CLASSI 11

Uno dei principali obiettivi di un esperto programmatore dovrebbe essere quello di produrre del software di qualità. La cosa non è semplice, perché coinvolge una serie complessa di fattori concomitanti che insieme qualificano il prodotto. Nel corso del libro si è cercato di concretizzare il concetto di qualità definendo i criteri e le tecniche indispensabili per raggiungere tale obiettivo, per lo meno con un linguaggio di tipo procedurale come il C; l'acquisizione del paradigma([1]) *della programmazione orientata agli oggetti richiede un affinamento di tali tecniche senza tuttavia invalidare nessuna delle precedenti e, soprattutto, un diverso modo di rapportarsi ai problemi. Di seguito sono elencate alcune definizioni che sottendono concetti estremamente importanti; sebbene questa classificazione non sia standardizzata, è in parte accettata da molti esperti del settore:*

— **robustezza:** *la capacità di un programma di funzionare correttamente anche in condizioni anomale;*
— **estensibilità:** *la facilità con la quale un software si può modificare in relazione a variazioni delle specifiche;*
— **compatibilità**: *la facilità con la quale un software si può interfacciare con altri prodotti software.*

Il meccanismo linguistico della **classe**, *discusso nel capitolo, si rivelerà uno strumento concettuale veramente potente e idoneo a soddisfare i requisiti necessari per sviluppare software di buona qualità.*

La classe rappresenta uno dei concetti dell'applicazione che si vuole realizzare e deve esserlo nel modo più astratto e generale possibile.

Mediante la classe è quindi possibile definire degli oggetti che costituiranno gli elementi su cui verterà l'elaborazione. Per soddisfare i requisiti di robustezza, estensibilità e compatibilità di cui sopra, un oggetto si deve comportare come una «scatola nera», vale a dire come un dispositivo di cui non si conosce la struttura interna, ma che possiede delle funzionalità ben definite. Tutte le variabili di tipo fondamentale del linguaggio C possono essere considerate esempi di oggetti di questo tipo: è possibile compiere delle operazioni su di loro senza che si conoscano i dettagli del modo in cui vengono eseguite internamente.

Il C++ offre in più la possibilità di definire nuove entità astratte dotate di un proprio insieme di caratteristiche, utilizzabili in modo indipendente dalla loro implementazione.

11.1 Introduzione

La caratteristica fondamentale del linguaggio che differenzia nettamente il *C++* dal *C* è la **classe** (si ricorda che, originariamente, questo linguaggio fu definito «*C* con classi»).

La classe presenta forti analogie con il tipo aggregato **struttura**. Nel paragrafo 5.6 si è visto che il concetto di struttura nasce dall'esigenza di manipolare insiemi eterogenei

([1]) *Vedi* Lettura *Paradigmi di programmazione*.

d'informazioni fra loro logicamente collegate. Ciò consente di creare delle nuove variabili che meglio modellano i dati riguardanti il problema che l'utente sta trattando.

L'inizializzazione di una **struttura** può avvenire direttamente in fase di dichiarazione o mediante istruzioni di assegnazione. In entrambi i casi, però, non è mai possibile avere la certezza che la struttura contenga dei dati validi; infatti, si possono sempre verificare degli errori accidentali che ne falsano il contenuto.

In che modo risolvere il problema della sicurezza e legittimità dei dati? Una soluzione parziale potrebbe consistere nell'assegnare i valori ai dati della struttura in fase di dichiarazione e, in seguito, di consentirne l'accesso solo attraverso funzioni: queste conterranno operazioni per controllare la validità e la coerenza dei nuovi valori immessi.

Il programma `PUNTO.CPP` fornisce una dimostrazione in tal senso. La struttura definisce un dato aggregato di nome `PUNTO`, con l'obiettivo di descrivere le coordinate di un punto in sistema di assi cartesiani. La funzione `setXY()` consente di assegnare ai membri della struttura solo dei valori compresi nell'intervallo `0 ÷ 100`:

Prog. Punto.cpp: Definizione della struttura `PUNTO` e della funzione di accesso ai dati membri `Set`

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

// Definizione del tipo PUNTO
struct PUNTO {
   float x, y;
};

// Prototipo della funzione Set
PUNTO setXY();    // Funzione utilizzata per modificare x e y

void main()
{
  PUNTO a = {0, 0};     // Valori di default

  a = setXY();
  cout << "Ascisse : " << a.x << endl;
  cout << "Ordinate: " << a.y << endl;
}

PUNTO setXY()
{
  PUNTO temp;

  // Acquisizione e test del valore delle ascisse
  do {
    cout << "Valore delle ascisse [0 - 100]: ";
    cin >> temp.x;
  } while (temp.x < 0 || temp.x > 100);

  // Acquisizione e test del valore delle ordinate
  do {
    cout << "Valore delle ordinate [0 - 100]: ";
    cin >> temp.y;
  } while (temp.y < 0 || temp.y > 100);

  return temp;
}
```

Nel caso di programmi complessi tutto ciò richiede un maggiore impegno, sia in fase di progettazione (definizione delle specifiche) sia in fase d'implementazione e successivo *de-*

bugging. È in ogni modo importante osservare che, nonostante le precauzioni adottate, è sempre possibile manipolare i dati della struttura attraverso un'assegnazione diretta del tipo:

```
a.x = 200;
```

vanificando, in tal modo, qualunque tipo di controllo sui valori. La possibilità che questa evenienza si verifichi non è improbabile, specie nel caso in cui più persone lavorano a uno stesso programma.

Certo, una scelta di comodo potrebbe essere quella di ignorare completamente certe tecniche «di sicurezza», riducendo di conseguenza la **robustezza** del programma stesso, ossia la capacità di funzionare correttamente anche in condizioni anomale, ma la strada da intraprendere è un'altra.

I linguaggi orientati agli oggetti permettono all'utente di dichiarare nuovi tipi di variabili con facili meccanismi di protezione dei dati (**incapsulamento**), senza tuttavia richiedere un eccessivo dispendio di energia in fase di progettazione e implementazione. Nel *C++* questi nuovi tipi prendono il nome di **classi** e costituiscono le nuove entità fondamentali che consentono la dichiarazione di **oggetti**.

L'introduzione del meccanismo linguistico di classe, nel mondo dei linguaggi di programmazione, ha come scopo quello di simulare un meccanismo concettuale tipico del pensiero umano: qualunque tentativo dell'uomo di descrivere la realtà che lo circonda (modellizzare) passa attraverso un processo di astrazione, generalizzazione e classificazione.

Questo modo di ragionare consente all'uomo di farsi un'idea di un problema anche complesso, senza per questo distoglierlo, almeno nella fase iniziale, dai dettagli relativi a un aspetto particolare del problema. È sufficiente sfogliare un comune vocabolario per riscoprire che ogni sostantivo rappresenta una classe astratta di oggetti, con in comune una serie di caratteristiche o di comportamenti.

In modo analogo la classe utilizza nuovi tipi di dati con maggiori proprietà di astrazione e dotati di sistemi di occultamento delle informazioni relative ai dettagli costitutivi della classe stessa: queste due caratteristiche prendono rispettivamente il nome di ***data abstraction*** e ***information hiding***; quest'ultima, oltre a semplificare l'uso della classe, impedisce un uso improprio dei dati in essa contenuti.

La definizione di classe, come si vedrà, è relativamente semplice e il paragrafo sarà interamente dedicato a chiarire questo concetto. Maggiori difficoltà s'incontreranno studiando e sviluppando tutti i collegamenti fra le classi e le altre caratteristiche peculiari del *C++*, che costituiscono la base della **programmazione orientata agli oggetti**.

11.2 Incapsulamento: le classi

L'incapsulamento (molto meglio in inglese ***encapsulation***), costituisce il primo principio della programmazione ***OOP*** e significa semplicemente unire insieme dei dati strutturati con le funzioni che li manipolano. Il meccanismo sintattico che consente di realizzare l'*encapsulation* è la **classe** ([2]).

Le funzioni di una classe sono definite ***member function*** o **metodi** della classe.

Una classe può nascondere all'utente la sua struttura interna e consentire l'accesso ai **dati membri** (o **attributi**) solo mediante le *member function*.

La definizione di una classe definisce un nuovo tipo o categoria di dati, che servirà all'utente per modellare in modo più accurato e sicuro i dati reali del problema trattato.

Il programma `CLASS.CPP` costituisce una possibile evoluzione dell'esempio precedente.

([2]) In realtà, in *C++*, anche le strutture e le unioni consentono l'*encapsulation*. *Vedi* anche il paragrafo 11.12 *Tipi di classe*.

Prog. Class.cpp: Un primo esempio di classe

```
#include <iostream>

using namespace std;

// Definizione di una classe
class CPunto {
public:
  void setXY();      // Member function che modifica x e y
  float getX();      // Member function per conoscere x
  float getY();      // Member function per conoscere y
private:
  float x, y;        // Dichiarazione di dati membri
};

// Definizione di member function (metodo) per modificare x e y
void CPunto::setXY()
{
  // Acquisizione e test del valore delle ascisse
  do {
    cout << "Valore delle ascisse [0 - 100]: ";
    cin >> x;
  } while (x < 0 || x > 100);
  // Acquisizione e test del valore delle ordinate
  do {
    cout << "Valore delle ordinate [0 - 100]: ";
    cin >> y;
  } while (y < 0 || y > 100);
}
// Member function per ricavare x
float CPunto::getX() { return x; }

// Member function per ricavare y
float CPunto::getY() { return y; }

// Programma principale
void main()
{
  CPunto P1; // Dichiarazione dell'oggetto P1 di classe CPunto

  P1.setXY();
  cout << "Ascisse : " << P1.getX() << endl;
  cout << "Ordinate: " << P1.getY() << endl;
}
```

Il programma è molto simile al precedente e, leggendolo con attenzione, è facile intuirne il funzionamento; però alcuni concetti fondamentali ed elementi nuovi richiedono un chiarimento.

Dichiarazione di un oggetto

La prima istruzione del `main` è una dichiarazione di una variabile di tipo `CPunto`, tale variabile prende il nome di **oggetto**. Per creare un oggetto, in altre parole un'**istanza** di una classe, occorre un'adeguata dichiarazione; ad esempio:

```
CPunto P1;
```

Questa dichiarazione crea un oggetto([3]) di tipo `CPunto`, il cui identificatore è `P1`. È importante porre l'accento sul fatto che **una classe non è un oggetto**, ma solo una sua descrizione e come tale non ha riservata della memoria.

Specificatori di accesso `public` e `private`

Affinché un oggetto sia effettivamente utile, è indispensabile che nessuno possa intenzionalmente o accidentalmente modificarne i dati in modo illecito. A tale scopo, i membri di una classe sono classificati **pubblici** o **privati** utilizzando degli opportuni specificatori di accesso, **`public`** e **`private`**. Nel primo caso sono accessibili in tutto lo spazio di visibilità dell'oggetto di quella classe; nel secondo, l'accesso è possibile solo attraverso le sue *member function*.

Ad esempio, se nel `main` del programma precedente fosse stata scritta l'istruzione:

```
P1.x = 200;
```

il compilatore avrebbe segnalato un errore del tipo:

```
cannot access 'private' member declared in class 'CPunto'
```

evidenziando così l'impossibilità di accedere direttamente a un membro della classe classificato come privato.

Le *member function* `public` sono l'unico mezzo che il programmatore possiede per accedere ai dati privati dell'oggetto.

L'ordine con cui le due parole chiave appaiono nella classe non ha importanza, perché ognuna mantiene il suo effetto fino a quando non incontra un nuovo specificatore di accesso([4]).

In tal senso si parla anche di ***private section*** o ***public section*** riferendosi alle due aree tipiche di un oggetto.

Member function e operatore `::`

Nelle definizioni, le *member function* (o metodi) sono associate alla classe di appartenenza mediante l'operatore di *scope resolution* **`::`** che, in questo caso, assume il significato di ***class scope***:

```
tipo NomeClasse::NomeFunzione([[tipo [[Parametro1]],...]])
{
  ...
  ...
  ...
}
```

La visibilità di *`NomeFunzione`* è legata a quella della classe *`NomeClasse`*. È possibile, pertanto, definire altre funzioni, appartenenti a diverse classi (o a nessuna), con lo stesso identificatore *`NomeFunzione`* senza generare ambiguità o conflitti.

Messaggi

Nella programmazione orientata agli oggetti un metodo applicato a un oggetto viene detto **messaggio**. Un messaggio viene inviato a un oggetto mediante l'operatore punto « **.** » o l'o-

---

([3]) Una dichiarazione crea un'**istanza** di un tipo. In *C++*, le istanze delle classi sono chiamate **oggetti** mentre quelle di tutti gli altri tipi **variabili**.

([4]) Se gli specificatori di accesso, `public` e `private`, sono entrambi omessi, tutti i membri di una classe saranno considerati **`private`** per *default*.

peratore freccia «->» in modo del tutto analogo alle operazioni sulle strutture; ad esempio l'istruzione:

```
P1.getX();
```

invia il messaggio `getX()` all'oggetto `P1`.

Poiché le *member function* di una classe possono essere definite esternamente alla dichiarazione della classe, è possibile nascondere la loro implementazione. Questo sistema consente di suddividere la definizione della classe in due parti distinte: una visibile, chiamata **sezione d'interfaccia**, in cui sono dichiarati i prototipi dei metodi e l'altra nascosta, chiamata **sezione d'implementazione**, nella quale sono definiti tali metodi([5]).

11.2.1 Costruttori di una classe

Una delle cause di gravi e insidiosi errori in un programma è legata all'uso di variabili non inizializzate oppure inizializzate con valori non consentiti. La probabilità di commettere questo tipo di errore, diventa poi molto elevata se, a essere dichiarati, sono dati strutturalmente complessi. Poiché uno degli obiettivi dell'*OOP* è quello di ottenere una maggiore robustezza del codice, per limitare questo genere di errori, si è convenuto di offrire al programmatore l'opportunità di inizializzare gli oggetti «automaticamente» al momento della loro dichiarazione.

Questa operazione è possibile attraverso una *member function* particolare: il **costruttore**.

Il costruttore ha lo stesso nome della classe di cui è membro e non possiede alcun valore di ritorno (nemmeno `void`).

Nell'esempio che segue, tra le *member function* della classe `CPunto` è stato inserito anche il suo costruttore:

```
class CPunto {
public:
  CPunto();         // Costruttore
  void   setXY(); // Member function per modificare x e y
  float  getX();  // Member function per conoscere x
  float  getY();  // Member function per conoscere y
private:
  float x, y;       // Dichiarazione di dati membri
};
// Definizione del costruttore della classe CPunto
CPunto::CPunto()
{
  x = y = 0;
}
```

Al momento della dichiarazione dell'oggetto, sarà **automaticamente** chiamato il costruttore corrispondente che provvederà alla corretta inizializzazione.

In questo esempio, il costruttore non ha parametri e, pertanto, non è possibile inizializzare l'oggetto con valori diversi da quelli stabiliti al momento della sua implementazione. Tuttavia, grazie all'*overloading* delle funzioni, è possibile dichiarare più costruttori, al fine di assegnare all'oggetto valori iniziali differenti.

Nell'esempio, il secondo costruttore consente di inizializzare un oggetto con valori e criteri diversi da quelli stabiliti dal costruttore precedente:

```
class CPunto {
public:
  // Prototipi dei costruttori
```

([5]) *Vedi*, a tale proposito, l'Approfondimento 11.2 *Organizzazione di un file C++*.

```
  CPunto();
  CPunto(float x0, float y0);

  void    setXY();      // Member function per modificare x e y
  float   getX();       // Member function per conoscere x
  float   getY();       // Member function per conoscere y

private:
  float x, y;           // Dichiarazione di dati membri
};

// Definizione di un costruttore della classe CPunto
CPunto::CPunto()
{
  x = y = 0;
}

// Definizione di un altro costruttore della classe CPunto
CPunto:: CPunto(float x0, float y0)
{
  // Inizializzazione e controllo della validità dei dati
  if(x0 < 0)
    x = 0;
  else if(x0 > 100)
    x = 100;
  else
    x = x0;

  if(y0 < 0)
    y = 0;
  else if(y0 > 100)
    y = 100;
  else
    y = y0;
}
```

Adesso gli oggetti della classe `CPunto` potranno essere automaticamente inizializzati in due modi diversi: se al momento della loro dichiarazione non sono forniti valori iniziali, sarà chiamato il primo costruttore, altrimenti il secondo:

```
CPunto P1,          // P1 viene inizializzato con i valori 0, 0
       P2(10, 20),  /* P2 viene inizializzato con i valori
                       indicati*/
       P3(80, 224); /* P3 viene inizializzato con i valori 80,
                       100*/
```

I costruttori, come tutte le altre funzioni, possono anche contenere argomenti di *default*; in questo caso è necessario, tuttavia, fare attenzione a non creare ambiguità. Ad esempio, la dichiarazione di questi due costruttori non è ammessa:

```
class CPunto {
public:
  // Prototipi di costruttori
  CPunto();
  CPunto(float x0 = 0, float y0 = 0);
  ...
};
```

perché il compilatore non sarebbe in grado di stabilire quale dei due richiamare nel caso che la dichiarazione fosse:

```
CPunto P1; (6)
```

Quando il compilatore analizza la dichiarazione di un oggetto, richiama sempre un costruttore. Se nella definizione di una classe non esistono costruttori, il compilatore ne crea automaticamente uno di *default*, senza parametri e il cui corpo non contiene alcuna istruzione.

Talvolta, i costruttori possono essere chiamati esplicitamente per creare, in modo temporaneo, degli oggetti di una determinata classe. Ad esempio, le seguenti tre istruzioni:

```
CPunto p = CPunto(7.8, 9.3);
TracciaLinea(CPunto(2, 5), CPunto(-3, 8.1));
return CPunto(5.23, -18.5);
```

creano e inizializzano oggetti di tipo `CPunto` che verranno automaticamente distrutti al termine della valutazione delle espressioni in cui compaiono.

La prima ne fa una copia sull'oggetto `p` che si sta inizializzando; la seconda li passa come argomenti attuali alla funzione `TracciaLinea` e l'ultima rappresenta il valore di ritorno di una funzione.

Osservazioni

Sebbene la definizione dei costruttori non sia obbligatoria, è assolutamente consigliata per assicurare una corretta impostazione dei valori iniziali.

11.2.2 Distruttori di una classe

Un **distruttore** è una *member function* chiamata **automaticamente** dal compilatore quando un oggetto di una classe esce dal proprio spazio di visibilità. Il suo scopo è assicurare che siano eseguite determinate operazioni, prima che l'oggetto sia distrutto.

Un distruttore ha lo stesso nome della classe preceduto dal simbolo tilde `~`; non può avere argomenti né possedere valore di ritorno e deve essere unico: non è possibile effettuare l'*overloading* di distruttori.

L'esempio che segue aggiunge il distruttore alla classe `CPunto`:

```
class CPunto {
public:
  // Prototipo del costruttore
  CPunto(float x0 = 0, float y0 = 0);
  // Prototipo del distruttore
  ~CPunto();
  ...
};
// Definizione del distruttore della classe CPunto
CPunto::~CPunto()
{
  cout << Oggetto distrutto! << endl;
}
```

(6) In questo caso il problema si può risolvere, senza perdite di funzionalità, eliminando il costruttore senza parametri.

In questo caso, al termine dell'esecuzione del programma `CLASS` dell'esempio precedente, allo `stdout` sarà inviato il messaggio:

```
Oggetto distrutto!
```

Un distruttore non è importante come il costruttore e il suo uso è richiesto solo per alcune applicazioni dove, ad esempio, prima di distruggere un oggetto è necessario rilasciare la memoria allocata dinamicamente.

11.2.3 *Member function* definite all'interno di una classe

Nel caso in cui le *member function* siano formate da poche istruzioni è possibile inserire la loro definizione direttamente all'interno della classe; ad esempio:

```
class CPunto {
public:
  // Prototipo del costruttore
  CPunto(float x0 = 0, float y0 = 0);
  // Prototipo del distruttore
  ~CPunto();
  void setXY();     //member function per modificare x e y
  float  getX() { return x; };
  float  getY() { return y; };
private:
  float x, y;     // Dichiarazione di dati membri
};
```

Le *member function* `getX` e `getY` della classe `CPunto` contengono poche istruzioni e, per semplicità di rappresentazione, sono state trascritte all'interno della definizione di classe. Questo stile di programmazione produce lo stesso effetto che si avrebbe utilizzando la parola chiave **`inline`** nella definizione esterna delle stesse funzioni([7]).

11.2.4 Classi: una rappresentazione grafica

Volendo rappresentare graficamente il concetto di classe, si potrebbe disegnare una corona circolare, come in fig. 11.1, che protegga i dati membri privati dall'esterno. Solo attraverso

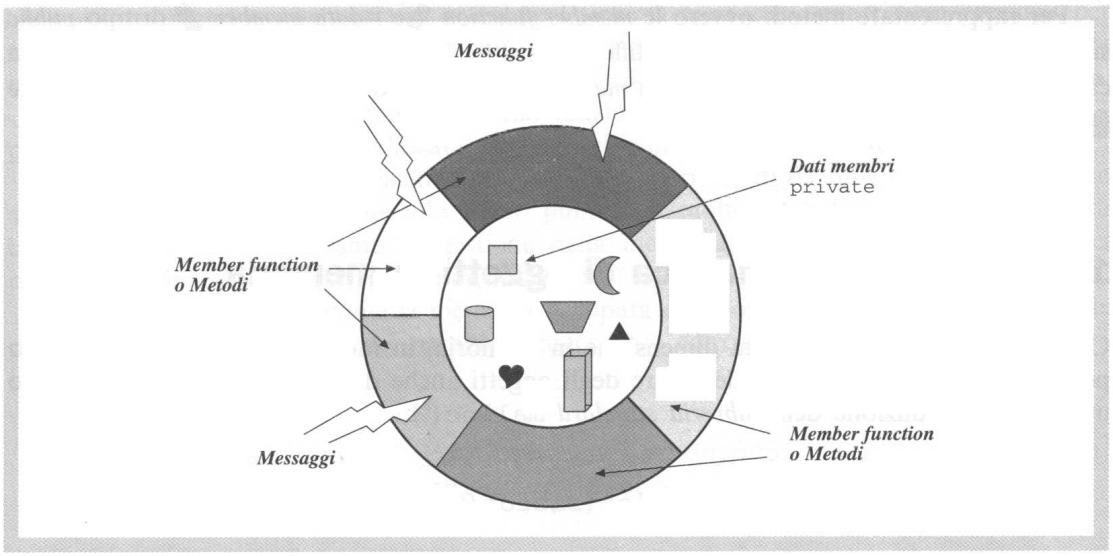


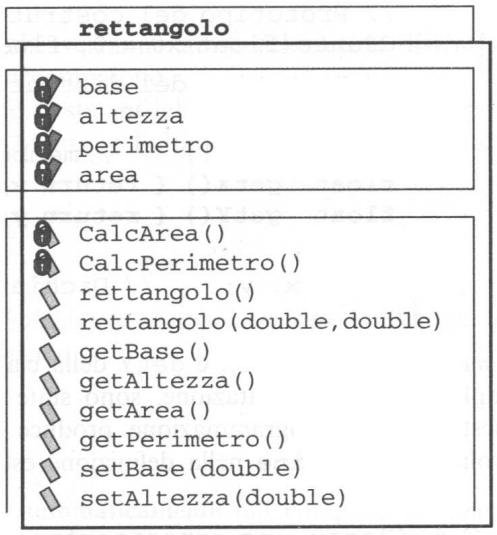
Fig. 11.1 Rappresentazione grafica di una classe.

([7]) In questo caso, però, non ha più molto senso separare la sezione d'implementazione da quella d'interfaccia.

le *member function* della corona è possibile inviare messaggi e avere informazioni sui dati membri privati della classe.

Un modello del genere forse riesce a rappresentare abbastanza bene il concetto di classe, ma certo non può considerarsi un sistema comodo per richiamare il concetto di classe nella fase di progettazione del programma. In questa fase si adotta un modello grafico più semplice rappresentato da un rettangolo. Quest'ultimo è solitamente suddiviso in tre parti: nella parte superiore s'inserisce il nome (identificatore) della classe; nella parte centrale s'inseriscono i dati membri della classe (attributi o *data member*); nell'ultima s'inseriscono i metodi della classe (*member function*). L'esempio mostra una classe e la relativa rappresentazione grafica.

```
class rettangolo{
private:
  void CalcArea();
  void CalPerimetro();
  double   base;
  double   altezza;
  double   perimetro;
  double   area;
public:
  rettangolo();
  rettangolo(double,double);
  double   getBase();
  double   getAltezza();
  double   getArea();
  double   getPerimetro();
  void setBase(double);
  void setAltezza(double);
};
```



Per rappresentare metodi ovvero le *member function* e i *data member* di tipo *public* si possono utilizzare rombi inclinati differentemente. Nella figura i lucchetti denotano la caratteristica di inaccessibilità dei *data member* privati o delle *member function* private della classe.

11.3 Allocazione dinamica di oggetti in memoria

Ogni oggetto occupa una certa dimensione in memoria, in modo rigido e per tutto il suo periodo di vita. È però possibile creare degli oggetti anche durante il *runtime*, ad esempio utilizzando la funzione della *libreria standard* `malloc`([8]):

```
CPunto *pPoint;
pPoint = (CPunto *)malloc(sizeof(CPunto));
```

([8]) *Vedi* capitolo 9 *Allocazione dinamica della memoria.*

L'uso di malloc, tuttavia, comporta alcuni problemi: l'area di memoria allocata non è inizializzata e non è nemmeno possibile chiamare esplicitamente il costruttore dell'oggetto, in essa contenuto.

11.3.1 L'operatore **new**

Il *C++* mette a disposizione dell'utente l'operatore **new** che consente di creare, in modo dinamico, variabili e oggetti la cui esistenza è limitata al solo periodo di effettiva utilità.

Inoltre, nel caso di un oggetto, l'operatore new ne conosce la classe e chiama automaticamente l'appropriato costruttore per inizializzare l'area di memoria allocata.

Ad esempio:

```
pPoint = new CPunto(21, 35);
```

in questo caso l'operatore new, dopo aver allocato un'adeguata regione di memoria in grado di contenere un oggetto CPunto, chiama il relativo costruttore per inizializzarla come richiesto. Per tale motivo gli oggetti allocati dinamicamente con new beneficiano delle stesse garanzie di quelli creati al momento della compilazione (*compile time*).

L'operatore new ritorna un puntatore al tipo della classe dell'oggetto che si vuole creare. Se la memoria disponibile non è sufficiente, il valore ritornato è **0** (che in *C++* equivale al puntatore NULL).

new può essere utilizzato anche per allocare variabili di tipo base:

```
// Alloca dello spazio per contenere un vettore di 80 caratteri
char *s = new char[80];

// Alloca dello spazio per un long ma non viene inizializzato
long *l = new long;

// Alloca un'area di memoria per un int e la inizializza con 128
int *i = new int(128);
```

11.3.2 L'operatore **delete**

Quando il programma esce dallo *scope* di variabili o di oggetti locali, creati dinamicamente con new, tali entità **non** vengono automaticamente eliminate dalla memoria, come invece accade ai loro puntatori con la stessa visibilità. Poiché la memoria è una risorsa preziosa, per evitare sprechi è bene restituirla al sistema quando non è più necessaria. A tale scopo esiste un operatore che consente di liberare la memoria allocata in precedenza con new: **delete**.

Nel caso di oggetti, diversamente dalla funzione free della *libreria standard* del *C*, prima di restituire la memoria allocata, delete chiama automaticamente il loro distruttore.

L'operatore delete è applicabile solo ai puntatori ritornati da new. Se viene applicato a puntatori diversi, o più volte allo stesso puntatore, molto probabilmente si provocherà la distruzione del meccanismo di gestione della memoria; è esclusiva responsabilità del programmatore assicurare che ciò non avvenga([9]).

Di seguito è mostrato come la memoria occupata dagli oggetti e dalle variabili allocate dinamicamente negli esempi del paragrafo precedente con new possa essere restituita utilizzando l'operatore delete:

```
/*
Viene deallocato l'oggetto puntato da pPoint. Prima della sua
eliminazione viene chiamato il suo distruttore
*/
```

([9]) Per questo motivo è bene assegnare il valore NULL a un puntatore al quale è stato applicato delete. L'applicazione di delete al puntatore NULL non provoca nessun effetto.

```
delete pPoint;

// Viene deallocato lo spazio occupato dal vettore s
delete [] s;

// Viene deallocata l'area di memoria occupata dalla variabile l
delete l;

// Viene deallocata l'area di memoria occupata dalla variabile i
delete i;
```

Si noti la singolarità della sintassi adottata per l'eliminazione di un vettore dinamico: quando si dichiara un vettore si pone la sua dimensione entro le parentesi quadre che seguono il nome del suo tipo; per cancellarlo si segue il procedimento opposto: si indicano delle parentesi quadre vuote prima del nome del suo puntatore([10]).

Il programma `STRINGA.CPP` riassume i concetti appena esposti: si vuole creare una classe che possa rappresentare in modo adeguato una stringa le cui dimensioni non sono specificate al momento della compilazione del programma.

Prog. Stringa.cpp: Creazione di una classe `CStringa`

```
#include <string.h>
#include <iostream>

using namespace std;

// Classe stringa
class CStringa {
public:
  // Costruttori
  CStringa() { lunghezza = 0; s = 0; };
  CStringa(const char *costante);

  // Distruttore
  ~CStringa() { lunghezza = 0; delete [] s; };

  // Ritorna la lunghezza della stringa
  unsigned len() { return lunghezza; };

  // Visualizza la stringa
  void print() { if(s) cout << s; };

private:
  // Dati della stringa
  unsigned lunghezza;
  char     *s;
};

/*
Definizione del secondo costruttore, usato per creare
una stringa ed inizializzarla con una stringa costante.
*/
CStringa::CStringa(const char *costante)
{
  lunghezza = strlen(costante);
```

([10]) Se si applica `delete` al nome di un vettore senza impiegare le parentesi quadre si libera solo la memoria occupata dal primo elemento. Tutti gli altri elementi rimarranno allocati ma inaccessibili.

```
    /*
    Viene richiesta della memoria per contenere la stringa di
    inizializzazione ed il suo carattere terminatore
    */
    if (s = new char[lunghezza + 1])
      strcpy(s, costante);
    else {
      cout << "\nMemoria insufficiente!" << endl;
      exit(1);
    }
}
void main()
{
  // Definizione della variabile Messaggio
  CStringa Messaggio("Questo è il C++\n");
  Messaggio.print();
}
```

L'uscita prodotta dal programma è la seguente:

```
Questo è il C++
```

Lo spazio occupato in memoria dalla variabile `lunghezza` (`unsigned`), sommato a quello occupato da `s` (`char *`), costituisce la quantità di memoria effettivamente occupata dall'oggetto `Messaggio`.

Quando un oggetto di questa classe esce dal proprio spazio di visibilità, `s` e `lunghezza` sono deallocate automaticamente, ma per eliminare lo spazio (eventualmente) allocato con `new` è necessario un distruttore adeguato.

Durante l'assegnazione di un oggetto a un altro oggetto è effettuata una copia membro a membro dei suoi dati. Nel caso di oggetti che contengono puntatori o che allocano la memoria in modo dinamico, questo comportamento di default non è solitamente desiderabile ([11]). *Per risolvere il problema, il C++ consente di ridefinire il comportamento dell'operatore di assegnamento per tutti gli oggetti appartenenti a una determinata classe.*

11.4 *Overloading* degli operatori

Nel capitolo 10 si è affrontato il problema dell'*overloading* delle funzioni evidenziandone le potenzialità offerte. Questo tipo di meccanismo consente di scrivere segmenti di codice sempre più indipendenti dai tipi di dati coinvolti nelle operazioni e di conseguenza sempre più generalizzabili.

In effetti, lavorando con dati di tipo complesso, in particolare con oggetti, risulterebbe molto comodo per il programmatore che questi ultimi, dal punto di vista formale, si comportassero esattamente come le variabili di tipo fondamentale. In tal modo, potrebbe, ad esempio, essere possibile sommare due oggetti, utilizzando l'operatore binario **+**, analogamente a qualunque normale operazione di somma fra numeri.

Il fatto che un operatore possa agire su operandi di tipo differente prende il nome di *overloading* dell'operatore.

La cosa non deve stupire troppo: l'operatore binario **+** viene già usato indifferentemente per sommare, ad esempio, dati di tipo `int`, `double`, `long` ... nonostante le operazioni

([11]) Nonostante i due oggetti, operandi dell'espressione di assegnamento, siano distinti, nell'ipotesi fatta possiedono dei dati membri che puntano alle stesse aree di memoria. Se uno dei due oggetti viene modificato, inevitabilmente, il cambiamento si rifletterà anche sull'altro oggetto.

effettuate internamente siano molto diverse fra loro: infatti, sommare due quantità integrali è differente dall'operazione che consente la somma di numeri reali.

Mentre per le variabili di tipo fondamentale è il compilatore che interpreta correttamente le operazioni da compiere, per i dati complessi come gli oggetti, il *C++* mette a disposizione dell'utente un meccanismo che consente di estendere le funzionalità degli operatori standard. In altre parole, l'utente stesso può ridefinire il loro comportamento affinché riconoscano correttamente come operandi i dati (oggetti) creati dall'utente e implementino su di essi le operazioni corrette.

L'*overloading* degli operatori è permesso per tutti gli operatori tranne quelli elencati in tab. 11.1.

Tab. 11.1 Operatori che non supportano l'*overloading*.

Operatore	*Definizione*
`.`	Operatore punto
`.*`	Operatore puntatore a membro
`::`	Operatore di *scope resolution*
`?:`	Operatore condizionale ternario
`#`	Simbolo del preprocessore
`##`	Simbolo del preprocessore

Sull'*overloading* degli operatori, vi sono inoltre alcune restrizioni che occorre rispettare:

— non è possibile creare un nuovo operatore;
— non è possibile modificare il numero di operandi che un operatore può avere, ovvero un operatore unario non può essere trasformato in binario e viceversa;
— non è possibile modificare la precedenza degli operatori;
— non è possibile modificare l'associatività degli operatori;
— non è possibile modificare il modo in cui gli operatori sono applicati ai tipi fondamentali;
— non sono previsti argomenti di *default*.

Per ridefinire il significato di un operatore, e poterlo così applicare a una determinata classe di oggetti, è necessario definire una *member function* con il nome:

operator*SimboloOperatore*

dove *SimboloOperatore* è l'operatore che si vuole sovrapporre. In questo modo, ogni volta che il compilatore incontrerà *SimboloOperatore* che coinvolge operandi di quella classe chiamerà automaticamente quella specifica *member function*.

Si supponga, ad esempio, di voler implementare l'operatore di assegnamento **=** per la classe `CStringa` definita nel paragrafo precedente.

```
class CStringa {
public:
        .
        .
        .
  // Overloading dell'operatore di assegnamento
  void operator=(const CStringa& sorgente([12]));
private:
  // Dati della stringa
  unsigned lunghezza;
  char     *s;
};
```

([12]) I passaggi di oggetti a funzione sono discussi nel paragrafo 11.7.

```
// Definizione dell'operatore = per la classe CStringa
void CStringa::operator=(const CStringa& sorgente)
{
  /*
  La lunghezza della stringa destinazione diventa uguale quella
  della stringa a destra dell'operatore =
  */
  lunghezza = sorgente.lunghezza;

  /*
  La vecchia stringa destinazione viene deallocata e riallocata
  nuovamente per contenere quella sorgente
  */
  delete [] s;
  if (s = new char[lunghezza + 1])
    /*
    Se esiste abbastanza spazio per allocare la stringa sorgente
    viene assegnata
    */
    strcpy(s, sorgente);
  else {
    cout << "\nMemoria insufficiente!" << endl;
    exit(1);
  }
}
```

Quando il compilatore incontrerà un'espressione come questa:

```
Destinazione = Sorgente;
```

dove `Destinazione` e `Sorgente` sono due oggetti della classe `CStringa`, la interpreterà come una chiamata alla *member function* **operator=**:

```
Destinazione.operator=(Sorgente) ([13]);
```

L'assegnamento è realizzato liberando prima la memoria occupata dalla stringa di destinazione([14]) e quindi riallocandola in modo che possa contenere la stringa sorgente; successivamente, `strcpy` effettua la copia.

Si noti che, nonostante l'operatore di assegnamento sia binario, la funzione `operator=` ha solo un argomento: l'operando di destra. Quello di sinistra è implicito perché `operator=` non è altro che un metodo applicato a quello specifico oggetto.

Sebbene l'*overloading* dell'operatore **=**, definito per questa classe di oggetti, sembri funzionare, in alcuni casi particolari può essere causa di errori. Si consideri, ad esempio, la situazione in cui una stringa è assegnata a se stessa:

```
Destinazione = Destinazione;
```

Per i dati di tipo fondamentale tale operazione è lecita e non modifica il valore dell'operando a sinistra del segno =. Diventa quindi naturale pretendere che anche il nuovo operatore = della classe `CStringa` abbia un comportamento analogo. Nel caso preso in esame, invece, non è così: la *member function* `operator=` libera il *buffer* puntato da `s` dell'operando

([13]) Benché queste funzioni siano implicitamente chiamate dal compilatore quando nel codice sorgente viene incontrato l'operatore corrispondente, possono esserlo anche esplicitamente, come una qualsiasi altra *member function*.

([14]) Si osservi che se il valore del membro `s` della stringa destinazione è `0`, `delete` non ha nessun effetto.

di sinistra (che coincide con quello dell'operando di destra) e ne alloca uno nuovo; la successiva chiamata a `strcpy` tenta poi di copiare il *buffer* appena allocato su se stesso, con risultati che non è possibile prevedere.

Per evitare il verificarsi di queste situazioni, è necessario che il *buffer* di destinazione non venga alterato se coincide con quello della stringa che si intende assegnare.

A tale scopo diventa indispensabile l'uso di un'entità del C++, il puntatore **this**.

11.5 Il puntatore `this`

this è un puntatore costante il cui valore punta all'oggetto a cui si sta riferendo una *member function* di una classe.

Si consideri, ad esempio, la *member function* `print` della classe `CStringa`, definita come segue:

```
void CStringa::print()
{
  cout << s;
}
```

e il seguente segmento di programma:

```
...
CStringa Messaggio("Il puntatore this");
...
Messaggio.print();
...
```

Al momento della valutazione dell'espressione:

```
Messaggio.print()
```

il puntatore `this` è inizializzato dal compilatore con l'indirizzo di memoria dell'oggetto `Messaggio` e **implicitamente** passato alla *member function* `print`.

Questo meccanismo è il modo che il C++ adotta per comunicare a una *member function* su quale oggetto di quella classe deve operare.

Dal momento che `this` è accessibile alle *member function* di una classe, all'interno di queste è possibile utilizzarlo anche in modo esplicito; ad esempio, sempre facendo riferimento a `print`, le tre espressioni che seguono sono equivalenti:

```
cout << s
cout << this->s
cout << (*this).s
```

Questo puntatore consente a una *member function* di verificare se l'oggetto che sta trattando coincide o meno con l'argomento che le è stato passato. Ciò permette di risolvere con semplicità problemi come, ad esempio, quello della *member function* `operator=` discusso nel paragrafo precedente:

```
void CStringa::operator=(const CStringa& sorgente)
{
  // sorgente equivale a Destinazione (this)?
  if (&sorgente == this)
    return;
  ...
}
```

Poiché `this` contiene l'indirizzo dell'operando di sinistra è possibile verificare se coincide con quello dell'operando di destra (`sorgente`).

`this` *può essere usato anche come valore di ritorno di una member function.*

Come esempio, si consideri ancora l'*overloading* dell'operatore di assegnamento per la classe `CStringa`: sia in *C* sia in *C++*, l'assegnamento è considerato un'espressione e come tale produce un valore, quello dell'operando che si sta assegnando. È per questo motivo, e per l'associatività (da destra a sinistra) di questo operatore, che è possibile realizzare degli assegnamenti multipli del tipo:

```
Frase = Destinazione = Sorgente;
```

dove `Frase` è un altro oggetto della classe `CStringa`.

Se si desidera dotare di questa proprietà anche la *member function* `operator=` della classe `CStringa`, è necessario che questa ritorni il risultato dell'assegnamento in modo che possa essere nuovamente assegnato all'eventuale operando di sinistra. Questa modifica è possibile impiegando il puntatore `this` nel seguente modo:

```
CStringa &CStringa::operator=(const CStringa& sorgente)
{
  // sorgente equivale a Destinazione (this)?
  if (&sorgente == this)
    return *this;
  ...
  // Ritorna un reference all'oggetto
  return *this;
}
```

Si noti che è più efficiente ritornare un *reference* all'oggetto che contiene il risultato dell'assegnamento piuttosto che l'oggetto stesso ([15]).

Con questa nuova versione della *member function* `operator=` è adesso possibile eseguire assegnamenti multipli.

A volte, quando il significato di un operatore viene esteso da una funzione `operator`, non sempre è possibile implementare tutte le sue proprietà formali.

Si supponga, ad esempio, di rappresentare la classe dei numeri complessi e, successivamente, di estendere il significato dell'operatore binario **+** in modo che possa effettuare anche la somma di due numeri appartenenti a quella classe.

La soluzione proposta implementa una classe, `CComplesso`, nella quale un numero viene rappresentato nella forma $a + jb$, dove a rappresenta la parte reale e b quella immaginaria:

```
class CComplesso {
public:
  // Costruttore
  CComplesso(double reale = 0, double immaginaria = 0)
  {
    a = reale; b = immaginaria;
  };
  // Overloading dell'operatore +
  CComplesso operator+(const CComplesso& x);
private:
  double a,     // Parte reale
         b;     // Parte immaginaria
};
```

([15]) Per ulteriori dettagli *vedi* il paragrafo 10.16 *Funzioni che ritornano un reference.*

Per estendere il significato di un operatore standard, all'interno della classe è stata definita una *member function* con il nome `operator` seguito dal simbolo **+**. Ogni volta che si presenta una somma che coinvolge numeri complessi sarà automaticamente chiamata questa funzione.

Una possibile definizione di `operator+` è la seguente:

```
CComplesso CComplesso::operator+(const CComplesso& x)
{
  return CComplesso([16])(a + x.a, b + x.b);
}
```

Una somma del tipo:

```
z = x + y;
```

dove `x`, `y` e `z` sono numeri complessi, viene interpretata dal compilatore come segue:

```
z = x.operator+(y);
```

viene prima chiamata la *member function* `operator+` dell'oggetto `x` passandogli come argomento un *reference* all'oggetto `y` dopodiché, il suo valore di ritorno (ancora un numero complesso) è assegnato all'oggetto `z`([17]).

Con questo «nuovo» operatore è possibile sommare anche un valore costante alla parte reale del numero complesso, come nell'espressione seguente:

```
z = x + 1.9;
```

La costante `1,9` viene infatti considerata dalla *member function* `operator+` come la parte reale di un numero complesso con parte immaginaria `0` (valore di *default* del secondo argomento del costruttore).

Non è invece possibile eseguire somme del tipo:

```
z = 1.9 + x;
```

perché non sono previste dal compilatore somme di valori a virgola mobile con oggetti di classe `CComplesso`([18]). Ciò impedisce la completa implementazione della **proprietà commutativa** dell'operatore «somma».

Per risolvere questo problema e, in generale, permettere che il primo operando di un'espressione possa essere diverso dall'oggetto che richiede l'overloading di un operatore, è necessario ricorrere alle **funzioni `friend`**.

11.6 Funzioni `friend`

L'accesso ai membri privati di una classe è consentito solo alle *member function* di quella classe; questo sistema di protezione ha il vantaggio di permettere la modifica della sezione di implementazione di una classe senza dovere intervenire sul programma che la usa.

Esistono tuttavia delle situazioni in cui è necessario, o si rivela più efficiente, operare sui membri di una classe mediante funzioni esterne anziché attraverso le sue *member function*. A tale scopo, il *C++* dispone di un ulteriore meccanismo mediante il quale una classe autorizza una funzione **esterna** ad accedere anche ai propri dati membri privati.

([16]) L'oggetto da ritornare è definito, in modo temporaneo, con una chiamata esplicita a questo costruttore.

([17]) Si osservi, che diversamente dagli oggetti di classe `CStringa`, visti nell'esempio precedente, per questa classe di oggetti non è necessario l'*overloading* dell'operatore di assegnamento standard.

([18]) Si ricorda che l'operatore somma è associativo da sinistra verso destra; in questo caso, quindi, è applicato alla costante reale (di tipo fondamentale) che non possiede metodi per aggiungere numeri complessi.

Se, all'interno di una classe, la parola chiave **friend** *precede la dichiarazione di una funzione esterna, quella funzione può liberamente accedere a tutti i dati membri di quella classe.*

Per esempio, nel caso del problema appena posto, è possibile estendere ulteriormente il significato dell'operatore «somma di complessi con reali» impiegando una funzione che non è membro di quella classe ma che gode degli stessi privilegi:

```
class CComplesso {
public:
  // Overloading dell'operatore +
  CComplesso operator+(const CComplesso &x);

  // Questa funzione non è un membro della classe ma ne è "amica"
  friend CComplesso operator+(double Reale, const CComplesso& x);
  ...
};
```

La dichiarazione `friend`([19]) di `operator+` indica al compilatore che quella funzione **non è un membro della classe**, ma ne è «amica». Ciò sta a significare che pur non possedendo la stessa visibilità può accedere ai suoi membri come se fossero tutti pubblici.

La definizione di una funzione `friend` è esterna alla classe e analoga, nella forma, a quella di una funzione generica: non vanno indicati né la parola chiave `friend`, né l'operatore di *class scope* **::** (necessario, invece, nel caso di *member function*):

```
// Definizione di una funzione friend della classe CComplesso
CComplesso operator+(double Reale, const CComplesso& x)
{
  return CComplesso([20])(x.a + Reale, x.b);
}
```

Come si può osservare, diversamente dalla *member function* `operator+` definita nell'esempio precedente, questa funzione richiede entrambi gli operandi della somma([21]). Mediante il meccanismo di *overloading* delle funzioni, se il primo operando della somma è un numero complesso, e il secondo un numero complesso o reale, verrà chiamata la *member function* `operator+`, in caso contrario la funzione `friend` appena definita([22]).

Un altro esempio in cui risulta comodo ricorrere alle funzioni `friend` è il seguente. Per visualizzare il valore di oggetti di classe `CComplesso` si potrebbe implementare un'apposita *member function*, ma l'uso dell'operatore di inserzione **<<**, come per i tipi fondamentali, rappresenta una soluzione sicuramente più «elegante». Si può ottenere effettuando l'*overloading* di quest'operatore con una funzione `friend`:

```
class CComplesso {
public:
  ...
  // Funzione friend della classe CComplesso
  friend ostream([23])& operator<<(ostream& os, CComplesso& x);
  ...
};
```

([19]) Una dichiarazione `friend` non è influenzata dagli specificatori di accesso `public` e `private` e può, pertanto, essere posta ovunque all'interno della dichiarazione di una classe.

([20]) La funzione `friend` ha accesso al costruttore della classe `CComplesso` come una qualsiasi altra *member function*.

([21]) Diversamente dalle *member function* della classe, alle funzioni `friend`, non è passato il puntatore `this`.

([22]) Non è possibile definire funzioni `friend`, che effettuano l'*overloading* degli operatori, che non abbiano almeno un oggetto come parametro. La definizione `friend` non è possibile nemmeno per funzioni che effettuano l'*overloading* dell'operatore di assegnamento che può essere realizzato solo da *member function*.

([23]) Per ulteriori approfondimenti sulla classe `ostream` *vedi* l'Approfondimento 12.1 *Iostream class library*.

```
// Overloading dell'operatore <<
ostream& operator<<(ostream& os, CComplesso& x)
{
  if (x.a)
    os << x.a;
  if(x.b < 0) {
    os << "-j";
    os << -x.b;
  }
  else if(x.b > 0) {
    if(x.a)
      os << '+';
    os << 'j' << x.b;
  }
  return os;
}
```

Adesso, se `z` è un oggetto di classe `CComplesso`, è possibile scrivere:

```
cout << z << endl;
```

Nel meccanismo di «amicizia» è sempre la classe a consentire l'accesso ai suoi dati membri e mai il contrario. Se così non fosse, infatti, chiunque potrebbe dichiararsi «amico» di una classe ed eludere il suo sistema di sicurezza.

In certi casi una funzione può risultare «amica» di due o più classi e questo rappresenta un metodo comodo per consentire a una funzione di accedere ai dati membri di classi indipendenti fra loro.

11.7 Classi `friend`

Analogamente alle funzioni, anche una classe può essere «amica» di un'altra classe. Una **classe `friend`** è una classe le cui *member function* sono funzioni `friend` di un'altra classe, possono avere, cioè, accesso a tutti i suoi dati membri:

```
class MiaClasse {
public:
  friend class ClasseAmica; // ClasseAmica è una classe friend
private:
  int datoPrivato;
};

// Classe friend di MiaClasse
class ClasseAmica {
public:
  void print(MiaClasse mc);
private:
  char Nome[40];
  char Data[11];
};

void ClasseAmica::print(MiaClasse mc)
{
  cout << mc.datoPrivato << endl; //Può accedere ai dati privati
}
```

Una classe `friend` ***non è un membro*** *della classe in cui è dichiarata e non è soggetta alle restrizioni di accesso tipiche degli altri membri.*

La relazione di amicizia ha le seguenti restrizioni:

— **non è reciproca**: a meno che non sia esplicitamente specificata. Nell'esempio, le *member function* di `MiaClasse` non possono accedere ai dati membri privati `Nome` e `Data` di `ClasseAmica`;
— **non è transitiva**; le eventuali classi amiche di `ClasseAmica` non possono accedere ai dati membri `private` di `MiaClasse`;
— **non è ereditabile** ([24]); le classi derivate da `ClasseAmica` non possono accedere ai dati membri `private` di `MiaClasse`.

Negli esempi che riguardano gli oggetti della classe `CComplesso` *sono state utilizzate funzioni che hanno dei parametri e valori di ritorno di tipo classe.*

11.8 Oggetti come argomenti e valori di ritorno di una funzione

Quando un oggetto è **inizializzato** con il valore di un altro oggetto, oppure quando è **passato** come argomento a una funzione, il compilatore ne effettua una copia, membro a membro, nel corrispondente parametro formale.

Analogamente, se è previsto che una funzione **ritorni** un oggetto, al momento dell'esecuzione dell'istruzione `return`, viene creato ([25]), e opportunamente inizializzato, un **oggetto temporaneo** che può essere usato come operando di destra in un'eventuale operazione di assegnamento. Quest'oggetto sarà distrutto al termine della valutazione dell'espressione che contiene la chiamata alla funzione.

Come nel caso dell'operatore di assegnamento di *default*, queste **copie** possono essere fonte di problemi se coinvolgono oggetti che contengono puntatori o che allocano la memoria in modo dinamico in quanto si potrebbero avere dei problemi dovuti alla condivisione delle stesse aree di memoria ([26]).

Per evitare questo genere di problemi, il *C++* consente di definire all'interno di una classe un costruttore particolare, chiamato **costruttore copia**, che viene automaticamente chiamato in questi casi.

11.8.1 Il costruttore copia

Un **costruttore copia** è un costruttore che possiede come **unico parametro** un ***reference*** a un oggetto della stessa classe. Se presente, il compilatore lo chiamerà implicitamente ogni volta che deve inizializzare un oggetto con i dati di un altro, altrimenti, per *default*, l'inizializzazione avverrà membro a membro.

A differenza dell'operatore di assegnamento, che modifica il contenuto dell'oggetto alla sua sinistra, un «costruttore copia» duplica l'oggetto a cui si riferisce il suo argomento.

Nell'esempio che segue, alla classe stringa è stato aggiunto il «costruttore copia»:

```
class CStringa {
public:
  // Costruttori
  CStringa() { s = 0; };
  CStringa(const char *costante);
```

([24]) Per il concetto di ereditarietà *vedi* il capitolo 12 *Ereditarietà e polimorfismo*.
([25]) Nello stesso spazio di visibilità della funzione chiamante.
([26]) *Vedi* gli oggetti della classe `CStringa` degli esempi precedenti.

```
    CStringa(const CStringa& sorgente); // Costruttore copia
    .
    .
    .
};
// Definizione del costruttore copia
CStringa::CStringa(const CStringa& sorgente)
{
  lunghezza = sorgente.lunghezza;
  if (s = new char[lunghezza + 1])
    //Se esiste abbastanza spazio l'oggetto viene inizializzato
    strcpy(s, sorgente);
  else {
    cout << "\nMemoria insufficiente!" << endl;
    exit(1);
}
```

Si osservi che, essendo un costruttore della classe, non possiede un valore di ritorno; inoltre, non è necessario premunirsi contro ipotetici autoassegnamenti (come nel caso dell'*overloading* dell'operatore di assegnamento) perché non sono assolutamente possibili ([27]).

Osservazioni

Nel caso di oggetti che hanno come dati membri dei puntatori o che allocano dinamicamente della memoria è consigliato implementare sempre sia un costruttore copia, sia una funzione che effettui l'overloading dell'operatore di assegnamento.

11.8.2 Passaggio e ritorno di *reference* a oggetti

Anziché passare un oggetto a una funzione si può passare un *reference*, il risultato è lo stesso, ma il meccanismo è più efficiente. Questo sistema consente di risparmiare il tempo necessario al programma per effettuare una copia membro a membro dell'oggetto (o di eseguire la chiamata del proprio costruttore copia, nel caso sia presente). Se, come nei passaggi per valore, non si desidera modificare l'oggetto, è sufficiente anteporre alla dichiarazione del parametro corrispondente la parola chiave `const`.

Analogamente, anziché tornare un oggetto da una funzione, si potrebbe ritornarne un *reference*. In questo modo vengono, di fatto, eliminate tutte le operazioni occorrenti alla creazione (e successiva distruzione) dell'oggetto temporaneo che rappresenta il ritorno della funzione.

Se viene tornato un *reference*, come per le funzioni che ritornano puntatori, si presti attenzione al fatto che si riferisca a locazioni di memoria valide ([28]).

11.9 Membri `static`

A volte è necessario che tutti gli oggetti appartenenti a una stessa classe condividano delle informazioni. Per farlo in modo efficiente e sicuro, il *C++* consente di dichiarare dei *data member* come **`static`**.

([27]) È il linguaggio stesso che non consente di inizializzare un oggetto con un altro che ancora non esiste, né di ridefinire un oggetto già esistente.

([28]) Non è sensato ritornare *reference* o puntatori a variabili locali, a meno che queste non rappresentino aree di memoria allocate in modo dinamico oppure siano a loro volta *reference* o puntatori a variabili definite all'esterno della funzione.

Diversamente dagli altri *data member*, replicati a ogni istanza di quella classe, un dato membro preceduto da questa parola chiave è allocato solo una volta, ma possiede, comunque, la stessa visibilità e i medesimi diritti di accesso degli altri *data member*.

Nell'esempio che segue, la classe `CConto` ha tre dati membri di cui uno, `Interesse`, è dichiarato `static`:

```
class CConto {
public:
  ...
  void AccreditaInteressi() { Saldo += Interesse * Saldo; };

private:
  unsigned NumeroConto;
  double Saldo;
  static double Interesse;
};
```

Qualunque sia il numero di istanze di `CConto`, esiste solo una copia del membro `Interesse`, condivisa da tutti gli oggetti di quella classe.

Un dato membro `static` può essere inizializzato solo una volta; l'operazione non può, quindi, essere fatta da un costruttore perché altrimenti si ripeterebbe a ogni istanza di quella classe.

Il *C++* impone che l'inizializzazione avvenga in un **livello esterno**, come per una variabile globale.

Ad esempio:

```
double CConto::Interesse = .05; // Definizione di Interesse al 5%
```

Nella fase d'inizializzazione, lo specificatore di accesso (`public` o `private`) non ha alcun effetto su un membro `static`, ma solo in questa fase, perché subito dopo condizionerà le modalità di accesso secondo quanto dichiarato.

Anche le *member function* possono essere dichiarate `static`. In questo caso potranno accedere **esclusivamente** a dati membri `static`.

Ad esempio:

```
class CConto {
public:
  ...
  void AccreditaInteressi() { Saldo += Interesse * Saldo; };

  static void ImpostaInteresse(double NuovoInteresse)
  { Interesse = NuovoInteresse; };

private:
  unsigned NumeroConto;
  double Saldo;
  static double Interesse;
};
```

Diversamente dalle altre *member function*, che agiscono su un oggetto specifico, `ImpostaInteresse` è un metodo generico di quella classe e non è applicato a nessuna sua istanza in particolare. Per questo motivo, le *member function* `static` non possiedono il puntatore `this` e non possono così accedere ad altri dati membri o richiamare *member function* che non siano `static`([29]).

([29]) I costruttori e distruttori di una classe non possono essere dichiarati `static`.

Se la definizione di una *member function* `static` avviene al di fuori della classe in cui è dichiarata, **non** deve essere preceduta dalla *keyword* `static`.

Per non peggiorare la leggibilità di un programma in cui sono inseriti dati membri `static`, è bene escludere a priori la possibilità che vengano interpretati erroneamente come dati membri «non-*static*». Ad esempio, per l'oggetto `Dipendenti` della classe `CConto`, definita precedentemente, la seguente sintassi:

```
Dipendenti.ImpostaInteresse(0.075)
```

benché corretta, sarebbe ambigua perché suggerisce l'idea che venga impostato il tasso di interesse del solo oggetto `Dipendenti`, mentre invece viene cambiato contemporaneamente per tutti gli oggetti appartenenti a quella classe. Pertanto, facendo riferimento a membri statici di una classe è sempre consigliabile utilizzare l'operatore di *class scope*:

```
CConto::ImpostaInteresse(0.075)
```

In questo modo chi legge non può avere dubbi: il tasso d'interesse impostato è valido per tutti gli oggetti della classe `CConto`.

11.10 *Member function* `const`

Se si dichiara un oggetto come `const`, non è possibile modificare i suoi *data member* né richiamare *member function* che non siano dichiarate anch'esse `const`.

Qualificare una *member function* come **`const`** ([30]) implica un cambiamento nella funzione che diventa così «*read-only*». Questa funzione non può modificare l'oggetto a cui appartiene, nemmeno indirettamente (chiamando, ad esempio, un'altra *member function* non `const`) ([31]).

Per dichiarare una funzione di questo tipo è necessario porre la *keyword* `const` dopo la parentesi chiusa della lista degli argomenti, sia nella dichiarazione, sia nella definizione della funzione:

```
class CData {
public:
  CData( int g, int m, int a );
  int leggiGiorno() const;        // Funzione "read-only"
  void impostaGiorno( int g );    /*Funzione che modifica:
                                    non può essere costante */
private:
  int giorno;
  ...
};

// Definizione della funzione costante
int CData::leggiGiorno() const
{
  return giorno;          // Non modifica niente!
}

void CData::impostaGiorno( int g )
{
  giorno = g;             // Modifica un dato membro
}
```

([30]) I costruttori e i distruttori di una classe non possono essere dichiarati `const`.

([31]) L'unica eccezione a questa regola è il caso di dati membri (non `const` né `static`) preceduti dalla *keyword* **`mutable`** i quali possono essere modificati anche da *member function* `const`.

Nel caso si desideri definire dei metodi validi sia per oggetti costanti e no, è possibile effettuare l'*overloading* di *member function* `const` con *member function* non `const` in modo da consentire al compilatore di chiamare una delle due versioni, a seconda di come è dichiarato l'oggetto.

11.11 Vettori di oggetti

In *C++* è possibile dichiarare anche vettori i cui elementi sono oggetti:

```
CStringa Elenco[30];
```

Ogni volta che è dichiarato un vettore di oggetti, è chiamato il costruttore per ogni suo elemento.

Per *default* è chiamato il costruttore senza parametri, ma è possibile chiamare anche un costruttore diverso. Nell'esempio che segue, i primi tre elementi sono inizializzati con delle stringhe costanti, mentre l'inizializzazione di tutti gli altri è lasciata al costruttore senza argomenti:

```
CStringa Elenco[30] = {
  "Aldobrandi Federico",
  "Berti Andrea",
  "Bussone Mario"
};
```

Se la dimensione del vettore non è indicata esplicitamente, è possibile ottenere un aumento di flessibilità del programma e un ragguardevole risparmio dello spazio occupato dai suoi dati:

```
CStringa Elenco[] = {
  "Aldobrandi Federico",
  "Berti Andrea",
  "Bussone Mario"
};
```

In questo caso saranno creati e inizializzati solo i tre oggetti elencati tra le parentesi graffe e non tutti e trenta, come nell'esempio precedente.

La dimensione effettiva del vettore può essere facilmente ricavata mediante l'espressione:

```
Dimensione = sizeof Elenco / sizeof(CStringa)
```

Così facendo, se si aggiungono nuovi oggetti nella lista d'inizializzazione del vettore, il programma dovrà essere modificato solo in quel punto.

A volte, per motivi di efficienza, potrebbe essere utile trattare dei vettori di puntatori a oggetti anziché vettori contenenti gli oggetti stessi.

Nella dichiarazione che segue, `pElenco` è dichiarato come un vettore di puntatori a oggetti di classe `CStringa`:

```
CStringa *pElenco[] = {
  new CStringa("Aldobrandi Federico"),
  new CStringa("Berti Andrea"),
  new CStringa("Bussone Mario")
};
```

Nella lista d'inizializzazione, l'operatore `new` crea dinamicamente tre oggetti i cui indirizzi sono memorizzati nei primi tre elementi del vettore.

È possibile risalire alla dimensione effettiva del vettore attraverso un'espressione analoga alla precedente:

```
Dimensione = sizeof pElenco / sizeof(CStringa *)
```

Alcune situazioni richiedono che la dimensione di un vettore di oggetti debba essere stabilita durante il periodo di tempo di esecuzione del programma (*runtime*).

Il *C++* consente di dichiarare dei vettori dinamici mediante l'operatore `new`:

```
CStringa *Elenco = new CStringa[Dim];
```

In questo caso, però, gli oggetti allocati saranno inizializzati dal costruttore senza parametri.

I vettori di oggetti allocati dinamicamente possono venire deallocati con l'operatore `delete`:

```
delete [] (32) Elenco;
```

Al momento della valutazione di questa espressione, verrà chiamato il distruttore (se esiste) per ogni oggetto contenuto nel vettore.

11.12 Tipi di classi

In *C++* anche i tipi `struct` e `union` vengono considerati al pari delle classi e, come tali, possono contenere *member function*, costruttori e distruttori (33).

A differenza delle classi, dove lo specificatore di accesso è `private` per *default*, negli altri due è `public`.

Tuttavia, per non complicare troppo la leggibilità di un programma, si consiglia, per quanto possibile, di utilizzare strutture e unioni come nel linguaggio *C*, riservando ai soli tipi `class` la dichiarazione di oggetti.

11.13 *Template*

Lo standard del *C++* è in continua evoluzione e ciò implica dei mutamenti nel linguaggio con l'obiettivo di migliorarne efficienza e competitività. Tra gli argomenti che meritano molta attenzione (e sui quali si prevede uno sviluppo più ampio) si ricordano i **tipi parametrizzati**, meglio noti con il nome di ***template***.

11.13.1 I tipi parametrizzati

Uno dei concetti principali dei nuovi linguaggi di programmazione, e in generale delle tecniche attuali per produrre *software*, è quello della riusabilità del codice. L'obiettivo è produrre moduli, funzioni, classi e quant'altro possa essere riutilizzato con poche modifiche. Un modo perché ciò possa avvenire consiste nell'individuare delle proprietà o delle funzionalità facilmente riusabili in varie occasioni. Il concetto non è certo nuovo. La filosofia su cui si basa la «funzione» è proprio quella di consentire, mediante il meccanismo del passaggio dei parametri, la ripetizione delle stesse operazioni su dati di valore diverso. In tal modo il codice della funzione è scritto solo una volta e riutilizzato in seguito

(32) Si presti attenzione alla coppia di parentesi quadre dopo la *keyword* `delete`. Si ricorda che se vengono omesse nel caso di vettori, viene deallocato solo il primo elemento mentre gli altri rimangono allocati, ma non più accessibili.

(33) Le `union`, diversamente da `class` e `struct` non possono contenere membri `static`.

illimitatamente. La tecnica dell'*overloading* delle funzioni fa un nuovo passo in avanti consentendo di variare anche il tipo dei dati, ciò nonostante richiede la riscrittura del codice e una serie di copie pari al numero dei tipi trattati.

Perché non creare un nuovo sistema in grado di aggregare le due filosofie in un'unica soluzione, unendo i vantaggi dell'una e dell'altra?

Nasce così l'idea dei **tipi parametrizzati** o ***template***: un codice particolare dove i parametri possono essere sia valori, sia nomi di tipo. Il concetto è semplice, rimane solo da capire come si realizza a livello sintattico.

La forma dichiarativa di un *template* è la seguente:

template < [[*typelist*]] [[, [[*arglist*]]]] > *declaration*

La parola chiave `template` specifica una **funzione** o una **classe** parametrizzata.

I parametri sono inseriti tra una coppia di parentesi angolari **<>** e separati fra loro da virgole. I parametri possono essere un elenco di tipi, nella forma **class** *identifier* o **typename** *identifier*, o di valori che saranno impiegati nel corpo del *template*. Il campo riguardante *declaration* è una dichiarazione di **funzione** o di **classe**, come si vedrà negli esempi seguenti.

L'istanza di una classe *template* è analoga a quella di una classe normale, con la differenza che contiene la lista degli argomenti tra parentesi angolari. Le chiamate a funzioni *template*, invece, sono uguali a quelle delle funzioni tradizionali.

11.13.2 Funzioni *template*

Il primo esempio, `TEMPLAT1.CPP`, è l'implementazione di una **funzione *template*** che calcola il massimo fra due valori([34]) omogenei di qualsiasi tipo fondamentale. Un modo molto comodo di definire un insieme di operazioni parametrizzate.

La funzione *template* `Massimo` calcola il massimo fra due valori, senza dover scrivere funzioni specializzate, come nel caso dell'*overloading*. L'identificatore `QUALUNQUE` costituisce il parametro: in ogni chiamata alla funzione ogni occorrenza di `QUALUNQUE` sarà sostituita con il tipo del relativo valore passato come argomento.

Poiché **il tipo** degli argomenti passati è conosciuto al momento della compilazione, a differenza dalle macro, il compilatore può effettuare un controllo sulla loro correttezza.

Prog. Templat1.cpp: *Template* di funzioni

```
#include <iostream>

using namespace std;

template <class QUALUNQUE> QUALUNQUE Massimo(QUALUNQUE a, QUALUNQUE b)
{
  return (a > b) ? a : b;
}

void main()
{
  int     x = 21, y = -9;
  double  Real = 2.71;
  char    Ch = 'B';
```

([34]) Il lettore potrebbe domandarsi: perché ripresentare algoritmi già ampiamente discussi? Per osservare l'evoluzione del linguaggio anche attraverso il confronto di strategie e sintassi adottate per implementare algoritmi semplici, ma comuni. Il fatto di non dover analizzare il problema dal punto vista algoritmico, poiché ormai ben noto, permette di concentrarsi sullo sviluppo di soluzioni dal punto di vista sintattico.

```
  cout << Massimo(x, y) << endl;
  cout << Massimo(-60, y) << endl;
  cout << Massimo(Real, double(y)) << endl;
  cout << Massimo(Real, double(x)) << endl;
  cout << Massimo(Ch, 'X') << endl;
}
```

L'uscita prodotta è la seguente:

```
21
-9
2.71
21
X
```

Il *template* può assumere anche una forma differente utilizzando l'identificatore di tipo **typename**; senza apportare modifiche al `main`, si ottengono gli stessi risultati:

```
template <typename T> T Massimo(T a, T b)
{
  return (a > b) ? a : b;
}
```

La *keyword* `typename`, inserita prima dell'identificatore `T`, indica al compilatore che `T` è il nome di un tipo.

Osservazioni

Nei template di funzioni si consiglia di utilizzare `typename`([35]) *anziché* `class`, *quando il parametro passato come argomento è il nome di un tipo. Le direttive dello standard stabiliscono che in un template, un nome non è considerato un tipo a meno che non sia stato dichiarato tale nel contesto in cui si trova o specificato mediante la keyword* `typename`.

11.13.3 Classi *template*

Utilizzando una forma dichiarativa analoga alla precedente, è possibile definire una classe parametrizzata. La sintassi è simile alla normale dichiarazione di una classe, come si può vedere dall'esempio:

```
template <class T, int i> class Generica {
public:
  Generica();  // Costruttore
  ~Generica(); // Distruttore
  void setMember( T a, int b );  // Inizializza un elemento
  T getMember(int b);  // Preleva il valore di un elemento
private:
  T     Vett[i];
  int   dimVett;
};
```

Nell'esempio, la classe *template* `Generica` usa due parametri: il primo, `class T`, è un tipo parametrizzato mentre il secondo è di tipo `int`. Al momento d'istanziare un oggetto

([35]) In molti programmi, elaborati prima della definizione dello standard, la parola `typename` (da non confondere con la *keyword* `typedef`) non compare mai per il semplice fatto che non esisteva ancora.

di questa classe, al parametro T deve essere passato il nome di un tipo([36]). Osservando il corpo della classe è facile individuare che il parametro int i, definito in seguito al momento della compilazione, rappresenta la dimensione di un vettore di tipo T.

L'esempio proposto TEMPLAT2.CPP è molto elementare, ma dovrebbe fornire un'idea della struttura e del funzionamento dei *template* applicati alle classi. Si vogliono creare due vettori, uno di 10 elementi di tipo char e l'altro di 5 elementi di tipo double, e assegnare due valori noti a due elementi in particolare, per poi visualizzarne il valore sul video.

Prog. Templat2.cpp: *Template* di classe

```
#include <iostream>

using namespace std;

//classe template
template <class T, int i> class Generica {
public:
  Generica();                    // Costruttore
  ~Generica();                   // Distruttore
  void setMember( T a, int b );// Inizializza un elemento
  T getMember(int b);  // Preleva il valore di un elemento

private:
  T     Vett[i];
  int   dimVett;
};

// Member function costruttore
template <class T, int i> Generica<T, i>::Generica<T, i>()
{ dimVett = i; };

// Member function distruttore
template <class T, int i> Generica<T, i>::~Generica<T, i>() {};

// Metodo setMember
template <class T, int i>
void Generica<T, i>::setMember( T a, int b ) { Vett[b] = a; };

// Metodo getMember
template <class T, int i> T Generica<T, i>::getMember(int b)
{ return Vett[b]; };

void main()
{
  // Istanza della classe Generica per tipi char
  Generica<char, 10> s;

  // Istanza della classe Generica per tipi double
  Generica<double, 5> d;

  s.setMember('x', 6);      // Assegna il valore 'x' al 7° elemento
  d.setMember(1.4142, 3); // Assegna 1,4241 al 4° elemento

  cout << "Lettera = " << s.getMember(6) << endl;
  cout << "Numero = " d.getMember(3) << endl;
}
```

([36]) T può essere il nome di un tipo qualsiasi, base o definito dal programmatore.

Il programma visualizzerà sul video le scritte

```
Lettera = x
Numero = 1.4142
```

Nell'esempio, sia il distruttore sia il costruttore sono stati inseriti solo allo scopo d'illustrarne la sintassi ([37]).

11.13.4 Considerazioni

È bene ricordare alcuni punti fondamentali nella costruzione di un *template*:

— la dichiarazione di *template* non genera automaticamente un codice, ma specifica semplicemente una famiglia di classi o funzioni;
— la dichiarazione di un *template* ha una visibilità a livello di spazio globale;
— è molto importante rammentare **i vincoli impliciti** del linguaggio: nell'esempio della funzione *template* `Massimo` del paragrafo precedente si effettua il confronto fra due identificatori mediante l'operatore **>**. Ciò presuppone che il tipo assegnato al momento di creare un'istanza del *template* supporti l'operatore **>** altrimenti il confronto porterebbe a un errore in fase di compilazione;
— non è possibile separare la sezione d'implementazione di un *template* dalle sue istanze;
— i tipi con nessun *linkage* non possono essere usati come argomenti di *template*.

([37]) Per un esempio più articolato e approfondito si rimanda all'esercizio 2 di questo capitolo.