C++, una panoramica sul linguaggio

Indice

Introduzione

Il C++ è un linguaggio di programmazione "all purpose", ovvero adatto alla realizzazione di qualsiasi tipo di applicazione da quelle real time a quelle che operano su basi di dati, da applicazioni per utenti finali a sistemi operativi. Il fatto che sia un linguaggio "all purpose" non vuol comunque dire che qualsiasi cosa va fatta in C++, esistono moltissimi linguaggi di programmazione alcuni dei quali altamente specializzati per compiti precisi e che quindi possono essere in molti casi una scelta migliore perché consentono di ottenere un rapporto "costi di produzione/prestazioni" migliore per motivi che saranno chiari tra poche righe.

Negli ultimi anni il C++ ha ottenuto un notevole successo per diversi motivi:

- Conserva una compatibilità quasi assoluta (alcune cose sono diverse) con il suo più diretto antenato, il C, da cui eredita la sintassi e la semantica per tutti i costrutti comuni, oltre alla notevole flessibilità e potenza;

- Permette di realizzare qualsiasi cosa fattibile in C senza alcun overhead addizionale;

- Estende le caratteristiche del C fornendo i meccanismi per l'astrazione dei dati e la programmazione orientata agli oggetti, introducendo costrutti innovativi (modelli, Run Time Type Information...) e fornendo uno dei migliori sistemi di tipi mai realizzato (cosa che manca nel C);

- Possibilità di portare facilmente le applicazioni verso altri sistemi;

Comunque il C++ presenta anche degli aspetti negativi (come ogni linguaggio), in parte ereditate dal C:

- La potenza e la flessibilità tipiche del C e del C++ non sono gratuite. Se da una parte è vero che è possibile ottenere applicazioni in generale più efficienti (rispetto ad agli altri linguaggi), e anche vero che tutto questo è ottenuto lasciando in mano al programmatore molti dettagli e compiti che negli altri linguaggi sono svolti dal compilatore; è quindi necessario un maggiore lavoro in fase di progettazione e una maggiore attenzione ai particolari in fase di realizzazione, pena una valanga di errori spesso subdoli e difficili da individuare che possono far levitare drasticamente i costi di produzione;

- Il compilatore e il linker del C++ soffrono di problemi relativi all'ottimizzazione del codice dovuti alla falsa assunzione che programmi C e C++ abbiano comportamenti simili a run time: il compilatore nella stragrande maggioranza dei casi si limita ad eseguire le ottimizzazioni tradizionali, sostanzialmente valide in linguaggi come il C, ma spesso inadatte a linguaggi pesantemente basati sulla programmazione ad oggetti; il linker poi da parte sua è rimasto immutato e non esegue alcun tipo di ottimizzazione che non possono essere effettuate a compile-time;

- Infine manca ancora uno standard per il linguaggio, cosa che crea problemi in fase di porting su altre piattaforme. Fortunatamente uno standard e ormai in avanzata fase di discussione ed è possibile consultare i primi draft;

Obiettivo di quanto segue è quello di introdurre alla programmazione in C++, spiegando sintassi e semantica dei suoi costrutti anche con l'ausilio di opportuni esempi. All'inizio verranno trattati gli aspetti basilari del linguaggio (tipi, dichiarazioni di variabili, funzioni,...), quando poi il lettore sarà in grado di comprendere, analizzare e realizzare un programma si procederà a trattare gli aspetti peculiari del linguaggio (classi, template, eccezioni); alla fine verranno analizzate alcune librerie relative all'input/output.

Il corso è rivolto a persone che non hanno alcuna conoscenza del linguaggio, ma potrà tornare utile anche a programmatori che possiedono una certa familiarità con esso. L'unico requisito richiesto è la conoscenza dei principi della programmazione orientata agli oggetti (OOP), tuttavia non essendo un corso di programmazione, la capacità di programmare in un qualsiasi altro linguaggio è ritenuta dote necessaria alla comprensione di quanto segue.

Per chi non avesse conoscenza di programmazione ad oggetti si rimanda a "I principi della programmazione orientata agli oggetti", Mondo Bit N.1 - Giugno 1995.
Salvo rare eccezioni non verranno discussi aspetti relativi a tematiche di implementazione dei vari meccanismi e altre note tecniche che esulano dagli obiettivi del corso.

Per eventuali domande e osservazioni sui contenuti di quanto segue potete fare riferimento al mio e-mail: Marotta@CLI.DI.UniPi.It, le vostre domande e le relative risposte (se ritenute di interesse generale) saranno ospitate in un apposito spazio.

Un ultimo avvertimento: quanto segue cerca di trarre i massimi benefici da una concezione stile ipertesto, in alcuni casi ci saranno dei link a pagine che dovranno ancora essere rese disponibili; mi scuso pertanto fin da ora per i disagi che ciò comporterà.

Elementi lessicali

Ogni programma scritto in un qualsiasi linguaggio di programmazione prima di essere eseguito viene sottoposto ad un processo di compilazione o interpretazione (a seconda che si usi un compilatore o un interprete). Lo scopo di questo processo è quello di tradurre il programma originale (codice sorgente) in uno semanticamente equivalente, ma eseguibile su una certa macchina. Il processo di compilazione è suddiviso in più fasi, ciascuna delle quali volta all'acquisizione di opportune informazioni necessarie alla fase successiva.

La prima di queste fasi è nota come analisi lessicale ed ha il compito di riconoscere gli elementi costitutivi del linguaggio sorgente, individuandone anche la categoria lessicale. Ogni linguaggio prevede un certo numero di categorie lessicali e in C++ possiamo distinguere in particolare le seguenti categorie lessicali:

- Commenti;
- Identificatori;
- Parole riservate;
- Costanti letterali;
- Segni di punteggiatura e operatori;

Analiziamole più in dettaglio.

Commenti

I commenti, come in qualsiasi altro linguaggio, hanno valore soltanto per il programmatore e vengono ignorati dal compilatore. È possibile inserirli nel proprio codice in due modi diversi:

1. secondo lo stile C ovvero racchiudendoli tra i simboli /* e */
2. facendoli precedere dal simbolo //

Nel primo caso è considerato commento tutto quello che è compreso tra /* e */, il commento quindi si può estendere anche su più righe o trovarsi in mezzo al codice:

```
void Func() {
  ...
  int a = 5; /* questo è un commento
                diviso su più righe */
  a = 4 /* commento */ + 5;
  ...
}
```

Nel secondo caso, proprio del C++, è invece considerato commento tutto ciò che segue // fino alla fine della linea, ne consegue che non è possibile inserirlo in mezzo al codice o dividerlo su più righe (a meno che anche l'altra riga non cominci con //):

```
void Func() {
  ...
  int a = 5; // questo è un commento valido
  a = 4      // sbagliato, il "+ 5;" è considerato commento + 5;
                e non è possibile dividerlo su più righe
  ...
}
```

Benché esistano due distinti metodi per commentare il codice, non è possibile avere commenti annidati, il primo simbolo tra // e /* determina il tipo di commento che l'analizzatore lessicale si aspetta. Bisogna anche ricordare di separare sempre i caratteri di inizio commento dall'operatore di divisione (simbolo /):

```
a + c //* commento */ 3
```

Tutto ciò che segue "a + c" viene interpretato come un commento iniziato da //, è necessario inserire uno spazio tra / e /*.

Identificatori

Gli identificatori sono simboli definiti dal programmatore per riferirsi a cinque diverse categorie di oggetti:

- Variabili;
- Costanti simboliche;
- Etichette;
- Tipi definiti dal programmatore;
- Funzioni;

Le variabili sono contenitori di valori di un qualche tipo; ogni variabile può contenere un singolo valore che può cambiare nel tempo, il tipo di questo valore viene comunque stabilito una volta per tutte e non può cambiare.
Le costanti simboliche servono ad identificare valori che non cambiano nel tempo, non possono essere considerate dei contenitori, ma solo un nome per un valore.
Una etichetta è un nome il cui compito è quello di identificare una istruzione del programma e sono utilizzate dall'istruzione di salto incondizionato goto.
Un tipo invece, come vedremo meglio in seguito, identifica un insieme di valori e di operazioni definite su questi valori; ogni linguaggio fornisce un certo numero di tipi primitivi (cui è associato un identificatore di tipo predefinito) e dei meccanismi per permettere la costruzione di nuovi tipi (a cui il programmatore deve poter associare un nome) a partire da quelli primitivi.
Infine funzione è il termine che il C++ utilizza per indicare i sottoprogrammi.
Parleremo comunque con maggior dettaglio di variabili, costanti, etichette, tipi e funzioni in seguito.
Un identificatore deve iniziare con una lettera o con underscore _ seguita da un numero qualsiasi di lettere, cifre o underscore; viene fatta distinzione tra lettere maiuscole e lettere minuscole. Benché il linguaggio non preveda un limite alla lunghezza massima di un identificatore, è praticamente impossibile non imporre un limite al numero di caratteri considerati significativi, per cui ogni compilatore distingue gli identificatori in base a un certo numero di caratteri iniziali tralasciando i restanti; il numero di caratteri considerati significativi varia comunque da sistema a sistema.

Parole riservate

Ogni linguaggio si riserva delle parole chiave (keywords) il cui significato è prestabilito e che non possono essere utilizzate dal programmatore come identificatori. Il C++ non fa eccezione:

asm	continue	float	new	signed	try
auto	default	for	operator	sizeof	typedef
break	delete	friend	private	static	union
case	do	goto	protected	struct	unsigned
catch	double	if	public	switch	virtual
char	else	inline	register	template	void
class	enum	int	return	this	volatile
const	extern	long	short	trow	while

Sono inoltre considerate parole chiave tutte quelle che iniziano con un doppio underscore __; esse sono riservate per le implementazioni del linguaggio e per le librerie standard e il loro uso da parte del programmatore dovrebbe essere evitato in quanto non sono portabili.

Costanti letterali

All'interno delle espressioni è possibile inserire direttamente dei valori, questi valori sono detti costanti letterali. La generica costante letterale può essere un carattere racchiuso tra apice singolo, una stringa racchiusa tra doppi apici, un intero o un numero in virgola mobile.

```
'a'    // Costante di tipo carattere
"a"    // Stringa di un carattere
"abc"  // Ancora una stringa
```

Un intero può essere:

- Una sequenza di cifre decimali, eventualmente con segno;
- Uno 0 (zero) seguito da un intero in ottale (base 8);
- 0x o 0X seguito da un intero in esadecimale (base 16);

Nella rappresentazione in esadecimale, oltre alle cifre decimali, è consentito l'uso delle lettere da "A" a "F" e da "a" a "f".
Il tipo (Vedi tipi di dato) in cui viene convertita la costante intera dipende dalla rappresentazione utilizzata e dal valore:

- Base 10: il più piccolo atto a contenerla tra int, long int e unsigned long int

- Base 8 o 16: il più piccolo atto a contenerla tra int, unsigned int, long int e unsigned long int

Si può forzare il tipo da utilizzare aggiungendo alla costante un suffisso costituito da u o U, e/o l o L: la lettera U seleziona i tipi unsigned e la L i tipi long; se solo una tra le due lettere viene specificata, viene scelto il più piccolo di quelli atti a contenere il valore e selezionati dal programmatore:

```
20      // intero in base 10
024     // 20 in base 8
0x14    // 20 in base 16
12ul    // forza unsigned long
12l     // forza long
12u     // forza unsigned
```

Un valore in virgola mobile è costituito da:

- Intero decimale, opzionalmente con segno;
- Punto decimale
- Frazione decimale;
- e o E e un intero decimale con segno;

È possibile omettere uno tra l'intero decimale e la frazione decimale, ma non entrambi. È possibile omettere uno tra il punto decimale e la lettera E (o e) e l'intero decimale con segno, ma non entrambi. L'uso della lettera E indica il ricorso alla notazione scientifica.
Il tipo scelto per rappresentare una costante in virgola mobile è double, se non diversamente specificato utilizzando i suffissi F o f per float, o L o l per long double Esempi:

```
.0          // 0 in virgola mobile
110E+4      // equivalente a 110 * 10^4 (10 elevato a 4)
.14e-2      // 0.0014
-3.5e+3     // -3500.0
3.5f        // forza float
3.4L        // forza long double
```

Segni di punteggiatura e operatori

Alcuni simboli sono utilizzati dal C++ per separare i vari elementi sintattici o lessicali di un programma o come operatori per costruire e manipolare espressioni:

```
[   ]   (   )   {   }   +   -   *   /   %   !   ^   &
|   \   ;   '   :   "   <   >   ?   ,   .   ~   =
```

Anche le seguenti combinazioni di simboli sono operatori:

```
++   --   ->   .*   ->*   <<   >>   <=   >=   ==   !=   &&
||   +=   -=   *=   <<=   /=   %=   &=   ^=   |=   ::   >>=
```

Esamineremo meglio i vari simboli più avanti.

Espressioni e istruzioni

Inizieremo ad esaminare i costrutti del C++ partendo proprio dalle istruzioni e dalle espressioni, perché in questo modo sarà più semplice esemplificare alcuni concetti che verranno analizzati nel seguito. Per adesso comunque analizzeremo solo le istruzioni per il controllo del flusso e l'assegnamento, le rimanenti (poche) istruzioni verranno discusse via via che sarà necessario nei prossimi capitoli.

Assegnamento

Il C++ è un linguaggio pesantemente basato sul paradigma imperativo, questo vuol dire che un programma C++ è sostanzialmente una sequenza di assegnamenti di valori a variabili. È quindi naturale iniziare parlando proprio dell'assegnamento.
L'operatore di assegnamento è denotato dal simbolo = (uguale) e viene applicato con la sintassi:

```
lvalue = rvalue;
```

Il termine *lvalue* indica una qualsiasi espressione che riferisca ad una regione di memoria (in generale un identificatore di variabile), mentre un *rvalue* è una qualsiasi espressione la cui valutazione produca un valore. Ecco alcuni esempi:

```
Pippo = 5;
Topolino = 'a';
Clarabella = Pippo;
Pippo = Pippo + 7;
Clarabella = 4 + 25;
```

Il risultato dell'assegnamento è il valore prodotto dalla valutazione della parte destra (rvalue) e ha come effetto collaterale l'assegnazione di tale valore alla regione di memoria denotato dalla parte sinistra (lvalue), ciò vuol dire che ad esempio che il primo assegnamento sopra produce come risultato il valore 5 e che dopo tale assegnamento la valutazione della variabile Pippo produrrà tale valore fino a che un nuovo assegnamento non verrà eseguito su tale variabile.

Si osservi che una variabile può apparire sia a destra che a sinistra di un assegnamento, se tale occorrenza si trova a destra produce il valore contenuto nella variabile, se invece si trova a sinistra essa denota la locazione di memoria cui riferisce. Ancora, poiché un identificatore di variabile può trovarsi contemporaneamente su ambo i lati di un assegnamento è necessaria una semantica non ambigua: come in qualsiasi linguaggio imperativo (Pascal, Basic, ...) la semantica dell'assegnamento impone che prima si valuti la parte destra e poi si esegua l'assegnamento del valore prodotto all'operando di sinistra.

Poiché un assegnamento produce come risultato il valore prodotto dalla valutazione della parte destra (è cioè a sua volta una espressione), è possibile legare in cascata più assegnamenti:

```
Clarabella = Pippo = 5;
```

Essendo l'operatore di assegnamento associativo a destra, l'esempio visto sopra è da interpretare come

```
Clarabella = (Pippo = 5);
```

cioè viene prima assegnato 5 alla variabile Pippo e il risultato di tale assegnamento (il valore 5) viene poi assegnato alla variabile Clarabella.
Esistono anche altri operatori che hanno come effetto collaterale l'assegnazione di un valore, la maggior parte di essi sono comunque delle utili abbreviazioni, eccone alcuni esempi:

```
Pippo += 5;     // equivale a Pippo = Pippo + 5;
Pippo -= 10;    // equivale a Pippo = Pippo - 10;
Pippo *= 3;     // equivale a Pippo = Pippo * 3;
```

si tratta cioè di operatori derivati dalla concatenazione dell'operatore di assegnamento con un altro operatore binario.
Gli altri operatori che hanno come effetto laterale l'assegnamento sono quelli di autoincremento e autodecremento, ecco come possono essere utilizzati:

```
Pippo++;        // cioè Pippo += 1;
++Pippo;        // sempre Pippo += 1;
Pippo--;        // Pippo -= 1;
--Pippo;        // Pippo -= 1;
```

Questi due operatori possono essere utilizzati sia in forma prefissa (righe 2 e 4) che in forma postfissa (righe 1 e 3), il risultato comunque non è proprio identico: la forma postfissa restituisce come risultato il valore della variabile e poi incrementa tale valore e lo assegna alla variabile, la forma prefissa invece prima modifica il valore associato alla variabile e poi restituisce tale valore:

```
Clarabella = ++Pippo;
// equivale a:
Pippo++;
Clarabella = Pippo;

// invece
Clarabella = Pippo++;
// equivale a:
Clarabella = Pippo;
Pippo++;
```

Espressioni

Le espressioni, per quanto visto sopra, rappresentano un elemento basilare del C++, tant'è che il linguaggio fornisce un ampio insieme di operatori. Eccone l'elenco completo:

SOMMARIO DEGLI OPERATORI

```
::                  risolutore di scope

.                   selettore di campi
->                  selettore di campi
[ ]                 sottoscrizione
( )                 chiamata di funzione
( )                 costruttore di valori
++                  post incremento
--                  post decremento

sizeof              dimensione di
++                  pre incremento
--                  pre decremento
~                   complemento
!                   negazione
-                   meno unario
+                   più unario
&                   indirizzo di
*                   dereferenzazione
new                 allocatore di oggetti
delete              deallocatore di oggetti
delete[ ]           deallocatore di array
( )                 conversione di tipo


.*                  selettore di campi
->*                 selettore di campi
```

```
*                   moltiplicazione
/                   divisione
%                   modulo (resto)

+                   somma
-                   sottrazione

<<                  shift a sinistra
>>                  shift a destra

<                   minore di
<=                  minore o uguale
>                   maggiore di
>=                  maggiore o uguale

==                  uguale a
!=                  diverso da

&                   AND di bit
^                   OR ESCLUSIVO di bit
|                   OR INCLUSIVO di bit
&&                  AND logico
||                  OR logico (inclusivo)
? :                 espressione condizionale

=                   assegnamento semplice
*=                  moltiplica e assegna
/=                  divide e assegna
%=                  modulo e assegna
+=                  somma e assegna
-=                  sottrae e assegna
<<=                 shift sinistro e assegna
>>=                 shift destro e assegna
&=                  AND e assegna
|=                  OR inclusivo e assegna
^=                  OR esclusivo e assegna

throw               lancio di eccezioni
,                   virgola
```

Gli operatori sono raggruppati in base alla loro precedenza: in alto quelli a precedenza maggiore, una linea vuota separa gli operatori con priorità maggiore da quelli con priorità minore. Gli operatori unari e quelli di assegnamento sono associativi a destra, gli altri a sinistra. L'ordine di valutazione delle sottoespressioni che compongono una espressione più grande non è definito.
Gli operatori di assegnamento e quelli di (auto)incremento e (auto)decremento sono già stati descritti, esaminiamo ora l'operatore per le espressioni condizionali. L'operatore ? : è l'unico operatore ternario:

```
<Cond> ? <Expr2> : <Expr3>
```

Per definire la semantica di questo operatore è necessario prima parlare di vero e falso in C++. A differenza di linguaggi quali il Pascal, il C++ non fornisce un tipo primitivo (vedi tipi primitivi) per codificare i valori booleani; essi sono rappresentati tramite valori interi: 0 (zero) indica falso e un valore diverso da 0 indica vero. Ciò implica che ovunque sia richiesta una condizione è possibile mettere una qualunque espressione che possa produrre un valore intero (quindi anche una somma, ad esempio). Non solo, dato che l'applicazione di un operatore booleano o relazionale a due sottoespressioni produce 0 o 1 (a seconda del valore di verità della formula), è possibile mescolare operatori booleani, relazionali e aritmetici.

Premesso ciò, la semantica associata all'operatore ? : è la seguente: si valuta Cond, se essa è vera (diversa da zero) il risultato di tale operatore è la valutazione di Expr2, altrimenti il risultato è Expr3.

Per quanto riguarda gli altri operatori, alcuni saranno esaminati quando sarà necessario, non verranno invece discussi gli operatori logici e quelli di confronto (la cui semantica viene considerata nota al lettore).
Rimangono gli operatori per lo spostamento di bit, ci limiteremo a dire che servono sostanzialmente a eseguire moltiplicazioni e divisioni per multipli di 2 in modo efficiente.

Controllo del flusso

Esamineremo ora le istruzioni per il controllo del flusso, ovvero quelle istruzioni che consentono di eseguire una certa sequenza di istruzioni, o eventualmente un'altra, in base al valore di una espressione.

IF-ELSE

L'istruzione condizionale if-else ha due possibili formulazioni:

```
if ( <Condizione> ) <Istruzione1> ;
```

oppure

```
if ( <Condizione> ) <Istruzione1> ;
else <Istruzione2> ;
```

L'else è quindi opzionale, ma, se utilizzato, nessuna istruzione deve essere inserita tra il ramo if e il ramo else. Vediamo ora la semantica di tale istruzione.
In entrambi i casi se Condizione è vera viene eseguita Istruzione1, altrimenti nel primo caso non viene eseguita alcuna istruzione, nel secondo si esegue Istruzione2. Si osservi che Istruzione1 e Istruzione2 sono istruzioni singole (una sola istruzione), se è necessaria una sequenza di istruzioni esse devono essere racchiuse tra una coppia di parentesi graffe { }, come mostra l'esempio (si considerino X, Y e Z variabili intere)

```
if ( X==10 ) X--;
else {
  Y++;
  Z*=Y;      }
```

Ancora alcune osservazioni: il linguaggio prevede che due istruzioni consecutive siano separate da ; (punto e virgola), in particolare si noti il punto e virgola tra il ramo if e l'else; l'unica eccezione alla regola è data dalle istruzioni composte (cioè sequenze di istruzioni racchiuse tra parentesi graffe) che non devono essere seguite dal punto e virgola (non serve, c'è la parentesi graffa).

Un'ultima osservazione, per risolvere eventuali ambiguità il compilatore lega il ramo else con la prima occorrenza libera di if che incontra tornando indietro (si considerino Pippo, Pluto e Topolino variabili intere):

```
if (Pippo) if (Pluto) Topolino = 1;
else Topolino =2;
```

viene interpretata come

```
if (Pippo)
  if (Pluto) Topolino = 1;
  else Topolino =2;
```

l'else viene cioè legato al secondo if.

WHILE & DO-WHILE

I costrutti while e do while consentono l'esecuzione ripetuta di una sequenza di istruzioni in base al valore di verità di una condizione. Vediamone la sintassi:

```
while ( <Condizione> ) <Istruzione> ;
```

Al solito, Istruzione indica una istruzione singola, se è necessaria una sequenza di istruzioni essa deve essere racchiusa tra parentesi graffe.
La semantica del while è la seguente: prima si valuta Condizione e se essa è vera (diversa da 0) si esegue Istruzione e poi si ripete il tutto; l'istruzione termina quando Condizione valuta a 0 (falsa).

Esaminiamo ora l'altro costrutto:

```
do <Istruzione> while ( <Condizione> ) ;
```

Nuovamente, Istruzione indica una istruzione singola, se è necessaria una sequenza di istruzioni essa deve essere racchiusa tra parentesi graffe; si noti inoltre che Istruzione non è seguita da punto e virgola.
Il do while differisce dall'istruzione while in quanto prima si esegue Istruzione e poi si valuta Condizione, se essa è vera si riesegue il corpo altrimenti l'istruzione termina; il corpo del do while viene quindi eseguito sempre almeno una volta.

Ecco un esempio:

```
// Calcolo del fattoriale tramite while
if (InteroPositivo) {
  Fattoriale = InteroPositivo;
  while (--InteroPositivo)
    Fattoriale *= InteroPositivo;
  }
else Fattoriale = 1;

// Calcolo del fattoriale tramite do-while
Fattoriale = 1;
if (InteroPositivo)
  do
    Fattoriale *= InteroPositivo
  while (--InteroPositivo);
```

IL CICLO FOR

Come i più esperti sapranno, il ciclo for è una specializzazione del while, tuttavia nel C++ la differenza tra for e while è talmente sottile che i due costrutti possono essere liberamente scambiati tra loro.
La sintassi del for è la seguente:

```
for ( <Inizializzazione>  ;  <Condizione>  ;  <Iterazione>  )
  <Istruzione> ;
```

Inizializzazione può essere una espressione che inizializza le variabili del ciclo o una dichiarazione di variabili (nel qual caso le variabili dichiarate hanno scope e lifetime limitati a tutto il ciclo); Condizione è una qualsiasi espressione a valori interi; e Iterazione è una istruzione da eseguire dopo ogni iterazione (solitamente un incremento).

Tutti e tre gli elementi appena descritti sono opzionali, in particolare se Condizione non viene specificata si assume che essa sia sempre verificata .

Ecco la semantica del for espressa tramite while (a meno di una istruzione continue contenuta in Istruzione):

```
<Inizializzazione> ;
while ( <Condizione> ) {
  <Istruzione> ;
  <Iterazione> ;
```

```
}
```

Una eventuale istruzione continue (vedi paragrafo successivo) in Istruzione causa un salto a Iterazione nel caso del ciclo for, nel while invece causa una uscita dal ciclo. Ecco come usare il ciclo for per calcolare il fattoriale:

```
for (Fatt = IntPos ? IntPos : 1; IntPos > 1; --IntPos)
  Fatt *= IntPos;
```

BREAK & CONTINUE

Le istruzioni break e continue consentono un maggior controllo sui cicli. Nessuna delle due istruzioni accetta argomenti. L'istruzione break può essere utilizzata dentro un ciclo o una istruzione switch (vedi paragrafo successivo) e causa la terminazione del ciclo in cui occorre (o dello switch).
L'istruzione continue può essere utilizzata solo dentro un ciclo e causa l'interruzione della corrente esecuzione del corpo del ciclo; a differenza di break quindi il controllo non viene passato all'istruzione successiva al ciclo, ma al punto immediatamente prima della fine del body del ciclo (pertanto il ciclo potrebbe ancora essere eseguito):

```
Fattoriale = 1;
while (1) {    // all'infinito...
  if (InteroPositivo > 1) {
    Fattoriale *=Interopositivo--;
    continue;
  }
  break;      // se eseguita allora InteroPositivo <= 1
              // continue provoca un salto in questo punto
}
```

SWITCH

L'istruzione switch è molto simile al case del Pascal (anche se più potente) e consente l'esecuzione di uno o più frammenti di codice a seconda del valore di una espressione:

```
switch ( <Espressione> ) {
  case <Valore1> : <Istruzione> ;
  /* ... */
  case <ValoreN> : <Istruzione> ;
  default : <Istruzione> ;
}
```

Espressione è una qualunque espressione capace di produrre un valore intero; Valore1...ValoreN sono costanti diverse tra loro; Istruzione è una qualunque sequenza di istruzioni (non racchiuse tra parentesi graffe).

All'inizio viene valutata Espressione e quindi viene eseguita l'istruzione relativa alla clausola case che specifica il valore prodotto da Espressione; se nessuna clausola case specifica il valore prodotto da Espressione viene eseguita l'istruzione relativa a default, se specificato (il ramo default è opzionale).

Ecco alcuni esempi:

```
switch (Pippo) {
  case 1 :
    Topolino = 5;
  case 4 :
    Topolino = 2;
    Clarabella = 7;
  default :
```

```
switch (Pluto) {
  case 5 :
    Pippo = 3;
  case 6 :
    Pippo = 5;
  case 10 :
    Orazio = 20;
```

```
        Topolino = 0;                Tip = 7;
    }                            }        // niente caso default
```

Il C++ (come il C) prevede il fall-through automatico tra le clausole dello switch, cioè il controllo passa da una clausola case alla successiva (default compreso) anche quando la clausola viene eseguita. Per evitare ciò è sufficiente terminare le clausole con break in modo che, alla fine dell'esecuzione della clausola, termini anche lo switch:

```
switch (Pippo) {
  case 1 :
    Topolino = 5;
    break;
  case 4 :
    Topolino = 2;
    Clarabella = 7;
    break
  default :
    Topolino = 0;
    break;
}
```

GOTO

Il C++ prevede la tanto deprecata istruzione goto per eseguire salti incondizionati. La cattiva fama del goto deriva dal fatto che il suo uso tende a rendere obbiettivamente incomprensibile un programma; tuttavia in certi casi (tipicamente applicazioni real-time) le prestazioni sono assolutamente prioritarie e l'uso del goto consente di ridurre al minimo i tempi. Comunque quando possibile è sempre meglio evitare l'uso di goto.
L'istruzione goto prevede che l'istruzione bersaglio del salto sia etichettata tramite un identificatore utilizzando la sintassi

```
<Etichetta> : <Istruzione> ;
```

che serve anche a dichiarare Etichetta. Il salto ad una istruzione viene eseguito con

```
goto <Etichetta> ;
```

ad esempio:

```
      if (Pippo == 7) goto OK;
      Topolino = 5;
      /* ... */
OK : Pluto = 7;
```

Si noti che una etichetta può essere utilizzata anche prima di essere dichiarata. Esiste una limitazione all'uso del goto: il bersaglio dell'istruzione (cioè Etichetta) deve trovarsi all'interno della stessa funzione dove appare l'istruzione di salto.

Dichiarazioni

Ogni identificatore che il programmatore intende utilizzare in un programma C++, sia esso per una variabile, una costante simbolica, di tipo o di funzione (fanno eccezione le etichette), va dichiarato prima di essere utilizzato. Ci sono diversi motivi che giustificano la necessità di una dichiarazione; nel caso di variabili, costanti o tipi:

- consente di stabilire la quantità di memoria necessaria alla memorizzazione di un oggetto;
- determina l'interpretazione da attribuire ai vari bit che compongono la regione di memoria utilizzata per memorizzare l'oggetto, l'insieme dei valori che può assumere e le operazioni che possono essere fatte su di esso;
- permette l'esecuzione di opportuni controlli per determinare errori semantici;
- fornisce eventuali suggerimenti al compilatore;

nel caso di funzioni, invece una dichiarazione:

- determina numero e tipo dei parametri e il tipo del valore tornato;
- consente controlli per determinare errori semantici;

Le dichiarazioni hanno anche altri compiti che saranno chiariti in seguito.

Tipi primitivi

Un tipo è una coppia <V, O>, dove V è un insieme di valori e O è un insieme di operazione per la creazione e la manipolazione di elementi di V.
In un linguaggio di programmazione i tipi rappresentano le categorie di informazioni che il linguaggio consente di manipolare. Il C++ fornisce quattro tipi fondamentali:

- *char*
- *int*
- *float*
- *double*

Il tipo *char* è utilizzato per rappresentare piccoli interi e caratteri; *int* è utilizzato per rappresentare interi in un intervallo più grande di *char*; infine *float* e *double* rappresentano entrambi valori in virgola mobile, *float* per valori in precisione semplice e *double* per quelli in doppia precisione.

Ai tipi fondamentali è possibile applicare i qualificatori *signed*, *unsigned*, *short* e *long* per selezionare differenti intervalli di valori; essi inoltre non sono liberamente applicabili a tutti i tipi: *short* si applica solo a *int*, *signed* e *unsigned* solo a *char* e *int* e infine *long* solo a *int* e *double*. In definitiva sono disponibili i tipi:

char
short int
int

long int
signed char
signed short int
signed int
signed long int
unsigned char
unsigned short int
unsigned int
unsigned long int
float

double
long double

Il tipo int è per default signed e quindi il generico tipo int è equivalente al corrispondente tipo signed, invece i tipi char, signed char e unsigned char sono considerati tipi distinti. I vari tipi sopra elencati, oltre a differire per l'intervallo dei valori rappresentabili, differiscono anche per la quantità di memoria richiesta per rappresentare un valore di quel tipo.

Il seguente programma permette di conoscere la dimensione di ciascun tipo come multiplo di char (di solito rappresentato su 8 bit):

```
#include <iostream.h>

void main() {
  cout << "char = " << sizeof(char) << endl;
  cout << "short int = " << sizeof(short int) << endl;
  cout << "int = " << sizeof(int) << endl;
  cout << "long int = " << sizeof(long int) << endl;
  cout << "signed char = " << sizeof(signed char) << endl;
  cout << "signed short int = " << sizeof(signed short int) << endl;
  cout << "signed int = " << sizeof(signed int) << endl;
  cout << "signed long int = " << sizeof(signed long int) << endl;
  cout << "unsigned char = " << sizeof(unsigned char) << endl;
  cout << "unsigned short int = " << sizeof(unsigned short int) << endl;
  cout << "unsigned int = " << sizeof(unsigned int) << endl;
  cout << "unsigned long int = " << sizeof(unsigned long int) << endl;
  cout << "float = " << sizeof(float) << endl;
  cout << "double = " << sizeof(double) << endl;
  cout << "long double = " << sizeof(long double) << endl;
}
```

Una veloce spiegazione sul listato:
la prima riga (`#include <iostream.h>`) richiede l'uso di una libreria per eseguire l'output su video; la libreria iostream.h dichiara l'oggetto cout il cui compito è quello di visualizzare l'output che gli viene inviato tramite l'operatore di inserimento <<.
L'operatore `sizeof(<Tipo>)` restituisce la dimensione di Tipo, mentre `endl` inserisce un ritorno a capo e forza la visualizzazione dell'output. Infine `main` è il nome che identifica la funzione principale, ovvero il corpo del programma.

Tra i tipi fondamentali sono definiti gli operatori di conversione, il loro compito è quello di trasformare un valore di un tipo in un valore di un altro tipo. Non esamineremo per adesso l'argomento, esso verrà ripreso in una apposita appendice.

Variabili e costanti

Siamo ora in grado di dichiarare variabili e costanti. La sintassi per la dichiarazione delle variabili è

```
<Tipo> <Lista Di Identificatori> ;
```

Ad esempio:

```
int a, b, B, c;
signed char Pippo;
unsigned short Pluto;     // se omesso si intende int
```

Innanzi tutto ricordo che il C++ è case sensitive, cioè distingue le lettere maiuscole da quelle minuscole, infine si noti il punto e virgola che segue sempre ogni dichiarazione.

La prima riga dichiara quattro variabili di tipo int, mentre la seconda una di tipo signed char. La terza dichiarazione è un po' particolare in quanto apparentemente manca la keyword int, in realtà poiché il default è proprio int essa può essere omessa;

in conclusione la terza dichiarazione introduce una variabile di tipo unsigned short int. Gli identificatori che seguono il tipo sono i nomi delle variabili, se più di un nome viene specificato essi devono essere separati da una virgola.

È possibile specificare un valore con cui inizializzare ciascuna variabile facendo seguire il nome dall'operatore di assegnamento = e da un valore o una espressione che produca un valore del corrispondente tipo:

```
int a = -5, b = 3+7, B = 2, c = 1;
signed char Pippo = 'a';
unsigned short Pluto = 3;
```

Se nessun valore iniziale viene specificato, il compilatore inizializza le variabili con 0.
La dichiarazione delle costanti è identica a quella delle variabili eccetto che deve sempre essere specificato un valore e la dichiarazione inizia con la keyword const:

```
const a = 5, c = -3;     // int è sottinteso
const unsigned char d = 'a', f = 1;
const float = 1.3;
```

Scope e lifetime

La dichiarazione di una variabile o di un qualsiasi altro identificatore si estende dal punto immediatamente successivo la dichiarazione (e prima dell'eventuale inizializzazione) fino alla fine del blocco di istruzioni in cui è inserita (un blocco di istruzioni è racchiuso sempre tra una coppia di parentesi graffe). Ciò vuol dire che quella dichiarazione non è visibile all'esterno di quel blocco, mentre è visibile in eventuali blocchi annidati dentro quello dove la variabile è dichiarata.
Il seguente schema chiarisce la situazione:

```
                // Qui X non è visibile
{
...             // Qui X non è visibile
int X = 5;      // Da ora in poi esiste una variabile X
...             // X è visibile già prima di =
  {             // X è visibile anche in questo blocco
  ...
  }
...
}               // X ora non è più visibile
```

All'interno di uno stesso blocco non è possibile dichiarare più volte lo stesso identificatore, ma è possibile ridichiararlo in un blocco annidato; in tal caso la nuova dichiarazione nasconde quella più esterna che ritorna visibile non appena si esce dal blocco ove l'identificatore viene ridichiarato:

```
{
...                 // qui X non è ancora visibile
int X = 5;
...                 // qui è visibile int X
  {
  ...               // qui è visibile int X
  char X = 'a';     // ora è visibile char X
  ...               // qui è visibile char X
  }                 // qui è visibile int X
...
}                   // X ora non più visibile
```

All'uscita dal blocco più interno l'identificatore ridichiarato assume il valore che aveva prima di essere ridichiarato:

```
{
  ...
```

```
  int X = 5;
  cout << X << endl;      // stampa 5
  while (--X) {                  // riferisce a int X
    cout << X << ' ';     // stampa int X
    char X = '-';
    cout << X << ' ';     // ora stampa char X
  }
  cout << X << endl;      // stampa di nuovo int X
}
```

Una dichiarazione eseguita fuori da ogni blocco introduce un identificatore globale a cui ci si può riferire anche con la notazione `::<ID>`. Ad esempio:

```
int X = 4;   // dichiarazione esterna ad ogni blocco;

void main() {
  int X = -5, y = 0;
  /* ... */
  y = ::X;     // a y viene assegnato 4
  y = X;       // assegna il valore -5
}
```

Abbiamo appena visto che per assegnare un valore ad una variabile si usa lo stesso metodo con cui la si inizializza quando viene dichiarata. L'operatore `::` è detto *risolutore di scope* e, utilizzato nel modo appena visto, permette di riferirsi alla dichiarazione globale di un identificatore.

Ogni variabile oltre a possedere uno scope, ha anche un propria durata (*lifetime*), viene creata subito dopo la dichiarazione (e prima dell'inizializzazione! ndr) e viene distrutta alla fine del blocco dove è posta la dichiarazione; fanno eccezione le variabili globali che vengono distrutte alla fine dell'esecuzione del programma. Da ciò si deduce che le variabili locali (ovvero quelle dichiarate all'interno di un blocco) vengono create ogni volta che si giunge alla dichiarazione, e distrutte ogni volta che si esce dal blocco; è tuttavia possibile evitare che una variabile locale (dette anche automatiche) venga distrutta all'uscita dal blocco facendo precedere la dichiarazione dalla *keyword static* :

```
void func() {
  int x = 5;          // x è creata e distrutta ogni volta
  static int c = 3;  // c si comporta in modo diverso
  /* ... */
}
```

La variabile x viene creata e inizializzata a 5 ogni volta che func() viene eseguita, e viene distrutta alla fine dell'esecuzione della funzione; la variabile c invece viene creata e inizializzata una sola volta, quando la funzione viene chiamata la prima volta, ma viene distrutta solo alla fine del programma. Le variabili statiche conservano sempre l'ultimo valore che viene assegnato ad esse e servono per realizzare funzioni il cui comportamento è legato a computazioni precedenti (all'interno della stessa esecuzione del programma). Infine la *keyword static* non modifica lo *scope*.

Costruire nuovi tipi

Il C++ permette la definizione di nuovi tipi. I tipi definiti dal programmatore vengono detti "Tipi definiti dall'utente" e possono essere utilizzati ovunque è richiesto un identificatore di tipo (con rispetto alle regole di visibilità viste precedentemente). I nuovi tipi vengono definiti applicando dei costruttori di tipi ai tipi primitivi (quelli forniti dal linguaggio) o a tipi precedentemente definiti dall'utente.

I costruttori di tipo disponibili sono:

- il costruttore di array: [ ]
- il costruttore di aggregati: struct
- il costruttore di unioni: union
- il costruttore di tipi enumerati: enum
- la keyword typedef
- il costruttore di classi: class

Per adesso tralasceremo il costruttore di classi, ci occuperemo di esso in seguito in quanto alla base della programmazione in C++ e meritevole di una trattazione separata.

Array

Per quanto visto precedentemente, una variabile può contenere un solo valore alla volta; il costruttore di array [ ] permette di raccogliere sotto un solo nome più variabili dello stesso tipo.
La dichiarazione

```
int Array[10];
```

introduce con il nome Array 10 variabili di tipo int; il tipo di Array è array di 10 int(eri).
La sintassi per la generica dichiarazione di un array è

```
<NomeTipo> <Identificatore>[ <NumeroDiElementi> ];
```

Al solito *NomeTipo* può essere sia un tipo primitivo che uno definito dal programmatore tramite uno degli altri meccanismi, *Identificatore* è un nome scelto dal programmatore per identificare l'array, mentre *NumeroDiElementi* deve essere un intero positivo.

Il generico elemento dell'array viene selezionato con la notazione Identificatore[Espressione], dove Espressione può essere una qualsiasi espressione che produca un valore intero; il primo elemento di un array è sempre Identificatore[0], e di conseguenza l'ultimo è Identificatore[NumeroDiElementi-1]:

```
float Pippo[10];
float Pluto;

Pippo[0] = 13.5;      // Assegna 13.5 al primo elemento
Pluto = Pippo[9];     // Seleziona l'ultimo elemento di Pippo
                      // e lo assegna a Pluto
```

È anche possibile dichiarare array multidimensionali (detti array di array o più in generale matrici) specificando più indici:

```
long double Qui[3][4];        // una matrice 3 x 4
short Quo[2][10];             // 2 array di 10 short int
int SuperPippo[12][16][20];   // matrice 12 x 16 x 20
```

E' anche possibile specificare i valori iniziali dei singoli elementi dell'array tramite una inizializzazione aggregata:

```
int Pippo[5]       =  { 10, -5, 6, 110, -96 };
short Pluto[2][4]  =  { 4, 7, 1, 4,
                        0, 3, 5, 9 );
float Minni[ ]     =  { 1.1, 3.5, 10.5 };
long Manetta[ ][3] =  { 5, -7, 2,
                        1, 0, 5 };
```

La prima dichiarazione è piuttosto semplice, dichiara un array di 5 elementi e per ciascuno di essi indica il valore iniziale a partire dall'elemento 0. La seconda dichiarazione è identica alla prima se si tiene conto che il primo indice a variare è l'ultimo, così che gli elementi vengono inizializzati nell'ordine Pluto[0][0], `Pluto[0][1], ..., Pluto[1][3].`

Le ultime due dichiarazioni sono più complesse in quanto non vengono specificati tutti gli indici degli array: in caso di inizializzazione aggregata il compilatore è in grado di determinare il numero di elementi relativi al primo indice in base al valore specificato per gli altri indici e al numero di valori forniti per l'inizializzazione, così che la terza dichiarazione introduce un array di 3 elementi e l'ultima una matrice 2 x 3. È possibile omettere solo il primo indice e solo in caso di inizializzazione aggregata.

Gli array consentono la memorizzazione di *stringhe* :

```
char Topolino[ ] = "investigatore" ;
```

La dimensione dell'array è pari a quella della stringa "investigatore" + 1, l'elemento in più è dovuto al fatto che in C++ le stringhe di default sono tutte terminate dal carattere nullo (\0) che il compilatore aggiunge automaticamente.
L'accesso agli elementi di Topolino avviene ancora tramite le regole viste sopra e non è possibile eseguire un assegnamento con la stessa metodologia dell'inizializzazione:

```
char Topolino[ ] = "investigatore" ;

Topolino[4] = 't';          // assegna 't' al quinto elemento
Topolino[ ] = "basso";      // errore
Topolino = "basso";         // ancora errore
```

È possibile inizializzare un array di caratteri anche nei seguenti modi:

```
char Topolino[ ] = { 'T', 'o', 'p', 'o', 'l', 'i', 'n', 'o'};
char Pluto[5]    = { 'P', 'l', 'u', 't', 'o' };
```

In questi casi però non si ottiene una stringa terminata da \0, ma semplici array di caratteri il cui numero di elementi è esattamente quello specificato.

Strutture

Gli array permettono di raccogliere sotto un unico nome più variabili omogenee e sono solitamente utilizzati quando bisogna operare su più valori contemporaneamente (ad esempio per eseguire una ricerca); da solo comunque il meccanismo degli array non consente la definizione di un nuovo tipo.
Solitamente per rappresentare entità complesse è necessario memorizzare informazioni di diversa natura; ad esempio per rappresentare una persona può non bastare una stringa per il nome ed il cognome, ma potrebbe essere necessario memorizzare anche età e codice fiscale.

Memorizzare tutte queste informazioni in un'unica stringa non è una buona idea poiché le singole informazioni non sono immediatamente disponibili, ma è necessario prima estrarle, inoltre nella rappresentazione verrebbero perse informazioni preziose quali il fatto che l'età è sempre data da un intero positivo.

D'altra parte avere variabili distinte per le singole informazioni non è certamente una buona pratica, diventa difficile capire qual è la relazione tra le varie componenti. La soluzione consiste nel raccogliere le variabili che modellano i singoli aspetti in un'unica struttura che consenta ancora di accedere ai singoli elementi:

```
struct Persona {
  char Nome[20];
  unsigned short Eta;
  char CodiceFiscale[16];
};
```

La precedente dichiarazione introduce un tipo struttura di nome Persona composto da tre campi: Nome, un array di 20 caratteri; Eta, un intero positivo; CodiceFiscale, un array di 16 caratteri.

La sintassi per la dichiarazione di una struttura è

```
struct <NomeTipo> {
  <Tipo> <NomeCampo> ;
  /* ... */
  <Tipo> <NomeCampo> ;
};
```

Si osservi che la parentesi graffa finale deve essere seguita da un punto e virgola, questo vale anche per le unioni, le enumerazioni e per le classi.
I singoli campi di una variabile di tipo struttura sono selezionabili tramite l'operatore di selezione . (punto), come mostrato nel seguente esempio:

```
struct Persona {
  char Nome[20];
  unsigned short Eta;
  char CodiceFiscale[16];
};

Persona Pippo = { "Pippo", 40, "PPP718F444E18DR0" };
Persona AmiciDiPippo[2] = { "Pluto", 40, "PLT",
                            "Minnie", 35, "MNN" };

// esempi di uso di strutture:

Pippo.Eta = 41;
unsigned short Var = Pippo.Eta;
strcpy(AmiciDiPippo[0].Nome, "Topolino");
```

Innanzi tutto viene dichiarato il tipo Persona e quindi si dichiara la variabile Pippo di tale tipo; in particolare viene mostrato come inizializzare la variabile con una inizializzazione aggregata del tutto simile a quanto si fa per gli array, eccetto che i valori forniti devono essere compatibili con il tipo dei campi e dati nell'ordine di dichiarazione dei campi.

Viene mostrata anche la dichiarazione di un array i cui elementi sono di tipo struttura, e il modo in cui eseguire una inizializzazione fornendo i valori necessari all'inizializzazione dei singoli campi di ciascun elemento dell'array (nell'ordine coerente alle dichiarazioni).

Le righe successive mostrano come accedere ai campi di una variabile di tipo struttura, in particolare l'ultima riga assegna un nuovo valore al campo Nome del primo elemento dell'array tramite una funzione di libreria. Si noti che prima viene selezionato l'elemento dell'array e poi il campo Nome di tale elemento; analogamente se è la struttura a contenere un campo di tipo non primitivo, prima si seleziona il campo e poi si seleziona l'elemento del campo che ci interessa:

```
struct Data {
  unsigned short Giorno, Mese;
  unsigned Anno;
};
```

```
struct Persona {
  char Nome[20];
  Data DataNascita;
};

Persona Pippo = { "pippo", 10, 9, 1950 };

Pippo.Nome[0] = 'P';
Pippo.DataNascita.Giorno = 15;
unsigned short UnGiorno = Pippo.DataNascita.Giorno;
```

Per le strutture, a differenza degli array, è definito l'operatore di assegnamento:

```
struct Data {
  unsigned short Giorno, Mese;
  unsigned Anno;
};

Data Oggi = { 10, 11, 1996 };
Data UnaData = { 1, 1, 1995};

UnaData = Oggi;
```

Ciò è possibile per le strutture solo perché, come vedremo, il compilatore le tratta come classi i cui membri sono tutti pubblici.
L'assegnamento è ovviamente possibile solo tra variabili dello stesso tipo struttura, ma quello che di solito sfugge è che due tipi struttura che differiscono solo per il nome sono considerati diversi:

```
// con riferimento al tipo Data visto sopra:

  struct DT {
    unsigned short Giorno, Mese;
    unsigned Anno;
  };

  Data Oggi = { 10, 11, 1996 };
  DT Ieri;

  Ieri = Oggi;     // Errore di tipo!
```

Unioni

Un costrutto sintatticamente simile alle strutture è quello delle unioni. Sintatticamente l'unica differenza è che nella dichiarazione di una unione viene utilizzata la keyword union anziché struct:

```
union TipoUnione {
  unsigned Intero;
  char Lettera;
  char Stringa[500];
};
```

Come per i tipi struttura, la selezione di un dato campo di una variabile di tipo unione viene eseguita tramite l'operatore di selezione . (punto).
Vi è tuttavia una profonda differenza tra il comportamento di una struttura e quello di una unione: in una struttura i vari campi vengono memorizzati in indirizzi diversi e non si sovrappongono mai, in una unione invece tutti i campi vengono memorizzati a partire dallo stesso indirizzo.
Ciò vuol dire che, mentre la quantità di memoria occupata da una struttura è data dalla somma delle quantità di memoria utilizzata dalle singole componenti, la quantità di memoria utilizzata da una unione è data da quella della componente più grande (Stringa nell'esempio precedente).

Dato che le componenti si sovrappongono, assegnare un valore ad una di esse vuol dire distruggere i valori memorizzati accedendo all'unione tramite una qualsiasi altra componente.
Le unioni vengono principalmente utilizzate per limitare l'uso di memoria memorizzando negli stessi indirizzi oggetti diversi in tempi diversi. C'è tuttavia un altro possibile utilizzo delle unioni, eseguire "manualmente" alcune conversioni di tipo. Tuttavia tale pratica è assolutamente da evitare (almeno quando esiste una alternativa) poiché tali conversioni sono dipendenti dall'architettura su cui si opera e pertanto non portabili.

Enumerazioni

A volte può essere utile poter definire un nuovo tipo estensionalmente, cioè elencando esplicitamente i valori che una variabile (o una costante) di quel tipo può assumere. Tali tipi vengono detti enumerati e vengono definiti tramite la keyword enum con la seguente sintassi:

```
enum <NomeTipo> {
  <Identificatore>,
  /* ... */
  <Identificatore>
};
```

Esempio:

```
enum Elemento {
  Idrogeno,
  Elio,
  Carbonio,
  Ossigeno
};

Elemento Atomo = Idrogeno;
```

Gli identificatori Idrogeno, Elio, Carbonio e Ossigeno costituiscono l'intervallo dei valori del tipo Elemento. Si osservi che come da sintassi, i valori di una enumerazione devono essere espressi tramite identificatori, non sono ammessi valori espressi in altri modi (interi, numeri in virgola mobile, costanti carattere...), inoltre gli identificatori utilizzati per esprimere tali valori devono essere distinti da qualsiasi altro identificatore visibile nello scope dell'enumerazione onde evitare ambiguità.
Il compilatore rappresenta internamente i tipi enumerazione associando a ciascun identificatore di valore una costante intera, così che un valore enumerazione può essere utilizzato in luogo di un valore intero, ma non viceversa:

```
enum Elemento {
  Idrogeno,
  Elio,
  Carbonio,
  Ossigeno
};

Elemento Atomo = Idrogeno;
int Numero;

Numero = Carbonio;    // Ok!
Atomo = 3;            // Errore!
```

Nell'ultima riga dell'esempio si verifica un errore perché non esiste un operatore di conversione da int a Elemento, mentre essendo i valori enumerazione in pratica delle costanti intere, il compilatore è in grado di eseguire la conversione a int.
È possibile forzare il valore intero da associare ai valori di una enumerazione:

```
enum Elemento {
  Idrogeno = 2,
```

```
  Elio,
  Carbonio = Idrogeno - 10,
  Ferro = Elio + 7,
  Ossigeno = 2
};
```

Non è necessario specificare un valore per ogni identificatore dell'enumerazione, non ci sono limitazioni di segno e non è necessario usare valori distinti. Si può utilizzare un identificatore dell'enumerazione precedentemente definito e non è necessario specificare un valore intero per ciascun identificatore dell'enumerazione.
La possibilità di scegliere i valori da associare alle etichette (identificatori) dell'enumerazione fornisce un modo alternativo di definire costanti di tipo intero.

La keyword typedef

Esiste anche la possibilità di dichiarare un alias per un altro tipo (non un nuovo tipo) utilizzando la parola chiave typedef:

```
typedef <Tipo> <Alias> ;
```

Il listato seguente mostra alcune possibili applicazioni:

```
typedef unsigned short int PiccoloIntero;
typedef long double ArrayDiReali[20];
typedef struct {
          long double ParteReale;
          long double ParteImmaginaria;
        } Complesso;
```

Il primo esempio mostra un caso molto semplice: creare un alias per un nome di tipo. Nel secondo caso invece viene mostrato come dichiarare un alias per un array di 20 long double. Infine il terzo esempio è il più interessante perché mostra un modo alternativo di dichiarare un nuovo tipo; in realtà ad essere pignoli non viene introdotto un nuovo tipo: la definizione di tipo che precede l'identificatore Complesso dichiara una struttura anonima e poi l'uso di typedef crea un alias per quel tipo struttura.

È possibile dichiarare tipi anonimi solo per i costrutti struct, union e enum e sono utilizzabili quasi esclusivamente nelle dichiarazioni (come nel caso di typedef oppure nelle dichiarazioni di variabili e costanti). La keyword typedef è utile per creare abbreviazioni per espressioni di tipo complesse, soprattutto quando l'espressione di tipo coinvolge puntatori e funzioni.

Sottoprogrammi e funzioni

Come ogni moderno linguaggio, sia il C che il C++ consentono di dichiarare sottoprogrammi che possono essere invocati nel corso dell'esecuzione di una sequenza di istruzioni a partire da una sequenza principale (il corpo del programma). Nel caso del C e del C++ questi sottoprogrammi sono chiamati funzioni e sono simili alle funzioni del Pascal. Anche il corpo del programma è modellato tramite una funzione il cui nome deve essere sempre main (vedi esempio “Dichiarazioni”).

Dichiarazione e chiamata di una funzione

Una funzione C/C++, analogamente ad una funzione Pascal, è caratterizzata da un nome che la distingue univocamente nel suo scope (le regole di visibilità di una funzione sono analoghe a quelle viste per le variabili), da un insieme (eventualmente vuoto) di argomenti (parametri della funzione) separati da virgole, e eventualmente il tipo del valore ritornato:

```
// ecco una funzione che riceve due interi
// e restituisce un altro intero
int Sum(int a, int b);
```

Gli argomenti presi da una funzione sono quelli racchiusi tra le parentesi tonde, si noti che il tipo dell'argomento deve essere specificato singolarmente per ogni argomento anche quando più argomenti hanno lo stesso tipo; la seguente dichiarazione è pertanto errata:

```
int Sum2(int a, b);   // ERRORE!
```

Il tipo del valore restituito dalla funzione deve essere specificato prima del nome della funzione e se omesso si sottintende int; se una funzione non ritorna alcun valore va dichiarata void, come mostra il seguente esempio:

```
// ecco una funzione che non
// ritorna alcun valore
void Foo(char a, float b);
```

Non è necessario che una funzione abbia dei parametri, in questo caso basta non specificarne oppure indicarlo esplicitamente:

```
// funzione che non riceve parametri
// e restituisce un int (default)
Funny();

// oppure
Funny2(void);
```

Il primo esempio vale solo per il C++, in C non specificare alcun argomento equivale a dire "Qualsiasi numero e tipo di argomenti"; il secondo metodo invece è valido in entrambi i linguaggi, in questo caso void assume il significato "Nessun argomento".

Anche in C++ è possibile avere funzioni con numero e tipo di argomenti non specificato:

```
void Esempio1(...);
void Esempio2(int Args, ...);
```

Il primo esempio mostra come dichiarare una funzione che prende un numero imprecisato (eventualmente 0) di parametri; il secondo esempio invece mostra come dichiarare funzioni che prendono almeno qualche parametro, in questo caso bisogna prima specificare tutti i parametri necessari e poi mettere ... per indicare eventuali altri parametri.

Quelli che abbiamo visto finora comunque non sono definizioni di funzioni, ma solo dichiarazioni, o per utilizzare un termine proprio del C++, prototipi di funzioni.

I prototipi di funzione, assenti nel C, sono stati introdotti nel C++ per informare il compilatore dell'esistenza di una certa funzione e consentire un maggior controllo sulle chiamate di funzione al fine di identificare errori di tipo (e non solo) e sono utilizzati soprattutto all'interno dei file header per la suddivisione di grossi programmi in più file e la realizzazione di librerie di funzioni, infine nei prototipi non è necessario indicare il nome degli argomenti della funzione:

```
// la funzione Sum vista sopra poteva
// essere dichiarata anche così:
int Sum(int, int);
```

Per implementare (definire) una funzione occorre ripetere il prototipo, specificando il nome degli argomenti (necessario per poter riferire ad essi, ma non obbligatorio se l'argomento non viene utilizzato), seguito da una sequenza di istruzioni racchiusa tra parentesi graffe:

```
int Sum(int x, int y) {
  return x+y;
}
```

La funzione Sum è costituita da una sola istruzione che calcola la somma degli argomenti e restituisce tramite la keyword return il risultato di tale operazione. Inoltre, benché non evidente dall'esempio, la keyword return provoca l'immediata terminazione della funzione; ecco un esempio non del tutto corretto, che però mostra il comportamento di return:

```
// calcola il quoziente di due numeri
int Div(int a, int b) {
  if (b==0) return "errore"
  return a/b;
}
```

Se il divisore è 0, la prima istruzione return restituisce (erroneamente) una stringa (anziché un intero) e provoca la terminazione della funzione, le successive istruzioni della funzione quindi non verrebbero eseguite.

Concludiamo questo paragrafo con alcune considerazioni:

- La definizione di una funzione non deve essere seguita da ; (punto e virgola), ciò tra l'altro consente di distinguere facilmente tra prototipo (dichiarazione) e definizione di funzione: un prototipo lo si riconosce perché la lista di argomenti di una funzione è seguita da ; (punto e virgola), mentre in una definizione la lista di argomenti è seguita da { (parentesi graffa aperta);

- Ogni funzione dichiarata non void deve restituire un valore, ne segue che da qualche parte nel corpo della funzione deve esserci una istruzione return con un qualche argomento (il valore restituito), in caso contrario viene segnalato un errore; analogamente l'uso di return in una funzione void costituisce un errore, salvo che la keyword sia utilizzata senza argomenti (provocando così solo la terminazione della funzione);

- La definizione di una funzione è anche una dichiarazione per quella funzione e all'interno del file che definisce la funzione non è obbligatorio scrivere il prototipo di quella funzione, vedremo meglio l'importanza dei prototipi più avanti;

- Non è possibile dichiarare una funzione all'interno del corpo di un'altra funzione.

Ecco ancora qualche esempio relativo alla seconda nota:

```
int Sum(int a, int b) {
```

```
  a + b;
}               // ERRORE! Nessun valore restituito.

int Sum(int a, int b) {
  return;
}               // ERRORE! Nessun valore restituito.

int Sum(int a, int b) {
  return a + b;
}               // OK!

void Sleep(int a) {
  for(int i=0; i<a; ++i) {};
}               // OK!

void Sleep(int Delay) {
  for(int i=0; i<a; ++i) {};
  return;
}               // OK!
```

La chiamata di una funzione può essere eseguita solo nell'ambito dello scope in cui appare la sua dichiarazione (come già detto le regole di scoping per le dichiarazioni di funzioni sono identiche a quelle per le variabili) specificando il valore assunto da ciascun parametro formale:

```
void Sleep(int Delay);  // definita da qualche parte
int Sum(int a, int b);  // definita da qualche parte

void main(void) {
  int X = 5;
  int Y = 7;
  int Result = 0;

  /* ... */
  Sleep(X);
  Result = Sum(X, Y);
  Sum(X, 8);              // Ok!
  Result = Sleep(1000);  // Errore!
}
```

La prima e l'ultima chiamata di funzione mostrano come le funzioni void (nel nostro caso Sleep) siano identiche alle procedure Pascal, in particolare l'ultima istruzione è un errore poiché Sleep non restituisce alcun valore.

La seconda chiamata di funzione (la prima di Sum) mostra come recuperare il valore restituito dalla funzione (esattamente come in Pascal). La chiamata successiva invece potrebbe sembrare un errore, in realtà si tratta di una chiamata lecita, semplicemente il valore tornato da Sum viene scartato; l'unico motivo per scartare il risultato dell'invocazione di una funzione è quello di sfruttare eventuali effetti laterali di tale chiamata.

Passaggio di parametri e argomenti di default

I parametri di una funzione si comportano all'interno del corpo della funzione come delle variabili locali e possono quindi essere usati anche a sinistra di un assegnamento (per quanto riguarda le variabili locali ad una funzione, si rimanda al capitolo III, paragrafo 3):

```
void Assign(int a, int b) { a = b;      // Tutto OK, operazione lecita!
}
```

tuttavia qualsiasi modifica ai parametri formali (quelli cioè che compaiono nella definizione, nel nostro caso a e b) non si riflette (per quanto visto finora) automaticamente sui parametri attuali (quelli effettivamente usati in una chiamata della funzione):

```
#include <iostream.h>

void Assign(int a, int b) {
  a = b;
}

void main() {
  int X = 5;
  int Y = 10;

  cout << "X = " << X << endl;
  cout << "Y = " << Y << endl;

  // Chiamata della funzione Assign
  // con parametri attuali X e Y
  Assign(X, Y);

  cout << "X = " << X << endl;
  cout << "Y = " << Y << endl;
}
```

L'esempio appena visto è perfettamente funzionante e se eseguito mostrerebbe come la funzione Assign, pur eseguendo una modifica ai suoi parametri formali, non modifichi i parametri attuali.

Questo comportamento è perfettamente corretto in quanto i parametri attuali vengono passati per valore: ad ogni chiamata della funzione viene cioè creata una copia di ogni parametro locale alla funzione stessa; tali copie vengono distrutte quando la chiamata della funzione termina ed il loro contenuto non viene copiato nelle eventuali variabili usate come parametri attuali.

In alcuni casi tuttavia può essere necessario fare in modo che la funzione possa modificare i suoi parametri attuali, in questo caso è necessario passare non una copia, ma un *riferimento* o un *puntatore* e agire su questo per modificare una variabile non locale alla funzione. Per adesso non considereremo queste due possibilità, ma rimanderemo la cosa al capitolo successivo non appena avremo parlato di puntatori e reference.

A volte siamo interessati a funzioni il cui comportamento è pienamente definito anche quando in una chiamata non tutti i parametri sono specificati, vogliamo cioè essere in grado di avere degli argomenti che assumano un valore di default se per essi non viene specificato alcun valore all'atto della chiamata. Ecco come fare:

```
int Sum (int a = 0, int b = 0) {
  return a+b;
}
```

Quella che abbiamo appena visto è la definizione della funzione Sum ai cui argomenti sono stati associati dei valori di default (in questo caso 0 per entrambi gli argomenti), ora se la funzione Sum viene chiamata senza specificare il valore di a e/o b il compilatore genera una chiamata a Sum sostituendo il valore di default (0) al parametro non specificato.

Una funzione può avere più argomenti di default, ma le regole del C++ impongono che tali argomenti siano specificati alla fine della lista dei parametri formali nella dichiarazione della funzione:

```
void Foo(int a, char b = 'a') {
  /* ... */
}       // Ok!

void Foo2(int a, int c = 4, float f) {
  /* ... */
}       // Errore!

void Foo3(int a, float f, int c = 4) {
```

```
  /* ... */
}       // Ok!
```

La dichiarazione di Foo2 è errata perché una volta che è stato specificato un argomento con valore di default, tutti gli argomenti seguenti (in questo caso f) devono possedere un valore di default; l'ultima definizione mostra come si sarebbe dovuto definire Foo2 per non ottenere errori.

La risoluzione di una chiamata di una funzione con argomenti di default naturalmente differisce da quella di una funzione senza argomenti di default in quanto sono necessari un numero di controlli maggiori:

sostanzialmente se nella chiamata per ogni parametro formale è specificato un parametro attuale, allora il valore di ogni parametro attuale viene copiato nel corrispondente parametro formale sovrascrivendo eventuali valori di default; se invece qualche parametro non viene specificato, quelli forniti specificano il valore dei parametri formali secondo la loro posizione e per i rimanenti parametri formali viene utilizzato il valore di default specificato (se nessun valore di default è stato specificato, viene generato un errore):

// riferendo alle precedenti definizioni:

```
Foo(1, 'b');        // chiama Foo con argomenti 1 e 'b'
Foo(0);             // chiama Foo con argomenti 0 e 'a'
Foo('c');           // ?????
Foo3(0);            // Errore, mancano parametri!
Foo3(1, 0.0);       // chiama Foo3(1, 0.0, 4)
Foo3(1, 1.4, 5);    // chiama Foo3(1, 1.4, 5)
```

Degli esempi appena fatti, il quarto, Foo3(0), è un errore poiché non viene specificato il valore per il secondo argomento della funzione (che non possiede un valore di default); è invece interessante il terzo (Foo('c');): apparentemente potrebbe sembrare un errore, in realtà quello che il compilatore fa è convertire il parametro attuale 'c' di tipo char in uno di tipo int e chiamare la funzione sostituendo al primo parametro il risultato della conversione di 'c' al tipo int.
La conversione di tipo sarà oggetto di una apposita appendice.

La funzione main()

Come già precedentemente accennato, anche il corpo di un programma C/C++ è modellato come una funzione. Tale funzione ha un nome predefinito, main, e viene invocata automaticamente dal sistema quando il programma viene eseguito.

Per adesso possiamo dire che la struttura di un programma è sostanzialmente la seguente:

```
< Dichiarazioni globali e dichiarazioni di funzioni >

int main(int argc, char* argv[ ]) {
  < Corpo della funzione >
}
```

Un programma è dunque costituito da un insieme (eventualmente vuoto) di dichiarazioni globali di costanti, variabili... e di dichiarazioni di funzioni (che non possono essere dichiarate localmente ad altre funzioni), infine il corpo del programma è costituito dalla funzione main, il cui prototipo per esteso è mostrato nello schema riportato sopra.

Nello schema main ritorna un valore di tipo int (che generalmente è utilizzato per comunicare al sistema operativo la causa della terminazione), ma può essere dichiarata void o in teoria tornare un tipo qualsiasi.
Inoltre main può accettare opzionalmente due parametri: il primo è di tipo int e indica il numero di parametri presenti sulla riga di comando attraverso cui è stato eseguito il programma; il secondo parametro (si comprenderà in seguito) è un array di stringhe terminate da zero (puntatori a caratteri) contenente i parametri, il primo dei quali (argv[0]) è il nome del programma come riportato sulla riga di comando.

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

```
void main(int argc, char * argv[ ]) {
  cout << "Riga di comando:" << endl;
  cout << "  Comando = " << argv[0] << endl;
  for(int i=1; i<argc; ++i)
    cout << "  Parametro " << i << " = " << argv[i] << endl;
}
```

Il precedente esempio mostra come accedere ai parametri passati sulla riga di comando; si provi a compilare e ad eseguirlo specificando un numero qualsiasi di parametri, l'output dovrebbe essere simile a:

```
> test a b c d        // questa è la riga di comando

Riga di comando: TEST.EXE
  Parametro 1 = a
  Parametro 2 = b
  Parametro 3 = c
  Parametro 4 = d
```

Funzioni inline

Le funzioni consentono di scomporre in più parti un grosso programma facilitandone la realizzazione (e anche la manutenzione), tuttavia spesso si è indotti a rinunciare a tale beneficio perché l'overhead imposto dalla chiamata di una funzione è tale da sconsigliare la realizzazione di piccole funzioni.

Le possibili soluzioni in C erano due:

1. Rinunciare alle funzioni piccole, tendendo a scrivere solo poche funzioni corpose;
2. Ricorrere alle macro;

La prima in realtà è una pseudo-soluzione e porta spesso a programmi difficili da capire e mantenere perché in pratica rinuncia ai benefici delle funzioni; la seconda soluzione invece potrebbe andare bene in C, ma non in C++: una macro può essere vista come una funzione il cui corpo è sostituito (espanso) dal preprocessore in luogo di ogni chiamata.

Il problema principale è che questo sistema rende difficoltoso se non impossibile ogni controllo statico di tipo; in C tutto sommato ciò non costituisce un grave problema perché il compilatore C non esegue controlli di tipo, tuttavia il C++ è un linguaggio fortemente tipizzato e l'uso di macro costituisce un grave ostacolo a tali controlli.

Per non rinunciare ai vantaggi forniti dalle (piccole) funzioni e a quelli forniti da un controllo statico dei tipi, sono state introdotte nel C++ le funzioni inline.
Quando una funzione viene definita inline il compilatore ne memorizza il corpo e, quando incontra una chiamata a tale funzione, semplicemente sostituisce alla chiamata della funzione il corpo; tutto ciò consente di evitare l'overhead della chiamata e, dato che la cosa è gestita dal compilatore, permette di eseguire tutti i controlli statici di tipo.

Se si desidera che una funzione sia espansa inline dal compilatore, occorre definirla esplicitamente inline:

```
inline int Sum(int a, int b) {
  return a + b;
}
```

La keyword inline informa il compilatore che si desidera che la funzione Sum sia espansa inline ad ogni chiamata; tuttavia ciò non vuol dire che la cosa sia sempre possibile: molti compilatori non sono in grado di espandere inline qualsiasi funzione, tipicamente le funzioni ricorsive sono molto difficili da trattare e il mio compilatore non riesce ad esempio a espandere funzioni contenenti cicli. In questi casi viene generata una normale chiamata di funzione e al più si viene avvisati che la funzione non può essere espansa inline.

Si osservi che, per come sono trattate le funzioni inline, non ha senso utilizzare la keyword inline in un prototipo di funzione perché il compilatore necessita del codice contenuto nel corpo della funzione:

```
inline int Sum(int a, int b);

int Sum(int a, int b) {
  return a + b;
}
```

In questo caso non viene generato alcun errore, ma la parola chiave inline specificata nel prototipo viene del tutto ignorata; perché abbia effetto inline deve essere specificata nella definizione della funzione:

```
int Sum(int a, int b);

inline int Sum(int a, int b) {
  return a + b;
}                    // Ora è tutto ok!
```

Un'altra cosa da tenere presente è che il codice che costituisce una funzione inline deve essere disponibile prima di ogni uso della funzione, altrimenti il compilatore non è in grado di espanderla (non sempre almeno!). Una funzione ordinaria può essere usata anche prima della sua definizione, poiché è il linker che si occupa di risolvere i riferimenti (il linker del C++ lavora in due passate); nel caso delle funzioni inline, poiché il lavoro è svolto dal compilatore (che lavora in una passata), non è possibile risolvere correttamente il riferimento.

Una importante conseguenza di tale limitazione è che una funzione può essere inline solo nell'ambito del file in cui è definita, se un file riferisce ad una funzione definita inline in un altro file (come, lo vedremo più avanti), in questo file (il primo) la funzione non potrà essere espansa inline; esistono comunque delle soluzioni al problema.

Le funzioni inline consentono quindi di conservare i benefici delle funzioni anche in quei casi in cui le prestazioni sono fondamentali, bisogna però valutare attentamente la necessità di rendere inline una funzione, un abuso potrebbe portare a programmi difficili da compilare (perché è necessaria molta memoria) e voluminosi in termini di dimensioni del file eseguibile.

Overloading delle funzioni

Il termine overloading (da to overload) significa sovraccaricamento e nel contesto del C++ overloading delle funzioni indica la possibilità di attribuire allo stesso nome di funzione più significati. Attribuire più significati significa fare in modo che lo stesso nome di funzione sia in effetti utilizzato per più funzioni contemporaneamente.

Un esempio di overloading ci viene dalla matematica, dove con spesso utilizziamo lo stesso nome di funzione con significati diversi senza starci a pensare troppo, ad esempio + è usato sia per indicare la somma sui naturali che quella sui reali...

Ritorniamo per un attimo alla nostra funzione Sum; per come è stata definita, Sum funziona solo sugli interi e non è possibile utilizzarla sui float. Quello che vogliamo è riutilizzare lo stesso nome, attribuendogli un significato diverso e lasciando al compilatore il compito di capire quale versione della funzione va utilizzata di volta in volta. Per fare ciò basta definire più volte la stessa funzione:

```
int Sum(int a, int b);          // per sommare due interi...
float Sum(float a, float b);      // per sommare due float...

float Sum(float a, int b);        // per la somma di un
float Sum(int a, float b);        // float e un intero
```

Nel nostro esempio ci siamo limitati solo a dichiarare più volte la funzione Sum, ogni volta con un significato diverso (uno per ogni possibile caso di somma in cui possono essere coinvolti, anche contemporaneamente, interi e reali); è chiaro che

poi da qualche parte deve esserci una definizione per ciascun prototipo (nel nostro caso tutte le definizioni sono identiche a quella già vista, cambia solo l'intestazione della funzione).

In alcune vecchie versioni del C++ l'intenzione di sovraccaricare una funzione doveva essere esplicitamente comunicata al compilatore tramite la keyword overload:

```
overload Sum;          // ora si può sovraccaricare Sum:

int Sum(int a, int b);            // per sommare due interi...
float Sum(float a, float b);      // per sommare due float...

float Sum(float a, int b);        // per la somma di un
float Sum(int a, float b);        // float e un intero
```

Comunque si tratta di una pratica obsoleta che non va più utilizzata (se possibile!). Le funzioni sovraccaricate si utilizzano esattamente come le normali funzioni:

```
#include <iostream.h>

int a = 5;
int Y = 10;
float f = 9.5;
float r = 0.5;

cout << "Sum utilizzata su due interi" << endl;
cout << Sum(a, Y) << endl;

cout << "Sum utilizzata su due float" << endl;
cout << Sum(f, r) << endl;

cout << "Sum utilizzata su un intero e un float" << endl;
cout << Sum(a, f) << endl;

cout << "Sum utilizzata su un float e un intero" << endl;
cout << Sum(r, f) << endl;
```

È il compilatore che decide quale versione di Sum utilizzare, in base ai parametri forniti; infatti è possibile eseguire l'overloading di una funzione solo a condizione che la nuova versione della funzione differisca dalle precedenti almeno nei tipi dei parametri (o che questi siano forniti in un ordine diverso, come mostrano le ultime due definizioni di Sum):

```
void Foo(int a, float f);
int Foo(int a, float f);        // Errore!
int Foo(float f, int a);        // Ok!
char Foo();                     // Ok!
char Foo(...);                  // OK!
```

La seconda dichiarazione è errata perché, per scegliere tra la prima e la seconda versione della funzione, il compilatore si basa unicamente sui tipi dei parametri che nel nostro caso coincidono; la soluzione è mostrata con la terza dichiarazione, ora il compilatore è in grado di distinguere perché il primo parametro anziché essere un int è un float. Infine le ultime due dichiarazioni non sono in conflitto per via delle regole che il compilatore segue per scegliere quale funzione applicare; in linea di massima e secondo la loro priorità:

- Match esatto: se esiste una versione della funzione che richiede esattamente quel tipo di parametri (i parametri vengono considerati a uno a uno secondo l'ordine in cui compaiono) o al più conversioni banali (tranne da T* a const T* o a volatile T*, oppure da T& a const T& o a volatile T&);

- Mach con promozione: si utilizza (se esiste) una versione della funzione che richieda al più promozioni di tipo (ad esempio da int a long int, oppure da float a double);

- Mach con conversioni standard: si utilizza (se esiste) una versione della funzione che richieda al più conversioni di tipo standard (ad esempio da int a unsigned int);

- Match con conversioni definite dall'utente: si tenta un matching con una definizione (se esiste), cercando di utilizzare conversioni di tipo definite dal programmatore;

- Match con ellissi: si esegue un matching utilizzando (se esiste) una versione della funzione che accetti un qualsiasi numero e tipo di parametri (cioè funzioni nel cui prototipo è stato utilizzato il simbolo ...);

Se nessuna di queste regole può essere applicata, si genera un errore (funzione non definita!).

La piena comprensione di queste regole richiede la conoscenza del concetto di conversione di tipo per il quale si rimanda ad una apposita appendice; si accenna inoltre ai tipi puntatore e reference che saranno trattati nel prossimo capitolo, infine si fa riferimento alla keyword volatile. Tale keyword serve ad informare il compilatore che una certa variabile cambia valore in modo aleatorio e che di conseguenza il suo valore va riletto ogni volta che il valore della variabile è richiesto:

```
volatile int ComPort;
```

La precedente definizione dice al compilatore che il valore di ComPort è fuori dal controllo del programma (ad esempio perché la variabile è associata ad un qualche registro di un dispositivo di I/O).

Il concetto di overloading di funzioni si estende anche agli operatori del linguaggio, ma questo è un argomento che riprenderemo più avanti.

Puntatori e reference

Oltre ai tipi primitivi visti precedentemente, esistono altri due tipi fondamentali usati solitamente in combinazione con altri tipi (sia primitivi che non): puntatori e reference.

L'argomento di cui ora parleremo potrà risultare particolarmente complesso, soprattutto per coloro che non hanno mai avuto a che fare con i puntatori: alcuni linguaggi non forniscono affatto i puntatori (come il Basic, almeno in alcune vecchie versioni), altri (Pascal) invece forniscono un buon supporto; tuttavia il C++ fa dei puntatori un punto di forza (se non il punto di forza) e fornisce un supporto ad essi persino superiore a quello fornito dal Pascal. È quindi caldamente consigliata una lettura attenta di quanto segue e sarebbe bene fare pratica con i puntatori non appena possibile.

Puntatori

I puntatori possono essere pensati come maniglie da applicare alle porte delle celle di memoria per poter accedere al loro contenuto sia in lettura che in scrittura, nella pratica una variabile di tipo puntatore contiene l'indirizzo di una locazione di memoria.
Vediamo alcuni esempi di dichiarazione di puntatori:

```
short * Puntatore1;
Persona * Puntatore3;
double * * Puntatore2;
int UnIntero = 5;
int * PuntatoreAInt = &UnIntero;
```

Il carattere * (asterisco) indica un puntatore, per cui le prime tre righe dichiarano rispettivamente un puntatore a short int, un puntatore a Persona e un puntatore a puntatore a double. La quinta riga dichiara un puntatore a int e ne esegue l'inizializzazione mediante l'operatore & (indirizzo di) che serve ad ottenere l'indirizzo della variabile (o di una costante o ancora di una funzione) il cui nome segue l'operatore.

Si osservi che un puntatore a un certo tipo può puntare solo a oggetti di quel tipo, non è possibile ad esempio assegnare l'indirizzo di una variabile di tipo float a un puntatore a char, o meglio in molti casi è possibile farlo, ma viene eseguita una coercizione (vedi appendice A):

```
float Reale = 1.1;
char * Puntatore = &Reale;       // errore !
```

È anche possibile assegnare ad un puntatore un valore particolare a indicare che il puntatore non punta a nulla:

```
Puntatore = 0;
```

In luogo di 0 i programmatori C usano la costante NULL, tuttavia l'uso di NULL comporta alcuni problemi di conversione di tipo; in C++ il valore 0 viene automaticamente convertito in un puntatore NULL di dimensione appropriata.

Nelle dichiarazioni di puntatori bisogna prestare attenzione a diversi dettagli che possono essere meglio apprezzati tramite esempi:

```
float * Reale, UnAltroReale;
int Intero = 10;
const int * Puntatore = &Intero;
int * const CostantePuntatore = &Intero;
const int * const CostantePuntatoreACostante = &Intero;
```

La prima dichiarazione contrariamente a quanto si potrebbe pensare non dichiara due puntatori a float, ma un puntatore a float (Reale) e una variabile di tipo float (UnAltroReale): * si applica solo al primo nome che lo segue e quindi il modo corretto di eseguire quelle dichiarazioni era

```
float * Reale, * UnAltroReale;
```

La terza riga mostra come dichiarare un puntatore a un intero costante, attenzione non un puntatore costante; la dichiarazione di un puntatore costante è mostrata nella penultima riga. Un puntatore a una costante consente l'accesso all'oggetto da esso puntato solo in lettura (ma ciò non implica che l'oggetto puntato sia effettivamente costante), mentre un puntatore costante è una costante di tipo puntatore (a ...), non è quindi possibile modificare l'indirizzo in essa contenuto e va inizializzato nella dichiarazione.
L'ultima riga mostra invece come combinare puntatori costanti e puntatori a costanti per ottenere costanti di tipo puntatore a costante (intera, nell'esempio).
Attenzione: anche const, se utilizzato per dichiarare una costante puntatore, si applica ad un solo nome (come *).
In alcuni casi è necessario avere puntatori generici, in questi casi il puntatore va dichiarato void:

```
void * Puntatore generico;
```

I puntatori void possono essere inizializzati come un qualsiasi altro puntatore tipizzato, e a differenza di questi ultimi possono puntare a qualsiasi oggetto senza riguardo al tipo o al fatto che siano costanti, variabili o funzioni; tuttavia non è possibile eseguire sui puntatori void alcune operazioni definite sui puntatori tipizzati.

Operazioni sui puntatori

Dal punto di vista dell'assegnamento, una variabile di tipo puntatore si comporta esattamente come una variabile di un qualsiasi altro tipo primitivo, basta tenere presente che il loro contenuto è un indirizzo di memoria:

```
int Pippo = 5, Topolino = 10;
char Pluto = 'P';
int * Minnie = &Pippo;
int * Basettoni;
void * Manetta;

// Esempi di assegnamento a puntatori:
Minnie = &Topolino;
Manetta = &Minnie;     // "Manetta" punta a "Minnie"
Basettoni = Minnie;    // "Basettoni" e "Minnie" puntano ora
                       // allo stesso oggetto
```

I primi due assegnamenti mostrano come assegnare esplicitamente l'indirizzo di un oggetto ad un puntatore: nel primo caso la variabile Minnie viene fatta puntare alla variabile Topolino, nel secondo caso al puntatore void Manetta si assegna l'indirizzo della variabile Minnie (e non quello della variabile Topolino); per assegnare il contenuto di un puntatore ad un altro puntatore non bisogna utilizzare l'operatore &, basta considerare la variabile puntatore come una variabile di un qualsiasi altro tipo, come mostrato nell'ultimo assegnamento.

L'operazione più importante che viene eseguita sui puntatori e quella di dereferenziazione o indirezione che permette l'accesso all'oggetto puntato; l'operazione viene eseguita tramite l'operatore di dereferenziazione * posto prefisso al puntatore, come mostra il seguente esempio:

```
short * P;
short int Val = 5;

P = &Val;     // P punta a Val (cioè Val e *P sono
              // lo stesso oggetto);
cout << "Ora P punta a Val:" << endl;
cout << "*P = " << *P << endl;
cout << "Val = " << Val << endl << endl;

*P = -10;     // Modifica l'oggetto puntato da P
cout << "Val è stata modificata tramite P:" << endl;
cout << "*P = " << *P << endl;
```

```
cout << "Val = " << Val << endl << endl;

Val = 30;
cout << "La modifica su Val si riflette su *P:" << endl;
cout << "*P = " << *P << endl;
cout << "Val = " << Val << endl << endl;
```

Il codice appena mostrato fa sì che il puntatore P riferisca alla variabile Val, ed esegue una serie di assegnamenti sia alla variabile che all'oggetto puntato da P mostrandone gli effetti. L'operatore * prefisso ad un puntatore seleziona l'oggetto puntato dal puntatore così che *P utilizzato come operando in una espressione produce l'oggetto puntato da P.
Ecco quale sarebbe l'output del precedente frammento di codice se eseguito:

```
Ora P punta a Val:
*P = 5
Val = 5

Val è stata modificata tramite P:
*P = -10
Val = -10

La modifica su Val si riflette su *P:
*P = 30
Val = 30
```

L'operazione di dereferenziazione può essere eseguita su un qualsiasi puntatore a condizione che questo non sia stato dichiarato void. In generale infatti non è possibile stabilire il tipo dell'oggetto puntato da un puntatore void e il compilatore non sarebbe in grado di trattare tale oggetto.
Ovviamente dereferenziare un puntatore a cui non è stato assegnato un indirizzo è un errore a run time; quando possibile comunque il compilatore segnala eventuali tentativi di dereferenziare puntatori che potrebbero non essere stati inizializzati tramite una warning.
Per i puntatori a strutture (o unioni) è possibile utilizzare un altro operatore di dereferenziazione che consente in un colpo solo di dereferenziare il puntatore e selezionare il campo desiderato:

```
Persona Pippo;
Persona * Puntatore = &Pippo;

Puntatore->Eta = 40;
cout << "Pippo.Eta = " << Puntatore->Eta << endl;
```

La terza riga dell'esempio dereferenzia Puntatore e contemporaneamente seleziona il campo Eta (il tutto tramite l'operatore ->) per eseguire un assegnamento a quest'ultimo. Nell'ultima riga viene mostrato come utilizzare -> per ottenere il valore di un campo dell'oggetto puntato.
Sui puntatori è definita una speciale aritmetica composta da somma e sottrazione. Se P è un puntatore di tipo T, sommare 1 a P significa puntare all'elemento successivo di un ipotetico array di tipo T cui P è immaginato puntare; analogamente sottrarre 1 significa puntare all'elemento precedente.
È possibile anche sottrarre da un puntatore un altro puntatore (dello stesso tipo), in questo caso il risultato è il numero di elementi che separano i due puntatori:

```
int Array[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int * P1 = &Array[5];
int * P2 = &Array[9];

cout << P1 - P2 << endl;    // visualizza 4
cout << *P1 << endl;        // visualizza 5
P1+=3;                      // equivale a P1 = P1 + 3;
cout << *P1 << endl;        // visualizza 8
cout << *P2 << endl;        // visualizza 9
P2-=5;                      // equivale a P2 = P2 - 5;
cout << *P2 << endl;        // visualizza 4
```

Sui puntatori sono anche definiti gli usuali operatori relazionali:

```
<           minore di
>           maggiore di
<=          minore o uguale
>=          maggiore o uguale
==          uguale a
!=          diverso da
```

Puntatori vs Array

Esiste una stretta somiglianza tra puntatori e array dovuta alla possibilità di dereferenziare un puntatore nello stesso modo in cui si seleziona l'elemento di un array e al fatto che lo stesso nome di un array è di fatto un puntatore al primo elemento dell'array:

```
int Array[ ] = { 1, 2, 3, 4, 5 };
int * Ptr = Array;          // equivalente a Ptr = &Array[0];

cout << Ptr[3] << endl;     // Ptr[3] equivale a *(Ptr + 3);
Ptr[4] = 7;                 // equivalente a *(Ptr + 4) = 7;
```

La somiglianza diviene maggiore quando si confrontano array e puntatori di caratteri:

```
char Array[ ] = "Una stringa"
char * Ptr = "Una stringa"

// la seguente riga stampa tutte e due le stringhe
// si osservi che non è necessario dereferenziare un char *
// (a differenza degli altri tipi di puntatori)

cout << Array << " == " << Ptr << endl;

// in questo modo, invece, si stampa solo un carattere:
// la dereferenziazione di un char * o l'indicizzazione
// di un array causano la visualizzazione di un solo carattere
// perché in effetti si passa all'oggetto cout non un puntatore a
// char, ma un oggetto di tipo char (che cout tratta giustamente
// in modi diversi)

cout << Array[5] << " == " << Ptr[5] << endl;
cout << *Ptr << endl;
```

In C++ le dichiarazioni char Array[ ] = "Una stringa" e char * Ptr = "Una stringa" hanno lo stesso effetto, entrambe creano una stringa (terminata dal carattere nullo) il cui indirizzo è posto rispettivamente in Array e in Ptr, e come mostra l'esempio un char * può essere utilizzato esattamente come un array di caratteri.
Esistono tuttavia profonde differenze tra puntatori e array: un puntatore è una variabile a cui si possono applicare le operazioni viste sopra e che può essere usato come un array, ma non è vero il viceversa: il nome di un array non è un puntatore a cui è possibile assegnare un nuovo valore (non è cioè modificabile). Ecco un esempio:

```
char Array[ ] = "Una stringa";
char * Ptr = "Una stringa";

Array[3] = 'a';  // Ok!
Ptr[7] = 'b';    // Ok!
Ptr = Array;     // Ok!
Ptr++;           // Ok!
Array++;         // errore, tentativo di assegnamento!
```

In definitiva un puntatore è più flessibile di quanto non lo sia un array, anche se a costo di un maggiore overhead.

Uso dei puntatori

I puntatori sono utilizzati sostanzialmente per tre scopi:

- Realizzazione di strutture dati dinamiche (es. liste linkate);
- Realizzazione di funzioni con effetti laterali sui parametri attuali;
- Ottimizzare il passaggio di parametri di grosse dimensioni.

Il primo caso è tipico di applicazioni che necessitano di strutture dati che si espandano e si comprimano dinamicamente durante l'esecuzione, ad esempio un editor di testo.
Ecco un esempio:

```
#include <iostream.h>

// Una lista è composta da tante celle linkate
// tra di loro; ogni cella contiene un valore
// e un puntatore alla cella successiva.
struct TCell {
  float AFloat;     // per memorizzare un valore
  TCell * Next;     // puntatore alla cella successiva
};

// La lista viene realizzata tramite questa
// struttura contenente il numero di celle
// della lista e il puntatore alla prima cella
struct TList {
  unsigned Size;    // Dimensione lista
  TCell * First;    // Puntatore al primo elemento
};

void main() {
  TList List;           // Dichiara una lista
  List.Size = 0;        // inizialmente vuota
  List.First = NULL;
  int FloatToRead;
  cout << "Quanti valori vuoi immettere? " ;
  cin >> FloatToRead;
  cout << endl;

  // questo ciclo richiede e memorizza
  // nella lista valori reali
  for(int i=0; i<FloatToRead; ++i) {
    TCell * Temp = List.First;
    cout << "Creazione di una nuova cella..." << endl;
    List.First = new TCell;         // new vuole il tipo di
                                    // elemento da creare
    cout << "Immettere un valore reale " ;
    cin >> List.First->AFloat;
    cout << endl;
    List.First->Next = Temp;      // aggiunge la cella in
                                  // testa alla lista
    ++List.Size;                  // Aggiorna la dimensione
  }                               // della lista

  // il seguente ciclo calcola la somma
  // dei valori contenuti nella lista;
  // via via che recupera i valori,
  // distrugge le relative celle
```

```
  float Total = 0.0;
  for(unsigned j=0; j<List.Size; ++j) {
    Total += List.First->AFloat;
    TCell * Temp = List.First;          // estrae la cella in
    List.First = List.First->Next;      // testa alla lista
    cout << "Distruzione della cella in testa alla lista..." << endl;
    delete Temp;                        // distrugge la cella
  }                                     // estratta

  cout << "Totale = " << Total << endl;
}
```

L'esempio mostra come creare e distruggere oggetti dinamicamente.
Il programma memorizza in una lista un certo numero di valori reali, aggiungendo per ogni valore una nuova cella; in seguito li estrae uno ad uno, distruggendo le relative celle, e li somma restituendo il totale. Il codice è ampiamente commentato e non dovrebbe essere difficile capire come funziona.

La creazione di un nuovo oggetto avviene allocando un nuovo blocco di memoria (sufficientemente grande) dalla heap-memory, mentre la distruzione avviene deallocando tale blocco (che ritorna a far parte della heap-memory); l'allocazione viene eseguita tramite l'operatore *new* cui va specificato il tipo di oggetto da creare (per sapere quanta RAM allocare), la deallocazione avviene invece tramite l'operatore *delete*, che richiede come argomento un puntatore all'aggetto da deallocare (la quantità di RAM da deallocare viene calcolata automaticamente).

In alcuni casi è necessario allocare e deallocare interi array, in questi casi si ricorre agli operatori new [ ] e delete [ ]:

```
// alloca un array di 10 interi
int * ArrayOfInt = new int [10];

// ora eseguiamo la deallocazione
delete [] ArrayOfInt;
```

Si noti inoltre che gli oggetti allocati nella heap-memory non ubbidiscono alle regole di scoping statico valide per le variabili ordinarie (tuttavia i puntatori a tali oggetti sono sempre soggetti a tali regole), la loro creazione e distruzione è compito del programmatore.

Consideriamo ora il secondo uso che si fa dei puntatori.
Esso corrisponde a quello che in Pascal si chiama "passaggio di parametri per variabile" e consente la realizzazione di funzioni con effetti laterali sui parametri:

```
void Change(int * IntPtr) {
  *IntPtr = 5;
}
```

La funzione Change riceve come unico parametro un puntatore a int, ovvero un indirizzo di una cella di memoria; anche se l'indirizzo viene copiato in una locazione di memoria visibile solo alla funzione, la dereferenziazione di tale copia consente comunque la modifica dell'oggetto puntato:

```
int A = 10;

cout << " A = " << A << endl;
cout << " Chiamata della funzione Change(&A)... " << endl;
Change(&A);
cout << " Ora A = " << A << endl;
```

l'output che il precedente codice produce è:

```
A = 10
Chiamata della funzione Change(&A)...
Ora A = 5
```

Quello che nell'esempio accade è che la funzione Change riceve l'indirizzo della variabile A e tramite esso è in grado di agire sulla variabile stessa.
L'uso dei puntatori come parametri di funzione non è comunque utilizzato solo per consentire effetti laterali, spesso una funzione riceve parametri di dimensioni notevoli e l'operazione di copia del parametro attuale in un'area privata della funzione ha effetti deleteri sui tempi di esecuzione della funzione; in questi casi è molto più conveniente passare un puntatore:

```
void Func(BigParam parametro);      // funziona, ma è meglio
                                    // la seguente dichiarazione
void Func(const BigParam * parametro);
```

Il secondo prototipo è più efficiente perché evita l'overhead imposto dal passaggio per valore, inoltre l'uso di *const* previene ogni tentativo di modificare l'oggetto puntato e allo stesso tempo comunica al programmatore che usa la funzione che non esiste tale rischio.
Infine quando l'argomento di una funzione è un array, il compilatore passa sempre un puntatore, mai una copia dell'argomento; in questo caso inoltre l'unico modo che ha la funzione per conoscere la dimensione dell'array è quello di ricorrere ad un parametro aggiuntivo, esattamente come accade con la funzione main (vedi capitolo precedente).
Ovviamente una funzione può restituire un tipo puntatore, in questo caso bisogna però prestare attenzione a ciò che si restituisce, non è raro infatti che un principiante scriva qualcosa del tipo:

```
int * Sum(int a, int b) {
  int Result = a + b;
  return &Result;
}
```

Apparentemente è tutto corretto e un compilatore potrebbe anche non segnalare niente, tuttavia esiste un grave errore: si ritorna l'indirizzo di una variabile locale. L'errore è dovuto al fatto che la variabile locale viene distrutta quando la funzione termina e riferire ad essa diviene quindi illecito. Una soluzione corretta sarebbe stata quella di allocare Result nello heap e restituire l'indirizzo di tale oggetto (in questo caso è cura di chi usa la funzione occuparsi della eventuale deallocazione dell'oggetto).

Reference

I reference (riferimenti) sono un costrutto a metà tra puntatori e variabili: come i puntatori essi sono contenitori di indirizzi, ma non è necessario dereferenziarli per accedere all'oggetto puntato (si usano come se fossero variabili). In pratica possiamo vedere i reference come un meccanismo per creare alias di variabili, anche se in effetti questa è una definizione non del tutto esatta.

Così come un puntatore viene indicato nelle dichiarazioni dal simbolo *, così un reference viene indicato dal simbolo &:

```
int Var = 5;
float f = 0.5;

int * IntPtr = &Var;
int & IntRef = Var;      // nei reference non è necessario
float & FloatRef = f;    // usare & a destra di =
```

Le ultime due righe dichiarano rispettivamente un riferimento di tipo int e uno di tipo float che vengono subito inizializzati usando le due variabili dichiarate prima; un riferimento va inizializzato immediatamente, e dopo l'inizializzazione non può essere più cambiato, si noti che non è necessario utilizzare l'operatore & (indirizzo di) per eseguire l'inizializzazione. Dopo l'inizializzazione il riferimento potrà essere utilizzato in luogo della variabile cui è legato, utilizzare l'uno o l'altro sarà indifferente:

```
cout << "Var = " << Var << endl;
cout << "IntRef = " << IntRef << endl;
cout << "Assegnamento a IntRef..." << endl;
```

```
IntRef = 8;
cout << "Var = " << Var << endl;
cout << "IntRef = " << IntRef << endl;
cout << "Assegnamento a Var..." << endl;
Var = 15;
cout << "Var = " << Var << endl;
cout << "IntRef = " << IntRef << endl;
```

Ecco l'output del precedente codice:

```
Var = 5
IntRef = 5
Assegnamento a IntRef...
Var = 8
IntRef = 8;
Assegnamento a Var...
Var = 15
IntRef = 15
```

Dall'esempio si capisce perché, dopo l'inizializzazione, un riferimento non possa essere più associato ad un nuovo oggetto: ogni assegnamento al riferimento si traduce in un assegnamento all'oggetto riferito.
Un riferimento può essere inizializzato anche tramite un puntatore:

```
int * IntPtr = new int (5);
    // il valore tra parentesi specifica il valore cui
    // inizializzare l'oggetto allocato. Per adesso il
    // metodo funziona solo con i tipi primitivi.

int & IntRef = *IntPtr;
```

Si noti che il puntatore va dereferenziato, altrimenti si legherebbe il riferimento al puntatore (in questo caso l'uso del riferimento comporta implicitamente un conversione da int * a int).
Ovviamente il metodo può essere utilizzato anche con l'operatore new:

```
double & DoubleRef = *new Double;

// Ora si può accedere all'oggetto allocato
// tramite il riferimento.

// Di nuovo, è compito del programmatore
// distruggere l'oggetto creato con new
delete &DoubleRef;

// Si noti che va usato l'operatore &, per
// indicare l'intenzione di deallocare
// l'oggetto riferito, non il riferimento!
```

L'uso dei riferimenti per accedere a oggetti dinamici è sicuramente molto comodo perché è possibile uniformare tali oggetti alle comuni variabili, tuttavia è una pratica che bisognerebbe evitare perché può generare confusione e di conseguenza errori assai insidiosi.

Uso dei reference

Nel paragrafo precedente sono stati mostrati alcuni possibili usi dei riferimenti. In verità comunque i riferimenti sono stati introdotti nel C++ come ulteriore meccanismo di passaggio di parametri (per riferimento).
Una funzione che debba modificare i parametri attuali può ora essere dichiarata in due modi diversi:

```
void Esempio(Tipo * Parametro);
```

oppure in modo del tutto equivalente

```
void Esempio(Tipo & Parametro);
```

Naturalmente cambierebbe il modo in cui chiamare la funzione:

```
long double Var = 0.0;
long double * Ptr = &Var;

// nel primo caso avremmo
Esempio(&Var);           // oppure
Esempio(Ptr);

// nel caso di passaggio per riferimento
Esempio(Var);
```

In modo del tutto analogo a quanto visto con i puntatori è anche possibile ritornare un riferimento:

```
double & Esempio(float Param1, float Param2) {
  ...
  double * X = new double;
  ...
  return *X;
}
```

Puntatori e reference possono essere liberamente scambiati, non esiste differenza eccetto che non è necessario dereferenziare un riferimento.

Probabilmente vi starete chiedendo che motivo c'era dunque di introdurre questa caratteristica, dato che i puntatori erano già sufficienti? Il problema in effetti non nasce con le funzioni, ma con gli operatori; il C++ consente anche l'overloading degli operatori e sarebbe spiacevole dover scrivere qualcosa del tipo:

```
&A + &B
```

non si riuscirebbe a capire se si desidera sommare due indirizzi oppure i due oggetti (che potrebbero essere troppo grossi per passarli per valore). I riferimenti invece risolvono il problema eliminando ogni possibile ambiguità e consentendo una sintassi più chiara.

Puntatori vs Reference

Visto che per le funzioni è possibile scegliere tra puntatori e riferimenti, come decidere quale metodo scegliere? I riferimenti hanno un vantaggio sui puntatori, dato che nella chiamata di una funzione non c'è differenza tra passaggio per valore o per riferimento, è possibile cambiare meccanismo senza dover modificare né il codice che chiama la funzione né il corpo della funzione stessa.

Tuttavia il meccanismo dei reference nasconde all'utente il fatto che si passa un indirizzo e non una copia, e ciò può creare grossi problemi in fase di debugging.
Quando è necessario passare un indirizzo è quindi meglio usare i puntatori, che consentono un maggior controllo sugli accessi (tramite la keyword const) e rendono esplicito il modo in cui il parametro viene passato.

Esiste comunque una eccezione nel caso dei tipi definiti dall'utente tramite il meccanismo delle classi. In questo caso vedremo che l'incapsulamento garantisce che l'oggetto passato possa essere modificato solo da particolari funzioni (funzioni membro e funzioni amiche), e quindi usare i riferimenti è più conveniente perché non è necessario dereferenziarli, migliorando così la chiarezza del codice; le funzioni membro e le funzioni amiche, in quanto tali, sono invece autorizzate a modificare l'oggetto e quindi quando vengono usate l'utente sa già che potrebbero esserci effetti laterali.

Linkage e file Header

Quello che è stato visto finora costituisce sostanzialmente il sottoinsieme C del C++ (salvo l'overloading, i reference e altre piccole aggiunte), è tuttavia sufficiente per poter realizzare un qualsiasi programma.
A questo punto, prima di proseguire, è necessario soffermarci per esaminare il funzionamento del linker C++ e vedere come organizzare un grosso progetto in più file separati.

Linkage

Abbiamo già visto che ad ogni identificatore è associato uno scope e una lifetime, ma gli identificatori di variabili, costanti e funzioni possiedono anche un linkage.
Per comprendere meglio il concetto è necessario sapere che in C e in C++ l'unità di compilazione è il file, un programma può consistere di più file che vengono compilati separatamente e poi linkati (collegati) insieme per ottenere un file eseguibile. Quest'ultima operazione è svolta dal linker e possiamo pensare al concetto di linkage sostanzialmente come a una sorta di scope dal punto di vista del linker.

Facciamo un esempio:

```
// File a.cpp
   int a = 5;

// File b.cpp
   extern int a;

   int GetVar() {
     return a;
   }
```

Il primo file dichiara una variabile intera e la inizializza, il secondo (trascuriamo per ora la prima riga di codice) dichiara una funzione che ne restituisce il valore. La compilazione del primo file non è un problema, ma nel secondo file GetVar() deve utilizzare un nome dichiarato in un altro file; perché la cosa sia possibile bisogna informare il compilatore che tale nome è dichiarato da qualche altra parte e che il riferimento a tale nome deve essere risolto dal linker (il compilatore non è in grado di farlo perché compila un file alla volta), tale dichiarazione viene effettuata tramite la keyword *extern*.
In effetti la riga extern int a; non dichiara un nuovo identificatore, ma dice "La variabile intera a è dichiarata da qualche altra parte, lascia solo lo spazio per risolvere il riferimento".

Se la keyword extern fosse stata omessa il compilatore avrebbe interpretato la riga come una nuova dichiarazione e avrebbe risolto il riferimento in GetVar() in favore di tale definizione; in fase di linking comunque si sarebbe verificato un errore perché a sarebbe stata definita due volte (una per file).

Naturalmente extern si può usare anche con le funzioni:

```
// File a.cpp
   int a = 5;

   int f(int c) {
     return a+c;
   }

// File b.cpp
   extern int f(int);

   int GetVar() {
     return f(5);
   }
```

Si noti che è necessario che extern sia seguita dal prototipo completo della funzione, al fine di consentire al compilatore di generare codice corretto e di eseguire i controlli di tipo sui parametri e il valore restituito.

Come già detto, il C++ ha un'alta compatibilità col C, tant'è che è possibile interfacciare codice C++ con codice C; anche in questo caso l'aiuto ci viene dalla keyword extern. Per poter linkare un modulo C con un modulo C++ è necessario indicare al compilatore le nostre intenzioni:

```
// Contenuto file C++
  extern "C" int CFunc(char *);
  extern "C" char * CFunc2(int);

  // oppure per risparmiare
  extern "C" {
    void CFunc1(void);
    int * CFunc2(int, char);
    char * strcpy(char *, const char *);
  }
```

La presenza di "C" serve a indicare che bisogna adottare le convenzioni del C sulla codifica dei nomi.

Un altro uso di extern è quello di ritardare la definizione di una variabile o di una funzione all'interno dello stesso file, ad esempio per realizzare funzioni mutuamente ricorsive:

```
extern Func2(int);

int Func1(int c) {
  if (c==0) return 1;
  return Func2(c-1);
}

int Func2(int c) {
  if (c==0) return 2;
  return Func1(c-1);
}
```

I nomi che sono visibili all'esterno di un file (come la variabile a) sono detti avere linkage esterno; tutte le variabili globali hanno linkage esterno, così come le funzioni globali non inline (che per il loro funzionamento richiedono che il codice sorgente sia sempre disponibile); le funzioni inline, tutte le costanti e le dichiarazioni fatte in un blocco hanno invece linkage interno (cioè non sono visibili all'esterno del file); i nomi di tipo non hanno alcun linkage, ma devono riferire ad una unica definizione:

```
// File 1.cpp
   enum Color { Red, Green, Blue };

   extern void f(Color);

// File2.cpp
   enum Color { Red, Green, Blue };

   void f(Color c) { /* ... */ }
```

Una situazione di questo tipo è illecita, ma molti compilatori potrebbero non accorgersi dell'errore.

Per quanto concerne i nomi di tipo, fanno eccezione quelli definiti tramite typedef in quanto non sono veri tipi, ma solo abbreviazioni.

È' possibile forzare un identificatore globale e forzarlo ad avere linkage interno utilizzando la keyword static:

```
// File a.cpp
   static int a = 5;     // linkage interno

   int f(int c) {        // linkage esterno
     return a+c;
   }

// File b.cpp
   extern int f(int);

   static int GetVar() {     // linkage interno
     return f(5);
   }
```

Si faccia attenzione al significato di static: nel caso di variabili locali static serve a modificarne la lifetime (durata), nel caso di nomi globali invece modifica il linkage.

L'importanza di poter restringere il linkage è ovvia; supponete di voler realizzare una libreria di funzioni, alcune serviranno solo a scopi interni alla libreria e non serve (anzi è pericoloso) esportarle, per fare ciò basta dichiarare static i nomi globali che volete incapsulare.

File header

Purtroppo non esiste un meccanismo analogo alla keyword static per forzare un linkage esterno, d'altronde i nomi di tipo non hanno linkage (e devono essere consistenti) e le funzioni inline non possono avere linkage esterno.

Esiste tuttavia un modo per aggirare l'ostacolo: racchiudere tali dichiarazioni e/o definizioni in un file header (file solitamente con estensione .h) e poi includere questo nei files che utilizzano tali dichiarazioni; possiamo anche inserire dichiarazioni e/o definizioni comuni in modo da non doverle ripetere.

Vediamo come procedere. Supponiamo di avere un certo numero di file che devono condividere delle costanti, delle definizioni di tipo e delle funzioni inline; quello che dobbiamo fare è creare un file contenente tutte queste definizioni:

```
// Esempio.h
enum Color { Red, Green, Blue };
struct Point {
  float X;
  float Y;
};

const int Max = 1000;

inline int Sum(int x, int y) {
  return x + y;
}
```

A questo punto basta utilizzare la direttiva #include "NomeFile" nei moduli che utilizzano le precedenti definizioni:

```
// Modulo1.cpp
#include "Esempio.h"

/* codice modulo */
```

La direttiva #include è gestita dal precompilatore che è un programma che esegue delle manipolazioni sul file prima che questo sia compilato; nel nostro caso la direttiva dice di copiare il contenuto del file specificato nel file che vogliamo compilare e passare quindi al compilatore il risultato dell'operazione.

In alcuni esempi abbiamo già utilizzato la direttiva per poter eseguire input/output, in quei casi abbiamo utilizzato le parentesi angolari (<>) al posto dei doppi apici (" "); la differenza è che utilizzando i doppi apici dobbiamo specificare (se

necessario) il path in cui si trova il file header, con le parentesi angolari invece il preprocessore cerca il file in un insieme di directory predefinite.

Un file header può contenere in generale qualsiasi istruzione C/C++, in particolare anche dichiarazioni extern da condividere tra più moduli:

```
// Esempio2.h
// dichiarazioni extern comuni ai moduli
extern int a;
extern double * Ptr;
extern void Func();
extern "C" {
  int CFunc1(int, float);
  void CFunc2(char *);
}
```

Librerie di funzioni

L'uso dei file header visto prima è molto utile quando si vuole partizionare un programma in più moduli, tuttavia la potenza dei file header si esprime meglio quando si vuole realizzare una libreria di funzioni.
L'idea è quella di separare l'interfaccia della libreria dalla sua implementazione: nel file header vengono dichiarati (ed eventualmente definiti) gli identificatori che devono essere visibili anche a chi usa la libreria (costanti, funzioni, tipi...), tutto ciò che è privato (implementazione di funzioni non inline, variabili...) viene invece messo in un altro file che include l'interfaccia.
Vediamo un esempio di semplicissima libreria per gestire date (l'esempio vuole essere solo didattico); ecco il file header:

```
// Date.h
struct Date {
  unsigned short dd;    // giorno
  unsigned short mm;    // mese
  unsigned yy;          // anno
  unsigned short h;     // ora
  unsigned short m;     // minuti
  unsigned short s;     // secondi
};

void PrintDate(Date);
/* altre funzioni */
```

ed ecco come sarebbe il file che la implementa:

```
// Date.cpp
#include <Date.h>
#include <iostream.h>

void PrintDate(Date dt) {
  cout << dt.dd << '/' << dt.mm << '/' << dt.yy;
  cout << "       " << dt.h << ':' << dt.m;
  cout << ':' << dt.s;
}

/* implementazione di altre funzioni */
```

A questo punto la libreria è pronta, per distribuirla basta compilare il file Date.cpp e fornire il file oggetto ottenuto insieme al file header Date.h. Chi deve utilizzare la libreria non dovrà far altro che includere nel proprio programma il file header e linkarlo al file oggetto contenente le funzioni di libreria. Semplicissimo!
C'è tuttavia un problema illustrato nel seguente esempio:

```
// Modulo1.h
```

```
#include <iostream.h>

/* altre dichiarazioni */

// Modulo2.h
#include <iostream.h>

/* altre dichiarazioni */

// Main.cpp
#include <iostream.h>
#include <Modulo1.h>
#include <Modulo2.h>

void main() { /* codice funzione */ }
```

Si tratta cioè di un programma costituito da più moduli, quello principale che contiene la funzione main() e altri che implementano le varie routine necessarie. Più moduli hanno bisogno di una stessa libreria, in particolare hanno bisogno di includere lo stesso file header (nell'esempio iostream.h) nei rispettivi file header.
Per come funziona il preprocessore, poiché il file principale include (direttamente e/o indirettamente) più volte lo stesso file header, il file che verrà effettivamente compilato conterrà più volte le stesse dichiarazioni (e definizioni) che daranno luogo a errori di definizione ripetuta dello stesso oggetto (funzione, costante, tipo...). Come ovviare al problema?

La soluzione ci è fornita dal precompilatore stesso ed è nota come compilazione condizionale; consiste cioè nello specificare quando includere o meno determinate porzioni di codice. Per far ciò ci si avvale delle direttive #define SIMBOLO, #ifndef SIMBOLO e #endif: la prima ci permette di definire un simbolo, la seconda è come l'istruzione condizionale e serve a testare un simbolo (la risposta è 1 se SIMBOLO non è definito, 0 altrimenti), l'ultima direttiva serve a capire dove finisce l'effetto della direttiva condizionale. Le ultime due direttive sono utilizzate per delimitare porzioni di codice; se #ifndef restituisce 1 il preprocessore lascia passare il codice (ed esegue eventuali direttive) tra l'#ifndef e #endif,
altrimenti quella porzione di codice viene nascosta al compilatore.
Ecco come tali direttive sono utilizzate (l'errore era dovuto all'inclusione multipla di iostream.h):

```
// Contenuto del file iostream.h
#ifndef __IOSTREAM_H
#define __IOSTREAM_H

/* contenuto file header */

#endif
```

si verifica cioè se un certo simbolo è stato definito, se non lo è (cioè #ifndef restituisce 1) si definisce il simbolo e poi si inserisce il codice C/C++, alla fine si inserisce l'#endif.

Ritornando all'esempio, ecco ciò che succede quando si compila il file Main.cpp:

1. Il preprocessore inizia a elaborare il file per produrre un unico file compilabile;
2. Viene incontrata la direttiva #include <iostream.h> e il file header specificato viene elaborato per produrre codice;
3. A seguito delle direttive contenute inizialmente in iostream.h, viene definito il simbolo __IOSTREAM_H e prodotto il codice contenuto tra #ifndef __IOSTREAM_H e #endif;
4. Si ritorna al file Main.cpp e il precompilatore incontra #include <Modulo1.h> e quindi va ad elaborare Modulo1.h;
5. La direttiva #include <iostream.h> contenuta in Modulo1.h porta il precompilatore ad elaborare di nuovo iostream.h, ma questa volta il simbolo __IOSTREAM_H è definito e quindi #ifndef __IOSTREAM_H fa sì che nessun codice venga prodotto;
6. Si prosegue l'elaborazione di Modulo1.h e viene generato l'eventuale codice;
7. Finita l'elaborazione di Modulo1.h, la direttiva #include <Modulo2.h> porta all'elaborazione di Modulo2.h che è analoga a quella di Modulo1.h;
8. Elaborato anche Modulo2.h, rimane la funzione main() di Main.cpp che produce il corrispondente codice;

9. Alla fine il precompilatore ha prodotto un unico file contenete tutto il codice di Modulo1.h, Modulo2.h e Main.cpp senza alcuna duplicazione e contenente tutte le dichiarazioni e le definizioni necessarie;
10. Il file prodotto dal precompilatore è passato al compilatore per la produzione di codice oggetto;

Utilizzando il metodo appena visto in tutti i file header (in particolare quelli di libreria) si può star sicuri che non ci saranno problemi di inclusione multipla. Tutto il meccanismo richiede però che i simboli definiti con la direttiva #define siano unici.

Programmazione a oggetti

I costrutti analizzati finora costituiscono già un linguaggio che ci consente di realizzare anche programmi complessi e di fatto, salvo alcune cose, quanto visto costituisce il linguaggio C; tuttavia il C++ è molto di più e offre caratteristiche nuove che estendono e migliorano il C: programmazione a oggetti, template (modelli) e gestione delle eccezioni.

Si potrebbe apparentemente dire che si tratta solo di qualche aggiunta, in realtà nessun'altra affermazione potrebbe essere più errata: mentre l'ultima caratteristica (la gestione delle eccezioni) è in effetti una estensione (l'aggiunta di qualcosa che mancava), le prime due non sono semplici aggiunte in quanto non si limitano a fornire nuove funzionalità, ma impongono un nuovo modo di concepire e realizzare codice e caratterizzano il linguaggio fino a influenzare il codice prodotto in fase di compilazione (notevolmente diverso da quello prodotto dal compilatore C).

Inizieremo ora a discutere dei meccanismi offerti dal C++ per la programmazione orientata agli oggetti, si ricorda che l'autore assume noti al lettore i concetti di tale paradigma, per chi non avesse tali conoscenze (o desiderasse riportarli alla mente) è disponibile un articolo sufficientemente completo all'indirizzo http://www.nsm.it/mondobit/mb1/mb1prog.html.

Strutture e campi funzione

La programmazione orientata agli oggetti (OOP) impone una nuova visione di concetti quali "Tipo di dato" e "Istanze di tipo". Sostanzialmente mentre gli altri paradigmi di programmazione vedono le istanze di un tipo di dato come una entità passiva, nella programmazione a oggetti invece tali istanze diventano a tutti gli effetti entità (oggetti) attive.

L'idea è che non bisogna più manipolare direttamente i valori di una struttura (intesa come generico contenitore di valori), meglio lasciare che sia la struttura stessa a manipolarsi e a compiere le operazioni per noi. Tutto ciò che bisogna fare è inviare all'oggetto un messaggio che specifichi l'operazione da compiere e attendere poi che l'oggetto stesso ci comunichi il risultato. Il meccanismo dei messaggi viene sostanzialmente implementato tramite quello della chiamata di funzione e l'insieme dei messaggi cui un oggetto risponde viene definito associando al tipo dell'oggetto un insieme di funzioni.

In C++ ciò può essere realizzato tramite le strutture:

```
struct Complex {
  float Re;
  float Im;

  // Ora nelle strutture possiamo avere dei campi
  // di tipo funzione;

  void Print();
  float Abs();
  void Set(float PR, float PI);
};
```

Ciò che sostanzialmente cambia, rispetto a quanto visto, è che una struttura può possedere campi di tipo funzione (detti "funzioni membro" oppure "metodi") che costituiscono (insieme ai campi ordinari ("membri dato" o "attributi") l'insieme dei messaggi a cui quel tipo è in grado di rispondere (interfaccia).

L'esempio non mostra come implementare le funzioni membro, per adesso ci basta sapere che esse vengono definite da qualche parte fuori dalla dichiarazione di struttura.

Una funzione dichiarata come campo di una struttura può essere invocata ovviamente solo se associata ad una istanza della struttura stessa, dato che quello che si fa è inviare un messaggio ad un oggetto, e nella pratica effettuata tramite la stessa sintassi utilizzata per selezionare un qualsiasi altro campo:

```
Complex A;
Complex * C;

A.Set(0.2, 10.3);
A.Print();
C = new Complex;
C->Set(1.5, 3.0);
float FloatVar = C->Abs();
```

Nell'esempio viene mostrato come inviare un messaggio: la quarta riga invia il messaggio Print() all'oggetto A, l'ultima invece invia il messaggio Abs() all'oggetto puntato da C e assegna il valore ottenuto alla variabile FloatVar.

Il vantaggio principale di questo modo di procedere è il non doversi più preoccupare di come è fatto quel tipo, se si vuole eseguire una operazione su una sua istanza (ad esempio visualizzarne il valore) basta inviare il messaggio corretto, sarà l'oggetto in questione ad eseguirla per noi. Ovviamente perché tutto funzioni è necessario evitare di accedere direttamente agli attributi di un oggetto, altrimenti crolla uno dei capisaldi della OOP, e sfortunatamente per noi il meccanismo delle strutture consente l'accesso diretto a tutto ciò che fa parte della dichiarazione di struttura, annullando di fatto ogni vantaggio:

```
// Con riferimento agli esempi riportati sopra:

A.Set(6.1, 4.3);    // Setta il valore di A
A.Re = 10;          // Ok!
A.Im = .5;          // ancora Ok!
A.Print();
```

Sintassi della classe

Il problema viene risolto introducendo una nuova sintassi per la dichiarazione di un tipo oggetto. Un tipo oggetto viene dichiarato tramite una dichiarazione di classe, che differisce dalla dichiarazione di struttura sostanzialmente per i meccanismi di protezione offerti; per il resto tutto ciò che si applica alle classi si applica allo stesso modo alla dichiarazione di struttura senza alcuna differenza.

Vediamo dunque come sarebbe stato dichiarato il tipo Complex tramite la sintassi della classe:

```
class Complex {
  public:
    void Print();   // definizione eseguita altrove!
    /* altre funzioni membro */
  private:
    float Re;       // Parte reale
    float Im;       // Parte immaginaria
};
```

La differenza è data dalle keyword public e private che consentono di specificare i diritti di accesso alle dichiarazioni che le seguono:

- **public** : le dichiarazioni che seguono questa keyword sono visibili a chi usa una istanza della classe e l'invocazione (selezione) di uno di questi campi è sempre possibile;
- **private** : tutto ciò che segue è visibile solo alla classe stessa, una invocazione di uno di questi campi è possibile solo dai metodi della classe stessa;

come mostra il seguente esempio:

```
Complex A;
Complex * C;

A.Re = 10.2;    // Errore!
```

```
C->Im = .5;      // ancora errore!
A.Print();       // Ok!
C->Print()       // Ok!
```

Ovviamente le due keyword sono mutuamente esclusive, nel senso che alla dichiarazione di un metodo o di un attributo si applica la prima keyword che si incontra risalendo in su; se la dichiarazione non è preceduta da nessuna di queste keyword, il default è private:

```
class Complex {
    float Re;   // private per
    float Im;   // default
  public:
    void Print();
    /* altre funzioni membro*/
};
```

In realtà esiste una terza categoria di visibilità definibile tramite la keyword protected (che però analizzeremo quando parleremo di ereditarietà); la sintassi per la dichiarazione di classe è dunque:

```
class ClassName {
  public:
    /* membri pubblici */
  protected:
    /* membri protetti */
  private:
    /* membri privati */
};          // notare il punto e virgola finale!
```

Non ci sono limitazioni al tipo di dichiarazioni possibili dentro una delle tre sezioni di visibilità: definizioni di variabili o costanti (attributi), funzioni (metodi) oppure dichiarazioni di tipi (enumerazioni, unioni, strutture e anche classi); tuttavia esiste una differenza per quanto riguarda le regole di scoping sui tipi annidati:

- tipi definiti nella sezione **public** sono visibili anche alle altre due sezioni, ma non viceversa;
- tipi definiti nella sezione **protected** sono visibili anche alla sezione private, ma non a public;
- tipi definiti nella sezione **private** sono visibili ovviamente solo ad essa;

Il motivo è ovvio, se ad esempio un metodo pubblico dovesse restituire un tipo privato, l'utente della classe non sarebbe in grado di gestire il valore ottenuto perché non è in grado di accedere alla definizione di tipo; questo naturalmente non vale per i metodi e gli attributi che se sono privati possono essere direttamente acceduti solo da metodi della classe stessa senza porre alcun problema di visibilità all'esterno della classe.

Definizione delle funzioni membro

La definizione dei metodi di una classe può essere eseguita o dentro la dichiarazione di classe, facendo seguire alla lista di argomenti una coppia di parentesi graffe racchiudente la sequenza di istruzioni:

```
class Complex {
  public:
    /* ... */
    void Print() {
      if (Im >= 0)
        cout << Re << " + i" << Im;
      else
        cout << Re << " - i" << fabs(Im);
        // fabs restituisce il valore assoluto!
    }
  private:
    /* ... */
};
```

oppure riportando nella dichiarazione di classe solo il prototipo e definendo il metodo fuori dalla dichiarazione di classe, nel seguente modo (anch'esso applicabile alle strutture):

```
// Questo modo di procedere richiede l'uso
// dell'operatore di risoluzione di scope e
// l'uso del nome della classe per indicare
// esattamente quale metodo si sta definendo
// (classi diverse possono avere metodi con
// lo stesso nome).

void Complex::Print() {
  if (Im >= 0)
    cout << Re << " + i" << Im;
  else
    cout << Re << " - i" << fabs(Im);
    // fabs restituisce il valore assoluto!
}
```

La differenza è che nel primo caso implicitamente si richiede una espansione inline del codice della funzione, nel secondo caso se si desidera tale accorgimento bisogna utilizzare esplicitamente la keyword inline nella definizione del metodo:

```
inline void Complex::Print() {
  if (Im >= 0)
    cout << Re << " + i" << Im;
  else
    cout << Re << " - i" << fabs(Im);
    // fabs restituisce il valore assoluto!
}
```

Se la definizione del metodo Print() è stata studiata con attenzione, il lettore avrà notato che la funzione accede ai membri dato senza ricorrere alla notazione del punto, ma semplicemente nominandoli: quando ci si vuole riferire ai campi dell'oggetto cui è stato inviato il messaggio non bisogna adottare alcuna particolare notazione, lo si fa e basta!

Il problema di risolvere correttamente ogni riferimento viene risolto automaticamente dal compilatore: all'atto della chiamata, ciascun metodo riceve un parametro aggiuntivo, un puntatore all'oggetto a cui è stato inviato il messaggio e tramite questo è possibile risalire all'indirizzo corretto; ciò inoltre consente la chiamata di un metodo da parte di un altro metodo:

```
class MyClass {
  public:
    void BigOp();
    void SmallOp();
  private:
    void PrivateOp();
    /* altre dichiarazioni */
};

/* definizione di SmallOp() e PrivateOp() */

void MyClass::BigOp() {
  /* ... */
  SmallOp();        // questo messaggio è inviato all'oggetto
                    // a cui è stato inviato BigOp()
  /* ... */
  PrivateOp();      //tutto Ok!
  /* ... */
}
```

Ovviamente un metodo può avere parametri e/o variabili locali che sono istanze della stessa classe cui appartiene (il nome della classe è già visibile all'interno della stessa classe), in questo caso per riferirsi al parametro o alla variabile locale deve utilizzare la notazione del punto:

```
class MyClass {
    /* ... */
    void Func(MyClass A, /* ... */ );
};

void MyClass::Func(MyClass A, /* ... */ ) {
  /* ... */
  BigOp();      // questo messaggio è inviato all'oggetto
                // a cui è stato inviato Func( /* ... */ )
  A.BigOp();    // questo invece viene inviato al parametro.
  /* ... */
}
```

In alcuni rari casi può essere utile avere accesso al puntatore che il compilatore aggiunge tra i parametri di un metodo, l'operazione è fattibile tramite la keyword *this* (che in pratica è il nome del parametro aggiuntivo), tale pratica quando possibile è comunque da evitare.

Costruttori

L'uso di un metodo Set() per eseguire l'inizializzazione di un oggetto (come mostrato per la struct Complex) è poco elegante e alquanto insicuro: il programmatore che usa la classe potrebbe dimenticare di chiamare tale metodo prima di cominciare ad utilizzare l'oggetto appena dichiarato. Si potrebbe pensare di scrivere qualcosa del tipo:

```
class Complex {
  public:
   /* ... */
  private:
   float Re = 6;      // Errore!
   float Im = 7;      // Errore!
};
```

ma il compilatore rifiuterà di accettare tale codice. Il motivo è semplice, stiamo definendo un tipo e non una variabile (o una costante) e non è possibile inizializzare i membri di una classe (o di una struttura) in quel modo.

Il metodo corretto è quello di fornire un *costruttore* che il compilatore possa utilizzare quando una istanza della classe viene creata, in modo che tale istanza sia sin dall'inizio in uno stato consistente. Un costruttore altro non è che un metodo il cui nome è lo stesso di quello della classe, che può avere dei parametri, ma che non restituisce alcun tipo (neanche void); il suo scopo è quello di inizializzare le istanze della classe:

```
Class Complex {
  public:
    Complex(float a, float b) {  // costruttore!
      Re = a;
      Im = b;
    }
    /* altre funzioni membro */
  private:
    float Re;       // Parte reale
    float Im;       // Parte immaginaria
};
```

In questo modo possiamo eseguire dichiarazione e inizializzazione di un oggetto Complex in un colpo solo:

```
Complex C(3.5, 4.2);
```

La definizione appena vista introduce un oggetto C di tipo Complex che viene inizializzato chiamando il costruttore con gli argomenti specificati tra le parentesi. Si noti che il costruttore non viene invocato come un qualsiasi metodo; un sistema alternativo di eseguire l'inizializzazione sarebbe:

```
Complex C = Complex(3.5, 4.2);
```

ma è poco efficiente perché quello che si fa è creare un oggetto Complex temporaneo e poi copiarlo in C, il primo metodo invece fa tutto in un colpo solo.

Un costruttore può eseguire compiti semplici come quelli dell'esempio, tuttavia non è raro che una classe necessiti di costruttori molto complessi, specie se alcuni membri sono dei puntatori; in questi casi un costruttore può eseguire operazioni complesse quali allocazione di memoria o accessi a unità a disco se si lavora con oggetti persistenti.

In alcuni casi, alcune operazioni possono richiedere la certezza assoluta che tutti o parte dei campi dell'oggetto, che si vuole creare, siano subito inizializzati prima ancora che incominci l'esecuzione del corpo del costruttore; la soluzione in questi casi prende il nome di lista di inizializzazione.
La lista di inizializzazione è una caratteristica propria dei costruttori e appare sempre tra la lista di argomenti del costruttore e il suo corpo:

```
class Complex {
  public:
    Complex(float, float);
    /* ... */
  private:
    float Re;
    float Im;
};

Complex::Complex(float a, float b) : Re(a), Im(b) { }
```

L'ultima riga dell'esempio implementa il costruttore della classe Complex; si tratta esattamente dello stesso costruttore visto prima, la differenza sta tutta nel modo in cui sono inizializzati i membri dato: la notazione Attributo(<Espressione>) indica al compilatore che Attributo deve memorizzare il valore fornito da Espressione; Espressione può essere anche qualcosa di complesso come la chiamata ad una funzione.
Nel caso appena visto l'importanza della lista di inizializzazione può non essere evidente, lo sarà di più quando parleremo di oggetti composti e di ereditarietà.

Una classe può possedere più costruttori, cioè i costruttori possono essere overloaded, in modo da offrire diversi modi per inizializzare una istanza; in particolare alcuni costruttori assumono un significato speciale:

- il costruttore di default *ClassName::ClassName();*
- il costruttore di copia *ClassName::ClassName(ClassName& X)*;
- altri costruttori con un solo argomento;

Il costruttore di default è particolare, in quanto è quello che il compilatore chiama quando il programmatore non utilizza altri costruttori in seguito alla dichiarazione di un oggetto:

```
#include <iostream.h>

class Trace {
  public:
    Trace() {
      cout << "costruttore di default" << endl;
    }
    Trace(int a, int b) : M1{a), M2(b) {
      cout << "costruttore Trace(int, int)" << endl;
    }
```

```
  private:
    int M1, M2;
};

void main {
  cout << "definizione di B...";
  MyClass B(1, 5);    // MyClass(int, int) chiamato!
  cout << "definizione di C...";
  MyClass C;          // costruttore di default chiamato!
}
```

Eseguendo tale codice si ottiene l'output:

```
definizione di B...costruttore Trace(int, int)
definizione di C...costruttore di default
```

Ma l'importanza del costruttore di default è dovuta soprattutto al fatto che se il programmatore della classe non definisce alcun costruttore, automaticamente il compilatore ne fornisce uno (che però non dà garanzie sul contenuto dei membri dato dell'oggetto). Se non si desidera il costruttore di default fornito dal compilatore, occorre definirne esplicitamente uno (anche se non di default).
Il costruttore di copia invece viene invocato quando un nuovo oggetto va inizializzato in base al contenuto di un altro; modifichiamo la classe Trace in modo da aggiungere il seguente costruttore di copia:

```
Trace::Trace(Trace& x) : M1(x.M1), M2(x.M2) {
  cout << "costruttore di copia" << endl;
}
```

e aggiungiamo il seguente codice a main():

```
cout << "definizione di D...";
Trace D = B;
```

Ciò che viene visualizzato ora, è che per D viene chiamato il costruttore di copia.
Se il programmatore non definisce un costruttore di copia, ci pensa il compilatore. In questo caso il costruttore fornito dal compilatore esegue una copia bit a bit degli attributi; in generale questo è sufficiente, ma quando una classe contiene puntatori è necessario definirlo esplicitamente onde evitare problemi di condivisione di aree di memoria.

I principianti tendono spesso a confondere l'inizializzazione con l'assegnamento; benché sintatticamente le due operazioni sono simili, in realtà esiste una profonda differenza semantica: l'inizializzazione viene compiuta una volta sola, quando l'oggetto viene creato; un assegnamento invece si esegue su un oggetto precedentemente creato. Per comprendere la differenza facciamo un breve salto in avanti.

Il C++ consente di eseguire l'overloading degli operatori, tra cui quello per l'assegnamento; come nel caso caso del costruttore di copia (e di quello di default), anche per l'operatore di assegnamento vale il discorso fatto nel caso che tale operatore non venga definito esplicitamente.
Il costruttore di copia viene utilizzato quando si dichiara un nuovo oggetto e si inizializza il suo valore con quello di un altro; l'operatore di assegnamento invece viene invocato successivamente in tutte le operazioni che assegnamo all'oggetto dichiarato un altro oggetto. Vediamo un esempio:

```
#include <iostream.h>

class Trace {
  public:
    Trace(Trace& x) : M1(x.M1), M2(x.M2) {
      cout << "costruttore di copia" << endl;
    }
    Trace(int a, int b) : M1(a), M2(b) {
      cout << "costruttore Trace(int, int)" << endl;
    }
```

```
    Trace& operator=(const Trace& x) {
      cout << "operatore =" << endl;
      M1 = x.M1;
      M2 = x.M2;
      return *this;
    }
  private:
    int M1, M2;
};

void main() {
  cout << "definizione di A...";
  Trace A(1,2);
  cout << "definizione di B...";
  Trace B(2,4);
  cout << "definizione di C...";
  Trace C = A;
  cout << "assegnamento a C...";
  C = B;
}
```

Eseguendo tale codice otteniamo il seguente output:

```
definizione di A...costruttore Trace(int, int)
definizione di B...costruttore Trace(int, int)
definizione di C...costruttore di copia
assegnamento a C...operatore =
```

Restano da esaminare i costruttori che prendono un solo argomento. A tale proposito basta semplicemente dire che a tutti gli effetti essi sono dei veri e propri operatori di conversione di tipo(vedi appendice A) che convertono il loro argomento in una istanza della classe. Ecco una classe che fornisce diversi operatori di conversione:

```
class MyClass {
  public:
    MyClass(int);
    MyClass(long double);
    MyClass(Complex);
    /* ... */
  private:
    /* ... */
};

void main() {
  MyClass A(1);
  MyClass B = 5.5;
  MyClass D = (MyClass) 7;
  MyClass C = Complex(2.4, 1.0);
}
```

Le prime tre dichiarazioni sono concettualmente identiche, in tutti e tre i casi convertiamo un valore di un tipo in quello di un altro; il fatto che l'operazione sia eseguita per inizializzare degli oggetti non modifica in alcun modo il significato dell'operazione stessa; al più l'unica differenza è che nel primo caso si esegue in un colpo solo conversione e inizializzazione, mentre nel secondo e nel terzo caso prima si esegue la conversione e poi si chiama il costruttore di copia.

Solo l'ultima dichiarazione può apparentemente sembrare diversa, in pratica è comunque la stessa cosa: si crea un oggetto di tipo Complex e poi lo si converte (implicitamente) al tipo MyClass, infine viene chiamato il costruttore di copia per inizializzare C.

Per finire, ecco un confronto tra costruttori e metodi (o normali funzioni) che riassume quanto detto:

	Costruttori	Metodi
Tipo restituito	nessuno	qualsiasi
Nome	quello della classe	qualsiasi
Parametri	nessuna limitazione	nessuna limitazione
Lista di inizializzazione	si	no
Overloading	si	si

Altre differenze e similitudini verranno esaminate nel seguito.

Distruttori

Poiché ogni oggetto ha una propria durata (lifetime) è necessario disporre di un metodo che permetta una corretta distruzione dell'oggetto, un distruttore.
Un distruttore è un metodo che non riceve parametri, non ritorna alcun tipo (neanche void) ed ha lo stesso nome della classe preceduto da una ~ (tilde):

```
class Trace {
  public:
    /* ... */
    ~Trace() {
      cout << "distruttore ~Trace()" << endl;
    }
  private:
    /* ... */
};
```

Il compito del distruttore è quello di assicurarsi della corretta deallocazione delle risorse e se non ne viene esplicitamente definito uno, il compilatore genera per ogni classe un distruttore di default che chiama alla fine della lifetime di una variabile:

```
void MyFunc() {
  TVar A;
  /* ... */
}              // qui viene invocato automaticamente il distruttore per A
```

Si noti che nell'esempio non c'è alcuna chiamata esplicita al distruttore, è il compilatore che lo chiama alla fine del blocco applicativo (le istruzioni racchiuse tra { }) in cui la variabile è stata dichiarata (alla fine del programma per variabili globali e statiche).

Poiché il distruttore fornito dal compilatore non tiene conto di aree di memoria allocate tramite membri puntatore, è sempre necessario definirlo esplicitamente ogni qual volta esistono membri puntatori; come mostra il seguente esempio:

```
#include <iostream.h>

class Trace {
  public:
    /* ... */
    Trace(long double);
    ~Trace();
  private:
    long double * ldPtr;            };

Trace::Trace(long double a) {
  cout << "costruttore chiamato..." << endl;
  ldPtr = new long double(a);       }
```

```
Trace::~Trace() {
  cout << "distruttore chiamato..." << endl;
  delete ldPtr;
}
```

In tutti gli altri casi, spesso il distruttore di default è più che sufficiente e non occorre scriverlo.
Solitamente il distruttore è chiamato implicitamente dal compilatore quando un oggetto termina il suo ciclo di vita, oppure quando un oggetto allocato con new viene deallocato con delete:

```
void func() {
  Trace A(5.5);                  // chiamata costruttore
  Trace * Ptr = new Trace(4.2); // chiamata costruttore
  /* ... */
  delete Ptr;                    // chiamata distruttore
}                                // chiamata distruttore per A
```

In alcuni rari casi può tuttavia essere necessario una chiamata esplicita, in questi casi però il compilatore può non tenerne traccia (in generale un compilatore non è in grado di ricordare se il distruttore per una certa variabile è stato chiamato) e quindi bisogna prendere precauzioni onde evitare che il compilatore, richiamando il costruttore alla fine della lifetime dell'oggetto, generi codice errato.
Facciamo un esempio:

```
void Example() {
  TVar B(10);
  /* ... */
  if (Cond) B.~TVar();
}                            // Possibile errore!
```

Si genera un errore poichè, se Cond è vera, è il programma a distruggere esplicitamente B, e la chiamata al distruttore fatta dal compilatore è illecita. Una soluzione al problema consiste nell'uso di ulteriore blocco applicativo e di un puntatore per allocare nello heap la variabile:

```
void Example() {
  TVar TVarPtr = new TVar(10);
  {
     /* ... */
     if (Cond) {
       delete TVarPtr;     // l'uso di delete provoca prima una chiamata
       TVarPtr = 0;        // a ~TVar e poi alla deallocazione della variabile
     }
     /* ... */
  }
  if (TVarPtr) delete TVarPtr;      }
```

Comunque si tenga presente che i casi in cui si deve ricorrere ad una tecnica simile sono rari e spesso (ma non sempre) denotano un frammento di codice scritto male (quello in cui si vuole chiamare il distruttore) oppure una cattiva ingegnerizzazione della classe cui appartiene la variabile.
Si noti che poiché un distruttore non possiede argomenti, non è possibile eseguirne l'overloading; ogni classe cioè possiede sempre e solo un unico distruttore.

Membri static

Normalmente istanze diverse della stessa classe non condividono direttamente risorse di memoria, l'unica possibilità sarebbe quella di avere puntatori che puntano allo stesso indirizzo, per il resto ogni istanza riceve nuova memoria per ogni attributo. Tuttavia in alcuni casi è desiderabile che alcuni attributi siano comuni a tutte le istanze; per utilizzare un termine tecnico, si vuole realizzare una comunicazione ad ambiente condiviso.
Per rendere un attributo comune a tutte le istanze occorre dichiararlo static:

```
class MyClass {
  public:
    MyClass();
    /* ... */
  private:
    static int Counter;
    char * String;
    /* ... */     };
```

Gli attributi static possono in pratica essere visti come elementi propri della classe, non dell'istanza. In questo senso non è possibile inizializzare un attributo static tramite la lista di inizializzazione del costruttore, tutti i metodi (costruttore compreso) possono accedere sia in scrittura che in lettura all'attributo, ma non si può assegnare un valore ad esso tramite lista di inizializzazione:

```
MyClass::MyClass() : Counter(0) { // Errore!
  /* ... */       }
```

L'inizializzazione di un attributo static va eseguita fuori dalla classe, nel seguente modo:

```
<MemberType> <ClassName>::<StaticMember> = <Value> ;
```

Nel caso dell'attributo Counter, si sarebbe dovuto scrivere:

```
int MyClass::Counter = 0;
```

Successivamente l'accesso a un attributo static avviene come se fosse un normale attributo, in particolare l'idea guida dell'esempio era quella di contare le istanze di classe MyClass esistenti in un certo momento; i costruttori e il distruttore sarebbero stati quindi più o meno così:

```
MyClass::MyClass() : /* inizializzazione membri non static */ {
  ++Counter;
  /* ... */
}

MyClass::~MyClass() {
  --Counter;
  /* ... */
}
```

Oltre ad attributi static è possibile avere anche metodi static; la keyword static in questo caso vincola il metodo ad accedere solo agli attributi statici della classe, un accesso ad un attributo non static costituisce un errore :

```
class MyClass {
  public:
    static int GetCounterValue();
    /* ... */
  private:
    static int Counter;
    /* ... */
};

int MyClass::Counter = 0;

static int MyClass::GetCounterValue() {
  return Counter;
}
```

Ci si può chiedere quale motivo ci possa essere per dichiarare un metodo static, ci sono essenzialmente tre motivi:

- maggiore controllo su possibili fonti di errore: dichiarando un metodo static, chiediamo al compilatore di accertarsi che il metodo non acceda ad altre categorie di attributi;
- minor overhead di chiamata: i metodi non static per sapere a quale oggetto devono riferire, ricevono dal compilatore un parametro aggiuntivo che altro non è che un puntatore all'istanza di classe per cui il metodo è stato chiamato; i metodi static per loro natura non hanno bisogno di tale parametro e quindi non richiedono tale overhead;
- i metodi static oltre a poter essere chiamati come un normale metodo, associandoli ad un oggetto (con la notazione del punto), possono essere chiamati come una normale funzione senza necessità di associarli ad una particolare istanza, ricorrendo al risolutore di scope come nel seguente esempio:

```
MyClass Obj;
int Var1 = Obj.GetCounterValue();       // Ok!
int Var2 = MyClass::GetCounterValue();  // Ancora Ok!
```

Non è possibile dichiarare static un costruttore o un distruttore.

Membri const

Oltre ad attributi di tipo static, è possibile dichiarare un attributo const; in questo caso però l'attributo const non è trattato come una costante: esso viene allocato per ogni istanza come un normale attributo, tuttavia il valore che esso assume per ogni istanza viene stabilito una volta per tutte all'atto della creazione dell'istanza stessa e non potrà mai cambiare durante la vita dell'oggetto. Il valore di un attributo const, infine, va settato tramite la lista di inizializzazione del costruttore:

```
class MyClass {
  public:
    MyClass(int a, float b);
    /* ... */
  private:
    const int ConstMember;
    float AFloat;
};

MyClass::MyClass(int a, float b) : ConstMember(a), AFloat(b) { };
```

Il motivo per cui bisogna ricorrere alla lista di inizializzazione è semplice: l'assegnamento è una operazione proibita sulle costanti, l'operazione che si compie tramite la lista di inizializzazione è invece concettualmente diversa (anche se per i tipi primitivi è equivalente ad un assegnamento).

È anche possibile avere funzioni membro const (in questo caso la keyword const va posta dopo la lista dei parametri del metodo) analogamente a quanto avviene per le funzioni membro statiche. Dichiarando un metodo const si stabilisce un contratto con il compilatore: la funzione membro si impegna a non accedere in scrittura ad un qualsiasi attributo della classe e il compilatore si impegna a segnalare con un errore ogni tentativo in tal senso.
Oltre a ciò esiste un altro vantaggio a favore dei metodi const: sono gli unici a poter essere eseguiti su istanze costanti.

```
class MyClass {
  public:
    MyClass(int a, float b) : ConstMember(a), AFloat(b) { };
    int GetConstMember() const {
      return ConstMember;
    }
    void ChangeFloat(float b) {
      AFloat = b;
    }
  private:
    const int ConstMember;
    float AFloat;
};
```

```
void main() {
  MyClass A(1, 5.3);
  const MyClass B(2, 3.2);

  A.GetConstMember();       // Ok!
  B.GetConstMember();       // Ok!
  A.ChangeFloat(1.2);       // Ok!
  B.ChangeFloat(1.7);       // Errore!
}
```

Come per i metodi static, non è possibile avere costruttori e distruttori const (sebbene essi vengano utilizzati per costruire e distruggere anche le istanze costanti).

Costanti vere dentro le classi

Poiché gli attributi const altro non sono che attributi a sola lettura, ma che vanno inizializzati tramite lista di inizializzazione, è chiaro che non è possibile scrivere codice simile:

```
class Bad {
  public:
    /* ... */
  private:
    const int Size;
    char String[Size];
};
```

perché non si può stabilire a tempo di compilazione il valore di Size. La soluzione al problema viene dalla keyword enum; se ricordate bene, è possibile stabilire quali valori interi associare alle costanti che appaiono tra parentesi graffe al fine di rappresentarle.
Nel nostro caso dunque la soluzione è:

```
class Ok {
  public:
    /* ... */
  private:
    enum { Size = 20 };
    char String[Size];
};
```

Chi ha fatto attenzione avrà notato che la keyword enum non è seguita da un identificatore, ma direttamente dalla parentesi graffa; il motivo è semplice: non ci interessava definire un tipo enumerato, ma disporre di una costante e quindi abbiamo creato una enumerazione anonima il cui unico effetto in questo caso è quello di creare una associazione nome-valore all'interno della tabella dei simboli del compilatore.

Membri volatile

Il C++ è un linguaggio adatto a qualsiasi tipo di applicazione, in particolare a quelle che per loro natura si devono interfacciare direttamente all'hardware. Una prova in tal proposito è fornita dalla keyword volatile che posta davanti ad un identificatore di variabile comunica al compilatore che quella variabile può cambiare valore in modo asincrono rispetto al sistema:

```
volatile int Var;

/* ... */
int B = Var;
/* ... */
```

In tal modo il compilatore non ottimizza gli accessi a tale risorsa e ogni tentativo di lettura di quella variabile è tradotto in una effettiva lettura della locazione di memoria corrispondente.
Gli oggetti volatile sono normalmente utilizzati per mappare registri di unità di I/O all'interno del programma e per essi valgono le stesse regole viste per gli oggetti const; in particolare solo funzioni membro volatile possono essere utilizzate su oggetti volatile e non si possono dichiarare volatile costruttori e distruttori (che sono comunque utilizzabili sui tali oggetti).
Si noti che volatile non è l'opposto di const: quest'ultima indica al compilatore che un oggetto non può essere modificato indipendentemente che sia trattato come una vera costante o una variabile a sola lettura, volatile invece dice che l'oggetto può cambiare valore al di fuori del controllo del sistema; quindi è possibile avere oggetti const volatile. Ad esempio unità di input, come la tastiera, sono solitamente mappati tramite oggetti dichiarati const volatile:

```
const volatile char Byte;
// Byte è un oggetto a sola lettura il
// cui stato varia in modo asincrono
// rispetto al sistema
```

Dichiarazioni friend

In taluni casi è desiderabile che una funzione non membro possa accedere direttamente ai membri (attributi e/o metodi) privati di una classe. Tipicamente questo accade quando si realizzano due o più classi, distinte tra loro, che devono cooperare per l'espletamento di un compito complessivo e si vogliono ottimizzare al massimo le prestazioni, oppure semplicemente quando si desidera eseguire l'overloading degli operatori ostream& operator<<(ostream& o, T& Obj) e istream& operator>>(istream& o, T& Obj) (T è la classe cui appartengono i membri privati) per estendere le operazioni di I/O alla classe T. In situazioni di questo genere, una classe può dichiarare una certa funzione friend (amica) abilitandola ad accedere ai propri membri privati.

Il seguente esempio mostra come eseguire l'overloading dell'operatore di inserzione in modo da poter visualizzare il contenuto di una nuova classe:

```
#include <iostream.h>

    class MyClass {
      public:
        /* ... */
      private:
        float F1, F2;
        char C
        void Func();
        /* ... */
        friend ostream& operator<<(ostream& o, MyClass& Obj);
    };

    void MyClass::Func() {
      /* ... */
    }

    // essendo stato dichiarato friend dentro MyClass, il seguente operatore
    // può accedere ai membri privati della classe come una qualunque
```

```
// funzione membro.
ostream& operator<<(ostream& o, MyClass& Obj) {
  o << Obj.F1 << ' ' << Obj.F2 << ' ' << Obj.C;
  return o;
}
```

in tal modo diviene possibile scrivere:

```
MyClass Object;
/* ... */

cout << Object;
```

L'esempio comunque risulterà meglio comprensibile quando parleremo di overloading degli operatori, per adesso è sufficiente considerare ostream& operator<<(ostream& o, MyClass& Obj) alla stessa stregua di una qualsiasi funzione.
La keyword friend può essere applicata anche a un identificatore di classe, abilitando così una classe intera:

```
class MyClass {
  /* ... */
  friend class AnotherClass;
};
```

in tal modo qualsiasi membro di AnotherClass può accedere ai dati privati di MyClass.
Si noti infine che deve essere la classe proprietaria dei membri privati a dichiarare una funzione (o una classe) friend e che non ha importanza la sezione (pubblica, protetta o privata) in cui tale dichiarazione è fatta.

Reimpiego di codice

La programmazione orientata agli oggetti è nata con lo scopo di risolvere il problema di sempre del modo dell'informatica: rendere economicamente possibile e facile il reimpiego di codice già scritto. Due sono sostanzialmente le tecniche di reimpiego del codice offerte: reimpiego per composizione e reimpiego per ereditarietà; il C++ ha poi offerto anche il meccanismo dei Template che può essere utilizzato anche in combinazione con quelli classici della OOP.

Per adesso rimanderemo la trattazione dei template ad un apposito capitolo, concentrando la nostra attenzione prima sulla composizione di oggetti e poi sulla ereditarietà il secondo pilastro (dopo l'incapsulazione di dati e codice) della programmazione a oggetti.

Reimpiego per composizione

Benché non sia stato esplicitamente menzionato, non c'è alcun limite alla complessità di un membro dato di un oggetto; un dato attributo può avere sia tipo elementare che tipo definito dall'utente, in particolare un attributo può a sua volta essere un oggetto.
Vediamo un esempio che mostra come definire una matrice 10x10 di numeri complessi:

```
class Complex {
  public:
    Complex(float Real=0, float Immag=0);
    Complex operator+(Complex &);
    Complex operator-(Complex &);
    /* ... */
  private:
    float Re, Im;
};

class Matrix {
  public:
    Matrix();
    Matrix operator+(Matrix &);
    /* ... */
  private:
    Complex Data[10][10];
};
```

L'esempio mostrato suggerisce un modo di reimpiegare codice già pronto quando si è di fronte ad una relazione di tipo *Has-a*, in cui una entità più piccola è effettivamente parte di una più grossa; tuttavia la composizione può essere utilizzata anche per modellare una relazione di tipo *Is-a*, in cui invece una istanza di un certo tipo può essere vista anche come istanza di un tipo più "piccolo":

```
class Person {
  public:
    Person(const char * name, unsigned age);
    void PrintName();
    /* ... */
  private:
    const char * Name;
    unsigned int Age;
};




class Student {
```

```
  public:
    Student(const char * name, unsigned age, unsigned code);
    void PrintName();
    /* ... */
  private:
    Person Self;
    unsigned int IdCode;    // numero di matricola
    /* ... */
};

Student::Student(const char * name, unsigned age, unsigned code) :
   Self(name, age), IdCode(code) {}

void Student::PrintName() {
  Self.PrintName();
}

/* ... */
```

In sostanza la composizione può essere utilizzata anche quando vogliamo semplicemente estendere le funzionalità di una classe realizzata in precedenza. Esistono due tecniche di composizione:

- Contenimento diretto;
- Contenimento tramite puntatori.

Nel primo caso un oggetto viene effettivamente inglobato all'interno di un altro (come negli esempi visti), nel secondo invece l'oggetto contenitore in realtà contiene un puntatore. Le due tecniche offrono vantaggi e svantaggi differenti. Nel caso del contenimento tramite puntatori:

- L'uso di puntatori permette di modellare relazioni 1-n altrimenti non modellabili se non stabilendo un valore massimo per n;
- Non è necessario conoscere il modo in cui va costruito una componente nel momento in cui l'oggetto che lo contiene viene istanziato;
- È possibile che più oggetti contenitori condividano la stessa componente;
- Il contenimento tramite puntatori può essere utilizzato insieme all'ereditarietà e al polimorfismo per realizzare classi di oggetti che non sono completamente definiti fino al momento in cui il tutto (compreso le parti accessibili tramite puntatori) non è totalmente costruito.

L'ultimo punto è probabilmente il più difficile da capire e richiede la conoscenza dei principi della OOP.

Sostanzialmente possiamo dire che poiché il contenimento avviene tramite puntatori, in effetti non possiamo conoscere l'esatto tipo del componente, ma solo una sua interfaccia generica (classe base) costituita dai messaggi cui l'oggetto puntato sicuramente risponde. Questo rende il contenimento tramite puntatori più flessibile e potente (espressivo) del contenimento diretto, potendo realizzare oggetti il cui comportamento può cambiare dinamicamente nel corso dell'esecuzione del programma.

Pensate al caso di una classe che modelli un'auto utilizzando un puntatore per accedere alla componente motore, se vogliamo testare il comportamento dell'auto con un nuovo motore non dobbiamo fare altro che fare in modo che il puntatore punti ad un nuovo motore. Con il contenimento diretto la struttura del motore (corrispondente ai membri privati della componente) sarebbe stata limitata e non avremmo potuto testare l'auto con un motore di nuova concezione (ad esempio uno a propulsione anziché a scoppio).

Come vedremo invece il polimorfismo consente di superare tale limite. Tutto ciò sarà comunque più chiaro in seguito.

Consideriamo ora i principali vantaggi e svantaggi del contenimento diretto:

- L'accesso ai componenti non deve passare tramite puntatori;
- La struttura di una classe è nota già in fase di compilazione, si conosce subito l'esatto tipo del componente e il compilatore può effettuare molte ottimizzazioni altrimenti impossibili (tipo espansione delle funzioni inline dei componenti);
- Non è necessario eseguire operazioni di allocazione e deallocazione per costruire le componenti, ma è necessario conoscere il modo in cui costruirle già quando si istanzia (costruisce) l'oggetto contenitore.

Se da una parte queste caratteristiche rendono il contenimento diretto meno flessibile ed espressivo di quello tramite puntatore e anche vero che lo rendono più efficiente, non tanto perché non è necessario passare tramite i puntatori, ma quanto per gli ultimi due punti.

Costruttori per oggetti composti

L'inizializzazione di un oggetto composto richiede che siano inizializzate tutte le sue componenti.
Implicitamente abbiamo visto che un attributo non può essere inizializzato mentre lo si dichiara (infatti gli attributi static vanno inizializzati fuori dalla dichiarazione di classe, vedi capitolo VIII, paragrafo 6); la stessa cosa vale per gli attributi di tipo oggetto:

```
class Composed {
  public:
    /* ... */
  private:
    unsigned int Attr = 5;   // Errore!
    Component Elem(10, 5);   // Errore!
    /* ... */
};
```

Il motivo è ovvio, eseguendo l'inizializzazione nel modo appena mostrato il programmatore sarebbe costretto ad inizializzare la componente sempre nello stesso modo; nel caso si desiderasse una inizializzazione alternativa, saremmo costretti a eseguire altre operazioni (e avremmo aggiunto overhead inutile).
La creazione di un oggetto che contiene istanze di altre classi richiede che vengano prima chiamati i costruttori per le componenti e poi quello per l'oggetto stesso; analogamente ma in senso contrario, quando l'oggetto viene distrutto, viene prima chiamato il distruttore per l'oggetto composto, e poi vengono eseguiti i distruttori per le singole componenti.

Il processo può sembrare molto complesso, ma fortunatamente è il compilatore che si occupa di tutta la faccenda, il programmatore deve occuparsi solo dell'oggetto con cui lavora, non delle sue componenti. Al più può capitare che si voglia avere il controllo sui costruttori da utilizzare per le componenti, l'operazione può essere eseguita utilizzando la lista di inizializzazione, come mostra l'esempio seguente:

```
#include <iostream.h>
class SmallObj {
  public:
    SmallObj() {
      cout << "Costruttore SmallObj()" << endl;
    }
    SmallObj(int a, int b) : A1(a), A2(b) {
      cout << "Costruttore SmallObj(int, int)" << endl;
    }
    ~SmallObj() {
      cout << "Distruttore ~SmallObj()" << endl;
    }
  private:
    int A1, A2;
};

class BigObj {
  public:
    BigObj() {
```

```
      cout << "Costruttore BigObj()" << endl;
    }
    BigObj(char c, int a = 0, int b = 1) : Obj(a, b), B(c) {
      cout << "Costruttore BigObj(char, int, int)" << endl;
    }
    ~BigObj() {
      cout << "Distruttore ~BigObj()" << endl;
    }
  private:
    SmallObj Obj;
    char B;
};

void main() {
  BigObj Test(15);
  BigObj Test2;
}
```

il cui output sarebbe:

```
Costruttore SmallObj(int, int)
Costruttore BigObj(char, int, int)
Costruttore SmallObj()
Costruttore BigObj()
Distruttore ~BigObj()
Distruttore ~SmallObj()
Distruttore ~BigObj()
Distruttore ~SmallObj()
```

L'inizializzazione della variabile Test2 viene eseguita tramite il costruttore di default, e poiché questo non chiama esplicitamente un costruttore per la componente SmallObj automaticamente il compilatore aggiunge una chiamata a SmallObj::SmallObj(); nel caso in cui invece desideriamo utilizzare un particolare costruttore per SmallObj bisogna chiamarlo esplicitamente come fatto in BigObj::BigObj(char, int, int) (utilizzato per inizializzare Test).
Il costruttore poteva anche essere scritto nel seguente modo:

```
BigObj::BigObj(char c, int a = 0, int b = 1) {
  Obj = SmallObj(a, b);
  B = c;
  cout << "Costruttore BigObj(char, int, int)" << endl;
}
```

ma benché funzionalmente equivalente al precedente, non genera lo stesso codice. Infatti poiché un costruttore per SmallObj non è esplicitamente chiamato nella lista di inizializzazione e poiché per costruire un oggetto complesso bisogna prima costruire le sue componente, il compilatore esegue una chiamata a SmallObj::SmallObj() e poi passa il controllo a BigObj::BigObj(char, int, int). Conseguenza di ciò è un maggiore overhead dovuto a due chiamate di funzione in più: una per SmallObj::SmallObj() (aggiunta dal compilatore) e l'altra per SmallObj::operator=(SmallObj&) (dovuta alla prima istruzione del costruttore).
Il motivo di un tale comportamento potrebbe sembrare piuttosto arbitrario, tuttavia in realtà un tale scelta è dovuta alla necessità di garantire sempre che un oggetto sia inizializzato prima di essere utilizzato.
Ovviamente poiché ogni classe possiede un solo distruttore, non esistono problemi di scelta!

In pratica possiamo riassumere quanto detto dicendo che:

1. la costruzione di un oggetto composto richiede prima la costruzione delle sue componenti, utilizzando le eventuali specifiche presenti nella lista di inizializzazione del suo costruttore; in caso non venga specificato il costruttore da

utilizzare per una componente, il compilatore utilizza quello di default. Alla fine viene eseguito il corpo del costruttore per l'oggetto composto;
2. la distruzione di un oggetto composto avviene eseguendo prima il suo distruttore e poi il distruttore di ciascuna delle sue componenti;

In quanto detto è sottinteso che se una componete di un oggetto è a sua volta un oggetto composto, il procedimento viene iterato fino a che non si giunge a componenti di tipo primitivo.

Reimpiego di codice con l'ereditarietà

Il meccanismo dell'ereditarietà è per molti aspetti simile a quello della composizione quando si vuole modellare una relazione di tipo *Is-a*.
L'idea è quella di dire al compilatore che una nuova classe (detta classe derivata) è ottenuta da una preesistente (detta classe base) "copiando" il codice di quest'ultima nella classe derivata e modificandolo (eventualmente) con nuove definizioni:

```
class Person {
  public:
    Person();
    ~Person();
    void PrintData();
    /* ... */
  private:
    char * Name;
    unsigned int Age;
    /* ... */
};

class Student : Person {
  public:
    Student();
    ~Student();
    /* ... */
  private:
    unsigned int IdCode;
    /* ... */
};
```

In pratica quanto fatto finora è esattamente la stessa cosa che abbiamo fatto con la composizione (vedi esempio), la differenza è che non abbiamo inserito nella classe Student alcuna istanza della classe Person ma abbiamo detto al compilatore di inserire tutte le dichiarazioni e le definizioni fatte nella classe Person nello scope della classe Student, a tal proposito si dice che la classe derivata eredita i membri della classe base.

Ci sono due sostanziali differenze tra l'ereditarietà e la composizione:

1. Con la composizione ciascuna istanza della classe contenitore possiede al proprio interno una istanza della classe componente; con l'ereditarietà le istanze della classe derivata non contengono nessuna istanza della classe base, le definizioni fatte nella classe base vengono "quasi" immerse tra quelle della classe derivata senza alcuno strato intermedio (il "quasi" è giustificato dal punto 2);
2. Un oggetto composto può accedere solo ai membri pubblici della componente, l'ereditarietà permette invece di accedere direttamente anche ai membri protetti della classe base (quelli privati rimangono inaccessibili alla classe derivata).

Accesso ai campi ereditati

La classe derivata può accedere ai membri protetti e pubblici della classe base come se fossero suoi (e in effetti lo sono):

```
class Person {
  public:
    Person();
    ~Person();
    void PrintData();
    void Sleep();       // nuovo metodo
  private:
    char * Name;
    unsigned int Age;
    /* ... */
};

class Student : Person {
  public:
    Student();
    ~Student();
    void DoNothing();   // nuovo metodo;
  private:
    unsigned int IdCode;
    /* ... */
};

void Student::DoNothing() {
  Sleep();              // richiama Person::Sleep()
}
```

Il codice ereditato continua a comportarsi nella classe derivata esattamente come si comportava nella classe base: se Person::PrintData() visualizzava i membri Name e Age della classe Person, il metodo PrintData() ereditato da Student continuerà a fare esattamente la stessa cosa.
Come alterare dunque tale codice? Tutto quello che bisogna fare è ridefinire il metodo ereditato; c'è però un problema, non possiamo accedere direttamente ai dati privati della classe base. Come fare?
Semplice riutilizzando il metodo che vogliamo ridefinire:

```
class Student : Person {
  public:
    Student();
    ~Student();
    void DoNothing();
    void PrintData();       // ridefinisco il metodo
  private:
    unsigned int IdCode;
    /* ... */
};

void Student::PrintData() {
  Person::PrintData();      // richiama Person::Sleep()
  cout << "Matricola: " << IdCode;
}
```

Si osservi la notazione usata per richiamare il metodo PrintData() della classe Person, se avessimo utilizzato la notazione usuale scrivendo

```
void Student::PrintData() {
  PrintData();
  cout << "Matricola: " << IdCode;    }
```

avremmo commesso un errore, poiché il risultato sarebbe stato una chiamata ricorsiva. Utilizzando il risolutore di scope (::) e il nome della classe base abbiamo invece forzato la chiamata del metodo PrintData() della classe Person.

Un'ultima osservazione...

Se fosse stato possibile avremmo potuto evitare la chiamata di Person::PrintData() utilizzando eventualmente altri membri della classe base, tuttavia è una buona norma della OOP evitare di ridefinire un metodo attribuendogli una semantica radicalmente diversa da quella del metodo originale: se Person::PrintData() aveva il compito di visualizzare lo stato dell'oggetto, anche Student::PrintData() deve avere lo stesso compito. Stando così le cose, richiamare il metodo della classe base significa ridurre la possibilità di commettere un errore.
È per questo motivo infatti che non tutti i membri vengono effettivamente ereditati: costruttori, distruttore, operatore di assegnamento e operatori di conversione di tipo non vengono ereditati perché la loro semantica è troppo legata alla effettiva struttura di una classe (il compilatore comunque continua a fornire per la classe derivata un costruttore di default, uno di copia e un operatore di assegnamento, esattamente come per una qualsiasi altra classe).
Naturalmente la classe derivata può anche definire nuovi membri, compresa la possibilità di eseguire l'overloading di una funzione ereditata.

Ereditarietà pubblica privata e protetta

Per default l'ereditarietà è privata, tutti i membri ereditati diventano cioè membri privati della classe derivata e non sono quindi parte della sua interfaccia. È possibile alterare questo comportamento richiedendo un'ereditarietà protetta o pubblica (è anche possibile richiedere esplicitamente l'ereditarietà privata), ma quello che bisogna sempre ricordare è che non si può comunque allentare il grado di protezione di un membro ereditato (i membri privati rimangono dunque privati e comunque non accessibili alla classe derivata):

- Con l'ereditarietà pubblica i membri ereditati mantengono lo stesso grado di protezione che avevano nella classe da cui si eredita (classe base immediata): i membri public rimangono public e quelli protected continuano ad essere protected;
- Con l'ereditarietà protetta i membri public della classe base divengono membri protected della classe derivata; quelli protected rimangono tali.

La sintassi completa per l'ereditarietà diviene dunque:

```
class <DerivedClassName> : [<Qualifier>] <BaseClassName> {
  /* ... */
};
```

dove Qualifier è opzionale e può essere uno tra public, protected e private; se omesso si assume private.
Lo standard ANSI in via di definizione consente anche la possibilità di esportare singolarmente un membro in presenza di ereditarietà privata o protetta, con l'ovvio limite di non rilasciare il grado di protezione che esso possedeva nella classe base:

```
class MyClass {
  public:
    void PublicMember(int, char);
    /* ... */
  protected:
    int ProtectedMember;
    /* ... */
  private:
    /* ... */
};




class Derived1 : private MyClass {
  public:
    MyClass::PublicMember;     // esporta un membro specifico
    MyClass::ProtectedMember;  // Errore!
  /* ... */
};

class Derived2 : private MyClass {
```

```
  public:
    MyClass::PublicMember;     // Ok!
  protected:
    MyClass::ProtectedMember;  // Ok!
  /* ... */
};

class Derived3 : private MyClass {
  public:
    /* ... */
  protected:
    MyClass::PublicMember;     // Ok era public!
    MyClass::ProtectedMember;  // Ok!
  /* ... */
};
```

L'esempio mostra sostanzialmente tutte le possibili situazioni, compresa il caso di un errore dovuto al tentativo di far diventare public un membro che era protected.
Si noti la notazione utilizzata, non è necessario specificare niente più del semplice nome del membro preceduto dal nome della classe base e dal risolutore di scope (per evitare confusione con una possibile ridefinizione).

La possibilità di esportare singolarmente un membro è stata introdotta per fornire un modo semplice per nascondere all'utente della classe derivata l'interfaccia della classe base, salvo alcune cose; si sarebbe potuto procedere utilizzando l'ereditarietà pubblica e ridefinendo le funzioni che non si desidera esportare in modo che non compiano azioni dannose, il metodo però presenta alcuni inconvenienti:

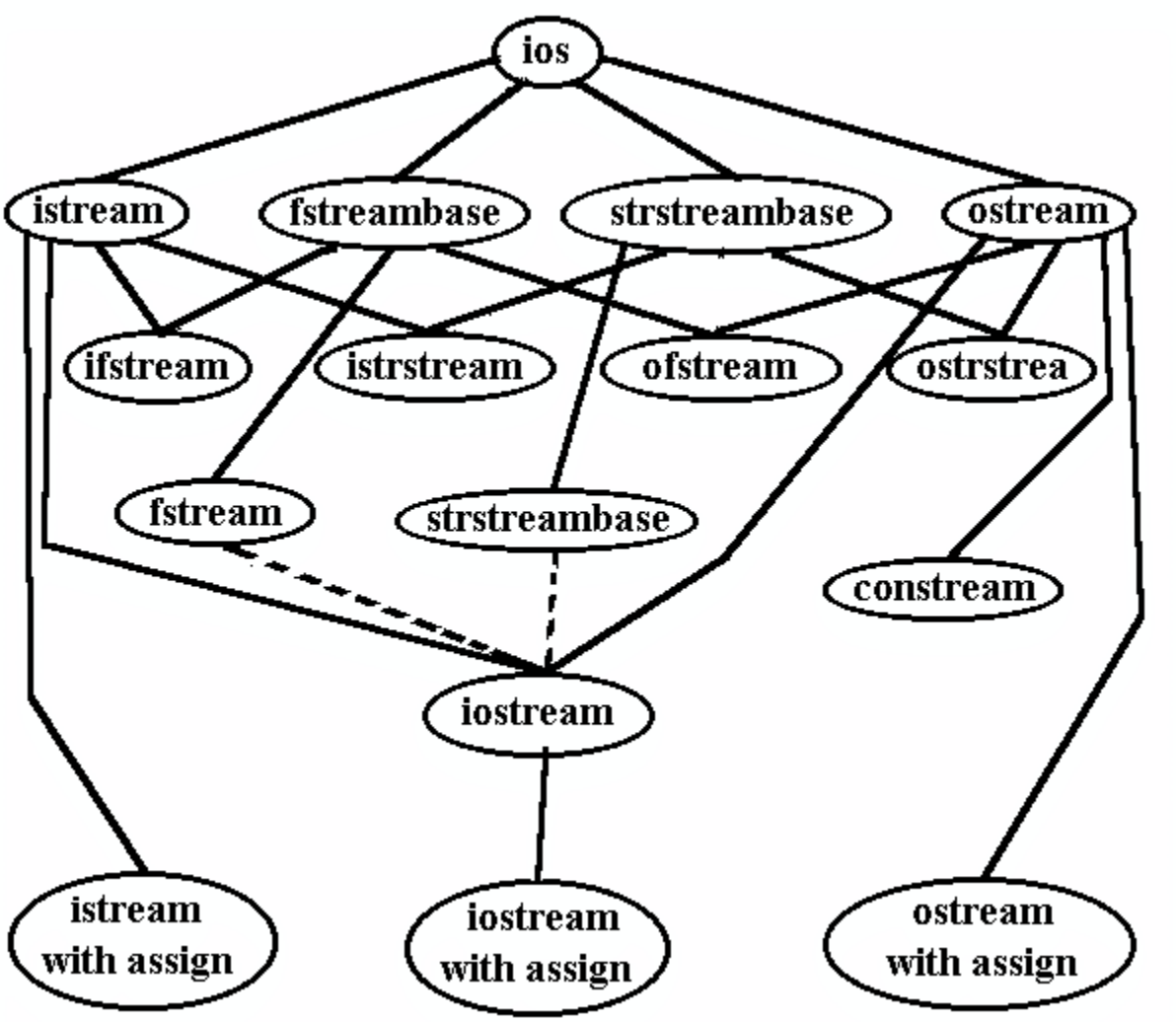
- Il tentativo di utilizzare una funzione non esportata viene segnalato solo a run-time;
- È una operazione che costringe il programmatore a lavorare di più aumentando la possibilità di errore e diminuendone la produttività.

I vari "tipi" di derivazione (ereditarietà) hanno conseguenze che vanno al di là della semplice variazione del livello di protezione di un membro.

Con l'ereditarietà pubblica si modella effettivamente una relazione di tipo *Is-a* poiché la classe derivata continua ad esportare l'interfaccia della classe base (è cioè possibile utilizzare un oggetto derived come un oggetto base); con l'ereditarietà privata questa relazione cessa, a meno che il programmatore non ridichiari l'intera interfaccia della classe base (in un certo senso possiamo vedere l'ereditarietà privata come una sorta di contenimento). L'ereditarietà protetta è invece una sorta di ibrido ed è scarsamente utilizzata.

Ereditarietà multipla

Implicitamente è stato supposto che una classe potesse essere derivata solo da una classe base, in effetti questo è vero per molti linguaggi, tuttavia il C++ consente l'ereditarietà multipla. In questo modo è possibile far ereditare ad una classe le caratteristiche di più classi basi, un esempio è dato dall'implementazione della libreria per l'input/output di cui si riporta il grafo della gerarchia (in alto le classi basi, in basso quelle derivate; fanno eccezione le classi collegate da linee tratteggiate):



La sintassi per l'ereditarietà multipla non si discosta da quella per l'ereditarietà singola, l'unica differenza è che bisogna elencare tutte le classi basi separandole con virgole; al solito se non specificato diversamente per default l'ereditarietà è privata. Ecco un esempio tratto dal grafo precedente:

```
class iostream : public istream, public ostream {
  /* ... */
};
```

L'ereditarietà multipla comporta alcune problematiche che non si presentano in caso di ereditarietà singola, quella a cui si può pensare per prima è il caso in cui le stesse definizioni siano presenti in più classi base (name class):

```
class BaseClass1 {
  public:
    void Foo();
    void Foo2();
    /* ... */
  private:
    int Member;
    /* ... */
};
```

```
class BaseClass2 {
  public:
    void Foo();
    /* ... */
  private:
    int Member;
    void Foo2();
    /* ... */
};

class Derived : BaseClass1, BaseClass2 {
  public:
    void DoSomething();
  /* ... */
};
```

La classe Derived eredita più volte gli stessi membri, e quindi una situazione del tipo

```
void Derived::DoSomething() {
  Member = 10;      // Errore, è ambiguo!
}
```

non può che generare un errore perché il compilatore non sa a quale membro si riferisce l'assegnamento. La soluzione consiste nell'utilizzare il risolutore di scope:

```
void Derived::DoSomething() {
  BaseClass1::Member = 10;      // Ok!
}
```

in questo modo non esiste più alcuna ambiguità.
Si faccia attenzione al fatto che non è necessario che la stessa definizione si trovi in più classi basi, è sufficiente che essa giunga alla classe derivata attraverso due classi basi distinte, ad esempio (con riferimento alla precedenti dichiarazioni):

```
class Derived2 : public BaseClass2 {
  /* ... */
};

class Derived3 : public BaseClass1, public Derived2 {
  /* ... */
};
```

Nuovamente Derived3 presenta lo stesso problema, è cioè sufficiente che la stessa definizione giunga attraverso classi basi indirette (nel precedente esempio BaseClass2 è una classe base indiretta di Derived3).
Il problema diviene più grave quando una o più copie della stessa definizione sono nascoste dalla keyword private nelle classi basi (dirette o indirette), in tal caso la classe derivata non ha alcun controllo su quella o quelle copie (in quanto vi accede indirettamente tramite le funzioni membro ereditate) e il pericolo di inconsistenza dei dati diviene più grave (vedi paragrafo successivo).

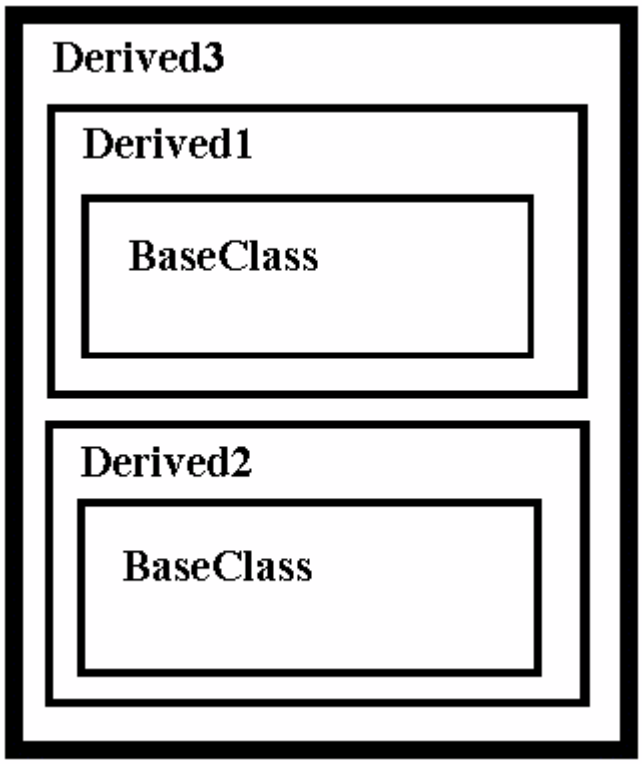
Classi base virtuali

Il problema dell'ambiguità che si verifica con l'ereditarietà multipla, può essere portato al caso estremo in cui una classe ottenuta per ereditarietà multipla erediti più volte una stessa classe base:

```
class BaseClass {
  /* ... */
};
```

```
class Derived1 : public BaseClass {
  /* ... */
};

class Derived2 : private BaseClass {
  /* ... */
};

class Derived3 : public Derived1, private Derived2 {
  /* ... */
};
```

Di nuovo quello che succede è che alcuni membri (in particolare tutta una classe) sono duplicati nella classe Derived3 (anche se una copia di questi non è immediatamente accessibile dalla classe derivata).
Consideriamo l'immagine in memoria di una istanza della classe Derived3, la situazione che avremmo sarebbe la seguente:



La classe Derived3 contiene una istanza di ciascuna delle sue classi base dirette: Derived1 e Derived2.

Ognuna di esse contiene a sua volta una istanza della classe base BaseClass e opera esclusivamente su tale istanza.

In alcuni casi situazioni di questo tipo non creano problemi, ma in generale si tratta di una possibile fonte di inconsistenza.

Supponiamo ad esempio di avere una classe Person e di derivare da essa prima una classe Student e poi una classe Employee al fine di modellare un mondo di persone che eventualmente possono essere studenti o impiegati; dopo un po' ci accorgiamo che una persona può essere contemporaneamente uno studente ed un lavoratore, così tramite l'ereditarietà multipla deriviamo da Student e Employee la classe Student-Employee. Il problema è che la nuova classe contiene due istanze della classe Person e queste due istanze vengono accedute (in lettura e scrittura) indipendentemente l'una dall'altra... Cosa accadrebbe se nelle due istanze venissero memorizzati dati diversi?

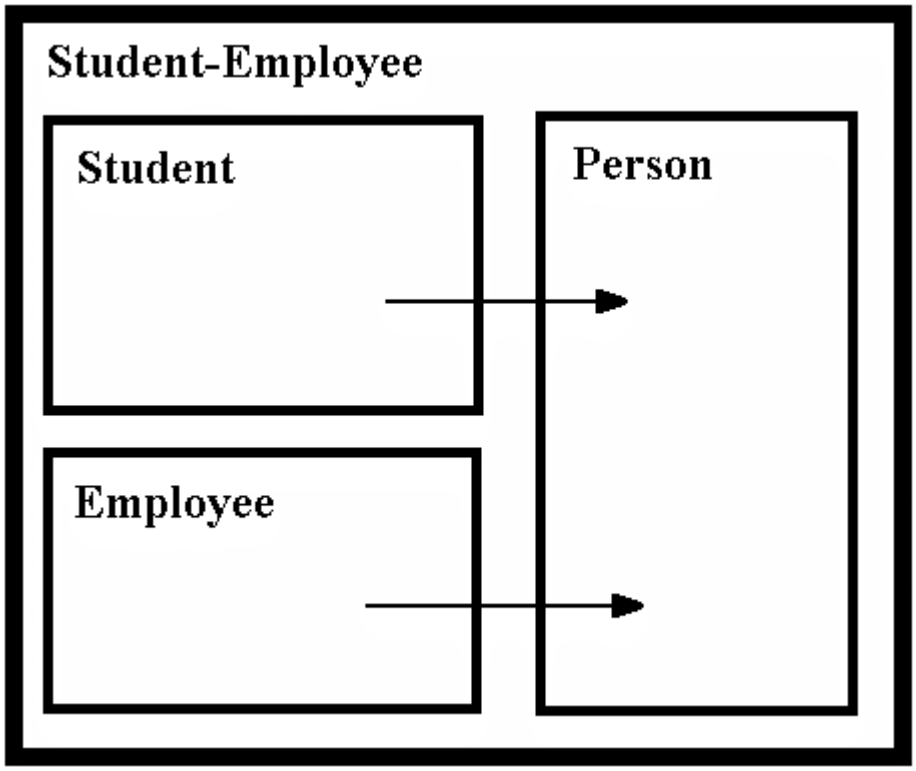
La soluzione viene chiamata ereditarietà virtuale, e la si utilizza nel seguente modo:

```
class Person {
  /* ... */
```

```
};

class Student : virtual private Person {
  /* ... */
};

class Employee : virtual private Person {
  /* ... */
};

class Student-Employee : private Student, private Employee {
  /* ... */
};
```

Quando una classe eredita tramite la keyword virtual il compilatore non si limita a copiare il contenuto della classe base nella classe derivata, ma inserisce nella classe derivata un puntatore ad una istanza della classe base. Quando una classe eredita (per ereditarietà multipla) più volte una classe base virtuale (è questo il caso di Student-Employee che eredita più volte Person), il compilatore inserisce solo una istanza della classe virtuale e fa si che tutti i puntatori a tale classe puntino a quell'unica istanza.

La situazione in questo caso è illustrata dalla seguente figura:



La classe Student-Employee contiene ancora una istanza di ciascuna delle sue classi base dirette: Student e Employee, ma ora esiste una sola istanza della classe base indiretta Person poiché essa è stata dichiarata virtual nelle definizioni di Student e Employee.

Il puntatore alla classe base virtuale non è visibile al programmatore, non bisogna tenere conto di esso poiché viene aggiunto dal compilatore a compile-time, semplicemente si accede ai membri della classe base virtuale come si farebbe con una normale classe base.
Il vantaggio di questa tecnica è che non è più necessario definire la classe Student-Employee derivandola da Person (al fine di eliminare la fonte di inconsistenza), in tal modo si risparmiano tempo e fatica riducendo la quantità di codice da produrre e limitando la possibilità di errori e la quantità di memoria necessaria al nostro programma per girare. C'è però un costo da pagare: un livello di indirezione in più perché l'accesso alle classi base virtuali (nell'esempio Person) avviene tramite un puntatore.

L'ereditarietà virtuale genera anche un nuovo problema: il costruttore di una classe derivata chiama i costruttori delle classi base e nel caso di ereditarietà virtuale una determinata classe base virtuale potrebbe essere inizializzata più volte.
Nel nostro esempio la classe base virtuale Person è inizializzata sia da Student che da Employee, entrambe le classi hanno il dovere di eseguire la chiamata al costruttore della classe base, ma quando queste due classi vengono fuse per derivare la classe Student-Employee il costruttore della nuova classe, chiamando i costruttori di Student e Employee, implicitamente chiamerebbe due volte il costruttore di Person.

Per evitare tale comportamento è stato deciso che i costruttori di una classe che possieda una classe base (diretta o indiretta) virtuale debbano eseguire esplicitamente una chiamata ad un costruttore della classe virtuale, il compilatore farà poi in modo che l'inizializzazione sia effettivamente eseguita solo dalla classe massimamente derivata (ovvero quella cui appartiene l'istanza che si sta creando). In questo modo ogni classe base virtuale è inizializzata una sola volta e in modo deterministico.

Il seguente codice

```
Person::Person() {
  cout << "Costruttore Person invocato..." << endl;
}

Student::Student() : Person() {
  cout << "Costruttore Student invocato..." << endl;
}

Employee::Employee() : Person() {
  cout << "Costruttore Employee invocato..." << endl;
}

Student-Employee::Student-Employee() : Person(), Student(), Employee() {
  cout << "Costruttore Student-Employee invocato..." << endl;
}

/* ... */

cout << "Definizione di Tizio:" << endl;
Person Tizio;
cout << endl << "Definizione di Caio:" << endl;
Student Caio;
cout << endl << "Definizione di Sempronio:" << endl;
Employee Sempronio;
cout << endl << "Definizione di Bruto:" << endl;
Student-Employee Bruto;
```

opportunamente completato, produrrebbe il seguente output:

```
Definizione di Tizio:
Costruttore Person invocato...

Definizione di Caio:
Costruttore Person invocato...
Costruttore Student invocato...



Definizione di Sempronio:
Costruttore Person invocato...
Costruttore Employee invocato...

Definizione di Sempronio:
Costruttore Person invocato...
```

```
Costruttore Student invocato...
Costruttore Employee invocato...
Costruttore Student-Employee invocato...
```

Come potete osservare il costruttore della classe Person viene invocato una sola volta, per verificare poi da chi viene invocato basta tracciare l'esecuzione con un debugger simbolico.
Analogamente anche i distruttori seguono una regola simile, solo che in questo caso viene eseguito tutto dal compilatore (i distruttori non devono mai essere invocati esplicitamente).

Funzioni virtuali

Il meccanismo dell'ereditarietà è stato già di per se una grande innovazione nel mondo della programmazione, tuttavia le sorprese non si esauriscono qui. Esiste un'altra caratteristica tipica dei linguaggi a oggetti (C++ incluso) che ha valso loro il soprannome di "Linguaggi degli attori", tale caratteristica consiste nella possibilità di rimandare a tempo di esecuzione il linking di una o più funzioni membro (late-binding).
L'ereditarietà pone nuove regole circa la compatibilità dei tipi, in particolare se Ptr è un puntatore di tipo T, allora Ptr può puntare non solo a istanze di tipo T ma anche a istanze di classi derivate da T (sia tramite ereditarietà semplice che multipla). Se Td è una classe derivata (anche indirettamente) da T, istruzioni del tipo

```
T * Ptr = 0;                // Puntatore nullo
/* ... */
Ptr = new Td;
```

sono assolutamente lecite e il compilatore non segnala ne errori ne warning.

Ciò consente ad esempio la realizzazione di una lista per contenere tutta una serie di istanze di una gerarchia di classi, magari per poter eseguire un loop su di essa e inviare a tutti gli oggetti della lista uno stesso messaggio. Pensate ad esempio ad un programma di disegno che memorizza gli oggetti disegnati mantenendoli in una lista, ogni oggetto sa come disegnarsi e se è necessario ridisegnare tutto il disegno basta scorrere la lista inviando ad ogni oggetto il messaggio di Paint.

Purtroppo la cosa così com'è non può funzionare poiché le funzioni sono linkate staticamente a compile-time. Anche se tutte le classi della gerarchia possiedono un metodo Paint(), noi sappiamo solo che Ptr punta ad un oggetto di tipo T o T-derivato, non conoscendo l'esatto tipo una chiamata a Ptr->Paint() non può che essere risolta chiamando Ptr->T::Paint() (che non farà ciò che vorremmo).
Il compilatore non può infatti rischiare di chiamare il metodo di una classe derivata, poiché questo potrebbe tentare di accedere a membri che non fanno parte dell'effettivo tipo dell'oggetto (causando inconsistenze o un crash del sistema), chiamando il metodo della classe T al più il programma non farà la cosa giusta, ma non metterà in pericolo la sicurezza e l'affidabilità del sistema (perché un oggetto derivato possiede tutti i membri della classe base).
Si potrebbe risolvere il problema inserendo in ogni classe della gerarchia un campo che stia ad indicare l'effettivo tipo dell'istanza:

```
enum TypeId { T-Type, Td-Type };

class T {
  public:
    TypeId Type;
    /* ... */
  private:
    /* ... */
};
```

```
class Td : public T {
  /* ... */
};
```

e risolvere il problema con una istruzione switch:

```
switch Ptr->Type {
  case T-Type :
    Ptr->T::Paint();
    break;
  case Td-Type :
    Ptr->Td::Paint();
  default :
    /* errore */
};
```

Una soluzione di questo tipo funziona ma è macchinosa, allunga il lavoro, una dimenticanza può costare cara e soprattutto ogni volta che si modifica la gerarchia di classi bisogna modificare anche il codice che la usa.
La soluzione migliore è invece quella di far in modo che il corretto tipo dell'oggetto puntato sia automaticamente determinato al momento della chiamata della funzione e rinviando il linking di tale funzione a run-time.
Per fare ciò bisogna dichiarare la funzione membro virtual:

```
class T {
  public:
    /* ... */
    virtual void Paint();
  /* ... */
};
```

La definizione del metodo procede poi nel solito modo:

```
void T::Paint() {           // non occorre mettere virtual
  /* ... */
}
```

I metodi virtuali vengono ereditati allo stesso modo di quelli "normali" (o meglio statici), possono anch'essi essere sottoposti a overloading ed essere ridefiniti, non c'è alcuna differenza eccetto che una loro invocazione non viene risolta se non a run-time. Quando una classe possiede un metodo virtuale, il compilatore associa alla classe (non all'istanza) una tabella che contiene per ogni metodo virtuale l'indirizzo alla corrispondente funzione, ogni istanza di quella classe conterrà poi al suo interno un puntatore alla tabella; una chiamata ad una funzione membro virtuale (e solo alle funzioni virtuali) viene risolta con del codice che accede alla tabella corrispondente al tipo dell'istanza tramite il puntatore contenuto nell'istanza stessa, ottenuta la tabella invocare il metodo corretto è semplice.

Le funzioni virtuali hanno il grande vantaggio di consentire l'aggiunta di nuove classi alla gerarchia e di renderle immediatamente e correttamente utilizzabili dal vostro programma senza doverne modificare il codice, basta solo ricompilare il tutto, il late-binding farà in modo che siano chiamate sempre le funzioni corrette senza che il vostro programma debba curarsi dell'effettivo tipo dell'istanza che sta manipolando.
L'invocazione di un metodo virtuale è più costosa di quella per una funzione membro ordinaria, tuttavia il compilatore può evitare tale overhead risolvendo a compile-time tutte quelle situazioni in cui il tipo è effettivamente noto; ad esempio:

```
Td Obj1;
T * Ptr = 0;
/* ... */
Obj1.Paint();       // Chiamata risolvibile staticamente
Ptr->Paint();       // Questa invece no
```

La prima chiamata al metodo Paint() può essere risolta in fase di compilazione perché il tipo di Obj1 è sicuramente Td, nel secondo caso invece non possiamo saperlo (anche se un compilatore intelligente potrebbe cercare di restringere le possibilità

e, in caso di certezza assoluta, risolvere staticamente la chiamata). Se poi volete avere il massimo controllo, potete costringere il compilatore ad una "soluzione statica" utilizzando il risolutore di scope:

```
Td Obj1;
T * Ptr = 0;
/* ... */
Obj1.Td::Paint();       // Chiamata risolta staticamente
Ptr->Td::Paint();       // ora anche questa.
```

Adesso sia nel primo che nel secondo caso, il metodo invocato è Td::Paint(). Fate attenzione però ad utilizzare questa possibilità con i puntatori (come nell'ultimo caso), se per caso il tipo corretto dell'istanza puntata non corrisponde, potreste avere delle brutte sorprese.

Il meccanismo delle funzioni virtuali è alla base del polimorfismo: poiché l'oggetto puntato da un puntatore può appartenere a tutta una gerarchia di tipi, possiamo considerare l'istanza puntata come un qualcosa che può assumere più forme (tipi) e comportarsi sempre nel modo migliore "recitando" di volta in volta il ruolo corretto (da qui il soprannome di "Linguaggi degli attori"), in realtà però un'istanza non può cambiare tipo, e solo il puntatore che può cambiare tipo e istanza.
Se decidete di utilizzare le funzioni virtuali dovete ricordare che quando derivate una nuova classe, se decidete di ridefinire un metodo virtuale di una classe base, esso dovrà essere dichiarato ancora *virtual*, altrimenti il meccanismo viene interrotto. Fate attenzione anche in casi di questo tipo:

```
class T {
  public:
    virtual void Foo();
    virtual void Foo2();
    void DoSomething();
  private:
    /* ... */
};

/* implementazione di T::Foo() e T::Foo2() */

void T::DoSomething() {
  /* ... */
  Foo();
  /* ... */
  Foo2();
  /* ... */
}

class Td : public T {
  public:
    virtual void Foo();
    void DoSomething();
  private:
    /* ... */    };

/* implementazione di Td::Foo2() */

void Td::DoSomething() {
  /* ... */
  Foo();            // attenzione chiama T::Foo()
  /* ... */
  Foo2();
  /* ... */    }
```

Si tratta di una situazione pericolosa: la classe Td ridefinisce un metodo statico (ma poteva anche essere virtuale), ma non uno virtuale da questo richiamato. Di per se non si tratta di un errore, la classe derivata potrebbe non aver alcun motivo per ridefinire il metodo ereditato, tuttavia può essere difficile capire cosa esattamente faccia il metodo Td::DoSomething(), soprattutto in un caso simile:

```
class Td2 : public Td {
  public:
    virtual void Foo();
  private:
    /* ... */
};
```

Questa nuova classe ridefinisce un metodo virtuale, ma non quello che lo chiama, per cui in una situazione del tipo:

```
Td2 * Ptr = new Td2;
/* ... */
Ptr->DoSomething();
```

viene chiamato il metodo Td::DoSomething() ereditato, ma in effetti questo poi chiama Td2::Foo() per via del linking dinamico. Consiglio vivamente di riflettere sull'evoluzione di una esecuzione di funzione che chiami funzioni virtuali, solo in questo modo si apprendono vantaggi e pericoli derivanti dall'uso di funzioni virtuali.

Per concludere l'argomento resta solo da dire che qualsiasi funzione membro di una classe può essere dichiarata virtual (anche un operatore, come vedremo), con l'eccezione dei costruttori. I costruttori infatti, in presenza di classi con funzioni virtuali, hanno il compito di inizializzare il puntatore alla tabella dei metodi virtuali contenuto nell'istanza e quindi non essendo ancora stabilito il link tra istanza e tabella non è materialmente possibile avere costruttori virtuali. Si noti inoltre che non conoscendo il momento in cui tale link sarà stato stabilito, non è lecito chiamare metodi virtuali all'interno dei costruttori (a meno che non si forzi il linking statico con il risolutore di scope).

Ovviamente non esiste alcun motivo per cui un distruttore non possa essere virtuale, anzi in presenza di funzioni virtuali è sempre bene che anche il distruttore lo sia (al 99,9% è necessario, negli altri casi è una garanzia per il funzionamento del programma, soprattutto in previsione di future revisioni).

Classi astratte

I meccanismi dell'ereditarietà e delle funzioni virtuali possono essere combinati per realizzare delle classi il cui unico scopo è quello di stabilire una interfaccia comune a tutta una gerarchia di classi:

```
class TShape {
  public:
    virtual void Paint() = 0;
    virtual void Erase() = 0;
    /* ... */
};
```

Notate l'assegnamento effettuato alle funzioni virtuali, funzioni di questo tipo vengono dette funzioni virtuali pure e l'assegnamento ha il compito di informare il compilatore che non intendiamo definire i metodi virtuali. Una classe che possiede funzioni virtuali pure è detta classe astratta e non è possibile istanziarla; essa può essere utilizzata unicamente per derivare nuove classi forzandole a fornire determinati metodi (quelli corrispondenti alle funzioni virtuali pure). Il compito di una classe astratta è quella di fornire una interfaccia senza esporre dettagli implementativi. Se una classe derivata da una classe astratta non implementa una qualche funzione virtuale pura, diviene essa stessa una classe astratta.

Le classi astratte possono comunque possedere anche attributi e metodi completamente definiti (costruttori e distruttore compresi) ma non possono comunque essere istanziate, servono solo per consentire la costruzione di una gerarchia di classi secondo un ordine incrementale:

```
class TShape {
  public:
    virtual void Paint() = 0;    // ogni figura può essere
    virtual void Erase() = 0;    // disegnata e cancellata!
};
```

```
class TPoint : public TShape {
  public:
    TPoint(int x, int y) : X(x), Y(y) {}
  private:
    int X, Y;                    // coordinate del punto
};

void TPoint::Paint() {
  /* traccia il punto */
}

void TPoint::Erase() {
  /* lo rimuove */
}
```

Non è possibile creare istanze della classe TShape, ma la classe TPoint ridefinisce tutte le funzioni virtuali pure e può essere istanziata e utilizzata dal programma; la classe TShape è comunque ancora utile al programma, perché possiamo dichiarare puntatori di tale tipo per gestire una lista di figure.

L'overloading degli operatori

Ogni linguaggio di programmazione è concepito per soddisfare determinati requisiti; i linguaggi procedurali (come il C) sono stati concepiti per realizzare applicazioni che non richiedano nel tempo più di poche modifiche. Al contrario i linguaggi a oggetti hanno come obiettivo l'estendibilità, il programmatore è in grado di estendere il linguaggio per adattarlo al problema da risolvere, in tal modo diviene più semplice modificare programmi creati precedentemente perché via via che il problema cambia, il linguaggio si adatta.

Famoso in tal senso è stato FORTH, un linguaggio totalmente estensibile (senza alcuna limitazione), tuttavia nel caso di FORTH questa grande libertà si rivelò controproducente perché spesso solo gli ideatori di un programma erano in grado di comprendere il codice.

Anche il C++ può essere esteso, solo che per evitare i problemi di FORTH vengono posti dei limiti: l'estensione del linguaggio avviene introducendo nuove classi, definendo nuove funzioni e (vedremo ora) eseguendo l'overloading degli operatori; queste modifiche devono tuttavia sottostare a precise regole, ovvero essere sintatticamente corrette per il vecchio linguaggio (in pratica devono seguire le regole precedentemente viste e quelle che vedremo adesso).

Le prime regole

Così come la definizione di classe deve soddisfare precise regole sintattiche e semantiche, così l'overloading di un operatore deve soddisfare un opportuno insieme di requisiti:

1. Non è possibile definire nuovi operatori, si può solamente eseguire l'overloading di uno per cui esiste già un simbolo nel linguaggio. Possiamo ad esempio definire un nuovo operatore *, ma non possiamo definire un operatore **. Questa regola ha lo scopo di prevenire possibili ambiguità.
2. Non è possibile modificare la precedenza di un operatore e non è possibile modificarne l'arietà o l'associatività, un operatore unario rimarrà sempre unario, uno binario dovrà applicarsi sempre a due operandi; analogamente uno associativo a sinistra rimarrà sempre associativo a sinistra.
3. Non è concessa la possibilità di eseguire l'overloading dell'operatore ternario ? : .
4. È possibile ridefinire un operatore sia come funzione globale che come funzione membro, i seguenti operatori devono tuttavia essere sempre funzioni membro non statiche: operatore di assegnamento (=), operatore di sottoscrizione ([ ]) e operatore -> .

A parte queste poche restrizioni non esistono altri limiti, possiamo ridefinire anche l'operatore virgola (,) e persino l'operatore chiamata di funzione (()); inoltre non c'è alcuna restrizione riguardo il contenuto del corpo di un operatore: *un operatore altro non è che un tipo particolare di funzione* e tutto ciò che può essere fatto in una funzione può essere fatto anche in un operatore.

Un operatore è indicato dalla keyword operator seguita dal simbolo dell'operatore, per eseguirne l'overloading come funzione globale bisogna utilizzare la seguente sintassi:

```
<ReturnType> operator@(<ArgumentList>) { <Body> }
```

ReturnType è il tipo restituito (non ci sono restrizioni); @ indica un qualsiasi simbolo di operatore valido; *ArgumentList* è la lista di parametri (tipo e nome) che l'operatore riceve, i parametri sono due per un operatore binario (il primo è quello che compare a sinistra dell'operatore quando esso viene applicato) mentre è uno solo per un operatore unario. Infine Body è la sequenza di istruzioni che costituiscono il corpo dell'operatore.
Ecco un esempio di overloading di un operatore come funzione globale:

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

Complex operator+(const Complex & A, const Complex & B) {
```

```
  Complex Result;
  Result.Re = A.Re + B.Re;
  Result.Im = A.Im + B.Im;
  return Result;
}
```

Si tratta sicuramente di un caso molto semplice, che fa capire che in fondo un operatore altro non è che una funzione. Il funzionamento del codice è chiaro e non mi dilungherò oltre; si noti solo che i parametri sono passati per riferimento, non è obbligatorio, ma solitamente è bene passare i parametri in questo modo (eventualmente utilizzando const come nell'esempio).
Definito l'operatore, è possibile utilizzarlo secondo l'usuale sintassi riservata agli operatori, ovvero come nel seguente esempio:

```
Complex A, B;
/* ... */
Complex C = A+B;
```

L'esempio richiede che sia definito su Complex il costruttore di copia, ma come già sapete il compilatore è in grado di fornirne uno di default. Detto questo il precedente esempio viene tradotto (dal compilatore) in

```
Complex C(operator+(A, B));
```

Volendo potete utilizzare gli operatori come funzioni, esattamente come li traduce il compilatore (cioè scrivendo Complex C = operator+(A, B) o Complex C(operator+(A, B))), ma non è una buona pratica in quanto annulla il vantaggio ottenuto ridefinendo l'operatore.

Quando un operatore viene ridefinito come funzione membro il primo parametro è sempre l'istanza della classe su cui viene eseguito e non bisogna indicarlo nella lista di argomenti, un operatore binario quindi come funzione globale riceve due parametri ma come funzione membro ne riceve solo uno (il secondo operando); analogamente un operatore unario come funzione globale prende un solo argomento, ma come funzione membro ha la lista di argomenti vuota.
Riprendiamo il nostro esempio di prima ampliandolo con nuovi operatori:

```
class Complex {
  public:
    Complex(float re, float im);
    Complex operator-() const;              // - unario
    Complex operator+(const Complex & B) const;
    const Complex & operator=(const Complex & B);
  private:
    float Re;
    float Im;
};

Complex::Complex(float re, float im = 0.0) {
  Re = re;
  Im = im;
}

Complex Complex::operator-() const {
  return Complex(-Re, -Im);
}

Complex Complex::operator+(const Complex & B) const {
  return Complex(Re+B.Re, Im+B.Im);
}

const Complex & Complex::operator=(const Complex & B) {
  Re = B.Re;
  Im = B.Im;
  return *this;
```

```
}
```

La classe Complex ridefinisce tre operatori. Il primo è il - (meno) unario, il compilatore capisce che si tratta del meno unario dalla lista di argomenti vuota, il meno binario invece, come funzione membro, deve avere un parametro. Successivamente viene ridefinito l'operatore + (somma), si noti la differenza rispetto alla versione globale. Infine viene ridefinito l'operatore di assegnamento che come detto sopra deve essere una funzione membro non statica; si noti che a differenza dei primi due questo operatore ritorna un riferimento, in tal modo possiamo concatenare più assegnamenti evitando la creazione di inutili temporanei, l'uso di const assicura che il risultato non venga utilizzato per modificare l'oggetto.

Infine, altra osservazione, l'ultimo operatore non è dichiarato const in quanto modifica l'oggetto su cui è applicato (quello cui si assegna), se la semantica che volete attribuirgli consente di dichiararlo const fatelo, ma nel caso dell'operatore di assegnamento (e in generale di tutti) è consigliabile mantenere la coerenza semantica (cioè ridefinirlo sempre come operatore di assegnamento, e non ad esempio come operatore di uguaglianza).

Ecco alcuni esempi di applicazione dei precedenti operatori e la loro rispettiva traduzione in chiamate di funzioni (A, B e C sono variabili di tipo Complex):

```
B = -A;       // analogo a B.operator=(A.operator-());
C = A+B;      // analogo a C.operator=(A.operator+(B));
C = A+(-B);   // analogo a C.operator=(A.operator+(B.operator-()))
C = A-B;      // errore!
              // complex & Complex::operator-(Complex &) non definito.
```

L'ultimo esempio è errato poiché quello che si vuole utilizzare è il meno binario, e tale operatore non è stato definito.
Passiamo ora ad esaminare con maggiore dettaglio alcuni operatori che solitamente svolgono ruoli più difficili da capire.

L'operatore di assegnamento

L'assegnamento è un operatore molto particolare, la sua semantica classica è quella di modificare il valore dell'oggetto cui è applicato con quello ricevuto come parametro, l'operatore ritorna poi il valore che ha assegnato all'oggetto e ciò, grazie all'associatività a destra, consente espressioni del tipo

```
A = B = C = <Valore>
```

Questa espressione è equivalente a

```
A = (B = (C = <Valore>));
```

Non lo si confonda con il costruttore di copia: il costruttore è utilizzato per costruire un nuovo oggetto inizializzandolo con il valore di un altro, l'assegnamento viene utilizzato su oggetti già costruiti.

```
Complex C = B;          // Costruttore di copia
/* ... */
C = D;                  // Assegnamento
```

Un'altra particolarità di questo operatore lo rende simile al costruttore (oltre al fatto che deve essere una funzione membro): se in una classe non ne viene definito uno nella forma X::operator=(X&), il compilatore ne fornisce uno che esegue la copia bit a bit. Il draft ANSI-C++ stabilisce che sia il costruttore di copia che l'operatore di assegnamento forniti dal compilatore debbano eseguire non una copia bit a bit, ma una inizializzazione o assegnamento a livello di membri chiamando il costruttore di copia o l'operatore di assegnamento relativi al tipo di quel membro. In ogni caso comunque e necessario definire esplicitamente sia l'operatore di assegnamento che il costruttore di copia ogni qual volta la classe contenga puntatori, onde evitare spiacevoli condivisioni di memoria.

Notate infine che, come per le funzioni, anche per un operatore è possibile avere più versioni overloaded; in particolare una classe può dichiarare più operatori di assegnamento, ma è quello di cui sopra che il compilatore fornisce quando manca.

L'operatore di sottoscrizione

Un altro operatore un po' particolare è quello di sottoscrizione [ ]. Si tratta di un operatore binario il cui primo operando è l'argomento che appare a sinistra di [, mentre il secondo è quello che si trova tra le parentesi quadre. La semantica classica associata a questo operatore prevede che il primo argomento sia un puntatore, mentre il secondo argomento deve essere un intero senza segno.

Il risultato dell'espressione Arg1[Arg2] è dato da *(Arg1+Arg2*sizeof(T)) dove T è il tipo del puntatore; in pratica si somma al puntatore una quantità tale da ottenere un puntatore che punti più avanti di Arg2-1 celle di tipo T.

Questo operatore può essere ridefinito unicamente come funzione membro non statica e ovviamente non è tenuto a sottostare al significato classico dell'operatore fornito dal linguaggio. Il problema principale che si riscontra nella definizione di questo operatore è fare in modo che sia possibile utilizzare indici multipli, ovvero poter scrivere Arg1[Arg2][Arg3]; il trucco per riuscire in ciò consiste semplicemente nel restituire un riferimento al tipo di Arg1, ovvero seguire il seguente prototipo:

```
X & X::operator[](T Arg2);
```

dove T può anche un riferimento o un puntatore.
Restituendo un riferimento l'espressione Arg1[Arg2][Arg3] viene tradotta in Arg1.operator[](Arg2).operator[](Arg3).
Il seguente codice mostra un esempio di overloading di questo operatore:

```
class TArray {
  public:
    TArray(unsigned int Size);
    ~TArray();
    int operator[](unsigned int Index);
  private:
    int * Array;
    unsigned int ArraySize;
};

TArray::TArray(unsigned int Size) {
  ArraySize = Size;
  Array = new int[Size];
}

TArray::~TArray() {
  delete[] Array;
}

int TArray::operator[](unsigned int Index) {
  if (Index<Size) return Array[Index];
  else /* Errore */
}
```

Si tratta di una classe che incapsula il concetto di array per effettuare dei controlli sull'indice, evitando così accessi fuori limite. La gestione della situazione di errore è stata appositamente omessa, vedremo meglio come gestire queste situazioni quando parleremo di eccezioni.
Notate che l'operatore di sottoscrizione restituisce un int e non è pertanto possibile usare indicizzazioni multiple, d'altronde la classe è stata concepita unicamente per realizzare array monodimensionali di interi; una soluzione migliore, più flessibile e generale avrebbe richiesto l'uso dei template che saranno argomento del successivo capitolo.

Operatori && e ||

Anche gli operatori di AND e OR logico possono essere ridefiniti, tuttavia c'è una profonda differenza tra quelli predefiniti e quelli che l'utente può definire. La versione predefinita di entrambi gli operatori eseguono valutazioni parziali degli

argomenti: l'operatore valuta l'operando di sinistra, ma valuta anche quello di destra solo quando il risultato dell'operazione è ancora incerto. In questi esempi l'operando di destra non viene mai valutato:

```
int var1 = 1;
int var2 = 0;

int var3 = var2 && var1;
var3 = var1 || var2;
```

In entrambi i casi il secondo operando non viene valutato poiché il valore del primo è sufficiente a stabilire il risultato dell'espressione.
Le versioni sovraccaricate definite dall'utente non si comportano in questo modo, entrambi gli argomenti dell'operatore sono sempre valutati (al momento in cui vengono passati come parametri).

Smart pointer

Un operatore particolarmente interessante è quello di dereferenziazione −> il cui comportamento è un po' difficile da capire. Se T è una classe che ridefinisce −> (l'operatore di dereferenziazione deve essere un funzione membro non statica) e Obj è una istanza di tale classe, l'espressione

```
Obj->Field;
```

è valutata come

```
(Obj.operator->())->Fielg;
```

Conseguenza di ciò è che il risultato di questo operatore deve essere uno tra

- un puntatore ad una struttura o una classe che contiene un membro Field;
- una istanza di un'altra classe che ridefinisce a sua volta l'operatore. In questo caso l'operatore viene applicato ricorsivamente all'oggetto ottenuto prima, fino a quando non si ricade nel caso precedente;

In questo modo è possibile realizzare puntatori intelligenti (smart pointer), capaci di eseguire controlli per prevenire errori disastrosi.
Pur essendo un operatore unario postfisso, il modo in cui viene trattato impone che ci sia sul lato destro una specie di secondo operando; se volete potete pensare che l'operatore predefinito sia in realtà un operatore binario il cui secondo argomento è il nome del campo di una struttura, mentre l'operatore che l'utente può ridefinire deve essere unario.

L'operatore virgola

Anche la virgola è un operatore (binario) che può essere ridefinito. La versione predefinita dell'operatore fa sì che entrambi gli argomenti siano valutati, ma il risultato prodotto è il valore del secondo (quello del primo argomento viene scartato).

Nella prassi comune, la virgola è utilizzata per gli effetti collaterali derivanti dalla valutazione delle espressioni:

```
int A = 5;
int B = 6;
int C = 10;

int D = (++A, B+C);
```

In questo esempio il valore assegnato a D è quello ottenuto dalla somma di B e C, mentre l'espressione a sinistra della virgola serve per incrementare A. A sinistra della virgola poteva esserci una chiamata di funzione (a patto che il valore restituito fosse del tipo opportuno), che serviva solo per alcuni suoi effetti collaterali. Quanto alle parentesi, esse sono necessarie perché l'assegnamento ha la precedenza sulla virgola. Questo operatore è comunque sovraccaricato raramente.

Autoincremento e autodecremento

Gli operatori ++ e -- meritano un breve accenno poiché esistono entrambi sia come operatori unari prefissi che unari postfissi.
Le prime versioni del linguaggio non consentivano di distinguere tra le due forme, la stessa definizione veniva utilizzata per le due sintassi. Le nuove versioni del linguaggi consentono invece di distinguere e usano due diverse definizioni per i due possibili casi.
La forma prefissa prende un solo argomento: l'oggetto cui è applicato, la forma postfissa invece possiede un parametro fittizio in più di tipo int. I prototipi delle due forme di entrambi gli operatori per gli interi sono ad esempio le seguenti:

```
int operator++(int A);        // caso ++Var
int operator++(int A, int);   // caso Var++
int operator--(int A);        // caso --Var
int operator--(int A, int);   // caso Var--
```

Il parametro fittizio non ha un nome e non è possibile accedere ad esso.

New e delete

Neanche gli operatori new e delete fanno eccezione, anche loro possono essere ridefiniti sia a livello di classe o addirittura globalmente.
Sia come funzioni globali che come funzioni membro, la new riceve un parametro di tipo size_t che al momento della chiamata è automaticamente inizializzato con il numero di byte da allocare e deve restituire sempre un void *; la delete invece riceve un void * e non ritorna alcun risultato (va dichiarata void). Anche se non esplicitamente dichiarate, come funzioni membro i due operatori sono sempre static.

Poiché entrambi gli operatori hanno un prototipo predefinito, non è possibile avere più versioni overloaded, è possibile averne al più una unica definizione globale e una sola definizione per classe come funzione membro. Se una classe ridefinisce questi operatori (o uno dei due) la funzione membro viene utilizzata al posto di quella globale per gli oggetti di tale classe; quella globale definita (anch'essa eventualmente ridefinita dall'utente) sarà utilizzata in tutti gli altri casi.

La ridefinizione di new e delete è solitamente effettuata in programmi che fanno massiccio uso dello heap al fine di evitarne una eccessiva frammentazione e soprattutto per ridurre l'overhead globale introdotto dalle singole chiamate. La ridefinizione di questi operatori richiede l'inclusione del file new.h fornito con tutti i compilatori.
Ecco un esempio di new e delete globali:

```
void * operator new(size_t Size) {
  return malloc(Size);
}

void operator delete(void * Ptr) {
  free(Ptr);     }
```

Le funzioni malloc() e free() richiedono al sistema (rispettivamente) l'allocazione di un blocco di Size byte o la sua deallocazione (in quest'ultimo caso non è necessario indicare il numero di byte).
Sia new che delete possono accettare un secondo parametro, nel caso di new ha tipo void * e nel caso della delete è di tipo size_t : nel caso della new il secondo parametro serve per consentire una allocazione di un blocco di memoria ad un indirizzo specifico (ad esempio per mappare in memoria un dispositivo hardware), mentre nel caso della delete il suo compito è di fornire la dimensione del blocco da deallocare (utile in parecchi casi). Nel caso in cui lo si utilizzi, è compito del programmatore supplire un valore per il secondo parametro (in effetti solo per il primo parametro della new è il compilatore che fornisce il valore).
Ecco un esempio di new che utilizza il secondo parametro :

```
void * operator new(size_t Size, void * Ptr = 0) {
  if (Ptr) return Ptr;
  return malloc(Size);
```

```
}
```

Questa new permette proprio la mappatura in memoria di un dispositivo hardware.

Per concludere c'è da dire che allo stato attuale non è possibile ridefinire le versioni per array di new e delete, non potete quindi ridefinire gli operatori new[ ] e delete[ ].

Conclusioni

Per terminare questo argomento restano da citare gli operatori per la conversione di tipo e analizzare la differenza tra operatori come funzioni globali o come funzioni membro.

Per quanto riguarda la conversione di tipo, si rimanda all'appendice A.

Solitamente non c'è differenza tra un operatore definito globalmente e uno analogo definito come funzione membro, nel primo caso per ovvi motivi l'operatore viene dichiarato friend nelle classi cui appartengono i suoi argomenti; nel caso di una funzione membro, il primo argomento è sempre una istanza della classe e l'operatore può accedere a tutti i suoi membri, per quanto riguarda l'eventuale secondo argomento può essere necessaria dichiararlo friend nell'altra classe. Per il resto non ci sono differenze per il compilatore, nessuno dei due metodi è più efficiente dell'altro; tuttavia non sempre è possibile utilizzare una funzione membro, ad esempio se si vuole permettere il flusso su stream della propria classe , è necessario ricorrere ad una funzione globale, perché il primo argomento non è una istanza della classe:

```
class Complex {
  public:
    /* ... */
  private:
    float Re, Im;
  friend ostream & operator<<(ostream & os, Complex & C);
};

ostream & operator<<(ostream & os, Complex & C) {
  os << C.Re << " + i" << C.Im;
  return os;
}
```

Adesso è possibile scrivere

```
Complex C(1.0, 2.3);

cout << C;
```

Appendice

Conversioni di tipo

Per conversione di tipo si intende una operazione volta a trasformare un valore di un certo tipo in un altro valore di altro tipo. Questa operazione è molto comune in tutti i linguaggi, anche se spesso il programmatore non se ne rende conto; si pensi ad esempio ad una operazione aritmetica (somma, divisione...) applicata ad un operando di tipo int e uno di tipo float. Le operazioni aritmetiche sono definite su operandi dello stesso tipo e pertanto non è possibile eseguire immediatamente l'operazione; si rende pertanto necessario trasformare gli operandi in modo che assumano un tipo comune su cui è possibile eseguire l'operazione. Quello che generalmente fa, nel nostro caso, un compilatore di un qualsiasi linguaggio e convertire il valore intero in un reale e poi eseguire la somma tra reali, restituendo un reale.

Non sempre comunque le conversioni di tipo sono decise dal compilatore, in alcuni linguaggi (C, C++, Turbo Pascal) le conversioni di tipo possono essere richieste dal programmatore. Si distingue quindi tra conversioni implicite e conversioni esplicite: le prime (dette anche coercizioni) sono eseguite dal compilatore in modo del tutto trasparente al programmatore, mentre le seconde sono quelle richieste esplicitamente:

```
int i = 5;
float f = 0.0;
double d = 1.0;

d = f + i;

d = (double)f + (double)i;
// questa riga è da leggere come
// d = ( ( (double)f ) + ( (double)i ) );
```

L'esempio precedente mostra entrambi i casi.

Nel primo assegnamento, l'operazione di somma è applicata ad un operando intero e uno di tipo float, per poter eseguire la somma il compilatore C++ prima converte i al tipo float, quindi esegue la somma (entrambi gli operandi hanno lo stesso tipo) e poi, poiché la variabile d è di tipo double, converte il risultato al tipo double e lo assegna alla variabile.

Nel secondo assegnamento, il programmatore richiede esplicitamente la conversione di entrambi gli operandi al tipo double prima di effettuare la somma e l'assegnamento (la conversione ha priorità maggiore delle operazioni aritmetiche).

Una conversione di tipo esplicita viene richiesta con la sintassi

```
( <NuovoTipo> ) <Valore>

// oppure

<NuovoTipo> ( <Valore> )

// ma questo metodo può essere utilizzato solo con
// nomi semplici (ad esempio non funziona con char*)
```

NuovoTipo può essere una qualsiasi espressione di tipo, anche una che coinvolga tipi definiti dall'utente; ad esempio:

```
int a = 5;
float f =2.2;

(float) a
// oppure...
float (a)

// se Persona è un tipo definito
// dal programmatore...
```

```
(Persona) f
// oppure...
Persona (f)
```

Le conversioni tra tipi primitivi sono già predefinite nel linguaggio e possono essere utilizzate in qualsiasi momento, il compilatore comunque le utilizza solo se il tipo di destinazione è compatibile con quello di origine (cioè può rappresentare il valore originale). Le conversioni da e a un tipo definito dall'utente richiedono che il compilatore sia informato riguardo a come eseguire l'operazione.
Per convertire un tipo primitivo (float, int, unsigned int...) in uno definito dall'utente è necessario che il tipo utente sia una classe (o una struttura) e che sia definito un costruttore (pubblico) che ha come unico argomento un parametro del tipo primitivo:

```
class Test {
  private:
    float member;
  public:
    Test(int a);
};

Test::Test(int a) {
  member = (float) a;
}
```

Il metodo va naturalmente bene anche quando il tipo di partenza è un tipo definito dall'utente.
Per convertire invece un tipo utente ad un tipo primitivo è necessario definire un operatore di conversione; con riferimento al precedente esempio, il metodo da seguire è il seguente:

```
Test::operator int() { return (int) member; }
```

Se cioè si desidera poter convertire un tipo utente X in un tipo primitivo T bisogna definire un operatore con nome T:

```
X::operator T() { /* codice operatore */ }
```

Si noti che non è necessario indicare il tipo del valore restituito, è indicato dal nome dell'operatore stesso; si faccia inoltre attenzione quando si definisce un operatore di conversione, poiché questo non è disponibile solo al programmatore, ma anche al compilatore che potrebbe utilizzarlo senza dare alcun avviso al programmatore.

Questa tecnica funziona anche per conversioni verso tipi definiti dal programmatore e in effetti per il compilatore le due tecniche sono praticamente equivalenti e non c'è motivo per preferire una tecnica all'altra; tuttavia, poiché i tipi predefiniti non sono classi, non è possibile utilizzare la tecnica del costruttore per trasformare un tipo utente in uno predefinito (ad esempio int o char *).

Un fattore da tenere presente, quando si parla di conversioni, è che non sempre una conversione di tipo preserva il valore:

ad esempio nella conversione da float a int in generale si riscontra una perdita di precisione, perché la rappresentazione adottata per gli interi è incompatibile con quella adottata per i reali. Da questo punto di vista si può distinguere tra conversione di tipo con perdita di informazione e conversione di tipo senza perdita di informazione. In particolare in quest'ultimo caso si parla di conversioni triviali e promozione di tipo.

Le conversioni triviali sono:

DA:	A:

```
T                       T&
T&                      T
T[ ]                    T*
T(args)                 T (*) (args)
T                       const T
T                       volatile T
T*                      const T*
T*                      volatile T*
```

Per promozione di tipo si intende invece una conversione che trasforma un valore di un tipo in un valore di un tipo più "grande", ad esempio un int in un long int oppure in un float, un tipo cioè il cui insieme di valori rappresentabili includa l'insieme dei valori del tipo di partenza.

Principi della programmazione orientata agli oggetti

Un problema che ha sempre assillato il mondo dell'informatica è quello di poter facilmente eseguire la manutenzione dei programmi, compito la cui difficoltà cresce al crescere della dimensione del codice.

Agli albori dell'informatica i programmi erano piccoli, non tanto perché i calcolatori di 30-40 anni fa erano assai meno potenti di un odierno PC, ma perché il linguaggio di programmazione di allora era l'assembler. Chiunque si sia cimentato a scrivere codice assembler avrà notato che il principale difetto di questo linguaggio è quello di produrre un codice sorgente non facilmente comprensibile e di grosse dimensioni anche per fare la cosa più semplice.

Da allora sono stati sviluppati nuovi linguaggi e paradigmi di programmazione che consentono un maggiore livello di astrazione e una più facile comprensione del codice prodotto: un esempio è dato dai linguaggi procedurali quali il Pascal; questi linguaggi implementano la metafora della scatola nera: un blocco di codice la cui complessità è racchiusa al proprio interno e che non produce alcun cambiamento nel mondo esterno ad esso (effetti collaterali). Essi tuttavia, pur avendo introdotto concetti ancora validi, hanno mancato l'obbiettivo principale, perché i loro progettisti fecero inconsciamente l'assunzione che una scatola nera una volta chiusa non va riaperta, cioè quando che un blocco di codice è stato realizzato ed è funzionante, non va più modificato.

Sia a causa di questa assunzione, sia perché i programmatori non rispettavano alla lettera il paradigma della programmazione procedurale, si ebbe tra la fine degli anni '60 e l'inizio degli anni '70 la "crisi del software": nessuno voleva più occuparsi di manutenzione.

Per migliorare il supporto che i linguaggi procedurali fornivano alla fase di manutenzione venne poi introdotto il concetto di sviluppo strutturato, che richiede una attenta pianificazione di ciò che si voleva realizzare e la produzione di una buona documentazione che descriva il funzionamento del sistema. Ma come ben si sa, quello che spesso accade è che si pensa a scrivere il codice, rimandando ad un successivo momento o ad altri la stesura della documentazione, con il risultato che nella migliore delle ipotesi la documentazione prodotta è inadeguata allo scopo per cui essa è stata realizzata. In sostanza, pur avendo prodotto molte utili tecniche, anche lo sviluppo strutturato si è rivelato poco efficace dal punto di vista della manutenzione del software; occorreva fare un ulteriore passo in avanti: la programmazione orientata agli oggetti.

Questo paradigma di programmazione ha introdotto molti nuovi concetti che hanno letteralmente rivoluzionato tutti (o quasi) i campi dell'informatica (e questi hanno influenzato la programmazione orientata agli oggetti). Prima di proseguire e di introdurre i concetti base della OOP (Object Oriented Programming) è bene fare una precisazione: pur essendo stata teorizzata per la prima volta parecchi anni fa, la OOP è lontana dal possedere uno standard di riferimento, tanto è vero che su molti concetti non esiste alcun accordo tra i teorici di questa disciplina, per cui non stupitevi se qualcun altro definirà un concetto in maniera più o meno diversa da quanto verrà fatto tra poco.

La caratteristica più rivoluzionaria dei linguaggi ad oggetti è quella di concepire i dati e le operazioni su di essi come una unica cosa (oggetto), e non entità separate (dopo tutto dati e operazioni su di essi sono in qualche modo correlati).

Una definizione formale direbbe che un oggetto è una entità software dotata di stato, comportamento e identità. Lo stato viene modellato con costanti e/o variabili dette attributi dell'oggetto; il comportamento è dato da procedure locali dette metodi (alcuni non fanno alcuna distinzione tra attributi e metodi, utilizzando unicamente il primo termine); infine l'identità è qualcosa di unico, immutabile e indipendente dal valore dello stato, che rende un oggetto diverso da ogni altro.

Un oggetto comunica con il mondo esterno tramite messaggi; un messaggio altro non è che una operazione che può essere compiuta su quell'oggetto e l'insieme dei messaggi a cui un oggetto risponde ne costituisce l'interfaccia. Alcune precisazioni: le componenti di uno stato possono essere valori elementari o complessi quali record, oggetti, liste di tipi elementari o oggetti e così via; in generale la OOP prevede che gli attributi di un oggetto non siano mai accessibili dall'esterno, quando ciò accade si dice che l'oggetto incapsula lo stato. Altra caratteristica di un oggetto è quella di possedere una identità che permane nel tempo e che distingue sempre due oggetti: come due gemelli pur essendo fisicamente identici non sono la stessa persona, così due oggetti pur avendo uguali il valore dello stato, il comportamento e l'interfaccia non sono mai uguali; se due oggetti sono uguali allora sono lo stesso oggetto.

Il concetto di messaggio consente l'astrazione dai dati: il programma che usa un oggetto non ha alcuna necessità di conoscerne la struttura interna, per compiere una operazione su di esso tutto quello che deve fare è inviargli un messaggio a cui l'oggetto risponderà restituendo uno o più valori memorizzati nel suo stato o calcolati da esso mediante alcuni dei suoi metodi; ciò consente di cambiare la rappresentazione interna dell'oggetto senza dovere modificare anche i programmi che usano quell'oggetto, che invece continueranno a funzionare regolarmente.

Inviare un messaggio ad un oggetto è possibile grazie ad un costrutto del tipo SEND oggetto arg1 arg2 ... argN; questo in pratica equivale ad un nuovo meccanismo di chiamata di procedura (si chiama una procedura dell'oggetto: quella relativa al messaggio inviato). Ciò benché sia semanticamente diverso dal chiamare una procedura qualsiasi, tende secondo alcuni a rendere le cose più difficili e di fatto molti linguaggi implementano il meccanismo dei messaggi nello stesso modo in cui sono implementati i metodi; l'unica differenza tra un metodo ed un messaggio è che il primo è locale (privato) all'oggetto, mentre il secondo non lo è (pubblico). Nel seguito, al fine di eliminare possibili incomprensioni, faremo distinzione tra attributo, metodo e messaggio.

Per poter creare un oggetto è necessario definire ciò che lo caratterizza, è cioè necessario dare una definizione di tipo oggetto. Un tipo oggetto definisce come minimo l'interfaccia di un insieme di possibili oggetti, ma per molti linguaggi dichiarare un tipo oggetto vuol dire definire la struttura dello stato, l'insieme dei metodi, l'interfaccia e implicitamente (almeno) un costruttore e (almeno) un distruttore.

Costruttori e distruttori sono procedure che il linguaggio usa per creare e rimuovere gli oggetti di cui il programma necessita; in generale costruttori e distruttori vengono realizzati dal linguaggio stesso sulla base della definizione data, ma viene sempre consentito al programmatore di specificare altre operazioni (per ciascun costruttore e distruttore) che questi devono intraprendere quando vengono invocati.

A questo punto può sorgere una domanda: distruggere un oggetto che al proprio interno racchiude un altro oggetto non pone alcun problema (non è necessario distruggere anche quest'ultimo, a meno che il programmatore non specifichi altrimenti), ma cosa succede nel caso contrario?

La risposta a questa domanda è dipende: se quella componente dello stato può assumere il valore indefinito, non ci sono problemi, altrimenti è necessario distruggere anche l'oggetto più grosso, iterando automaticamente il procedimento (in quei linguaggi che trattano gli oggetti indipendentemente dalla loro complessità), oppure lasciando al programmatore il compito di eseguire esplicitamente la distruzione di tutti gli oggetti dal più piccolo al più grosso (tutto dipende dal fatto che il linguaggio tratti gli oggetti come un tutt'uno, indipendentemente dalla loro complessità, o meno). Il motivo di tutto ciò è semplice, dato che un oggetto possiede una identità unica, assegnarlo ad un campo di un altro oggetto non vuol dire copiarlo in quel campo, ma attivare un riferimento all'oggetto assegnato; ciò implica che in seguito alla distruzione dell'oggetto tutti i riferimenti ad esso divengono non validi e un tentativo di accedere all'oggetto distrutto genera un errore a tempo di esecuzione. Molti linguaggi object oriented non offrono alcun modo di verificare ciò, lasciando al programmatore il compito di assicurarsi della corretta creazione e distruzione degli oggetti.

Un altro importante concetto della OOP è quello di ereditarietà. Supponiamo di aver definito un tipo Persona e di aver bisogno anche di un tipo Studente, in fondo uno studente è sempre una persona, anche se con delle caratteristiche in più: oltre agli attributi ed ai metodi di una persona, possiede una matricola, si iscrive... ecc. Tutto questo un linguaggio non orientato agli oggetti non consente di esprimerlo con facilità, ne tanto meno di farlo senza dover riscrivere parte del codice del tipo Persona nel tipo Studente;
in un linguaggio ad oggetti invece ciò si fa dichiarando il tipo Studente discendente del tipo Persona, in tal modo il tipo Studente (detto tipo figlio e in generale sottotipo) eredita tutti gli attributi e i metodi del tipo Persona (tipo padre e in generale supertipo), bisognerà poi dichiarare solo ciò che è proprio del tipo Studente aggiungendo nuove definizioni e/o sovrascrivendone alcune del tipo padre.
Naturalmente il tipo figlio eredita dal padre anche l'interfaccia. L'ereditarietà è quindi un meccanismo innovativo che consente il reimpiego di codice precedentemente scritto; si noti inoltre che tale concetto stabilisce una gerarchia tra i tipi: in cima sta quello definito per primo, poi quelli definiti a partire da questo e via via iterando ai livelli successivi al primo (naturalmente si tratta di una relazione di ordinamento parziale).

Se la ridefinizione di un attributo è consentita solo per specializzazione si dice che l'ereditarietà è stretta, altrimenti si parla di ereditarietà debole; se un linguaggio implementa l'ereditarietà stretta, là dove è richiesto un supertipo può essere utilizzato un sottotipo (in quanto il sottotipo "riempie" completamente il supertipo), si ha cioè anche l'ereditarietà del contesto, ciò

invece non è vero in caso di ereditarietà debole, in questo caso vale solo che il sottotipo risponde anche ai messaggi del supertipo.

La ridefinizione di un metodo può riguardare tanto gli argomenti e il risultato, quanto l'implementazione stessa del metodo; se si ridefinisce l'implementazione del metodo, in quella nuova è possibile eseguire una chiamata allo stesso metodo come implementato prima (in questo modo si può essere certi che i campi ereditati, se non ridefiniti, vengano manipolati correttamente, limitando così la possibilità di errore).

L'ereditarietà stretta pone un problema: supponiamo di avere una procedura F(x:Persona) che chiami il metodo Presentati ridefinito nel sottotipo Studente, e siano Caio di tipo Persona e Tizio di tipo Studente; la chiamata ad F(Caio) è del tutto lecita e non comporta problemi, ma la chiamata ad F(Tizio), lecita perché in presenza di ereditarietà del contesto, quale metodo Presentati chiamerà? Quello relativo al tipo Persona o al tipo Studente? Questo problema viene risolto chiamando sempre il metodo più specializzato tramite la tecnica di collegamento ritardato (late binding) che rinvia a tempo di esecuzione la scelta del metodo da applicare e quindi F(Tizio) chiamerà il metodo Presentati relativo al tipo Studente.
Ovviamente ciò può non essere gradito, ad esempio perché il metodo restituisce qualcosa in più che non vogliamo; per risolvere tale problema tutti i linguaggi ad oggetti consentono di forzare la chiamata del metodo relativo al tipo dell'argomento.
Chiaramente se un sottotipo non ridefinisce un metodo, una chiamata ad esso viene risolta chiamando l'implementazione che per prima si incontra risalendo la gerarchia dei tipi (relativa a quel tipo), generando un errore se tale metodo non e stato definito.

Il concetto di ereditarietà può essere esteso in modo tale da consentire ad un tipo oggetto di ereditare da più supertipi, distinguendo così tra ereditarietà semplice ed ereditarietà multipla. Non tutti sono favorevoli all'ereditarietà multipla, molti sostengono che ciò che essa consente è ottenibile anche con l'ereditarietà semplice; entrambe le fazioni hanno validi motivi a sostegno delle proprie idee: molte situazioni che in passato richiedevano l'ereditarietà multipla oggi sono risolvibili utilizzando opportune tecniche, ma in alcune situazioni l'ereditarietà multipla è ancora la situazione migliore: tipicamente quando si vogliono combinare le caratteristiche di più tipi per ottenere un sottotipo più flessibile: ad esempio un tipo FileDiInput ed un tipo FileDiOutput combinati insieme per ottenere un tipo FileDiInputOutput. A fronte di questi vantaggi l'ereditarietà multipla pone tuttavia un problema di ambiguità quando due o più tipi utilizzano lo stesso identificatore, in questi casi bisogna disporre di uno strumento per risolvere il problema: ad esempio una soluzione (troppo restrittiva) potrebbe semplicemente impedire che la cosa accada; inoltre è molto più difficile gestire bene una gerarchia di tipi in presenza di ereditarietà multipla.

Il meccanismo dell'ereditarietà nasconde un'altra importante possibilità che va oltre la riusabilità del codice: il polimorfismo. La possibilità di avere oggetti diversi con la stessa interfaccia (o parte di essa) consente di realizzare applicazioni capaci di utilizzare correttamente oggetti di diverso tipo senza alcuna distinzione: supponiamo di voler realizzare una applicazione che tracci e cancelli poligoni di varie forme, quello che bisogna fare è dichiarare un tipo Poligono che risponda ai messaggi TRACCIA e CANCELLA e poi derivare da esso i tipi Triangolo, Quadrato ecc. i quali risponderanno ancora ai messaggi del supertipo, ma in maniera diversa. Il nostro programma non dovrà fare altro che gestire il solo tipo Poligono limitandosi a inviare ad esso solo i messaggi a cui risponde, sarà poi l'oggetto in questione che penserà a tracciarsi o cancellarsi.

Programmi così fatti possono facilmente essere estesi senza dover letteralmente toccarne l'implementazione, basta aggiungere nuovi tipi derivandoli dal supertipo che sta in cima alla gerarchia. La possibilità di avere oggetti polimorfi, che quindi si comportano in maniera differente (secondo il loro tipo effettivo) ha non a caso dato ai linguaggi ad oggetti il soprannome di "linguaggi attori", proprio perché come un attore un oggetto può "recitare" ruoli diversi a seconda della situazione.

Una utile estensione al paradigma ad oggetti visto finora (molto utile nel campo delle basi di dati) è il concetto di classe; una classe è un insieme di oggetti dello stesso tipo con associati operatori per estrarre, inserire e cercare elementi di quel tipo. Anche per le classi come per i tipi oggetto si parla di gerarchia: una classe C2 si dice sottoclasse di C1 se il tipo T2 degli elementi di C2 è sottotipo di T1, tipo di C1 (vincolo intensionale); ciò comporta che gli elementi di C2 sono anche elementi di C1 (vincolo estensionale), cioè C2 è un sottoinsieme di C1 (ricordate? un oggetto di un tipo è anche un oggetto di ogni suo supertipo).
Spesso si parla anche di metaclassi, intendendo con questo termine una classe i cui elementi sono altre classi.

Fate attenzione al fatto che in molti linguaggi (compreso il C++) il termine di classe è usato come sinonimo di tipo e quindi non ha niente a che vedere con la definizione di prima; in tali linguaggi può divenire difficile gestire gli oggetti di un tipo, specie se ce ne sono molti; in questi casi purtroppo il programmatore non dispone di alcun aiuto e deve programmarsi una opportuna struttura e gli operatori su di essa.

Un'altra estensione prevede che gli oggetti possano assumere e perdere dinamicamente ruoli, rispettivamente trasformandoli in un oggetto del sottotipo o riportandoli ad un supertipo. Tutto questo nasce dalla constatazione che nella realtà le entità assumono e perdono delle caratteristiche: ad esempio una persona salendo su un'auto diviene passeggero o automobilista, scendendo ritorna ad essere un pedone. Ancora una volta non tutti i linguaggi offrono tale possibilità.

Benché la OOP abbia risolto diversi problemi, non ha centrato tutti gli obbiettivi: la riusabilità del codice è una realtà locale alla singola applicazione, è assai difficile che il codice di una applicazione possa essere facilmente utilizzato in un'altra: la cosa è fattibile solo relativamente a oggetti e tools di applicazione generale, in quanto spesso è necessario apportare grosse modifiche al codice già disponibile, al punto che è più conveniente ricominciare da capo.

Tuttavia è innegabile che la manutenzione del software sia oggi molto più semplice di quanto lo fosse una volta, ma è necessario uno studio approfondito del sistema da realizzare, scrivere del codice il più possibile autoesplicativo e naturalmente PENSARE AD OGGETTI.