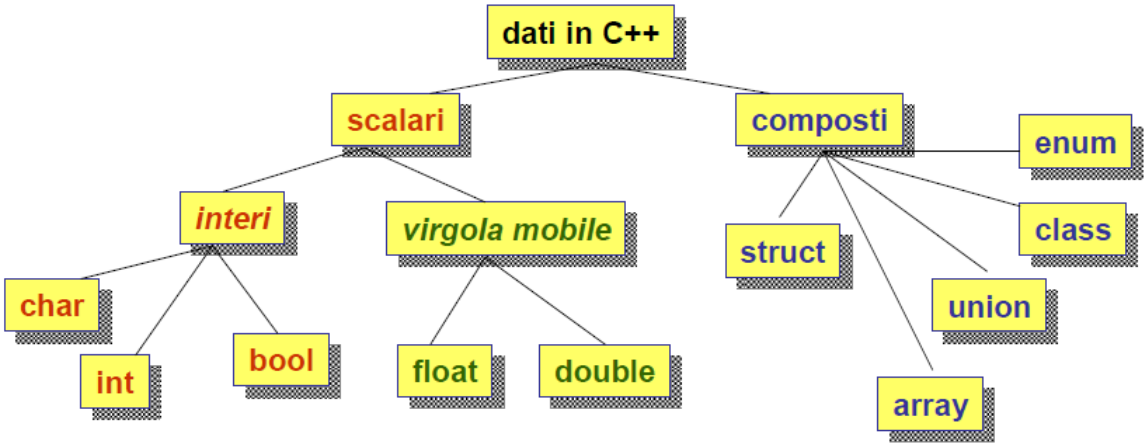
Corso di Laboratorio 2 – Programmazione C++

Silvia Arcelli

20 Ottobre 2014

Tipi non Predefiniti

- Abbiamo visto:
  - Tipi elementari (int, float, char,...)
  - Tipi composti (arrays e tipi non predefiniti)



- C++ permette la definizione di nuovi tipi da parte dell'utente:
  - enum, union, struct (anche in C) e class

Tipi non Predefiniti: enum

- Tipo enum: utile per la leggibilità del programma nel rappresentare casi distinti (i nomi sono più descrittivi dei numeri):

```
enum tag {nome1,nome2,nome};
enum tag {nome1=value1,nome2=value2,nomeN=valueN};
enum tag {nome1,nome2=value,nomeN};
```

- Il nome (tag) che segue la keyword enum è opzionale
- Gli enumeratori nome1,nome2,... Devono essere assegnati a interi costanti.
- Il primo enumeratore, se non specificato esplicitamente, prende valore 0 e i successivi gli interi seguenti.
- Agli enumeratori non corrispondono locazioni di memoria, per cui non possono essere riassegnati una volta definiti, né si possono usare i loro indirizzi.

Tipi non Predefiniti: enum

Definizione del tipo enum:

```
enum color {red,blue,green};

enum {max=100}; //enumeratore anonimo

enum position {center=0,bottom,top=max}; //ok, posso usare enumeratori
                                         // precedentemente definiti

enum {value=25.3}; //illegale, solo interi!

const int  max_value=30;  //questo è ok
enum {max=max_value};

int  max_value=30;  //questo no, max_value deve essere const
enum {max=max_value};
```

Tipi non Predefiniti: enum

Dichiarazione e assegnazione di variabili di tipo enum:

```
color shirt; position pos;
shirt=blue; //ok
shirt=3; // illegale, il tipo intero non converte a enum type

pos=bottom; //ok
pos=red;//illegale, enumeratore non corrispondente (C++ è type safe)

int num=max;//ok, il tipo enum converte a intero.
double vec[max]; // ok, fa la funzione di un const int
```

Tipi non Predefiniti: enum

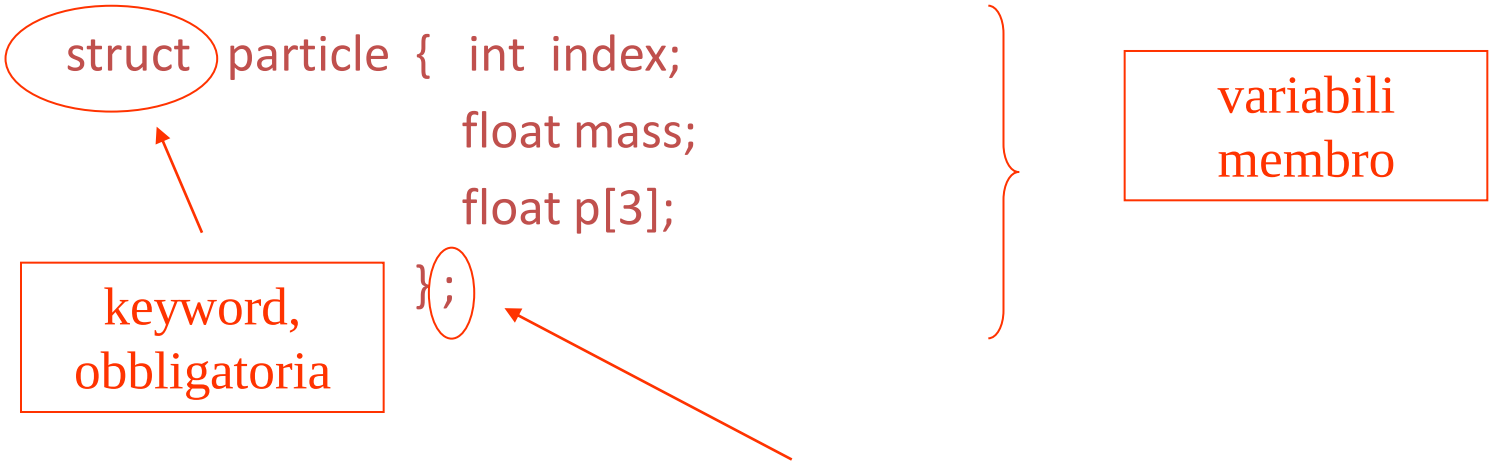
```
#include <iostream>
using namespace std;
float get_mean(double val[]){ /*...*/;}
float get_rms(double val[]){ /*...*/;}

int main(){
enum choice { mean = 1, rms, mean_rms};
double val[100], x[2]; int i;
cout << "please select your choice: 1= mean, 2=rms, 3=both" << endl;
cin >>i;
switch( i ){
case mean: x[0]=get_mean(val); break;
case rms: x[1]=get_rms(val); break;
case mean_rms: x[0]=get_mean(val); x[1]=get_rms(val); break;
} return 0; }
```

Tipi non Predefiniti: struct

- Come gli array, in C++ le strutture sono gruppi di dati; a differenza dagli array, i singoli componenti di una struttura possono essere di tipo diverso. Utilizzato per incapsulare un insieme di variabili diverse ma concettualmente correlate in un tipo di dato composto.

Esempio di definizione di una struttura:



- Dopo la parentesi graffa di chiusura, è obbligatoria la presenza del punto e virgola .

Tipi non Predefiniti: struct

- Dopo la *parola-chiave* struct segue l'identificatore (tag) della struttura, che è opzionale

- Fra parentesi graffe, l'elenco dei componenti della struttura (membri); ogni membro può essere di qualunque tipo, anche array o puntatore o a sua volta una struttura (si possono anche nidificare).

```
struct   momentum  {   float p[3];     } ;

struct   particle  {        int  index;
                                  float mass;
                                  momentum ptot;
                             } ;
```

Tipi non Predefiniti: struct

- La definizione non alloca memoria. Per dichiarare una variabile associata alla struttura (il che, invece, alloca memoria), la sintassi è:

```
struct particle  aParticle; // una variabile di tipo struct particle
struct particle *pToaParticle; // un puntatore a ....
struct particle particles[10]; // un vettore di....
```

- in C++ si può omettere struct in dichiarazione, cioè si può scrivere direttamente:

```
particle  aParticle;
```

Tipi non Predefiniti: struct

- N.B.Se nella definizione è stata omessa la tag l'identificatore, è possibile dichiarare variabili del tipo definito dalla struttura solo contestualmente alla definizione:

```
struct   {      int  index;
                float mass;
                float p[3];
} aParticle1, aParticle2[3] ;
```

Dichiarazione di due variabili aParticle1, aParticle2[3]

- In questo caso non è possibile dichiararne altre dello stesso tipo nel corso del programma, a meno di non usare un typedef:

```
typedef  struct   { int  index;  float mass; float p[3];} Particle;
Particle  aParticle1;
Particle aParticle2[3];
```

Tipi non Predefiniti-typedef

Si può creare un “alias” per un tipo esistente (predefinito o generato dal programmatore) usando la keyword typedef*:*

```
typedef   int Vec[10];  // array di interi con 10 elementi
Vec  myVec={1,2,3};

typedef struct {int giorno; int mese; int anno;} Data;
Data d1;
Data d2;
```

N:B: typedef non crea un nuovo tipo, aumenta la leggibilità del codice

Tipi non Predefiniti: struct

- L'inizializzazione dei membri della struttura non è consentita nella definizione.

- Si possono inizializzare/assegnare valori ai membri della struttura solo in relazione alla variabili che sono state dichiarate di quel tipo: ad esempio:

```
struct particle aParticle={1,3.,-1.5,4.3,5.5};
```

variabili

```
particle aParticle;
aParticle.index=1;
aParticle.mass=3.;
aParticle.p[0]=-1.5;
aParticle.p[1]=4.3;
aParticle.p[2]=5.5;
```

puntatori

```
particle *aParticle;
aParticle->index=1;
aParticle->mass=3.;
aParticle->p[0]=-1.5;
aParticle->p[1]=4.3;
aParticle->p[2]=5.5;
```

Tipi non Predefiniti: struct

- in C++ le strutture possono incorporare come membri anche delle funzioni:

```
struct   particle  { int  index;
                     float mass;
                     float p[3];
                     float E(){
              return  sqrt(mass*mass+p[0]*p[0]+ p[1]*p[1]+ p[2]*p[2]);}
                    } ;
```

- Per invocarle, stessa sintassi come per un membro qualsiasi:

```
aParticle.E(); //calcola l'energia totale della particella (c=1)
```

Tipi non Predefiniti: union

- E' molto simile alla struttura, e serve a raggruppare in un unico costrutto variabili di diverso tipo, con la condizione che le variabili non possono mai essere utilizzate contemporaneamente.

- Si comunica al compilatore di allocare memoria corrispondente alla variabile membro più grande, e in diverse fasi del programma si utilizza la stessa memoria per rappresentare la variabile in uso. La sintassi è identica a struct:

```
union   particle  { int  index;
                    float mass;
                    float p[3];} ;
```

- Utile nel caso si acceda ai singoli membri separatamente in fasi diverse del programma, e si voglia limitare l'utilizzo di memoria (qui per un esempio)

Programmazione a Oggetti

- Finora differenze C/C++ limitate ad alcune estensioni (overloading delle funzioni, reference,...). Nuove caratteristiche che veramente differenziano il C++ dal C:
  - Programmazione a oggetti:
    - Incapsulamento *(data hiding)*
    - Polimorfismo*:* *ereditarietà e programmazione generica (template)*

- Da un' ottica procedurale ad un'ottica di programmazione a oggetti: nuovi concetti per tipo di dato e operazioni sui dati:
  - Ottica procedurale: entità passive (dati) su cui agiscono delle entità attive (procedure,operazioni sui dati)
  - Ottica OOP: dati ed operazioni sui dati sono fusi in entità attive (gli oggetti) che interagiscono fra loro

Programmazione a Oggetti

Matematica dei numeri complessi in ottica procedurale:

```
struct Complex {
double Re;
double Im;
};

void Print(Complex& Val)
{ cout << Val.Re << " + i" << Val.Im << endl; }
double Abs(Complex& Val)
{ return sqrt(Val.Re*Val.Re + Val.Im*Val.Im);}

int main() {
Complex C; C.Re = 0.5; C.Im = 2.3;
cout << C.Re << " + i" << C.Im;
Print(C); cout << Abs(C) << endl;
return 0; }
```

- Rappresentazione di Complex separata dalle operazioni su di esso definite:
  - la struct contiene gli attributi del numero complesso e definisce il tipo di dato
  - due function "esterne" definiscono le operazioni su di esso
- Si può accedere alla rappresentazione dei dati direttamente e replicare qualsiasi operazione definita nelle function

Programmazione a Oggetti

- Al fine del mantenimento/estendibilità/portabilità del codice questo modo di procedere comporta:
  - Maggiore possibilità di errore (si dovrebbero sempre usare le funzioni definite per accedere alla rappresentazione dei dati, evitando di farlo direttamente)
  - Difficoltà di modifica del codice: per modificare un'operazione sui dati occorre cercare nel programma tutti i punti in cui si accede alla rappresentazione

- Se dati e operazioni sono visti come un' entità congiunta, queste difficoltà sono risolte.

Programmazione a Oggetti

Vogliamo quindi che dati e operazioni sui dati siano fusi in una unica entità. Il C++ consente all'interno delle strutture la presenza, oltre che di membri dato (o attributi) , anche di campi funzione (metodi):

```
struct Complex {
double Re;
double Im;
// campi di tipo funzione;
void Print() { cout << Re << " + i" <<
Im<<endl; }
double Abs(){ return sqrt(Re*Re + Im*Im); }
};
int main() {
Complex C; C.Re = 0.5; C.Im = 2.3;
C.Print(); cout << C.Abs() << endl;
return 0;
}
```

- Le operazioni sui dati non sono procedure esterne, ma messaggi inviati agli oggetti (istanze del nuovo tipo di dato)
- Operazione sull'oggetto invocabile tramite l'oggetto (che non è più un parametro d'ingresso)
- Tuttavia si ha ancora accesso diretto alla rappresentazione interna dell'oggetto....

Programmazione a Oggetti

•Le strutture nella loro declinazione naturale violano una proprietà della OOP, l'incapsulamento, che prevede che gli attributi di un oggetto non devono essere accessibili se non attraverso i metodi dell'oggetto stesso.

•Problema risolto introducendo una nuova sintassi, che differisce da quella di struttura per i meccanismi di protezione. Definizione di Complex con la sintassi di classe:

```
class Complex {
public: // member function pubbliche;
void Print() { cout << Re << " + i" << Im << endl; }
double Abs(){ return sqrt(Re*Re + Im*Im); }
private: // data member, privati
double Re;
double Im;
};
```

•Keyword public e private specificano l'accessibilità

•Mutuamente esclusive

Programmazione a Oggetti

- public: le dichiarazioni che seguono questa keyword sono accessibili sia alla classe che a tutto il resto del programma, sempre invocabili

- private: quanto segue questa keyword (fino a un'eventuale keyword public successiva) è accessibile solo all'interno della classe stessa

- Il default è sempre private

```
class Complex {
public: // member function pubbliche;
void Print() { cout << Re << " + i" << Im<<endl; }
double Abs(){ return sqrt(Re*Re + Im*Im); }
private: // data member, privati
double Re;
double Im;
};
```

```
Complex A;
Complex * C;
A.Re = 10.2; // Errore!
C -> Im = 0.5; // Ancora errore!
 A.Print(); // Ok!
C -> Print() // Ok!
```

La rappresentazione interna dell'oggetto non è più direttamente accessibile

Programmazione a Oggetti

- In realtà, in C++ è possibile applicare le keyword public e private anche nel contesto di una struct:

- Il default per le struct (in assenza di una specifica di tipo private, cioè) è sempre public.

```
struct Complex {
public: // member function pubbliche;
void Print() { cout << Re << " + i" << Im<<endl; }
double Abs(){ return sqrt(Re*Re + Im*Im); }
private: // data member, privati
double Re;
double Im;
};
```

```
Complex A;
Complex * C;
A.Re = 10.2; // Errore!
C -> Im = 0.5; // Ancora errore!
 A.Print(); // Ok!
C -> Print() // Ok!
```

- Esiste anche un terzo tipo di keyword , protected, usata in caso di ereditarietà in relazione all'accesso, da parte di classi derivate, dei membri specificati come tali

Programmazione a Oggetti

•Quindi nella definizione di una classe compaiono una serie di membri dato e membri funzione, e una serie di keyword che specificano l'accessibilità dei membri:

```
class  class_name {
public:
<lista di dichiarazione per attributi e
member function pubbliche>
protected:
<lista di dichiarazione per attributi e
member function protette>
private:
<lista di dichiarazione per attributi e
member function private>
};
```

- Possono essere presenti:
  - dichiarazioni di variabili o costanti
  - funzioni
  - dichiarazioni di tipi nuovi (enum, union, struct e anche class, posso dichiarare una class all'interno di un'altra),

- Ordine delle keyword ininfluente, ma per la leggibilità del codice, si tende a specificare prima ciò che è pubblico (interfaccia) e dopo ciò che è privato

Classi e Attributi

Buona norma è che gli attributi siano se possibile tutti o private o eventualmente protected, e implementare delle member function pubbliche per consentire un accesso indiretto all'attributo dall'esterno(ad es. *setters/getters*)

```
class Complex {
public: // member function pubbliche;
/*....*/
double GetIm() const { return Im; }
void SetIm(double b){ Im=b; }

private: // data member, privati
double Re; double Im;
};
```

- tuttavia si deve evitare di dichiarare private (o protected) cio` che deve essere visibile e istanziabile anche all'esterno della classe, in particolare eventuali nuove definizioni dei tipi di parametri e dei valori di ritorno dei metodi public.

Classi e Attributi

- Ogni variabile automatica dichiarata nei metodi localmente ha prevalenza sull'attributo. Evitare di usare negli argomenti dei metodi nomi uguali a quelli utilizzati per gli attributi (se non volete ogni volta usare il risolutore di ambito)

- Nella dichiarazione degli attributi, rispettare le dipendenze

- I metodi della classe riconoscono comunque il simbolo anche se , all'interno della dichiarazione di classe, è dichiarato successivamente (il corpo di un metodo è esaminato dopo aver "letto" tutta la definizione di classe) . Questo non è vero se il simbolo (nuovo tipo, ad esempio) definisce il tipo del valore di ritorno del metodo o dei suoi argomenti, come può succedere se all'interno della classe è definita una classe annidata.

Classi Annidate

- In C++, le classi possono essere definite in un qualsiasi scope,: in particolare, una classe può anche essere definita all'interno di un'altra classe.

- Le classi definite all'interno delle altre classi sono dette: classi-membro o classi *annidate*.

- Sono tipicamente definite nella sezione privata della classe contenitore. Esprimono la necessità di definire un nuovo tipo che sia di supporto alla classe contenitrice ma non di uso ad altre.

Classi Annidate

- Se sono collocate nella sezione privata della classe di appartenenza, possono essere istanziate solo dai metodi di detta classe.

- L'accesso ai membri delle classe annidata, invece, non dipende dalla collocazione nella classe di appartenenza (public o private), ma solo da come sono dichiarati gli stessi membri al suo interno (se pubblici o privati).

- Se la classe annidata è dichiarata nella sezione pubblica della classe che la contiene, allora possono essere istanziate anche esternamente (ma occorre qualificarle con il risolutore di ambito)

- Esempio di uso di classi annidate

Classi e Metodi, protezione

Con particolare riferimento ai metodi:

- **Metodi pubblici:** liberamente invocabili dall'esterno del programma, solo attraverso una istanza della classe (o –caso particolare- direttamente, se si tratta di un metodo statico). La loro funzione è di presentare all'utente una interfaccia di servizi disponibili, ed è tutto quello che l'utente deve "sapere" della classe

- **Metodi privati**: accessibili solo all'interno degli altri metodi della classe. Servono per facilitare l'implementazione dei metodi pubblici e/o a supporto della logica interna della classe. Non possono essere invocati dall'esterno. In generale, nella logica OOP il funzionamento interno della classe non deve essere esposto all'utente, e deve poter essere cambiato in maniera totalmente trasparente.

- **Metodi protetti**: sono dei metodi che possono essere considerati pubblici per le classi derivate dalla classe in questione. Non sono tuttavia accessibili al resto del programma.

Classi e Incapsulamento

Riassumendo, con la sintassi di classe si ha l'opportunità di fare in modo che:

- Tutti i dettagli relativi all'implementazione di un oggetto siano nascosti al mondo esterno, e che l'oggetto interagisca solo attraverso i messaggi che manda e riceve (i metodi).

- Questa gerarchia di protezione dell'informazione (su dati e metodi) è alla base di quello che si chiama incapsulamento *(information hiding),* che è alla base del paradigma OOP e che consente di avere una struttura del codice più robusta e con minimi effetti collaterali per eventuali modifiche.

- Se si assume la gerarchia di protezione di default, e' anche la principale differenza concettuale fra una class e una struct C++, che sono altrimenti assolutamente analoghe.

Classi e Oggetti

- Una classe è un nuovo tipo di dato che incorpora dati (attributi), che normalmente non sono direttamente accessibili al resto del programma e metodi (member functions), anch'essi non necessariamente tutti accessibili al resto del programma.La classe definisce il prototipo dell'oggetto.

- Definire un oggetto della classe data crea una istanza della classe, che ha come conseguenza l'allocazione della memoria riservata alla sua rappresentazione e l'associazione con le funzioni che fanno parte dei metodi della classe . Fra classi e oggetti c'è la stessa relazione che esiste fra un tipo e una variabile.

- Un oggetto ha in ogni istante del programma uno stato definito (definito dai suoi attributi), che può cambiare in seguito all'invocazione di un suo metodo che fa parte della sua interfaccia pubblica

Classi-definizione delle funzioni membro

- La definizione dei metodi di una classe puo` essere eseguita o dentro la definizione di classe, omettendo la dichiarazione di funzione (e in questo caso si trasmette implicitamente una richiesta di rendere inline la funzione)

```
class Complex {
 public:
 /* ... */
void Print() { cout << Re << " + i" << Im;}
  private:
 /* ... */
};
```

- Oppure si può fare esternamente (ma si deve anche mettere la dichiarazione dentro il corpo della classe) utilizzando l'operatore di risoluzione di ambito (e in questo caso se lo si desidera si deve specificare inline esplicitamente):

```
(inline) void Complex::Print() {cout << Re << " + i" << Im; }
```

Classi-definizione delle funzioni membro

- All'interno dei metodi gli attributi dell'oggetto corrente possono essere utilizzati direttamente (non attraverso l'operatore ., per intenderci). I nomi dei membri sono tutti nello scope dell'oggetto in questione e l'informazione relativa a quale istanza stiamo considerando è gestita automaticamente dal compilatore attraverso il puntatore nascosto (*this*).
- Altre istanze della classe possono essere a loro volta argomenti dei metodi della classe. In questo caso l'uso dell'operatore di accesso ai membri "." è necessario.

```
class MyClass {
public:
void Func(MyClass & C);
private:
void DoSomething(){cout << "done!"<<endl;}
};
void MyClass::Func(MyClass & C) {
 DoSomething(); // messaggio all'oggetto corrente
C.DoSomething(); // messaggio al parametro }
```

Nel main:

```
MyClass A,B;
A.Func(B); //OK
A.DoSomething(); // Errore!
```

Il puntatore “nascosto” *this*

- Il compilatore trasforma il codice sorgente introducendo un puntatore costante "*nascosto*" (identificato dalla *parola-chiave* this) ogni volta che incontra la chiamata di una funzione-membro, e inserendo lo stesso puntatore come primo argomento della funzione

  void foo() → void foo(myclass* const this)

  this è un esempio di puntatore costante, che non si può riassegnare, ma solo utilizzare in lettura. Non posso “spostare” il puntatore all’oggetto corrente all’interno del codice della classe.

- Se poi la funzione è un metodo in sola lettura (metodo const) , il compilatore trasforma la definizione del metodo in questo modo:

  void foo() → void foo(const myclass* const this)

- this diventa un puntatore costante a costante. Questo fa sì che si possano definire due metodi identici, l'uno const e l'altro no, perchè in realtà i tipi del primo argomento sono diversi (e quindi l'overload è ammissibile).