PROGRAMMAZIONE ORIENTATA AGLI OGGETTI
in C++

Classi ed oggetti. Classi derivate, ereditarietà e polimorfismo. Template

Capitoli 12, 13, 14

Luis Joyannes Aguilar. Fondamenti di Programmazione in C++.
Algoritmi, strutture dati ed oggetti. McGraw-Hill

N.B.: il materiale fornito non è sostitutivo del testo. Per una preparazione completa si rimanda al testo suggerito

Classi ed oggetti

- Il paradigma computazionale object oriented (OOP) nacque nel 1969 ad opera del norvegese Nygaard
- simulazione del movimento delle navi nei fiordi
- difficile simulare maree, movimenti delle navi, forme delle linee di costa con i tradizionali metodi di programmazione
- Più facile concepire gli elementi da modellare come *oggetti* dotati di dati e funzioni proprie

Classi ed oggetti

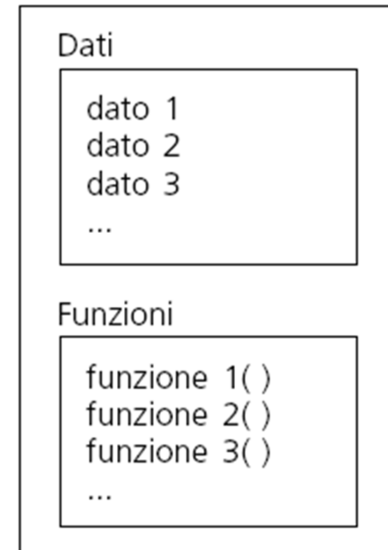
- OOP definisce un oggetto come qualsiasi cosa, reale o astratta, nella quale si possono immagazzinare *dati* ed *operazioni* che li manipolano
- Definiremo un oggetto come
  - una collezione di dati, detto suo stato, e
  - procedure in grado di alterare lo stato
  - Es.: calciatore FIFA2009 (nome, squadra, ruolo, posizione_in_campo, *insegui_avversario*, *colpisci_avversario*, *cambia_ruolo*);
  - Es.: ADT lista con i relativi operatori
- Gli oggetti comunicano fra loro tramite *messaggi*
  - *un messaggio è la finestra con la quale un oggetto interagisce con il mondo esterno*
- La comunicazione avviene tramite l'interfaccia di un oggetto
  - operazioni che costituiscono una *vista* dell'oggetto
  - non si sa come l'oggetto sia fatto internamente, ma lo si utilizza tramite le operazioni disponibili

Classi ed oggetti

- Una classe è un insieme di oggetti che condividono struttura e comportamenti
  - oggetti omogenei: sono strutturati e si comportano nello stesso modo
  - classe dei calciatori di FIFA2009
- Una classe contiene la specifica dei dati che descrivono l'oggetto che ne fa parte, insieme alla descrizione delle azioni che l'oggetto stesso è capace di eseguire
- In C++ questi dati si denominano *attributi* o *variabili*, mentre le azioni si dicono *funzioni membro* o *metodi*

Le classi

- Le classi definiscono tipi di dato personalizzati in funzione dei problemi da risolvere, facilitando la scrittura e la comprensione delle applicazioni; esse possono separare l'interfaccia dall'implementazione; solo il programmatore della classe conoscerà i dettagli implementativi, l'utilizzatore deve soltanto conoscere l'interfaccia
- Le classi sono esempi di Abstract Data Type
  - la classe è il mezzo naturale per tradurre l'astrazione di un tipo definito dall'utente che combina la rappresentazione dei dati (attributi) con le funzioni (metodi) che manipolano i dati
- La collocazione di dati e funzioni in una sola entità, la classe, è l'idea centrale dell' OOP



Definizione di una classe

- ha due parti:
  - *dichiarazione*: descrive i dati e l'interfaccia (cioè le "funzioni membro", anche dette "metodi")
  - *definizioni dei metodi*: descrive l'implementazione delle funzioni membro

```
class NomeClasse          // Identificatore valido
{
  dichiarazioni dei dati         // attributi
  definizione delle funzioni     // metodi
};
```

- gli attributi sono variabili semplici (interi, strutture, array, float, ecc.) o complessi (oggetti istanze di altre classi)
- i metodi sono funzioni semplici che operano sugli attributi (*dati*)

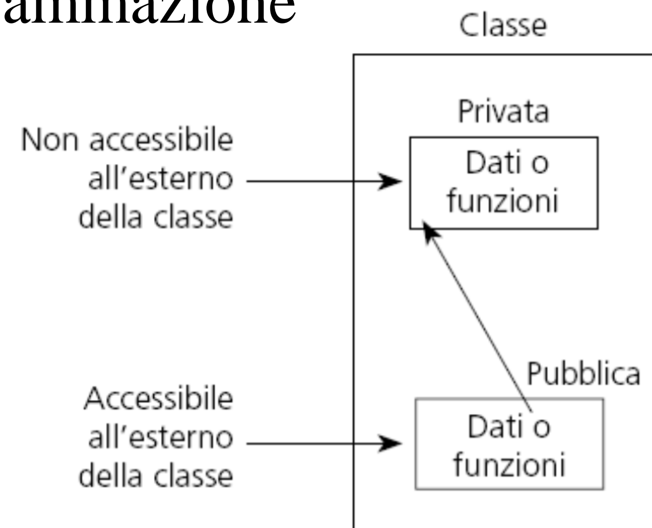
Specificatori di accesso

- per default, i membri di una classe sono nascosti all'esterno, cioè, i suoi dati ed i suoi metodi sono *privati*
- è possibile controllare la *visibilità* esterna mediante specificatori d'accesso:
  - la sezione `public` contiene membri a cui si può accedere dall'esterno della classe
  - la sezione `private` contiene membri ai quali si può accedere solo dall'interno della classe
  - ai membri che seguono lo specificatore `protected` si può accedere anche da metodi di classi *derivate* della stessa

```
class NomeClasse
{
   public:
      Sezione pubblica    // dichiarazione di membri pubblici
   protected:
      Sezione protetta    // dichiarazione di membri protetti
   private:
      Sezione privata     // dichiarazione di membri privati
};
```

Information hiding / incapsulamento

- questa caratteristica della classe si chiama *occultamento di dati (information hiding)* ed è una proprietà dell'OOP
- si tratta di una tecnica, spesso denominata *incapsulamento*, che limita molto gli errori rispetto alla programmazione strutturata




```
class Semaforo
{
   public:
      void cambiareColore();
      //...
   private:
      enum Colore {VERDE, ROSSO, GIALLO};
      Colore c;
};
```

Regole pratiche

- le dichiarazioni dei metodi (cioè le intestazioni delle funzioni), normalmente, si collocano nella sezione pubblica e le dichiarazioni dei dati (attributi), normalmente, si mettono nella sezione privata

- è indifferente collocare prima la sezione pubblica o quella privata; è però consigliabile collocare la sezione pubblica prima per mettere in evidenza le operazioni che fanno parte dell'interfaccia utente pubblica

- le parole chiavi `public` e `private` seguite da due punti, segnalano l'inizio delle rispettive sezioni pubbliche e private; benché non sia comune, una classe può avere varie sezioni pubbliche e private

Oggetti

- definita una classe, possono essere generate *istanze* della classe, cioè *oggetti*

```
nome_classe   identificatore ;
```

```
Punto P;
Semaforo S;
```

- un oggetto sta alla sua classe come una variabile al suo tipo
- quello che nelle `struct` era l'operatore di accesso al campo (`.`), qui diventa l'operatore *di accesso* al membro

```
Punto p;
p.Fissarex (100);
cout << " coordinata x è " << p.Leggerex();
```

Dati membro

- possono essere di qualunque tipo valido, con eccezione del tipo della classe che si sta definendo
- non è permesso inizializzare un membro dato di una classe all'atto della sua definizione; la seguente definizione di classe genera quindi errori:

```
class C {
private:
    int T = 0;                      // Errore
    const int CInt = 25;            // Errore
    int& Dint = T                   // Errore
// ...
};
```

non avrebbe senso inizializzare un membro dato dentro la definizione della classe, perché essa indica semplicemente il *tipo* di ogni membro dato e non riserva realmente memoria (sarebbe come voler inizializzare un campo di una struttura); si deve invece inizializzare i membri dato ogni volta che si crea un'*istanza specifica* della classe mediante il *costruttore* della classe

Funzioni membro

- i metodi possono essere sia dichiarati che definiti all'interno delle classi; la definizione di un metodo consiste di quattro parti:
  - ✓ il tipo restituito dalla funzione
  - ✓ il nome della funzione
  - ✓ la lista dei parametri formali (eventualmente vuota) separati da virgole
  - ✓ il corpo della funzione racchiuso tra parentesi graffe

le tre prime parti formano il prototipo della funzione che *deve essere definito* dentro la classe, mentre il corpo della funzione può essere definito altrove

```
class Articolo_Vendite {
public:
 double prezzo_medio(); // dichiarazione prototipo (definito altrove)
 bool articolo_uguale (const Articolo_Vendite & art) // definizione
   {return iva == art.iva; }                         // funzione
private:   // membri privati
   // ...
};
```

Chiamate a funzioni membro

- i metodi di una classe s'invocano così come si accede ai dati di un oggetto, tramite l'operatore punto (.) con la seguente sintassi:

nomeOggetto.nomeFunzione (valori dei parametri)

```
class Demo
{
private:
   // ...
public:
   void funz1 (int P1)
      {...}
   void funz2 (int P2)
      {...}
};
Demo d1, d2;       // definizione degli oggetti d1 e d2
...
d1.funz1(2005);
d2.funz1(2010);
```

Funzioni *inline* e *offline*

- i metodi definiti nella classe sono funzioni in linea*;* per funzioni grandi è preferibile codificare nella classe solo il prototipo della funzione
- nella definizione *fuori linea* della funzione bisogna premettere il nome della classe e l'*operatore di risoluzione di visibilità* **::**;

```
class Punto {
public:
   void FissareX(int valx);
private:
   int x;
   int y;
};
...
void Punto::FissareX(int valx)
{
   // ...
}
```

Header file ed intestazioni di classi

- il codice sorgente di una classe si colloca normalmente in un file indipendente con lo stesso nome della classe ed estensione `.cpp`
- le dichiarazioni si collocano normalmente in header file indipendenti da quelli che contengono le implementazioni dei metodi

- Es.: la dichiarazione di una classe `mia_classe` sarà inserita in in un header file `mia_classe.h`, mentre l'implementazione sarà definita in `mia_classe.cpp`
- Un file che crea oggetti di una classe dichiarata in `mia_classe.h` deve utilizzare la direttiva `#include <miaclasse>`

Costruttori

- a volte può essere conveniente che un oggetto si possa autoinizializzare all'atto della sua creazione, senza dover effettuare una successiva chiamata ad una sua qualche funzione membro
- un *costruttore* è appunto un metodo di una classe che viene automaticamente eseguito all'atto della creazione di un oggetto di quella classe
- ha lo stesso nome della propria classe e può avere qualunque numero di parametri ma non restituisce alcun valore (neanche `void`)

```
class Rettangolo
{
   private:
      int Sinistro;
      int Superiore;
      int Destro;
      int Inferiore;
   public:
      Rettangolo(int Sin, int Sup, int Des, int Inf); // Costruttore
      // definizioni di altre funzioni membro
};
```

Definizione oggetto con costruttore

- quando si definisce un oggetto, si passano i valori dei parametri al costruttore utilizzando la sintassi di una normale chiamata di funzione:

```
Rettangolo rect(25, 75,25,75); // rect è ISTANZA di Rettangolo

Rettangolo* nr = new Rettangolo(25, 75,25,75); // nr punta
                                      // una nuova ISTANZA di Rettangolo
```

- un costruttore che non ha parametri si chiama *costruttore di default*; normalmente inizializza i membri dato assegnandogli valori di default
- C++ crea automaticamente un costruttore di default quando non vi sono altri costruttori, tuttavia esso non inizializza i membri dato della classe a valori predefiniti
- un *costruttore di copia* è creato automaticamente dal compilatore quando si passa un oggetto per valore ad una funzione (si costruisce una copia locale dell'oggetto) o quando si definisce un oggetto inizializzandolo ad un altro oggetto dello stesso tipo

Definizione oggetto con costruttore

```
class Punto2D
{
public:
   Punto2D();
   Punto2D(int coord1, int coord2);
private:
int x;
int y;

};

Punto2D P; // chiama il costruttore di default
Punto2D Origine(0,0); // chiama il costruttore alternativo
```

Distruttore

- si può definire anche una funzione membro speciale nota come *distruttore*, che viene chiamata automaticamente quando si distrugge un oggetto
- il distruttore ha lo stesso nome della sua classe preceduto dal carattere ~
- neanche il distruttore ha tipo di ritorno ma, al contrario del costruttore, non accetta parametri e *non* ve ne può essere più d'uno

```
class Demo
{
private:
   int dati;
public:
   Demo()  {dati = 0;}          // costruttore
   ~Demo() {}                   // distruttore
};
```

- serve normalmente per liberare la memoria assegnata dal costruttore
- se non si dichiara esplicitamente un distruttore, C++ ne crea automaticamente uno vuoto

Sovraccaricamento di metodi

- anche le funzioni membro possono essere sovraccaricate, ma soltanto nella loro propria classe, con le stesse regole utilizzate per sovraccaricare funzioni ordinarie
- due funzioni membro sovraccaricate non possono avere lo stesso numero e tipo di parametri
- l'*overloading* permette di utilizzare uno stesso nome per più metodi che si distingueranno solo per i parametri passati all'atto della chiamata

```
class Prodotto
{
public:
   int prodotto (int m, int n);                    // metodo 1
   int prodotto (int m, int p, int q);             // metodo 2
   int prodotto (float m, float n);                // metodo 3
   int prodotto (float m, float n, float p);       // metodo 4
}
```

ESERCITAZIONE GUIDATA #1

- Creazione di una classe Persona

CLASSI DERIVATE

Classi derivate

- Una *classe derivata* eredita attributi e metodi dalla *classe base* già esistente
- Un oggetto della classe derivata è un oggetto della classe base (Es.: bicicletta IS-A veicolo
- la dichiarazione di una classe derivata deve includere il nome della classe base da cui deriva ed, eventualmente, uno specificatore d'accesso indicante il tipo di ereditarietà (`public`, `private` o `protected`) secondo la seguente sintassi:

```
class ClasseDerivata : specific_accesso_opz ClasseBase {
   membri;
};
```

Ereditarietà pubblica: specificatore di accesso `public`; significa che i membri pubblici della classe base sono tali anche per quella derivata

Ereditarietà privata: specificatore di accesso `private`

Ereditarietà protetta: specificatore di accesso `protected`

Se si omette lo specificatore di accesso si assumerà per default `private`

esempio di classi derivate

```
class Pubblicazione {
  public:
     void InserireEditore(const char *s);
     void InserireData(unsigned long dat);
  private:
     string editor;
     unsigned long data;
};

class Rivista : public Pubblicazione {
  public:
     void InserireTiratura(unsigned n);
     void InserireVenduto(unsigned long n);
  private:
     unsigned tiratura;
     unsigned long venduto;
};

class Libro : public Pubblicazione {
  public:
     void InserireISBN(const char *s);
     void InserireAutore (const char *s);
  private:
     string ISBN;
     string autore;
};
```

Tipi di ereditarietà

- in una classe, gli elementi pubblici sono accessibili a tutte le funzioni, quelli privati sono accessibili soltanto ai membri della stessa classe e quelli protetti possono essere acceduti anche da classi derivate (*proprietà dell'ereditarietà*)
- vi sono tre tipi di ereditarietà: *pubblica, privata* e *protetta*, la più utilizzata delle quali è la prima
- una classe derivata non può accedere a variabili e funzioni private della sua classe base
- per occultare dettagli una classe base utilizza normalmente elementi protetti invece che elementi privati
- supponendo ereditarietà pubblica, gli elementi protetti sono accessibili alle funzioni membro di tutte le classi derivate
- per default, l'ereditarietà è privata; se accidentalmente si dimentica la parola riservata `public`, gli elementi della classe base saranno inaccesibili

Ereditarietà pubblica

- *ereditarietà pubblica* significa che una classe derivata ha accesso agli elementi pubblici e protetti della sua classe base
- gli elementi pubblici si ereditano come elementi pubblici; gli elementi protetti come protetti
- si rappresenta con lo specificatore `public` nella derivazione di classi

Esempio ereditarietà

`eredita.cpp`

`ereditaerrore.cpp`

`ereditaerrorecorretto.cpp`

Ereditarietà privata

- con l'ereditarietà privata un utente (dell'interfaccia) della classe derivata non ha accesso ad alcun elemento della classe base:

```
class ClasseDerivata : private ClasseBase {
public:
    // sezione pubblica
protected:
    // sezione protetta
private:
    // sezione privata
};
```

- i membri pubblici e protetti della classe base diventano membri privati della classe derivata
- l'ereditarietà *privata* è quella di *default*; essa occulta la classe base all'utente perché sia possibile cambiare l'implementazione della classe base o eliminarla del tutto senza richiedere alcuna modifica all'utente dell'interfaccia

Ereditarietà protetta

- con l'ereditarietà protetta, i membri pubblici e protetti della classe base diventano membri protetti della classe derivata ed i membri privati della classe base diventano inaccessibili

Tipo di ereditarietà	Accesso a membro classe base	Accesso a membro classe derivata

`public`	`public`
	`protected`
	`private`
		`public`
		`protected`
		inaccessibile

`protected`	`public`
	`protected`
	`private`
		`protected`
		`protected`
		inaccessibile

`private`	`public`
	`protected`
	`private`
		`private`
		`private`
		inaccessibile

Esempio

```
class Base{
public:
  int i1;
protected:
  int i2;
private:
  int i3;
};
```

```
class D1: private Base {
void f();
};
…
void D1::f() {
 i1=0; // corretto
 i2=0  // corretto
 i3=0  // ERRATO
};
```

```
class D2: protected Base {
void g();
};
```

```
class D3: public Base {
void h();
};
```

- Le tre classi accedono ai membri i1 e i2
- i3 è inaccessibile a tutte

… e dall’esterno?

```
void main()
{
    Base b;
    b.i1=0; // ok
    b.i2=0; // ERRORE
    b.i3=0; // ERRORE
    D1 d1;
    d1.i1=0; // ERRORE
    d1.i2=0; // ERRORE
    d1.i3=0; // ERRORE
    D2 d2;
    d2.i1=0; // ERRORE
    d2.i2=0; // ERRORE
    d2.i3=0; // ERRORE
    D3 d3;
    d1.i1=0; // ok
    d1.i2=0; // ERRORE
    d1.i3=0; // ERRORE
}
```

Costruttore di una classe derivata

- la sintassi di un costruttore di una classe derivata è:

```
ClasseDer::ClasseDer(paramD):ClasseBase(paramB), Inizial {
// corpo del costruttore della classe derivata};
```

Inizial è l'inizializzazione di membri dato della classe

```
class Data {
public:
   Data(int, int, int);
private:
   int giorno, mese, anno;
};
```

```
class Persona {
public:
  Persona(string, string, int g, int m, int a);
private:
  string nome;
  string cognome;
  Data data_nascita;
};
```

```
class Impiegato : public Persona {
public:
   Impiegato(string, string, string, int, int, int, int, int, int);
```

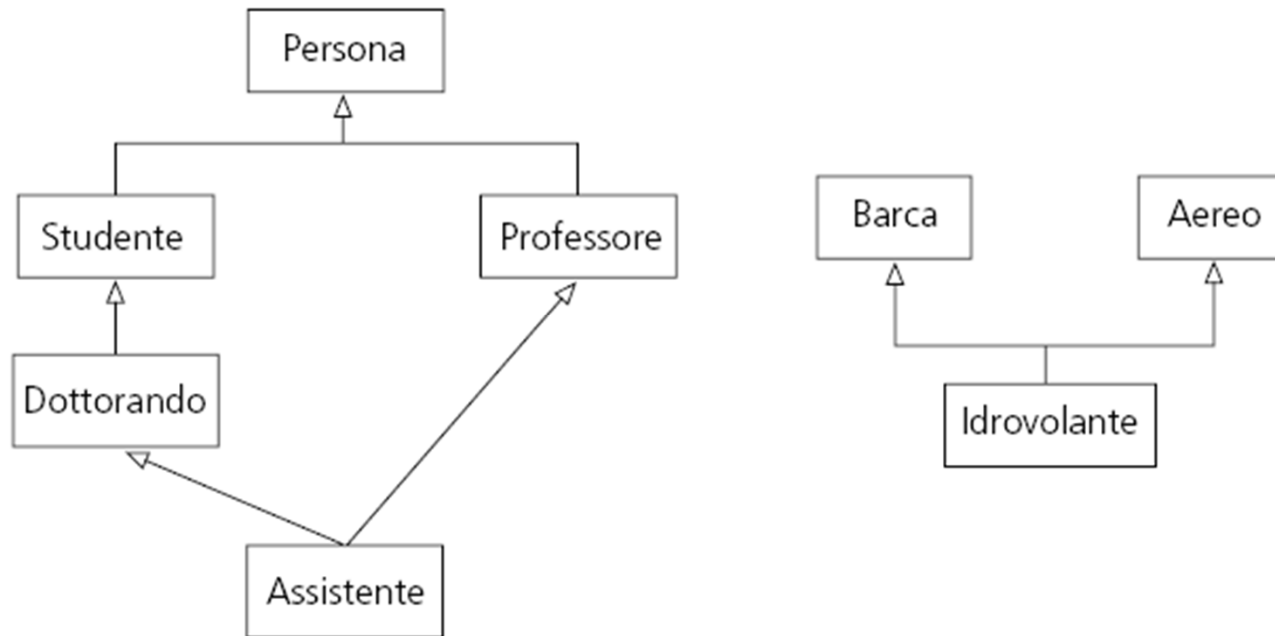
```
// costruttore per la classe Impiegato

Impiegato::Impiegato(string n, string c, string mat, int gg, int mm, int
aa, int giornoa, int mesea, int annoa):Persona(n,c,gg,mm,aa),
 data_assunzione(giornoa,mesea,annoa)
```

Distruttori ed ereditarietà

- i distruttori non si ereditano, ma se necessario si genera un distruttore di default
- si utilizzano normalmente solo quando un corrispondente costruttore ha assegnato spazio in memoria che deve essere liberato

Ereditarietà multipla

- una classe può ereditare attributi e comportamento di più di una classe base



Ereditarietà multipla

- la sintassi è:

```
class Derivata :   [virtual][tipo_accesso] Base1,
                   [virtual][tipo_accesso] Base2,
                   [virtual][tipo_accesso] Basen {
public:
   // sezione pubblica
private:
   // sezione privata
...
};
```

derivata: nome della classe derivata
tipo_accesso: `public`, `private` o `protected`
Base1, *Base2,...*: classi base con nomi differenti
virtual..: è opzionale e specifica una classe base compatibile.
Funzioni o dati membro che abbiano lo stesso nome in *Base1*, *Base2, Basen*, costituiranno motivo di ambiguità

Esempio

```
class Studente {
private:
long matricola;
…
public:
long getMatricola() const;
…
};
class Lavoratore {
private:
   long stipendio;
public:
long getStipendio() const;
…
};
class studente_lavoratore: public studente, public lavoratore
}
```

ESERCITAZIONE GUIDATA #2

- Ereditarietà: Creazione di una classe Studente derivata da Persona

ESERCITAZIONE #3

- Arricchimento della classe Studente con nuove funzioni per la gestione degli esami

Binding dinamico

- per *binding* s'intende la connessione tra la chiamata di una funzione ed il codice che l'implementa
- esso può essere *statico*, se il collegamento avviene in fase di compilazione, *dinamico* se la connessione avviene durante l'esecuzione
- il binding dinamico fa sì che il codice da eseguire verrà determinato solo all'atto della chiamata; solo durante l'esecuzione del programma si determinerà il binding effettivo (tipicamente tramite il valore di un puntatore ad una classe base) tra le diverse possibilità (una per ogni classe derivata)
- il principale vantaggio del binding dinamico, rispetto a quello statico, è che offre un alto grado di flessibilità e praticità nella gestione delle gerarchie di classi
- tra gli svantaggi vi è il fatto che il binding dinamico è meno efficiente di quello statico
- in C++ il binding per default è quello statico e per specificare il binding dinamico si fa precedere la dichiarazione della funzione dalla parola riservata `virtual`

Funzioni virtuali

- `virtual` anteposto alla dichiarazione di una funzione indica al compilatore che essa può essere definita in una classe derivata e che, in questo caso, la funzione sarà invocata direttamente tramite un puntatore
- si deve qualificare un metodo di una classe con `virtual` solo quando esiste la possibilità che da quella classe se ne possano derivare altre
- le funzioni virtuali servono nella dichiarazione di classi astratte e nel polimorfismo

```
class figura {
public:
   virtual double calcolare_area(void) const;
   virtual void disegnare(void) const;
   // altre funzioni membro che definiscono
un'interfaccia a
   //tutti i tipi di figure geometriche
};
```

Funzioni virtuali

- Ogni classe derivata deve definire le sue proprie versioni delle funzioni dichiarate virtuali nella classe base; se le classi `cerchio` e `rettangolo` derivano dalla classe `figura`, debbono entrambe definire le funzioni membro `calcolare_area` e `disegnare`

```
class cerchio : public figura
{
   public:
      virtual double calcolare_area(void) const;
      virtual void disegnare(void) const;
      // ...
   private:
      double xc, yc;          // coordinata del centro
      double raggio;           // raggio del cerchio
};
#define PI 3.14159       // valore di "pi"
// Implementazione di calcolare_area
double cerchio::calcolare_area(void) const
{
   return(PI * raggio * raggio);
}
// Implementazione della funzione "disegnare"
void cerchio::disegnare(void) const
{
   // ...
}
```

Binding statico

Binding dinamico tramite funzioni virtuali

- Le funzioni vengono chiamate tramite un puntatore a `figura` del tipo:

```
figura* s[10];  // 10 puntatori ad oggetti figura
int i, numfigure = 10;
// crea figure e immagazzina puntatori nell'array s
// disegna le figure
   for (i = 0; i < numfigure; i++)
      s[i] -> disegnare();
```

- il compilatore C++ non può sapere l'implementazione specifica della funzione `disegnare()` che sarà chiamata a tempo d'esecuzione

Polimorfismo

- è la proprietà in base alla quale oggetti differenti possono rispondere in maniera diversa ad uno stesso messaggio

```
class figura {
   tipoenum tenum;          //tipoenum è un tipo enumerativo
public:
   virtual void Copiare();
   virtual void Disegnare();
   virtual double Area();
};
class cerchio : public figura {
   ...
public:
   void Copiare();
   void Disegnare();
   double Area();
};
class rettangolo : public figura {
   ...
public:
   void Copiare();// il polimorfismo permette ad oggetti differenti
                  // di avere metodi con lo stesso nome
   void Disegnare();
   double Area();
};
```

Polimorfismo

- si può passare lo stesso messaggio ad oggetti differenti:

```
switch(...) {
...
case Cerchio:
   MioCerchio.Disegnare();
   d = MioCerchio.Area();
   break;
case Rettangolo:
   MioRettangolo.Disegnare();
   d = MioRettangolo.Area();
   break;
...
};
```

NON E’ UNA BUONA PRATICA DI PROGRAMMAZIONE!
SE AGGIUNGESSIMO UNA NUOVA CLASSE DERIVATA DA FIGURA DOVREMMO MODIFICARE IL CASE

- oppure con il binding dinamico:

```
// crea e inizializza un array di figure
figura* figure[]= { new cerchio, new rettangolo, new triangolo};
...
figure[i].Disegnare();
```

Vantaggi del polimorfismo

- Permette quindi di utilizzare una stessa interfaccia (come i metodi `Disegnare` ed `Area`) per lavorare con oggetti di diverse classi

- Regole per sfruttare i vantaggi del polimorfismo:
  - creare una gerarchia di classi in cui le operazioni più importanti siano definite nei metodi *virtual* della classe base
  - implementare i metodi nelle classi derivate
  - riferimenti agli oggetti delle classi derivate tramite puntatori (binding dinamico)
- le sue applicazioni più frequenti sono:
  - specializzazione di classi derivate (quadrato è una specializzazione di rettangolo)
  - strutture di dati eterogenei: manipolazioni di oggetti simili realizzati con strutture dati diverse
  - gestione delle gerarchie di classi

Template

Genericità

- E’ una tecnica di programmazione che permette di definire una classe (o una funzione) senza specificare il tipo di dato di uno o più dei suoi membri (o parametri)
- Consente di sfruttare il fatto che gli algoritmi di risoluzione di numerosi problemi non dipendono dal tipo di dato da elaborare; ad esempio, un algoritmo che gestisca una pila di caratteri sarà essenzialmente lo stesso che gestisce una pila di interi o di qualunque altro tipo
- in vecchi linguaggi procedurali come Pascal e COBOL, è necessario sviluppare un programma diverso per ogni tipo di dato da inserire nella pila
- nei nuovi linguaggi OOP, come l'ANSI/ISO C++ ed il Java (come pure nel "vecchio" Ada), esistono i *template*
- essi permettono di definire *classi generiche* (o *parametriche*) che implementano strutture e classi indipendenti dal tipo di elemento da processare
- dovranno poi essere istanziati dall'utente per produrre sottoprogrammi o classi che lavorino con specifici tipi di dato

Template in C++

- costruzione per scrivere *funzioni* e *classi* molto generali che possono applicarsi a dati di tipo diverso

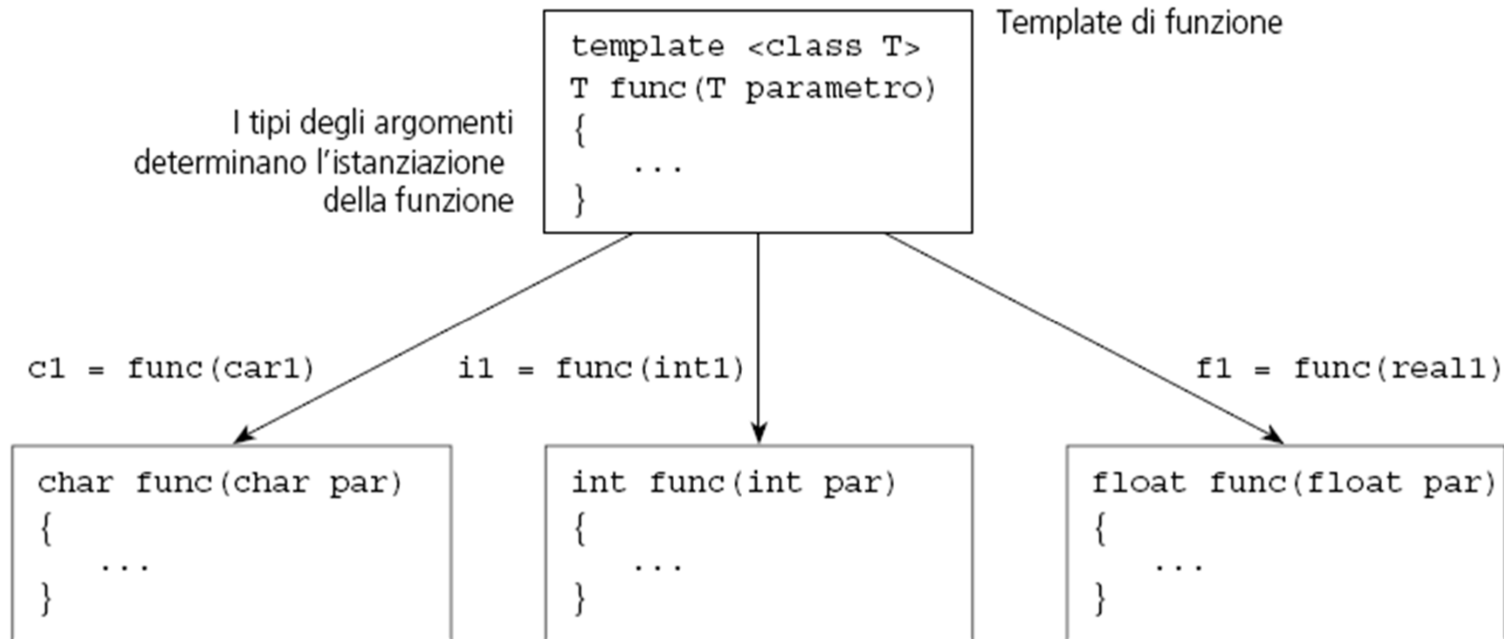
- questa generalità non implica perdita di rendimento e non obbliga a sacrificare i vantaggi del C++ in tema di controllo stretto dei tipi di dato
- **così come una classe è un modello per istanziare oggetti (della classe) a tempo d'esecuzione, un *template* è un modello per istanziare classi o funzioni (del template) a tempo di compilazione**
- **i template sono quindi funzioni e classi generiche, implementate per un tipo di dato da definirsi in seguito**
- per utilizzarli il programmatore deve solo specificare i tipi con i quali essi debbono lavorare

Template di funzioni

- Insieme indeterminato di funzioni *sovraccaricate* che descrive l'algoritmo *specifico* di una funzione *generica*

- L'algoritmo è *specifico*, ovvero è *determinato* (es.: scambio di due variabili)
- La funzione è *generica*, poiché indipendente dal tipo dei dati su cui lavora
- L'idea è quella di rappresentare il tipo di dato usato dalla funzione con un nome che indichi "qualunque tipo" (T)

Template di funzioni



- ANSI/ISO C++ scrive `typename` al posto di `class`
- possono avere più di un parametro di tipo:

```
template <typename T>
const T& max(const T& a, const T& b)
{ return a > b ? a : b;}
...
long x = max <long> (40, 50);
int x = max <int> (40, 50);
```

← istanza del template

Template di classi

- Permettono di definire classi parametriche che possono gestire differenti tipi di dato

```
template <typename T>
class tipopar {
    ...
};
```

dove `T` è il nome del tipo utilizzato dal template e `tipopar` (per esempio *Pila*) è il nome del tipo parametrizzato del template; `T` non è limitato a tipi di dato predefiniti
il codice viene sempre preceduto da un'istruzione nella quale si dichiara `T` come parametro di tipo, e possono esserci più parametri tipo

```
template <typename T>
struct Punto {
   T x, y;
};
...
Punto<int> pt = {45, 15};
```

Modelli di compilazione di templates

- Quando il compilatore vede una definizione di template, non genera codice immediatamente
- Esso produce istanze specifiche di tipi del template solo quando vede una chiamata al template (di funzione o di classe che sia)
- Per generare un'*istanziazione* il compilatore deve accedere al codice sorgente che definisce il template
- C++ standard definisce due modelli per la compilazione del codice dei templates: "compilazione per inclusione" (supportato da tutti i compilatori) e " compilazione separata" (supportato solo da alcuni)
- Per entrambi i modelli le definizioni delle classi e le dichiarazioni delle funzioni vanno in header files, mentre le definizioni di membri e metodi vanno in files sorgenti
- I due modelli differiscono però nel modo in cui si rendono disponibili al compilatore le definizioni dei files sorgenti

Compilazione per inclusione

- il compilatore deve vedere la definizione di qualunque template
- includere nell'header file non solo le dichiarazioni, ma anche le definizioni
- permette di mantenere la separazione tra header files e files d'implementazione, anche se s'inserisce una direttiva `#include` nel header file perché inserisca le definizioni del file `.ccp`

```
//header file demo.h
#ifndef DEMO_H
#define DEMO_H
template<class T> int confrontare(const T&, const T&);
//altre dichiarazioni
#include "demo.cpp"   //definizioni di confrontare
#endif
//implementazione del file demo.cc
template<class T> int confrontare(const T &a, const T &b)
{
   if(a < b) return -1;
   if(b < a) return 1;
   return 0;
}
//altre definizioni
```

compilazione separata

- permette di scrivere le dichiarazioni e funzioni in due files (estensioni `.h` e `.cpp`)
- si deve utilizzare la parola riservata `export` per ottenere la compilazione separata di definizioni di templates e dichiarazioni di funzioni di templates
- la dichiarazione del template di funzione si mette in un header file, ma la dichiarazione non deve specificare `export`

```
//definizione del template in un file compilato separatamente
export template<typename T>
T somma(T t1, T t2)
```

- l'uso di `export` in un template di classe è un po' più complicato

```
//intestazione del template di classe sta nel file
//di intestazione condiviso
template <class T> class Pila {...};
//File pila.ccp dichiara Pila come esportata
export template <class T> class Pila;
#include "Pila.h"
//definizioni di funzioni membro di Pila
```

Templates e polimorfismo

- una funzione è polimorfica se almeno uno dei suoi parametri può supportare tipi di dato differenti
- qualunque funzione che abbia un parametro come puntatore ad una classe può essere una funzione polimorfica e si può utilizzare con tipi di dato diversi
- una funzione è una funzione template solo se è preceduta da un'appropriata clausola template
- scrivere una funzione template implica pensare in astratto, evitando qualunque dipendenza da tipi di dato, costanti numeriche, ecc.
- una funzione template è solo un modello e non una vera funzione
- la clausola template è un generatore automatico di funzioni sovraccaricate
- le funzioni templates lavorano anche con tipi aritmetici
- le funzioni polimorfiche debbono utilizzare puntatori
- la genericità polimorfica si limita a gerarchie
- i templates tendono a generare un codice eseguibile grande, poiché duplicano le funzioni

ESERCITAZIONE #4

- Modifica della classe Studente per gestire gli esami mediante una LISTA CON TEMPLATE