EREDITARIETÀ E POLIMORFISMO

12

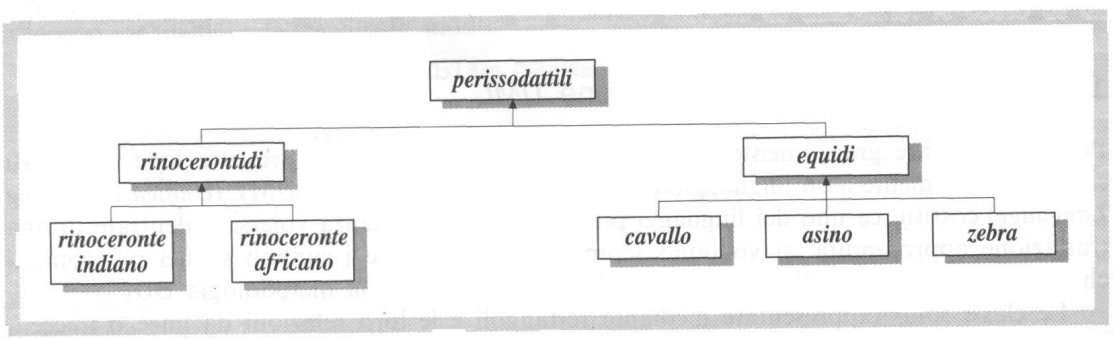
*La classe rappresenta uno dei concetti fondamentali del C++: è un tipo definito dall'utente, consente di nascondere, inizializzare e accedere ai propri dati solo mediante le proprie funzioni membro. Nella fase iniziale di progettazione di un programma, secondo la tecnica OOP, diventa prioritaria l'identificazione delle classi indispensabili per affrontare il problema. Tuttavia, non tutte le classi necessarie sono completamente indipendenti da altre, ma possono condividere alcuni particolari oppure, semplicemente, estenderne le funzionalità. Il C++ mette a disposizione dell'utente un meccanismo sintattico, l'**ereditarietà** (**inheritance**), che permette di creare nuove classi con caratteristiche che derivano da classi già definite.*

In questo modo si potranno creare degli oggetti imparentati fra loro da relazioni di gerarchia. Ognuno di questi oggetti sarà poi in grado di definire e svolgere in modo differente certe azioni comuni. Ciò costituisce un altro dei cardini fondamentali su cui si basa la programmazione Object Oriented: il ***polimorfismo*** (***polymorphism***).

12.1 Introduzione

La realtà che ci circonda è composta di entità animate e inanimate e solo mediante uno sforzo di astrazione è facile attribuire loro alcune qualità comuni, classificarle e sostituire alle infinite varietà esistenti un numero ridotto di categorie. A costo di un inevitabile impoverimento, si crea un ordine, non nella realtà, ma nella visione che ne ha l'uomo.

Questi sforzi di catalogazione e ordinamento sono importantissimi: qualunque riflessione o teoria scientifica sul «reale» avviene, di fatto, sul **modello della realtà** che la mente umana ha elaborato. Le classificazioni vengono di solito definite mediante criteri, differenti secondo le diverse entità da rappresentare, aventi come fine comune quello di produrre conoscenza. È noto, almeno a livello elementare, il sistema con cui gli zoologi catalogano gli animali o i botanici le piante. Facendo riferimento alla zoologia, si può avere, per esempio, la seguente classificazione:



.1
o di
azione.

In una prima fase, la classificazione dà inizio a una categorizzazione delle classi e dei suoi oggetti e determina gli attributi che consentono di identificarli come appartenenti a questa o a quella classe. La classificazione ha, in via di principio, la struttura di un albero genealogico, caratterizzato da un punto di partenza e da un numero variabile di nodi e ramificazioni in ordine gerarchico.

Attenzione, non si sta parlando di *C++*, ma semplicemente di teoria delle classificazioni: i concetti di classe, oggetto e gerarchia sono propri di un modo usuale dell'uomo di ordinare le proprie conoscenze. L'ordine gerarchico di una classificazione presuppone che una classe possa derivare da un'altra, di cui mantiene le caratteristiche fondamentali.

Quello descritto è uno dei meccanismi più potenti del pensiero umano che consente di acquisire conoscenza per **inferenza**, ovvero di reagire a situazioni mai incontrate, compiere azioni mai fatte prima o di apprendere il funzionamento di oggetti o strumenti mai visti, semplicemente perché simili ad altri ormai noti.

Un meccanismo concettuale analogo è usato nei linguaggi di programmazione orientati agli oggetti (*OOP*) per creare nuove classi da classi già definite. Tale meccanismo si definisce di **derivazione**, perché la nuova classe deriva da un'altra precedentemente definita: alla base del meccanismo di derivazione vi è il concetto fondamentale di **ereditarietà**.

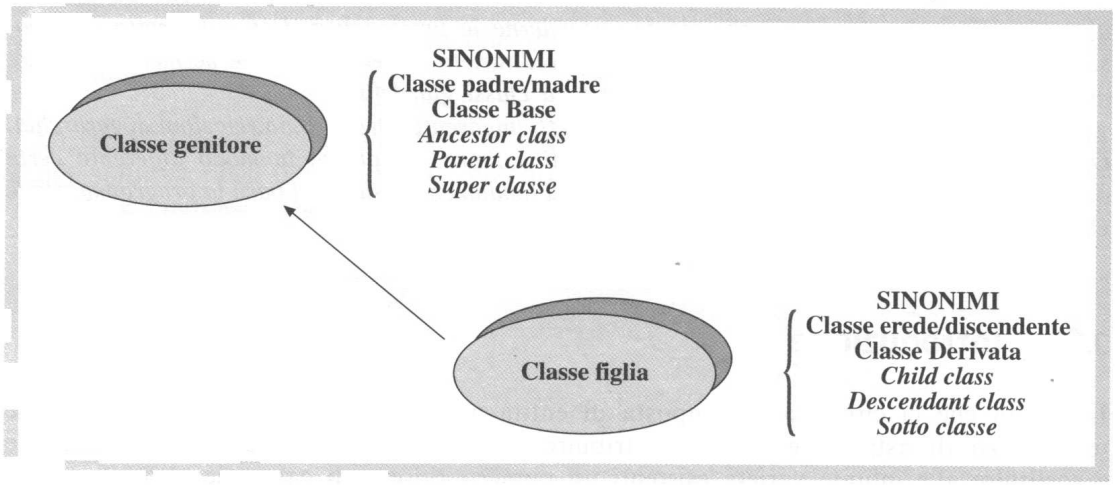


Fig. 12.2 Derivazione: la freccia indica la classe dalla quale si discende.

Una classe derivata (*child*) eredita tutte le proprietà della classe base (*parent*)

In questo modo, è facile intuire che la costruzione di classi complesse diventa sicuramente più semplice perché si risparmia buona parte del lavoro; è sufficiente, infatti, scegliere una classe genitore il più possibile simile alla classe figlia che si vuole realizzare per poi aggiungere, a quest'ultima, le nuove caratteristiche desiderate.

Un'altra ragione per usare l'ereditarietà è la possibilità di riutilizzare facilmente il codice di un progetto precedente o più semplicemente, quando è necessario, apportare delle modifiche a tale progetto, con un dispendio minimo di energie.

12.1.1 Il linguaggio di modellazione *UML*

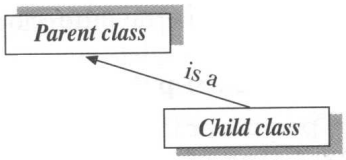
Per rappresentare graficamente su carta un modello dell'analisi e progettazione di oggetti si ricorre usualmente a un linguaggio di modellazione. Il linguaggio ***UML*** (***Unified Modelling Language***) costituisce uno dei linguaggi più noti e adottati. Senza volersi addentrare in una trattazione approfondita, si vogliono semplicemente indicare i simboli grafici più comuni che sono adottati per «disegnare» un programma, secondo la metodologia *OOP*.

Le classi sono rappresentate mediante rettangoli e le loro relazioni da linee o frecce.

Relazione di ereditarietà

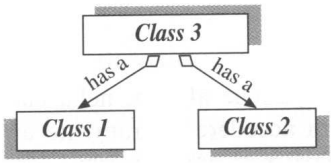
La direzione della freccia rappresenta il concetto di ereditarietà. La freccia punta dalla classe più specializzata alla classe più generale.

Si può affermare che la classe derivata è anche una classe base, in altre parole s'implementa una relazione del tipo «***is a***».



Relazione di contenimento

Facilmente un oggetto è composto di numerosi oggetti; ad esempio una motocicletta è costituita da due ruote, un motore, una sella, ecc. In questo caso un certo oggetto «ha» altri oggetti, e proprio in tal senso la relazione di contenimento implementa una relazione del tipo «***has a***». Il simbolo è un piccolo rombo unito a una freccia.



Relazione di associazione

Fra due oggetti può esistere semplicemente una relazione, in altre parole nessuno dei due contiene l'altro o eredita dall'altro le caratteristiche. Il simbolo è un segmento che unisce gli oggetti.

Ogni classe andrà poi dettagliata secondo quanto indicato nel capitolo precedente([1]).

12.2 Ereditarietà (*Inheritance*)

Si consideri una classe di nome `CBase` con un solo membro di tipo intero e, per semplicità, pubblico:

```
class CBase {
public:
  int numero;      //data member di tipo intero
};
```

Per derivare una classe dalla classe `CBase` è necessario impostare un meccanismo sintattico appropriato. La classe che eredita dalla classe `CBase`, che banalmente si chiamerà `CDerivata`, si dichiara nel seguente modo:

```
class CDerivata : public CBase {
public:
  CDerivata(int i); // costruttore della classe CDerivata
  int getNumero() { return numero; };
  int numero;       // membro ereditato
};
```

([1]) *Vedi* sottoparagrafo 11.2.4 *Classi: una rappresentazione grafica.*

È importante notare che `CDerivata` è una classe senza **nuovi** dati membri e che contiene due *member function*: il costruttore e il metodo `getNumero` il quale consente di accedere al valore del dato membro `numero`, **ereditato** dalla classe base.

La *child class* è stata dichiarata facendo seguire al suo nome, il simbolo « **:** », la *keyword* **public** e il nome della classe da cui eredita le proprietà. In questa situazione, `public` assume il significato di **modificatore di accesso** ([2]).

La dichiarazione di una *child class* conterrà tutto ciò che serve a distinguerla dalla classe di origine.

Il costruttore della classe `CDerivata` si può definire come segue:

```
// Costruttore con parametri per la classe CDerivata
CDerivata::CDerivata(int i) {
  numero = i;
}
```

Si può osservare che il costruttore assegna il valore di `i` a `numero` che, a tutti gli effetti, è considerato come un dato membro della classe derivata.

L'istanza di un oggetto di classe `CDerivata` dovrà avvenire specificando gli argomenti necessari al suo costruttore:

```
CDerivata d(5);
```

La semplicità dell'esempio è legata anche all'uso, nella classe di origine, dello specificatore d'accesso `public` che consente di accedere liberamente ai suoi membri. La maggior parte delle classi contiene la parola chiave **private** perché, come già ampiamente discusso, uno degli aspetti fondamentali di un oggetto è la sua capacità di rendere inaccessibili i propri dati dall'esterno (*information hiding*).

Una child class non può modificare i dati private della classe da cui eredita, né tanto meno accedervi.

Questa restrizione non dovrebbe sorprendere perché, se fosse vero l'opposto, chiunque, unicamente dichiarando una *child class*, potrebbe modificare i dati privati della classe di partenza.

Come risolvere il problema? Nello stesso e unico modo con il quale è possibile modificare i dati *private* di una classe: attraverso le sue *member function*. Analogamente, una *child class* può accedere ai dati privati della propria classe base utilizzando solo le *member function public* di quest'ultima.

Il programma seguente ripresenta l'esempio precedente con qualche variazione:

```
class CBase{
private:
  int numero;
public:
  CBase(int i) { numero = i; };
  void  setNumero(int i) { numero = i; };
  int   getNumero() { return numero; };
};
```

La classe `CBase` dispone ora di un costruttore e di due nuovi metodi per modificare (`setNumero`) e leggere (`getNumero`) il valore del *data member private* `numero`.

La classe `CDerivata`, che erediterà le proprietà di `CBase`, è la seguente:

```
class CDerivata : public CBase {
private:
  char lettera;
```

([2]) Il nome della classe base può essere preceduto anche dalle parole chiave **private**, **protected** o **virtual**, come si vedrà in seguito.

```
public:
  CDerivata(int , char); //costruttore della classe CDerivata
  void setLetteraNumero(int, char);
  char getLettera() { return lettera; };
};
```

In questo caso, volendo leggere o modificare `numero`, si devono necessariamente utilizzare le *member function* pubbliche ereditate dalla classe `Cbase`, perché la classe `CDerivata` non ha accesso ai membri *private* della classe `CBase`.

Il metodo `setLetteraNumero` della classe `CDerivata` evidenzia questo meccanismo: al suo interno usa la *member function* `setNumero`, ereditata dalla classe `CBase`:

```
void CDerivata::setLetteraNumero(int i, char a)
{
  lettera = a;
  setNumero(i); // setNumero è un metodo ereditato da CBase
}
```

12.2.1 Ereditarietà pubblica e privata

La *keyword* **public**, nella sintassi del meccanismo dell'ereditarietà, assume il significato che tutti i membri *public*, ereditati dalla *parent class*, rimangono *public* anche nella *child class*. A volte potrebbe essere utile che tali membri siano trasformati in membri *private* nella classe derivata. Il meccanismo sintattico per realizzare questa scelta, com'è facile intuire, consiste semplicemente nell'utilizzare la *keyword* **private** nella dichiarazione di ereditarietà:

```
class Stack : private List {
  ...
}
```

In questo caso, tutti i membri pubblici della classe `List` diventano membri privati nella classe `Stack`.

12.2.2 Lista d'inizializzazione

Considerato che una *child class* erediterà tutte le proprietà di una *parent class*, è necessario che tutti i dati membri di quest'ultima siano inizializzati dal proprio costruttore, prima di essere usati. Per questo motivo, il costruttore di una *child class* chiama, come prima operazione, il costruttore senza parametri della *parent class*.

Che cosa succede se la classe da cui si eredita possiede solo costruttori con parametri? In questo caso i costruttori della *child class* devono necessariamente contenere anche dei riferimenti ai costruttori con parametri della *parent class*.

Poiché il costruttore con parametri della *parent class* è eseguito **prima** del costruttore della *child class*, la sua chiamata va inserita prima del corpo di quest'ultimo:

```
CDerivata::CDerivata(int i, char a) : CBase(i)
{
  lettera = a;
}
```

Lista di inizializzazione

Tutto ciò che segue «**:**» prende il nome di **lista di inizializzazione**.

Mediante questo meccanismo, l'argomento `i`, passato al costruttore della classe `CDerivata`, verrà utilizzato dal costruttore con parametri della classe `CBase` per inizializzare il membro *private* `numero`.

In questa espressione riferita all'esempio precedente

```
CDerivata d(5, 's');
```

`d` è definito come un oggetto della classe `CDerivata` e inizializzato con i valori tra parentesi. Il primo `(5)` è impiegato dal costruttore della classe `CBase` per inizializzare `numero`, il secondo `('s')` dal costruttore della classe `CDerivata` per definire `lettera`.

Tutte queste considerazioni valgono anche per il costruttore copia.

☞ **Osservazioni**

La lista d'inizializzazione può anche essere utilizzata per fornire dei valori iniziali ai data member della classe. Ad esempio, riferendosi al costruttore della classe `CDerivata` *dell'esempio precedente, si può inizializzare* `lettera` *aggiungendo identificatore e rispettivo valore nella lista d'inizializzazione:*

```
CDerivata::CDerivata(int i, char a) : CBase(i), lettera(a) {}
```

12.2.3 Distruttori nelle *child class*

Quando il programma esce dallo *scope* di un oggetto viene automaticamente richiamato il suo distruttore. Quando a essere distrutto è un oggetto di una classe derivata, dopo il proprio distruttore è richiamato il distruttore della *parent class*.

☞ **Osservazioni**

Si ricorda che gli oggetti allocati dinamicamente con l'operatore `new` *non vengono automaticamente distrutti quando il programma esce dallo spazio di visibilità dei loro puntatori. Per farlo è necessario utilizzare, esplicitamente, l'operatore* `delete`[3].

12.3 La parola chiave `protected`

Nella programmazione strutturata tradizionale, il concetto di *information hiding* è sostanzialmente una metodologia, ossia un insieme di regole dettate dall'esperienza e dal buon senso dei programmatori allo scopo di produrre un *software* più sicuro.

Nella programmazione orientata agli oggetti, il problema della sicurezza diventa più delicato e occorrono pertanto dei meccanismi più rigorosi, caratteristici del linguaggio, che consentano di ridurre o meglio eliminare il rischio di errori imprevisti.

Il meccanismo d'incapsulamento ne costituisce un esempio: definisce con precisione che cosa deve essere accessibile e che cosa no.

Nel caso di un programma complesso, in cui l'ereditarietà costituisce il cardine su cui si poggia l'intero progetto, l'impossibilità di rendere accessibili alle classi derivate alcuni dei dati *private* delle *parent class* può essere una restrizione troppo vincolante.

D'altra parte dichiarare *public* tutti i dati *private* della *parent class* non è certo una scelta accettabile, in tal modo i dati si renderebbero accessibili a tutti e non solo alle classi eredi.

---

[3] *Vedi* il sottoparagrafo 11.3.2 *L'operatore* `delete`.

Per risolvere questo problema, il *C++* mette a disposizione dell'utente una terza parola chiave per controllare la visibilità dei membri di una classe: **protected**.

Un membro protetto, ai fini della *parent class* e delle sue classi *friend*, si comporta come un membro privato. Per le classi derivate **direttamente**, e **solo** per queste, si comporta, invece, come un membro pubblico. Ad esempio:

```
class CBase {
public:
  int a;
protected:
  int b;
private:
  int c;
};
class CDerivata : public CBase {
public:
  void  setProtected(int i);
  int   getProtected();
};
void CDerivata::setProtected(int i)
{
  // Nella child class è consentito l'accesso ai membri protected
  b = i;
  // ERRORE! È vietato l'accesso ai membri private
  c = i;
}
int CDerivata::getProtected()
{
  return b; // L'accesso ai membri protected è ammesso
}
void main()
{
  CBase     Uno;
  CDerivata Due;
  int       Val;
  ...
  Uno.a = 1; // Ammesso l'accesso ai membri public di CBase
  Uno.b = 2; // ERRORE! È vietato l'accesso ai membri protected
  Due.a = 3; // Ammesso. a è anche un membro public di CDerivata
  Due.b = 4; // ERRORE! È vietato l'accesso ai membri protected
  Due.c = 5; // ERRORE! È vietato l'accesso ai membri private
  /*
  Queste due funzioni accedono al membro b con un metodo pubblico
  della classe CDerivata
  */
  Due.setProtected(31);
  Val = Due.getProtected();
}
```

È importante osservare che la dichiarazione `protected` non è ereditata automaticamente dalle classi derivate successive alla prima; infatti, uno dei punti fondamentali della protezione

dei dati nel linguaggio *C++* consiste nel fatto che il diritto di accesso ai dati di una classe è concesso dalla *parent class* e non implicitamente acquisito dalla *child class*. Ai fini di una programmazione più pulita, si consiglia in ogni modo di non abusare delle possibilità offerte dal livello di visibilità `protected`.

☞ **Osservazioni**

Se nella dichiarazione di ereditarietà, il nome della classe base è preceduto dalla keyword **`protected`**, *tutti i membri pubblici e protetti di quella classe diventano membri protetti nella classe derivata.*

12.4 Le funzioni virtuali

Parlando di ereditarietà, occorre precisare che un oggetto di una *child class* è a tutti gli effetti anche un oggetto della *parent class* e, di conseguenza, può essere puntato da un puntatore alla *parent class* senza eseguire un *cast* di tipo.

Questa particolare proprietà dell'ereditarietà è alla base dell'argomento di questo paragrafo il cui obiettivo è quello di descrivere un nuovo meccanismo offerto dal *C++* per operare su oggetti, istanze di classi derivate, in modo efficiente e particolarmente elegante: **le funzioni virtuali**.

Una funzione virtuale è dichiarata in una *parent class* e definita nelle varie classi derivate. La dichiarazione è effettuata facendo precedere il prototipo della funzione dalla parola chiave **`virtual`**; la definizione, invece, avviene come quella delle altre *member function* delle *child class*.

La chiamata a una funzione virtuale è realizzata mediante un puntatore, di tipo **classe base**, inizializzato con l'indirizzo di un oggetto della classe derivata: la funzione che sarà effettivamente eseguita durante il *runtime* sarà la *member function* (della *child class*) omonima della funzione virtuale della classe base.

Allo scopo di illustrare meglio questo nuovo argomento, si propone un esempio più articolato che servirà anche come riepilogo sull'ereditarietà delle classi. Sono presi in considerazione dei semplici componenti elettrici passivi (resistori, condensatori, induttori), la cui caratteristica comune è avere due terminali di collegamento, ossia di essere dei componenti bipolari. In tal caso, è ragionevole creare una *parent class* di nome `bipolo`, dalla quale deriveranno le classi dei componenti specifici. La classe `bipolo` oltre ai dati membri comuni alle *child class*, contiene la funzione **virtuale** `visual`.

Il programma, `VIRTUAL1` è stato suddiviso in moduli d'interfaccia (`.h`) e d'implementazione (`.cpp`)([4]). Il file `BIPOLO.H` rappresenta la sezione d'interfaccia della classe `bipolo`, implementata in `BIPOLO.CPP`.

```
/************************ BIPOLO.H ************************/
#include <iostream>
#include <string>

using namespace std;

// Classe base
class bipolo {
public:
```

([4]) *Vedi* l'Approfondimento 11.2 *Organizzazione di un file C++*.

```
  // Dichiarazione delle funzioni virtuali
  virtual void visual();

  virtual ~bipolo(){}; // Distruttore virtuale

protected:
  string Caratteristica;
  string UnitaMisura;
  double Valore;
};

/********************** BIPOLO.CPP ***********************/
#include "bipolo.h"

/* Definizione (opzionale per la classe base) della funzione
   virtuale visual */
void bipolo::visual()
{
  cout << "Nessun valore inizializzato" << endl;
}
```

Nell'esempio, `visual`, dichiarata come `virtual`, non esegue nulla di rilevante, tranne visualizzare un messaggio informativo per l'utente. Anche il distruttore di `bipolo` è dichiarato come `virtual`. Entrambe le loro definizioni, come *member function* della classe base, non sono obbligatorie.

Sono quindi definite le classi derivate relative ai componenti: `resistore`, `condensatore` e `induttore`. Ogni classe comprende i relativi costruttori e il proprio metodo di visualizzazione `visual`. Il file d'interfaccia è `COMPONEN.H`, quello d'implementazione `COMPONEN.CPP`.

```
/********************** COMPONEN.H ***********************/
#include "bipolo.h"

// Classe derivata resistore
class resistore : public bipolo {
public:
  resistore(double R = 0);
  ~resistore() { cout << "Resistore distrutto" << endl; };

  void visual()
  {
    cout << Caratteristica << " = " << Valore << UnitaMisura
         << endl;
  }
};

// Classe derivata condensatore
class condensatore : public bipolo {
public:
  condensatore(double C = 0, double Vc0 = 0);
  ~condensatore() { cout << "Condensatore distrutto" << endl; };

  void visual()
  {
```

```
    cout << Caratteristica << " = " << Valore << UnitaMisura
         << ", Vc(0) = " << Stato << "V" << endl;
  };
private:
  double Stato; // Tensione iniziale ai capi del condensatore
};

// Classe derivata induttore
class induttore : public bipolo {
public:
  induttore(double L = 0, double Il0 = 0);
  ~induttore() { cout << "Induttore distrutto" << endl; };

  void visual()
  {
    cout << Caratteristica << " = " << Valore << UnitaMisura
         << ", Il(0) = "<< Stato << "A" << endl;
  };
private:
  double Stato; // Corrente iniziale dell'induttore */
};

/********************* COMPONEN.CPP ********************/
#include "componen.h"

// Costruttore di resistore
resistore::resistore(double R)
{
  Caratteristica = "Resistenza";
  UnitaMisura = " ohm";
  Valore = R;
}

// Costruttore di condensatore
condensatore::condensatore(double C, double Vc0)
{
  Caratteristica = "Capacità";
  UnitaMisura = " F";
  Valore = C;
  Stato = Vc0;
}

// Costruttore di induttore
induttore::induttore(double L, double Il0)
{
  Caratteristica = "Induttanza";
  UnitaMisura = " H";
  Valore = L;
  Stato = Il0;
}
```

Nella funzione `main` del modulo principale, `VIRTUAL1.CPP`, è definito un vettore di puntatori a oggetti di tipo `bipolo`, che vengono creati dinamicamente mediante l'operatore `new`([5]).

Prog. Virtual1.cpp: Funzioni virtuali

```
#include "componen.h"

void main()
{
  /*
  Definizione del vettore di puntatori alla classe bipolo
  e creazione dinamica degli oggetti
  */
  bipolo *ptr[] = {
       new resistore(100),
       new condensatore(2.2e-3, 5),
       new induttore(1e-2, .15)
  };

  // Determinazione del numero di oggetti effettivamente presenti
  const unsigned Oggetti = sizeof ptr / sizeof(bipolo *);

  // Visualizzazione dei parametri di tutti i bipoli esistenti
  for (register i = 0; i < Oggetti; i++)
    ptr[i]->visual();

  // cancella gli oggetti puntati dai puntatori alla classe bipolo
  for (i = 0; i < Oggetti; i++) delete ptr[i];
}
```

Il programma eseguibile `VIRTUAL1` si ottiene collegando, mediante il *linker*, i vari moduli oggetto generati dal compilatore.

Eseguendo il programma si produce l'uscita:

```
Resistenza = 100 Ohm
Capacità = 0.0022 F, Vc(0) = 5 V
Induttanza = 0.01 H, Il(0) = 0.15 A
Resistore distrutto
Condensatore distrutto
Induttore distrutto
```

Se il metodo `visual` e il distruttore `~bipolo` non fossero definiti come `virtual` nella *parent class* `bipolo`, a ogni chiamata verrebbe eseguita sempre la stessa *member function*, `bipolo::visual` e verrebbe invocato solo il distruttore `~bipolo::bipolo`. In questo caso l'uscita ottenuta sarebbe stata:

```
Nessun valore inizializzato
Nessun valore inizializzato
Nessun valore inizializzato
```

Invece, in `VIRTUAL1`, sono eseguite le corrispondenti *member function* `visual` di ogni *child class*, in base alla sequenza indicata nel ciclo `for`.

([5]) *Vedi* paragrafo 11.11 *Vettori di oggetti.*

In questo modo, una funzione, dichiarata `virtual` nella classe base, può assumere differenti forme nelle varie classi derivate. Questa particolare «abilità» prende il nome di **polimorfismo**.

Il file `BIPOLO.H` impiega la classe **`string`** della *standard C++ library*(6) per creare le stringhe `Caratteristica` e `UnitaMisura`.

I metodi della classe `string` sono peraltro molto simili a quelli del *container* **`vector`**. L'esempio `VIRTUAL2.CPP` utilizza questo *template* della *STL* per implementare il vettore di puntatori alla classe `bipolo`.

Ogni volta, il metodo **`push_back`** inserisce alla fine della sequenza un puntatore all'oggetto creato dinamicamente con `new`.

`componenti` e `iter` sono *alias*, rispettivamente della classe *template* `vector` e del suo iteratore.

In entrambe le implementazioni del programma `VIRTUAL`, l'uscita non cambia.

Prog. Virtual2.cpp: *Container* e allocazione dinamica degli oggetti

```
#include "componen.h"
#include <vector>

typedef vector<bipolo *> componenti;
typedef componenti::iterator iter;

void main()
{
  componenti ptr;
  /*
  Creazione dinamica degli oggetti
  */
  ptr.push_back(new resistore(100));
  ptr.push_back(new condensatore(2.2e-3, 5));
  ptr.push_back(new induttore(1e-2, .15);

  // Visualizzazione dei parametri di tutti i bipoli esistenti
  for (iter i = ptr.begin(); i != ptr.end(); i++)
    (*i)->visual();

  // cancella il contenuto dei puntatori alla classe bipolo
  for (i = ptr.begin(); i != ptr.end(); i++) delete *i;

}
```

12.4.1 Distruttori virtuali

Se una classe possiede metodi virtuali, è bene che anche il suo distruttore sia virtuale in modo che, se degli oggetti derivati sono distrutti mediante il puntatore alla classe base, sia sempre chiamato il distruttore corretto.

Un distruttore virtuale è implementato dichiarando come `virtual` il distruttore della *parent class*. Questo distruttore assicura che, applicando l'operatore `delete` a un puntatore alla classe base, sia chiamato prima il distruttore della *child class* corrispondente.

(6) *Vedi* l'Approfondimento 11.1 *La classe* `string`.

Nei file `VIRTUAL`, degli esempi precedenti, l'operatore `delete` è applicato ai puntatori alla classe `bipolo`. Mediante il meccanismo appena descritto, ognuno di loro richiamerà, nell'ordine, il distruttore della corrispondente *child class* e quindi quello della *parent class*.

12.5 Polimorfismo (*Polymorphism*)

Uno dei principi fondamentali della programmazione orientata agli oggetti, insieme all'incapsulamento e all'ereditarietà, è il **polimorfismo**.

In generale, è possibile far svolgere una determinata azione a un oggetto inviandogli un messaggio: questa caratteristica diventa estremamente importante quando oggetti differenti, ma logicamente imparentati, sono in grado di rispondere allo stesso messaggio svolgendo lo stesso tipo di azione, ognuno, però, in modo diverso. In *C++* il polimorfismo s'implementa mediante il meccanismo delle funzioni virtuali.

Il meccanismo sintattico delle funzioni virtuali si basa su un diverso modo di operare del compilatore e del *linker*, definito ***late binding***, o **collegamento ritardato**. I compilatori tradizionali, invece, operano secondo la tecnica ***early bindig***, o **collegamento anticipato**, dove il *linker* sostituisce il nome simbolico delle funzioni chiamate con il loro indirizzo di memoria.

Nel caso di chiamate a funzioni virtuali, non essendo noto **in anticipo** (nel *compile time*) quale funzione sarà effettivamente eseguita, tale sostituzione non è possibile al momento del *linking*, ma dovrà avvenire, inevitabilmente, **più tardi** durante il *runtime*.

È il *late binding* che consente, alle classi derivate, di applicare il proprio metodo ai loro oggetti con la stessa azione vale a dire una chiamata all'omonima funzione virtuale.

12.5.1 Il meccanismo delle *virtual function*

Quando il compilatore e il *linker* processano una chiamata a una normale *member function*, inseriscono nel file oggetto il codice macchina corrispondente, il quale comprende anche l'indirizzo della funzione.

Se la *member function* è dichiarata come `virtual`, la classe che la contiene e quelle derivate hanno un *data member* nascosto (`vfptr`) che punta a una tabella (`vtable`) di puntatori alle omonime funzioni, effettivamente implementate nelle classi derivate. Detta tabella è **condivisa da tutti** gli oggetti della classe **derivata** ([7]).

Quando un oggetto viene istanziato, il costruttore di ogni classe derivata inizializza automaticamente `vfptr` con l'indirizzo della `vtable` corrispondente, consentendo, così, il corretto *binding* di ogni chiamata a funzioni virtuali ([8]).

Ad esempio, un'espressione come quella del prog.Virtual1.cpp:

```
ptr[i]->visual();
```

è di fatto sostituita dalla seguente espressione:

```
( *( ( ptr[i]->vfptr )[0] ) )();
```

([7]) Questo tipo d'implementazione dipende dal compilatore utilizzato e non è imposto dallo standard del linguaggio. `vfptr` e `vtable` sono nomi fittizi.

([8]) Per questo motivo non è possibile dichiarare costruttori virtuali.

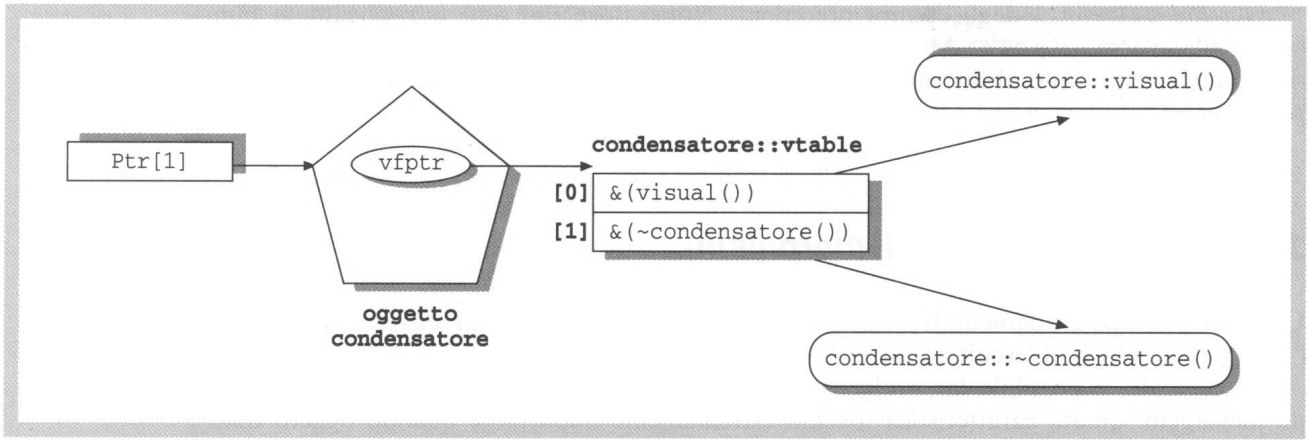


Fig. 12.3 *Late binding.*

Viene prima esaminato il *data member* `vfptr` dell'oggetto puntato da `ptr[i]`. In questo caso punta a una `vtable` che contiene due elementi: l'indirizzo dei metodi virtuali, `visual` (elemento `[0]`) e del distruttore (elemento `[1]`). La funzione effettivamente chiamata (`visual`) dipende dal valore dell'indice di questa tabella (`[0]`).

La fig. 12.3 mostra, graficamente, il funzionamento di questo meccanismo di chiamata nel caso l'oggetto puntato da `ptr` sia di classe `condensatore`.

Osservazioni

Il meccanismo delle funzioni virtuali è molto potente, ma può dar luogo a errori insidiosi se impiegato in modo non corretto. Per limitare questi rischi, se è previsto che una classe possa essere derivata, è bene che i metodi esposti nella sua sezione d'interfaccia (che potrebbero essere ridefiniti), siano dichiarati sempre come `virtual`.

12.5.2 Classi astratte

Le classi astratte rappresentano espressioni di concetti generali dalle quali possono essere derivate classi più specializzate.

Una classe che contiene **almeno una funzione virtuale pura** è considerata una **classe astratta**.

Una funzione virtuale è considerata «pura» se utilizza la sintassi dello **specificatore puro** (**=0**). Nell'esempio precedente avremmo potuto inserire nella *parent class* una funzione virtuale pura mediante l'espressione:

```
virtual void visual()=0;
```

Per queste funzioni non è necessaria la definizione nella classe base. Invece, le classi derivate dalle classi astratte devono implementare le funzioni virtuali pure della classe, altrimenti sono considerate a loro volta classi astratte. In tal modo, unendo ereditarietà e polimorfismo, s'individua una strategia per definire una struttura comune a tutta una gerarchia di classi.

Non è possibile istanziare oggetti di classi astratte, ma è possibile utilizzare puntatori e *reference* a classi astratte([9]).

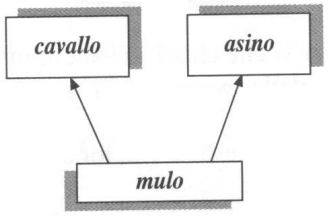
([9]) *Vedi*, a tale proposito, l'esercizio 1 di questo capitolo *Prede e predatori*.

12.6 Ereditarietà multipla

Nell'introduzione si è discussa l'importanza che la classificazione assume nel produrre conoscenza.

Nelle classificazioni esistono però delle regioni d'indeterminazione nelle quali la semplice ramificazione ad albero non è sufficiente a descriverle. In questi casi è necessario ricorrere a sistemi più raffinati di rappresentazione; ad esempio, certi oggetti potrebbero avere elementi in comune con più classi. La semplice ereditarietà si rivela quindi un meccanismo troppo limitato e non consente di risolvere questi problemi.

Supponendo di voler spiegare a un bambino che cos'è un mulo, il modo più semplice è descriverlo come l'animale che deriva dall'incrocio di un cavallo con un asino e che eredita le caratteristiche di entrambi. In questo modo si riuscirà a fornire una descrizione sufficientemente chiara senza perdersi nel dettaglio dei singoli particolari.



Un'innovazione apportata al linguaggio *C++* è proprio l'**ereditarietà multipla**, che consiste nella possibilità di derivare un oggetto da più classi.

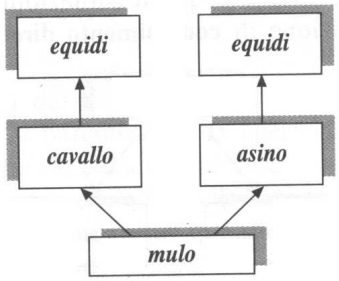
La sintassi usata per rappresentarla è una naturale estensione dell'ereditarietà semplice. Nell'esempio:

```
class mulo : public cavallo, public asino {
  ...
}
```

la classe `mulo` eredita tutti i *data member* e tutte le *member function* delle classi `cavallo` e `asino`.

Nella lista d'inizializzazione una stessa classe non può apparire due volte come classe base.

In particolare, si potrebbe anche verificare la situazione, non voluta, che le due classi base derivino, a loro volta, da una stessa classe:

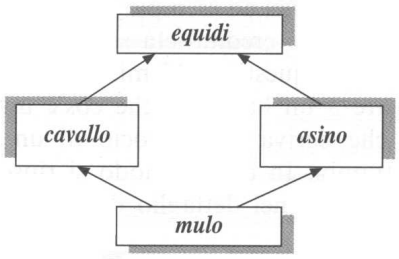


Nell'esempio, fra i membri delle classi `cavallo` e `asino`, sono presenti anche quelli della classe `equidi`, che sono così ereditati due volte dalla classe `mulo`. La sintassi per implementare una tale struttura è la seguente:

```
class equidi {...};
class cavallo : public equidi {...};
class asino : public equidi {...};
class mulo : public cavallo, public asino {....};
```

12.7 Classi virtuali

In alcuni casi, l'ereditarietà multipla può essere una scelta voluta, ma spesso si preferisce evitarlo: riferendosi all'esempio precedente, non è necessario che le caratteristiche (membri) della classe `equidi` compaiano più di una volta nella classe `mulo`. Nell'ipotesi appena formulata, la struttura gerarchica assume il seguente aspetto a rombo([10]):



Questa soluzione ha il pregio di far sì che classi «vicine», in questo caso `cavallo` e `asino`, ereditino le stesse caratteristiche della *parent class* `equidi` trasferendole solo una volta alla classe `mulo`.

Il *C++* consente di implementare questa astrazione mediante la parola chiave **`virtual`**:

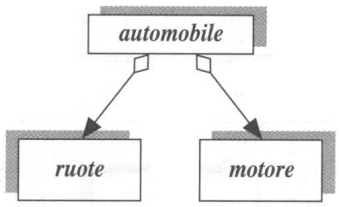
```
class equidi {...};
class cavallo : virtual public equidi {...};
class asino : virtual public equidi {...};
class mulo : public cavallo, public asino {....};
```

Questa sintassi dichiara la *parent class* `equidi` come virtuale e fa sì che i suoi membri compaiano una sola volta nelle classi ottenute per derivazione successiva.

I meccanismi di ereditarietà semplice, multipla e di classe virtuale si possono combinare a piacere per realizzare sistemi raffinati e veramente potenti.

12.8 Classi composte

Esistono delle applicazioni in cui è impossibile descrivere degli oggetti mediante il meccanismo dell'ereditarietà. Ad esempio, sebbene un'automobile sia costituita da oggetti di tipo ruota, motore ... non si può certo dire che sia derivata da un particolare tipo di ruota o di motore. In generale, molti oggetti sono composti da altri oggetti, composti da altri ancora; il *C++* consente di modellizzare queste situazioni attraverso il concetto di **composizione**, altresì noto come relazione di **contenimento diretto**.



Si supponga di definire le classi `ruote` e `motore` come segue:

```
class ruote {
public:
  ruote(unsigned Diametro) { Formato = Diametro; };
  ...
```

([10]) Un esempio concreto di questo tipo di struttura si può trovare nell'Approfondimento 12.1 *Iostream class library*.

```
private:
  unsigned Formato;
};

class motore {
public:
  motore(unsigned Power) { Potenza = Power; };
  ...
private:
  unsigned Potenza;
};
```

È possibile, allora, dichiarare la classe automobile:

```
class automobile {
public:
  automobile(ruote a, ruote p, motore hp, unsigned Posti);
  ...
private:
  ruote Anteriori, Posteriori;
  motore Cavalli;
  unsigned Passeggeri;
};
```

che comprende due dati membri di tipo ruote e motore.

Il comportamento di una classe così definita è identico a quello di una classe contenente solo dati membri di tipo fondamentale, ma è necessario che gli oggetti membri siano inizializzati **prima** che il programma entri nel corpo del costruttore della classe composta.

Per provvedere a questo compito, il costruttore dell'oggetto composto deve necessariamente comprendere, nella propria **lista di inizializzazione**, un elenco delle chiamate ai costruttori degli oggetti componenti, analogamente a quanto avviene per le classi derivate.

```
// Costruttore della classe automobile
automobile::automobile(ruote a, ruote p, motore hp, unsigned Posti) :
                        Anteriori(a), Posteriori(p), Cavalli(hp)
{
  Passeggeri = Posti;
};
```

La lista di inizializzazione inizia dai due punti che seguono l'intestazione del costruttore e termina quando è incontrata la parentesi graffa aperta con la quale inizia il suo corpo.

Se la classe è derivata da altre, le chiamate ai costruttori di queste devono essere poste nella lista di inizializzazione **insieme** a quelle delle classi componenti.

Analogamente, ma con sequenza inversa, quando l'oggetto sarà distrutto, sarà automaticamente richiamato il distruttore dell'oggetto composto e poi i distruttori degli oggetti contenuti.

12.9 Conclusioni

La programmazione orientata costituisce una particolare filosofia di approccio ai problemi e il *C++* rappresenta un valido strumento, sebbene complesso, per implementare un programma secondo questa tecnica.

Di seguito sono elencate alcune regole pratiche per programmare con un linguaggio di tipo *OOP*.

1) Non serve, almeno inizialmente, definire che cosa fa il sistema, ma con che cosa opera, ovvero è necessario identificare le singole entità. È chiaro che, per ultimare l'applicazione, si dovrà necessariamente rispondere alla prima domanda, ma questo avverrà solo in un secondo tempo. L'individuazione delle entità fondamentali del problema che si sta affrontando, assume un'importanza prioritaria al fine poi di esplicitarle sotto forma di idee o concetti. La struttura del programma dovrà riflettere queste idee nel modo più diretto possibile. Tutto ciò può disorientare la maggior parte dei programmatori, ma l'approccio *OOP* costituisce l'unico modo per ottenere quelle caratteristiche di riusabilità ed estensibilità alle quali si è più volte accennato.
2) Ogni idea deve trasformarsi in una classe. Se due o più classi possiedono qualcosa in comune, occorre definire una classe fondamentale da cui derivare le altre. Gli obiettivi principali nella programmazione orientata agli oggetti possono quindi riassumersi nei seguenti punti:

— identificare le classi fondamentali;
— definire gli oggetti;
— trovare il tipo di relazioni fra gli oggetti;
— creare una gerarchia di oggetti;
— definire la struttura del programma che consente la comunicazione fra gli oggetti.

La struttura di un programma secondo la tecnica *OOP* è un processo di tipo iterativo in cui si può seguire ogni fase in modo indipendente da quelle precedenti che, se necessario, si possono rimettere in discussione e modificare.