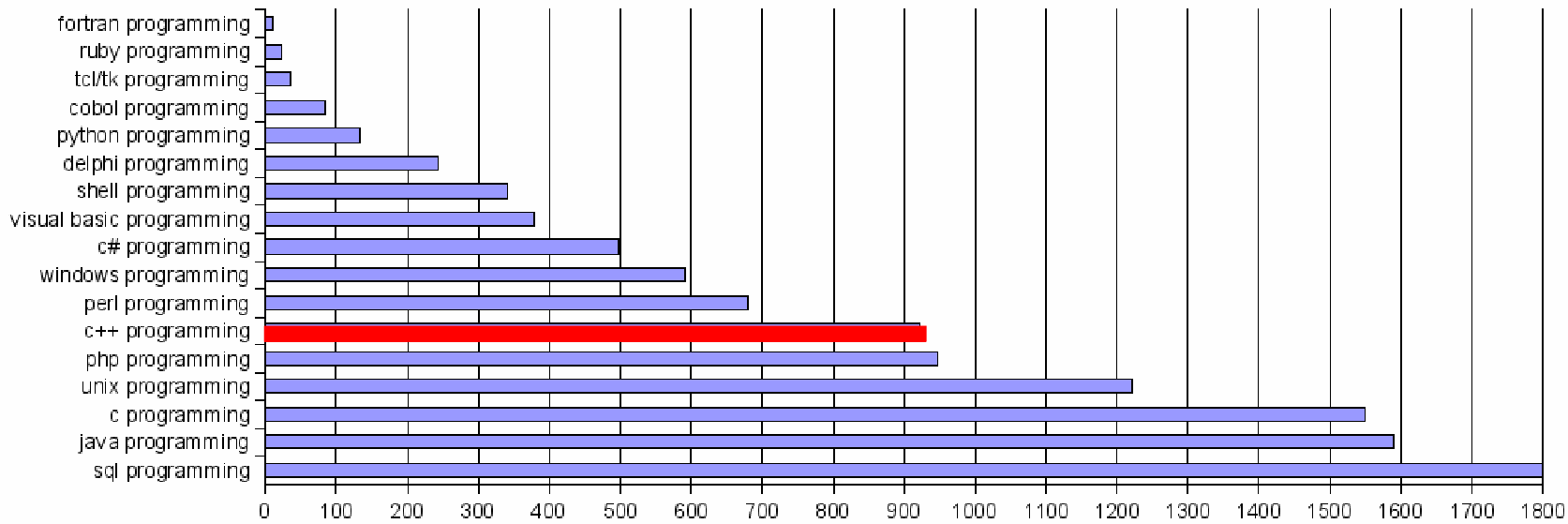
Il linguaggio C++

- Supporta solo la modalità di *compilazione* (linguaggio compilato)
- È una estensione del linguaggio C, quindi supporta la *programmazione procedurale*
- Supporta inoltre la *programmazione ad oggetti* e la *programmazione generica*.
- Compilatori facilmente disponibili (e spesso gratuiti).
- È il 3^o linquaggio più utilizzato (dopo *C* e *Java*) nel mondo (statistiche Luglio 2005).
- È un linguaggio complesso e l'utilizzo nel calcolo scientifico richiede attenzione per evitare degradazione di performance.

C++ e lavoro

N. offerte di lavoro che richiedono
conoscenza di un linguaggio di programmazione

Programming Language Popularity (D.N. Welton, 2004)



Alternative per il calcolo scientifico

- FortranXX. Programmazione procedurale. Fortran90 ha introdotto i *moduli* e la *allocazione dinamica della memoria* che permette di implementare una programmazione "quasi" ad oggetti (con un certo sforzo). Estremamente efficiente per le operazioni matematiche.
- Java. Lo sviluppo di applicazioni numeriche in Java è limitato principalmente a scopi didattici (web computing).
- C. La maggior parte dei codici commerciali di calcolo scientifico è scritta in questo linguaggio.

esempio_ch1

```
#include <iostream> // include il modulo per i/o
// della libreria standard
int main() {using namespace std;
int n, m; // n e m sono 2 interi
cout « ''Dammi due interi:'' « endl;
cin » n » m;

if (n > m) {
int temp = n; n = m;
m = temp; }
double sum = 0.0;
for (int i = n; i <= m; i++)
sum += i;// sum += i significa sum = sum + i;
cout « ''Somma='' « sum « endl;
}
```

la direttiva "include"

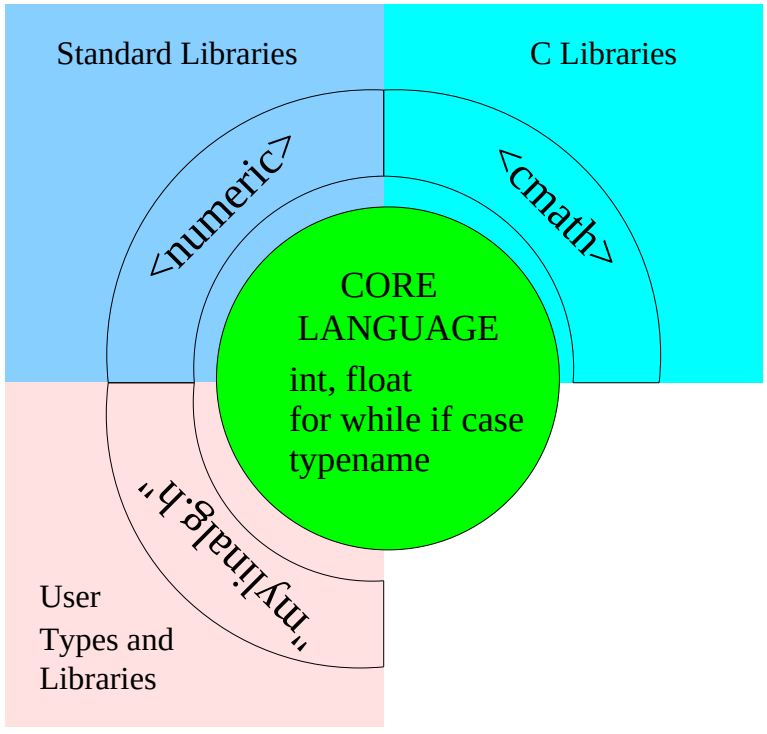
```
#include<iostream>
using namespace std;
```

Il linguaggio C++ consiste in un insieme di istruzioni di base (*core language*) e "moduli" (*header o include files*) opzionali che possono essere richiamati con in comando `#include<XXX>`.

In particolare la libreria standard (*Standard template library* o STL) mette a disposizione il modulo `iostream` per l'i/o, che definisce (tra l'altro) `std::cin` e `std::cout` per l'i/o da tastiera.

`using namespace std;` permette di omettere il qualificatore (*scope qualifier*) `std::` alle variabili e funzioni introdotte dal modulo.

Struttura del linguaggio



Oltre agli *header file* standard, l'utente ne può aggiungere degli altri.

La lista completa degli standard C++ headers files si trova a pag. 125 del libro di testo.

Il programma principale

```
int main() {
....
for (int i = n; i <= m; i++) a[i]=i;
}
```

Per produrre un eseguibile il sorgente del programma deve contenere uno e un solo `main()` (*programma principale* o *main program*).
In C++ il *main* ritorna sempre un intero (`int`) e può prendere in ingresso due argomenti (si veda l'esercitazione) oppure nessuno, come in questo caso.
Le istruzioni terminano *sempre* con il punto e virgola (;) che funge da separatore. Non vi è un formato predefinito, a differenza di *Python* o del *Fortran77*.

Le variabili

```
int n, m;
float x, y;
Triangle t; ...
for (int i = n; i <= m; i++)
```

Vi sono due gruppi di variabili: le variabili native del linguaggio (*in-built*) e quelle definite dal programmatore.

Tra le variabili native, il tipo `int` identifica una variabile che memorizza un valore *intero* con segno.

A differenza del *Matlab* le variabili devono sempre essere dichiarate e (a differenza del *Fortran*) possono essere dichiarate *ovunque* (ma prima di essere utilizzate).

Le istruzioni di controllo e i cicli

```
if (n > m) {
...Blocco di istruzioni
}...
for (int i = n; i <= m; i++) sum+=i
```

Il linguaggio prevede *istruzioni di controllo*: `if{...}else{...}` e di *ciclo*: `for(){...}`, `do{...} while()`.

L'input/output

```
cout « ''Dammi due interi:'' « endl;
cin » n » m;
```

Uno *stream* può essere immaginato come una sorgente o un "pozzo" di dati (tipicamente caratteri), usualmente associati a lettura/scrittura da file o teminale.

La *standard template library* attraverso il modulo `iostream` mette a disposizione tre *stream* per i/o da/su terminale:

std::cin	Standard Input
std::cout	Standard Output
std::cerr	Standard Error

Tipi di variabili

I principali tipi "nativi" (anche detti "primitivi") forniti dal linguaggio sono

`float`	Numero reale in singola precisione (4 bytes)
`double`	Numero reale in doppia precisione (8 bytes)
`long double`	Numero reale in precisione estesa (16 bytes)
`int`	Intero con segno (tip. 4 bytes)
`long int`	Intero esteso con segno (tip. 8 bytes)
`unsigned`	Prefisso per interi positivi
`bool`	Viariabile logica
`char`	Carattere (4 bytes).

Virgola mobile

Tipicamente le variabili `float` obbediscono allo standard IEEE. Il C++ permette però di controllare i limiti numerici ed in particolare di trovare l'epsilon macchina per i vari tipi a virgola mobile. Si deve usare il modulo della Standard Template Library `<limits>`.

```
#include <limits>
...
std::numeric_limits<float>::epsilon();
std::numeric_limits<float>::max();
```

Si veda anche l'esempio in `numeric_limits`.

char

```
char p='A';
char q='\n';
```

Un carattere è individuato da gli apici singoli. Vi sono dei caratteri speciali, quali `'\n'`. Quest'ultimo indica *andare a capo* (carriage return).

Una variabile char contiene un singolo carattere. Vedremo in seguito come trattare le stringhe di caratteri.

bool

```
bool l=true;
bool s;int a=5;
s= (a==5); //s è pari a true
```

La variabile `bool` può assumere solo due valori, vero o falso, indicati dalle due parole riservate true e false. Si può usare in alternativa 1 e 0.

Le variabili `bool` vengono usualmente utilizzate per memorizzare il risultato di espressioni logiche.

Principali operazioni

`i=g+g;` La somma (+), moltiplicazione (*) etc. sono definite per tutti i tipi numerici.

`a=pow(3.5,4)`. $(3.5)^4$. Richiede <cmath>.

`a*=5`. Equivalente a a=a*5. Ma più efficiente. In generale *op* ha anche l'equivalente *op*=

`++i` e `i++`. Preincremento e post-incremento. Entrambe pari a $i = i + 1$ ma la seconda ritorna il valore di i prima dell'incremento.

`-i` e `i-`. Come sopra ma decrementando di 1.

Operatori logici: `&&`, `||` e `!`.

Test logici: ==, !=, <, <= etc.

Dichiariazione e definizione

Questi termini sono di suo comune e vengono spesso confusi.

- Dichiarazione. Associa a una variabile un *tipo*. Permette al compilatore di conoscerne la *dimensione in memoria* e di fare le verifiche di consistenza.
- Definizione. Indica come una variabile viene costruita (e le sue funzionalità).
- Istanza. Momento in cui una variabile viene «costruita»in memoria.
- Inizializzazione. Assegnazione di un valore iniziale a una variabile (coincide temporalmente con l'istanza)
- Assegnazione Attribuisce a una variabile *già istanziata* in precedenza un valore, usualmente attraverso l'operatore di assegnazione =.

Riprenderemo questi concetti quando introdurremo le funzioni e i *tipi* creati dall'utente.

Inizializzazione

Per i tipi nativi la dichiarazione è sempre anche una definizione (a meno che non si usi la keyword *extern*)

```
extern int z; Dichiarazione
int a; Dichiarazione (e inizializzazione di default)
float b=3.14; Inizializzazione
float c(3.14); Inizializzazione
float e=atan(1)*4; Inizializzazione
long int=3L; Inizializzazione
a=10; Assegnazione
```

Alcune regole importanti

- Una variabile deve sempre essere dichiarata prima di poter essere utilizzata.
- Una variabile può essere dichiarata più volte, purchè la dichiarazione sia identica.
- Una variabile può essere definita *una sola volta* nel programma.
- La definizione è anche una dichiarazione.

Ciclo for

```
for(init;test;update) {
corpo (body)
..}
```

init è una istruzione che viene eseguita all'inizio del ciclo, tipicamente per inizializzare una variabile locale, per esempio `int i=0`.

test. Una espressione logica che viene eseguita all'inizio del *corpo* del ciclo. Se è vera le istruzioni nel *corpo* vengono eseguite. Es.: `i<10`.

update è una istruzione che viene eseguita alla fine del *corpo* del ciclo. Es.: `++i`.

Le variabili definite nel ciclo sono *locali* al ciclo:

```
i=90;
for(i=0;i<=10;++i) ...

cout«i; // i qui vale 90
```

Conversioni tra variabili

```
int a=5; float b=3.14;double c,z;
c=a+b
c=double(a)+double(b) (conv. per costruzione)
z=static_cast<double>(a) (conv. per casting)
```

Il C++ prevede una serie di regole per la conversione implicita tra tipi "nativi".
È possibile anche indicare esplicitamente la conversione, come indicato sopra.

Nota: È meglio, per la leggibilità del codice, usare sempre *conversioni esplicite*.

Conversioni particolari

```
int a=5; char A='A'
bool c=bool(a); non serve la conver. esplicita
float b=3.14;
c=b; c è pari a true
a=A; a è il codice ASCII di 'A'
```

Vi è la regola che ogni valore non nullo di variabili intere e a virgola mobile e ogni carattere diverso dal carattere nullo sono convertiti in true.

Nota: Questa regola si rivelerà molto utile.

Conversione con *casting statico*

Esiste un altro modo di eseguire una conversione esplicita, più efficiente di quanto visto in precedenza (chiamata *conversione per costruzione*).
Esso utilizza l'istruzione di *casting statico*:

```
float a=10; double b=static_cast<double>(a);
```

Il significato del *casting* verrà approfondito in seguito.

Identificatori

Un nome di variabile (o di funzione) deve essere un valido *identificatore*. Un identificatore in C++ è una successione di caratteri alfanumerici in cui il primo carattere è alfabetico. È ammesso il carattere *underscore* (_), anche come primo carattere.

Il linguaggio distingue lettere maiuscole da minuscole: `pippo` è un identificatore diverso da `Pippo`.

Un identificatore NON può essere uguale a una parola chiave (*keyword*). La lista delle parole chiavi del C++ (in tutto 74) è contenuta a pag. 20 del libro di testo.

Due modalità

```
int a=0; // Commento in linea
/* Commento su più linee
È ereditato dal C */
int main(){
....
```

Esistono delle tecniche di documentazione (si veda il programma DoxyGen) che permettono di produrre automaticamente il manuale di riferimento processando il codice sorgente.

Lo *scope* (ambito di visibilità)

Lo *scope* (ambito di visibilità) di una variabile (o una funzione) definisce la porzione di programma in cui la variabile è *visibile*. Una variabile non in scope può essere tuttavia accessibile usando lo *scope resolution operator* `::` (ne parleremo più in dettaglio più avanti...).

```
..
int x=10;//variabile globale
int main()
int x=20;//variabile locale
cout « x« ::x;
...
```

- Ogni parte di programma racchiusa tra parentesi graffe identifica uno *scope* separato.
- Gli scope sono annidati tra loro, e uno scope "interno" eredita le definizioni delle variabili di quello esterno.
- Una variabile istanziata in uno scope è locale a tale scope.
- Il nome di una variabile (o funzione) dichiarata in uno *scope* nasconde una variabile o funzione dichiarata in uno scope più esterno (*name hiding* o *shading*).
- Lo *scope* più esterno è detto globale.
- L'operatore `::` (*scope resolution operator*) permette di accedere a variabili/funzioni globali.
- Le istruzioni di ciclo `for`, `while` etc. definiscono uno scope, che include la parte che concerne il test.
- Una variabile viene distrutta alla termine dello scopo in cui e stata istanziata (a meno che non sia dichiarata *static*, come si vedrà in seguito).

Array (monodimensionali)

```
double a[5]={1.6,2.7,3.4,7,8};
b=a[0]+a[3];//b vale 8.6
int c[2]; c[0]=7; c[1]=8;
```

In C++ gli array vengono indirizzati a partire da 0 usando l'operatore `[]`.
Un'alternativa: usare i `vector<>` della Standard Template Library

```
#include <vector>
vector<double> a(5);
a.push_back(1.6)\1.6 -> a[0]
```