Introduzione a C++ e programmazione Object Oriented

Stefano Lacaprara

INFN LNL

Contatti

- Stefano.Lacaprara@pd.infn.it

- http://www.pd.infn.it/~lacaprar/Didattica/C++/

Contenuto del Corso

- Introduzione C++/OO
  - Essendo un laboratorio, sarete voi a scrivere programmi su mie specifiche
  - Vi descrivo il problema, gli strumenti che servono e poi vi guido verso una soluzione
  - Programmi di tipo fisico/matematico, spero interessanti.
- Introduzione a ROOT
  - Strumento per analisi dati molto usato in fisica:
  - Alcuni esempi concreti (istogrammi, fit, etc)
  - Piu' o meno quello che vi servira' per i laboratori sperimentali

Modalita' d'esame

- La vostra attivita' in laboratorio e' parte fontamentale della valutazione del vs rendimento
- Esame finale: vi daremo alcuni esercizi da fare a casa che poi discuteremo insieme
- Probabilmente anche piccola esercitazione in laboratorio
- Parte orale di statistica con prof. Paganoni.

Scopo dell'introduzione

- Elementi di informatica – cosa è un computer
- Cosa servirà a noi…
- Elementi di C
  - Saper leggere un programma
  - Saper scrivere un programma elementare
- Elementi di C++
  - Perche' passare a C++ e programmazione ad oggetti
- Elementi di OO
  - Concetti generali di programmazione a oggetti

Elementi di informatica (e di programmazione)

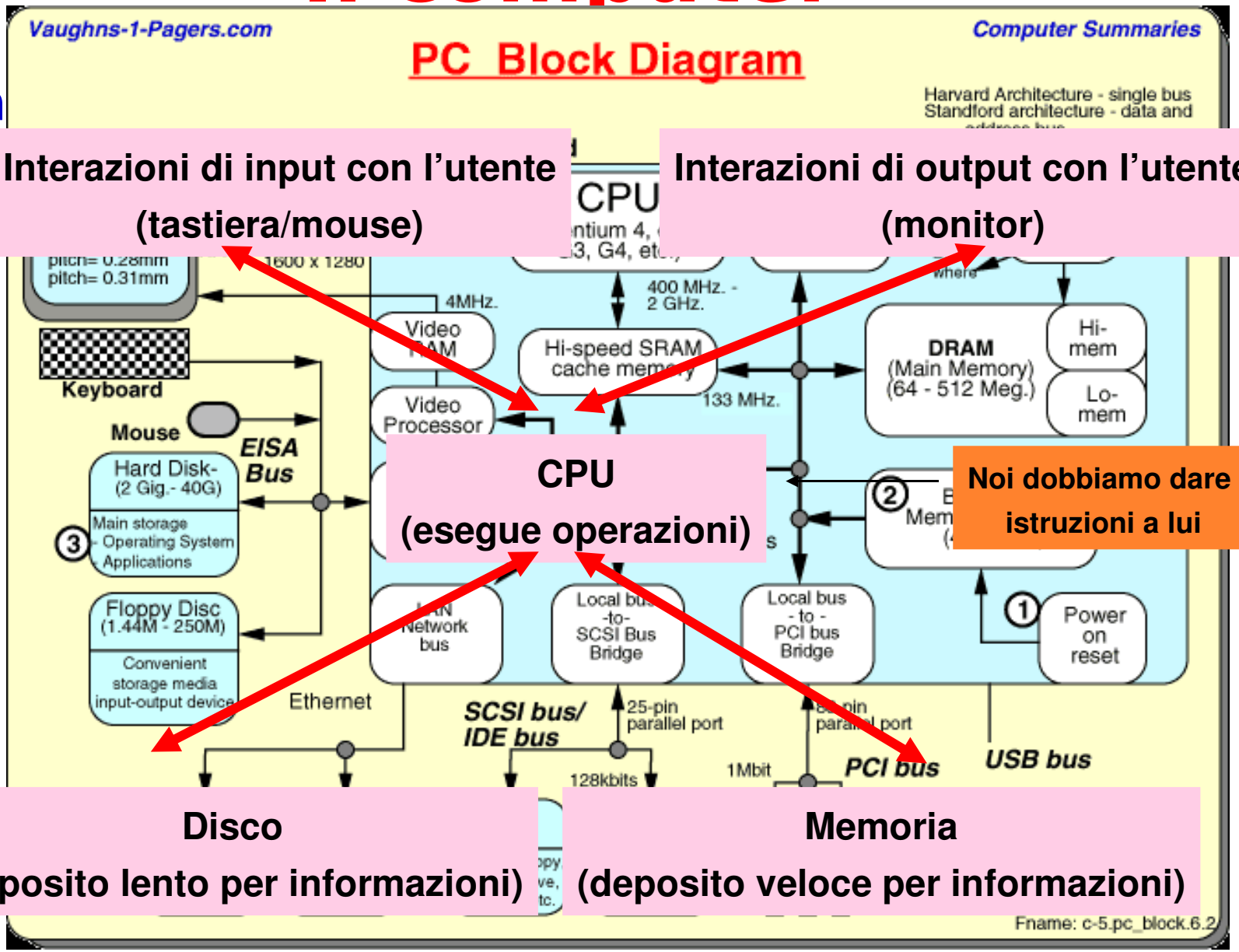
- Prima Legge Della Programmazione dei Computer

> Un computer fa quello che gli dici, non quello che vuoi.

- Corollario: I computer sono stupidi
- Programmare = convincere una (stupida) scatola di latta a fare quello che vogliamo noi
- Stupida, ma veloce!
  - Essere umano, a parte casi particolari esegue un' operazione ogni ~5 secondi
  - Computer attuali almeno un miliardo di operazioni al secondo

Il computer

- Im



Interazioni di input con l'utente (tastiera/mouse)

Interazioni di output con l'utente (monitor)

CPU (esegue operazioni)

Noi dobbiamo dare istruzioni a lui

Disco (deposito lento per informazioni)

Memoria (deposito veloce per informazioni)

```
; centigrade (celsius) to fahrenheit
calculation and vice-versa.

; it may not be accurate, because of
integer division.

; this program prints out the result
in binary code.

; to see result in hexadecimal or
decimal form click vars.

name "celsi"
```

Equivalente a

Cel = (Fahreneit – 32)*5/9

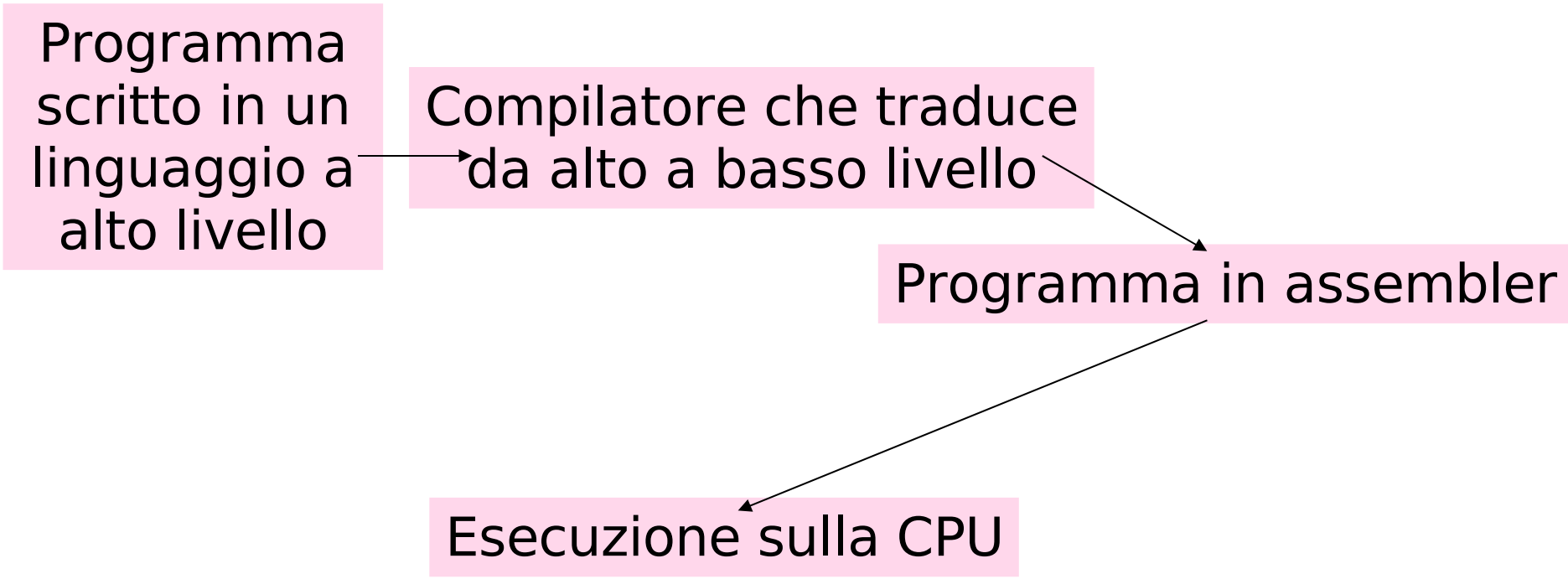
```
start:rint   ; print bl
```

```
; convert celsius to fahrenheit
according

; to this formula: f = c * 9 / 5 + 32

mov cl, tc

mov al, 9

imul cl

mov cl, 5
```

```
according

; to this formula: c = (f - 32) * 5 /
9


mov cl, tf

sub cl, 32

mov al, 5

imul cl

mov cl, 9
```

Per fortuna …

- Non dobbiamo (più) programmare così
- Linguaggi di alto livello: con una sintassi più vicina a quella degli essere umani



Nel nostro caso…

- Noi utilizzeremo linguaggi ad alto livello C++, e un compilatore
  - Compilatore: programma che traduce da alto a basso livello…
  - Oltre a fare questo i compilatori fanno un controllo della sintassi dei nostri comandi e molto altro

Tipi di programmazione …

- I teorici della programmazione si sono sbizzarriti nel modellizzare e creare **paradigmi** di programmazione diversi, che meglio si adattino a determinati usi
  - Scienziato che vuole realizzare una simulazione in meno di due ore
  - Programmatore professionista che scrive un programma che debba sopravvivere 10 anni (o dopo il suo licenziamento)

- Noi fortunatamente siamo (quasi sempre) nella prima categoria …
- Ma purtroppo l'informatica si è evoluta nella seconda direzione; spesso dobbiamo adeguarci
- Io lavoro in un esperimento da 9 anni (e deve ancora partire…)

Tipi di programmazione

- Procedurale
  - “Un passo per volta”; molto vicina al modo di pensare della persona “normale”
    - Per raggiungere il mio scopo, prima faccio questo e poi quello e alla fine di molti piccoli passi raggiungo il risultato
    - Linguaggio C (fortran, pascal, ...)
- A Oggetti
  - Invece di partire dalle azioni, parto dalle entita' che devono compiere le azioni
  - “Prima di cominciare a programmare, pensa bene a quello che fai e definisci le entità su cui ti muovi”
    - capiremo più tardi cosa questo voglia dire
    - C++ (python, java, ...)

Linguaggio procedurale

- Nei linguaggi procedurali, gli oggetti su cui operare sono semplici
  - Numeri interi / floating-point (quasi reali...)
  - Caratteri
  - Stringhe di caratteri
- Mentre le azioni possono essere composte
  - Calcolare *arctanh(θ)*
  - *Controllare se un numero intero e' primo oppure no*
- O elementari
  - Sommare due numeri

Linguaggio OO

- Definisco oggetti complessi (nuovi **tipi**) e adatti alle mie esigenze
  - Container, strutture, iteratori, vettori, persone, aerei, conti correnti...
  - Type che modellizzano il problema che devo affrontare
- Definisco interfaccia dei nuovi oggetti
  - Cosa sa fare un nuovo tipo
  - Interazione con gli altri tipi
- Complessita' della azioni e' all'interno dei nuovi tipi
  - Codice di alto livello (cliente) usa le funzionalita' con operazioni semplici

In soldoni…

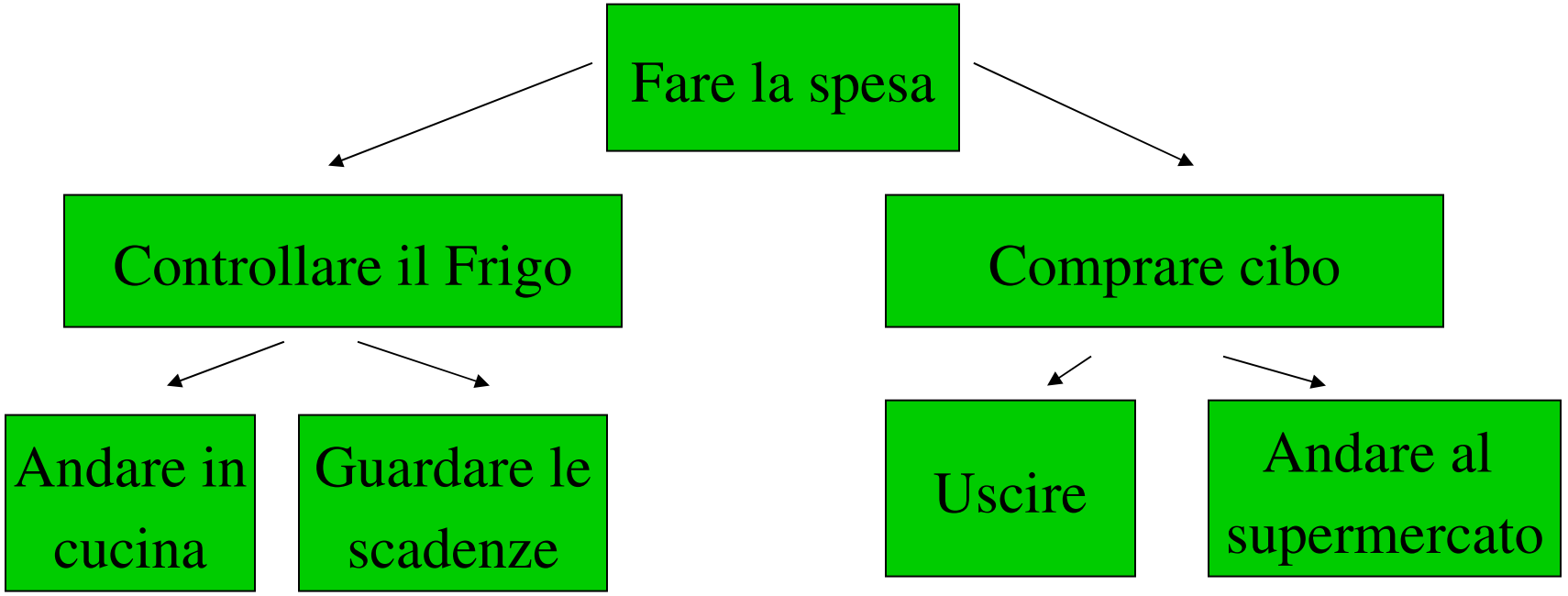
- Linguaggi procedurali:
  - L’idea è:
    - Operazioni complesse
    - Su oggetti semplici
- Linguaggi a oggetti:
  - L’idea è:
    - Operazioni semplici
    - Su oggetti complessi

Quindi ...

- Oggetti semplici
  - I numeri e i caratteri
- Azioni complesse
  - Somme sottrazioni moltiplicazioni e chi più ne ha più ne metta
  - Ma anche raggruppamento di azioni complesse
    - Funzioni!
    - Così come per eseguire il seno di un numero sono necessarie molte somme e sottrazioni (serie di Taylor), possiamo definire le nostre funzioni, matematiche e non

Tecnicamente …

- … questo modo di operare è chiamato Programmazione Top Down: scomporre il problema in pezzi indipendenti, e fare lo stesso per ciascuno di questi



Cominciamo …

- Non vi presento prima C e poi C++
- Non serve (anche se aiuta) sapere il C per passare al C++: la sintassi e’ quasi uguale.
- A volte e’ quasi dannoso! (eg uso di *malloc* e *free)*
- Partiamo invece da un C++ procedurale, per poi fare il salto verso la programmazione a oggetti

- … sintassi!

Gli oggetti elementari

- Un oggetto può essere/contenere
  - Un numero intero (int)
  - Un numero floating point (float)
  - Un carattere (char)
  - Una variabile logica vera o falsa (bool)
  - Una variabile “senza tipo” (void)
- Il C è dichiarativo (strong-typed): bisogna dichiarare ogni variabile che deve essere utilizzata (non e’ vero per tutti i linguaggi, eg python)
- È case sensitive: *pippo, PIPPO, PiPpO* sono variabili **diverse!**
- int a;
  rende a una variabile in grado di contenere un numero intero
- float a;
  la rende invece capace di contenere un numero floating point (quasi un reale, ma diverso!)

Nomi riservati

- Il nome di una variabile puo’ essere qualunque, tranne per alcuni nomi riservati dal C++

asm, auto, bool, break, case, catch, char, class, const, const_cast, continue, default, delete, do, double, dynamic_cast, else, enum, explicit, export, extern, false, float, for, friend, goto, if, inline, int, long, mutable, namespace, new, operator, private, protected, public, register, reinterpret_cast, return, short, signed, sizeof, static, static_cast, struct, switch, template, this, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar_t, while

- In generale, e’ ottima abitudine usare nomi esplicativi (SinTheta, NomeECognome, NumStudenti) piuttosto che nomi corti (a, i, n, aa,...), con l’unica eccezione per variabili interne di loop (vediamo dopo)

Insiemi di oggetti

- Array: se ho bisogno di 1000 numeri interi, non posso mettermi a definire
  - int a1,a2,a3,a4,a5 ... a1000;
- Posso invece fare
  - int a[1000];
- In questo modo posso poi accedere ai singoli elementi come
  - a[0] = 9; (nota [0] è il primo, non [1]!!)
  - a[999] = 66; (nota [999] è l'ultimo, non [1000]!!)
- Non chiamiamoli *vettori* (c'e' una classe vector che vedremo piu' avanti)

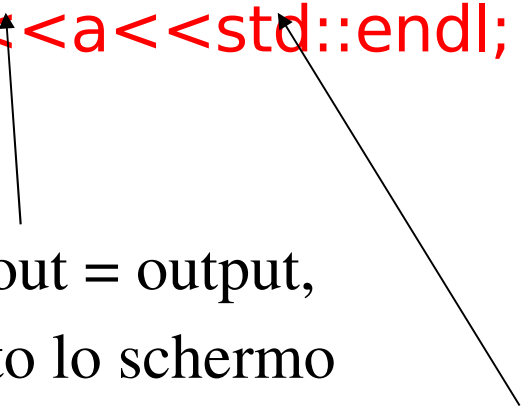
Operazioni elementari

- Questi oggetti permettono operazioni matematiche elementari “ovvie”:
  - si può assegnare un valore
    ```
    int a,b;
    a=7+5-b;
    ```
  - si possono effettuare operazioni di incremento
    ```
    int a;
    a=a-1;  //oppure  a-- o –a;
    ```
  - anche complesse
    ```
    int a;
    float b = (2-7.)*1.5+a*sin(12.);
    ```

// significa che da quel momento in poi tutto quello che c’è sulla riga è un commento

Interazioni con l'utente

- Stampare su schermo qualcosa
- Sintassi C++
  - int a=7;
  - std::cout << " ciao "<< 5<< " io sono a "<<a<<std::endl;



std::cout = output, di solito lo schermo

Stampa il valore di a

Std::endl dice di andare a capo

<< immaginatelo come una freccia: ciao va a finire sull'output

Sullo schermo: ciao 5 io sono a 7

Input

- Leggere informazioni dall'utente (generalmente la tastiera)
  - int a;
  - std::cin >> a;

Struttura di un programma

- Ci serve:
  - Un punto da cui il programma cominci e dove finisca
    - C/C++: la funzione main è una funzione speciale, la prima da cui parte l'esecuzione del codice
    - Caricare le funzioni necessarie dalle librerie del C++
  - Siamo arrivati al …PRIMO PROGRAMMA

```
#include <iostream>

int main(){
  int a;
  int b=5;

  std::cout <<" Inserisci un numero "<<std::endl;

  std::cin>> a;
  std::cout <<"il tuo numero incrementato di 5 fa "<< a+b
<<std::endl;
}
```

Carica le funzioni per interagire con l'utente

Proviamo …

Compilare

- Il comando per compilare è
  - g++
- Sintassi:
  - g++ -o NomeDelProgrammaDaCreare programma.c
  - E poi si può lanciare il programma con
    - ./NomeDelProgrammaDaCreare
- Proviamo

Le funzioni

- Abbiamo (quasi) visto che esistono funzioni elementari (seno, coseno ...)
- Possiamo definirne di nuove, e non solo di matematiche
- Esempio

Tipo di oggetto che viene restituito

Nome della funzione

Variabile data in input alla funzione

```
int somma5(int k){
   std::cout<<" Ho ricevuto in input "<<
k<<std::endl;
   return (k+5);
}
```

Quello che viene restituito

La uso!

definizione

```
#include <iostream>

int somma5(int k){
  std::cout<<" Ho ricevuto in input "<<k<<std::endl;
  return (k+5);
}

int main(){
  int a;

  std::cout <<" Inserisci un numero "<<std::endl;

  std::cin>> a;
  std::cout <<"il tuo numero incrementato di 5 fa
"<<somma5(a)<<std::endl;
}
```

utilizzo

Funzioni

- Le funzioni non devono essere necessariamente matematiche, possono anche non ritornare un valore
- In tal caso il C/C++ definisce il tipo di variabile void

```
void stampa(int k){
  std::cout<<" Ho ricevuto in input "<<
k<<std::endl;
//return;
}
```

Visibilità delle variabili

```
#include <iostream>

int somma5(int k){
  std::cout<<" Ho ricevuto in input "<<k<<std::endl;
  std::cout<<" Tra parentesi, b vale "<<b<<std::endl;
  return (k+5);
}

int main(){
  int a;
  int b=8;

  std::cout <<" Inserisci un numero "<<std::endl;

  std::cin>> a;
  std::cout <<"il tuo numero incrementato di 5 fa "<<
somma5(a)<<std::endl;
}
```

Provo ad utilizzarlo qui

Definisco b qui

Non va…

Regola generale

- Una variabile esiste ed è usabile solo all'interno dello stesso blocco di {} (scope) in cui è stata definita

```
int VarGlobale;

int main(){

        float VarLocale;

        {

                int VarInScope=8;

                VarLocale = 10 ; // OK

        }

        std::cout << VarLocale <<
std::endl;

        VarGlobale=5; // OK

        short VarGlobale = 10; // Nuova
var

        VarInScope=10; // ERRORE
```

Questi due b sono 2 variabili diverse

10

Errore

Troppo facile….

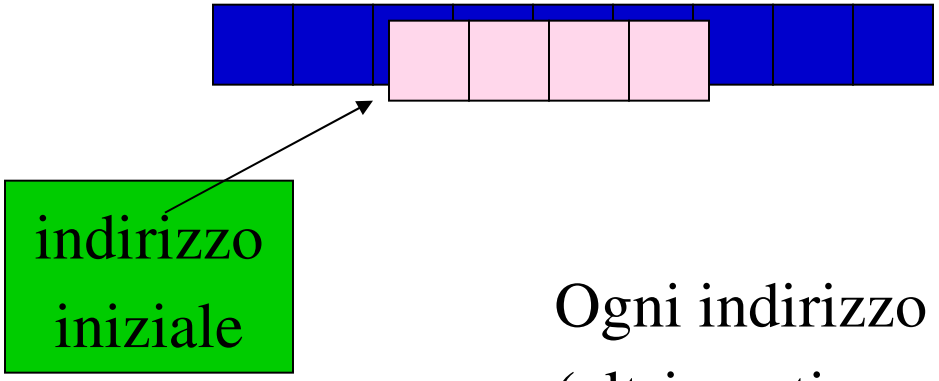
- Quindi entriamo direttamente nei casini del C/C++
  - I puntatori!

- Fra i vari tipi di variabili definiti da C/C++, ci sono le variabili di tipo puntatore ad un oggetto
  - int *a;

La variabile a può puntare ad una variabile di tipo intero

Pensiamo alla memoria …

- Sapete come è fatta la memoria di un PC?
  Una variabile intera è contenuta in 4 bytes (int) contigui

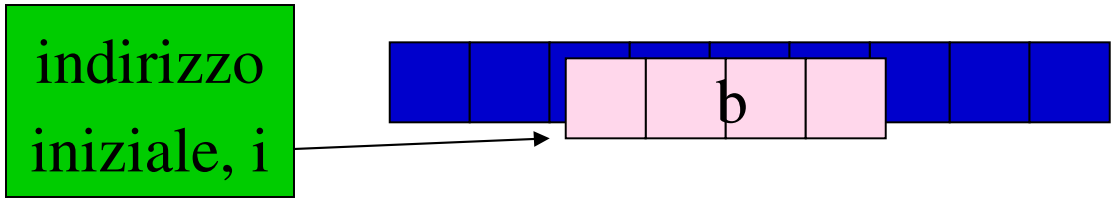


int a;

Ogni indirizzo di memoria ha un nome (altrimenti come farei a accedervi?)

Tipicamente questo è un numero esadecimale del tipo 0x46564afb55

- Se b è una variabile di tipo intero, sarà da qualche parte in memoria; supponiamo che occupi le locazioni fra 0x10000a e 0x10000d
- Quindi l'indirizzo di b è 0x10000a, e questo lo posso assegnare ad una variabile di tipo *b
- int b=8;
- int *i;
- i = &b;



&b rappresenta l'indirizzo di inizio di b

Operatore AddressOf

Altre operazioni con i puntatori

- int a;
- int *b;

- Sono operazioni valide

- b=&a;
- b=5;
- *b = 5;
- a=*b; // operatore di “dereferenziazione”

Se b di tipo puntatore, *b è la variabile a cui punta b

E quindi …

Ma perché soffrire tanto?

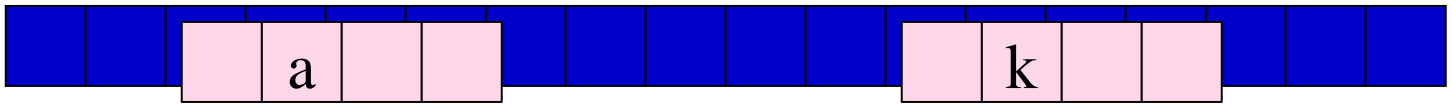
- I puntatori ci permettono di lavorare direttamente con le locazioni di memoria: perche' ci serve?
- Appunto, c'è almeno un buon motivo per cui sono necessari i puntatori.

- **Passaggio di puntatori nelle funzioni!**
-
- **Ci sono altri ottimi motivi che vedremo (polimorfismo)**

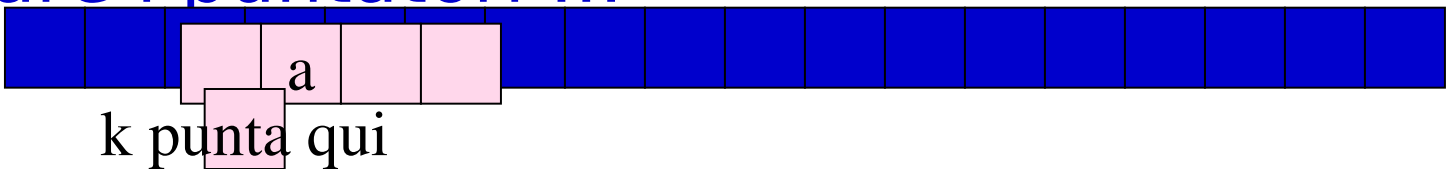
```
#include <iostream>

int somma5(int k){
  std::cout<<" Ho ricevuto in input
"<<k<<std::endl;
  k=k+5;
  return (k);
}

int main(){
  int a;

  std::cout <<" Inserisci un numero
"<<std::endl;

  std::cin>> a;
  std::cout <<"il tuo numero incrementato di 5
fa "<< somma5(a)<<std::endl;
  std::cout <<"il tuo numero vale adesso "<<
a<<std::endl;
```

Incremento la variabile che ho ricevuto

Cosa è successo?



- Quando passiamo dalla main alla funzione somma5, il valore della variabile a in main (di solito si dice main::a) viene copiato in somma5::k, ma le due variabili non coincidono!
- Quindi il problema diventa: come posso fare se voglio che una funzione possa modificare la variabile che passo?
- Usare i puntatori ...



Cioè...

- Invece di passare il valore della variabile, passo il valore del suo puntatore

```
#include <iostream>

int somma5(int* k){
  std::cout<<" Ho ricevuto in input "<<*k<<std::endl;
  *k=*k+5;
  return (*k);
}

int main(){
  int a;

  std::cout <<" Inserisci un numero "<<std::endl;

  std::cin>> a;
  std::cout <<"il tuo numero incrementato di 5 fa "<<
somma5(&a)<<std::endl;
  std::cout <<"il tuo numero vale adesso "<< a<<std::endl;

}
```

Quindi ...

- Se voglio che la mia funzione possa modificare la mia variabile, la passo per puntatore
- Se voglio che non possa, la passo per valore
  - Ma così sono costretto ad usare dovunque *
  - Il C++ introduce le Reference: una variabile è passata per puntatore anche se la sintassi è la stessa di una passata

```
#include <iostream>

int somma5(int& k){
  std::cout<<" Ho ricevuto in input "<<k<<std::endl;
  k=k+5;
  return (k);
}

int main(){
  int a;

  std::cout <<" Inserisci un numero "<<std::endl;

  std::cin>> a;
  std::cout <<"il tuo numero incrementato di 5 fa "<< somma5(a)<<std::endl;
  std::cout <<"il tuo numero vale adesso "<< a<<std::endl;

}
```

Puntatori e Reference

- Sembrano quasi la stessa cosa
  - Entrambi servono a gestire l'indirizzo di memoria che contiene un oggetto
- Differenza fondamentale:
  - Pointer: e' una locazione di memoria che puo' non contenere nulla
  - Reference: e' una locazione di memoria che **deve** contenere un oggetto definito
- Posso avere pointer to void
- Reference e' un alias, un altro nome **per un oggetto che gia' esiste!**

```
#include <iostream>
int main() {
  int a; // oggetto (int) a
  a=10;

  int* pa; // puntatore ad oggetto a : dichiarato ma non inizializzato
  pa=&a;  // assegno puntatore pa all'indirizzo di a

  std::cout << "A vale " << a << " pa:" <<pa << " *pa:" << *pa <<
std::endl;

  int& ra; // reference to oggetto a ERRORE!!

  int& ra = a; // Ok dichiaro e assegno

  std::cout << "A vale " << a << " ra:" <<ra << std::endl;

  a = 20;  // cambio a ma anche ra!

  std::cout << "A vale " << a << " ra:" <<ra << std::endl;
  return 0;
}
```

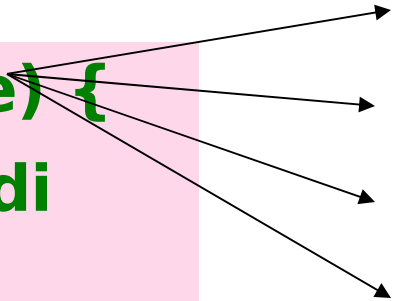
Operatori

- Assegnazione : a=10; NB e’ diverso da a==10;
- Aritmetici: + - * / %
- Mod+Assegnazione +=, -=, *= , ...
- Incremento(decremento) A++, --B
- Confronto : ==, !=, >, <, >=, <=
- Logici: !, ||, &&
- Condizionale: a>b? a : b
- Comma (,):   a = (b=3 , b++)
- ...

Esecuzione condizionale

- Altro elemento importante dei linuaggi di programmazione è la possibilità di avere un'esecuzione condizionale

```
if(condizione) {
        comandi
}
else if{
comandi }
else{ comandi }
```



condizione è genericamente una struttura che abbia valore di bool

a==7

a>7

a<=4

a!=4

o una combinazione logica di condizioni

Ca && Cb

Ca || Cb

((!Ca) && Cb)

Cicli ...

- Un’altra cosa che capita spesso di fare:
  - Supponiamo di dover sommare il vettore di prima

- O faccio
  - int somma;
  - somma = a[0]+a[1]+ ... a[999];
    - Noioso ..
- O uso:

Cicli

- For:

```
for (int i=0; i<10; i++)
  std::cout<<i<<std::endl;
```

- While:

```
int i=0;
while (i<10) {
  std::cout<<i<<std::endl;
  i++;
}
```

- Do While:

```
int i=0;
do {
  std::cout<< “do-while” <<std::en
} while (false);
```

Esegui il ciclo facendo variare i fra 0 fino a che è minore di 10, incrementatndo tutte le volte di 1

Esegui il ciclo mentre i è minore di 10

Esegui il ciclo sempre almeno la prima volta

Switch case

- Una alternativa a lunghi *if elseif elseif ... else*

```
switch (x) {
  case 1:
    cout << "x is 1";
    break;
  case 2:
    cout << "x is 2";
    break;
  default:
    cout << "value of x unknown";
  }
```

Lettura / scrittura dal disco

- Ultima cosa che manca per poter scrivere un programma elementare

- Molto semplice in C++: fare esattamente come se fosse una tastiera (se sto leggendo) o lo schermo (se sto scrivendo)

File streams

```
#include <iostream>
#include <fstream>

int main(){
  int a;
  std::ofstream out;
  out.open("pippo");

  out<<12<<std::endl;
  out.close();

  std::ifstream in;
  in.open("pippo");
  in >> a ;
  in.close();

  std::cout << "il tuo numero era " << a <<
std::endl;
}
```

if/ofstream e' un particolare stream che scrive su file input/output

Notate che da questo si capisce che cin e cout sono solo stream speciali che noi non dobbiamo aprire

Namespace…

- Francamente mi sono stufato di scrivere sempre std:: davanti a tutto
  - Perché è così ?
  - Come si evita ?

- In C++ i vari oggetti (files, e anche altro vedremo) sono suddivisi in compartimenti (namespace)
  - Posso avere una variabile a definita in un compartimento, e questa sarà diversa dalla variabile a definita in un'altro
  - int pippo::a;
  - int pluto::a;

In C++

- Le funzioni più usate sono definite nel namespace std, per cui cin e cout, per esempio, devono essere chiamati come std::cin e std::cout;

- Oppure: posso dire una volta per tutte al compilatore che anche se non specifico nulla, assuma che io voglio lavorare in un namespace definito
  - using namespace std;

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

namespace first{
  int var = 5;
}

namespace second{
  double var = 3.1416;
}

int main () {
  cout << first::var << endl;
  cout << second::var << endl;
}
```

Basta…

- Con la programmazione procedurale
- Cominciamo con la programmazione a oggetti
- Perche’ e’ utile?
- Partiamo con un esempio concreto

Calcolo del modulo di un vettore

- Supponiamo di avere una funzione che calcolo il modulo di un vettore
- Io la voglio chiamare dando in input x,y,z del vettore, per cui avrò una funzione del tipo

```
float modulo(float x, float y, float
z){
  return (x*x+y*y+z*z);
}
```

Ma poi …

- Un giorno mi accorgo che è molto più veloce per quello che serve a me se uso coordinate cilindriche, allora dovrei riscrivere

```
float modulo(float r,
float phi, float z){
  return (r*r+z*z);
}
```

- E devo cambiare il modo di chiamarla dovunque

- Peggio ancora: se la funzione modulo non è scritta da me, e io la uso solamente, ogni volta che il suo programmatore cambia idea io devo cambiare molte cose nel mio codice.

OO / C++

- Per un progetto grande e che debba vivere a lungo, questi sono grossi problemi.
- La programnmazione ad oggetti, fra le altre cose, permette di dare una risposta a queste esigenze.

- (ci sono molti altri motivi per cui si pensa sia meglio, ma io sono un fisico delle alte energie e purtroppo ho a che fare con programmi di simulazione e ricostruzione O(1.000.000) righe, e questa feature del C++ mi aiuta moltissimo)

Come rendere mantenibile un codice?

- Separare bene le varie componenti
- Essere strettamente aderenti ad un interfaccia:
  - se il pacchetto A parla col pacchetto B, deve farlo in un modo ben stabilito che NON cambi al cambiare del codice interno di A e B
  - Questo vuol dire che devono esistere 2 livelli di codice: quello pubblico, stabile, e quello privato, non conosciuto all' esterno del pacchetto
- Quando esistono vari pezzi di codice che fanno la stessa cosa, devono essere intercambiabili
- Se ho due pacchetti che fanno la stessa cosa, le parti di codice in comune devono essere riutilizzate
  - curare i bug solo una volta!

Programmazione a oggetti

- Promette tutto questo, e anche di più!
- Filosofia: in un linguaggio procedurale contano più le azioni che gli oggetti su cui vengono effettuate, che sono sempre semplici.
- In un linguaggio a oggetti, contano più gli oggetti, e le azioni hanno senso solo in quanto associate ad un oggetto

C: **void calcolaRumore(oggetto,…)**

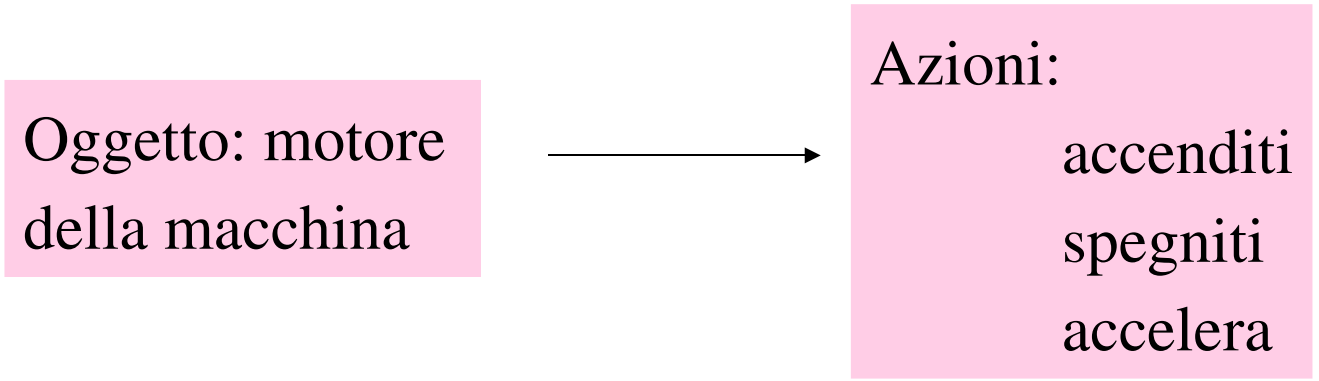
C++: **oggetto.calcolaRumore(…)**

è l'oggetto stesso che possiede la caratteristica di saper calcolare il SUO rumore

Quindi …

- Sposto la complessità dal codice procedurale, e quindi dalla definizione di operazioni complesse alla creazione e definizione di oggetti complessi.

- L'oggetto semplice che avevo in C, l'intero, diventa un oggetto complesso che sa operare azioni complesse




430
1050
700
200
IRON AGE
Menu
Chat
Phalanx
120/120
Attack 19
Armor 5
Range 0

Lo abbiamo già visto

- Ma io avevo fatto finta di nulla:

```
#include <iostream>
#include <fstream>

int main(){
  int a;
  std::ofstream out;
  out.open("pippo");

  out<<12<<std::endl;
  out.close();

  std::ifstream in;
  in.open("pippo");
  in >> a ;
  in.close();

  std::cout << "il tuo numero era " << a <<
std::endl;
```

L'oggetto di tipo ofstream e ifstream che sa:
- aprirsi
- chiudersi

Nel Linguaggio a Oggetti

- Io definisco delle classi
  - La classe del motore
  - La classe del vettore
- Dando a ciascuno di essi delle capacità
  - Sapersi aprire e chiudere

- E poi “istanzio” (creo) degli oggetti della classe

Una definizione complicata..

- Cerchiamo di distinguere adesso fra Classe e Oggetto
- Una classe è la categoria astratta: la sedia
- Un oggetto è un caso particolare concreto di classe: questa sedia
- In C++, io definisco una classe, nel senso che spiego come debba essere fatta senza che necessariamente ne voglia una: la classe è una specie di “manuale della sedia”; funziona anche senza sedia!
- Poi, seguendo il modello della classe, posso definire quanti oggetti voglia: uso il manuale per costruire 12 sedie reali, anche diverse fra loro.
  - Ho solo una classe sedia
  - Ho 12 oggetti sedia

Definizione di una classe

- Definiamo la nostra prima classe

```
class quadrato{
 public:
 quadrato(float lato){
    miolato = lato;
  }
 quadrato(){
    miolato = 12;
  }
  float perimetro(){
    return miolato*4.;
  }
 private:
  float miolato;
};
```

Definisco la classe di tipo quadrato

Definisco tre metodi, due dei quali con lo stesso nome della classe

La classe contiene anche una variabile float

Studiamola pezzo per pezzo

- **class quadrato{ .. };**
  - **Niente altro che la definizione**

- **public:**
  - **Introduce la sezione pubblica, cioè la parte di codice che sarà accessibile e visibile all'utente della classe;**

- **private:**
  - **Introduce la sezione privata, cioè quella che l'utente della classe non può e non deve saper conoscere**

Incapsulamento

- Il codice viene diviso in due parti abbastanza stagne: pubblico e privato
- Pubblico: definisce la cosiddetta interfaccia, cioè definisce i metodi di comunicazione fra classi, e tendenzialmente deve cambiare poco
  - idealmente: un programmatore per ogni classe, se cambio la parte pubblica devo andare nell' ufficio accanto e comunicarlo al collega
- Privato: gli sporchi dettagli della mia implementazione
  - se li cambio, non importa che lo dica a nessuno; finchè non inserisco bug sono solo affari miei

Construttore

- quadrato(float lato){
  - C’è un metodo con lo stesso nome della classe; questo è chiamato costruttore e è quello chiamato quando un oggetto di questa classe viene chamato
  - Ci possono essere più costruttori definiti in una classe; quello che sarà usato dipende da come istanzierò l’oggetto

```
quadrato ilmioquadrato;
quadrato ilmioquadratone(72);
```

Uso il costruttore senza parametri

Uso il costruttore con 1 parametro

Data Member

- **float miolato;**
  - **È una variabile privata della classe, dall'esterno non è possibile né vederla né cambiarla**

  NB: se io metto le variabili pubbliche, allora l'incapsulamento e' perso!
  Data Member solo **privati!**

- **Vediamola all'opera**

Ora vediamo davvero i pregi del C++

- Decido che non voglio salvarmi il lato, ma il perimetro

```
class quadrato{
 public:
 quadrato(float lato){
     mioperimetro = lato*4;
  }
 quadrato(){
     mioperimetro = 12*4;
  }
  float perimetro(){
     return mioperimetro;
  }
 private:
  float mioperimetro;
};
```

Cosa devo cambiare nel main???

NULLA!
Il codice dell'utente è insensibile ai cambiamenti interni della classe!

Il distruttore

- Così come quando l’oggetto viene creato viene eseguito un costruttore, quando l’oggetto viene distrutto
- Si chiama ~nome delle classe
- Proviamo …

Ereditarietà - 1

- Caratteristiche evolute del C++: OO si basa molto su questa feature
- Supponiamo che in un mio programma io debba lavorare con oggetti di tipo quadrato e rettangolo.
- Visto che un quadrato non è altro che un rettangolo speciale, mi posso aspettare che alcuni metodi siano uguali
  - == stesse linee di codice
  - Per esempio, se ho una funzione che mi ritorna il numero di lati, la risposta sarà 4 sia per il qudrato che per il rettangolo

Criterio di economia

- Perchè scrivere 2 volte lo stesso codice?
- Se poi lo modifico, devo modificarlo in due punti!

- In questo caso, voglio poter riutilizzare per la classe **quadrato** alcuni dei pezzi di codice del **rettangolo**

- Scriverò
  - class quadrato : public rettangolo { ... };

Ereditarieta’

- Ci sono diversi tipi di ereditarieta’: la piu’ comune e’ l’ereditarieta’ *public*
- C’e’ anche *private* ma per ora lo ignoriamo
- *class Quadrato: public rettangolo {...};*
- Ereditarieta’ pubblica significa “isa”
- Cioe’ un *quadrato* e’ un (*“isa”*) *rettangolo*

E’ proprio vero??

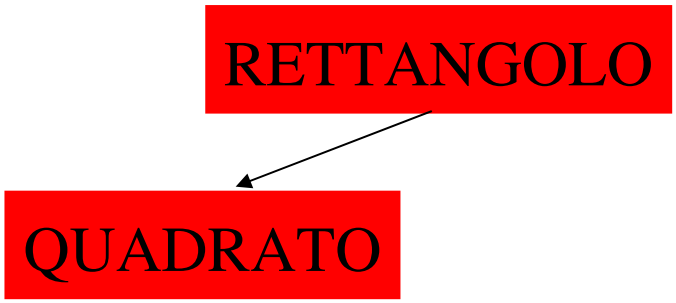
Cioe’: e’ vero che tutte le operazioni che posso fare su un rettangolo le posso fare anche su un quadrato?

- Per un rettangolo ha senso raddoppiare la base lasciando l’altezza uguale
- Non ha senso per un quadrato

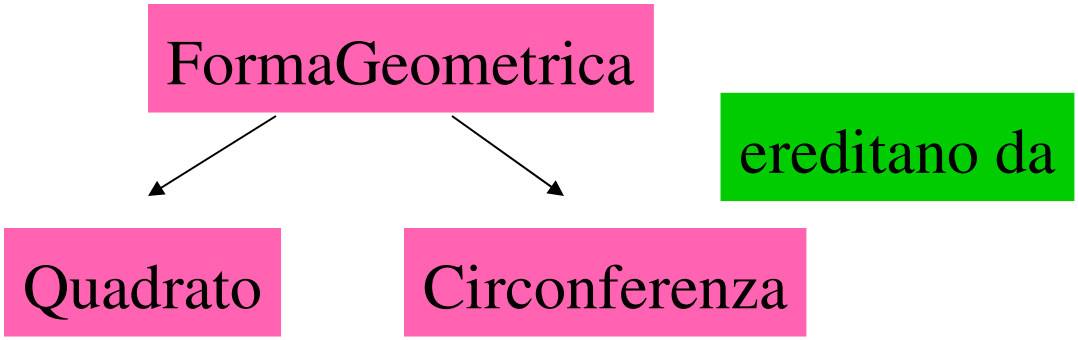
- In OO: un quadrato NON e’ un rettangolo !!
- Devo ripensare la mia ereditarieta’
- cioe’ rifare il design delle mie classi o architettura
- idee???

Ereditarietà - 2

- Ancora meglio: pensiamo ad una circonferenza e ad un quadrato
- Non posso far derivare l'uno dall'altro, ma
- Entrambi hanno perimetro e area, perché
  - Entrambi sono forme geometriche
  - Hanno la stessa interfaccia!
- Posso usare un tipo diverso di ereditarietà

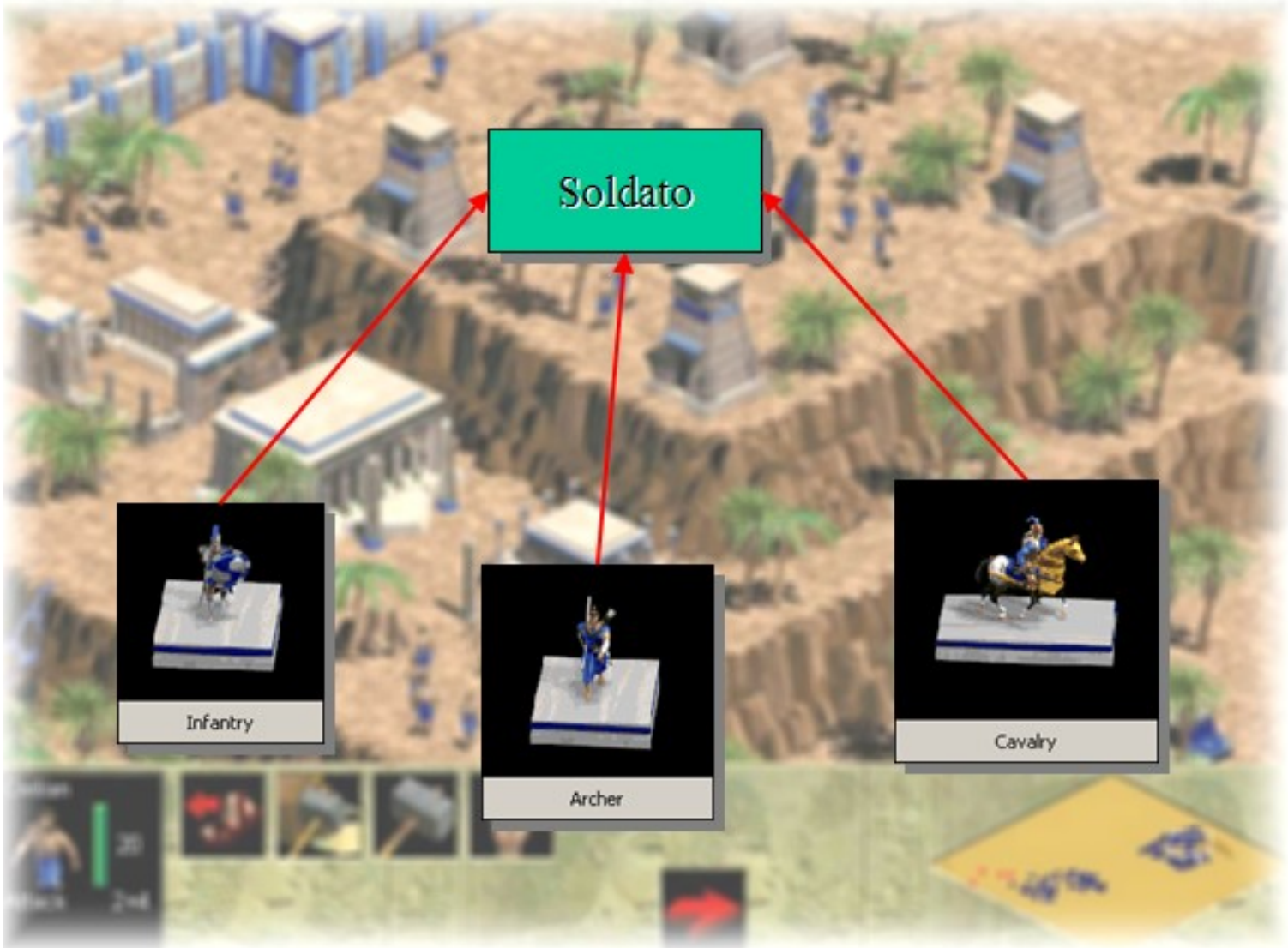


Ereditarietà



- FormaGeometrica può definire tutte le caratteristiche astratte (di sicuro la forma ha un perimetro ma non esiste un modo (comandi!) per calcolare il perimetro di una forma geometrica non meglio identificata)
- Quadrato e Circonferenza possono specializzare i metodi (Quadrato saprà calcolare il suo perimetro, in un modo che non andrebbe bene per la circonferenza)

Esempio OO



Polimorfismo

- Come abbiamo visto, ha senso parlare per una FormaGeometrica di perimetro, però il modo di calcolarlo per Quadrato e Circonferenza deve essere diverso.
- In tal modo, chiamare perimetro() su di un quadrato, farà agire codice diverso rispetto a chiamarlo su di una circonferenza.
- Però, all'utente NON INTERESSA! Sa solo che chiamando perimetro(), otterrà sempre la rispostra giusta (a meno di bugs...)
- L'interfaccia d FormaGeometrica e' sempre la stessa, l'implementazione cambia.

```
#include <iostream>

using namespace std;

class FormaGeometrica{
 public:
  virtual float perimetro() = 0;
  virtual float area() = 0;
};


int main () {
  FormaGeometrica pippo;
  cout <<"Ciao "<<endl;
}
```

Virtual = delle classi erediteranno da questa e reimplementeranno questo metodo

=0 : nella classe FormaGeometrica non è spiegato come calcolarle (e come potrebbe!)

Crash! Questo non lo posso fare; FormaGeometrica serve solo a definire cosa dovranno avere in comune tutte le

- Una volta definita la classe astratta, posso definire le classi che ereditano, quadrato e circonferenza

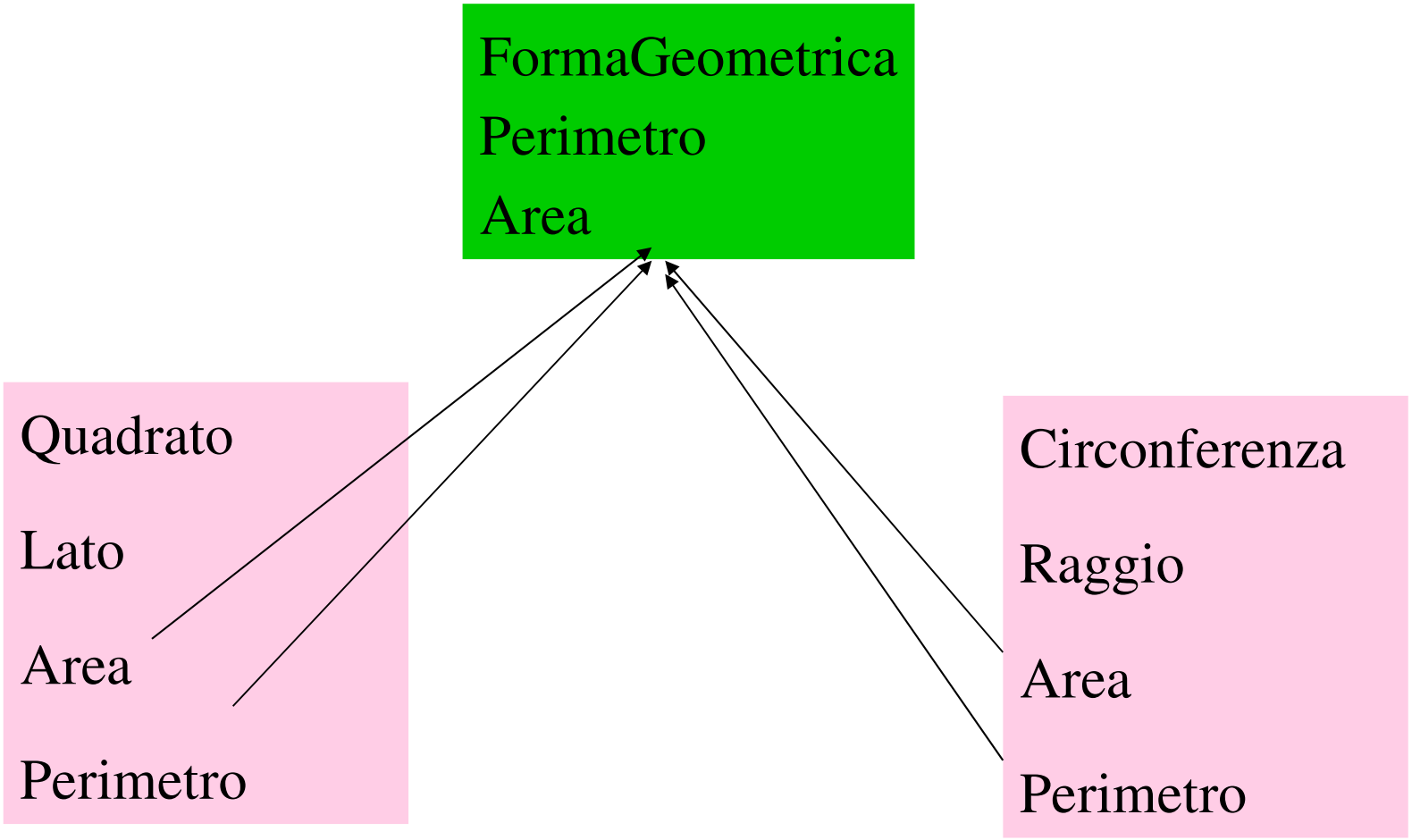
Esempio

```
class FormaGeometrica{
public:
  virtual float perimetro() = 0;
  virtual float area() = 0;
};
class Quadrato: public FormaGeometrica{
public:
  Quadrato(){myLato=0;}
  Quadrato(float lato){myLato=lato;}
  float lato(){return myLato;}
  virtual float perimetro(){return 4*myLato;}
  virtual float area(){return myLato*myLato;}
private:
   float myLato;
};
class Circonferenza: public FormaGeometrica{
public:
  Circonferenza(){myRaggio=0;}
  Circonferenza(float r){myRaggio = r;}
  float raggio(){return myRaggio;}
  virtual float perimetro(){return
4*3.14*myRaggio;}
  virtual float area(){return
3.14*myRaggio*myRaggio;}
private:
   float myRaggio;
```

classe astratta

classe concreta

classe concreta


FormaGeometrica
Perimetro
Area
Quadrato
Lato
Area
Perimetro
Circonferenza
Raggio
Area
Perimetro

Ereditarieta’ pubblica

- Pure virtual method *virtual float perimetro() = 0;*

Deve essere reimplementato dalle classi derivate

- Virtual method *virtual string colore();*

Puo’ essere reimplementato, ma c’e’ un default

- Non virtual method *int numeroDimensioni();*

Ultima cosina per oggi

- Non per altro, così la settimana prossima si parte con qualcosa di più divertente…
- Per ora, per istanziare un oggetto, facevamo qualcosa del tipo
  - int I;
  - Quadrato q(23.);
- C’è un altro modo: creazione usando l’operatore new

- Posso ottenere un puntatore all'oggetto invece che l'oggetto
- Invece di
  - Quadrato q(1);
  - Quadrato *q = new Quadrato(1);

- Come si usa?
- E chi lo distrugge?

```
int main () {
  Quadrato * q = new Quadrato(3.);
  Circonferenza c(66.);
  cout <<"La circonferenza ha raggio "<<
       c.raggio()<<" perimetro "<<
       c.perimetro()<<" e area "<<c.area()<<endl;
  cout <<"Il quadrato ha lato "<<
       (*q).lato()<<" perimetro "<<
       (*q).perimetro()<<" e area "<<(*q).area()<<endl;
  cout <<"Il quadrato ha lato "<<
       q->lato()<<" perimetro " <<
       q->perimetro() <<" e area "<<q->area()<<e

  delete q;
}
```

Lo creo

Lo uso (ricordate, è un puntatore)

In C++ (*q). si può anche scrivere q->

Lo cancello

New / delete

- Cosa fa?
  - Alloca la quantita' di memoria sufficiente per contenere l'oggetto
  - Chiama il construttore e mette l'oggetto creato nella memoria
  - Ritorna un puntatore alla memoria
- ATTENZIONE
  - Voi avete allocato la memoria: quindi e' vostra responsabilita' liberarla quando avete finito! delete
  - In C usavate malloc (e free: ci sono anche in C++ ma **NON VANNO MAI USATI!! perche?**

New vs malloc

- Molto semplice: *malloc* non sa cosa sia un costruttore, mentre *new* si.
- Quadrato *P = new quadrato();

Crea un oggetto *quadrato* e mi ritorna il pointer

- Quadrato P = static_cast<quadrato*>(malloc(sizeof(quadrato))

Alloca lo spazio per un *quadrato* ma **non** lo costruisce! Dovrei creare a parte un *quadrato* e poi metterlo a mano in quella memoria. Che senso ha??

Free/delete

- Idem come sopra: free non chiama il distruttore, delete si
- Supponete che la mia classe quadrato nel costruttore faccia un **lato = new float(10);*
- Metto nel distruttore *delete lato;*
- Se uso *free* il distruttore non viene chiamato e nessuno libera la memoria creata con new
- **In C++ non si usa mai malloc/free**

Static type e Dynamic Type

- FormaGeometrica *a = new Quadrato(10);
- Che cosa e' (*a)?? E una FG o e' un Q??
- Tipo Statico e' quello dichiarato:
  - FormaGeometrica *a;
- Tipo Dinamico e' quello che e' effettivamente l'oggetto
  - In questo caso (*a) e' un Quadrato!

binding

- Se un metodo e' *virtual*
  - *a->MetodoVirtual();*
  - Chiama il metodo del tipo dinamico (quadrato::MV();)
  - dynamic binding
- Se un metodo non e' *virtual*
  - *a->MetodoNonVirtual();*
  - Chiama il metodo del tipo statico (FG::MNV();)
  - Static binding
- Proviamo!

Referenze

- Se non avete capito nulla, mi dispiace
- Se sapevate già tutto, mi dispiace
- Se avete capito il senso di tutto, mi sembra ottimo!
- Come tutti i linguaggi, lezioni teoriche hanno poco senso. Provate, provate, provate! Stefano.Lacaprara@pd.infn.it per aiuti...
- Vi metto una lista di referenze carine, principalmente tutorial e corsi di C e C++ su internet

Reference

- Tutorial C++:
  - http://www.cprogramming.com/tutorial.html
  - http://www.cplusplus.com/doc/tutorial/
- Tutorial C
  - http://pmode.impazz.it/tuts/cansi1.htm
  - http://www-math.unice.fr/laboratoire/help/C/ctutor/ctutor.html

Libri

- Se proprio ci tenete ☺
  - C:
    - Brian W. Kerningham, Dennis M. Ritchie “Linguaggio C - 2a edizione (ANSI)” Addison-Wesley
  - C++:
    - Koenig, Moo, “Accelerated C++: Practical Programming by Example”, Addison-Wesley
  - *La Bibbia del C++ (c'e' tutto, molto avanzato!)*
    - *Brian Stroustrup,* “The C++ Programming language, third ed.”, Addison-Wesley
  - http://www.mindview.net/Books/TICPP/ThinkingInCPP2e.html
  - http://www.icce.rug.nl/documents/cplusplus/

Libri (2)

- Introduzioni
  - A.Koenig & B.E. Moo Accelerated C++: Practical Programming by Example, Addison-Wesley
  - S.B.Lippman: C++ Primer, Addison-Wesley
  - I.Pohl: Object-Oriented Programming Using C++, Addison-Wesley
- Intermedi
  - E.Gamma et al.: Design Patterns Addison-Wesley
  - Scott Meyers: Effective C++, Addison-Wesley
  - Scott Meyers: More Effective C++, Addison-Wesley
  - Scott Meyers: Effective STL, Addison-Wesley
  - N.M.Josuttis: The C++ Standard Library - A Tutorial and Reference, Addison-Wesley
  - R.B.Murray: C++ Strategies and Tactics, Addison-Wesley

Libri (3)

- Avanzati
  - G.Booch: Object-Oriented Analysis and Design Addison-Wesley
  - Bjarne Stroustrup: The C++ Programming Language Addison-Wesley
  - A.Alexandrescu: Modern C++ Design Addison-Wesley