Paolo Marotta

C++

una panoramica sul linguaggio

Seconda edizione

Prefazione

18 / 7 / 1999

Ho iniziato a scrivere questo tutorial circa tre anni fa, spinto da diversi fattori; da una parte il mio entusiasmo verso questo potente linguaggio, dall'altra il desiderio di rendere disponibile una valida risorsa ai programmatori italiani che desiderano iniziare a impararne le potenzialita`. Inoltre non esistono ancora in rete molte risorse in lingua italiana e spesso i neofiti hanno difficolta` con l'inglese (che resta comunque bagaglio culturale indispensabile a chi ha scelto di dedicarsi alla programmazione e all'informatica).
Nel corso di questi tre anni il C++ si e` evoluto e ha subito un profondo processo di standardizzazione; le modifiche e le aggiunte apportate dallo standard sono state notevoli e mi hanno indotto ad eseguire un'opera di revisione che ha portato a questa seconda edizione.
Poiche` i miei impegni non mi lasciano molto tempo libero, ho deciso di rendere disponibile questo materiale via via che le varie parti sono pronte (in effetti mancano ancora capitoli su RTTI, I/O su stream, e un capitolo sulla libreria standard). In particolare le cose aggiunte per ultime potranno subire ulteriori rimaneggiamenti e revisioni.
Nel frattempo se dovreste accorgervi di errori o mancanze in cio` che segue vi prego di segnalarmeli all'indirizzo e-mail Marotta@CLI.DI.UniPi.It, aiuterete me e voi stessi, oltre che tutte le persone desiderose di imparare. Naturalmente non manchero` di segnalare i nominativi di quanti hanno contribuito in un apposito spazio.

Note per la distribuzione

Questo materiale viene reso disponibile gratuitamente e puo` essere distribuito sotto qualsiasi forma (anche cartacea) alle seguenti condizioni:

- il materiale dovra` essere distribuito senza alcuna modifica ai suoi contenuti (salvo esplicita autorizzazione);
- niente sara` dovuto da chi lo acquisisce a colui che lo distribuisce, salvo rimborso di eventuali spese per il supporto che lo contiene;

E` possibile contattare l'autore all'indirizzo e-mail Marotta@CLI.DI.UniPi.It, oppure al seguente indirizzo:

Paolo Marotta
Via F.lli Bandiera, 115
97019 Vittoria (RG)

Ringraziamenti

Questa pagina e` dedicata a coloro che hanno contribuito alla stesura di questo materiale con il loro lavoro di revisione, in particolare **Antonio Bonifati** (http://Assembly_Page_di_Ant.freeweb.org) che con i suoi bugs report mi ha permesso di correggere numerosi errori scappatti nelle precedenti revisioni.

Paolo Marotta

Nota dell'impaginatore :)
(itaprogaming@multiplayer.it)

Ciao! Sono Francesco e questa che state leggendo è la versione riveduta e corretta (nella forma, non nei contenuti) del corso su Delphi di Vanni Brutto. Per farla ho chiesto e ottenuto l'autorizzazione di Vanni. Se volete mettere questi files sul vostro sito, gradirei che me lo comunicaste e che metteste anche un link al mio sito e a quello di Vanni, dato che ci è costato tempo e fatica ;).
L'indirizzo del mio sito è http://www.multiplayer.it/itaprogaming, mentre quello di Vanni lo trovate un po' più su. Ciao e grazie per la collaborazione!"

Nota per la versione PDF

La versione PDF è stata redatta da Stefano SteO Arcidiacono (arcidiacono@infomedia.it), scaricabile da #GameProg-Ita, http://www.gpi.infomedia.it

Introduzione

Il C++ e` un linguaggio di programmazione "all purpose", ovvero adatto alla realizzazione di qualsiasi tipo di applicazione, da quelle real time a quelle che operano su basi di dati, da applicazioni per utenti finali a sistemi operativi. Il fatto che sia un linguaggio "all purpose" non implica comunque dire 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 migliore poiche` consentono di ottenere un rapporto "costi di produzione/prestazioni" migliore per motivi che saranno chiari tra poche righe.
Negli ultimi anni il C++ ha ottenuto un notevole successo per diversi motivi:

- Conserva una compatibilita` quasi assoluta (alcune cose sono diverse) con il suo piu` diretto antenato, il C, da cui eredita la sintassi e la semantica per tutti i costrutti comuni, oltre alla notevole flessibilita` 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 (che manca soprattutto di un buon sistema dei tipi). In particolare l'introduzione di costrutti quali i *Template* e *le Classi* rende il C++ un linguaggio multiparadigma (con particolare predilezione per il paradigma ad oggetti e la programmazione generica);

- Possibilita` 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 flessibilita` tipiche del C e del C++ non sono gratuite. Se da una parte e` vero che e` possibile ottenere applicazioni in generale piu` efficienti (rispetto ad agli altri linguaggi), e anche vero che tutto questo e` ottenuto lasciando in mano al programmatore molti dettagli e compiti che negli altri linguaggi sono svolti dal compilatore; e` 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 e` 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 e` 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 del linguaggio (classi, template, eccezioni...); alla fine analizzeremo (almeno in parte) l'input/output tramite stream e la libreria standard del linguaggio.
Il corso è rivolto a persone che non hanno alcuna conoscenza del linguaggio, ma potrà tornare utile anche a programmatori che possiedono una certa familiarità con esso. 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.

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 e` quello di tradurre il programma originale (codice sorgente) in uno semanticamente equivalente, ma eseguibile su una certa macchina. Il processo di compilazione e` suddiviso in piu` fasi, ciascuna delle quali volta all'acquisizione di opportune informazioni necessarie alla fase successiva.
La prima di queste fasi e` 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 piu` in dettaglio.

Commenti

I commenti, come in qualsiasi altro linguaggio, hanno valore soltanto per il programmatore e vengono ignorati dal compilatore. E` 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 e` considerato commento tutto quello che e` compreso tra /* e */, il commento quindi si puo` estendere anche su piu` righe o trovarsi in mezzo al codice:

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

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

```
void Func() {
  ...
  int a = 5;  // questo e` un commento valido
  a = 4       // Errore! il "+ 5;" e` commento  + 5;
              e non e` possibile dividerlo
              su piu` righe
```

```
    ...
}
```

Benche` esistano due distinti metodi per commentare il codice, non e` 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 cio` che segue "a + c" viene interpretato come un commento iniziato da //, e` 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 puo` contenere un singolo valore che puo` cambiare nel tempo, il tipo di questo valore viene comunque stabilito una volta per tutte e non puo` 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 e` un nome il cui compito e` 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 e` 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 e` 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 e` 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.
Benche` il linguaggio non preveda un limite alla lunghezza massima di un identificatore, e` 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 e` 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 e` possibile inserire direttamente dei valori, questi valori sono detti costanti letterali. La generica costante letterale puo` 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 puo` 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, e` consentito l'uso delle lettere da "A" a "F" e da "a" a "f".
Si noti che un segno puo` essere espresso solo in base 10, negli altri casi esso e` 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 scegliera` il tipo (Vedi tipi di dato) da utilizzare sulla base delle seguenti regole:

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

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

Si puo` 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 piu` 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 e` 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.
E` possibile omettere uno tra l'intero decimale e la frazione decimale, ma non entrambi. E` 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 e` 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:

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

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 analizzeremo solo le istruzioni per il controllo del flusso e l'assegnamento, le rimanenti (poche) istruzioni verranno discusse via via che sarà necessario nei prossimi capitoli.

Assegnamento

Il C++ e` un linguaggio pesantemente basato sul paradigma imperativo, questo vuol dire che un programma C++ e` sostanzialmente una sequenza di assegnamenti di valori a variabili. E` quindi naturale iniziare parlando proprio dell'assegnamento.
L'operatore di assegnamento e` 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 e` 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 e` 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). Cio` 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 produrra` tale valore fino a che un nuovo assegnamento non verra` eseguito su di essa.
Si osservi che una variabile puo` 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, poiche` un identificatore di variabile puo` trovarsi contemporaneamente su ambo i lati di un assegnamento e` 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.

Poiche` un assegnamento produce come risultato il valore prodotto dalla valutazione della parte destra (e` cioe` a sua volta una espressione), e` possibile legare in cascata piu` assegnamenti:

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

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

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

cioe` 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 cioe` 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++;          // cioe` 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 e` proprio identico poiche` 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'e` 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 piu` 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 piu` grande non e` definito, ad esempio nell'espressione

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

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

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

L'operatore ? : e` l'unico operatore ternario:

```
<Cond> ? <Expr1> : <Expr2>
```

La semantica di questo operatore non e` molto complicata: Cond puo` essere una qualunque espressione che produca un valore booleano (Vedi paragrafo successivo), se essa e` verificata il risultato di tale operatore e` la valutazione di Expr1, altrimenti il risultato e` Expr2.
Per quanto riguarda gli altri operatori, alcuni saranno esaminati quando sara` 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 verita` venivano rappresentati tramite valori interi: 0 (zero) indicava falso e un valore diverso da 0 indicava vero. Cio` 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 verita` 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 verita`. Come conseguenza di cio`, la` 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 perche` i costrutti del linguaggio sono stati adattati di conseguenza. Comunque sia per compatibilita` con il C ed il codice C++ precedentemente prodotto e` 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;   // modalita`, 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 e` 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 e` vera viene eseguita Istruzione1, altrimenti nel primo caso non viene eseguito alcunche`, nel secondo caso invece si esegue Istruzione2.

Si osservi che Istruzione1 e Istruzione2 sono istruzioni singole (una sola istruzione), se e` 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 e` data

dalle istruzioni composte (cioe` sequenze di istruzioni racchiuse tra parentesi graffe) che non devono essere seguite dal punto e virgola (non serve, c'e` la parentesi graffa).

Per risolvere eventuali ambiguita` 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 cioe` 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 verita` di una condizione.
Vediamone la sintassi:

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

Al solito, Istruzione indica una istruzione singola, se e` necessaria una sequenza di istruzioni essa deve essere racchiusa tra parentesi graffe.
La semantica del while e` la seguente: prima si valuta Condizione e se essa e` 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 e` 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 e` 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 piu` esperti sapranno, il ciclo for e` una specializzazione del while, tuttavia nel C++ la differenza tra for e while e` talmente sottile che i due costrutti possono essere liberamente scambiati tra loro.
La sintassi del for e` la seguente:

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

Inizializzazione puo` 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 e` una qualsiasi espressione booleana; e Iterazione e` 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 puo` 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 puo` 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 e` molto simile al case del Pascal (anche se piu` potente) e consente l'esecuzione di uno o piu` frammenti di codice a seconda del valore di una espressione:

```
switch ( <Espressione> ) {
```

```
  case <Valore1> : <Istruzione> ;
  /* ... */
  case <ValoreN> : <Istruzione> ;
  default : <Istruzione> ;
 }
```

Espressione e` una qualunque espressione capace di produrre un valore intero; Valore1...ValoreN sono costanti a valori interi; Istruzione e` 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 e` 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, cioe` il controllo passa da una clausola case alla successiva (default compreso) anche quando la clausola viene eseguita. Per evitare cio` e` 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 e` 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 puo` essere utilizzata anche prima di essere dichiarata. Esiste una limitazione all'uso del goto: il bersaglio dell'istruzione (cioe` Etichetta) deve trovarsi all'interno della stessa funzione dove appare l'istruzione di salto.

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 necessita` di una dichiarazione; nel caso di variabili, costanti o tipi:

- consente di stabilire la quantita` 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 puo` 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 e` una coppia < V, O >, dove V e` un insieme di valori e O e` 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 verita`; 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 e` 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 e` utilizzato per rappresentare interi in un intervallo piu` 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 e` 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 e` per default signed e quindi e` 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 quantita` di memoria richiesta per rappresentare un valore di quel tipo (che pero` puo` 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 e` 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 e` 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 e` il nome che identifica la funzione principale, ovvero il corpo del programma, parleremo in seguito e piu` in dettaglio di main().

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

Variabili e costanti

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

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

Ad esempio:

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

Innanzi tutto ricordo che il C++ e` case sensitive, cioe` distingue le lettere maiuscole da quelle minuscole, infine si noti il punto e virgola che segue sempre ogni dichiarazione.
La prima riga dichiara quattro variabili di tipo int, mentre la seconda una di tipo signed char. La terza dichiarazione e` un po' particolare in quanto apparentemente manca la keyword int, in realta` poiche` il default e` proprio int essa puo` essere omessa; in conclusione la terza dichiarazione introduce una variabile di tipo unsigned short int. Gli identificatori che seguono il tipo sono i nomi delle variabili, se piu` di un nome viene specificato essi devono essere separati da una virgola.

E` 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 e` 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 e` 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 e` inserita (un blocco di istruzioni e` racchiuso sempre tra una coppia di parentesi graffe). Cio` vuol dire che quella dichiarazione non e` visibile all'esterno di quel blocco, mentre e` visibile in eventuali blocchi annidati dentro quello dove la variabile e` dichiarata.
Il seguente schema chiarisce la situazione:

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

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

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

All'uscita dal blocco piu` 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 puo` 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 :: e` 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 e` posta la dichiarazione; fanno eccezione le variabili globali che vengono distrutte alla fine dell'esecuzione del programma. Da cio` 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; e`

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 e` 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 e` legato a computazioni precedenti (all'interno della stessa esecuzione del programma) oppure per ragioni di efficenza. Infine la keyword static non modifica lo scope.

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 visibilita` 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 puo` contenere un solo valore alla volta; il costruttore di array [ ] permette di raccogliere sotto un solo nome piu` 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 e` array di 10 int(eri).
La sintassi per la generica dichiarazione di un array e`

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

Al solito Tipo puo` essere sia un tipo primitivo che uno definito dal programmatore, Identificatore e` 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 puo` essere una qualsiasi espressione che produca un valore intero; il primo elemento di un array e` sempre Identificatore[0], e di conseguenza l'ultimo e` 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
```

E` anche possibile dichiarare array multidimensionali (detti array di array o piu` in generale matrici) specificando piu` 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];
```

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

```
int Pippo[5]       = { 10, -5, 6, 110, -96 };
short Pluto[2][4]  = { {4, 7, 1, 4}, {0, 3, 5, 9} };

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 e` 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 e` l'ultimo indice cosi` 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 piu` complesse in quanto non vengono specificati tutti gli indici degli array: in caso di inizializzazione aggregata il compilatore e` 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, cosi` che la terza dichiararzione introduce un array di 3 elementi e l'ultima una matrice 2 x 3. E` 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 e` pari a quella della stringa "investigatore" + 1, l'elemento in piu` e` 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 e` 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!
```

E` 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 pero` non si ottiene una stringa terminata da '\0', ma semplici array di caratteri il cui numero di elementi e` esattamente quello specificato.

Strutture

Gli array permettono di raccogliere sotto un unico nome piu` variabili omogenee e sono solitamente utilizzati quando bisogna operare su piu` valori dello stesso tipo contemporaneamente (ad esempio per eseguire una ricerca). Tuttavia in generale per rappresentare entita` complesse e` necessario memorizzare informazioni di diversa natura; ad esempio per rappresentare una persona puo` non bastare una stringa per il nome ed il cognome, ma potrebbe essere necessario memorizzare anche eta` e codice fiscale. Memorizzare tutte queste informazioni in un'unica stringa non e` una buona idea poiche` le singole informazioni non sono immediatamente disponibili, ma e` necessario prima estrarle, inoltre nella rappresentazione verrebbero perse informazioni preziose quali il fatto che l'eta` e` sempre data da un intero positivo. D'altra parte avere variabili distinte per le singole informazioni non e` certamente una buona pratica, diventa difficile capire qual'e` la relazione tra le varie componenti. La soluzione consiste nel raccogliere le variabili che modellano i singoli aspetti in un'unica entita` 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 e`

```
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 e` 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, e` 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;
```

Cio` e` possibile per le strutture solo perche`, come vedremo, il compilatore le tratta come classi i cui membri sono tutti pubblici.
L'assegnamento e` ovviamente possibile solo tra variabili dello stesso tipo struttura, ma quello che di solito sfugge e` 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 e` che nella dichiarazione di una unione viene utilizzata la keyword union anzicche` 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 e` 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. Cio` vuol dire che, mentre la quantita` di memoria occupata da una struttura e` data dalla somma delle quantita` di memoria utilizzata dalle singole componenti, la quantita` di memoria utilizzata da una unione e` data da quella della componente piu` 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'e` tuttavia un altro possibile utilizzo delle unioni, eseguire "manualmente" alcune conversioni di tipo. Tuttavia tale pratica e` assolutamente da evitare (almeno quando esiste una alternativa) poiche` tali conversioni sono dipendenti dall'architettura su cui si opera e pertanto non portabili, ma anche potenzialmete scorrette.

Enumerazioni

A volte puo` essere utile poter definire un nuovo tipo estensionalmente, cioe` elencando esplicitamente i valori che una variabile (o una costante) di quel tipo puo` 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 ambiguita`.
Il compilatore rappresenta internamente i tipi enumerazione associando a ciascun identificatore di valore una costante intera, cosi` che un valore enumerazione puo` 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 perche` non esiste un operatore di conversione da int a Elemento, mentre essendo i valori enumerazione in pratica delle costanti intere, il compilatore e` in grado di eseguire la conversione a int. E` 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 e` necessario specificare un valore per ogni identificatore dell'enumerazione, non ci sono limitazioni di segno e non e` necessario usare valori distinti (anche se cio` probabilmente comporterebbe qualche problema). Si puo` utilizzare anche un identificatore dell'enumerazione precedentemente definito.
La possibilita` 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 possibilita` 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 e` il piu` interessante perche` mostra un modo alternativo di dichiarare un nuovo tipo; in realta` 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.
E` 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 e` utile per creare abbreviazioni per espressioni di tipo complesse, soprattutto quando l'espressione di tipo coinvolge puntatori 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 e` modellato tramite una funzione il cui nome deve essere sempre main (vedi esempio).

Funzioni

Una funzione C/C++, analogamente ad una funzione Pascal, e` caratterizzata da un nome che la distingue univocamente nel suo scope (le regole di visibilita` 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 piu` argomenti hanno lo stesso tipo; la seguente dichiarazione e` 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 e` 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 e` valido in entrambi i linguaggi, in questo caso void assume il significato "Nessun argomento".
Anche in C++ e` 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 piu` file e la realizzazione di librerie di funzioni; infine nei prototipi non e` necessario indicare il nome degli argomenti della funzione:

```
// la funzione Sum vista sopra poteva
// essere dichiarata anche cosi`:
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 e` costituita da una sola istruzione che calcola la somma degli argomenti e restituisce tramite la keyword return il risultato di tale operazione. Inoltre, benche` non evidente dall'esempio, la keyword return provoca l'immediata terminazione della funzione; ecco un esempio non del tutto corretto, che pero` 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 e` 0, la prima istruzione return restituisce (erroneamente) una stringa (anzicche` 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), cio` tra l'altro consente di distinguere facilmente tra prototipo (dichiarazione) e definizione di funzione poiche` un prototipo e` terminato da ; (punto e virgola), mentre in una definizione la lista di argomenti e` 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 cosi`

solo la terminazione della funzione);

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

- Non e` 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 puo` essere eseguita solo nell'ambito dello scope in cui appare la sua dichiarazione (come gia` 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 e` un errore poiche` 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 realta` 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 e` 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 cioe` 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 e` 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 e` perfettamente corretto in quanto i parametri attuali vengono passati per valore: ad ogni

chiamata della funzione viene cioe` 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 puo` essere necessario fare in modo che la funzione possa modificare i suoi parametri attuali, in questo caso e` 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 possibilita`, ma rimanderemo la cosa al capitolo successivo non appena avremo parlato di puntatori e reference.

A volte siamo interessati a funzioni il cui comportamento e` pienamente definito anche quando in una chiamata non tutti i parametri sono specificati, vogliamo cioe` 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 e` 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 puo` avere piu` 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 e` errata poiche` 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 e` 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), e` un errore poiche` non viene specificato il valore per il secondo argomento della funzione (che non possiede un valore di default); e` invece interessante il terzo (Foo('c');): apparentemente potrebbe sembrare un errore, in realta` quello che il compilatore fa e` 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 sara` oggetto di una apposita appendice.

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 potra` 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. E` 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 puo` puntare solo a oggetti di quel tipo, (non e` 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 e` possibile farlo, ma viene eseguita una coercizione (vedi appendice A):

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

E` 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 avra` sicuramente influito il fatto che l'asterisco stava attacato al nome del tipo, tuttavia cambiando stile il problema non si risolve piu` di tanto. La soluzione migliore solitamente consigliata e` 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 e` mostrata nella penultima riga. Un puntatore a una costante consente l'accesso all'oggetto da esso puntato solo in lettura (ma cio` non implica che l'oggetto puntato sia effettivamente costante), mentre un puntatore costante e` una costante di tipo puntatore (a ...), non e` 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 e` 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 e` 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 e` 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 piu` 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 (cioe` 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 e` 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 si` 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 cosi` 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 e` stata modificata tramite P:
*P = -10
Val = -10

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

L'operazione di dereferenzazione puo` essere eseguita su un qualsiasi puntatore a condizione che questo non sia stato dichiarato void. In generale infatti non e` 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 e` 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) e` 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 e` definita una speciale aritmetica composta da somma e sottrazione. Se P e` un puntatore di tipo T, sommare 1 a P significa puntare all'elemento successivo di un ipotetico array di tipo T cui P e` immaginato puntare; analogamente sottrarre 1 significa puntare all'elemento precedente. E` possibile anche sottrarre da un puntatore un altro puntatore (dello stesso tipo), in questo caso il risultato e` 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 possibilita` 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 e` 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 e` 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 perche` 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 e` posto rispettivamente in Array e in Ptr, e come mostra l'esempio un char* puo` essere utilizzato esattamente come un array di caratteri. Esistono tuttavia profonde differenze tra puntatori e array: un puntatore e` una variabile a cui si possono applicare le operazioni viste sopra e che puo` essere usato come un array, ma non e` vero il viceversa, in particolare il nome di un array non e` un puntatore a cui e` possibile assegnare un nuovo valore (non e` cioe` 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 e` piu` flessibile di quanto non lo sia un array, anche se a costo di un maggiore overhead.

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

Linkage

Abbiamo gia` visto che ad ogni identificatore e` associato uno scope e una lifetime, ma gli identificatori di variabili, costanti e funzioni possiedono anche un linkage.
Per comprendere meglio il concetto e` necessario sapere che in C e in C++ l'unita` di compilazione e` il file, un programma puo` consistere di piu` file che vengono compilati separatamente e poi linkati (collegati) per ottenere un file eseguibile. Quest'ultima operazione e` 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 e` un problema, ma nel secondo file GetVar() deve utilizzare un nome dichiarato in un altro file; perche` la cosa sia possibile bisogna informare il compilatore che tale nome e` dichiarato da qualche altra parte e che il riferimento a tale nome non puo` essere risolto se non quando tutti i file sono stati compilati, solo il linker quindi puo` 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 e` 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 perche` a sarebbe stata definita due volte (una per file), il perche` di tale errore sara` chiaro piu` avanti.
Naturalmente extern si puo` usare anche con le funzioni (anche se come vedremo e` 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 e` 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 gia` detto, il C++ ha un'alta compatibilita` col C, tant'e` che e` 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++ e` 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 e` 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 e` necessario l'uso di extern, il solo prototipo e` sufficiente, e` 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, cosi` come le funzioni globali non inline; le funzioni inline, tutte le costanti e le dichiarazioni fatte in un blocco hanno invece linkage interno (cioe` 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 e` 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.
E` 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 e` ovvia; supponete di voler realizzare una libreria di funzioni,

alcune serviranno solo a scopi interni alla libreria e non serve (anzi e` pericoloso) esportarle, per fare cio` 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 e` 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 e` 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 e` gestita dal precompilatore che e` 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 gia` utilizzato la direttiva per poter eseguire input/output, in quei casi abbiamo utilizzato le parentesi angolari (< >) al posto dei doppi apici (" "); la differenza e` 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 e` stato specificato l'estensione del file (.h), questo non dipende dall'uso degli apici, ma dal fatto che ad essere incluso e` 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 puo` contenere in generale qualsiasi istruzione C/C++ (in particolare anche dichiarazioni extern) da condividere tra piu` moduli:

```
// Esempio2.h
```

```
// Un header puo` 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 piu` moduli, tuttavia la potenza dei file header si esprime meglio quando si vuole realizzare una libreria di funzioni.
L'idea e` 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 cio` che e` 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 e` 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 dovra` 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 e` 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 cioe` di un programma costituito da piu` moduli, quello principale che contiene la funzione main() e altri che implementano le varie routine necessarie. Piu` 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, poiche` il file principale include (direttamente e/o indirettamente) piu` volte lo stesso file header, il file che verra` effettivamente compilato conterra` piu` 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 e` fornita dal precompilatore stesso ed e` nota come compilazione condizionale; consiste cioe` nello specificare quando includere o meno determinate porzioni di codice. Per far cio` ci si avvale delle direttive #define SIMBOLO, #ifndef SIMBOLO e #endif: la prima ci permette di definire un simbolo, la seconda e` come l'istruzione condizionale e serve a testare un simbolo (la risposta e` positiva se SIMBOLO non e` 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 cioe` se un certo simbolo e` stato definito, se non lo e` (cioe` #ifndef e` verificata) si definisce il simbolo e poi si inserisce il codice C/C++, alla fine si inserisce l'#endif. Ritornando all'esempio, ecco cio` 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 e` definito e quindi #ifndef __IOSTREAM_H fa si` 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 e` 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 e` 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 puo` star sicuri che non ci saranno problemi di inclusione multipla. Tutto il meccanismo richiede pero` 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 piu` file e` legato alla necessita` di utilizzare identificatori globali unici. Quello che spesso accade e` che al progetto lavorino piu` 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 segnalera` 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 e` utilizzato. Molto meglio prevenire tale situazione suddividendo lo spazio globale dei nomi in parti piu` 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 piu` conflitto tra le definizioni dei due file, perche` per accedere ad esse e` 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 :: e` 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 e` quella del namespace MikeLib. Ovviamente perche` non ci siano conflitti e` necessario che i due namespace abbiano nomi diversi, ma e` piu` 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 perche` nello spazio globale non c'e` alcun Pi.
I namespace sono dunque dei contenitori di nomi su cui sono defite delle regole ben precise:

- Un namespace puo` 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;
- E` possibile creare un alias di un namespace con la sintassi: namespace < ID1 > = < ID2 >;
- E` 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 puo` 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 gia` visto in numerosi esempi:

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

/* ... */
```

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

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

using namespace MikeLib;
// Da questo momento in poi non e` 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 piu` direttive using namespace fanno si` che uno stesso nome venga importato da namespace diversi, si viene a creare una potenziale situazione di ambiguita` che diviene visibile (genera cioe` un errore) solo nel momento in cui ci si riferisce a quel nome. In questi casi per risolvere l'ambiguita` basta ricorrere ricorrere al risolutore di scope (::) qualificando totalmente il nome.

E` anche possibile usare la using per importare singoli nomi:

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

```
#include "MikeLib"
using namespace MikeLib;
using SamLib::Sum(int, int);

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

I costrutti analizzati fin'ora costituiscono gia` un linguaggio che ci consente di realizzare anche programmi complessi e di fatto, salvo alcune cose, quanto visto costituisce il linguaggio C; tuttavia il C++ e` molto di piu` e offre caratteristiche nuove che estendono e migliorano il C: programmazione a oggetti, RTTI (Run Time Type Information), template (modelli) e programmazione generica, gestione delle eccezioni. Si potrebbe apparentemente dire che si tratta solo di qualche aggiunta, in realta` nessun'altra affermazione potrebbe essere piu` errata: le eccezioni semplificano la gestione di situazioni anomale a run time, mentre il supporto alla programmazione ad oggetti e alla programmazione generica (e cio` 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. Per coloro che non avessero conoscenze in merito ai principi che stanno alla base di tale filosofia e` presente una breve appendice a scopo puramente introduttivo e assolutamente non completa (una trattazione approfondita richiederebbe ben altro spazio).

Strutture e campi funzione

La programmazione orientata agli oggetti (OOP) impone una nuova visione di concetti quali "Tipo di dato" e "Istanze di tipo". Sostanzialmente mentre altri paradigmi di programmazione vedono le istanze di un tipo di dato come una entita` passiva, nella programmazione a oggetti invece tali istanze diventano a tutti gli effetti entita` (oggetti) attive.
L'idea e` che non bisogna piu` manipolare direttamente i valori di una struttura (intesa come generico contenitore di valori), meglio lasciare che sia la struttura stessa a manipolarsi e a compiere le operazioni per noi. Tutto cio` che bisogna fare e` inviare all'oggetto un messaggio che specifichi l'operazione da compiere e attendere poi che l'oggetto stesso ci comunichi il risultato. Il meccanismo dei messaggi viene sostanzialmente implementato tramite quello della chiamata di funzione e l'insieme dei messaggi cui un oggetto risponde viene definito associando al tipo dell'oggetto un insieme di funzioni.
In C++ cio` puo` essere realizzato tramite le strutture:

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

  // Ora nelle strutture possiamo avere
  // dei campi di tipo funzione;
  void Print();
  float Abs();
  void Set(float PR, float PI);
};
```

Cio` che sostanzialmente cambia, rispetto a quanto visto, e` che una struttura puo` possedere campi di tipo funzione (detti funzioni membro oppure metodi) che costituiscono insieme ai campi ordinari (membri dato o attributi) l'insieme dei messaggi (interfaccia) a cui quel tipo e` in grado di rispondere. L'esempio non mostra come implementare le funzioni membro, per adesso ci basta sapere che esse vengono definite da qualche parte fuori dalla dichiarazione di struttura in modo pressocche` identico alle ordinarie funzioni.
Una funzione dichiarata come campo di una struttura puo` essere invocata ovviamente solo se associata ad una istanza della struttura stessa, dato che quello che si fa e` inviare un messaggio ad un oggetto. Cio` nella pratica si fa tramite la stessa sintassi utilizzata per selezionare un qualsiasi altro campo (solo che ora

ci sono anche campi funzione):

```
Complex A;
Complex* C;

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

Nell'esempio viene mostrato come inviare un messaggio: la quarta riga invia il messaggio Print() all'oggetto A, l'ultima invece invia il messaggio Abs() all'oggetto puntato da C e assegna il valore ottenuto alla variabile FloatVar. Anche la terza riga invia un messaggio ad A, in questo caso il messaggio richiede dei parametri che vengono forniti nello stesso modo in cui vengono forniti alle funzioni.
Il vantaggio principale di questo modo di procedere e` il non doversi piu` preoccupare di come e` fatto quel tipo, se si vuole eseguire una operazione su una sua istanza (ad esempio visualizzarne il valore) basta inviare il messaggio corretto, sara` l'oggetto in questione ad eseguirla per noi. Ovviamente perche` tutto funzioni e` necessario evitare di accedere direttamente agli attributi di un oggetto, altrimenti crolla uno dei capisaldi della OOP, e sfortunatamente per noi il meccanismo delle strutture consente l'accesso diretto a tutto cio` che fa parte della dichiarazione di struttura, annullando di fatto ogni vantaggio:

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

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

Sintassi della classe

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

```
class Complex {
  public:
    void Print();     // definizione eseguita altrove!

    /* altre funzioni membro */

  private:
    float Re;         // Parte reale
    float Im;         // Parte immaginaria
};
```

La differenza e` 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 cio` che sta fuori della classe e l'invocazione (selezione) di uno di questi campi e` sempre possibile;

- private: tutto cio` che segue e` visibile solo alla classe stessa, l'accesso ad uno di questi campi e` 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 e` preceduta da nessuna di queste keyword, il default e` private:

```
class Complex {
    float Re;          // private per
    float Im;          // default
  public:
    void Print();

    /* altre funzioni membro*/
};
```

In realta` esiste una terza categoria di visibilita` definibile tramite la keyword protected (che pero` analizzeremo quando parleremo di ereditarieta`); la sintassi per la dichiarazione di classe e` 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 visibilita`: definizioni di variabili o costanti (attributi), funzioni (metodi) oppure dichiarazioni di tipi (enumerazioni, unioni, strutture e anche classi), l'importante e` prestare attenzione a evitare di dichiarare private o protected cio` 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 puo` essere eseguita o dentro la dichiarazione di classe, facendo seguire alla lista di argomenti una coppia di parentesi graffe racchiudente la sequenza di istruzioni:

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

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

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

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

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

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

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() e` stata studiata con attenzione, il lettore avra` 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 e` 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 e` come si fa a stabilire dall'interno di un metodo qual'e` 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 e` stato inviato il messaggio e tramite questo e` possibile risalire all'indirizzo corretto; cio` inoltre consente la chiamata di un metodo da parte di un altro metodo:

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

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

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

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

Ovviamente un metodo puo` avere parametri e/o variabili locali che sono istanze della stessa classe cui appartiene (il nome della classe e` gia` 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 e` stato inviato Func(MyClass)
  A.BigOp();  // questo invece arriva al parametro.
  /* ... */
}
```

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

La programmazione orientata agli oggetti e` nata con lo scopo di risolvere il problema di sempre del modo dell'informatica: rendere economicamente possibile e facile il reimpiego di codice gia` scritto. Due sono sostanzialmente le tecniche di reimpiego del codice offerte: reimpiego per composizione e reimpiego per ereditarieta`; il C++ ha poi offerto anche il meccanismo dei Template che puo` 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'ereditarieta` il secondo pilastro (dopo l'incapsulazione di dati e codice) della programmazione a oggetti.

Reimpiego per composizione

Benche` non sia stato esplicitamente mostrato, non c'e` alcun limite alla complessita` di un membro dato di un oggetto; un attributo puo` avere sia tipo elementare che tipo definito dall'utente, in particolare un attributo puo` 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 gia` pronto quando si e` di fronte ad una relazione di tipo Has-a, in cui una entita` piu` piccola e` effettivamente parte di una piu` grossa. In questo caso il reimpiego e` servito per modellare una proprieta` della classe Lavoratore, ma sono possibili casi ancora piu` 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 puo` essere utilizzata anche per modellare una relazione di tipo Is-a, in cui invece una istanza di un certo tipo puo` essere vista anche come istanza di un tipo piu` "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 puo` essere utilizzata anche quando vogliamo semplicemente estendere le funzionalita` 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 realta` 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 e` necessario conoscere il modo in cui va costruito una componente nel momento in cui l'oggetto che la contiene viene istanziato;
- E` possibile che piu` oggetti contenitori condividano la stessa componente;
- Il contenimento tramite puntatori puo` essere utilizzato insieme all'ereditarieta` 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 e` totalmente costruito.

L'ultimo punto e` probabilmente il piu` difficile da capire e richiede la conoscenza del concetto di ereditarieta` che sara` esaminato in seguito. Sostanzialmente possiamo dire che poiche` 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 piu` flessibile e potente (espressivo) del contenimento diretto, potendo realizzare oggetti il cui comportamento puo` 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 anzicche` a scoppio). Come vedremo invece il polimorfismo consente di superare tale limite. Tutto cio` sara` comunque piu` 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 e` nota gia` in fase di compilazione, si conosce subito l'esatto tipo del componente e il compilatore puo` effettuare molte ottimizzazioni (e controlli) altrimenti impossibili (tipo espansione delle funzioni inline dei componenti);
- Non e` necessario eseguire operazioni di allocazione e deallocazione per costruire le componenti, ma e` necessario conoscere il modo in cui costruirle gia` 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 piu` efficente, non tanto perche` non e` 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 puo` 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 e` 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 puo` sembrare molto complesso, ma fortunatamente e` il compilatore che si occupa di tutta la faccenda, il programmatore deve occuparsi solo dell'oggetto con cui lavora, non delle sue componenti. Al piu` puo` capitare che si voglia avere il controllo sui costruttori da utilizzare per le componenti; l'operazione puo` 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 poiche` 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 benche` funzionalmente equivalente al precedente, non genera lo stesso codice. Infatti poiche` un costruttore per SmallObj non e` esplicitamente chiamato nella lista di inizializzazione e poiche` 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 cio` e` un maggiore overhead dovuto a due chiamate di funzione in piu`: 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 realta` una tale scelta e`

dovuta alla necessita` di garantire sempre che un oggetto sia inizializzato prima di essere utilizzato.
Ovviamente poiche` 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 e` sottointeso che se una componete di un oggetto e` a sua volta un oggetto composto, il procedimento viene iterato fino a che non si giunge a componenti di tipo primitivo.

Ora che e` 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 e` 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.

Ogni linguaggio di programmazione e` concepito per soddisfare determinati requisiti; i linguaggi procedurali (come il C) sono stati concepiti per realizzare applicazioni che non richiedano nel tempo piu` di poche modifiche. Al contrario i linguaggi a oggetti hanno come obiettivo l'estendibilita`, il programmatore e` in grado di estendere il linguaggio per adattarlo al problema da risolvere, in tal modo diviene piu` semplice modificare programmi creati precedentemente perche` via via che il problema cambia, il linguaggio si adatta. Famoso in tal senso e` stato FORTH, un linguaggio totalmente estensibile (senza alcuna limitazione), tuttavia nel caso di FORTH questa grande liberta` si rivelo` controproducente perche` spesso solo gli ideatori di un programma erano in grado di comprendene il codice.
Anche il C++ puo` 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

Cosi` come la definizione di classe deve soddisfare precise regole sintattiche e semantiche, cosi` l'overloading di un operatore deve soddisfare un opportuno insieme di requisiti:

1. Non e` possibile definire nuovi operatori, si puo` solamente eseguire l'overloading di uno per cui esiste gia` 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 ambiguita`.
2. Non e` possibile modificare la precedenza di un operatore e non e` possibile modificarne l'arieta` o l'associativita`, un operatore unario rimarra`sempre unario, uno binario dovra` applicarsi sempre a due operandi; analogamente uno associativo a sinistra rimmarra sempre associativo a sinistra.
3. Non e` concessa la possibilita` di eseguire l'overloading di alcuni operatori, ad esempio l'operatore ternario ? :, l'operatore sizeof e gli operatori di cast e in particolare l'operatore .* e l'operatore punto (per la selezione dei campi di una struttura).
4. E` 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'e` alcuna restrizione riguardo il contenuto del corpo di un operatore: un operatore altro non e` che un tipo particolare di funzione e tutto cio` che puo` essere fatto in una funzione puo` essere fatto anche in un operatore.
Un operatore e` 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 e` il tipo restituito (non ci sono restrizioni); @ indica un qualsiasi simbolo di operatore valido; ArgumentList e` la lista di parametri (tipo e nome) che l'operatore riceve, i parametri sono due per un operatore binario (il primo e` quello che compare a sinistra dell'operatore quando esso viene applicato) mentre e` uno solo per un operatore unario. Infine Body e` 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 e` che una funzione. Il funzionamento del codice e` chiaro e non mi dilunghero` oltre; si noti solo che i parametri sono passati per riferimento, non e` obligatorio, ma solitamente e` bene passare i parametri in questo modo (eventualmente utilizzando const come nell'esempio).
Definito l'operatore, e` 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 gia` sapete il compilatore e` 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 (cioe` scrivendo Complex C = operator+(A, B) o Complex C(operator+(A, B))), ma non e` una buona pratica in quanto annulla il vantaggio ottenuto ridefinendo l'operatore.
Quando un operatore viene ridefinito come funzione membro il primo parametro e` 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 e` 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 piu` 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 e` dichiarato const in quanto modifica l'oggetto su cui e` 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) e` consigliabile mantenere la coerenza semantica (cioe` 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 e` errato poiche` quello che si vuole utilizzare e` il meno binario, e tale operatore non e` stato definito.
Passiamo ora ad esaminare con maggiore dettaglio alcuni operatori che solitamente svolgono ruoli piu` difficili da capire.

L'operatore di assegnamento

L'assegnamento e` un operatore molto particolare, la sua semantica classica e` quella di modificare il valore dell'oggetto cui e` applicato con quello ricevuto come parametro e restituire poi tale valore al fine di consentire espressioni del tipo

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

che e` equivalente a

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

Non lo si confonda con il costruttore di copia: il costruttore e` utilizzato per costruire un nuovo oggetto inizializzandolo con il valore di un altro, l'assegnamento viene utilizzato su oggetti gia` costruiti.

```
Complex C = B;      // Costruttore di copia
/* ... */
C = D;              // Assegnamento
```

Un'altra particolarita` di questo operatore lo rende simile al costruttore (oltre al fatto che deve essere una funzione membro): se in una classe non ne viene definito uno nella forma X::operator=(X&), il compilatore ne fornisce uno che esegue la copia bit a bit. 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.
Notate infine che, come per le funzioni, anche per un operatore e` possibile avere piu` versioni overloaded; in particolare una classe puo` dichiarare piu` operatori di assegnamento, ma e` quello di cui sopra che il compilatore fornisce quando manca.

L'operatore di sottoscrizione

Un altro operatore un po' particolare e` quello di sottoscrizione [ ]. Si tratta di un operatore binario il cui primo operando e` l'argomento che appare a sinistra di [, mentre il secondo e` 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] e` dato da *(Arg1+Arg2) cioe` il valore contenuto all'indirizzo Arg1+Arg2. Questo operatore puo` essere ridefinito unicamente come funzione membro non statica e ovviamente non e` tenuto a sottostare al significato classico dell'operatore fornito dal linguaggio. Il problema principale che si riscontra nella definizione di questo operatore e` fare in modo che sia possibile utilizzare indici multipli, ovvero poter scrivere Arg1[Arg2][Arg3]; il trucco per riuscire in cio` consiste semplicemente nel restituire un riferimento al tipo di Arg1, ovvero seguire il seguente prototipo:

```
X& X::operator[](T Arg2);
```

dove T puo` 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 < Size) return Array[Index];
  else /* Errore */
}
```

Si tratta di una classe che incapsula il concetto di array per effettuare dei controlli sull'indice, evitando cosi` accessi fuori limite. La gestione della situazione di errore e` stata appositamente omessa, vedremo meglio come gestire queste situazioni quando parleremo di eccezioni.
Notate che l'operatore di sottoscrizione restituisce un int e non e` pertanto possibile usare indicizzazioni multiple, d'altronde la classe e` stata concepita unicamente per realizzare array monodimensionali di interi; una soluzione migliore, piu` flessibile e generale avrebbe richiesto l'uso dei template che saranno argomento del successivo capitolo.

Il meccanismo dell'ereditarieta` consente il riutilizzo di codice precedentemente scritto, l'idea e` quella di riconoscere le proprieta` di un certo insieme di valori (tipo) e di definirle realizzando una classe base (astratta) da specializzare poi caso per caso secondo le necessita`. 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 piu` opportuna.
Esiste un'altro approccio che per certi versi procede in senso opposto; anzicche` partire dai valori per determinarne le proprieta`, si definiscono a priori le proprieta` scrivendo codice che lavora su tipologie (non note) di oggetti che soddisfano tali proprieta` (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 proprieta`.
Quest'ultima tecnica prende il nome di programmazione generica ed il C++ la rende disponibile tramite il meccanismo dei template.
Un template altro non e` che codice parametrico, dove i parametri possono essere sia valori, sia nomi di tipo. Tutto sommato questa non e` una grossa novita`, le ordinarie funzioni sono gia` 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 proprieta` degli oggetti contenuti), nella realta` 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 funzionalita`. Tuttavia il codice riportato e` sufficiente ai nostri scopi.
Una implementazione di questo tipo funziona, ma soffre di (almeno) un grave difetto: la lista puo` memorizzare tutta una gerarchia di oggetti, e questo e` 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 (poiche` 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 e` necessario che il compilatore fornisca il codice macchina relativo alla lista ancora prima che un oggetto lista sia istanziato, e` sufficiente che tale codice sia generabile nel momento in cui e` 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 perche` cio` 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++ e` 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 e` molto complicata, la prima cosa che bisogna fare e` 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 utilizzera` un generico tipo T che sara` noto solo quando tale codice verra` 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 sara` 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 e` limitato alle sole dichiarazioni, vedremo in seguito come definire i metodi del template.
Intanto, si noti che e` 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 piu` riferimento ad un tipo esistente, la` dove e` stato utilizzato il nome fittizio TInfo. Inoltre il contenitore non memorizza piu` 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 e` stata istanziato il template ovvero tramite la notazione

```
NomeTemplate < Tipo >
```

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

Il tipo di ListOfReal e` 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 sara` 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);
```

```
}
```

Cioe` 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 puo` avere un qualsiasi numero di parametri non c'e` 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 e` prefissata. Fortunatamente un template puo` 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;
```

```
};
```

E` 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 puo` sembrare corretto, ma in effetti il compilatore non produrra` il risultato voluto. Il problema e` che il compilatore non puo` sapere in anticipo se T::TId e` 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 e` l'unico modo per ovviare a cio` e` 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 e` 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 e` il seguente:

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

    Alias Object
};
```

Un altro modo corretto di utilizzare typename e` nella dichiarazione di template:

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

In effetti se teniamo conto che il significato di class in una dichiarazione di template e` unicamente quella di indicare un nome di tipo che e` parametro del template e che tale parametro puo` non essere una classe (ma anche int o una struct, o un qualsiasi altro tipo), si capisce come sia piu` corretto utilizzare typename in luogo di class. La ragione per cui spesso troverete class invece di typename e` 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) e` che la loro istanziazione possa richiedere che su uno o piu` 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 e` 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 e` definito e` pero` un errore che puo` 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 e` lasciato ai messaggi di errore del compilatore e alla buona volonta` 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 e` 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 piu` 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 e`

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 e` 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, perche` gli argomenti del template vengono specificati esplicitamente ad ogni istanziazione.

Template ed ereditarieta`

E` possibile utilizzare contemporaneamente ereditarieta` 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 e` una classe base delle altre due, e` possibile memorizzare in ShapesList anche oggetti TRectangle e TTriangle perche` 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 e` poi tanto strana se si pensa al modo in cui sono gestiti e istanziati i template.
Un altro modo di combinare ereditarieta` e template e` 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'ereditarieta` e` stata utilizzata per estendere le caratteristiche della classe template Base, Tuttavia anche in questo caso tra le istanze dei template non vi e` alcuna relazione di discendenza, in particolare non esiste tra B e C; un puntatore a Base < T > non potra` mai puntare a Derived < T >.

Conclusioni

Template ed ereditarieta` sono strumenti assai diversi pur condividendo lo stesso fine: il riutilizzo di codice gia` 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 e` detto polimorfismo per inclusione, quello della programmazione generica invece e` detto polimorfismo parametrico). Le conseguenze dei due approcci sono diverse e diversi sono i limiti e le possibilita` (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 cio` in C++ ci permette da un lato di scegliero lo strumento che piu` 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.

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 e` 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 difficolta` connessa al trattamento delle eccezioni e` 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 funzionalita`, 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 e` 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 puo` 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 e` 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 e` un numero negativo? Le possibilita` sono due:

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

Probabilmente la prima possibilita` e` eccessivamente drastica, tanto piu` che non sappiamo a priori se e` 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 possibilita` sarebbe di per se` una buona soluzione, cosa succede se il chiamante ignora l'errore proseguendo come se nulla fosse?
E` 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 e` 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 e` 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 e` stata chiamata). Il compito di questo oggetto e` trasportare tutte le informazioni utili sull'evento.
L'argomento di throw puo` essere sia un tipo predefinito che un tipo definito dal programmatore.
Per compatibilita` con il vecchio codice, una funzione non e` tenuta a segnalare la possibilita` che possa lanciare una eccezione, ma e` buona norma avvisare dell'eventualita` 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 possibilita` che venga sollevata una eccezione di tipo char*; nel caso della seconda funzione invece a seconda dei casi puo` 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 e` sufficiente, non basta poter sollevare (segnalare) una eccezione ma e` 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 puo` generare una eccezione si segnala al compilatore che siamo pronti a gestire l'eventuale eccezione.
Ci si potra` chiedere per quale motivo sia necessario informare il compilatore dell'intenzione di catturare e gestire l'eccezione, il motivo sara` chiaro in seguito, al momento e` sufficiente sapere che cio` 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 dovra` 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 e` " << Divide(a, b);
  }
  catch(char* String) {
    cout << String << endl;
    return -1;
  }
  return 0;
}
```

Il generico blocco catch potra` gestire in generale solo una categoria di eccezioni o una eccezione generica. Per fornire codice diverso per diverse tipologie di errori bisognera` utilizzare piu` blocchi catch:

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

```
/* Altre catch */

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

/* Altro */
```

Ciascuna catch e` detta exception handler e riceve un parametro che e` il tipo di eccezione che viene gestito in quel blocco. Nel caso generale un blocco try sara` seguito da piu` 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 cioe` 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 e` 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 e` stato ancora creato e quindi nessun distruttore per esso viene invocato.
Il controllo e` 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 poiche` il tipo dell'eccezione e` 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 diverra` 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 e` ovviamente legato alla natura del programma e a cio` che si desidera fare; ad esempio ci potrebbero essere le

operazioni per eseguire dell'output su un file di log. E` buona norma studiare gli exception handler in modo che al loro interno non possano verificarsi eccezioni.

Nei casi in cui non interessa distinguere tra piu` tipologie di eccezioni, e` 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 piu` esterno, ad esempio perche` a quel livello e` 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 puo` portare a conoscenza dei blocchi piu` 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 trova un blocco try seguito da una catch compatibile con quel tipo di eccezione;

Esaminiamo il primo punto.
Per default una funzione che non specifica una lista di possibili tipi di eccezione puo` sollevare una eccezione di qualsiasi tipo. Una funzione che specifica una lista dei possibili tipi di eccezione e` teoricamente tenuta a rispettare tale lista, ma 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. Tuttavia e` possibile alterare tale comportamento definendo una funzione che non riceve alcun parametro e restituisce void ed 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 >. E` 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 e` importante, quello che conta e` 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, risalendo i blocchi applicativi, non si riesca a trovare un blocco try oppure una catch compatibile con il tipo di eccezione sollevata.
Nel caso si trovi un blocco try ma nessuna catch idonea, il processo viene iterato fino a quando una catch adatta viene trovata, oppure non si riesce a trovare alcun altro blocco try. Se nessun blocco try viene trovato, viene chiamata la funzione terminate().
Anche in questo caso, come per unexpected(), terminate() e` implementata tramite puntatore ed e` 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.

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 puo` 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!
```

e` possibile osservare che il distruttore per l'oggetto B istanza di Composed non viene eseguito perche` solo al termine del costruttore tale oggetto puo` essere considerato totalmente realizzato.

Le conseguenze di questo comportamento possono passare inosservate, ma e` 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 gia` allocato delle risorse (nel nostro caso della memoria); poiche` il distruttore non verrebbe eseguito e` 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. Cio` 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 cio` 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...

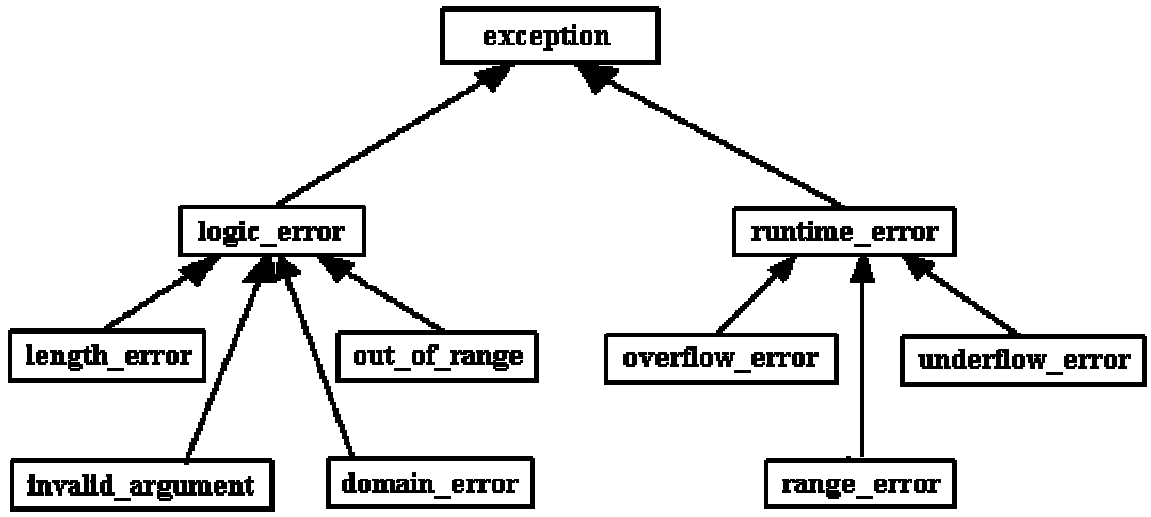
Come si potra` osservare, il blocco try della main viene eseguito normalmente e l'oggetto B viene distrutto non in seguito all'eccezione, ma solo perche` si esce dallo scope del blocco try cui appartiene.

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

1. Eseguire eventuali pulizie all'interno del costruttore se non si e` 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 puo` sollevare una eccezione di tipo bad_alloc, alcune classi standard (ad esempio la gerarchia degli iostream) ne prevedono altre. E` 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*) e` in grado di fornire una descrizione dell'evento (cosa esattamente c'e` scritto nell'area puntata dal puntatore restituito dipende dall'implementazione).
Le differenze tra logic_error e runtime_error sono sostanzialmente astratte, la prima classe e` 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:



Le varie classi sostanzialmente differiscono solo concettualmente, non ci sono differenze nel loro codice, in questo caso la derivazione e` 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 puo` 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 e` stata pensata per essere impiegata direttamente, il costruttore di tutte le altre classi riceve come unico argomento un const string&; il tipo string e` 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 e` raccomandabile fare diverse prove ed esercizi per comprendere cio` che sta dietro le quinte. La principale difficolta` e` quella di riconoscere i contesti in cui bisogna utilizzare le eccezioni ed ovviamente la strategia da seguire per gestire tali eventi. Cio` che bisogna tener presente e` che il meccanismo delle eccezioni e` sostanzialmente non locale poiche` il controllo ritorna indietro risalendo i vari blocchi applicativi. Cio` 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 complessita` del compito.
In generale non e` concepibile occuparsi di una possibile eccezione al livello di ogni singola funzione, a questo livello cio` che e` pensabile fare e` solo lanciare una eccezione; e` invece bene cercare di rendere i propri applicativi molto modulari e isolare e risolvere all'interno di ciascun blocco quante piu` 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 e` 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 e` possibile eseguire immediatamente l'operazione; si rende quindi necessario trasformare gli operandi in modo che assumano un tipo comune su cui e` 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 e` 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, poiche` la variabile d e` 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 priorita` maggiore delle operazioni aritmetiche).
Una conversione di tipo esplicita puo` essere richiesta con la sintassi

```
( < NuovoTipo > ) < Valore >
```

oppure

```
< NuovoTipo > ( < Valore > )
```

ma quest'ultimo metodo puo` essere utilizzato solo con nomi semplici (ad esempio non funziona con char*).
NuovoTipo puo` essere una qualsiasi espressione di tipo, anche una che coinvolga tipi definiti dall'utente; ad esempio:

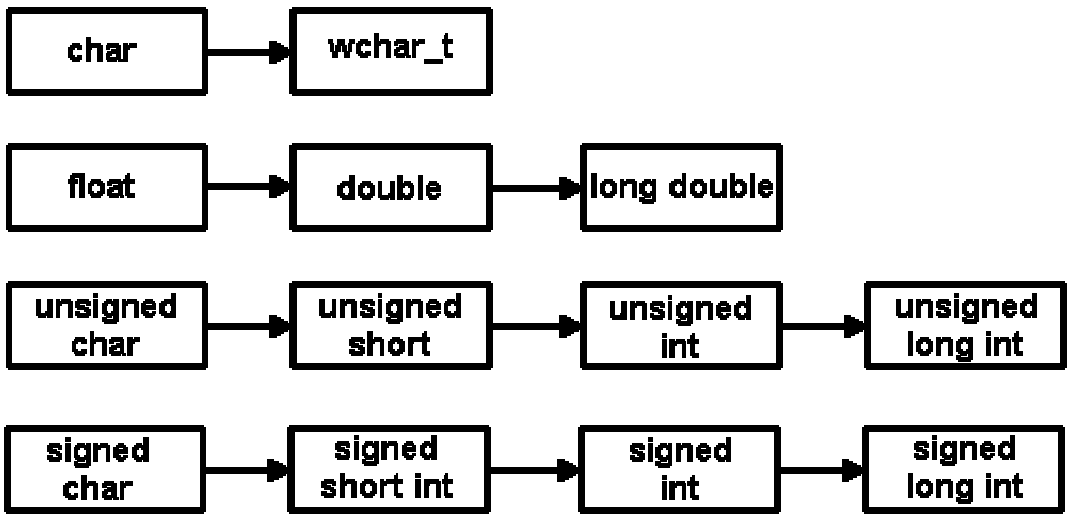
```
int a = 5;
float f = 2.2;

(float) a
// oppure...
float (a)

// se Persona e` un tipo definito dal programmatore...

(Persona) f
```

```
// oppure...
Persona (f)
```

Le conversioni tra tipi primitivi sono gia` predefinite nel linguaggio e possono essere esplicitamente utilizzate in qualsiasi momento, il compilatore comunque le utilizza implicitamente solo se il tipo di destinazione e` compatibile con quello di origine (cioe` puo` rappresentare il valore originale).
Un fattore da tener presente, quando si parla di conversioni, e` 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 e` rappresentabile il risultato e` indefinito).
Da questo punto di vista si puo` 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:



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 portabilita`.

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 e` 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 e` anch'esso un tipo definito dal programmatore.
Per convertire invece un tipo utente ad un tipo primitivo e` necessario definire un operatore di conversione. Con riferimento al precedente esempio, il metodo da seguire e` il seguente:

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

  private:
    float member;
};

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

Se cioe` 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 e` necessario indicare il tipo del valore restituito, e` implicito nel nome dell'operatore stesso.

C'e` un aspetto che bisogna sempre tener presente: quando si definisce un operatore di conversione, questo non necessariamente e` disponibile solo al programmatore, ma lo puo` 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 e` applicabile solo ai costruttori, non e possibile applicarla agli operatori di conversione; come conseguenza di cio` per impedire al compilatore l'uso automatico di un operatore di conversione e` 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 e` 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 e` 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 eventualita`;
- 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 e` utilizzato per risolvere un qualunque cast (eccetto quelli risolti da const_cast), usate questa sintassi quando siete sicuri che l'operazione e` correttamente fattibile.
reinterpret_cast e` in assoluto il tipo di cast piu` pericoloso perche` 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 e` 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 puo` essere eseguita solo se la classe base e` polimorfica (ha cioe` metodi virtuali), questa operazione richiede il RTTI ed e` 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 puo` 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 fattibilita` 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 e` 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 e` 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 perche` nel primo caso Studente e` un sottotipo di Persona e

l'operatore di conversione e` implicitamente definito; nel secondo caso l'operatore invece e` gia` 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 realta` (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 verita` 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 piu` 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++), e` sicuramente il paradigma object oriented.
Ad essere precisi quello object oriented non e` 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 e` 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 (realta`) tramite un insieme di entita` attive (che cioe` svolgono azioni) piu` 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 piu` o meno da una coppia: un gestore di documenti il cui compito e` occuparsi di tutto cio` 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 cosi` 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 piu` 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 attivita` (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 entita` attive della OOP (Object Oriented Programming) sono dette oggetti. Un oggetto e` una entita` software dotata di stato, comportamento e identita`. Lo stato viene generalmente modellato tramite un insieme di attributi (contenitori di valori), il comportamento e` costituito dalle azioni (metodi) che l'oggetto puo` compiere e infine l'identita` e` unica, immutabile e indipendente dallo stato, puo` essere pensata in prima approssimazione come all'indirizzo fisico di memoria in cui l'oggetto si trova (in realta` e` improprio identificare identita` e indirizzo, perche` 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 e` modellato dalle variabili f, g . Il comportamento e` 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 puo` occuparsi di ricevere ordini relativi all'esecuzione di certe operazioni aritmetiche su certi dati, per l'espletamento di tale compito puo` affidarsi ad un altro oggetto Calcolatrice fornendo il tipo dell'operazione da realizzare e gli operandi; l'oggetto Calcolatrice a sua volta puo` smistare le varie richieste a oggetti specializzati per le moltiplicazioni o le addizioni.
L'insieme dei messaggi cui un oggetto risponde e` detto interfaccia ed il meccanismo utilizzato per inviare messaggi e ricevere risposte e` quello della chiamata di procedura; nell'esempio di prima, l'interfaccia e` data dai metodi void Foo() e long double Foo2(int).

Ogni oggetto e` caratterizzato da un tipo; un tipo in generale e` 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 e` quello di occuparsi della corretta costruzione e inizializzazione delle singole istanze (costruttori) e della loro distruzione quando esse non servono piu` (distruttori).

Quando lo stato di un oggetto non e` 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'incapsulazione ha diverse importanti conseguenze, in particolare forza il programmatore a pensare e realizzare codice in modo tale che gli oggetti siano in sostanza delle unita` 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 e` l'esplicita presenza di strumenti atti a conseguire un facile reimpiego di codice precedentemente prodotto. L'obiettivo puo` essere raggiunto in diversi modi, ciascuna modalita` e` 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 ereditarieta`, distinguendo tra:
    - ereditarieta` di interfaccia
    - ereditarieta` 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 piu` oggetti di tipi anche diversi tra loro). L'oggetto composto fornisce alcune o tutte le funzionalita` della sua componente facendo da tramite tra questa e il mondo esterno, mentre le nuove funzionalita` sono implementate per mezzo di metodi e attributi propri dell'oggetto composto.
Un oggetto composto puo` contenere l'oggetto (e in generale gli oggetti) piu` 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 e` in generale piu` 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 e` 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 puo` avvenire per gradi, costruendo le sottocomponenti in tempi diversi;
- Una componente puo` essere condivisa da piu` 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 piu` ci fa comodo (anche cambiando a run time la componente stessa);
- In linguaggi come il C++ in cui un puntatore e` molto simile ad un array, possiamo realizzare relazioni in cui un oggetto puo` 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 piu` grande possiede uno o piu` oggetti tramite i quali espleta determinate funzioni (il caso dell'esempio del Lavoratore che possiede un Lavoro). Tuttavia e` 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'ereditarieta`.
L'idea di base e` 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) piu` 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 ereditarieta` 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 e` in presenza di ereditarieta` di interfaccia, la classe derivata ha l'obbligo di implementare tutto cio` che eredita (a meno che non si voglia derivare una nuova interfaccia), poiche` l'unica cosa che si eredita e` un insieme di nomi (identificatori di messaggi) cui non e` 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 cio` potrebbe non essere vero in altri linguaggi).
Alcuni modelli di OOP consentono l'ereditarieta` dell'implementazione (es. il C++), che puo` essere vista come caso generale in cui si eredita tutto cio` 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'ereditarieta` di implementazione viene fuori in quelle situazioni in cui il sottotipo esegue sostanzialmente gli stessi compiti del supertipo allo stesso modo (cambiano al piu` poche cose). Qualora il sottotipo dovesse gestire un messaggio in modo differente, viene comunque data la possibilita` 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 cio` che a quest'ultima apparteneva (interfaccia, stato, implementazione dell'interfaccia); Derived aggiunge nuove funzionalita` (Foo2()) e ridefinisce alcune di quelle ereditate (ridefinizione di Foo()), mentre altre funzionalita` vanno bene cosi` come sono (Sum()) e dunque la classe non le ridefinisce.
In alcuni sistemi potrebbe essere fornita la sola ereditarieta` di interfaccia, cosi` che le sole possibilita` 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'ereditarieta` modella in generale una relazione di tipo Is-a poiche` 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 e` di per se garantita dalla ereditarieta`, perche` cio` avvenga deve valere il principio di sostituibilita` di Liskov.

Tale principio afferma che la sostituibilita` e` 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).
E` anche possibile utilizzare l'ereditarieta` per modellare relazioni Has-a, ma si tratta spesso (praticamente sempre) di un grave errore e quindi tale caso non verra` preso in esame poiche` un linguaggio (o una tecnologia) OO fornisce sempre almeno il contenimento (o una qualche sua espressione).

Infine la delegation e` un meccanismo che tenta di mediare composizione e ereditarieta`. L'idea di base e` quella di consentire ad un oggetto di delegare dinamicamente certi compiti a altri oggetti.
Esprimere tale possibilita` in C++ non e` semplice, perche` 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 ereditarieta` (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.
Per una piu` approfondita discussione sul polimorfismo si rimanda al capitolo IX, paragrafo 9.