Introduzione

Il C++ è un linguaggio di programmazione "**all purpose**", ovvero utilizzabile per la 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. Ciò tuttavia non implica che qualsiasi applicazione debba essere realizzata 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 più conveniente.
Negli ultimi anni il C++ ha ottenuto un notevole successo per diversi motivi:

- Conserva una compatibilità quasi assoluta 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, rimediando almeno in parte alle carenze del suo predecessore. In particolare l'introduzione di meccanismi quali i *Template* e le *Classi* rende il C++ un linguaggio multiparadigma (con particolare predilezione per il paradigma ad oggetti e la programmazione generica);

- 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 mediamente assai 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 non è cambiato molto e non esegue alcune ottimizzazioni che non sono fattibili a compile-time.
  La sempre maggiore diffusione del C++ sta comunque cambiando questa situazione ed è prevedibile nei prossimi anni una sostanziale evoluzione di compilatori e linker, grazie anche alla recente adozione di uno standard.

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. Verranno inizialmente trattati gli aspetti basilari del linguaggio (tipi, dichiarazioni di variabili, funzioni,...), per poi passare all'esame degli aspetti peculiari (classi, template, eccezioni...); alla fine analizzeremo (almeno in parte) l'input/output tramite stream e la libreria standard del linguaggio.
Questo materiale 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. La capacità di programmare in un qualsiasi altro

linguaggio è invece ritenuta dote necessaria alla comprensione di quanto segue. Salvo rare eccezioni non verranno discussi aspetti relativi a tematiche di implementazione dei vari meccanismi e altre note tecniche che esulano dagli obiettivi di queste pagine.

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:

- 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       // Errore! il "+ 5;" è commento  + 5;
                 e non è possibile dividerlo
                 su più righe
  ...
}
```

Benchè 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 */ su
               una sola riga
```

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 (o quasi) fornisce un certo numero di tipi primitivi (cui è associato un identificatore predefinito) e dei meccanismi per permettere la costruzione di nuovi tipi (a cui il programmatore deve poter associare un nome) a partire da questi.
Infine, funzione è il termine che il C++ utilizza per indicare i sottoprogrammi.
In effetti potremmo considerare una sesta categoria di identificatori, gli identificatori di **macro**; una macro è sostanzialmente un alias per un frammento di codice. Le macro comunque, come vedremo in seguito, non sono trattate dal compilatore ma da un precompilatore che si occupa di eseguire alcune elaborazioni sul codice sorgente prima che questo venga effettivamente sottoposto a compilazione.
Parleremo comunque con maggior dettaglio di variabili, costanti, etichette, tipi, funzioni e macro in seguito.

Un identificatore deve iniziare con una lettera o con carattere di underscore (_) che possono essere seguiti da un numero qualsiasi di lettere, cifre o underscore; viene fatta distinzione tra lettere maiuscole e lettere minuscole. Tutti gli identificatori presenti in un programma devono essere diversi tra loro, indipendentemente dalla categoria cui appartengono.
Benchè 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	**auto**	**bool**	**break**
case	**catch**	**char**	**class**
const	**continue**	**const_cast**	**default**
delete	**do**	**double**	**dynamic_cast**
else	**enum**	**explicit**	**extern**
false	**float**	**for**	**friend**
goto	**if**	**inline**	**int**
long	**mutable**	**namespace**	**new**
operator	**private**	**protected**	**public**
register	**reinterpret_cast**	**return**	**short**
signed	**sizeof**	**static**	**static_cast**
struct	**switch**	**template**	**this**
trow	**true**	**try**	**typedef**
typeid	**typename**	**union**	**unsigned**
using	**virtual**	**void**	**volatile**
wchar_t	**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".
Si noti che un segno può essere espresso solo in base 10, negli altri casi esso è sempre **+**:

```
+45          // Costante intera in base 10,
055          // in base 8
0x2D         // ed in base 16
```

La base in cui viene scritta la costante determina il modo in cui essa viene memorizzata. Il compilatore sceglierà il tipo (Vedi tipi di dato) da utilizzare sulla base delle seguenti regole:

- 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
0x20ul       // forza unsigned long
0x20l        // forza long
0x20u        // 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;

L'uso della lettera **E** indica il ricorso alla notazione scientifica.
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.
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       // 110 * 10000 (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:

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

Espressioni e istruzioni

Inizieremo ad esaminare i costrutti del C++ partendo proprio dalle istruzioni e dalle espressioni, perchè in questo modo sarà più semplice esemplificare alcuni concetti che verranno analizzati nel seguito. Per adesso comunque analizzaremo 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 ad esempio vuol dire 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 di essa.
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, poichè 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.

Poichè 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 derivanti 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 poichè 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++;
```

Altri operatori

Le espressioni, per quanto visto sopra, rappresentano un elemento basilare del C++, tant'è che il linguaggio fornisce un ampio insieme di operatori.
La tabella che segue riassume brevemente quasi tutti gli operatori del linguaggio, per completarla dovremmo aggiungere alcuni particolari operatori di conversione di tipo per i quali si rimanda all'appendice A.

SOMMARIO DEGLI OPERATORI

::	risolutore di scope
. **->** **[ ]** **( )** **( )** **++** **--**	selettore di campi selettore di campi sottoscrizione chiamata di funzione costruttore di valori post incremento post decremento
sizeof **++** **--** **~** **!** **-** **+** **&** * **new** **new[ ]** **delete** **delete[ ]** **( )**	dimensione di pre incremento pre decremento complemento negazione meno unario più unario indirizzo di dereferenzazione allocatore di oggetti allocatore di array deallocatore di oggetti 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. 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, ad esempio nell'espressione

```
Pippo = 10*13 + 7*25;
```

non si sa quale tra ***10*13*** e ***7*25*** verrà valutata per prima (si noti che comunque verranno rispettate le regole di precedenza e associatività).

Gli operatori di assegnamento e quelli di (auto)incremento e (auto)decremento sono già stati descritti, esamineremo ora l'operatore per le espressioni condizionali.

L'operatore **? :** è l'unico operatore ternario:

<Cond> ? <Expr1> : <Expr2>

La semantica di questo operatore non è molto complicata: ***Cond*** può essere una qualunque espressione che produca un valore booleano (Vedi paragrafo successivo), se essa è verificata il risultato di tale operatore è la

valutazione di ***Expr1***, altrimenti il risultato è ***Expr2***.
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.

Vero e falso

Prima che venisse approvato lo standard, il C++ non forniva un tipo primitivo (vedi tipi primitivi) per rappresentare valori booleani. Esattamente come in C i valori di verità venivano rappresentati tramite valori interi: 0 (zero) indicava falso e un valore diverso da 0 indicava vero. Ciò implicava che ovunque fosse richiesta una condizione era possibile mettere una qualsiasi espressione che producesse un valore intero (quindi anche una somma, ad esempio). Non solo, dato che l'applicazione di un operatore booleano o relazionale a due sottoespressioni produceva 0 o 1 (a seconda del valore di verità della formula), era possibile mescolare operatori booleani, relazionali e aritmetici.
Il comitato per lo standard ha tuttavia approvato l'introduzione di un tipo primitivo appositamente per rappresentare valori di verità. Come conseguenza di ciò, là dove prima venivano utilizzati i valori interi per rappresentare vero e falso, ora si dovrebbero utilizzare il tipo **bool** e i valori **true** (vero) e **false** (falso), anche perchè i costrutti del linguaggio sono stati adattati di conseguenza. Comunque sia per compatibilità con il C ed il codice C++ precedentemente prodotto è ancora possibile utilizzare i valori interi, il compilatore converte automaticamente ove necessario un valore intero in uno booleano e viceversa (**true** viene convertito in 1):

```
10 < 5                    // produce false
10 > 5                    // produce true
true || false             // produce true

Pippo = (10 < 5) && true; // possiamo miscelare le due
Clarabella = true && 5;   // modalità, in questo caso
                          // si ottiene un booleano
```

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 booleana.

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 eseguito alcunchè, nel secondo caso invece 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 il seguente esempio (si considerino ***X***, ***Y*** e ***Z*** variabili intere):

```
if ( X==10 ) X--;
else {                  // istruzione composta
  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).

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 (**true**) si esegue ***Istruzione*** e poi si ripete il tutto; l'istruzione termina quando ***Condizione*** valuta a **false**.

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.
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 veriabili dichiarate hanno scope e lifetime limitati a tutto il ciclo); ***Condizione*** è una qualsiasi espressione booleana; e ***Iterazione*** è una istruzione da eseguire dopo ogni iterazione (solitamente un incremento). Tutti e tre gli elementi appena descitti 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 di seguito) in ***Istruzione*** causa un salto a ***Iterazione*** nel caso del ciclo **for**, nel **while** invece causa un salto all'inizio del ciclo.
Ecco come usare il ciclo **for** per calcolare il fattoriale:

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

Si noti la mancanza del terzo argomento del **for**, omesso in quanto inutile.

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 corpo del ciclo (pertanto il ciclo potrebbe ancora essere eseguito):

```
  Fattoriale = 1;
```

```
while (true) {                  // all'infinito...
  if (InteroPositivo > 1) {
    Fattoriale *= InteroPositivo--;
    continue;
  }
  break;   // se 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 a valori interi; ***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** qualora specificato (il ramo **default** è opzionale).
Ecco alcuni esempi:

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

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

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 obiettivamente 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 evitarne.
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 PLUTO;
  Topolino = 5;
  /* ... */
PLUTO : 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

Ad eccezione delle etichette, ogni identificatore che il programmatore intende utilizzare in un programma C++, sia esso per una variabile, una costante simbolica, di tipo o di funzione, 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 restituito;
- 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 operazioni 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 sei tipi fondamentali (o primitivi):

- **bool**
- **char**
- **wchar_t**
- **int**
- **float**
- **double**

Abbiamo gia visto (vedi Vero e falso) il tipo **bool** e sappiamo che esso serve a rappresentare i valori di verità; su di esso sono definite sostanzialmente le usuali operazioni logiche (**&&** per l'AND, **||** per l'OR, **!** per la negazione...) e non ci soffermeremo oltre su di esse, solo si faccia attenzione a distinguerle dalle operazioni logiche su bit (rispettivamente **&**, **|**, **~**...).
Il tipo **char** è utilizzato per rappresentare piccoli interi (e quindi su di esso possiamo eseguire le usuali operazioni aritmetiche) e singoli caratteri; accanto ad esso troviamo anche il tipo **wchar_t** che serve a memorizzare caratteri non rappresentabili con **char** (ad esempio i caratteri unicode).
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** (con segno), **unsigned** (senza segno), **short** (piccolo) e **long** (lungo) per selezionare differenti intervalli di valori; essi tuttavia 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:

1. **bool**
2. **char**
3. **wchar_t**
4. **short int**
5. **int**
6. **long int**
7. **signed char**
8. **signed short int**
9. **signed int**
10. **signed long int**
11. **unsigned char**
12. **unsigned short int**
13. **unsigned int**

14. **unsigned long int**
15. **float**
16. **double**
17. **long double**

Il tipo **int** è per default **signed** e quindi è equivalente a tipo **signed int**, invece i tipi **char**, **signed char** e **unsigned char** sono considerate categorie distinte. 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 (che però può variare da implementazione a implementazione). Il seguente programma permette di conoscere la dimensione di alcuni tipi come multiplo di **char** (di solito rappresentato su 8 bit), modificare il codice per trovare la dimensione degli altri tipi è molto semplice e viene lasciato per esercizio:

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

int main(int, char* []) {
  cout << "bool: " << sizeof(bool) << endl;
  cout << "char: " << sizeof(char) << endl;
  cout << "short int: " << sizeof(short int) << endl;
  cout << "int: " << sizeof(int) << endl;
  cout << "float:" << sizeof(float) << endl;
  cout << "double: " << sizeof(double) << endl;
  return 0;
}
```

Una veloce spiegazione sul listato:
le prime due righe permettono di utilizzare una libreria (standard) per eseguire l'output su video; la libreria **iostream** 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. L'ultima istruzione serve a terminare il programma. Infine **main** è il nome che identifica la funzione principale, ovvero il corpo del programma, parleremo in seguito e più in dettaglio di **main()**.

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;
```

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**. 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;
```

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 sottointeso
const unsigned char d = 'a', f = 1;
const float g = 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

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

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, e 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) oppure per ragioni di efficenza. 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 sia 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 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 e maggiormente approfondita.

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** (anche se solitamente si parla di una variabile di tipo array); il tipo di ***Array*** è **array di 10 int(eri)**.
La sintassi per la generica dichiarazione di un array è

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

Al solito ***Tipo*** può essere sia un tipo primitivo che uno definito dal programmatore, ***Identificatore*** è un nome scelto dal programmatore per identificare l'array, mentre ***NumeroElementi*** deve essere un intero positivo e indica il numero di singole variabili che compongono l'array.
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[NumeroElementi-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
```

La selezione di un elemento da un array multidimensionale avviene specificando un valore per ciascuno degli indici:

```
int Pluto = SuperPippo[5][7][9];
Quo[1][7] = Superpippo[2][2][6];
```

È 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} };

int Quo[4][3][2]    = { {{1, 2}, {3, 4}, {5, 6}},
                        {{7, 8}, {9, 10}, {11, 12}},
                        {{13, 14}, {15, 16}, {17, 18}},
                        {{19, 20}, {21, 22}, {23, 24}}
                      };

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. Nella seconda riga viene dichiarata una matrice bidimensionale e se ne esegue l'inizializzazione, si noti l'uso delle parentesi graffe per raggruppare opportunamente i valori; la terza dichiarazione chiarisce meglio come procedere nel raggruppamento dei valori, si tenga conto che a variare per primo è l'ultimo indice così che gli elementi vengono inizializzati nell'ordine ***Quo[0][0][0], Quo[0][0][1], Quo[0][1][0], ..., Quo[3][2][1]***.
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 dichiararzione 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 per 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 Minnie[ ]   = { 'M', 'i', 'n', 'n', 'i', 'e' };
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 dello stesso tipo contemporaneamente (ad esempio per eseguire una ricerca). Tuttavia in generale 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 poichè 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 entità 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[7];
};

Persona Pippo           = { "Pippo", 40, "PPP718" };
Persona AmiciDiPippo[2] = { {"Pluto", 40, "PLT712"},
                            {"Minnie", 35, "MNN431"}
                          };

// 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 definito nella dichiarazione. 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. 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 perchè, 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 sotto certi aspetti simile alle strutture e quello delle unioni. Sintatticamente l'unica differenza è che nella dichiarazione di una unione viene utilizzata la keyword **union** anzicchè **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) poichè tali conversioni sono dipendenti dall'architettura su cui si opera e pertanto non portabili, ma anche potenzialmete scorrette.

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 sono 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 perchè 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 (anche se ciò probabilmente comporterebbe qualche problema). Si può utilizzare anche un identificatore dell'enumerazione precedentemente definito.
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 tipo **"array di 20 long double"**. Infine il terzo esempio è il più interessante perchè 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).

Funzioni

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 parametro 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 sottointende **int**; se una funzione non ritorna alcun valore va dichiarata **void**, come mostra quest'altro 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.

Quelle 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 sono stati introdotti nel C++ per informare il compilatore dell'esistenza di una certa funzione e consentire un maggior controllo 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, benchè 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 (anzicchè 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 poichè un prototipo è terminato 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 il caso in cui 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 indicarne il prototipo, 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 a) {
  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!
  return 0;
}
```

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 chiamata a ***Sleep*** è un errore poichè ***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 fin'ora) automaticamente sui parametri attuali (quelli effettivamente usati in una chiamata della funzione):

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

void Assign(int a, int b) {
  cout << "Inizio Assign, parametro a = " << a << endl;
  a = b;
  cout << "Fine Assign, parametro a = " << a << endl;
}

int main(int, char* []) {
  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;
  return 0;
}
```

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 localmente 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 poichè quando viene 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 viene 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 poichè 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 a.

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 funzioni >

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

Un programma è dunque costituito da un insieme (eventualmente vuoto) di dichiarazioni e di definizioni globali di costanti, variabili... ed un insieme di dichiarazioni e definizioni di funzioni (che non possono essere dichiarate e/o definite 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). I vecchi compilatori non standard spesso lasciavano ampia libertà circa il prototipo di ***main***, alcuni consentivano di dichiararla **void**, ora a norma di standard ***main*** deve avere tipo **int** e se nel corpo della funzione non viene inserito esplicitamente una istruzione **return**, il compilatore inserisce automaticamente una **return 0;**.
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 >
using namespace std;

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

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 sia la realizzazione che la successiva manutenzione. Tuttavia spesso si è indotti a rinunciare a tale beneficio perchè 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 perchè 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 ogni controllo statico di tipo poichè gli errori divengono evidenti solo quando la macro viene espansa; in C tutto sommato ciò non costituisce un grave problema perchè il C non è fortemente tipizzato.
Al contrario il C++ possiede un rigido sistema di tipi e l'uso di macro costituisce un grave ostacolo allo sfruttamento di tale caratteristica. Esistono poi altri svantaggi nell'uso delle macro: rendono difficile il debugging e non sono flessibili come le funzioni (è ad esempio difficile rendere fattibili macro ricorsive).

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 lo sostituisce alla chiamata della funzione; 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 comunque la cosa generalmente non è grave, poichè un ciclo tipicamente richiede una quantità di tempo ben maggiore di quello necessario a chiamare la funzione, per cui l'espansione inline non avrebbe portato grossi benefici. Quando l'espansione inline della funzione non è

possibile solitamente si viene avvisati da una warning.
Si osservi che, per come sono trattate le funzioni inline, non ha senso utilizzare la keyword **inline** in un prototipo di funzione perchè 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; perchè 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 tener 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, poichè è il linker che si occupa di risolvere i riferimenti (il linker del C++ lavora in due passate); nel caso delle funzioni inline, poichè 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; 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 (perchè 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 vuol dire 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);
float Sum(float a, float b);
float Sum(float a, int b);
float Sum(int a, float b);
```

Comunque si tratta di una pratica obsoleta che infatti non è prevista nello standard.
Le funzioni sovraccaricate si utilizzano esattamente come le normali funzioni:

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

/* Dichiarazione ed implementazione delle varie Sum */

int main(int, char* []) {
  int a = 5;
  int y = 10;
  float f = 9.5;
  float r = 0.5;

  cout << "Sum(int, int):" << endl;
  cout << "    " << Sum(a, y) << endl;

  cout << "Sum(float, float):" << endl;
  cout << "    " << Sum(f, r) << endl;

  cout << "Sum(int, float):" << endl;
  cout << "    " << Sum(a, f) << endl;

  cout << "Sum(float, int):" << endl;
  cout << "    " << Sum(r, a) << endl;

  return 0;
}
```

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 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*** viste sopra):

```
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 perchè, 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 perchè il primo parametro anzicchè 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:

1. **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&**);

2. **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**);

3. **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**);

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

5. **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 all'appendice A; 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 esso sia richiesto:

```
volatile int ComPort;
```

La precedente definizione dice al compilatore che il valore di ***ComPort*** è fuori dal controllo del programma (ad esempio perchè 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, soprattuto 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 alcune 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 ottere 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**, come mostra il codice seguente), 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;
```

A contribuire all'errore avrà sicuramente influito il fatto che l'asterisco stava attacato al nome del tipo, tuttavia cambiando stile il problema non si risolve più di tanto. La soluzione migliore solitamente consigliata è quella di porre dichiarazioni diverse in righe diverse.

Ritorniamo all'esempio da cui siamo partiti.
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 esso 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 *****) e valgono quindi le stesse raccomandazioni fatte sopra.

In alcuni casi è necessario avere puntatori generici, in questi casi il puntatore va dichiarato **void**:

```
void* PuntatoreGenerico;
```

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 tener 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" ora
                        // puntano 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** al fine di ottenere accesso all'oggetto puntato; l'operazione viene eseguita tramite l'**operatore di dereferenzazione *** 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 dereferenzazione può essere eseguita su un qualsiasi puntatore a

condizione che questo non sia stato dichiarato **void**. In generale infatti non è possibile stabilite il tipo dell'oggetto puntato da un puntatore void e il compilatore non sarebbe in grado di trattare tale oggetto.
Quando si dereferenzia un puntatore bisogna prestare attenzione che esso sia stato inizializzato correttamente; la dereferenzazione di un puntatore inizializzato a 0 è sempre un errore, la dereferenzazione di un puntatore non inizializzato causa errori non definiti (e potenzialmente difficili da scovare). 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 dereferenzazione 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;       // equivale 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 a 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 dereferenzazione di un char* o l'indicizzazione
// di un array causano la visualizzazione di un solo
// carattere perchè 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, in particolare 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 quattro scopi:

1. Realizzazione di strutture dati dinamiche (es. liste linkate);
2. Realizzazione di funzioni con effetti laterali sui parametri attuali;
3. Ottimizzare il passaggio di parametri di grosse dimensioni;
4. Rendere possibile il passaggio di parametri di tipo funzione.

Il primo caso è tipico di applicazioni per le quali non è noto a priori la quantità di dati che si andranno a manipolare. Senza i puntatori non sarebbe possibile manipolare contemporaneamente un numero non predefinito di dati, anche utilizzando un array porremmo un limite massimo al numero di oggetti di un certo tipo immediatamente disponibili.
Utilizzando i puntatori invece è possibile realizzare ad esempio una lista il cui numero massimo di elementi non è definito a priori:

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

// 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 corrente di celle
// della lista e il puntatore alla prima cella

struct TList {
  unsigned Size;  // Dimensione lista
  TCell* First;   // Puntatore al primo elemento
};

int main(int, char* []) {
  TList List;        // Dichiara una lista
  List.Size = 0;     // inizialmente vuota
  int FloatToRead;
  cout << "Quanti valori vuoi immettere? " ;
  cin >> FloatToRead;
  cout << endl;

  // questo ciclo richiede valori reali
  // e li memorizza nella lista

  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
                               // variabile da creare
    cout << "Immettere un valore reale " ;

    // cin legge l'input da tastiera e l'operatore di
    // estrazione >> lo memorizza nella variabile.
    cin >> List.First -> AFloat;
    cout << endl;
    List.First -> Next = Temp; // aggiunge la cella in
                               // testa alla lista
    ++List.Size;       // incrementa 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(int j=0; j < List.Size; ++j) {
      Total += List.First -> AFloat;

      // estrae la cella in testa alla lista...
      TCell* Temp = List.First;
      List.First = List.First -> Next;

      // e quindi la distrugge
      cout << "Distruzione della cella estratta..."
           << endl;
      delete Temp;
    }
    cout << "Totale = " << Total << endl;
    return 0;
  }
```

Il programma sopra riportato 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 e li somma restituendo il totale; via via che un valore viene estratto dalla lista, la cella corrispondente viene distrutta. 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 (una porzione di memoria riservata all'avvio di un programma per operazioni di questo tipo), 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;
```

La dimensione massima di strutture dinamiche è unicamente determinata dalla dimensione della heap memory che a sua volta è generalmente limitata dalla quantità di memoria del sistema.
Un altro importante aspetto degli oggetti allocati dinamicamente è che essi non ubbidiscono alle normali regole di scoping statico, solo i puntatori in quanto tali sono soggetti a tali regole, un oggetto allocato dinamicamente può quindi essere creato in un certo scope ed essere acceduto in un altro semplicemente trasmettendone l'indirizzo (il valore del puntatore).
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 attuali:

```
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 dereferenzazione di tale copia consente comunque la modifica dell'oggetto puntato:

```
int A = 10;

cout << " A = " << A << endl;
cout << " Chiamata a Change(int*)... " << endl;
Change(&A);
cout << " Ora A = " << A << endl;
```

l'output che il precedente codice produce è:

```
A = 10
Chiamata a Change(int*)...
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 un funzione riceve parametri di dimensioni notevoli e l'operazione di copia del parametro attuale in un'area privata della funzione ha effetti deleterei sui tempi di esecuzione della funzione stessa; in questi casi è molto più conveniente passare un puntatore che generalmente occupa pochi byte:

```
void Func(BigParam parametro);

// funziona, ma è meglio quest'altra dichiarazione

void Func(const BigParam* parametro);
```

Il secondo prototipo è più efficiente perchè 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 la funzione ha 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).
Infine un uso importante dei puntatori è per passare come parametro un'altra funzione. Si tratta di un meccanismo che sta alla base dei linguaggi funzionali e che permette di realizzare algoritmi generici (anche se in C++ molte di queste cose sono spesso più semplici da ottenere con i template, in alcuni casi però il vecchio approccio risulta migliore):

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

// Definiamo un tipo funzione:
typedef bool Eval(int, int);

bool Max(int a, int b) {
  return (a>=b)? true: false;
}

bool Min(int a, int b) {
  return (a<=b)? true: false;
}

// Notare il tipo del primo parametro
void Check(Eval* Func, char* FuncName,
           int Param1, int Param2) {
  cout << "È vero che " << Param1 << " = " << FuncName
       << '(' << Param1 << ',' << Param2 << ") ? ";

  // Utilizzo del puntatore per eseguire la chiamata
  // alla funzione puntata (nella condizione dell'if)
  if (Func(Param1, Param2)) cout << "Si" << endl;
  else cout << "No" << endl;
}

int main(int, char* []) {
  for(int i=0; i&lt10; ++i) {
    cout << "Immetti un intero: ";
    int A;
    cin >> A;
    cout << endl << "Immetti un altro intero: ";
    int B;
    cin >> B;
    cout << endl;

    // Si osservi il modo in cui viene
    // ricavato l'indirizzo di una funzione
    // (primo parametro della Check)
    Check(Max, "Max", A, B);
    Check(Min, "Min", A, B);
    cout << endl << endl;
  }
  return 0;
}
```

La **typedef** dice che ***Eval*** è un tipo **"funzione che prende due interi e restituisce un bool"**, quindi conformemente al tipo ***Eval*** definiamo due funzioni ***Max*** e ***Min*** dall'evidente significato. Si definisce quindi una funzione ***Check*** che riceve

quattro parametri: un puntatore a ***Eval***, una stringa e due interi. La funzione ***Check*** usa ***Func*** per eseguire la chiamata alla funzione puntata e ricavarne il valore restituito. Si noti che la chiamata alla funzione puntata viene eseguita come se ***Func*** fosse esso stesso la funzione (ovvero utilizzando l'operatore **()** e passando normalmente i parametri).
Si noti infine che la funzione **main** ricava l'indirizzo di ***Max*** e ***Min*** senza ricorrere all'operatore **&**, analogamente a quanto si fa con gli array.

Reference

I reference (riferimenti) sono sotto certi aspetti un costrutto a metà tra puntatori e le usuali variabili: come i puntatori essi sono contenitori di indirizzi, ma non è necessario dereferenziarli per accedere all'oggetto puntato (si usano come se fossero normali 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 *, un reference viene indicato dal simbolo &:

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

int* IntPtr = &Var;
int& IntRef = Var;        // nei reference non serve
float& FloatRef = f;      // utilizzare & a destra di =
```

Le ultime due righe dichiarano rispettivamente un riferimento a **int** e uno a **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 perchè, 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 una 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.

DoubleRef = 7.3;
// Di nuovo, è compito del programmatore
// distruggere l'oggetto crato 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 perchè è possibile uniformare tali oggetti alle comuni variabili, tuttavia è una pratica che bisognerebbe evitare perchè può generare confusione e di conseguenza errori assai insidiosi.

Uso dei reference

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);

// oppure
Esempio(*Ptr);
```

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 e che i riferimenti non possono associati ad un'altra variabile dopo l'inizializzazione. 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 nè il codice che chiama la funzione nè 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 piùconveniente perchè 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.
Non si tratta comunque di una regola generale, come per tante altre cose, i progettisti del linguaggio hanno pensato di non limitare l'uso dei costrutti con rigide regole e schemi predefiniti, ma di lasciare al buon senso del programmatore il compito di decidere quale fosse di volta in volta la soluzione migliore.

Linkage e file header

Quello che èstato visto fin'ora 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 doveroso 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) 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 (trascuriamone 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; perchè la cosa sia possibile bisogna informare il compilatore che tale nome è dichiarato da qualche altra parte e che il riferimento a tale nome non può essere risolto se non quando tutti i file sono stati compilati, solo il linker quindi può risolvere il problema collegando insieme i due file. Il compilatore deve dunque essere informato dell'esistenza della variabile al fine di non generare un messaggio di errore; tale operazione 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 perchè ***a*** sarebbe stata definita due volte (una per file), il perchè di tale errore sarà chiaro più avanti.
Naturalmente **extern** si può usare anche con le funzioni (anche se come vedremo è ridondante):

```
// 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 tempo
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 (in quanto il compilatore C++ codifica internamente i nomi degli identificatori in modo assai diverso).
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 int 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);
  }
```

Tuttavia nel caso delle funzioni non è necessario l'uso di **extern**, il solo prototipo è sufficiente, è invece necessario ad esempio per le variabili:

```
  int Func2(int);          // extern non necessaria
  extern int a;            // extern necessaria


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

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

  int a = 10;              // definisce la variabile
                           // precedentemente dichiarata
```

I nomi che sono visibili all'esterno di un file sono detti avere **linkage esterno**; tutte le variabili globali hanno linkage esterno, così come le funzioni globali non **inline**; 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 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 per ragioni pratiche (la compilazione è legata al singolo file sorgente). 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.
Si noti inoltre che questa volta è stato specificato l'estensione del file (.h), questo non dipende dall'uso degli apici, ma dal fatto che ad essere incluso è l'header di un file di libreria (ad esempio quando si usa la libreria iostream), infatti in teoria tali header potrebbero non essere memorizzati in un normale file.

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

```
// Esempio2.h

// Un header può includere un altro header
#include "Header1.h"

// o dichiarazioni extern comuni ai moduli
extern "C" {                // Inclusione di un
  #include "HeaderC.h"      // file header C
}
extern "C" {
  int CFunc1(int, float);
  void CFunc2(char*);
}
extern int a;
extern double* Ptr;
extern void Func();
```

Librerie di funzioni

I file header sono molto utili 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);
```

ed ecco come sarebbe il file che la implementa:

```
// Date.cpp
#include "Date.h"
```

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

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

A questo punto la libreria è pronta, per distribuirla basta compilare il file ***Date.cpp*** e fornire il file oggetto ottenuto ed il 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!
Esistono tuttavia due problemi, il primo è illustrato nel seguente esempio:

```
// Modulo1.h
#include < iostream >
using namespace std;

/* altre dichiarazioni */


// Modulo2.h
#include < iostream >
using namespace std;

/* altre dichiarazioni */

// Main.cpp
#include < iostream >
using namespace std;

#include < Modulo1.h >
#include < Modulo2.h >

int main(int, char* []) {
  /* 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) nei rispettivi file header.
Per come funziona il preprocessore, poichè 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 positiva se ***SIMBOLO*** non è definito, negativa 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** e verificata 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):

```
// 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 è verificata) 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 >*** e il file header specificato viene elaborato per produrre codice;
3. A seguito delle direttive contenute inizialmente in ***iostream***, 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 >*** contenuta in ***Modulo1.h*** porta il precompilatore ad elaborare di nuovo ***iostream***, 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 previsto 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.

I namespace

Il secondo problema che si verifica con la ripartizione di un progetto in più file è legato alla necessità di utilizzare identificatori globali unici. Quello che spesso accade è che al progetto lavorino più persone ognuna delle quali si

occupa di parti diverse che devono poi essere assemblate. Per quanto possa sembrare difficile, spesso accade che persone che lavorano a file diversi utilizzino gli stessi identificatori per indicare funzioni, variabili, costanti...
Pensate a due persone che devono realizzare due moduli ciascuno dei quali prima di essere utilizzato vada inizializzato, sicuramente entrambi inseriranno nei rispettivi moduli una funzione per l'inizializzazione e molto probabilmente la chiameranno ***InitModule()*** (o qualcosa di simile). Nel momento in cui i due moduli saranno linkati insieme (e sempre che non siano sorti problemi prima ancora), inevitabilmente il linker segnalerà errore.
Naturalmente basterebbe che una delle due funzioni avesse un nome diverso, ma modificare tale nome richiederebbe la modifica anche dei sorgenti in cui il modulo è utilizzato. Molto meglio prevenire tale situazione suddividendo lo spazio globale dei nomi in parti più piccole (i **namespace**) e rendere unicamente distinguibili tali parti, a questo punto poco importa se in due **namespace** distinti un identificatore appare due volte... Ma vediamo un esempio:

```
// File MikeLib.h
namespace MikeLib {
  typedef float PiType;
  PiType Pi = 3.14;
  void Init();
}

// File SamLib.h
namespace SamLib {
  typedef double PiType;
  PiType Pi = 3.141592;
  int Sum(int, int);
  void Init();
  void Close();
}
```

In una situazione di questo tipo non ci sarebbe più conflitto tra le definizioni dei due file, perchè per accedere ad esse è necessario specificare anche l'identificatore del **namespace**:

```
#include "MikeLib.h"
#include "SamLib.h"

int main(int, char* []) {
  MikeLib::Init();
  SamLib::Init();
  MikeLib::PiType AReal = MikeLib::Pi * 3.7;

  Areal *= Pi;      // Errore!

  SamLib::Close();
}
```

L'operatore **::** è detto risolutore di scope e indica al compilatore dove cercare l'identificatore seguente. In particolare l'istruzione ***MikeLib::Init();*** dice al compilatore che la ***Init()*** cui vogliamo riferirci è quella del **namespace** ***MikeLib***.
Ovviamente perchè non ci siano conflitti è necessario che i due **namespace** abbiano nomi diversi, ma è più facile stabilire pochi nomi diversi tra loro, che molti.
Si noti che il tentativo di riferire ad un nome senza specificarne il **namespace** viene interpretato come un riferimento ad un nome globale esterno ad ogni **namespace** e nell'esempio precedente genera un errore perchè nello spazio globale non c'è alcun ***Pi***.

I namespace sono dunque dei contenitori di nomi su cui sono defite delle regole ben precise:

- Un **namespace** può essere creato solo nello scope globale;
- Se nello scope globale di un file esistono due **namespace** con lo stesso nome (ad esempio i due **namespace** sono definiti in file header diversi, ma inclusi da uno stesso file), essi vengono fusi in uno solo;
- È possibile creare un alias di un **namespace** con la sintassi: **namespace < *ID1* > = < *ID2* >;**
- È possibile avere **namespace** anonimi, in questo caso gli identificatori contenuti nel **namespace** sono visibili al file che contiene il **namespace** anonimo, ma essi hanno tutti automaticamente linkage interno. I **namespace** anonimi di file diversi non sono mai fusi insieme.

La direttiva using

Qualificare totalmente gli identificatori appartenenti ad un **namespace** può essere molto noioso, soprattutto se siamo sicuri che non ci sono conflitti con altri **namespace**. In questi casi ci viene in aiuto la direttiva **using**, che abbiamo già visto in numerosi esempi:

```
#include "MikeLib.h"
using namespace MikeLib;
using namespace SamLib;

/* ... */
```

La direttiva **using** utilizzata in congiunzione con la keyword <namespace importa in un colpo solo tutti gli identificatori del **namespace** specificato nello scope in cui appare la direttiva (che può anche trovarsi nel corpo di una funzione):

```
#include "MikeLib.h"
#include "SamLib.h"

using namespace MikeLib;
// Da questo momento in poi non è necessario
// qualificare i nomi del namespace MikeLib

void MyFunc() {
  using namespace SamLib;
  // Adesso in non bisogna qualificare
  // neanche i nomi di SamLib
  /* ... */
}
// Ora i nomi di SamLib devono
// essere nuovamente qualificati con ::

/* ... */
```

Naturalmente se dopo la **using** ci fosse una nuova definizione di identificatore del **namespace** importato, quest'ultima nasconderebbe quella del **namespace**.

L'identificatore del **namespace** sarebbe comunque ancora raggiungibile qualificandolo totalmente:

```
#include "SamLib.h"
using namespace SamLib;

int Pi = 5;              // Nasconde la definizione
                         // presente in SamLib

int a = Pi;              // Riferisce al precedente Pi

double b = SamLib::Pi;   // Pi di samLib
```

Se più direttive **using namespace** fanno sì che uno stesso nome venga importato da **namespace** diversi, si viene a creare una potenziale situazione di ambiguità che diviene visibile (genera cioè un errore) solo nel momento in cui ci si riferisce a quel nome. In questi casi per risolvere l'ambiguità basta ricorrere ricorrere al risolutore di scope (::) qualificando totalmente il nome.

È anche possibile usare la **using** per importare singoli nomi:

```
#include "SamLib.h"
#include "MikeLib"
using namespace MikeLib;
using SamLib::Sum;

void F() {
  PiType a = Pi;         // Riferisce a MikeLib
  int r = Sum(5, 4);     // SamLib::Sum(int, int)
}
```

Programmazione a oggetti

I costrutti analizzati fin'ora costituiscono già un linguaggio che ci consente di realizzare anche programmi complessi e di fatto, salvo alcuni dettagli, 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, RTTI (Run Time Type Information), programmazione generica, gestione delle eccezioni sono solo alcune delle caratteristiche che rendono il C++ diverso dal C e migliore di quest'ultimo sotto molti aspetti. Si potrebbe apparentemente dire che si tratta solo di qualche aggiunta, in realtà nessun'altra affermazione potrebbe essere più errata: le eccezioni semplificano la gestione di situazioni anomale a run time (un compito già di per sè complesso), mentre il supporto alla programmazione ad oggetti e alla programmazione generica (e ciò che ruota attorno ad esse) rivoluzionano addirittura il 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, cercando contemporaneamente di esporre i principi alla base di tale metodologia di codifica. È bene sottolineare subito che non esiste un unico modello di programmazione orientata agli oggetti, ma esistono differenti formulazioni spesso differenti in pochi dettagli che hanno però una influenza notevole, quanto segue riferirà unicamente al modello offerto dal C++.

L'idea di base

La programmazione orientata agli oggetti (OOP) impone una nuova visione di concetti quali "Tipo di dato" e "Operazioni sui dati".
In contrapposizione al paradigma procedurale dove si distingue tra entità passive (Dati) e entità attive (operazioni sui dati), la OOP vede queste due categorie come due aspetti di una unica realtà. In ottica procedurale volendo realizzare una libreria per la matematica sui complessi, sarremmo portati a scrivere

```
#include
#include
using namespace std;

struct Complex {
  double Re;
  double Im;
};

void Print(Complex& Val) {
  cout << Val.Re << " + i" << Val.Im;
  cout << endl;
}

double Abs(Complex& Val) {
  return sqrt(Val.Re*Val.Re + Val.Im*Val.Im);
}

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

Tutto ciò è corretto, ma perchè separare la rappresentazione di un ***Complex*** dalle operazioni definite su di esso (***Print*** e ***Abs***). In particolare il problema insito nella visione procedurale è che si può continuare ad accedere direttamente alla rappresentazione dei dati eventualmente per scavalcare le operazioni definite su di essa:

```
int main() {
  Complex C;

  // Le seguenti 4 linee di codice non
  // sono una buona pratica;
  C.Re = 0.5;
  C.Im = 2.3;
  cout << Val.Re << " + i" << Val.Im;
  cout << endl;

  cout << Abs(C) << endl;
  return 0;
}
```

Si tratta di un comportamento abbastanza comune in chi non ha molta esperienza nella nanutenzione del codice. Tale comportamento nasconde infatti due pericoli:

- Maggiore possibilità di errore;
- Difficoltà nella modifica del codice;

Nel primo caso ogni qual volta si replica del codice si rischia di introdurre nuovi errori, utilizzando invece direttamente le funzioni previste ogni errore non può che essere localizzato nella funzione stessa.
Nel secondo caso, la modifica della rappresentazione di un ***Complex*** è resa difficile dal fatto che bisogna cercare nel programma tutti i punti in cui si opera direttamente sulla rappresentazione (se si fossero utilizzate solo e direttamente le funzioni previste, tale problema non esisterebbe).
Tutti questi problemi sono risolti dalla OOP fondendo insieme dati e operazioni sui dati secondo delle regole ben precise. Nelle applicazioni object oriented non ci sono più entità attive (procedure) che operano sui dati, ma unicamente entità attive (oggetti) che cooperano tra loro. Se il motto della programmazione procedurale è **"Strutture dati + algoritmi = programmi"**, quello della OOP non può che essere **"Oggetti + cooperazione = programmi"**.

Strutture e campi funzione

Come precedentemente detto, l'idea di partenza è quella di fondere in una unica entità la rappresentazione dei dati e le operazioni definite su questi. La soluzione del C++ (e sostanzialmente di tutti i linguaggi object oriented) è quella di consentire la presenza di campi funzione all'interno delle strutture:

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

  // Ora nelle strutture possiamo avere
  // dei campi di tipo funzione;
  void Print();
  double Abs();
};
```

I campi di tipo funzione sono detti **funzioni membro** oppure **metodi**, i restanti campi della struttura vengono denominati **membri dato** o **attributi**.
La seconda cosa che si può notare è la scomparsa del parametro di tipo ***Complex***. Questo parametro altri non sarebbe che il dato su cui si vuole eseguire l'operazione, e che ora viene specificato in altro modo:

```
Complex A;
Complex* C;

/* ... */

A.Print();
C = new Complex;
C -> Print();
float FloatVar = C -> Abs();
```

Nella OOP non ci sono più procedure eseguite su certi dati, ma **messaggi** inviati ad oggetti. Gli oggetti sono le istanze di una struttura; i messaggi sono le operazioni che possono essere eseguiti su di essi ("***Print***, ***Abs***").
Un messaggio viene inviato ad un certo oggetto utilizzando sempre il meccanismo di chiamata di funzione, il legame tra messaggio e oggetto destinatario viene realizzato con la notazione del punto ("***A.Print()***") o, se si dispone di un puntatore a oggetto, tramite l'operatore **->** ("***C -> Print()***"). Non è possibile invocare un metodo (inviare un messaggio) senza associarvi un oggetto:

```
Complex A;

/* ... */

Print();        // Errore!
```

Un messaggio deve sempre avere un destinatario, ovvero una richiesta di operazione deve sempre specificare chi deve eseguire quel compito.
Il compilatore traduce la notazione vista prima con una normale chiamata di funzione, invocando il metodo selezionato e passandogli un parametro nascosto che altri non è che l'indirizzo dell'oggetto stesso, ma questo lo riprenderemo in seguito.
Quello che ora è importante notare è che siamo ancora in grado di accedere direttamente agli attributi di un oggetto esattamente come si fa con le normali strutture:

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

A.Re = 10;         // Ok!
A.Im = .5;         // ancora Ok!

// anzicchè A.Print()...
cout << A.Re << " + i" << A.Im;
cout << endl;
```

A questo punto ci si potrà chiedere quali sono in vantaggi di un tale modo di procedere, se poi i problemi precedentemente esposti non sono stati risolti; in fondo tutto ciò è solo una nuova notazione sintattica.
Il problema è che le strutture violano un concetto cardine della OOP, l'**incapsulamento**. In sostanza il problema è che non c'è alcun modo di impedire l'accesso agli attributi. Tutto è visibile all'esterno della definizione della struttura, compresi i campi ***Re*** e ***Im***.
Il concetto di incapsulamento dice in sostanza che gli attributi di un oggetto non devono essere accessibili se non ai soli metodi dell'oggetto stesso.

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 (e vicevera) 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 sia alla classe che a ciò che sta fuori 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, l'accesso ad 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 = 0.5;            // 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 effetti è possibile applicare gli specificatori di accesso (**public, private** e come vedremo **protected**) anche alle strutture, ma per le strutture il default e **public** (per compatibilità col C).
Esiste anche una terza classe di visibilità specificata dalla keyword **protected**, ma analizzaremo questo punto solo in seguito parlando di ereditarietà.
La sintassi per la dichiarazione di classe è dunque:

```
class <NomeClasse> {
   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), l'importante è prestare attenzione a evitare di dichiarare **private** (o **protected**) ciò che deve essere visibile anche all'esterno della classe, in particolare le definizioni dei tipi di parametri e valori di ritorno dei metodi **public**.

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:

```
/* 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);
}
```

I due metodi non sono comunque del tutto identici: 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);
}
```

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 (i nomi di tutti i membri della classe sono nello scope di tutti i metodi della stessa classe)!
La domanda corretta da porsi è come si fa a stabilire dall'interno di un metodo qual'è l'effettiva istanza cui ci si riferisce. Il compito di risolvere correttamente ogni riferimento viene svolto **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. Il programmatore non deve comunque preoccuparsi di ciò è il compilatore che risolve tutti i legami tramite tale puntatore. Allo stesso modo a cui si accede agli attributi dell'oggetto, un metodo può anche invocare un altro metodo dell'oggetto stesso:

```
class MyClass {
  public:
    void BigOp();
    void SmallOp();

  private:
    void PrivateOp();
    /* altre dichiarazioni */
};

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

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

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 ai campi del parametro o della variabile locale si deve utilizzare la notazione del punto:

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

void MyClass::Func(MyClass A, /* ... */ ) {
  /* ... */
  BigOp();    // questo messaggio arriva all'oggetto
              // cui è stato inviato Func(MyClass)
  A.BigOp();  // questo invece arriva 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'inizializzazione di un oggetto potrebbe essere eseguita dichiarando un metodo

ad hoc (diciamo ***Set(/* ... */)***) da utilizzare eventualmente anche per l'assegnamento. Tuttavia assegnamento e inizializzazione sono operazioni semanticamente molto diverse e l'uso di una tecnica simile non va bene a nessuno dei due scopi in quanto si tratta di operazioni eseguite oltretutto in contesti diversi e a cui sono delegate responsabilità diverse. Per adesso vedremo come viene inizializzata una istanza di classe, più avanti vedremo un modo elegante di eseguire l'assegnamento utilizzando il meccanismo di overloading degli operatori.
Un primo motivo per cui un metodo tipo

```
class Complex {
  public:
    void Set(float re, float im);
    /* ... */

  private:
    float Re;
    float Im;
};

void Complex::Set(float re, float im) {
  Re = re;
  Im = im;
}
```

non può andare bene è che il programmatore che usa la classe potrebbe dimenticare di chiamare tale metodo prima di cominciare ad utilizzare l'oggetto appena dichiarato. L'inizializzazione è una operazione troppo importante e non ci si può concedere il lusso di dimenticarsene (un tempo la NASA perse un satellite per una simile dimenticanza!).
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...
E poi in questo modo ogni istanza della classe sarebbe sempre inizializzata con valori prefissati, e la situazione sarebbe sostanzialmente quella di prima.
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. Un costruttore può avere un qualsiasi numero di parametri, ma non restituisce mai 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 (il nome del costruttore non è cioè esplicitamente mensionato, esso è implicito nel tipo dell'istanza); un sistema alternativo di eseguire l'inizializzazione sarebbe:

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

ma è poco efficiente perchè quello che si fa è creare un oggetto ***Complex*** temporaneo e poi copiarlo in ***C*** (sarà chiaro in seguito il perchè della cosa), 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 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 esplicitamente un costruttore nella dichiarazione di un oggetto:

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

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;
};

int main(int, char* []) {
  cout << "definizione di B... ";
  Trace B(1, 5);   // Trace(int, int) chiamato!
  cout << "definizione di C... ";
  Trace C;         // costruttore di default chiamato!
  return 0;
}
```

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; modifichamo 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 (non è proprio così, ma avremo modo di vederlo in seguito) 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; benchè sintatticamente le due operazioni siano 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, anche per l'operatore di assegnamento vale il discorso fatto nel caso che tale operatore non venga definito esplicitamente, anche in questo caso il compilatore fornisce automaticamente un operatore di assegnamento. 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 assegnano all'oggetto dichiarato un altro oggetto.
Vediamo un esempio:

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

  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;
  };

  int main(int, chra* []) {
    cout << "definizione di A... " << endl;
    Trace A(1,2);
    cout << "definizione di B... " << endl;
    Trace B(2,4);
    cout << "definizione di C... " << endl;
    Trace C = A;
    cout << "assegnamento a C... " << endl;
```

```
  C = B;
  return 0;
}
```

Eseguendo questo codice si ottiene 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.
Essi sono a tutti gli effetti 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:
    /* ... */
};

int main(int, char* []) {
  MyClass A(1);
  MyClass B = 5.5;
  MyClass D = (MyClass) 7;
  MyClass C = Complex(2.4, 1.0);
  return 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.
Solo l'untima 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 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

Poichè ogni oggetto ha una propria durata (lifetime) è necessario disporre anche di un metodo che permetta una corretta distruzione dell'oggetto stesso, 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). Poichè 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 >
using namespace std;

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 al
                                // distruttore
  }                             // chiamata al
                                // 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 un ulteriore blocco applicativo e di un puntatore per allocare nello heap la variabile:

```
  void Example() {
    TVar* TVarPtr = new TVar(10);
    {
      /* ... */
      if (Cond) {
        delete TVarPtr;
        TVarPtr = 0;
      }
      /* ... */
    }
    if (TVarPtr) delete TVarPtr;
  }
```

L'uso del puntatore permette di capire se la variabile è stata deallocata (nel qual caso il puntatore viene posto a 0) oppure no (puntatore non nullo).
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 poichè 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 nuova istanza riceve nuova memoria per ogni attributo. Tuttavia in alcuni casi è desiderabile che alcuni attributi fossero comuni a tutte le istanze; per utilizzare un termine tecnico, si vuole realizzare una comunicazione ad ambiente condiviso. Si pensi ad esempio ad un contatore che indichi il numero di istanze di una certa classe in un certo istante...
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 inizializzarlo tramite un costruttore:

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

Il motivo è abbastanza ovvio, qualunque operazione sul membro **static** nel corpo del costruttore verrebbe eseguita ogni volta che si istanzia la classe, una inizializzazione eseguita tramite costruttore verrebbe quindi ripetuta più volte rendendo inutili i membri statici.
L'inizializzazione di un attributo **static** va eseguita successivamente alla sua dichiarazione ed al di fuori della dichiarazione di classe:

```
class MyClass {
  public:
    MyClass();
    /* ... */

  private:
    static int Counter;
    char * String;
    /* ... */
};

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;         // Ok, non è una inizializzazione
  /* ... */
}

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

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 = 0;
    /* ... */
};

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

Si noti che nella definizione della funzione membro statica la keyword **static** non è stata ripetuta, essa è necessaria solo nella dichiarazione (anche in caso di definizione **inline**). Ci si può chiedere quale motivo ci possa essere per dichiarare un metodo **static**, ci sono essenzialmente tre giustificazioni:

- 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 puntatore all'istanza di classe per la quale 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 particolrare istanza, ricorrendo al risolutore di scope come nel seguente esempio:

```
MyClass Obj;
```

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

Non è possibile dichiarare **static** un costruttore o un distruttore.

Membri const e mutable

Oltre ad attributi di tipo **static**, è possibile avere attributi **const**; in questo caso però l'attributo **const** non è trattato come una normale 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), la cosa diverrà più evidente quando vedremo che il generico membro di una classe può essere a sua volta una istanza di una generica classe.

È anche possibile avere funzioni membro **const** 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 (che per loro natura non possono essere modificate). Per dichiarare una funzione membro **const** è necessario far seguire la lista dei parametri dalla keyword **const**, come mostrato nel seguente esempio:

```
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;
};

int main(int, char* []) {
  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!
  return 0;
}
```

Si osservi che se la funzione membro ***GetConstMember()*** fosse stata definita fuori dalla dichiarazione di classe, avremmo dovuto nuovamente esplicitare le nostre intenzioni:

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

    int GetConstMember() const;

    /* ... */
};

int MyClass::GetConstMember() const {
  return ConstMember;
}
```

Avremmo dovuto cioè esplicitare nuovamente il **const** (cosa che non avviene con le funzioni membro **static**).
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).

Talvolta può essere necessario che una funzione membro costante possa accedere in scrittura ad uno o più attributi della classe, situazioni di questo genere sono rare ma possibili (si pensi ad un oggetto che mappi un dispositivo che debba trasmettere dati residenti in ROM attraverso una porta hardware, solo metodi **const** possono accedere alla ROM...). Una soluzione potrebbe essere quella di eseguire un cast per rimuovere la restrizione del **const**, ma una soluzione di questo tipo sarebbe nascosta a chi usa la classe.
Per rendere esplicita una situazione di questo tipo è stata introdotta la keyword **mutable**, un attributo dichiarato **mutable** può essere modificato anche da funzioni membro costanti:

```
class AccessCounter {
  public:
    AccessCounter();
    const double GetPIValue() const;
    const int GetAccessCount() const;

  private:
    const double PI;
    mutable int Counter;
};
```

```
AccessCounter::AccessCounter() : PI(3.14159265),
                                 Counter(0) {}

const double AccessCounter::GetPIValue() const {
  ++Counter;       // Ok!
  return PI;
}

const int AccessCounter::GetAccessCount() const {
  return Counter;
}
```

L'esempio (giocattolo) mostra il caso di una classe che debba tenere traccia del numero di accessi in lettura ai suoi dati, senza **mutable** e senza ricorrere ad un cast esplicito la soluzione ad un problema simile sarebbe stata più artificiosa e complicata.

Costanti vere dentro le classi

Poichè 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 BadArray {
  public:
    /* ... */

  private:
    const int Size;
    char String[Size];      // Errore!
};
```

perchè non si può stabilire a tempo di compilazione il valore di ***Size***. Le possibili soluzioni al problema sono due.
La soluzione tradizionale 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 Array {
  public:
    /* ... */

  private:
    enum { Size = 20 };
    char String[Size];      // Ok!
};
```

Si osservi 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.
Questa soluzione, pur risolvendo il nostro problema, soffre di una grave limitazione: possiamo avere solo costanti intere. Una soluzione definitiva al

nostro problema la si trova utilizzando contemporaneamente le keyword **static** e **const**:

```
class Array {
  public:
    /* ... */

  private:
    static const int Size = 20;
    char String[Size];            // Ok!
};
```

Essendo **static**, ***Size*** viene inizializzata prima della creazione di una qualunque istanza della classe ed essendo **const** il suo valore non può essere modificato e risulta quindi prefissato già a compile time. Le costanti dichiarate in questo modo possono avere tipo qualsiasi e in questo caso il compilatore può non allocare alcuna memoria per esse, si ricordi solo che non tutti i compilatori potrebbero accettare l'inizializzazione della costante nella dichiarazione di classe, in questo caso è sempre possibile utilizzare il metodo visto per gli attributi static:

```
class Array {
  public:
    /* ... */

  private:
    static const int Size;
    char String[Size];          // Ok!
};

const int Array::Size = 20;
```

Anche in questo caso non bisogna riutilizzare **static** in fase di inizializzazione.

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;
int C = 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):

```
class VolatileClass {
  public:
    VolatileClass(int ID);
    long int ReadFromPort() volatile;
    /* ... */

  private:
    volatile long int ComPort;
    const int PortID;
    /* ... */
};

VolatileClass::VolatileClass(int ID) : PortID(ID) {}

long int VolatileClass::ReadFromPort() volatile {
  return ComPort;
}
```

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, possono essere mappati tramite oggetti dichiarati **const volatile**:

```
class Keyboard {
  public:
    Keyboard();
    const char ReadInput() const volatile;
    /* ... */

  private:
    const volatile char* Buffer;
};
```

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 ad esempio si desidera eseguire l'overloading degli operatori **ostream& operator<<(ostream& o, T& Obj)** e **istream& operator>>(istream& o, T& Obj)** per estendere le operazioni di I/O alla classe ***T*** che si vuole realizzare. 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 >
using namespace std;

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 sull'ereditarietà il secondo pilastro (dopo l'incapsulazione di dati e codice) della programmazione a oggetti.

Reimpiego per composizione

Benchè non sia stato esplicitamente mostrato, non c'è alcun limite alla complessità di un membro dato di un oggetto; un attributo può avere sia tipo elementare che tipo definito dall'utente, in particolare un attributo può a sua volta essere un oggetto.

```
class Lavoro {
  public:
    Lavoro(/* Parametri */);

    /* ... */

  private:
    /* ... */
};

class Lavoratore {
  public:
    Lavoratore(Lavoro* occupazione);
    /* ... */

  private:
    Lavoro* Occupazione;
    /* ... */
};
```

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. In questo caso il reimpiego è servito per modellare una proprietà della classe ***Lavoratore***, ma sono possibili casi ancora più 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];
};
```

In questo secondo esempio invece il reimpiego della classe ***Complex*** ci consente anche di definire le operazioni sulla classe ***Matrix*** in termini delle operazioni su ***Complex*** (un approccio matematicamente corretto).
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;
    unsiggned int Age;
};

class Student {
  public:
    Student(const char  name, unsigned age,
            const unsigned code);
    void PrintName();
    /* ... */

  private:
    Person Self;
    const unsigned int IdCode;  // numero di matricola
    /* ... */
};

Student::Student(const char* name, unsigned age,
                 const 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 tecnologie basate su questo approccio).

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 la 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 del concetto di ereditarietà che sarà esaminato in seguito. Sostanzialmente possiamo dire che poichè 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 (con il contenimento diretto invece oltre all'interfaccia viene fissato anche il comportamento ovvero l'implementazione del componente). 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 anzicchè 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 (e controlli) 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 caratteristice rendono il contenimento diretto meno flessibile ed espressivo di quello tramite puntatore e anche vero che lo rendono più efficente, non tanto perchè non è necessario passare tramite i puntatori, ma quanto per gli ultimi due punti.

Costruttori per oggetti composti

L'inizializzazione di un ogggetto composto richiede che siano inizializzate tutte le sue componenti. 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 >
using namespace std;

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;
};

int main(int, char* []) {
  BigObj Test(15);
  BigObj Test2;
  return 0;
}
```

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 poichè questo non chiama esplicitamente un costruttore per la componente ***SmallObj*** automaticamente il compilatore aggiunge una chiamata a ***SmallObj::SmallObj()***; nel caso in cui invece desiderassimo utilizzare un particolare costruttore per ***SmallObj*** bisogna chiamarlo esplicitamente come fatto in ***BigObj::BigObj(char, int, int)*** (utilizzato per inizializzare ***Test***).

Si poteva pensare di realizzare il costruttore 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 benchè funzionalmente equivalente al precedente, non genera lo stesso codice. Infatti poichè un costruttore per ***SmallObj*** non è esplicitamente chiamato nella lista di inizializzazione e poichè 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 una tale scelta è dovuta alla necessità di garantire sempre che un oggetto sia inizializzato prima di essere utilizzato.
Ovviamente poichè ogni classe possiede un solo distruttore, in questo caso 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 sottointeso 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.

Ora che è noto il meccanismo che regola l'inizializzazione di un oggetto composto, resta chiarire come vengono esattamente generati il costruttore di default e quello di copia.

Sappiamo che il compilatore genera automaticamente un costruttore di default se il programmatore non ne definisce uno qualsiasi, in questo caso il costruttore di default fornito automaticamente, come è facile immaginare, non fa altro che chiamare i costruttori di default delle singole componenti, generando un errore se per qualche componente non esiste un tale costruttore. Analogamente il costruttore di copia che il compilatore genera (solo se il programmatore non lo definisce esplicitamente) non fa altro che richiamare i costruttori di copia delle singole componenti.

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 eventualmente sostituendone una parte qualora una qualche funzione membro venisse ridefinita:

```
class Person {
  public:
    Person();
    ~Person();
    void PrintData();
    /* ... */

  private:
    char* Name;
    unsigned int Age;
    /* ... */
};

class Student : Person {  // Dichiara che la classe
  public:                 // Student eredita da Person
    Student();
    ~Student();
    /* ... */

  private:
    unsigned int IdCode;
    /* ... */
};
```

In pratica quanto fatto fin'ora 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 formalmente 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();

    private:
      char* Name;
      unsigned int Age;
      /* ... */
  };

/* Definizione dei metodi di Person */


  class Student : Person {
    public:
      Student();
      ~Student();
      void DoNothing();   // Metodo proprio di Student

    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, solo che riferirà agli attributi propri dell'istanza di ***Student*** su cui il metodo verrà invocato.

In molti casi è desiderabile che una certa funzione membro, ereditata dalla classe base, si comporti diversamente nella classe derivata. Come alterare dunque il comportamento (codice) ereditato? 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();
    cout << "Matricola: " << IdCode;
  }
```

Poichè ciò che desideriamo è che ***PrintData()*** richiamato su una istanza di ***Student*** visualizzi (oltre ai valori dei campi ereditati) anche il numero di matricola, si ridefinisce il metodo in modo da richiamare la versione ereditata (che visualizza i campi ereditati) e quindi si aggiunge il comportamento (codice) da noi desiderato.
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, poichè 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()*** di ***Person***.

Il linguaggio non pone alcuna limitazione circa il modo in cui ***PrintData()*** (o una qualunque funzione membro ereditata) possa essere ridefinita, in particolare avremmo potuto eliminare la chiamata a ***Person::PrintData()***, ma avremmo dovuto trovare un altro modo per accedere ai campi privati di *Person*. Al di là della fattibilità della cosa, non sarebbe comunque buona norma agire in tal modo, non è bene ridefinire un metodo con una semantica differente. 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 e risparmiare tempo e fatica.
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 perchè 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 e con una semantica prestabilita); il codice di questi membri è comunque disponibile all'interno della classe derivata (nel senso che possiamo richiamarli tramite il risolutore di scope ::).

Naturalmente la classe derivata può anche definire nuovi metodi, compresa la possibilità di eseguire l'overloading di una funzione ereditata (naturalmente la versione overloaded deve differire dalle precedenti per tipo e/o numero di parametri). Infine non è possibile ridefinire gli attributi (membri dato) della classe base.

Costruttori per classi derivate

La realizzazione di un costruttore per classi derivate non è diversa dal solito:

```
Student::Student() {
  /* ... */
}
```

Si deve però considerare che non si può accedere ai campi privati della classe base, e non è neanche possibile scrivere codice simile:

```
Student::Student() {
  Person(/* ... */);
  /* ... */
}
```

perchè quando si giunge all'interno del corpo del costruttore, l'oggetto è già stato costruito; ne esiste la possibilità di eseguire un assegnamento ad un attributo di tipo classe base. Come inizializzare dunque i membri ereditati? Nuovamente la soluzione consiste nell'utilizzare la lista di inizializzazione:

```
Student::Student() : Person(/* ... */) {
  /* ... */
}
```

Nel modo appena visto si chiede al compilatore di costruire e inizializzare i membri ereditati utilizzando un certo costruttore della classe base con i parametri attuali da noi indicati. Se nessun costruttore per la classe base viene menzionato il compilatore richiama il costruttore di default, generando un errore se la classe base non ne possiede uno.
Se il programmatore non specifica alcun costruttore per la classe derivata, il compilatore ne fornisce uno di default che richiama quello di default della classe base. Considerazioni analoghe valgono per il costruttore di copia fornito dal compilatore (richiama quello della classe base).

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 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);
    void Member2();
    /* ... */

  protected:
    int ProtectedMember;
    /* ... */

  private:
    /* ... */
};

class Derived1 : private MyClass {
  public:
    MyClass::PublicMember;     // esporta una specifica
                               // funzione membro
    using MyClass::Member2;    // si puo ricorrere
                               // anche alla using

    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. Un metodo alternativo è dato dall'uso della direttiva **using** per importare nel **namespace** della classe derivata, un nome appartenente al **namespace** della classe base.
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 desiderava esportare in modo che non compiano azioni dannose, il metodo però presenta alcuni inconvenienti:

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

D'altronde l'uso di funzioni di forward (cioè funzioni "guscio" che servono a richiamarne altre), risolverebbe il primo punto, ma non il secondo.

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*** poichè 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, 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):

Errore. L'argomento parametro è sconosciuto.

come si può vedere esistono diverse classi ottenute per ereditarietà multipla, **iostream** ad esempio che ha come classi basi **istream** e **ostream**.
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 clash):

```
class BaseClass1 {
  public:
    void Foo();
    void Foo2();
```

```
    /* ... */
};

class BaseClass2 {
  public:
    void Foo();
    /* ... */
};

class Derived : BaseClass1, BaseClass2 {
  // Non ridefinisce Foo()
  /* ... */
};
```

La classe ***Derived*** eredita più volte gli stessi membri e in particolare la funzione ***Foo()***, quindi una situazione del tipo

```
Derived A;
/* ... */
A.Foo()      // Errore, è ambiguo!
```

non può che generare un errore perchè il compilatore non sa a quale membro si riferisce l'assegnamento. Si noti che l'errore viene segnalato al momento in cui si tenta di chiamare il metodo e non al momento in cui ***Derived*** eredita, il fatto che un membro sia ereditato più volte non costituisce di per se alcun errore. Rimane comunque il problema di eliminare l'ambiguità nella chiamata di ***Foo()***, la soluzione consiste nell'utilizzare il risolutore di scope indicando esplicitamente quale delle due ***Foo()***:

```
Derived A;
/* ... */
A.BaseClass1::Foo()      // Ok!
```

in questo modo non esiste più alcuna ambiguità.
Alcune osservazioni:

1. quanto detto vale anche per i membri dato;

2. non è necessario che la stessa definizione si trovi in più classi basi dirette, è sufficiente che essa giunga alla classe derivata attraverso due classi basi distinte, ad esempio (con riferimento alla precedenti dichiarazioni):

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

   class SecondDerived : public BaseClass1,
                         public FirstDerived {
     /* ... */
   };
   ```

   Nuovamente ***SecondDerived*** 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 ***SecondDerived***);

3. il problema non si sarebbe posto se ***Derived*** avesse ridefinito la funzione membro ***Foo()***.

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.

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, public Derived2 {
  /* ... */
};
```

Di nuovo quello che succede è che alcuni membri (in particolare tutta una classe) sono duplicati nella classe ***Derived3***.
Consideriamo l'immagine in memoria di una istanza della classe ***Derived3***, la situazione che avremmo sarebbe la seguente:

Errore.Errore. L'argomento parametro è sconosciuto.

La classe ***Derived3*** contiene una istanza di ciasciuna 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? Una gravissima forma di inconsistenza!
La soluzione viene chiamata ereditarietà virtuale, e la si utilizza nel seguente modo:

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

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

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

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

Si tratta di un esempio che nella pratica non avrebbe alcuna validità, ma ottimo da un punto di vista didattico.
Vediamo più in dettaglio cosa e cambiato e come **virtual** opera. 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 da ***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:

Errore. L'argomento parametro è sconosciuto.

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

Il puntatore alla classe base virtuale non è visibile al programmatore, non bisogna tener conto di esso poichè viene aggiunto dal compilatore a compile-time, tutto il meccanismo è completamente trasparente, 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 ***Student*** (al fine di eliminare la fonte di inconsistenza) e aggiungendo a mano le definizioni di ***Employee***, in tal modo si risparmiano tempo e fatica riducendo la quantità di codice da produrre e limitando la possibilità di errori.
C'è però un costo da pagare: un livello di indirezione in più perchè l'accesso alle classi base virtuali (nell'esempio ***Person***) avviene tramite un puntatore.
L'ereditarietà virtuale risolve dunque l'ambiguità di cui sopra, nelle classi derivate le definizioni di una classe base virtuale sono presenti una volta sola, tuttavia il problema dell'ambiguità non è del tutto risolto, esistono ancora situazioni in cui il problema si ripropone. Supponiamo ad esempio che una delle classi intermedie ridefinisca una funzione membro della classe base:

```
class Person {
  public:
    void DoSomething();
  /* ... */
```

```
};

class Student : virtual public Person {
  public:
    void DoSomething();
  /* ... */
};

class Employee : virtual public Person {
  public:
    void DoSomething();
  /* ... */
};

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

Se ***Student-Employee*** non ridefinisce il metodo ***DoSomething()***, la situazione seguente presenterebbe ancora ambiguità:

```
Student-Employee Caio;
/* ... */
Caio.DoSomething();       // Ambiguo!
```

perchè la classe ***Student-Employee*** eredita nuovamente due diverse definizioni del metodo ***DoSomething()***.
Esiste anche un caso apparentemente ambiguo e simile al precedente:

```
class Person {
  public:
    void DoSomething();
  /* ... */
};

class Student : virtual public Person {
  public:
    void DoSomething();
  /* ... */
};

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

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

La situazione è però assai diversa, in questo caso solo una delle due classi base dirette ridefinisce il metodo ereditato da ***Person***; in ***Student-Employee*** abbiamo ancora due definizioni di ***DoSomething()***, ma una è in un certo senso "più aggiornata" delle altre. In situazioni del genere si dice che ***Student::DoSomething()*** **domina** ***Person::DoSomething()*** e in questi casi ambiguità tipo

```
  Student-Employee Caio;
  /* ... */
  Caio.DoSomething();
```

vengono risolte dal compilatore in favore della definizione dominante.
Si noti che **ci deve essere una sola definizione che domina tutte le altre**, altrimenti ci sarebbe ancora ambiguità.

Ritorniamo a parlare a proposito della classe base virtuale.
Nei vari costruttori delle classi derivate c'è implicitamente o esplicitamente una chiamata al costruttore della classe base virtuale, ma ora abbiamo una sola istanza di tale classe e non possiamo certo inizializzarla 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***.
È necessario stabilire un criterio deterministico che stabilisca chi deve inizializzare la classe virtuale. Lo standard stabilisce che il compito di inizializzare la classe base virtuale spetta alla **classe massimamente derivata**.
La classe massimamente derivata è quella che noi stiamo istanziando: se vogliamo creare un oggetto di tipo ***Student*** la classe massimamente derivata è in questo caso ***Student***, se invece stiamo istanziando ***Student-Employee*** allora è quest'ultima la classe massimamente derivata.
È dunque il costruttore della classe massimamente derivata che inizializza la classe virtuale.
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 Bruto:
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.
Naturalmente ci sarebbe un problema simile anche con il distruttore, bisogna evitare che si tenti di distruggere la classe base virtuale più volte; nuovamente è il compilatore che si assume l'onere di fare in modo che l'operazione venga eseguita una sola volta

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 linquaggi a oggetti (C++ incluso) che ha valso loro il soprannome di "Linguaggi degli attori": la possibilità di avere oggetti capaci di "recitare" di volta in volta il ruolo più appropriato, ma andiamo con ordine.
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 errori o 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 poichè le funzioni sono linkate staticamente dal linker. 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 in generale ciò che vorremmo). Il compilatore non può infatti rischiare di chiamare il metodo di una classe derivata, poichè 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 (perchè 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();
                  break;
  default      :  /* errore */
};
```

Una soluzione di questo tipo funziona ma è macchinosa, allunga il lavoro e 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();

  private:
    /* ... */
};
```

La definizione del metodo procede poi nel solito modo:

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

```
}
```

I metodi virtuali vengono ereditati allo stesso modo di quelli non **virtual**, 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 (**VTABLE**) che contiene per ogni metodo virtuale l'indirizzo alla corrispondente funzione, ogni istanza di quella classe conterrà poi al suo interno un puntatore (**VPTR**) alla **VTABLE**; una chiamata ad una funzione membro virtuale (e solo alle funzioni virtuali) viene risolta con del codice che accede alla **VTABLE** corrispondente al tipo dell'istanza tramite il puntatore contenuto nell'istanza stessa, ottenuta la **VTABLE** 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 (ovviamente il programma dovrà comunque essere modificato in modo che possa istanziare le nuove classi, ma il codice che gestisce che lavorava sul generico supertipo non avrà bisogno di modifiche), 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 perchè 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**: poichè 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 ha la possibilità di puntare a tutta una gerarchia di classi.

Limiti e regole del polimorfismo

Esistono limitazioni e regole connesse all'uso delle funzioni virtuali legate più o meno direttamente al funzionamento del polimorfismo; vediamole in dettaglio.

Una funzione membro virtuale non può essere dichiarata **static**: i metodi virtuali sono fortemente legati alle istanze, mentre i metodi statici sono legati alle classi.

Se una classe base dichiara un metodo **virtual**, tale metodo resterà sempre **virtual** in tutte le classi derivate anche quando la classe derivata lo ridefinisce senza dichiararlo esplicitamente **virtual**. Scrivere cioè

```
class Base {
  public:
    virtual void Foo();
};

class Derived: public Base {
  public:
    void Foo();
};
```

è lo stesso che scrivere

```
class Base {
  public:
    virtual void Foo();
};

class Derived: public Base {
  public:
    virtual void Foo();
};
```

Entrambe le forme sono lecite, possiamo cioè omettere **virtual** all'interno delle classi derivate quando vogliamo ridefinire un metodo ereditato.

È possibile che una classe derivata ridefinisca **virtual** un membro che non lo era nella classe base. In questo caso il comportamento del compilatore può sembrare strano; vediamo un esempio:

```
#include < iostream >
using std::cout;

class Base {
  public:
    void Foo() {
      cout << "Base::Foo()" << endl;
    }
};

class Derived1: public Base{
  public:
    virtual void Foo(){
```

```
      cout << "Derived1::Foo()" << endl;
    }
};

class Derived2: public Derived1 {
  public:
    virtual void Foo(){
      cout << "Derived2::Foo()" << endl;
    }
};

int main(int, char* []) {
  Base* BasePtr = new Derived1;
  Derived1* DerivedPtr = new Derived2;
  BasePtr -> Foo();
  DerivedPtr -> Foo();
  return 0;
}
```

Eseguendo il precedente codice, otterreste questo output:

```
Base::Foo()
Derived2::Foo()
```

Essendo ***BasePtr*** un puntatore alla classe base, il compilatore non può forzare il late binding perchè l'istanza puntata potrebbe essere effettivamente di tipo ***Base***; per non rischiare una operazione che potrebbe mandare in crash il sistema, il compilatore adotta un approccio conservativo e chiama sempre ***Base::Foo()*** (al più il programma non farà la cosa giusta, ma l'integrità del resto del sistema non sarà compromessa). Nel secondo caso, poichè ***DerivedPtr*** può puntare solo a istanze di ***Derived1*** o sue sottoclassi, viene eseguito regolarmente il late binding perchè il metodo ***Foo()*** sarà sempre **virtual**.
Da questo comportamento deriva un suggerimento ben preciso: se una classe potrebbe in futuro essere derivata (se pensate cioè che ci siano validi motivi per specializzarla ulteriormente), è bene che i metodi dell'interfaccia (potenzialmente soggetti a ridefinizione) siano sempre **virtual**, onde evitare potenziali errori; d'altronde se non si usa il polimorfismo è comunque possibile forzare la risoluzione del binding a compile time.

Non è possibile avere costruttori virtuali. Il meccanismo che sta alla base del late binding richiede che l'istanza che si sta creando sia correttamente collegata alla sua **VTABLE**, ma il compito di settare correttamente il puntatore (**VPTR**) alla **VTABLE** spetta proprio al costruttore che di conseguenza non può essere dichiarato **virtual**. Questo problema avviamente non esiste per i distruttori, anzi è bene che una classe che possieda metodi virtuali abbia anche il distruttore **virtual** in modo che distruggendo oggetti polimorfi venga sempre invocato il distruttore corretto.

Metodi virtuali chiamati dentro il costruttore o nel distruttore sono sempre risolti staticamente. Il motivo è semplice e riconducibile all'ordine in cui sono chiamati costruttori e distruttori.
Consideriamo il seguente esempio:

```
class Base {
  public:
    Base();
    ~Base();
    virtual void Foo();
};
```

```
Base::Base() {
  Foo();
}

Base::~Base() {
 Foo();
}

void Base::Foo() {
  /* ... */
}

class Derived: public Base {
  public:
    virtual void Foo();
};

void Derived::Foo() {
  /* ... */
}
```

La costruzione di un oggetto ***Derived*** richiede che sia prima eseguito il costruttore di ***Base***. Al momento in cui viene eseguita la chiamata a ***Foo()*** contenuta nel costruttore della classe base il puntatore alla **VTABLE** può al più puntare alla **VTABLE** della classe ***Base*** perchè il costruttore della classe derivata non è ancora stato eseguito e non è quindi possibile chiamare ***Derived::Foo()***, si può chiamare solo la versione locale di ***Foo()***. La situazione è analoga nel caso dei distruttori, al momento in cui viene eseguito il distruttore della classe bae, il distruttore della classe derivata è già stato eseguito ed il puntatore alla **VTABLE** non è più valido; di conseguenza si può invocare solo ***Base::Foo()***.
Il suggerimento è quindi quello di evitare per quanto possibile la chiamata di metodi virtuali all'interno di costruttori e distruttori, il risultato che potreste ottenere molto probabilmente non sarebbe quello desiderato.

Un potenziale errore legato all'uso di funzioni virtuali è possibile quando una funzione membro richiama un'altra funzione membro virtuale. Consideriamo questo frammento di codice:

```
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 Foo2();
    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 non virtuale (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.
Il risultato in queste situazioni è che il comportamento che una classe può avere è molto difficile da controllare e potrebbe essere potenzialmente errato; l'errore legato a situazioni di questo tipo è noto in letteratura come **fragile class problem** è può essere causa di forti inconsistenze.

Il polimorfismo è uno strumento estremamente potente, tuttavia richiede una approfondita comprensione del suo funzionamente per essere utilizzato correttamente e in modo profiquo.

Classi astratte

Il meccanismo dell'ereditarietà e quello del polimorfismo 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 e 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 che rispetti una certa interfaccia:

```
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() {
  /* ... */
}

void TPoint::Erase() {
  /* ... */
}
```

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, perchè possiamo dichiarare puntatori di tale tipo per gestire una lista di oggetti polimorfi che possiamo utilizzare senza preoccuparci del fatto che possano essere punti, triangoli o quant'altro.

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 perchè 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 perchè spesso solo gli ideatori di un programma erano in grado di comprendene 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 rimarràsempre unario, uno binario dovrà applicarsi sempre a due operandi; analogamente uno associativo a sinistra rimmarrà sempre associativo a sinistra.

3. Non è concessa la possibilità di eseguire l'overloading di alcuni operatori, ad esempio l'operatore ternario **? :**, l'operatore **sizeof**, l'operatore **.*** e l'operatore punto (per la selezione dei campi di una struttura).

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 l'operatore **->**.

A parte queste poche restrizioni non esistono molti 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 A, B;
/* ... */
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 B;
}
```

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;       // B.operator=(A.operator-());
C = A+B;      // C.operator=(A.operator+(B));
C = A+(-B);   // C.operator=(A.operator+(B.operator-()))
C = A-B;      // errore!
              // complex& Complex::operator-(Complex&)
              // non definito.
```

L'ultimo esempio è errato poichè 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 importante e bisogna prestare attenzione quando lo si ridefinisce. La sua semantica classica è quella di modificare il valore dell'oggetto cui è applicato con quello ricevuto come parametro e restituire poi tale valore al fine di consentire espressioni del tipo

```
A = B = C = < Valore >
```

che è equivalente a

```
A = (B = (C = < Valore >));
```

Il prototipo standard dell'operatore di assegnamento è

```
X& X::operator=(const X&);
```

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& X::operator=(const X&)***, il compilatore ne fornisce uno automaticamente. Lo standard 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.
Ci sono due buone norme da tenere presenti quando si ridefinisce l'operatore di assegnamento:

1. Evitare che un oggetto assegni a se stesso;
2. Ritornare un reference a ***this**.

È importante evitare autoassegnamenti, perchè se una operazione di assegnamento comporta la deallocazione di risorse, è facile che un autoassegnamento porti l'oggetto in uno stato inconsistente:

```
class X {
  public:
    X& operator=(const X& rsh);

  private:
    char* Str;
};

X& X::operator=(const X& rsh) {
  delete[] Str;
  // Se this==&rsh le due istruzioni successive
  // hanno comportamento non definito
  Str = new char[strlen(rsh.Str)+1];
  strcpy(Str, rsh.Str);
  /* ... */
}
```

Inoltre è importante ritornare un riferimento a ***this** perchè l'argomento dell'operatore è un riferimento a costante, mentre il valore generalmente restituito è un reference non const. Si osservi che eliminare il **const** dal parametro non è una soluzione praticabile poichè impedirebbe l'assegnamento di valori costanti:

```
Integer& Integer::operator=(Integer& rsh) {
  /* ... */
}

Integer I = 3;  // Errore!
```

In generale dunque la struttura di un buon operatore di assegnamento è:

```
X& X::operator=(X& rsh) {
  if (this!=&rsh) {
    /* ... */
  }
  return *this;
}
```

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 (da tipi diversi), ma è quello di cui sopra che il compilatore fornisce quando manca.

L'operatore di sottoscrizione

Anche l'operatore di sottoscrizione **[ ]** può essere sottoposto a overloading. 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]*** dell'operatore predefinito è dato da ****(Arg1+Arg2)*** cioè il valore contenuto all'indirizzo ***Arg1+Arg2***. 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 essere 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 < ArraySize) 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 poichè 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 dereferenzazione **->** il cui comportamento è un po' difficile da capire.
Se ***T*** è una classe che ridefinisce **->** (l'operatore di dereferenzazione deve essere un funzione membro non statica) e ***Obj*** è una istanza di tale classe, l'espressione

```
Obj -> Field;
```

è valutata come

```
(Obj.operator ->()) -> Field;
```

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, che serviva solo per alcuni suoi effetti collaterali. Quanto alle parentesi, esse sono necessarie perchè l'assegnamento ha la precedenza sulla virgola.
Questo operatore è comunque sovraccaricato raramente.

Autoincremento e autodecremento

Gli operatori **++** e **--** meritano un breve accenno poichè 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.
Come operatori globali, 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.
Ridefiniti come funzioni membro, la versione prefissa non presenta nel suo prototipo alcun parametro (il parametro è l'oggetto su cui l'operatore è chiamato), la forma postfissa ha un prototipo con il solo argomento fittizio.

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**.
Poichè entrambi gli operatori hanno un prototipo predefinito, non è possibile avere più versioni overloaded di **new** e **delete**, 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.
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**: nella **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);
}

int main() {
  // Supponiamo di voler mappare un certo
  // dispositivo hardware tramite una istanza di
  // un apposito tipo
  const void* DeviceAddr = 0xA23;

  // Si osservi il modo in cui viene fornito
  // il secondo parametro della new
  TMyDevice Unit1 = new(DeviceAddr) TMyDevice;

  /* ... */

  return 0;
}
```

Si noti che non c'è una **delete** duale per questa forma di **new** (perchè una **delete** non può sapere se e come è stata allocato l'oggetto da deallocare), questo vuol dire che gli oggetti allocati nel modo appena visto (cioè fornendo alla **new** un indirizzo) vanno deallocati con tecniche diverse.
È possibile sovraccaricare anche le versioni per array di questi operatori. I prototipi di **new[]** e **delete[]** sono identici a quelli già visti in particolare il valore che il compilatore fornisce come primo parametro alla **new[]** è ancora la dimensione complessiva del blocco da allocare.

Per terminare il discorso su questi operatori, bisogna accennare a ciò che accade quando una allocazione non riesce (generalmente per mancanza di memoria). In caso di fallimento della **new**, lo standard prevede che venga chiamata una apposita funzione (detta *new-handler*) il cui comportamento di default è sollevare una eccezione di tipo **std::bad_alloc** che bisogna intercettare per gestire il possibile fallimento.
È possibile modificare tale comportamento definendo e istallando una nuova *new-handler*. La generica *new-handler* deve essere una funzione che non riceve alcun parametro e restituisce **void**, tale funzione va installata tramite una chiamata a ***std::set_new_handler*** il cui prototipo è dato dalle seguenti definizioni:

```
typedef void (*new_handler)();
// new_handler è un puntatore ad una funzione
// che non prende parametri e restituisce void

new_handler set_new_handler(new_handler HandlePtr);
```

La funzione ***set_new_handler*** riceve come parametro la funzione da utilizzare quando la **new** fallisce e restituisce un puntatore alla vecchia *new-handler*. Ecco un esempio di come utilizzare questo strumento:

```
void NoMemory() {
  // cerr è come cin, ma si usa per inviare
  // messaggi di errore...
  cerr << "Out of memory... Program aborted!" << endl;
  abort();
}

int main(int, char* []) {
  new_handler OldHandler = set_new_handler(NoMemory);

  char* Ptr = new char[1000000000];

  set_new_handler(OldHandler);

  /* ... */
}
```

Il precedente esempio funziona perchè la funzione standard ***abort()*** provoca la terminazione del programma, in realtà la *new-handler* viene richiamata da **new** finchè l'operatore non è in grado di restituire un valore valido, per cui bisogna tenere conto di ciò quando si definisce una routine per gestire i fallimenti di **new**.

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 solitamente dichiarato **friend** delle 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, perchè 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;
```

Template

Il meccanismo dell'ereditarietà consente il riutilizzo di codice precedentemente scritto, l'idea è quella di riconoscere le proprietà di un certo insieme di valori (tipo) e di definirle realizzando una classe base (astratta) da specializzare poi caso per caso secondo le necessità. Quando riconosciamo che gli oggetti con cui si ha a che fare sono un caso particolare di una qualche classe della gerarchia, non si fa altro che specializzarne la classe più opportuna.
Esiste un'altro approccio che per certi versi procede in senso opposto; anzicchè partire dai valori per determinarne le proprietà, si definiscono a priori le proprietà scrivendo codice che lavora su tipologie (non note) di oggetti che soddisfano tali proprietà (ad esempio l'esistenza di una relazione di ordinamento) e si riutilizza tale codice ogni qual volta si scopre che gli oggetti con cui si ha a che fare soddisfano quelle proprietà.
Quest'ultima tecnica prende il nome di **programmazione generica** ed il C++ la rende disponibile tramite il meccanismo dei **template**.
Un **template** altro non è che codice parametrico, dove i parametri possono essere sia valori, sia nomi di tipo. Tutto sommato questa non è una grossa novità, le ordinarie funzioni sono già di per se del codice parametrico, solo che i parametri possono essere unicamente valori di un certo tipo.

Classi contenitore

Supponiamo di voler realizzare una lista generica facilmente riutilizzabile. Sulla base di quanto visto fino ad ora l'unica soluzione possibile sarebbe quella di realizzare una lista che contenga puntatori ad una generica classe **TInfo** che rappresenta l'interfaccia di un generico oggetto memorizzabile nella lista:

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

class TList {
  public:
    TList();
    ~TList();
    void Store(TInfo* Object);
    /* ... */

  private:
    class TCell {
```

```
      public:
        TCell(TInfo* Object, TCell* Next);
        ~TCell();
        TInfo* GetObject();
        TCell* GetNextCell();
      private:
        TInfo* StoredObject;
        TCell* NextCell;
    };

    TCell* FirstCell;
};

TList::TCell::TCell(TInfo* Object, TCell* Next)
     : StoredObject(Object), NextCell(Next) {}

TList::TCell::~TCell() {
  delete StoredObject;
}

TInfo* TList::TCell::GetObject() {
  return StoredObject;
}

TList::TCell* TList::TCell::GetNextCell() {
  return NextCell;
}

TList::TList() : FirstCell(0) {}

TList::~TList() {
  TCell* Iterator = FirstCell;
  while (Iterator) {
    TCell* Tmp = Iterator;
    Iterator = Iterator -> GetNextCell();
    delete Tmp;
  }
}

void TList::Store(TInfo* Object) {
  FirstCell = new TCell(Object, FirstCell);
}
```

L'esempio mostra una parziale implementazione di una tale lista (che assume la proprietà degli oggetti contenuti), nella realtà ***TInfo*** e/o ***TList*** molto probabilmente sarebbero diverse al fine di fornire un meccanismo per eseguire delle ricerche all'interno della lista e varie altre funzionalità. Tuttavia il codice riportato è sufficiente ai nostri scopi.
Una implementazione di questo tipo funziona, ma soffre di (almeno) un grave difetto: la lista può memorizzare tutta una gerarchia di oggetti, e questo è utile e comodo in molti casi, tuttavia in molte situazioni siamo interessate a liste di oggetti omogenei e una soluzione di questo tipo non permette di verificare a compile time che gli oggetti memorizzati corrispondano tutti ad uno specifico tipo. Potremmo cercare (e trovare) delle soluzioni che ci permettano una verifica a run time, ma non a compile time. Supponete di aver bisogno di una lista per memorizzare figure geometriche e una'altra per memorizzare stringhe, nulla vi impedisce di memorizzare una stringa nella lista delle figure geometriche (poichè le liste memorizzano puntatori alla classe base comune ***TInfo***). Sostanzialmente una lista di questo tipo annulla i vantaggi di un type checking statico.
Alcune osservazioni...
In effetti non è necessario che il compilatore fornisca il codice macchina

relativo alla lista ancora prima che un oggetto lista sia istanziato, è sufficiente che tale codice sia generabile nel momento in cui è noto l'effettivo tipo degli oggetti da memorizzare. Supponiamo di avere una libreria di contenitori generici (liste, stack...), a noi non interessa il modo in cui tale codice sia disponibile, ci basta poter dire al compilatore "istanzia una lista di stringhe", il compilatore dovrebbe semplicemente prendere la definizione di lista data sopra e sostituire al tipo ***TInfo*** il tipo ***TString*** e quindi generare il codice macchina relativo ai metodi di ***TList***. Naturalmente perchè ciò sia possibile il tipo ***TString*** dovrebbe essere conforme alle specifiche date da ***TInfo***, ma questo il compilatore potrebbe agevolmente verificarlo. Alla fine tutte le liste necessarie sarebbero disponibili e il compilatore sarebbe in grado di eseguire staticamente tutti i controlli di tipo.
Tutto questo in C++ è possibile tramite il meccanismo dei **template**.

Classi template

La definizione di codice generico e in particolare di una classe template (le classi generiche vengono dette **template class**) non è molto complicata, la prima cosa che bisogna fare è dichiarare al compilatore la nostra intenzione di scrivere un **template** utilizzando appunto la keyword **template**:

```
template < class T >
```

Questa semplice dichiarazione (che non deve essere seguita da ";") dice al compilatore che la successiva dichiarazione utilizzerà un generico tipo **T** che sarà noto solo quando tale codice verrà effettivamente utilizzato, il compilatore deve quindi memorizzare quanto segue un po' cose se fosse il codice di una funzione **inline** per poi istanziarlo nel momento in cui ***T*** sarà noto.
Vediamo come avremmo fatto per il caso della lista vista sopra:

```
template < class TInfo >
class TList {
  public:
    TList();
    ~TList();
    void Store(TInfo& Object);
    /* ... */

  private:
    class TCell {
      public:
        TCell(TInfo& Object, TCell* Next);
        ~TCell();
        TInfo& GetObject();
        TCell* GetNextCell();
      private:
        TInfo& StoredObject;
        TCell* NextCell;
    };

    TCell* FirstCell;
};
```

Al momento l'esempio è limitato alle sole dichiarazioni, vedremo in seguito come definire i metodi del **template**.
Intanto, si noti che è sparita la dichiarazione della classe ***TInfo***, la keyword **template** dice al compilatore che ***TInfo*** rappresenta un nome di tipo qualsiasi (anche un tipo primitivo come **int** o **long double**). Le dichiarazioni quindi non fanno più riferimento ad un tipo esistente, là dove è stato utilizzato il nome fittizio ***TInfo***. Inoltre il contenitore non memorizza più tipi puntatore, ma

riferimenti alle istanze di tipo.
Supponendo di aver fornito anche le definizioni dei metodi, vediamo come istanziare la generica lista:

```
TList < double > ListOfReal;
double* AnInt = new double(5.2);
ListOfReal.Store(*AnInt);

TList < Student > MyClass;
Student* Pippo = new Student(/* ... */);
ListOfReal.Store(*Pippo);                  // Errore!
MyClass.Store(*Pippo);                     // Ok!
```

La prima riga istanzia la **classe template *TList*** sul tipo **double** in modo da ottenere una lista di **double**; si noti il modo in cui è stata istanziato il **template** ovvero tramite la notazione

***NomeTemplate* < *Tipo* >**

(si noti che ***Tipo*** va specificato tra parentesi angolate).

Il tipo di ***ListOfReal*** è dunque ***TList* < *double* >**. Successivamente viene mostrato l'inserzione di un **double** e il tentativo di inserimento di un valore di tipo non opportuno, l'errore sarà ovviamente segnalato in fase di compilazione.

La definizione dei metodi di ***TList*** avviene nel seguente modo:

```
template < class TInfo >
TList < TInfo >::
  TCell::TCell(TInfo& Object, TCell* Next)
       : StoredObject(Object), NextCell(Next) {}

template < class TInfo >
TList < TInfo >::TCell::~TCell() {
  delete &StoredObject;
}

template < class TInfo >
TInfo& TList < TInfo >::TCell::GetObject() {
  return StoredObject;
}

template < class TInfo >
  TList < TInfo >::TCell*
    TList < TInfo >::TCell::GetNextCell() {
      return NextCell;
    }

template < class TInfo >
TList < TInfo >::TList() : FirstCell(0) {}

template < class TInfo >
TList < TInfo >::~TList() {
  TCell* Iterator = FirstCell;
  while (Iterator) {
    TCell* Tmp = Iterator;
    Iterator = Iterator -> GetNextCell();
    delete Tmp;
  }
}

template < class TInfo >
```

```
void TList < TInfo >::Store(TInfo& Object) {
  FirstCell = new TCell(Object, FirstCell);
}
```

Cioè bisogna indicare per ogni membro che si tratta di codice relativo ad un **template** e contemporaneamente occorre istanziare la classe template utilizzando il parametro del **template**.

Un **template** può avere un qualsiasi numero di parametri non c'è un limite prestabilito; supponete ad esempio di voler realizzare un array associativo, l'approccio da seguire richiederebbe un **template** con due parametri e una soluzione potrebbe essere la seguente:

```
template < class Key, class Value >
class AssocArray {
  public:
    /* ... */
  private:
    static const int Size;
    Key KeyArray[Size];
    Value ValueArray[Size];
};

template < class Key, class Value >
const int AssociativeArray < Key, Value >::Size = 100;
```

Questa soluzione non pretende di essere ottimale, in particolare soffre di un limite: la dimensione dell'array è prefissata. Fortunatamente un **template** può ricevere come parametri anche valori di un certo tipo:

```
template < class Key, class Value, int size >
class AssocArray {
  public:
    /* ... */
  private:
    static const int Size;
    Key KeyArray[Size];
    Value ValueArray[Size];
};

template < class Key, class Value, int size >
const int AssocArray < Key, Value, size >::Size = size;
```

La keyword typename

Consideriamo il seguente esempio:

```
template < class T >
class TMyTemplate {
  public:
    /* ... */
  private:
    T::TId Object;
};
```

È chiaro dall'esempio che l'intenzione era quella di utilizzare un tipo dichiarato all'interno di T per istanziarlo all'interno del template.
Tale codice può sembrare corretto, ma in effetti il compilatore non produrrà il risultato voluto. Il problema è che il compilatore non può sapere in anticipo se ***T::TId*** è un identificatore di tipo o un qualche membro pubblico di ***T***.
Per default il compilatore assume che ***TId*** sia un membro pubblico del tipo ***T*** è l'unico modo per ovviare a ciò è utilizzare la keyword **typename** introdotta dallo standard:

```
template < class T >
class TMyTemplate {
  public:
    /* ... */
  private:
    typename T::TId Object;
};
```

La keyword **typename** indica al compilatore che l'identificatore che la segue deve essere trattato come un nome di tipo, e quindi nell'esempio precedente ***Object*** è una istanza di tale tipo. Si ponga attenzione al fatto che **typename** non sortisce l'effetto di una **typedef**, se si desidera dichiarare un alias per ***T::TId*** il procedimento da seguire è il seguente:

```
template < class T >
class TMyTemplate {
  public:
    /* ... */
  private:
    typedef typename T::TId Alias;

    Alias Object
};
```

Un altro modo corretto di utilizzare **typename** è nella dichiarazione di template:

```
template < typename T >
class TMyTemplate {
  /* ... */
};
```

In effetti se teniamo conto che il significato di **class** in una dichiarazione di template è unicamente quella di indicare un nome di tipo che è parametro del template e che tale parametro può non essere una classe (ma anche ***int*** o una ***struct***, o un qualsiasi altro tipo), si capisce come sia più corretto utilizzare **typename** in luogo di **class**. La ragione per cui spesso troverete ***class*** invece di ***typename*** è che prima dello standard tale keyword non esisteva.

Vincoli impliciti

Un importante aspetto da tenere presente quando si scrivono e si utilizzano **template** (siano essi **classi template** o, come vedremo, funzioni) è che la loro istanziazione possa richiedere che su uno o più dei parametri del **template** sia definita una qualche funzione o operazione. Esempio:

```
template < typename T >
class TOrderedList {
  public:
    /* ... */
    T& First();            // Ritorna il primo valore
                           // della lista
    void Add(T& Data);
    /* ... */

  private:
    /* ... */
};

/* Definizione della funzione First() */

template < typename T >
void TOrderedList< T >::Add(T& Data) {
  /* ... */
  T& Current = First();
  while (Data < Current) {    // Attenzione qui!
    /* ... */
  }
  /* ... */
}
```

la funzione ***Add*** tenta un confronto tra due valori di tipo ***T*** (parametro del **template**). La cosa è perfettamente legale, solo che implicitamente si assume che sul tipo ***T*** sia definito **operator <**; il tentativo di istanziare tale **template** con un tipo su cui tale operatore non è definito è però un errore che può essere segnalato solo quando il compilatore cerca di creare una istanza del template. Purtroppo il linguaggio segue la via dei vincoli impliciti, ovvero non fornisce alcun meccanismo per esplicitare assunzioni fatte sui parametri dei **template**, tale compito è lasciato ai messaggi di errore del compilatore e alla buona volontà dei programmatori che dovrebbero opportunamente commentare situazioni di questo genere.
Problemi di questo tipo non ci sarebbero se si ricorresse al polimorfismo, ma il prezzo sarebbe probabilmente maggiore dei vantaggi.

Funzioni template

Oltre a **classi template** è possibile avere anche **funzioni template**, utili quando si vuole definire solo un'operazione e non un tipo di dato, ad esempio la libreria standard definisce la funzione ***min*** più o meno in questo modo:

```
template < typename T >
T& min(T& A, T& B) {
  return (A < B)? A : B;
}
```

Si noti che la definizione richiede implicitamente che sul tipo ***T*** sia definito **operator <**. In questo modo è possibile calcolare il minimo tra due valori senza che sia definita una funzione ***min*** specializzata:

```
int main(int, char* []) {
  int A = 5;
  int B = 10;
```

```
    int C = min(A, B);

    TMyClass D(/* ... */);
    TMyClass E(/* ... */);

    TMyClass F = min(D, E);

    /* ... */
    return 0;
  }
```

Ogni qual volta il compilatore trova una chiamata alla funzione ***min*** istanzia (se non era gia stato fatto prima) la **funzione template** (nel caso delle funzioni l'istanziazione è un processo totalmente automatico che avviene quando il compilatore incontra una chiamata alla **funzione template** producendo una nuova funzione ed effettuando una chiamata a tale istanza. In sostanza con un template possiamo avere tutte le versioni overloaded della funzione ***min*** che ci servono con un'unica definizione.
Si osservi che affinche la funzione possa essere correttamente istanziata, i parametri del **template** devono essere utilizzati nella lista dei parametri formali della funzione in quanto il compilatore istanzia le **funzioni template** sulla base dei parametri attuali specificati al momento della chiamata:

```
template < typename T > void F1(T);
template < typename T > void F1(T*);
template < typename T > void F1(T&);
template < typename T > void F1();            // Errore
template < typename T, typename U > void F1(T, U);
template < typename T, typename U > int F1(T);// Errore
```

Questa restrizione non esiste per le **classi template**, perchè gli argomenti del **template** vengono specificati esplicitamente ad ogni istanziazione.

Template ed ereditarietà

È possibile utilizzare contemporaneamente ereditarietà e template in vari modi. Supponendo di avere una gerarchia di figure geometriche potremmo ad esempio avere le seguenti istanze di ***TList***:

```
TList < TBaseShape > ShapesList;
TList < TRectangle > RectanglesList;
TList < TTriangle > TrianglesList;
```

tuttavia in questi casi gli oggetti ***ShapesList***, ***RectanglesList*** e ***TrianglesList*** non sono legati da alcun vincolo di discendenza, indipendentemente dal fatto che le classi ***TBaseShape***, ***TRectangle*** e ***TTriangle*** lo siano o meno. Naturalmente se ***TBaseShape*** è una classe base delle altre due, è possibile memorizzare in ***ShapesList*** anche oggetti ***TRectangle*** e ***TTriangle*** perchè in effetti ***TList*** memorizza dei riferimenti, sui quali valgono le stesse regole per i puntatori a classi base.
Istanze diverse dello stesso **template** non sono mai legate dunque da relazioni di discendenza, indipendentemente dal fatto che lo siano i parametri delle istanze del **template**. La cosa non è poi tanto strana se si pensa al modo in cui sono gestiti e istanziati i **template**.
Un altro modo di combinare ereditarietà e **template** è dato dal seguente esempio:

```
template< typename T >
class Base {
  /* ... */
};

template< typename T >
class Derived : public Base< T > {
  /* ... */
};

Base < double > A;
Base < int > B;
Derived < int > C;
```

in questo caso l'ereditarietà è stata utilizzata per estendere le caratteristiche della **classe template *Base***, Tuttavia anche in questo caso tra le istanze dei **template** non vi è alcuna relazione di discendenza, in particolare non esiste tra ***B*** e ***C***; un puntatore a ***Base < T >*** non potrà mai puntare a ***Derived < T >***.

Conclusioni

Template ed ereditarietà sono strumenti assai diversi pur condividendo lo stesso fine: il riutilizzo di codice già sviluppato e testato. In effetti la programmazione orientata agli oggetti e la programmazione generica sono due diverse scuole di pensiero relative al modo di implementare il polimorfismo (quello della OOP è detto polimorfismo per inclusione, quello della programmazione generica invece è detto polimorfismo parametrico). Le conseguenze dei due approcci sono diverse e diversi sono i limiti e le possibilità (ma si noti che tali differenze dipendono anche da come il linguaggio implementa le due tecniche). Tali differenze non sempre comunque pongono in antitesi i due strumenti: abbiamo visto qualche limite del polimorfismo della OOP nel C++ (ricordate il problema delle classi contenitore) e abbiamo visto il modo elegante in cui i **template** lo risolvono. Anche i **template** hanno alcuni difetti (ad esempio quello dei vincoli impliciti, o il fatto che i **template** generano eseguibili molto piu grandi) che non troviamo nel polimorfismo della OOP. Tutto ciò in C++ ci permette da un lato di scegliero lo strumento che più si preferisce (e in particolare di scegliere tra un programma basato su OOP e uno su programmazione generica), e dall'altra parte di rimediare ai difetti dell'uno ricorrendo all'altro. Ovviamente saper mediare tra i due strumenti richiede molta pratica e una profonda conoscenza dei meccanismi che stanno alla loro base.

Le eccezioni

Durante l'esecuzione di un applicativo possono verificarsi delle situazioni di errore non verificabili a compile-time, che in qualche modo vanno gestiti.
Le possibili tipologie di errori sono diverse ed in generale non tutte trattabili allo stesso modo. In particolare possiamo distinguere tra errori che non compromettono il funzionamento del programma ed errori che invece costituiscono una grave impedimento al normale svolgimento delle operazioni.
Tipico della prima categoria sono ad esempio gli errori dovuti a errato input

dell'utente, facili da gestire senza grossi problemi. Meno facili da catturare e gestire è invece la seconda categoria cui possiamo inserire ad esempio i fallimenti relativi all'acquisizione di risorse come la memoria dinamica; questo genere di errori viene solitamente indicato con il termine di **eccezioni** per sottolineare la loro caratteristica di essere eventi particolarmente rari e di comportare il fallimento di tutta una sequenza di operazioni.
La principale difficoltà connessa al trattamento delle eccezioni è quella di riportare lo stato dell'applicazione ad un valore consistente. Il verificarsi di un tale vento comporta infatti (in linea di principio) l'interruzione di tutta una sequenza di operazioni rivolta ad assolvere ad una certa funzionalità, allo svuotamento dello stack ed alla deallocazione di eventuali risorse allocate fino a quel punto relativamente alla richiesta in esecuzione. Le informazioni necessarie allo svolgimento di queste operazioni sono in generale dipendenti anche dal momento e dal punto in cui si verifica l'eccezione e non è quindi immaginabile (o comunque facile) che la gestione dell'errore possa essere incapsulata in un unico blocco di codice richiamabile indipendentemente dal contesto in cui si verifica il problema.
In linguaggi che non offrono alcun supporto, catturare e gestire questi errori può essere particolarmente costoso e difficile, al punto che spesso si rinuncia lasciando sostanzialmente al caso le conseguenze. Il C++ comunque non rientra tra questi linguaggi e offre alcuni strumenti che saranno oggetto dei successivi paragrafi di questo capitolo.

Segnalare le eccezioni

Il primo problema che bisogna affrontare quando si verifica un errore è capire dove e quando bisogna gestire l'anomalia.
Poniamo il caso che si stia sviluppando una serie di funzioni matematiche, in particolare una che esegue la radice quadrata. Come comportarsi se l'argomento della funzione è un numero negativo? Le possibilità sono due:

- Terminare il processo;
- Segnalare l'errore al chiamante.

Probabilmente la prima possibilità è eccessivamente drastica, tanto più che non sappiamo a priori se è il caso di terminare oppure se il chiamante possa prevedere azioni alternative da compiere in caso di errore (ad esempio ignorare l'operazione e passare alla successiva). D'altronde neanche la seconda possibilità sarebbe di per sè una buona soluzione, cosa succede se il chiamante ignora l'errore proseguendo come se nulla fosse?
È chiaramente necessario un meccanismo che garantisca che nel caso il chiamante non catturi l'anomalia qualcuno intervenga in qualche modo.
Ma andiamo con ordine, e supponiamo che il chiamante preveda del codice per gestire l'anomalia.
Se al verificarsi di un errore grave non si dispone delle informazioni necessarie per decidere cosa fare, la cosa migliore da farsi è segnalare la condizione di errore a colui che ha invocato l'operazione. Questo obiettivo viene raggiunto con la keyword **throw**:

```
int Divide(int a, int b) {
  if (b) return a/b;
  throw "Divisione per zero";
}
```

L'esecuzione di **throw** provoca l'uscita dal blocco in cui essa si trova (si noti che in questo caso la funzione non è obbligata a restituire alcun valore tramite **return**) e in queste situazioni si dice che la funzione ha **sollevato (o lanciato) una eccezione**.

La **throw** accetta un argomento come parametro che viene utilizzato per creare un oggetto che non ubbidisce alle normali regole di scope e che viene restituito a chi ha tentato l'esecuzione dell'operazione (nel nostro caso al blocco in cui ***Divide*** è stata chiamata). Il compito di questo oggetto è trasportare tutte le informazioni utili sull'evento.
L'argomento di **throw** può essere sia un tipo predefinito che un tipo definito dal programmatore.
Per compatibilità con il vecchio codice, una funzione non è tenuta a segnalare la possibilità che possa lanciare una eccezione, ma è buona norma avvisare dell'eventualità segnalando quali tipologie di eccezioni sono possibili. Allo scopo si usa ancora **throw** seguita da una coppia di parentesi tonde contenente la lista dei tipi di eccezione che possono essere sollevate:

```
int Divide(int a, int b) throw(char*) {
  if (b) return a/b;
  throw "Errore";
}

void MoreExceptionTypes() throw(int, float, MyClass&) {
  /* ... */
}
```

Nel caso della ***Divide*** si segnala la possibilità che venga sollevata una eccezione di tipo **char***; nel caso della seconda funzione invece a seconda dei casi può essere lanciata una eccezione di tipo **int**, oppure di tipo **float**, oppure ancora una di tipo ***MyClass&*** (supponendo che ***MyClass*** sia un tipo precedentemente definito).

Gestire le eccezioni

Quanto abbiamo visto chiaramente non è sufficiente, non basta poter sollevare (segnalare) una eccezione ma è necessario poterla anche catturare e gestire.
L'intenzione di catturare e gestire l'eventuale eccezione deve essere segnalata al compilatore utilizzando un **blocco try**:

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

int Divide(int a, int b) throw(char*) {
  if (b) return a/b;
  throw "Errore";
}

int main() {
  cout << "Immettere il dividendo: ";
  int a;
  cin >> a;
  cout << endl << "Immettere il divisore: ";
  int b;
  cin >> b;
  try {
    cout << Divide(a, b);
  }
  /* ... */
}
```

Utilizzando **try** e racchidendo tra parentesi graffe (le parentesi si devono utilizzate sempre) il codice che può generare una eccezione si segnala al

compilatore che siamo pronti a gestire l'eventuale eccezione.
Ci si potrà chiedere per quale motivo sia necessario informare il compilatore dell'intenzione di catturare e gestire l'eccezione, il motivo sarà chiaro in seguito, al momento è sufficiente sapere che ciò ha il compito di indicare quando certi automatismi dovranno arrestarsi e lasciare il controllo a codice ad hoc preposto alle azioni del caso.
Il codice in questione dovrà essere racchiuso all'interno di un **blocco catch** che deve seguire il **blocco try**:

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

int Divide(int a, int b) throw(char*) {
  if (b) return a/b;
  throw "Errore, divisione per 0";
}

int main() {
  cout << "Immettere il dividendo: ";
  int a;
  cin >> a;
  cout << endl << "Immettere il divisore: ";
  int b;
  cin >> b;
  cout << endl;
  try {
    cout << "Il risultato è " << Divide(a, b);
  }
  catch(char* String) {
    cout << String << endl;
    return -1;
  }
  return 0;
}
```

Il generico **blocco catch** potrà gestire in generale solo una categoria di eccezioni o una eccezione generica. Per fornire codice diverso per diverse tipologie di errori bisognerà utilizzare più **blocchi catch**:

```
try {
   /* ... */
}
catch(Type1 Id1) {
  /* ... */
}
catch(Type2 Id2) {
  /* ... */
}

/* Altre catch */

catch(TypeN IdN) {
  /* ... */
}

/* Altro */
```

Ciascuna **catch** è detta **exception handler** e riceve un parametro che è il tipo di eccezione che viene gestito in quel blocco. Nel caso generale un blocco **try** sarà seguito da più blocchi **catch**, uno per ogni tipo di eccezione possibile

all'interno di quel **try**. Si noti che le **catch** devono seguire immediatamente il blocco **try**.

Quando viene generata una eccezione (**throw**) il controllo risale indietro fino al primo blocco **try**. Gli oggetti staticamente allocati (che cioè sono memorizzati sullo stack) fino a quel momento nei blocchi da cui si esce vengono distrutti invocando il loro distruttore (se esiste). Nel momento in cui si giunge ad un blocco **try** anche gli oggetti staticamente allocati fino a quel momento dentro il blocco **try** vengono distrutti ed il controllo passa immediatamente dopo la fine del blocco.
Il tipo dell'oggetto creato con **throw** viene quindi confrontato con i parametri delle **catch** che seguono la **try**. Se viene trovata una **catch** del tipo corretto, si passa ad eseguire le istruzioni contenute in quel blocco, dopo aver inizializzato il parametro della **catch** con l'oggetto restituito con **throw**. Nel momento in cui si entra in un blocco **catch**, l'eccezione viene considerata gestita ed alla fine del blocco **catch** il controllo passa alla prima istruzione che segue la lista di **catch** (sopra indicato con "/* Altro */").
Vediamo un esempio:

```
#include < iostream >
#include < string.h >
using namespace std;

class Test {
    char Name[20];
  public:
    Test(char* name){
      Name[0] = '\0';
      strcpy(Name, name);
      cout << "Test constructor inside "
           << Name << endl;
    }
    ~Test() {
      cout << "Test distructor inside "
           << Name << endl;
    }
};

int Sub(int b) throw(int) {
  cout << "Sub called" << endl;
  Test k("Sub");
  Test* K2 = new Test("Sub2");
  if (b > 2) return b-2;
  cout << "exception inside Sub..." << endl;
  throw 1;
}

int Div(int a, int b) throw(int) {
  cout << "Div called" << endl;
  Test h("Div");
  b = Sub(b);
  Test h2("Div 2");
  if (b) return a/b;
  cout << "exception inside Div..." << endl;
  throw 0;
}

int main() {
  try {
    Test g("try");
    int c = Div(10, 2);
```

```
    cout << "c = " << c << endl;
  }       // Il controllo ritorna qua
  catch(int exc) {
    cout << "exception catched" << endl;
    cout << "exception value is " << exc << endl;
  }
  return 0;
}
```

La chiamata a ***Div*** all'interno della ***main*** provoca una eccezione nella ***Sub***, viene quindi distrutto l'oggetto ***k*** ed il puntatore ***k2***, ma non l'oggetto puntato (allocato dinamicamente). La deallocazione di oggetti allocati nello heap è a carico del programmatore.
In seguito alla eccezione, il controllo risale a ***Div***, ma la chiamata a ***Sub*** non era racchiusa dentro un blocco **try** e quindi anche ***Div*** viene terminata distruggendo l'oggetto ***h***. L'oggetto ***h2*** non è stato ancora creato e quindi nessun distruttore per esso viene invocato.
Il controllo è ora giunto al blocco che ha chiamato la ***Div***, essendo questo un blocco **try**, vengono distrutti gli oggetti ***g*** e ***c*** ed il controllo passa nel punto in cui si trova il commento.
A questo punto viene eseguita la **catch** poichè il tipo dell'eccezione è lo stesso del suo argomento e quindi il controllo passa alla **return** della ***main***.
Ecco l'output del programma:

```
Test constructor inside try
Div called
Test constructor inside Div
Sub called
Test constructor inside Sub
Test constructor inside Sub 2
exception inside Sub...
Test distructor inside Sub
Test distructor inside Div
Test distructor inside try
exception catched
exception value is 0
```

Si provi a tracciare l'esecuzione del programma e a ricostruirne la storia, il meccanismo diverrà abbastanza chiaro.

Il compito delle istruzioni contenute nel **blocco catch** costituiscono quella parte di azioni di recupero che il programma deve svolgere in caso di errore, cosa esattamente mettere in questo blocco è ovviamente legato alla natura del programma e a ciò che si desidera fare; ad esempio ci potrebbero essere le operazioni per eseguire dell'output su un file di log. È buona norma studiare gli exception handler in modo che al loro interno non possano verificarsi eccezioni.

Nei casi in cui non interessa distinguere tra più tipologie di eccezioni, è possibile utilizzare un unico **blocco catch** utilizzando le ellissi:

```
try {
  /* ... */
}
catch(...) {
  /* ... */
}
```

In altri casi invece potrebbe essere necessari passare l'eccezione ad un blocco

try ancora più esterno, ad esempio perchè a quel livello è sufficiente (o possibile) fare solo certe operazioni, in questo caso basta utilizzare **throw** all'interno del blocco **catch** per reinnescare il meccanismo delle eccezioni a partire da quel punto:

```
try {
  /* ... */
}
catch(Type Id) {
  /* ... */
  throw;    // Bisogna scrivere solo throw
}
```

In questo modo si può portare a conoscenza dei blocchi più esterni della condizione di errore.

Casi particolari

Esistono ancora due problemi da affrontare

- Cosa fare se una funzione solleva una eccezione non specificata tra quelle possibili;
- Cosa fare se non si riesce a trovare una **exception handler** compatibile con l'eccezione sollevata;

Esaminiamo il primo punto.
Per default una funzione che non specifica una lista di possibili tipi di eccezione può sollevare una eccezione di qualsiasi tipo, ma funzione che specifica una lista dei possibili tipi di eccezione è tenuta a rispettare tale lista. Nel caso non lo facesse, in seguito ad una **throw** di tipo non previsto, verrebbe eseguita immediatamente la funzione predefinita ***unexpected()***. Per default ***unexpected()*** chiama ***terminate()*** provocando la terminazione del programma. Questo comportamento può però essere alterato definendo una nuova funzione da sostituire a ***unexpected()***, questa funzione non deve riceve alcun parametro e deve restituisce **void**. La nuova funzione va attivata utilizzando ***set_unexpected()*** come mostrato nel seguente esempio:

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

void MyUnexpected() {
  /* ... */
}

typedef void (* OldUnexpectedPtr) ();

int main() {
  OldUnexpectedPtr = set_unexpected(MyUnexpected);
  /* ... */
  return 0;
}
```

unexpected() e ***set_unexpected()*** sono dichiarate nell'header **< exception >**. È importante ricordare che la vostra ***unexpected*** non deve ritornare, in altre parole deve terminare l'esecuzione del programma:

```
#include < exception >
#include < stdlib.h >
using namespace std;

void MyUnexpected() {
  /* ... */
  abort();     // Termina il programma
}

typedef void (* OldHandlerPtr) ();

int main() {
  OldhandlerPtr = set_unexpected(MyUnexpected);
  /* ... */
  return 0;
}
```

Il modo in cui terminate l'esecuzione non è importante, quello che conta è che la funzione non ritorni.
set_unexpected() infine restituisce l'indirizzo della ***unexpected*** precedentemente installata e che in talune occasioni potrebbe servire.

Rimane da trattare il caso in cui in seguito ad una eccezione, non si trova un handler idoneo...I casi possibili sono due:

- l'eccezione viene lanciata in un blocco di codice tale che risalendo all'indietro non si trova un blocco **try**;
- si trova il blocco **try** ma non si trova una **catch** compatibile.

Il comportamento sempre seguito dal compilatore è quello di risalire all'indietro fino a trovare un blocco **try** con una **catch** compatibile con l'eccezione. Se non si trova la **catch** il compilatore continua a risalire, alla fine o si trova la **catch** oppure si arriva al punto in cui non ci sono più blocchi **try**. Se nessun blocco **try** viene trovato, viene chiamata la funzione ***terminate()***.
Anche in questo caso, come per ***unexpected()***, ***terminate()*** è implementata tramite puntatore ed è possibile alterarne il funzionamento utilizzando ***set_terminate()*** in modo analogo a quanto visto per ***unexpected()*** e ***set_unexpected()*** (ovviamente la nuova ***terminate*** non deve ritornare).
set_terminate() restituisce l'indirizzo della ***terminate()*** precedentemente installata.
Al solito la funzione che sostituisce ***terminate*** non deve ricevere parametri, deve restituire void e deve terminare l'esecuzione del programma.

Eccezioni e costruttori

Il meccanismo di **stack unwinding** (srotolamento dello stack) che si innesca quando viene sollevata una eccezione garantisce che gli oggetti allocati sullo stack vengano distrutti via via che il controllo esce dai vari blocchi applicativi.
Ma cosa succede se l'eccezione viene sollevata nel corso dell'esecuzione di un costruttore? In tal caso l'oggetto non può essere considerato completamente costruito ed il compilatore non esegue la chiamata al suo distruttore, viene comunque eseguita la chiamata dei distruttori per le componenti dell'oggetto che sono state create:

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

```
class Component {
  public:
    Component() {
      cout << "Component constructor called..." << endl;
    }
    ~Component() {
      cout << "Component distructor called..." << endl;
    }
};

class Composed {
  private:
    Component A;

  public:
    Composed() {
      cout << "Composed constructor called..." << endl;
      cout << "Throwing an exception..." << endl;
      throw 10;
    }
    ~Composed() {
      cout << "Composed distructor called..." << endl;
    }
};

int main() {
  try {
    Composed B;
  }
  catch (int) {
    cout << "Exception handled!" << endl;
  };
  return 0;
}
```

Dall'output di questo programma:

```
Component constructor called...
Composed constructor called...
Throwing an exception...
Component distructor called...
Exception handled!
```

è possibile osservare che il distruttore per l'oggetto ***B*** istanza di ***Composed*** non viene eseguito perchè solo al termine del costruttore tale oggetto può essere considerato totalmente realizzato.

Le conseguenze di questo comportamento possono passare inosservate, ma è importante tenere presente che eventuali risorse allocate nel corpo del costruttore non possono essere deallocate dal distruttore. Bisogna realizzare con cura il costruttore assicurandosi che risorse allocate prima dell'eccezione vengano opportunamente deallocate:

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

int Divide(int a, int b) throw(int) {
  if (b) return a/b;
  cout << endl;
```

```
    cout << "Divide: throwing an exception..." << endl;
    cout << endl;
    throw 10;
  }

  class Component {
    public:
      Component() {
        cout << "Component constructor called..." << endl;
      }
      ~Component() {
        cout << "Component distructor called..." << endl;
      }
  };

  class Composed {
    private:
      Component A;
      float* FloatArray;
      int AnInt;
    public:
      Composed() {
        cout << "Composed constructor called..." << endl;
        FloatArray = new float[10];
        try {
          AnInt = Divide(10,0);
        }
        catch(int) {
          cout << "Exception in Composed constructor...";
          cout << endl << "Cleaning up..." << endl;
          delete[] FloatArray;
          cout << "Rethrowing exception..." << endl;
          cout << endl;
          throw;
        }
      }
      ~Composed() {
        cout << "Composed distructor called..." << endl;
        delete[] FloatArray;
      }
  };

  int main() {
    try {
      Composed B;
    }
    catch (int) {
      cout << "main: exception handled!" << endl;
    };
    return 0;
  }
```

All'interno del costruttore di ***Composed*** viene sollevata una eccezione. Quando questo evento si verifica, il costruttore ha già allocato delle risorse (nel nostro caso della memoria); poichè il distruttore non verrebbe eseguito è necessario provvedere alla deallocazione di tale risorsa. Per raggiungere tale scopo, le operazioni soggette a potenziale fallimento vengono racchiuse in una **try** seguita dall'opportuna **catch**. Nel exception handler tale risorsa viene deallocata e l'eccezione viene nuovamente propagata per consentire alla ***main*** di intraprendere ulteriori azioni.
Ecco l'output del programma:

```
Component constructor called...
Composed constructor called...

Divide: throwing an exception...

Exception in Composed constructor...
Cleaning up...
Rethrowing exception...

Component distructor called...
main: exception handled!
```

Si noti che se la **catch** del costruttore della classe ***Composed*** non avesse rilanciato l'eccezione, il compilatore considerando gestita l'eccezione, avrebbe terminato l'esecuzione del costruttore considerando ***B*** completamente costruito. Ciò avrebbe comportato la chiamata del distruttore al termine dell'esecuzione della ***main*** con il conseguente errore dovuto al tentativo di rilasciare nuovamente la memoria allocata per ***FloatArray***.
Per verificare ciò si modifichi il programma nel seguente modo:

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

int Divide(int a, int b) throw(int) {
  if (b) return a/b;
  cout << endl;
  cout << "Divide: throwing an exception..." << endl;
  cout << endl;
  throw 10;
}

class Component {
  public:
    Component() {
      cout << "Component constructor called..." << endl;
    }
    ~Component() {
      cout << "Component distructor called..." << endl;
    }
};

class Composed {
  private:
    Component A;
    float* FloatArray;
    int AnInt;
  public:
    Composed() {
      cout << "Composed constructor called..." << endl;
      FloatArray = new float[10];
      try {
        AnInt = Divide(10,0);
      }
      catch(int) {
        cout << "Exception in Composed constructor...";
        cout << endl << "Cleaning up..." << endl;
        delete[] FloatArray;
      }
    }
    ~Composed() {
```

```
        cout << "Composed distructor called..." << endl;
      }
  };

  int main() {
    try {
      Composed B;
      cout << endl << "main: no exception here!" << endl;
    }
    catch (int) {
      cout << endl << "main: Exception handled!" << endl;
    };
  }
```

eseguendolo otterrete il seguente output:

```
  Component constructor called...
  Composed constructor called...

  Divide: throwing an exception...

  Exception in Composed constructor...
  Cleaning up...

  main: no exception here!
  Composed distructor called...
  Component distructor called...
```

Come si potrà osservare, il **blocco try** della ***main*** viene eseguito normalmente e l'oggetto ***B*** viene distrutto non in seguito all'eccezione, ma solo perchè si esce dallo scope del **blocco try** cui appartiene.

La realizzazione di un costruttore nella cui esecuzione può verificarsi una eccezione, è dunque un compito non banale e in generale sono richieste due operazioni:

1. Eseguire eventuali pulizie all'interno del costruttore se non si è in grado di terminare correttamente la costruzione dell'oggetto;

2. Se il distruttore non termina correttamente (ovvero l'oggetto non viene totalmente costruito), propagare una eccezione anche al codice che ha invocato il costruttore e che altrimenti rischierebbe di utilizzare un oggetto non correttamente creato.

La gerarchia exception

Lo standard prevede tutta una serie di eccezioni, ad esempio l'operatore **::new** può sollevare una eccezione di tipo **bad_alloc**, alcune classi standard (ad esempio la gerarchia degli **iostream**) ne prevedono altre. È stata prevista anche una serie di classi da utilizzare all'interno dei propri programmi, in particolare in fase di debugging.
Alla base della gerarchia si trova la classe ***exception*** da cui derivano ***logic_error*** e ***runtime_error***. La classe base attraverso il metodo virtuale ***what()*** (che restituisce un **char***) è in grado di fornire una descrizione dell'evento (cosa esattamente c'è scritto nell'area puntata dal puntatore restituito dipende dall'implementazione).
Le differenze tra ***logic_error*** e ***runtime_error*** sono sostanzialmente astratte, la prima classe è stata pensata per segnalare errori logici rilevabili attraverso le precondizioni o le invarianti, la classe ***runtime_error*** ha invece lo scopo di riportare errori rilevabili solo a tempo di esecuzione.

Da ***logic_error*** e ***runtime_error*** derivano poi altre classi secondo il seguente schema:

Errore. L'argomento parametro è sconosciuto.

Le varie classi sostanzialmente differiscono solo concettualmente, non ci sono differenze nel loro codice, in questo caso la derivazione è stata utilizzata al solo scopo di sottolineare delle differenze di ruolo forse non immediate da capire.
Dallo standard:

- ***logic_error*** ha lo scopo di segnalare un errore che presumibilmente potrebbe essere rilevato prima di eseguire il programma stesso, come la violazione di una precondizione;
- ***domain_error*** va lanciata per segnalare errori relativi alla violazione del dominio;
- ***invalid_argument*** va utilizzato per segnalare il passaggio di un argomento non valido;
- ***length_error*** segnala il tentativo di costruire un oggetto di dimensioni superiori a quelle permesse;
- ***out_of_range*** riporta un errore di argomento con valore non appartenente all'intervallo di definizione;
- ***runtime_error*** rappresenta un errore che può essere rilevato solo a runtime;
- ***range_error*** segnala un errore di intervallo durante una computazione interna;
- ***overflow_error*** riporta un errore di overflow;
- ***underflow_error*** segnala una condizione di underflow;

Eccetto che per la classe base che non è stata pensata per essere impiegata direttamente, il costruttore di tutte le altre classi riceve come unico argomento un ***const string&***; il tipo ***string*** è definito nella libreria standard del linguaggio.

Conclusioni

I meccanismi che sono stati descritti nei paragrafi precedenti costituiscono un potente mezzo per affrontare e risolvere situazioni altrimenti difficilmente trattabili. Non si tratta comunque di uno strumento facile da capire e utilizzare ed è raccomandabile fare diverse prove ed esercizi per comprendere ciò che sta dietro le quinte. La principale difficoltà è quella di riconoscere i contesti in cui bisogna utilizzare le eccezioni ed ovviamente la strategia da seguire per gestire tali eventi. Ciò che bisogna tener presente è che il meccanismo delle eccezioni è sostanzialmente **non locale** poichè il controllo ritorna indietro risalendo i vari blocchi applicativi. Ciò significa che bisogna pensare ad una strategia globale, ma che non tenti di raggruppare tutte le situazioni in un unico errore generico altrimenti si verrebbe schiacciati dalla complessità del compito.
In generale non è concepibile occuparsi di una possibile eccezione al livello di

ogni singola funzione, a questo livello ciò che è pensabile fare è solo lanciare una eccezione; è invece bene cercare di rendere i propri applicativi molto modulari e isolare e risolvere all'interno di ciascun blocco quante più situazioni di errore possibile, lasciando filtrare una eccezione ai livelli superiori solo se le conseguenze possono avere ripercussioni a quei livelli. Ricordate infine di catturare e trattare le eccezioni standard che si celano dietro ai costrutti predefiniti quali l'operatore globale **::new**.

Appendice A

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 generalmente definite su operandi dello stesso tipo e pertanto non è possibile eseguire immediatamente l'operazione; si rende quindi necessario trasformare gli operandi in modo che assumano un tipo comune su cui è possibile operare. 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 anche dal programmatore, distinguendo quindi tra conversioni implicite e conversioni esplicite. Le prime (dette anche **coercizioni**) sono eseguite dal compilatore in modo del tutto trasparente al programmatore (come nel caso esposto sopra), mentre le seconde sono quelle richieste esplicitamente con una opportuna sintassi.

```
int i = 5;
float f = 0.0;
double d = 1.0;

d = f + i;

d = (double)f + (double)i;
// questa riga si legge: 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, poichè 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 può essere richiesta con la sintassi

(< NuovoTipo >) < Valore >
oppure
< NuovoTipo > (< Valore >)
ma quest'ultimo 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 esplicitamente utilizzate in qualsiasi momento, il compilatore comunque le utilizza implicitamente solo se il tipo di destinazione è compatibile con quello di origine (cioè può rappresentare il valore originale).
Un fattore da tener 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, (in effetti in una conversione **float** a **int** il compilatore non fa altro che scartare la parte frazionaria, se il valore non è rappresentabile il risultato è indefinito).
Da questo punto di vista si può distinguere tra conversione di tipo con perdita di informazione e conversione di tipo senza perdita di informazione. Tra le conversioni senza perdita di informazioni (**safe**) troviamo le conversioni triviali:

DA:	A:
T	T&
T&	T
T[ ]	T*
T(args)	T (*) (args)
T	const T
T	volatile T
T*	const T*
T*	volatile T*

Altre conversioni considerate **safe** sono:

Errore. L'argomento parametro è sconosciuto.

Le conversioni riportate nella figura precedente insieme a quelle triviali sono le uniche ad essere garantite **safe**, alcune implementazioni potrebbero comunque fornire altre conversioni **safe** ma per esse non ci sarebbero garanzie di portabilità.

Le conversioni da e verso un tipo definito dal programmatore richiedono che il compilatore sia informato riguardo a come eseguire l'operazione.
Per convertire un tipo primitivo (**float**, **int**, **unsigned int**...) in un nuovo tipo è necessario che questo nuovo tipo sia una classe (o una struttura) e che sia

definito un costruttore che ha come unico argomento un parametro del tipo primitivo:

```
class Test {
  public:
    Test(int a);
  private:
    float member;
};

Test::Test(int a) {
  member = (float) a;
}
```

Il metodo va naturalmente bene anche quando il tipo di partenza è anch'esso un tipo definito dal programmatore.
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:

```
class Test {
  public:
    Test(int a);
    operator int();

  private:
    float member;
};

Test::operator int() { return (int) member; }
```

Se cioè si desidera poter convertire un tipo utente ***X*** in un tipo primitivo (o anche un altro tipo utente) ***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 implicito nel nome dell'operatore stesso.

C'è un aspetto che bisogna sempre tener presente: quando si definisce un operatore di conversione, questo non necessariamente è disponibile solo al programmatore, ma lo può essere anche al compilatore (se viene dichiarato nella sezione **public** della classe)che potrebbe quindi utilizzarlo senza dare alcun avviso.
Nel caso dei costruttori pubblici il linguaggio fornisce un meccanismo di controllo per impedirne un uso automatico del compilatore:

```
class Test {
  public:
    explicit Test(int a);
    Test(char c);

  private:
    float member;
};

Test::Test(int a): member((float) a) {}

Test::Test(char c): member((float) c) {}
```

```
int main(int, char* []) {
  Test A(5);    // Ok!
  Test B('c');  // Ok!

  A = 7;        // Errore cast implicito non possibile!
  A = Test(7);  // Ok, cast esplicito!
  A = 'b';      // Ok, cast implicito possibile!
  return 0;
}
```

La keyword **explicit** purtroppo è applicabile solo ai costruttori, non e possibile applicarla agli operatori di conversione; come conseguenza di ciò per impedire al compilatore l'uso automatico di un operatore di conversione è necessario renderlo privato o protetto e definire una funzione di forwarding (se siamo interessati a rendere fruibile l'operazione dall'esterno della classe):

```
class Test {
  public:
    explicit Test(int a);
    Test(char c);
    int ToInt();

  private:
    operator int();
    float member;
};

int Test::ToInt() {
  return int();
}
```

Riassumendo è possibile definire in diversi modi una operazione di conversione, in alcuni casi possiamo scegliere tra utilizzare un costruttore, oppure definire un operatore di conversione; in altri casi non abbiamo scelte (tipicamente per i cast verso un tipo primitivo).

La notazione che abbiamo visto sopra per richiedere esplicitamente un cast è derivata direttamente dal C e soffre di alcuni problemi:

- Alcuni cast tipici del C++ sono soggetti a potenziali fallimenti (si pensi ad un cast da classe base a classe derivata) e deve essere possibile gestire tale eventualità;
- I cast sono una violazione del type system, si tratta di operazioni rischiose e solitamente non portabili. La vecchia sintassi non consente una veloce individuazione indispensabile nella manutenzione del software.

Il C++ introduce di conseguenza una nuova sintassi:

```
const_cast < T > (Expr)
static_cast < T > (Expr)
reinterpret_cast < T > (Expr)
dynamic_cast < T* > (Ptr)
```

Nella prima forma (**const_cast**), ***Expr*** deve essere di tipo ***T*** eccetto che per l'uso dei modificatori **const** e/o **volatile**, tale sintassi serve solo a rimuovere (aggiungere) tali modificatori da (a) ***Expr*** in qualunque combinazione.
static_cast è utilizzato per risolvere un qualunque cast (eccetto quelli risolti da **const_cast**), usate questa sintassi quando siete sicuri che l'operazione è correttamente fattibile.
reinterpret_cast è in assoluto il tipo di cast più pericoloso perchè esegue una semplice reinterpretazione dell'argomento che viene visto come una sequenza di bit da mappare sulla base di ***T***.

Infine **dynamic_cast** si usa prevalentemente per eseguire operazioni di downcast (conversione verso classi derivate) quando è possibile il fallimento (in caso contrario potrebbe essere utilizzato **static_cast**). Si noti che l'argomento (***Ptr***) deve essere un puntatore o un riferimento e che **dynamic_cast** restituisce un puntatore (vedi sintassi) o in alternativa un riferimento. L'operazione di downcast può essere eseguita solo se la classe base è polimorfica (ha cioè metodi virtuali), questa operazione richiede il **RTTI** ed è eseguita a run time. In caso di fallimento di un downcast, viene sollevata una eccezione (**bad_cast**) per i cast a riferimento, altrimenti (conversione verso puntatore) viene restituito il puntatore nullo.
dynamic_cast può comunque essere utilizzato anche per eseguire upcast (cast verso classe base), in tal caso l'operazione viene risolta a compile time.

Eccone alcuni esempi d'uso della nuova sintassi:

```
// Downcast (risolto a run time):
Persona* Caio = new Studente(/*...*/;
Studente* Pippo = dynamic_cast < Studente* > (Caio);

// rimozione di const:
const long ConstObj = 10;
long* LongPtr = const_cast < long* > ( & ConstObj );

// cast bruto:
int* Ptr = new int(7);
double* DPtr = reinterpret_cast < double* > (Ptr);

// cast risolto a compile time:
Caio = static_cast < Persona* > (Pippo);
```

L'operazione di downcast (il primo cast dell'esempio) viene risolta a run time, il compilatore genera codice per verificare la fattibilità dell'operazione e se fattibile procede alla conversione (chiamando l'apposito operatore), altrimenti verrebbe restituito il puntatore nullo.
Il secondo esempio mostra come eliminare il **const**: viene calcolato l'indirizzo dell'oggetto costante (tipo **const long***) e quest'ultimo viene poi convertito in **long***.
Il terzo esempio mostra invece un tipico cast in cui semplicemente si vuole interpretare una sequenza di bit secondo un nuovo significato, nel caso in esame un **int*** viene interpretato come se fosse un **double***. Questo genere di conversione è tipicamente dipendente dall'implementazione adottata.
Infine l'ultimo esempio mostra come risolvere a run time un cast verso classe base a partire da una classe derivata (operazione che sappiamo essere sicura).

Si noti che quella vista è solo una sintassi, l'operazione di cast effettiva viene svolta richiamando gli appositi operatori che devono quindi essere definiti; ad esempio:

```
Studente Sempronio(/* ... */
Persona Ciccio = static_cast < Persona > (Sempronio);

int Integer = 5;
double Real = static_cast < double > (Integer);

Integer = static_cast < Persona > (Ciccio);
```

I primi due cast possono essere risolti perchè nel primo caso ***Studente*** è un sottotipo di ***Persona*** e l'operatore di conversione è implicitamente definito; nel secondo caso l'operatore invece è già definito dal linguaggio. L'ultimo esempio invece genera un errore se la classe ***Persona*** non definisce un operatore di

conversione a **int**.

Appendice B

Introduzione alla OOP

Nel corso degli anni sono stati proposti diversi paradigmi di programmazione, ovvero diversi modi di vedere e modellare la realtà (paradigma imperativo, funzionale, logico...).
Obiettivo comune di tutti era la risoluzione dei problemi legati alla manutenzione e al reimpiego di codice . Ciascun paradigma ha poi avuto un impatto differente dagli altri, con conseguenze anch'esse diverse. Assunzioni apparentemente corrette, si sono rivelate dei veri boomerang, basti pensare alla crisi del software avutasi tra la fine degli anni '60 e l'inizio degli anni '70. In verità comunque la colpa dei fallimenti non era in generale dovuta solo al paradigma, ma spesso erano le cattive abitudini del programmatore, favorite dalla implementazione del linguaggio, ad essere la vera causa dei problemi. L'evoluzione dei linguaggi e la nascita e lo sviluppo di nuovi paradigmi mira dunque a eliminare le cause dei problemi e a guidare il programmatore verso un modo "ideale" di vedere e concepire le cose impedendo (per quanto possibile e sempre relativamente al linguaggio) "cattivi comportamenti".

Di tutti i paradigmi proposti, uno di quelli più attuali e su cui si basano linguaggi nuovissimi come Java o Eiffel (e linguaggi derivati da altri come l'Object Pascal di Delphi e lo stesso C++), è sicuramente il paradigma **object oriented**.
Ad essere precisi quello object oriented non è un vero e proprio paradigma, ma un metaparadigma. La differenza sta nel fatto che un paradigma definisce un modello di computazione (ad esempio quello funzionale modella un programma come una funzione matematica), mentre un metaparadigma generalmente si limita a imporre una visione del mondo reale non legata ad un modello computazionale. Di fatto esistono implementazioni del metaparadigma object oriented basate sul modello imperativo (C++, Object Pascal, Java) o su modelli funzionali (CLOS ovvero la versione object oriented del Lisp).
Nel seguito, parleremo di paradigma ad oggetti (anche se il termine è improprio) e faremo riferimento sostanzialmente al modello fornito dal C++; ma sia chiaro fin d'ora che non esiste un unico modello object oriented e non esiste neanche una terminologia universalmente accettata.

Il paradigma ad oggetti tende a modellare una certa situazione (realtà) tramite un insieme di entità attive (che cioè svolgono azioni) più o meno indipendenti l'una dall'altra, con funzioni generalmente differenti, ma cooperanti per l'espletamento di un compito complessivo. Tipico esempio potrebbe essere rappresentato dal modello **doc/view** in cui un editor viene visto come costituito più o meno da una coppia: un gestore di documenti il cui compito è occuparsi di tutto ciò che attiene all'accesso ai dati e ad eseguire le varie possibili operazioni su di essi, ed un modulo preposto alla visualizzazione dei dati ed alla interazione con chi usa tali dati (mediando così tra utente e gestore dei documenti).
Possiamo tentare un parallelo tra gli oggetti della OOP (Object Oriented Programming) e le persone che lavorano in una certa industria... ci saranno diverse tipologie di addetti ai lavori con mansioni diverse: operai più o meno specializzati in certi compiti, capi reparto, responsabili e dirigenti ai vari livelli. Svalgono tutti compiti diversi, ma insieme lavorano per realizzare

certi prodotti ognuno occupandosi di problemi diversi direttamente connessi alla produzione, altri col compito di coordinare le attività (interazioni).
Comunque sia chiaro che gli oggetti della OOP sono in generale diversi da quelli del mondo reale (siano esse persone, animali o cose).

Le entità attive della OOP (Object Oriented Programming) sono dette oggetti. Un oggetto è una entità software dotata di stato, comportamento e identità. Lo stato viene generalmente modellato tramite un insieme di attributi (contenitori di valori), il comportamento è costituito dalle azioni (metodi) che l'oggetto può compiere e infine l'identità è unica, immutabile e indipendente dallo stato, può essere pensata in prima approssimazione come all'indirizzo fisico di memoria in cui l'oggetto si trova (in realtà è improprio identificare identità e indirizzo, perchè generalmente l'indirizzo dell'oggetto e l'indirizzo del suo stato, mentre altre informazioni e caratteristiche dell'oggetto generalmente stanno altrove).
Vediamo come tutto questo si traduca in C++:

```
class TObject {
  public:
    void Foo();
    long double Foo2(int i);

  private:
    const int f;
    float g;
};
```

In questo esempio lo stato è modellato dalle variabili ***f, g*** . Il comportamento è invece modellato dalle funzioni ***Foo()*** e ***Foo2(int)***.

Gli oggetti cooperano tra loro scambiandosi messaggi (richieste per certe operazioni e risposte alle richieste). Ad esempio un certo oggetto **A** può occuparsi di ricevere ordini relativi all'esecuzione di certe operazioni aritmetiche su certi dati, per l'espletamento di tale compito può affidarsi ad un altro oggetto **Calcolatrice** fornendo il tipo dell'operazione da realizzare e gli operandi; l'oggetto **Calcolatrice** a sua volta può smistare le varie richieste a oggetti specializzati per le moltiplicazioni o le addizioni.
L'insieme dei messaggi cui un oggetto risponde è detto **interfaccia** ed il meccanismo utilizzato per inviare messaggi e ricevere risposte è quello della chiamata di procedura; nell'esempio di prima, l'interfaccia è data dai metodi ***void Foo()*** e ***long double Foo2(int)***.

Ogni oggetto è caratterizzato da un **tipo**; un tipo in generale è una definizione astratta (un modello) per un generico oggetto. Non esiste accordo su cosa debba essere un tipo, ma in generale e accettata l'idea secondo cui un tipo debba definire almeno l'interfaccia di un oggetto.
In C++ il tipo di un generico oggetto si definisce tramite la realizzazione di una **classe**. Una classe (termine impropriamente utilizzato dal C++ come sinonimo di tipo) in C++ non definisce solo l'interfaccia di un oggetto, ma anche la struttura del suo stato (vedi esempio precedente) e l'insieme dei valori ammissibili.
Ogni tipo (non solo in C++) deve inoltre fornire dei metodi speciali il cui compito è quello di occuparsi della corretta costruzione e inizializzazione delle singole istanze (**costruttori**) e della loro distruzione quando esse non servono più (**distruttori**).

Quando lo stato di un oggetto non è direttamente accessibile dall'esterno, si dice che l'oggetto incapsula lo stato, taluni linguaggi (come il C++) non costringono a incapsulare lo stato, in questi casi gli attributi accessibili dall'esterno divengono parte dell'interfaccia.
L'incapsulamento ha diverse importanti conseguenze, in particolare forza il

programmatore a pensare e realizzare codice in modo tale che gli oggetti siano in sostanza delle **unità di elaborazione** che ricevono dati in input (i messaggi) e generano altri messaggi (generalmente diretti ad altri oggetti) in output che rappresentano il risultato della loro elaborazione. In tal modo un applicativo assume la forma di un insieme di oggetti che comunicando tra loro risolvono un certo problema.

Altro punto fondamentale del paradigma ad oggetti è l'esplicita presenza di strumenti atti a conseguire un facile reimpiego di codice precedentemente prodotto. L'obiettivo può essere raggiunto in diversi modi, ciascuna modalità è spesso legata a caratteristiche intrinseche di un certo modello di programmazione object oriented. In particolare attualmente le metodologie su cui si discute sono:

1. **Reimpiego per composizione**, distinguendo tra
   - **contenimento diretto**
   - **contenimento indiretto**
2. **Reimpiego per ereditarietà**, distinguendo tra:
   - **ereditarietà di interfaccia**
   - **ereditarietà di implementazione**
3. **Delegation**

ciascuna con i suoi vantaggi e suoi svantaggi.
Nel reimpiego per composizione, quando si desidera estendere o specializzare le caratteristiche di un oggetto, si crea un nuovo tipo che contiene al suo interno una istanza del tipo di partenza (o in generale più oggetti di tipi anche diversi tra loro). L'oggetto composto fornisce alcune o tutte le funzionalità della sua componente facendo da tramite tra questa e il mondo esterno, mentre le nuove funzionalità sono implementate per mezzo di metodi e attributi propri dell'oggetto composto.
Un oggetto composto può contenere l'oggetto (e in generale gli oggetti) più piccolo direttamente (ovvero tramite un attributo del tipo dell'oggetto contenuto) oppure tramite puntatori (contenimento indiretto):

```
class Lavoro {
  public:
    Lavoro(/* Parametri */);

    /* ... */

  private:
    /* ... */
};

class Lavoratore {
  public:
    /* ... */

  private:
    Lavoro Occupazione;   // contenimento diretto
    /* ... */
};

class LavoratoreAlternativo {
  public:
    /* ... */
```

```
  private:
    Lavoro* Occupazione;   // contenimento indiretto
    /* ... */
};
```

Il contenimento diretto è in generale più efficiente per diversi motivi:

- Non si passa attraverso puntatori ogni qual volta si debba accedere alla componente;
- Nessuna operazione di allocazione o deallocazione da gestire e semplificazione di problematiche legate alla corretta creazione e distruzione delle istanze;
- Il tipo della componente è completamente noto e sono possibili tutta una serie di ottimizzazioni altrimenti non fattibili.

Il contenimento per puntatori per contro ha i seguenti vantaggi:

- La costruzione di un oggetto composto può avvenire per gradi, costruendo le sottocomponenti in tempi diversi;
- Una componente può essere condivisa da più oggetti;
- Come vedremo utilizzando puntatori possiamo riferire a tutto un insieme di tipi per quella componente, ed utilizzare di volta in volta il tipo che più ci fa comodo (anche cambiando a run time la componente stessa);
- In linguaggi come il C++ in cui un puntatore è molto simile ad un array, possiamo realizzare relazioni in cui un oggetto può avere da *0* a *n* componenti, con *n* determinabile a run time (la composizione diretta richiederebbe di fissare il valore massimo per *n*).

Concettualmente la composizione permette di modellare facilmente una relazione ***Has-a*** in cui un oggetto più grande **possiede** uno o più oggetti tramite i quali espleta determinate funzioni (il caso dell'esempio del ***Lavoratore*** che possiede un ***Lavoro***). Tuttavia è anche possibili simulare una relazione di tipo ***Is-a***:

```
class Persona {
  public:
    void Presentati();

  /* ... */
};

class Lavoratore {
  public:
    void Presentati();
    /* ... */
  private:
    Persona Io;
    char* DatoreLavoro;
    /* ... */
};

void Lavoratore::Presentati() {
  Io.Presentati();
  cout << "Impiegato presso " << DatoreLavoro << endl;
}
```

Molte tecnologie ad oggetti (ma non tutte) forniscono un altro meccanismo per il

reimpiego di codice: l'**ereditarietà**.
L'idea di base è quella di fornire uno strumento che permetta di dire che un certo tipo (detto **sottotipo** o tipo derivato) risponde agli stessi messaggi di un altro (**supertipo** o tipo base) più un insieme (eventualmente vuoto) di nuovi messaggi.
Quando si eredita solo l'interfaccia di un tipo (ma non la sua implementazione, ne l'implementazione dello stato e/o di altre caratteristiche del supertipo) si parla di **ereditarietà di interfaccia**:

```
class Interface {
  public:
    void Foo();
    double Sum(int a, double b);
};

class Derived: public Interface {
  public:
    void Foo();
    double Sum(int a, double b);
    void Foo2();
};

void Derived::Foo() {
  /* ... */
}

double Derived::Sum(int a, double b) {
  /* ... */
}

void Derived::Foo2() {
  /* ... */
}
```

Si noti che quando si è in presenza di ereditarietà di interfaccia, la classe derivata ha l'obbligo di implementare tutto ciò che eredita (a meno che non si voglia derivare una nuova interfaccia), poichè l'unica cosa che si eredita è un insieme di nomi (identificatori di messaggi) cui non è associata alcuna gestione. Infine (almeno in C++) per (ri)definire un metodo dichiarato in una classe base, la classe derivata deve ripetere la dichiarazione (ma ciò potrebbe non essere vero in altri linguaggi).
Alcuni modelli di OOP consentono l'**ereditarietà dell'implementazione** (es. il C++), che può essere vista come caso generale in cui si eredita tutto ciò che definiva il supertipo; dunque non solo l'interfaccia ma anche la gestione dei messaggi che la costituiscono e pure l'implementazione dello stato del supertipo. Il vantaggio dell'ereditarietà di implementazione viene fuori in quelle situazioni in cui il sottotipo esegue sostanzialmente gli stessi compiti del supertipo allo stesso modo (cambiano al più poche cose). Qualora il sottotipo dovesse gestire un messaggio in modo differente, viene comunque data la possibilità di ridefinirne la politica di gestione:

```
class Base {
  public:
    void Foo() { return; }
    double Sum(int a, double b) { return a+b; }
    /* ... */
  private:
    /* ... */
};
```

```
class Derived: public Base {
  public:
    void Foo();
    double Sum(int a, double b);
    void Foo2();
};

void Derived::Foo() {
  Base::Foo();
  /* ... */
}

void Derived::Foo2() {
  /* ... */
}
```

Nell'esempio appena visto la classe ***Derived*** eredita da ***Base*** tutto ciò che a quest'ultima apparteneva (interfaccia, stato, implementazione dell'interfaccia); ***Derived*** aggiunge nuove funzionalità (***Foo2()***) e ridefinisce alcune di quelle ereditate (ridefinizione di ***Foo()***), mentre altre funzionalità vanno bene così come sono (***Sum()***) e dunque la classe non le ridefinisce.
In alcuni sistemi potrebbe essere fornita la sola ereditarietà di interfaccia, così che le sole possibilità sono ereditare da una interfaccia per definire una nuova interfaccia, oppure utilizzare le interfacce per definire le operazioni che possono essere compiute su un certo oggetto (in questo caso si definisce la struttura di un certo insieme di oggetti dicendo che essi rispondono a quella interfaccia utilizzando una certa implementazione).
L'ereditarietà modella in generale una relazione di tipo ***Is-a*** poichè un sottotipo rispondendo ai messaggi del supertipo potrebbe essere utilizzato in sostituzione di quest'ultimo. La sostituzione di un supertipo con un sottotipo comunque non è di per se garantita dalla ereditarietà, perchè ciò avvenga deve valere il **principio di sostituibilità di Liskov**.
Tale principio afferma che la sostituibilità è legata non (solo) all'interfaccia dell'oggetto, ma al comportamento; nulla infatti vieta in molti linguaggi OO (C++ compreso) di fare in modo che un sottotipo risponda ad un messaggio con un comportamento non coerente a quello del supertipo (ad esempio il metodo ***Presentati()*** del tipo ***Lavoratore*** potrebbe fare qualcosa totalmente diversa dalla versione del tipo ***Persona*** come visualizzare il risultato di una somma).
È anche possibile utilizzare l'ereditarietà per modellare relazioni ***Has-a***, ma si tratta spesso (praticamente sempre) di un grave errore e quindi tale caso non verrà preso in esame poichè un linguaggio (o una tecnologia) OO fornisce sempre almeno il contenimento (o una qualche sua espressione).

Infine la ***delegation*** è un meccanismo che tenta di mediare composizione e ereditarietà. L'idea di base è quella di consentire ad un oggetto di delegare dinamicamente certi compiti a altri oggetti.
Esprimere tale possibilità in C++ non è semplice, perchè dovremmo ricorrere comunque al contenimento per implementare tale meccanismo:

```
class TRectangle {
  public:
    int GetArea();
  /* ... */
};

class TSquare {
  public:
    int GetArea() {
      return RectanglePtr -> GetArea();
    }
  private:
```

```
    TRectangle* RectanglePtr;
};
```

Tuttavia in un linguaggio con ***delegation*** potrebbero essere forniti strumenti opportuni per gestire dinamicamente problemi di delega e probabilmente essere soggetti a vincoli di natura diversa da quelli imposti dal C++.

In presenza di ereditarietà (sia essa di interfaccia che di implementazione), viene spesso fornito un meccanismo che permette di lavorare uniformemente con tutta una gerarchia di classi astraendo dai dettagli specifici della generica classe e sfruttando solo una interfaccia comune (quella della classe base da cui deriva la gerarchia). Tale meccanismo viene indicato con il termine **polimorfismo** e viene implementato fornendo un meccanismo di **late binding**, ovvero ritardando a tempo di esecuzione il collegamento tra un generico oggetto della gerarchia e i suoi membri.