CARLO PESCIO

C++ MANUALE DI Stile

Una guida fondamentale per:

- **Migliorare lo stile e la chiarezza del codice**
- **Conoscere le tecniche dei programmatori professionisti**
- **Evitare gli errori più comuni**
- **Approfondire gli aspetti più complessi del linguaggio**
- **Adeguarsi allo standard ANSI/ISO**

Reload

Sommario

Introduzione alla versione Reloaded

Maggio 2010

Ho scritto "C++ Manuale di Stile" nel 1995. Nel 2005, per contratto, i diritti del libro sono tornati a me (in qualità di autore). La casa editrice, peraltro, ha nel frattempo cessato l'attività. Recentemente, ho deciso di distribuirlo **gratuitamente** in formato elettronico.

Ho reso disponibile il testo originale, digitalizzato, all'indirizzo

http://www.eptacom.net/pubblicazioni/cpp_stile/

Per migliorare la leggibilità del testo, e permetterne successive integrazioni ed aggiornamenti, ho intrapreso con un gruppo di volontari una revisione della bozza piu' recente in mio possesso, in modo da allinearla al testo definitivo.

Il testo che state leggendo è il risultato di questo lavoro, ed è aderente all'originale, salvo per:

- La numerazione delle pagine e delle note a piè pagina (dovuta ad integrazioni del testo con nuove note).

- L'integrazione delle modifiche riportate nell'errata corrige online, e di qualunque altra correzione a testo o listati emersa durante la revisione.

- L'impaginazione: ho usato un formato A5, in modo da facilitare la stampa in formato booklet su fogli A4.

- Alcune porzioni, scritte in colore blu, dove ho ritenuto importante modificare o integrare il testo originale.

Ogni forma di feedback costruttivo sarà sicuramente apprezzata.

Buona lettura!

Carlo Pescio

pescio@eptacom.net

(C) Copyright

Ringraziamenti

Ringrazio i volontari che mi hanno aiutato a creare la versione Reloaded del testo. In ordine alfabetico (per cognome):

Andrea Baliello

Paolo Bernardi - http://paolobernardi.wordpress.com

Fulvio Esposito

Nicola Fiorillo

Alessandro Gentilini - http://sites.google.com/site/alessandrogentilini/

Bruno Giavotto

Matteo Italia

Nicola Iuretigh

Filippo Mariotti

Michel Mazumder

Stefano Mazza

Roberto Melis

Roberto dell'Oglio

Daniele Pallastrelli

Eros Pedrini - http://www.contezero.net

Michelangelo Riccobene

Federica Romagnoli

Stefano Tondi - http://www.2centesimi.net

Corrado Valeri

Gianfranco Zuliani

Un ringraziamento particolare a:

Andrea Baliello, che mi ha fatto notare diversi punti in cui la prosa poteva essere migliorata.

Roberto Melis, che riesaminando il documento PDF non solo ha trovato errori sfuggiti alle prime revisioni, ma anche un utilizzo tecnicamente scorretto di “dereferenziare” nel testo originale.

Eros Pedrini, che ha trovato errori residui in diversi listati, sopravvissuti a tutte le precedenti letture.

1

Introduzione

"Any clod can have the facts,
but having opinions is an art"
Charles McCabe

Il C++ ha ormai raggiunto lo status di linguaggio di riferimento per la programmazione professionale; le ragioni di un simile successo sono molteplici, ma un elemento chiave è indubbiamente la sua flessibilità, che da un lato gli consente una astrazione sufficiente a modellare il dominio del problema, e dall'altro permette al programmatore il grande controllo spesso necessario in applicazioni reali.

Con la flessibilità, tuttavia, viene inevitabilmente anche la complessità; per quanto il C++ consenta un utilizzo graduale delle sue potenzialità, la completa padronanza del linguaggio, con le sue peculiarità, le sue stranezze, ma anche con la sua potenza e completezza, può derivare solo da uno studio attento e da un utilizzo intenso.

Questo libro si rivolge a chi già conosce il C++ e desidera approfondirne gli aspetti più complessi, nonché rivedere in una luce più critica (ma costruttiva e propositiva) l'utilizzo dei diversi costrutti del linguaggio; in alcuni casi, contraddistinti chiaramente dal simbolo "ISO C++", la trattazione è stata integrata con alcune annotazioni circa l'evoluzione che i diversi costrutti avranno nello standard ISO di prossimo rilascio.

Per ogni punto considerato, verranno esaminati diversi metodi di utilizzo, osservandone i punti a favore e le debolezze, nell'ottica di una programmazione professionale, attenta ai criteri di correttezza, leggibilità, mantenibilità, portabilità ed eleganza del codice; da ogni esempio, trarremo delle preziose raccomandazioni sullo stile di codifica più idoneo, che eviti gli errori più comuni e garantisca una più elevata comprensibilità ed espressività del codice. Ogniqualvolta fosse possibile, ho utilizzato frammenti di codice reale, tratti da programmi commerciali, anziché esempi interessanti ma accademici: sono convinto che la validità di un metodo di codifica vada comunque verificata "sul campo" piuttosto che giustificata su

basi esclusivamente teoriche, che devono fornire l'intuizione e la direzione ma che non possono, in sé, apportare la necessaria completezza. Ogni suggerimento in questo libro nasce da una sinergia tra teoria e pragmatica che spero porterà al lettore i benefici di entrambe.

L'insieme delle raccomandazioni qui raccolte non va comunque visto come un corpus di regole dogmatiche: non a caso, ognuna di esse è derivata solo come conseguenza di una attenta e dettagliata analisi delle alternative; lo scopo è di incoraggiare una filosofia di chiarezza del codice, non di limitare il programmatore all'interno di dettami difficili da seguire e contrastanti con le esigenze reali della programmazione.

Nondimeno, questo testo potrà essere usato anche come riferimento durante le revisioni o le ispezioni del codice, per risolvere una discussione sullo stile più opportuno, nonché come base di uno standard di codifica per un gruppo di lavoro. In effetti, uno standard di codifica può portare grandi benefici ad un team di sviluppo, riducendo il tempo dedicato ad elaborare i dettagli e permettendo agli sviluppatori di concentrarsi sulle parti fondamentali del codice. Proprio a tal fine, ho ritenuto opportuno includere una prima parte riguardante il layout del codice, la scelta dei nomi, e così via: nonostante argomenti come "lo stile di indentazione" siano più adatti a scatenare guerre sante che a portare alla cooperazione in un team di sviluppo, una trattazione completa dello stile di codifica non poteva prescindere da alcune considerazioni generali sull'aspetto "visivo" del codice stesso. In effetti, in grandi progetti il disaccordo su tali argomenti può facilmente portare a problemi ed incomprensioni, in termini di rapporti umani se non di ordine tecnico, tali da influenzare negativamente la vita del progetto stesso.

Chi non desideri comunque confrontare le proprie abitudini relative al layout o alla scelta dei nomi, può saltare i capitoli 2-4 ed iniziare direttamente la lettura con la trattazione di aspetti più semantici e meno tipografici; il potenziale lettore, che tema di trovarsi di fronte ad uno sterile elenco di norme arbitrarie e superficiali, può iniziare la lettura dal capitolo 7, relativo alle classi, o dal capitolo 10, che discute l'uso dell'ereditarietà, uno degli aspetti più interessanti del C++. Sono certo che riprenderà con interesse la lettura a partire dall'inizio.

2

La Scelta dei Nomi

"A good name is like a precious ointment;
it filleth all around about, and will not easy away..."
Francis Bacon

Generalità

Molti articoli sono stati scritti nel corso degli anni, con lo scopo di identificare delle regole generali per la scelta dei nomi di variabili, tipi, costanti, procedure, e così via (tra i più rilevanti, possiamo citare [Sim77], [Ana88], [Kel90]). Ci si dovrebbe comunque chiedere come prima cosa quale sia il fine nella ricerca attenta di nomi significativi: da esso dipenderà infatti il giudizio finale su cosa sia o meno un "buon identificatore".

Per un programmatore professionista, o che aspiri ad essere tale, lo scopo di un buon nome è una più elevata comprensibilità del codice; gli identificatori devono quindi aggiungere informazioni utili alla comprensione del codice stesso, informazioni di livello semantico che non sarebbero altrimenti presenti nel testo del programma. In tal senso, un programmatore che desideri migliorare il suo stile di programmazione deve sempre tenere presente che il suo programma ha due potenziali lettori: il compilatore ed altri esseri umani; il compilatore non ha alcun interesse per commenti, nomi significativi, e così via, che invece sono estremamente utili per i programmatori. Un buon programmatore sa che il suo compito è una attività umana e che il risultato del suo lavoro deve essere utilizzabile da altri programmatori: buon codice verrà riutilizzato e modificato, e ciò significa che dovrà essere facilmente comprensibile; se è più semplice riscrivere una funzione che modificarla o correggerla, chi ha scritto la routine è un programmatore incompleto, che vede la sua professione come un dialogo uomo-macchina, anziché come un dialogo uomo-macchina e uomo-uomo.

Cosa significa quindi "identificatore significativo"? In generale, l'identificatore dovrà chiarire lo scopo dell'elemento identificato, sia esso

una classe, una variabile, o una funzione. Esistono molte regole specifiche, che vedremo di seguito nel dettaglio, e che si applicano ai diversi elementi sintattici; tuttavia possiamo anche identificare alcune norme generali, che hanno validità pressoché assoluta.

Consideriamo ad esempio la lunghezza di un identificatore di variabile: è evidente che utilizzando identificatori più lunghi è possibile convogliare una maggiore quantità di informazioni, ma d'altra parte se ogni variabile avesse un identificatore di trenta o quaranta lettere, la leggibilità del codice sarebbe seriamente compromessa. In realtà, non si può prescindere dall'analisi del contesto per giudicare la bontà di un identificatore; consideriamo il **Listato 1**:

Listato 1

```
// GLOBAL.H

int* p ;
void f( int x ) ;

// MAIN.CPP

int main()
  {
  const int SIZE = 10 ;

  p = new int[ SIZE ] ;
  for( int i = 0; i < SIZE; i++ )
    p[ i ] = 0 ;

  // ...

  for( int j = 0; j < SIZE; j++ )
    f( p[ j ] ) ;

  // ...

  return( 0 ) ;
  }
```

Alcuni identificatori sono realmente poco chiari: ad esempio, che cosa dovrebbe contenere (a livello logico, o di dominio del problema) l'area di memoria puntata da *p*? Qual è la semantica astratta della funzione *f*?

D'altra parte, la brevità dei nomi non è necessariamente una caratteristica negativa: l'uso delle variabili *i* e *j* come indici dei loop è molto chiaro, e non sarebbe realmente più espressivo utilizzare nomi più lunghi come *index* (per quanto alcuni autori lo suggeriscano); in fondo, nella usuale notazione matematica si usano sempre variabili di una sola lettera, senza grandi problemi. Anzi, usare un identificatore eccessivamente prolisso per un indice o un contatore, utilizzato localmente in poche righe di codice, potrebbe essere addirittura controproducente per la comprensione, in quanto sposterebbe l'attenzione dal costrutto in uso (ad esempio il *for*) alla variabile indice.
È evidente che devono esistere considerazioni di livello più alto, poiché in alcune situazioni un identificatore breve è perfettamente adeguato, in altre rende il codice praticamente incomprensibile.

In effetti, vi sono due regole fondamentali per la scelta degli identificatori, siano essi relativi a tipi, variabili, funzioni, o quant'altro:

1. adattare la prolissità del nome al contesto nel quale l'identificatore dovrà logicamente comparire.

2. cercare di programmare nello spazio del problema, non nello spazio della soluzione.

La prima regola fornisce una adeguata interpretazione ai problemi evidenziati dal **Listato 1**: quando utilizziamo oggetti globali, il contesto di utilizzo fornisce un aiuto molto limitato alla comprensione. Ciò significa che il loro nome dovrà avere una espressività sufficiente ad identificare chiaramente la natura e lo scopo di tali elementi, indipendentemente dal contesto di utilizzo. Nomi come *p* o *f* sono ben poco significativi in sé, e quindi non sono adatti per elementi globali (vedremo in seguito con precisione cosa va ritenuto "globale"). Per contro, lo scopo di variabili locali utilizzate in un breve frammento di codice, come un loop, è in gran parte chiarito dal contesto in cui appaiono; pertanto, utilizzare *p* come puntatore ad un elemento di array in un loop di inizializzazione è sufficientemente chiaro, ed un identificatore come *pointerToCurrentItem* non aggiunge informazioni significative. Diverso è il caso di un loop che si estenda per molte righe (che comunque potrebbe beneficiare di una maggiore astrazione funzionale), dove un identificatore più lungo potrebbe evitare confusione.

Una volta raggiunta una buona dimestichezza con la "regola 1", i maggiori benefici per la comprensione del codice vengono da una corretta

applicazione della “regola 2”; il **Listato 2** è un frammento di codice reale, scritto da un programmatore attento e preparato, che tuttavia ha sempre trascurato la “regola 2” (sino al momento di una lunga discussione, durante la quale ho avuto modo di convincerlo della validità della stessa).

Listato 2

```
HashTable hashTable ;

// ...

descr = hashTable.Find( code ) ;

// ...
```

Possiamo facilmente immaginare che *descr* sia la descrizione di “qualcosa” che viene cercato in una hash table tramite il suo codice (*code*). Il frammento non ci aiuta minimamente a capire *cosa* venga estratto dalla tavola; a dire il vero, ci fornisce anche un dettaglio totalmente irrilevante, ed in modo ridondante: la variabile *hashTable* è semplicemente una tavola codice/descrizione relativa a qualche oggetto, che per ragioni implementative è stata modellata come una hash table. Il nome stesso della variabile, che altro non è se non il nome del tipo, è già di per sé indice di una cattiva scelta; è infatti affetto dai seguenti problemi:

- mancanza di information hiding: espone un dettaglio implementativo non rilevante. La tavola potrebbe ad esempio essere memorizzata come un albero binario senza che il resto del codice richieda modifiche.

- accoppiamento sull’implementazione: se decidessimo, ad esempio, di modificare il tipo della variabile da *HashTable* a *SearchTree*, dovremmo anche modificare il nome della variabile stessa. Ciò avviene perché il nome è stato scelto nello spazio della soluzione, che tende a cambiare di frequente, e non nello spazio del problema, che è di norma molto più stabile.

- scarsità di contenuto: il nome non ci rivela molto su cosa sia contenuto nella tavola, o per quale scopo la tavola sia stata creata.

Portando all’eccesso lo stile del **Listato 2**, ogni array si chiamerebbe *array*, ogni matrice *matrix*, e così via, fornendo ben pochi legami tra l’implementazione e le fasi di più alto livello (design ed analisi) che

avevano identificato il dominio del problema. Tra l'altro, un indizio tipico di una scelta degli identificatori troppo legata alla soluzione, piuttosto che al problema, è l'uso (o l'abuso) di identificatori scelti dal vocabolario dell'informatica, come appunto *hashTable*, *vector*, *binaryTree*, ecc. Mentre questi sono ovviamente corretti come identificatori delle classi che li implementano (in una libreria di classi base, tali identificatori apparterranno in effetti al dominio del problema) in tutte le altre situazioni avere variabili o funzioni con nomi simili dovrebbe far seriamente riflettere sull'opportunità di scegliere un identificatore più descrittivo. In tal senso, una versione significativamente migliore del **Listato 2** è quella del **Listato 3**:

Listato 3

```
HashTable sensorList ;

// ...

sensorDescr = sensorList.Find( sensorCode ) ;

// ...
```

Tralasciando i dettagli minori, come l'uso di un prefisso univoco *sensor* per identificare tutti gli elementi che in qualche modo si riferiscono alla stessa classe di elementi, il "significativo miglioramento" (per quanto sia piuttosto altisonante quando riferito a poche righe di codice) è la modifica dell'identificatore *hashTable* in *sensorList*. All'interno del programma in questione, infatti, il ruolo della variabile era di contenere una lista di sensori, sulla quale venivano eseguite diverse operazioni di inserimento, ricerca e modifica; osserviamo che ora il nome della variabile non è più legato alla sua implementazione, che pertanto può tranquillamente cambiare, e che il nuovo nome ci fornisce una reale indicazione circa lo scopo ed il contenuto della variabile stessa.

Esiste comunque un problema abbastanza comune, ovvero l'esistenza dei sinonimi, o di espressioni complesse aventi lo stesso significato, nonché l'ambiguità intrinseca delle abbreviazioni. Ad esempio, un membro *pageNum* in una classe *Document* conterrà il numero di pagine o il numero della pagina corrente? In questi casi, esiste una terza regola d'oro per i gruppi di lavoro: adottare una convenzione per i casi più frequenti, come indici, contatori, dimensioni, numero di elementi. Ciò si può estendere anche alle funzioni: ad esempio, funzioni che ritornano il valore di membri di una classe possono o meno avere il prefisso *Get*, funzioni che verificano

alcune proprietà possono o meno avere un prefisso *Is*, e così via. Rimanendo a livello di esempio, potremmo dire che *numOfPages* per indicare il numero totale di pagine è una scelta migliore di *pageNum*, in quanto meno ambiguo; l'ambiguità può comunque essere rimossa anche da opportune convenzioni sui prefissi.
Esistono delle convenzioni sulla scelta dei nomi che si estendono a coprire anche i prefissi più comuni; una di queste è la convenzione *Hungarian*, sulla quale ritorneremo in seguito, in termini tuttavia piuttosto critici. In realtà, la migliore opportunità per un gruppo di lavoro è di definire un proprio insieme di prefissi per i casi più comuni, e cercare di aderirvi fedelmente, senza lasciare tuttavia che il prefisso prenda il sopravvento sul resto del nome: è in genere più facile capire che una variabile è utilizzata come contatore, piuttosto che capire *cosa* sta contando: nuovamente, ponete l'accento sulle informazioni che provengono dall'analisi e dal design (dominio del problema) piuttosto che su quelle che possono essere capite direttamente leggendo il codice (dominio della soluzione). Così come aggiungere ad una riga "*i++ ;*" il commento "*// incrementa i*" è totalmente inutile, perché non aggiunge informazione al codice, così chiamare una hash table *hashTable* rappresenta un'occasione mancata per rendere il nostro codice più chiaro e stabile.

Raccomandazione 1

Utilizzare identificatori più lunghi e descrittivi per gli elementi privi di contesto; limitare la lunghezza degli identificatori quando il contesto aiuta a comprenderne il significato logico.

Raccomandazione 2

Scegliete gli identificatori nel dominio del problema, non in quello della soluzione.

Raccomandazione 3

Sviluppate delle convenzioni locali per i prefissi più comuni, come indici e numero di elementi; non lasciate però che il prefisso prenda il sopravvento sul corpo dell'identificatore.

Veniamo infine ad un problema abbastanza noto, riguardante l'uso di identificatori simili (ad esempio, differenti solo nella prima lettera, maiuscola in un caso e minuscola nell'altro). Non si tratta di un esempio

accademico, in quanto è abbastanza frequente trovare codice come quello del **Listato 4**:

Listato 4

```
class Rectangle
  {
  int Width ;
  // ...
  } ;

void Rectangle :: SetWidth( int width )
  {
  width = Width ;
  }
```

Il problema maggiore in questi casi è che gli esseri umani tendono molto spesso a leggere quello che già si aspettano di trovare: nel caso non lo aveste notato, il **Listato 4** è errato, ovvero assegna il membro della classe al parametro formale e non viceversa[1].
L'uso di nomi "simili" dovrebbe essere limitato ai casi in cui non sia possibile utilizzare l'uno al posto dell'altro senza un errore di compilazione, o nel caso una possibile confusione sia irrilevante ai fini della correttezza del programma; un esempio tipico è quello del **Listato 2**, dove la classe *HashTable* e la variabile *hashTable* differiscono solo nel case della prima lettera. In tal caso, tuttavia, i contesti in cui possiamo utilizzare l'una o l'altra sono disgiunti, tranne che per *sizeof(HashTable*) dove tuttavia l'uso di uno o dell'altro è irrilevante. Vale comunque la pena di insistere sul fatto che cercando di assegnare alle variabili nomi presi dal dominio del problema, casi simili dovrebbero essere ridotti al minimo.

[1]In effetti la possibilità di modificare il valore di un parametro formale (modifica che ovviamente non si rifletterà sul parametro attuale) è più che altro un'eredità dei tempi in cui risparmiare anche i pochi byte necessari per una variabile risultava fondamentale; in ogni caso, rimane una possibilità del linguaggio, anche se può essere prevenuta dichiarando i parametri formali come *const*.

Raccomandazione 4

Evitare nomi che differiscono solo nel case o solo per caratteri simili, come 1 ed l (“uno” ed “elle minuscola”) oppure 0 ed O (“zero ed “o maiuscola”). In realtà sarebbe opportuno evitare totalmente l’uso dei numeri negli identificatori.

La scelta di una lingua

Negli esempi precedenti, ho utilizzato nomi di classe e di variabili ispirati alla lingua inglese; in effetti, tale scelta sarà mantenuta per l’intero testo, anche se i commenti saranno in Italiano a beneficio dei (presumibilmente pochi) lettori che trovino ostiche altre lingue. Un elemento di reale fastidio per la lettura è però l’uso, tutt’altro che infrequente, di una forma distorta di Italiano, Inglese e “computerese” nella scelta degli identificatori: se siete così fortunati da non avere mai incontrato simile codice, il **Listato 5** rappresenta un piccolo campionario di esempi:

Listato 5

```
void PrintTesto( Testo t ) ;

int valoriSorted[ 10 ] ;

while( s.Contiene( ‘a’ ) == TRUE )
  // ...

void Tabella :: SwapElementi( int index1, int index2 )
  {
  int temp = tabella[ index1 ] ;
  tabella[ index1 ] = tabella[ index2 ] ;
  tabella[ index2 ] = temp ;
  }

int fine ; // Italiano “termine” o Inglese “bello”,
           // usato anche come OK
           // (altro significato: “multa”)?
```

L’uso di un linguaggio misto può essere idoneo per discussioni tecniche tra amici e colleghi, in quanto consente di fatto una scelta di termini molto ampia; tuttavia durante una discussione il contesto generale può chiarire il significato del termine, oppure la pronuncia può risolvere l’ambiguità, ed infine è sempre possibile chiedere un chiarimento, mentre durante la lettura

del codice scritto da altri programmatori non si ha nessuno di tali preziosi vantaggi. In realtà, la migliore soluzione è di scegliere una lingua ed usarla in modo consistente: se i programmatori del team non conoscono l'Inglese, possono scegliere l'Italiano, ma allora dovrebbero utilizzare solo l'Italiano per evitare le possibili ambiguità (che non possono evitare in altri modi, poiché non conoscono l'Inglese).

Indubbiamente, avendo il C++ poche parole chiave (in Inglese), si ha minore dissonanza che in altri linguaggi (es. Pascal, ma anche Cobol) dovuta alla inevitabile presenza di identificatori in Italiano e parole chiave in Inglese (come in *while(s.Contiene('a'))*), e quindi l'uso dell'Italiano è più che "tollerabile". Rimane il grosso problema delle librerie esistenti, tutte o quasi sviluppate utilizzando l'Inglese come lingua di riferimento, e certamente l'uso dell'Italiano costituisce un elemento limitativo enorme alla diffusione del codice, al suo riuso in ambito internazionale, e dovrebbe pertanto essere evitato durante lo sviluppo di librerie. In ogni caso, sarebbe estremamente consigliabile scegliere una lingua e rimanervi fedeli durante lo sviluppo; se possibile, scegliete l'Inglese.

Raccomandazione 5

Cercate di evitare l'uso contemporaneo di più lingue: sceglietene una (preferibilmente l'Inglese) e restate coerenti con tale scelta durante l'intero sviluppo.

Classi e tipi

Considerando quanto sopra, la scelta degli identificatori per le classi ed i tipi dovrebbe ricadere su nomi piuttosto lunghi e descrittivi, avendo le classi ed i tipi identificatori di livello globale. Una eccezione è rappresentata dalle classi annidate (molto rare) e dai tipi locali ad una classe (es. enumerati), dove la classe stessa fornisce un contesto sufficiente a chiarire un nome breve. Un nome lungo e descrittivo per le classi riduce anche la probabilità di una collisione di nomi tra classi di librerie diverse, viceversa non risolubile se il compilatore non supporta i *namespace*.

Il nome della classe dovrebbe poi essere estratto dal dominio del problema, come abbiamo visto in precedenza; in mancanza di documenti di analisi e design object oriented, il programmatore potrebbe trovarsi a dover scegliere il nome più opportuno in totale autonomia: in tal caso, è meglio orientarsi su un nome che descriva i dati contenuti, piuttosto che i compiti svolti, e

che rifletta una visione astratta della classe, piuttosto che una implementazione concreta.

Un esempio abbastanza semplice potrebbe essere una classe *Set* (insieme), implementata come un vettore di bit. La scelta tra *Set* e *BitVector* dovrebbe cadere sulla prima, oppure dovrebbe esistere una classe base *BitVector* ed una classe derivata (con ereditarietà privata, di implementazione) *Set*. Le ragioni sono ovvie: se il nostro scopo è di modellare un insieme, la scelta del vettore di bit è solo una delle tante implementazioni possibili: non dovremmo pertanto rendere evidente l'implementazione nel nome della classe.

Spostare l'attenzione sui dati, piuttosto che sulle funzioni, è un ulteriore metodo per garantire stabilità al nome della classe, in quanto nella manutenzione del codice i servizi che la classe esporta tendono (di norma) a cambiare molto più spesso dei dati che la classe possiede. Ciò corrisponde al fatto che nel mondo reale, le procedure ed i processi cambiano più rapidamente dei prodotti.

Vi sono rari casi in cui una classe esporta solo funzioni (o quasi), ed è quindi difficile scegliere un nome basato sui dati; classi di questo tipo sono abbastanza sospette (in [Mey88], l'autore associa alla definizione di tali classi "il grande errore" dei principianti), e nei rari casi in cui sono realmente necessarie è importante trovare un nome sufficientemente rappresentativo delle azioni compiute.

Raccomandazione 6

Scegliete i nomi delle classi dal dominio del problema; il nome dovrebbe normalmente rappresentare il tipo di dato astratto, non la specifica implementazione.

Funzioni

Le funzioni del C++ rappresentano l'unificazione di due concetti che in altri linguaggi vengono mantenuti separati, ovvero la funzione propriamente detta (che restituisce un valore e non ha effetti collaterali) e la procedura (che può o meno restituire un valore, ed ha effetti collaterali). In C++, non essendo la differenza esprimibile direttamente nel linguaggio, è compito del programmatore scegliere in modo opportuno gli identificatori, in modo che chi legge il codice sappia quando viene richiesto il calcolo di

un valore e quando l'esecuzione di un comando. Notiamo che la distinzione operata tra procedura e funzione deve essere piuttosto astratta: una funzione che modifichi alcuni membri di un oggetto per conservare una cache degli ultimi valori letti, può nondimeno essere considerata una "funzione pura" se gli effetti collaterali non hanno influenza sul risultato degli altri metodi pubblici della classe.

Una volta stabilito se la funzione C++ rappresenta una "funzione pura" od una procedura, la scelta del nome dovrebbe seguire un principio abbastanza universale, il cui scopo è rendere il codice più vicino al linguaggio naturale:

- I nomi di funzione dovrebbero descrivere il risultato.

- I nomi di procedura dovrebbero descrivere il compito svolto: un verbo (seguito da un oggetto se *non* si tratta di una funzione membro) è normalmente la scelta migliore.

In questo modo, la lettura del codice proseguirà in modo naturale, con statement del tipo *if(stack.IsEmpty())* oppure *buffer.Flush()* e così via; usare un verbo in una funzione, ad esempio trasformando *IsEmpty* in *CheckEmpty*, rende la lettura significativamente più difficile: *if(stack.CheckEmpty())* è addirittura ambiguo: un risultato TRUE significa che è effettivamente vuoto o che non lo è? Dovreste sempre concentrare l'attenzione sulla possibile ambiguità dei nomi di funzione.

Sia per le funzioni che per le procedure, il nome dovrebbe descrivere, ad un adeguato livello di astrazione, *tutto* ciò che la funzione calcola o *tutto* ciò che la procedura esegue; cercate di evitare nomi volutamente indefiniti come *HandleData(), Check()*, e così via. Se non è possibile trovare un buon nome, spesso significa che la funzione/procedura esegue troppi compiti e sarebbe meglio suddividerla: *ShowPromptAndAcceptUserInput* è più elegante e riutilizzabile se partizionata in *ShowPrompt* ed *AcceptUserInput*.

In linguaggi imperativi (e quindi anche in C++ quando scriviamo procedure non-membro) è sempre opportuno far seguire all'azione il nome dell'oggetto al quale si applica, e definire nel nome della funzione la sorgente dei valori. Per contro, se stiamo scrivendo una funzione membro, è in genere meglio evitare di specificare l'oggetto, che è implicito nella classe stessa. Il codice del **Listato 6**, scritto (realmente) da un programmatore convertitosi "istantanemente" dal Pascal al C++, rappresenta un classico esempio di cosa *non* fare:

Listato 6

```
class Sensor
  {
  public :
    const char* GetSensorName() ;
  // ...
  } ;


class Device
  {
  public :
    const char* GetDeviceName() ;
  // ...
  } ;


class AnalysisModule
  {
  public :
    const char* GetModuleName() ;
  // ...
  } ;
```

Ogni classe ha un metodo per restituire il nome dell'oggetto, tuttavia ogni classe ha un nome diverso per la stessa funzione (a livello concettuale); ovviamente, in seguito tale struttura si era rivelata piuttosto limitativa, ed era stata modificata come da **Listato 7** (la classe *NamedProcedureObject* è una astrazione di tutti gli oggetti con un nome all'interno di uno schema detto "procedura" nel dominio del problema):

Listato 7

```
class NamedProcedureObject
  {
  public :
    const char* GetName() const ;
  // ...
  } ;

class Sensor : public NamedProcedureObject
  {
  // ...
  } ;
```

```
class Device : public NamedProcedureObject
  {
  // ...
  } ;


class AnalysisModule : public NamedProcedureObject
  {
  // ...
  } ;
```

Notiamo che, al di là di una struttura migliore per il riuso del codice, il nome della funzione è anche più breve; seguendo quanto detto in precedenza, ciò non dovrebbe stupire: la classe opera da contesto per chiarire lo scopo della funzione al momento della dichiarazione, e l'oggetto opera da contesto per chiarire lo scopo al momento della chiamata. Ovviamente, ciò richiede che la classe e gli oggetti abbiano a loro volta nomi significativi.

Raccomandazione 7

Funzioni "pure" devono avere un nome che rappresenti adeguatamente il risultato restituito.

Raccomandazione 8

Funzioni con side-effects (procedure) devono avere un nome che descriva ad un giusto livello di astrazione tutti i compiti eseguiti. Usate un verbo seguito da un complemento oggetto per le funzioni non membro, ed un verbo per le funzioni membro, lasciando l'oggetto implicito.

Variabili e Costanti

Le variabili sono in genere più numerose delle classi e delle funzioni, ed i programmatori sono pertanto più tentati di usare per esse nomi brevi, talvolta poco significativi o poco stabili; la stabilità di un nome è molto importante, poiché durante la manutenzione ed il debugging difficilmente si avrà il tempo necessario per modificare i nomi delle variabili a fronte delle modifiche: pertanto dovremmo cercare non solo di massimizzare la chiarezza del nome pur mantenendo una lunghezza contenuta, ma anche di scegliere nomi resilienti rispetto alla fase di manutenzione.

Il metodo migliore per ottenere un nome "che dura nel tempo" è di specificare nell'identificatore l'*utilizzo* che si vuole fare della variabile; mentre molte altre caratteristiche (il tipo, il range di valori assunti dinamicamente, la lifetime) tendono a cambiare, l'uso che si fa della variabile tende a persistere. Analogamente, scegliere i nomi nello spazio della soluzione (vedere nuovamente il **Listato 2** ed il **Listato 3** per un esempio concreto) produce identificatori più stabili.

Esistono alcuni studi sulla lunghezza "ottimale" per i nomi di variabili, legati principalmente all'attività di debugging, che suggeriscono una lunghezza media tra i 10 ed i 16 caratteri; in effetti, studi più approfonditi [Shn80] dimostrano che nomi brevi sono più indicati per variabili locali o indici di loop, e nomi lunghi sono più indicati per variabili globali. Ciò non fa che confermare la validità delle regole viste in precedenza: usate la lunghezza adeguata per descrivere, nella sua interezza, l'uso che fate della variabile, assumendo comunque che un eventuale contesto possa aiutare nella comprensione.

Osserviamo che usare nomi significativi per le variabili previene il "riuso eterogeneo", ovvero l'utilizzo di una variabile intera *x* per contenere dapprima una lunghezza, poi un'area, e così via, all'interno della stessa routine. Mentre questa pratica può essere utile quando si programma in assembler, dove minimizzare il numero di registri utilizzati può significativamente migliorare le prestazioni di un programma, in un linguaggio ad alto livello dovremmo lasciare al compilatore il compito di gestire la lifetime delle variabili: ulteriori consigli in questo senso verranno dati nel capitolo 6.

Infine, i membri dato delle classi possono essere ampiamente assimilati a variabili, e pertanto la scelta dei nomi può seguire le stesse regole; spesso in questo caso si è facilitati poiché le fasi di analisi e design dovrebbero averci fornito nomi adeguati per gli attributi che intendiamo modellare. I nomi dei campi hanno un contesto (la classe) che può aiutare nella comprensione: in ogni caso, è bene non abusarne ed utilizzare comunque identificatori significativi.

Raccomandazione 9

Scegliete gli identificatori di variabile e costante per rappresentarne l'uso, riferendovi al dominio del problema e non all'implementazione; utilizzate identificatori corti se il loro uso è chiarito dal contesto locale, ed identificatori lunghi per variabili globali o con lifetime estesa.

Hungarian ed altre convenzioni

Abbiamo visto in precedenza che un notevole beneficio può venire dall'introduzione di alcune convenzioni, usate in modo sistematico, che riducano le scelte arbitrarie del singolo programmatore e semplifichino così la lettura del codice: ad esempio, l'uso di un prefisso o suffisso comune per ogni variabile usata come contatore, e così via.
Una convenzione molto diffusa è *Hungarian*[2] [Sim77], [SH91] che fornisce un metodo per codificare all'interno dell'identificatore il tipo e l'uso "implementativo" di una variabile, nonché alcuni schemi di utilizzo "astratto": ad esempio, *hwndMenu* indica l'handle per una finestra (window) con funzione di menu. La convenzione si estende anche alle procedure, ed è stata anche adattata alle classi: molti ad esempio seguono la tendenza di far iniziare ogni identificatore di classe per 'C'.
Come molti altri aspetti metodologici, Hungarian ha fermi sostenitori e altrettanto fermi detrattori; mentre i vantaggi delle convenzioni sono chiari, vorrei evidenziare ciò che a mio parere è il più grande difetto di Hungarian: è nata come supporto alla programmazione in un periodo storico in cui nel linguaggio C non esisteva il type checking statico. Per risolvere tale problema, la notazione usata espone direttamente l'implementazione delle variabili: se una variabile ha tipo intero, il suo identificatore inizierà per *i*; ciò aiutava il programmatore ad evitare assegnazioni errate, ma oggi è soltanto una cattiva pratica di programmazione, contraria ai principi di information hiding che permeano il C++, il quale peraltro dispone di un type checking statico che sicuramente non richiede l'esposizione dell'implementazione e lo sforzo del programmatore. Pertanto, mentre si può considerare una convenzione accettabile per il C, è ampiamente criticabile nel caso del C++: se sostituite un intero con un enumerato, o con una classe, dovrete cambiare nome a tutte le variabili che contengono tali valori.

[2]La sua diffusione è dovuta principalmente all'impiego nei sistemi di sviluppo Microsoft.

Analoga critica si può muovere alla “convenzione della ‘C‘ iniziale”: nel **Listato 8** possiamo vedere come un tipo enumerato possa essere usato esattamente come una classe: in effetti leggendo solo il corpo di *main()* non vi è modo di sapere se *Color* è un enumerato o una classe; in tal modo possiamo in seguito modificare l’implementazione di *Color* quando necessario, senza conseguenze per il codice che ne fa uso. Violare l’incapsulazione dichiarando il tipo come *TColor* o *EColor* significa seguire ciecamente una fede, più che affrontare razionalmente un problema.

Listato 8

```
enum Color { red, white, blue } ;

int main()
  {
  Color w = white ; // come un copy constructor
  Color b( blue ) ; // come uno "standard" constructor
  Color r( 0 ) ;    // come un "typecast" constructor

  return( 0 ) ;
  }
```

Se desiderate veramente adottare una convenzione di codifica, potreste utilizzare Hungarian privata del prefisso che specifica l’implementazione, o definire una vostra convenzione; nella mia esperienza, i vantaggi maggiori si hanno definendo dei *suffissi* standard per i concetti più comuni, come l’accumulazione di un totale, le dimensioni degli array, e così via, e ponendo invece come prefisso la parte realmente importante del nome, come in *sensorCode* piuttosto che *codeSensor*. Le vostre convenzioni influenzeranno anche la comprensibilità da parte di chi non le adotta: la diffusione è forse l’unico vero vantaggio di una convenzione discutibile come Hungarian. Se tuttavia il vostro business non sono le librerie di classi per i sistemi Microsoft, esistono indubbiamente alternative più moderne ed adatte al C++ di Hungarian.

Esistono infine una serie di convenzioni “tipografiche” tipiche del C (che è un linguaggio case-sensitive) che sono state in gran parte propagate all’interno del “folklore” del C++: ad esempio, le funzioni non-membro iniziano normalmente con una lettera minuscola, così come le variabili, mentre le costanti hanno spesso nomi formati da sole lettere maiuscole. Le classi e le funzioni membro iniziano normalmente con una maiuscola, anche se non si tratta di una “regola” seguita da tutti i programmatori.

Anche in questo caso, aderire a regole generalmente accettate (come quelle sopra riportate) è molto vantaggioso, se non altro perché l'abitudine vi renderà più semplice leggere il codice altrui.
Il punto fondamentale di questo paragrafo è però un altro: se lavorate come parte di un team di sviluppo, *dovete* avere una convenzione di codifica che copra almeno i seguenti aspetti:

- uso delle maiuscole e minuscole negli identificatori di variabile, costante, classe, funzione e funzione membro.

- uso o meno di prefissi o suffissi per identificare il tipo delle variabili e dei risultati di funzione (se accettate di esporre l'implementazione).

- uso o meno di prefissi o suffissi per identificare l'uso delle variabili, nei casi più comuni.

La convenzione potrà essere più o meno estesa e formale, ma è importante che sia seguita con grande coerenza, e che sia corretta se si rivela inadeguata; al di là di questo, ogni scelta specifica è troppo soggetta al giudizio ed alle sensazioni individuali per dare una risposta definitiva al problema.

Raccomandazione 10

Sviluppate ed adottate una convenzione di codifica che permetta una scelta di nomi consistente tra gli sviluppatori.

3

Strutturare i progetti in file

"...True love in this differs from gold and clay,
That to divide is not to take away."
Percy Bysshe Shelley

Ogni progetto non banale viene di norma strutturato in più file separati, con lo scopo di:

- fornire una suddivisione fisica che rispecchi la suddivisione logica del programma.
- ridurre i tempi di ricompilazione dopo le modifiche.
- incoraggiare il riuso del codice senza l'overhead dovuto a codice non usato.
- isolare le porzioni dipendenti dall'hardware o dal sistema operativo.

Purtroppo il supporto del C (e del C++) per la strutturazione di progetti in più file è piuttosto primitivo: viene supportata solo la compilazione ***separata***, non la compilazione ***indipendente*** come in altri linguaggi (es. Modula 2). Ciò significa che è compito del programmatore eseguire il link di tutti i moduli necessari, e che il compilatore non è tenuto a verificare che si stia eseguendo il link di un modulo che è stato compilato prima di aver eseguito delle modifiche ad un altro modulo condiviso. Significa anche che il programmatore deve posizionare le informazioni condivise (ad esempio la dichiarazione di classi, di funzioni e variabili globali) in un file header, per permettere agli altri moduli di accedere alle dichiarazioni stesse. In questo capitolo chiameremo "header file" i file dove le classi, le funzioni e le variabili globali vengono dichiarate, e "file di implementazione" i file dove le classi, le funzioni, le variabili globali vengono definite ed implementate.

Il primo problema che i programmatori C e C++ incontrano è in questo caso dovuto alle "inclusioni multiple", ovvero quando (ad esempio) due header che includono lo stesso terzo header sono inclusi in un altro file. In tal caso il compilatore emetterà dei messaggi di errore dovuti alla ridefinizione degli identificatori. Esistono sostanzialmente due strategie per prevenire le inclusioni multiple:

1. Non includere alcun header file all'interno di header file: ciò significa che chi include il nostro header (in un file di implementazione) deve anche essere a conoscenza di tutti gli header richiesti per una corretta compilazione. Per tale ragione si suggerisce di commentare adeguatamente il file header; resta comunque una soluzione piuttosto macchinosa per chi utilizza le nostre classi (inclusi noi stessi).

2. Utilizzare il preprocessore stesso per prevenire le definizioni e/o le inclusioni multiple; questo approccio richiede alcune linee aggiuntive di codice negli header file, ma permette un uso più diretto degli stessi. Per tale ragione, è spesso preferibile al precedente.

La tecnica che propongo di seguito persegue i due scopi di evitare le definizioni e le inclusioni multiple; in realtà si tratta di due tecniche in una, la seconda delle quali è sussidiaria alla prima ed evita anche problemi nel caso, per dimenticanza del programmatore, la prima non sia in opera[3]. Il **Listato 9** mostra un tipico esempio, con un header file di base (ovvero che non include a sua volta altri header), due header "di secondo livello" (che includono l'header base) ed infine un file di implementazione che include gli header di secondo livello:

Listato 9

```
// BASE.H

#ifndef BASE_

#define BASE_

class Base
  {
  // ...
  } ;
```

[3] Nel gergo delle compagnie di telecomunicazione, soluzioni "doppie" di questo tipo vengono spesso chiamate "cintura e bretelle".

```
#endif       // ifndef BASE_


// DERIVED1.H

#ifndef DERIVED1_

#define DERIVED1_

#infdef BASE_
  #include “base.h”
#endif

class Derived1 : public Base
  {
  // ...
  } ;

#endif       // ifndef DERIVED1_


// DERIVED2.H

#ifndef DERIVED2_

#define DERIVED2_

#infdef BASE_
  #include “base.h”
#endif


class Derived2 : public Base
  {
  // ...
  } ;

#endif       // ifndef DERIVED2_


// IMPLEMENT.CPP

#include “derived1.h”
#include “derived2.h”

// ...
```

Vediamo la tecnica nel dettaglio:

il contenuto di ogni include file è racchiuso in una "parentesi" #ifndef <*symbol*> / #endif, all'interno della quale si definisce il simbolo <*symbol*>. Per ragioni che vedremo, è molto importante che il simbolo sia algoritmicamente definito in termini del nome del file: nell'esempio, ho usato il nome del file, senza l'estensione .h, terminato con underscore '_'[4]. Questa è la tecnica principale, ed è di per sé sufficiente ad impedire le definizioni multiple.

- prima di includere un header all'interno di un altro header, si verifica che non sia già stato incluso (sapendone il nome, possiamo sapere anche quale simbolo controllare: per tale ragione è importante usare una tecnica standard per decidere l'identificatore da definire); questa seconda verifica ha due importanti conseguenze:

diminuisce i tempi di compilazione, in quanto non è necessario per il compilatore aprire nuovamente il file header incluso e processarne il contenuto (che verrebbe saltato in virtù della tecnica precedente). Questa è la ragione principale per usare la tecnica secondaria insieme alla principale.

1. nel caso il programmatore avesse commesso un errore nell'uso della tecnica principale, definendo il simbolo ma non inserendo tutte le definizioni all'interno del #ifdef/#endif, o se avesse sbagliato a scrivere il simbolo dopo #ifndef, la tecnica secondaria garantisce comunque l'assenza di definizioni multiple all'interno dell'header file.

Ovviamente non è necessario seguire strettamente la tecnica su esposta o la convenzione proposta per i simboli da definire; l'importante è fornire un metodo, il più possibile contenuto dentro gli include stessi e quindi non a carico del programmatore che usa le nostre classi, per evitare le definizioni multiple e se possibile evitare anche le inclusioni multiple, a beneficio dei tempi di compilazione.

[4] Spesso viene usato un simbolo con doppio underscore, "emulando" le librerie standard del compilatore. Questo non è corretto secondo lo standard. Per i dettagli, si veda [Pes00].

Raccomandazione 11

Ogni include file deve contenere un meccanismo che eviti le definizioni multiple, e possibilmente anche le inclusioni multiple.

Abbiamo visto che la tecnica precedente richiede di derivare un identificatore da un nome di file: questo ci porta direttamente ad un'ulteriore considerazione, ovvero come dovrebbero essere scelti i nomi dei file. Per quanto possa essere banale dirlo, il nome del file dovrebbe rappresentarne il contenuto, pur nei limiti delle eventuali restrizioni di lunghezza e set di caratteri imposte dal sistema operativo. Ciò significa che, ad esempio, se il file contiene l'header o l'implementazione di una classe il nome della classe stessa è anche il nome più adeguato per il file; se il file non contiene una classe, ma un insieme omogeneo di funzioni[5], cerchiamo la caratteristica comune delle funzioni ed usiamola per dare un nome al file. Va anche detto che, in assenza di ambienti di sviluppo sofisticati, il programmatore che vuole includere un header si trova spesso a dover "indovinare" il nome del file, noto il nome della classe: anche in questo caso, utilizzare un metodo prefissato per passare dal nome della classe al nome del file può sensibilmente ridurre il tempo perso dal programmatore, specialmente in progetti con centinaia di file.
Per la stessa ragione sarebbe sempre opportuno non definire più di una classe in un header, che è il file più comunemente utilizzato dai programmatori come "quick reference" per l'interfaccia di una classe: non è giusto richiedere al programmatore di conoscere quale agglomerato di classi è stato inserito nello stesso header file. Eventualmente si possono invece implementare classi accessorie nello stesso file di implementazione di una classe principale, anche se tale pratica non è da incoraggiare.

Il nome di un file dovrebbe essere scelto in modo da evitare possibili collisioni (l'uso del nome della classe contenuta è di norma una buona garanzia); sono ad esempio da evitare nomi come "defs.h" per definizioni locali ad una parte di un progetto, poiché rendono l'integrazione ed il riuso dei sorgenti più problematico.

Molti linker moderni sono in grado di prevenire il linking di funzioni definite ma non chiamate all'interno di un programma; in altri casi, tuttavia, il linker collega semplicemente tutti i moduli oggetto specificati, senza eseguire alcuna operazione di filtratura. Se il vostro linker è uno di essi,

[5] nel qual caso dovremmo comunque chiederci perché non sono organizzate in una classe.

potreste trovare utile la riduzione della granularità dei file: anziché inserire l'intera implementazione di una classe (o un'intera famiglia di funzioni) nello stesso file, potreste arrivare sino ad avere una funzione per file (nuovamente, è però necessaria una convenzione sui nomi di file). Ciò può ridurre sensibilmente le dimensioni dell'eseguibile, ed in molti casi anche della ricompilazione in seguito a modifiche[6].

Raccomandazione 12

Utilizzare nomi di file unici in un ampio contesto: se il file contiene l'header o l'implementazione di una classe, usare il nome della classe seguito da una estensione opportuna.

Portabilità

Anche se nello sviluppo del software poniamo il giusto accento sulla portabilità, è probabile che alcune porzioni del codice siano inerentemente legate all'architettura hardware, al sistema operativo, o al compilatore utilizzato. In questi casi, ammesso che abbia senso ipotizzare il porting dell'applicazione ad un'altra architettura (normalmente ha senso considerare almeno l'uso di un altro compilatore) esistono due strategie fondamentali, spesso utilizzate di concerto:

1. utilizzare la compilazione condizionale per *definire* un insieme di tipi e macro che permetta la compilazione su diverse architetture.

2. spostare le parti di codice non portabile in file separati.

Le due tecniche non sono totalmente interscambiabili, ed ognuna è più indicata per affrontare problemi specifici: la prima è utile quando alcuni tipi di base non siano compatibili tra le diverse architetture (un intero a 32 bit può essere un *int* su una macchina e un *long* su un'altra), mentre la seconda è utile quando un'intera funzione sia così specializzata per una architettura o un sistema operativo da rendere impraticabile l'uso della compilazione condizionale *all'interno* della funzione, se si persegue anche la chiarezza del codice. In tal caso, è molto più semplice spostare la funzione in un file separato, ed utilizzare la compilazione condizionale *all'esterno* della funzione, in pratica scrivendone una diversa versione per ogni target. Se

[6]su alcuni sistemi il linker è l'elemento più lento, ed avere più file porta invece ad un incremento dei tempi totali di compilazione e linking.

abbiamo propriamente isolato le funzioni realmente dipendenti dal sistema, tranne casi molto particolari tali funzioni saranno in numero ridotto e di lunghezza contenuta; viceversa, va seriamente considerato l'uso di una architettura software diversa, con un grado di astrazione superiore, ottenuto ad esempio con delle classi *layer* per separare le parti più astratte dell'applicazione dai dettagli dei livelli sottostanti.

Raccomandazione 13

Se il vostro progetto deve essere portabile su altre piattaforme, isolate le parti dipendenti dall'hardware e dal sistema operativo e spostatele in file separati. Considerate comunque l'opportunità di definire delle classi intermedie per isolare l'applicazione dal sistema.

Raccomandazione 14

Limitate l'uso di funzionalità specifiche del compilatore; spostate sempre le parti di codice dipendenti dal compilatore in file separati.

Ricordate infine che certi caratteri non sono ammessi su alcuni sistemi operativi come parte dei nomi di file: ad esempio, + * < > ~ | : " / \ ed in generale i caratteri con codice ASCII minore di 32 o maggiore di 127 possono non essere ammessi. Alcuni sistemi hanno limiti molto restrittivi sulla lunghezza massima di un filename, ed anche questo può creare problemi in fase di porting: se prevedete uno sviluppo multipiattaforma, cercate il massimo comun denominatore nel set di caratteri ammesso, la lunghezza minima, ed in genere le condizioni più restrittive, ed aderitevi strettamente.

Ridurre i tempi di compilazione

Anche se i compilatori moderni sono spesso molto veloci, i tempi di ricompilazione per programmi complessi possono nondimeno essere lunghi, specialmente quando si modifica l'header di una classe incluso in altri header, nel qual caso si ha spesso una esplosione combinatoria del numero di file da compilare. Ciò è particolarmente fastidioso quando si modificano solo dettagli privati della classe, non rilevanti per i moduli che la utilizzano; il problema si può affrontare sia a livello di organizzazione del codice sia a livello di design dettagliato. Vedremo qui alcuni

suggerimenti relativi al primo approccio, lasciando la discussione del secondo al capitolo 14.

Un buon modo per ridurre i tempi di compilazione è limitare l'inclusione di header in altri header ai soli casi indispensabili; consideriamo il **Listato 10**:

Listato 10

```
// DERIVED.H

#ifndef BASE_
  #include "base.h"
#endif

#ifndef PART_
  #include "part.h"
#endif

#ifndef INDIRECT_
  #include "indirect.h"
#endif

#ifndef REFERENCE_
  #include "reference.h"
#endif

#ifndef INLINE_
  #include "inline.h"
#endif

class Derived : public Base
  {
  private :
    Part part ;
    Indirect* indirect ;
    Reference& reference ;
    Inline* inl ;
  // ...
  } ;

inline int Derived :: f()
  {
  return( inl-> g() ) ;
  }
```

Abbiamo sostanzialmente cinque possibilità di utilizzo per una classe all'interno di un header file, tutte rappresentate nel listato precedente:

1. l'uso come classe base nella dichiarazione di un'altra classe.

2. l'uso diretto nella dichiarazione di una variabile (inclusi i membri dato delle classi ed i parametri di funzione), o del tipo del risultato di una funzione.

3. come (2), ma tramite puntatore.

4. come (2), ma tramite reference.

5. nel caso una delle sue funzioni membro sia usata all'interno di una funzione inline della classe definita nell'header.

Il caso (1), il caso (2) ed il caso (5) richiedono effettivamente l'inclusione dell'header che dichiara la classe all'interno dell'header che la utilizza; il caso (3) ed il caso (4), peraltro abbastanza frequenti, non impongono invece tale inclusione come necessaria. Una versione alternativa del listato precedente è infatti quella del **Listato 11**:

Listato 11

```
// DERIVED.H

#ifndef BASE_
  #include "base.h"
#endif

#ifndef PART_
  #include "part.h"
#endif

#ifndef INLINE_
  #include "inline.h"
#endif

class Indirect ;
class Reference ;



class Derived : public Base
```

```
  {
  private :
    Part part ;
    Indirect* indirect ;
    Reference& reference ;
    Inline* inl ;
  // ...
  } ;

inline int Derived :: f()
  {
  return( inl-> g() ) ;
  }
```

Per le classi utilizzate attraverso puntatori e reference, è sufficiente dichiarare l'identificatore come identificatore di una classe, evitando così di includere il file header relativo. Naturalmente, questo implica che dovremo includere i relativi file nel file implementazione della classe (*derived.cpp* nell'esempio dato), ma questo è un piccolo prezzo confrontato con i vantaggi che ne conseguono: se ad esempio il file header di *Indirect* venisse modificato, nel primo caso dovremmo ricompilare anche tutti i file che includono *derived.h*, mentre nel secondo tale ricompilazione non sarà necessaria (ma verrà comunque ricompilato *derived.cpp*); i benefici conseguenti in termini di tempo di ricompilazione sono notevoli, specialmente nel caso di grossi progetti. Notiamo che anche se la classe *Inline* è usata tramite puntatore, poiché utilizziamo una delle sue funzioni in una funzione inline di *Derived* è necessario comunque includere l'header di *Inline*.

Raccomandazione 15

Se l'accesso ad una classe all'interno di un header file avviene solo tramite reference o puntatore, non includete l'header di tale classe, ma dichiarate semplicemente l'identificatore della classe nell'header stesso.

Diversi compilatori consentono ora la precompilazione dei file header, una caratteristica che se ben utilizzata può rendere significativamente più veloce la compilazione. Tuttavia può anche essere responsabile di rallentamenti o di occupazione abnorme del disco: il sistema tipicamente utilizzato consiste nella creazione di un apposito file, dove vengono memorizzate in un formato proprietario le necessarie informazioni *per ogni sequenza di header file* utilizzata nel progetto. Normalmente non viene gestita la cancellazione di informazioni obsolete, lasciando il compito di

controllare la crescita del file di precompilazione al programmatore; notiamo che la modifica di un header incluso in molti altri può richiedere la scrittura di un notevole quantitativo di informazioni nel file di precompilazione, con conseguente rallentamento del processo di sviluppo ed aumento delle dimensioni del file.
È quindi evidente che sarebbe opportuno non precompilare gli header che sono spesso soggetti a modifiche, ed infatti i compilatori permettono di norma in un modo o nell'altro la specifica di quali header precompilare, spesso con una direttiva #pragma che permette di decidere in quale punto dell'header terminare la precompilazione.
In tal caso, un buon suggerimento può essere quello di includere prima gli header di libreria, che difficilmente varieranno nel corso dello sviluppo, poi indicare la fine della precompilazione ed infine includere gli header privati del progetto. Notiamo che il risultato è anche più strutturato di una sequenza di header di libreria e privati messi alla rinfusa; pertanto, anche se il vostro attuale compilatore non supporta gli header precompilati, sarebbe una buona pratica di programmazione organizzare l'inclusione secondo tale ordine; ricordate inoltre di usare <> per racchiudere i nomi degli header di libreria, e "" per racchiudere i nomi degli header privati: in tal modo il compilatore cercherà i primi nelle sole directory specifiche di libreria ed i secondi anche nelle directory specifiche del progetto. Ciò riduce le probabilità di un conflitto tra il nome di un file privato e di un file di libreria.

Raccomandazione 16

Se utilizzate gli header precompilati, includete gli header di libreria per primi, poi gli header più stabili, ed infine quelli più frequentemente modificati; usate il meccanismo del vostro compilatore per fermare la precompilazione dopo gli header di sistema o dopo quelli più stabili.

Raccomandazione 17

Racchiudete i nomi degli header di libreria tra <> e degli header privati tra "".

Directory

Esistono due problemi legati all'uso delle directory: come sorgente dei file header, e come destinazione dei file oggetto; il primo è ben noto, mentre il secondo passa talvolta inosservato anche ai programmatori esperti.

Quando un progetto è composto da molti moduli, spesso risultanti in diversi eseguibili e librerie a caricamento dinamico, i sorgenti dei vari moduli vengono normalmente memorizzati in directory separate, che riflettono la struttura logica del progetto. Quasi inevitabilmente, ci si trova in almeno una delle seguenti situazioni:

1. Un file posizionato in una directory deve includere un header file posizionato in un'altra.

2. Più moduli, risultanti in diversi eseguibili e librerie, condividono la stessa directory per l'output dei file oggetto.

La situazione (1) è spesso risolta specificando il percorso completo o relativo per il file da includere, all'interno del file che lo include: un esempio (da *non* seguire) è visibile nel **Listato 12**:

Listato 12

```
// DERIVED.H

#include "../library/base.h"

// ...
```

La tecnica ha due problemi ben noti: introduce una dipendenza dal sistema operativo (non esiste infatti un modo standard di specificare le directory) e vincola il codice sorgente al posizionamento fisico dei file.
Di norma, ogni compilatore, anche quelli a linea di comando, dispone della possibilità di specificare le directory in cui cercare gli include file: ciò consente di utilizzare i soli nomi dei file (senza path) dopo un #include; il lato negativo della tecnica è che si aumenta la probabilità di collisione tra nomi di file, tuttavia i vantaggi, soprattutto per la possibilità di riorganizzare la struttura delle directory senza dover modificare il codice sorgente, compensano largamente il rischio di un conflitto, che è peraltro risolubile senza troppi problemi.

Un problema meno noto è dovuto alla situazione (2) di cui sopra, nel caso i diversi moduli utilizzino diversi switch di compilazione: in tal caso, i file oggetto non sono necessariamente compatibili, ma nessuno strumento a me noto sarà in grado di tener traccia di tali incompatibilità. Utilizzando ad esempio *make*, se due progetti utilizzano la stessa directory per i file oggetto, ma diverse direttive di compilazione, compilando uno dei due si eseguirà il link con i file oggetto condivisi, così come generati dalla precedente compilazione (potenzialmente con altri switch). Se siete fortunati, le diverse direttive genereranno un name mangling diverso, ed il linker vi avvertirà di un problema di "symbol not found"; di norma, tuttavia, ciò non avviene e vi troverete ad eseguire il link, ad esempio, di codice che assume certi registri integri con codice che li utilizza per il passaggio dei parametri, o altre combinazioni che porteranno a problemi estremamente difficili da identificare. Se volete evitare il problema alla radice, non usate la stessa directory per i file oggetto di più progetti: vi troverete forse a sprecare spazio su disco, ma eviterete di sprecare il vostro tempo all'interno del debugger.

Raccomandazione 18

Non specificate le directory nei file header inclusi: utilizzate il supporto del compilatore per specificare le directory da utilizzare.

Raccomandazione 19

Utilizzate una diversa directory per i moduli oggetto di ogni progetto.

4

Struttura e Layout del Codice

"...Either write things worthy reading,
or do things worthy writing."
Benjamin Franklin

Quando scriviamo un programma, in qualunque linguaggio formale, dobbiamo perseguire due forme di comunicazione: con la macchina, attraverso il compilatore, e con altri programmatori (inclusi noi stessi), che in seguito avranno il compito di comprendere, modificare, o riutilizzare il nostro codice.

Comunicare con la macchina è un passo relativamente semplice: tutto ciò che si richiede al programmatore è una sintassi corretta. Il compilatore tradurrà il nostro codice in un eseguibile, indipendentemente dai commenti, dalla formattazione, dai nomi scelti per le variabili. Ovviamente la macchina farà esattamente ciò che abbiamo scritto -non ciò che intendevamo scrivere- ma questo è un problema ben noto. Ciò che tuttavia distingue i programmatori più esperti dai principianti è spesso la capacità di comunicare, attraverso il loro codice, anche con altri programmatori. L'uso attento dei commenti, nonché la scelta appropriata dei nomi, costituiscono due passi fondamentali verso la comunicazione tra sviluppatori; tuttavia, esistono fattori che potremmo definire puramente estetici, ma che hanno nondimeno una grande influenza sulla comprensibilità del codice.

Se avete avuto occasione di leggere o modificare codice scritto da altre persone, con uno stile di indentazione, spaziature, e convenzioni molto diverse dalle vostre, avrete già sperimentato l'effetto deleterio che piccoli dettagli "tipografici" possono avere sulla comprensione globale del codice; questo capitolo riguarda proprio questi (apparentemente irrilevanti) criteri di layout, che possono invece influire in modo consistente sul risultato di un progetto, specialmente nel caso in cui al progetto stesso lavorino più persone, anche in tempi diversi.

Purtroppo, criteri di ordine estetico sono difficili da discutere, poiché si tende rapidamente a passare dall'oggettivo al soggettivo, ed a difendere strenuamente le proprie abitudini con affermazioni del tipo "ho sempre fatto così", o "il codice di XX è scritto così" (dove XX è in genere un produttore di compilatori o l'ideatore del linguaggio), e così via. A molti sarà capitato di scivolare in "guerre sante" a proposito dello stile di indentazione migliore.

In questo capitolo, cercherò di porre l'accento su dati oggettivi, ogni volta che ciò sarà possibile; quando una decisione di layout è necessaria, ma non esistono evidenze della superiorità di una scelta, presenterò le diverse alternative e lascerò al lettore la libertà (e la responsabilità) di effettuare la selezione più adeguata per la propria organizzazione. Regole e raccomandazioni per il layout sono fondamentali per una buona riuscita di lavori di gruppo e per il futuro riutilizzo del codice: qualunque sia la vostra valutazione dei punti qui elencati, siete quindi invitati a definire un vostro insieme di princìpi, possibilmente ben fondati su argomentazioni oggettive piuttosto che basati sul semplice gusto estetico personale.

Obiettivi del layout

Secondo una classificazione degli stili di codifica data all'Università di Stanford, esistono fondamentalmente quattro livelli di "qualità comunicativa" del software:

1. Livello da pubblicazione. Ogni dettaglio sulla scelta degli identificatori, commenti, formattazione, è mirato per la migliore comunicazione possibile con il lettore. Esiste una visione ad alto livello degli algoritmi, e non vengono usate scorciatoie ineleganti ("hacks") permesse dal linguaggio. Il tipo di codice che può essere pubblicato in un libro e mostrato con orgoglio: tale perfezione è ottenibile solo con una forte determinazione sin dall'inizio dello sviluppo.

2. Livello di peer-review: in alcune compagnie esistono dei meeting tra gli sviluppatori per la revisione del codice, in modo che i propri "pari" evidenzino eventuali carenze logiche, bug, o scelte opinabili a livello di algoritmi o di dettagli. Molto spesso il codice che viene presentato e che emerge da una peer-review è molto buono dal punto di vista della leggibilità, ed è cura del programmatore evitare l'uso di tecniche che lo pongano in cattiva luce tra i suoi "pari".

3. “Nessuno guarderà mai questo codice in futuro”: uno degli stili di codifica più diffuso, soprattutto in piccoli progetti dove ogni programmatore si occupa solo del suo codice, e non esiste alcun processo per garantire una qualità ripetibile del codice stesso. Lo stile di indentazione è eterogeneo, gli identificatori non sono significativi, non esiste una visione ad alto livello degli algoritmi e spesso il codice è replicato anziché essere astratto in funzioni e classi.

4. “Come sono furbo”: il tipico prodotto del programmatore “artista” che ama utilizzare i costrutti meno comprensibili, talvolta alla ricerca di una minima ottimizzazione, talvolta per il semplice piacere di farlo. Le “prestazioni” sono normalmente portate come giustificazione di uno stile che non è chiaro per nessuno, neppure per chi lo ha scritto. La manutenzione in genere richiede un costo pari a quello del ri-sviluppo.

Molti di noi sarebbero felici se tutto il codice fosse al “livello di pubblicazione” (e molti altri se fosse al livello “come sono furbo”), anche se ogni programmatore con un minimo di esperienza sa che in progetti reali, con i tempi di sviluppo ristretti che normalmente li accompagnano, raggiungere un livello di “peer-review” è già da considerarsi un successo. È auspicabile che le pressioni esercitate da organismi come ISO o il SEI riescano nel tempo a spostare il livello standard del software verso i livelli più alti (non a caso, esiste una relazione molto stretta tra i “gradi di qualità” visti sopra e gli indici di Capability Maturity Model [Pau93] del Software Engineering Institute); indubbiamente, chi legge questo libro si trova o desidera trovarsi ad uno dei gradini più alti della scala su riportata.

Notiamo che al livello 1 un programma è visto come una pubblicazione, e come tale riconosce una fondamentale importanza al layout; come può un accurato layout del codice rendere il codice stesso più comprensibile? Osservate il frammento di codice del **Listato 13**, estratto da un progetto reale e scritto da un programmatore piuttosto esperto: pochi lo definirebbero tuttavia chiaro, comprensibile e semplice da modificare.

Listato 13

```
BOOL TranslationTable :: OpenFiles()
{
struct stat intstat;
struct stat ndxstat;
BOOL Rebuilt = FALSE;
result = FALSE;
```

```
if (NdxFile != HFILE_ERROR || IntFile != HFILE_ERROR)
return TRUE; // already open
ChangeExtension(Filename,"INT");
IntFile = NdxFile = HFILE_ERROR;
IntFile = _lopen(Filename,READ | OF_SHARE_DENY_NONE );
if (IntFile == HFILE_ERROR) return FALSE;
  // can't open .INT file
stat(Filename,&intstat);
struct ftime intTime ;
getftime( IntFile, &intTime ) ;

ChangeExtension(Filename,"NDX");
if (access(Filename,0)) {
            RebuildNdx( &intTime );
            Rebuilt = TRUE;
            }
NdxFile = _lopen(Filename,READ | OF_SHARE_DENY_NONE );
stat(Filename,&ndxstat);
if (ndxstat.st_ctime < intstat.st_ctime) {
            _lclose(NdxFile);
            NdxFile = HFILE_ERROR;
            unlink(Filename);
            if (!Rebuilt) RebuildNdx( &intTime );
            NdxFile=

_lopen(Filename,READ|OF_SHARE_DENY_NONE );
            if (NdxFile == HFILE_ERROR) {
                  _lclose(IntFile);
                  IntFile = HFILE_ERROR;
                  return FALSE;
                  // can't open .NDX file
                  }
            }
IdxCache->SetFiles(NdxFile,IntFile);
return( TRUE ) ;
}
```

In effetti, il programmatore che lo ha scritto (che spero non si offenderà nel leggere il mio commento) si preoccupava più dell'efficienza del codice, e della velocità con cui riusciva a scriverlo, che della facilità di comprensione da parte di altri programmatori. Ciò nonostante, i nomi di variabili sono scelti con una certa cura, e la relativa semplicità del codice potrebbe anche giustificare la carenza di commenti; è semplicemente il layout, o meglio l'assenza di qualunque criterio di layout, a rendere il codice più difficile da leggere di quanto in realtà non sia. Ovviamente, un listato scritto ad arte potrebbe essere *molto* più complesso da leggere; ho tuttavia preferito

utilizzare un frammento di codice reale, proprio per evidenziare la concretezza del problema.

Nel corso degli anni, molti studi empirici e psicologici [SE88], [SBE83], [Sch76], [She81], [SKC81], [KS81] hanno affrontato problemi quali la giusta indentazione, l'uso delle spaziature, la rappresentazione dei blocchi applicativi, e così via; tali studi sono spesso riferiti al tempo di debugging, o alla capacità di un programmatore di comprendere il codice scritto da un altro, e sono pertanto estremamente importanti per gli obiettivi di questo testo. Esiste tuttavia anche un altro punto di vista da tenere in considerazione: la visione di chi deve scrivere il programma. Un buon programmatore dovrebbe essere abbastanza flessibile da modificare il suo stile di codifica, se vi sono prove concrete che un approccio differente sia effettivamente migliore, ma è necessario in ogni caso cercare di definire regole semplici da ricordare e da utilizzare. In altre parole, uno stile troppo macchinoso da utilizzare o da mantenere durante le modifiche *non* verrà seguito dai programmatori perché, al di là di studi accademici, nel mondo reale i tempi di sviluppo sono spesso troppo brevi: solo regole abbastanza semplici da apprendere e che non richiedano tempo extra in fase di codifica verranno effettivamente rispettate. In effetti, un insieme di regole coerente e facile da mettere in atto passerà presto a livello inconscio, e non costituirà un carico extra per gli sviluppatori.

Possiamo ora introdurre alcuni princìpi che definiscono il fine ultimo di un buon layout:

- La struttura fisica del codice dovrebbe ricalcare la struttura logica del codice stesso. L'indentazione e l'uso degli spazi, ad esempio, dovrebbero essere tali da indicare esattamente il flusso del controllo, le dipendenze, le relazioni.

- Strutture logiche simili dovrebbero essere rappresentate in modi simili: in altre parole, occorre perseguire la massima consistenza possibile.

- La leggibilità del codice è più importante dell'estetica del codice. Non necessariamente ciò che si presenta meglio è anche più comprensibile: potrebbe distrarre l'occhio verso dettagli irrilevanti, anziché evidenziare le parti fondamentali.

- La facilità di manutenzione del codice è più importante dell'estetica del codice: modificare una porzione del codice non dovrebbe richiedere riaggiustamenti di altre parti per mantenere la consistenza.

- Le norme di layout dovrebbero essere semplici da ricordare e da applicare; con un minimo di pratica, non dovrebbe essere necessario alcuno sforzo conscio per rispettare le regole: dovrebbero cioè diventare parte integrante del nostro modo di scrivere il codice.

Macro-layout

"Un programma è come una pubblicazione": se questo è il nostro obiettivo, anche se ideale, dovremmo prima osservare come è strutturata una pubblicazione. Innanzitutto noteremmo che vi sono spazi bianchi fra le parole: ciò può sembrare ovvio e scontato, ma non di rado le spaziature tra i caratteri sono poco omogenee od assenti all'interno del codice: *x=y+z->f();*. Il secondo elemento che contraddistingue un testo ben scritto è l'uso di linee bianche per separare fisicamente, anche all'interno dello stesso capitolo o paragrafo, i periodi che sono logicamente separati. Raramente si fa uso di cornicette o di elementi vistosi, se non nei casi in cui si voglia focalizzare l'attenzione su alcune righe fondamentali (come le "raccomandazioni" in questo testo); anche in questo caso, il parallelo con il codice ci indica che l'uso di "cornici di commento" può essere eccessivo rispetto ad una semplice spaziatura tra le righe. Infine, quando una serie di punti va letta come subordinata ad una frase principale (vedere ad esempio i "princìpi di buon layout" poco sopra) essi sono indentati rispetto alla frase principale, e sono tra essi allineati.

Come vedete, gli elementi tipografici essenziali per una pubblicazione sono di pari importanza anche per il testo di un programma: spaziature, linee di separazione, indentazione ed allineamento. Tuttavia, prima di discutere questi elementi di micro-layout, è necessario osservare che esiste anche un livello di layout più astratto, che potremmo chiamare "layout architetturale": un libro è diviso in capitoli, ognuno dei quali ha un titolo, un numero, ed una sequenza logica di paragrafi (introduzione, sviluppi, conclusioni), a sua volta ogni paragrafo è strutturato utilizzando linee vuote per separare i periodi, e così via. Chi legge un libro sa che se vuole trovare l'introduzione ad un capitolo dovrà cercare all'inizio del capitolo: nuovamente, se questa osservazione vi sembra ovvia, dove dovremmo cercare il distruttore della classe nei vostri file di implementazione? Ed in quelli dei vostri colleghi? E nelle librerie che usate più spesso?

Chi ritiene che un programma di "pretty printing" o un "code beautifer" sia la panacea di tutti i mali di layout pecca purtroppo di inesperienza. Da un

lato, alcuni elementi del layout, come la separazione ed il raggruppamento degli statement logicamente correlati, riguardano aspetti semantici che non potranno mai essere inseriti automaticamente da un programma, informazioni che solo un programmatore attento può trasmettere a chi legge il codice. Dall'altro, uno dei momenti in cui si desidera disperatamente di avere un codice indentato in modo più consono alle proprie abitudini è quando, durante la manutenzione di codice altrui, il codice stesso non può neppure essere compilato per la presenza di errori sintattici. Molto spesso, i code beautifer assumono che il codice sia sintatticamente corretto e hanno comportamenti singolari quando tale assunzione viene disattesa.

Purtroppo l'idea di raggiungere uno standard universale per il layout architetturale dei programmi, così come esiste oggi per i libri, le riviste, gli articoli, è un'utopia che non possiamo neppure immaginare di perseguire. Nulla ci impedisce, tuttavia, di esercitare tale controllo e realizzare tale coerenza all'interno dei nostri programmi, a beneficio nostro e di chi leggerà il nostro codice. Esistono ovviamente numerose scelte possibili per il layout strutturale, ed anche nel caso dei programmi le opzioni si propagano ai diversi livelli; in quanto segue, esamineremo le varie scelte, talvolta suggerendo un possibile stile, che potrete usare come base per un vostro standard personale (ovunque, le parentesi quadre indicheranno un elemento opzionale):

Separazioni

Come abbiamo visto poco sopra, una delle caratteristiche di un buon layout è quella di raggruppare linee di codice logicamente correlate, e separare quelle logicamente distinte; notiamo che di norma, anche all'interno di singole funzioni esistono partizioni logiche (come inizializzazione, allocazioni dinamiche, corpo, rilascio delle risorse) esattamente come il singolo paragrafo di un libro è suddiviso in periodi. Non è tuttavia insolito che i programmatori si dilettino in suddivisioni "a cornicette", come nel **Listato 14**:

Listato 14

```
/*******************************************************/
/        COSTRUTTORE                                    /
/*******************************************************/
X :: X( int n )
  {
/*******************************************************/
/      INIZIALIZZA CAMPI STATICI                        /
/*******************************************************/
```

```
  len = n ;
  current = 0 ;
/*****************************************************/
/       INIZIALIZZA CAMPI DINAMICI                    /
/*****************************************************/
  buffer = new int[ len ] ;
  }
```

L'effetto negativo di tali separazioni è che tendono ad attirare l'attenzione su elementi marginali del programma, distraendo invece dal codice vero e proprio; certamente, leggendo solo le parti incorniciate si ha una visione ad alto livello di cosa avviene all'interno del programma, ed è altresì possibile utilizzare un semplice tool per ricavare la "documentazione" di ogni funzione. Tuttavia lo stesso scopo può essere raggiunto in modo meno invasivo, come nel **Listato 15**:

Listato 15

```
// Costruttore

X :: X( int n )
  {
  // Inizializza campi statici
  len = n ;
  current = 0 ;

  // Inizializza campi dinamici
  buffer = new int[ len ] ;
  }
```

L'uso appropriato di linee bianche consente una più agevole lettura del codice: in fondo, non credo vorreste che i periodi di un articolo o di un libro fossero racchiusi tra cornicette. Tra l'altro, molti editor guidati dalla sintassi consentono una colorazione diversa per i commenti, che se usata accuratamente può, se lo si deridera, porli comunque in evidenza senza arrivare agli eccessi.

Raccomandazione 20

Utilizzate linee vuote per separare elementi logicamente distinti, anche all'interno della singola funzione o blocco applicativo. Cercate di limitare gli elementi decorativi nei commenti, che deviano l'attenzione di chi legge.

Indentazione

Un modo per evidenziare le dipendenze logiche tra le varie linee è quello di indentare le linee subordinate rispetto alle linee principali, ad esempio il corpo di una funzione rispetto al prototipo della stessa, o la clausola di un *if* rispetto alla condizione. L'indentazione è un elemento fondamentale per la corretta comprensione dei programmi; provate a leggere il **Listato 16**, ed a prevederne l'output:

Listato 16

```
int f( int x, int y, int z )
  {
  if( x >= 0 )
    if( y > 0 ) z = 1 ;
  else if( y > 0 ) z = 2 ;

  return( z ) ;
  }

int main()
  {
  printf( "%d\n", f( -1, 2, 0 ) ) ;

  return( 0 ) ;
  }
```

La risposta corretta è che viene stampato '0', ma è tutt'altro che raro che chi legge si faccia fuorviare dall'indentazione e risponda che viene stampato '2'.

In generale, gli statement andrebbero indentati *sotto* a quelli cui sono logicamente subordinati: in tal senso una buona indentazione del listato precedente potrebbe essere:

Listato 17

```
int f( int x, int y, int z )
  {
  if( x >= 0 )
    if( y > 0 )
      z = 1 ;
    else if( y > 0 )
      z = 2 ;

  return( z ) ;
  }


int main()
  {
  printf( "%d\n", f( -1, 2, 0 ) ) ;

  return( 0 ) ;
  }
```

difficilmente la forma correttamente indentata trarrà in inganno il programmatore.

Al di là degli esempi un po' accademici, di norma ogni buon programmatore indenta il suo codice in modo corretto (anche se il listato che abbiamo visto all'inizio del capitolo dimostra che ciò non sempre avviene). Il vero problema in un team di lavoro è l'uso di stili di indentazione diversi da parte dei singoli programmatori: le tensioni tendono ad essere esplosive, specialmente quando all'interno dello stesso modulo, in seguito a manutenzione, si cominciano ad avere stili di indentazione diversi.
Esistono molti stili di indentazione, spesso differenti in minimi dettagli, ma i due più diffusi sono rappresentati nel **Listato 18**:

Listato 18

```
// Stile "Kernighan & Ritchie
while( condition ) {
  body
  }

// Stile "Pascal"
while( condition )
  {
  body
  }
```

Non c'è dubbio che chi è abituato ad uno dei due faccia una certa fatica a leggere l'altro, soprattutto in presenza di molti blocchi annidati; per tale ragione, è importante che all'interno di un gruppo di lavoro si scelga uno stile consistente, anche a costo di richiedere uno sforzo di adattamento ad alcuni programmatori.

Lo stile "K&R" ha due vantaggi sullo stile "Pascal":

1. si risparmia una riga

2. é uno degli stili più diffusi tra i programmatori C

Per contro, lo stile "Pascal" per molti risulta più leggibile (grazie alla riga in più) ed ha l'innegabile vantaggio di permettere un rapido matching delle parentesi aperte e chiuse, poiché si trovano sulla stessa colonna.

Nella mia esperienza, ho notato che chi usa lo stile "K&R" riesce a leggere senza troppi problemi codice indentato "alla Pascal", mentre non è vero il viceversa. Ciò significa o che i programmatori più svegli usano lo stile "K&R" o che lo stile "Pascal" è inerentemente più leggibile, ed io propendo pesantemente per la seconda ipotesi. Trattandosi però di un argomento ai confini con la fede, la scelta deve essere vostra: l'importante è avere una convenzione effettivamente rispettata da tutti i membri del team.

Raccomandazione 21

Indentate gli statement subordinati *sotto* agli statement da cui dipendono.

Raccomandazione 22

Definite uno standard di indentazione che sia rispettato da ogni membro del team di sviluppo.

Ovviamente, uno standard di layout non può limitarsi allo stile di indentazione: per essere completo, dovrebbe coprire anche altre aree, ad esempio come gestire liste di parametri troppo lunghe per stare in una riga (tra l'altro, dovrebbe anche stabilire la lunghezza massima di una riga). Vedremo in seguito alcune considerazioni di micro-layout, pur senza pretendere di essere esaustivi, in quanto il soggetto richiederebbe un intero volume e risulterebbe più noioso che utile; in ogni caso, uno degli aspetti

più importanti del layout è la consistenza: meglio una norma discutibile, ma applicata ovunque, che il caos totale. È infatti più facile adattarsi a qualcosa che non apprezziamo che adattarsi a variazioni continue: se proprio volete usare uno stile personale, siate almeno consistenti nel suo uso, a beneficio di chi dovrà leggere il vostro codice.

Tabulazioni

Ho ritenuto necessario spendere alcune parole sull'uso del tab, per quanto possa sembrare un elemento di dettaglio irrilevante, poiché in un grosso progetto al quale ho partecipato "la lunghezza del tab" è stata motivo di tensioni fortissime, con minacce di licenziamenti e discussioni interminabili. Anche in questo caso, il problema di fondo era la mancanza di uno standard accettato da tutti: alcuni programmatori definivano il tab come otto spazi, altri due, altri tre; inoltre, alcuni indentavano usando *solo* il tab, altri usavano liberamente il tab e gli spazi. Ovviamente, provando a caricare nel proprio editor il codice scritto da un altro programmatore, raramente ci si trovava in condizioni di leggibilità: linee sconnesse, parametri fuori dal video, disallineamenti vari erano la regola anziché l'eccezione; il problema è peggiorato quando alcuni file sono stati modificati da programmatori che usavano uno stile diverso, propagando di fatto il problema stesso dal livello del file al livello della singola funzione o blocco. La causa della tensione era che nessun programmatore si dimostrava disposto a cedere sul terreno del "tab": un classico esempio di come una guerra santa possa pregiudicare la riuscita di un progetto. Possiamo vedere rapidamente le ragioni di ogni "fazione"

tab corti (2 o 3 spazi), indentazione con soli tab

1. Posso indentare con il solo tab, ed è più rapido premere il tab che due o tre spazi.

2. Il codice indentato con soli tab può essere letto sia da chi vuole tab = 2 spazi o tab = 3 spazi, perché rimane comunque strutturato e leggibile.

tab di default (8 spazi), indentazione con tab o spazi (2)

1. È più rapido premere due volte la barra che una volta il tab, perché non devo muovere le mani dalla tastiera.

2. Posso usare i tool a linea di comando, la redirezione, il pipe, stampare direttamente il file, eccetera, senza utilizzare necessariamente programmi che permettano di "personalizzare" il tab. Il default più comune per il tab di ogni device (schermo, stampante, ecc) è 8 spazi.

Come vedete, non solo le motivazioni (1) di entrambe le fazioni sono contraddittorie (ed è ragionevole, riguardando elementi estremamente e giustamente personali) ma si tratta di convinzioni difficili da controbilanciare. Personalmente, ritengo che la giustificazione (2) di chi usa il tab di default sia di gran lunga più oggettiva e rilevante di tutte le altre, e per tale ragione il mio giudizio pendette a suo tempo per la fazione del "tab di default". Un'ottima alternativa potrebbe essere quella di usare un editor che espande i tab in spazi, cosicché il codice rimanga identico indipendentemente dalla definizione in uso per i tab; ciò sarebbe comunque in disaccordo con la motivazione (2) della fazione dei tab corti, ma le due posizioni sono comunque inconciliabili.

In conclusione, mentre mi auguro che una simile diatriba non si ripeta nei vostri (e nei miei) gruppi di lavoro, è necessario che venga stabilita una convenzione interna sull'uso dei tab, e che venga rispettata da ogni programmatore. È meglio risolvere le dispute all'inizio di un progetto, che trovarsi nel mezzo di una guerra civile a sviluppo inoltrato.

Raccomandazione 23

Definite uno standard interno sulla lunghezza e l'uso del tab che sia rispettato da tutti i programmatori del team.

File header

I file header sono un elemento estremamente importante di un programma: nel caso di una libreria, dal punto di vista del layout sono più importanti del file di implementazione stesso. Ai file header si fa infatti riferimento con una frequenza piuttosto elevata, in tutti i casi in cui non esiste una documentazione accurata e aggiornata dell'interfaccia di una classe o dell'insieme di funzioni implementate da un modulo. Sarebbe pertanto auspicabile che i file header avessero una struttura omogenea, che faciliti la ricerca di un qualunque elemento (es. un tipo enumerato) all'interno di esso. Un possibile macro-layout per i file header è il seguente:

Nome del file (commento)

Breve descrizione del contenuto (commento)

[Include di altri header]

[Macro costanti]

[Macro funzioni]

[Typedef]

[Enum]

[Classe]

[Funzioni inline]

[Const extern]

[Variabili extern]

[Funzioni extern]

[Operatori extern]

Esistono ovviamente delle considerazioni di tipo logico alla base del layout strutturale proposto: in particolare, l'idea di base è stata quella di trovare un layout che fosse possibile rispettare nella maggior parte dei casi. Supponiamo infatti di decidere che i tipi enumerati debbano apparire per ultimi: in tal caso non potremmo utilizzarli come parametri o tipi di ritorno delle funzioni (membro o meno), o per dichiarare le variabili esterne. Parimenti se le classi venissero per ultime non potremmo definire degli operatori che le utilizzano. Il layout proposto si basa sulle seguenti considerazioni, normalmente valide:

- le macro costanti sono in genere indipendenti dagli altri elementi.
- le macro funzioni al più usano le macro costanti.
- i typedef sono indipendenti o usano dei simboli definiti dalle macro.

- gli enum sono normalmente autonomi.
- le classi usano quanto sopra, ma nella parte dichiarativa non usano quanto segue.
- se una classe dichiara funzioni inline, queste dovranno seguire la dichiarazione (vedi oltre).
- le costanti extern potrebbero avere come tipo una classe precedentemente dichiarata.
- lo stesso per le variabili extern.
- lo stesso per le funzioni extern, con riferimento al tipo del risultato o dei parametri; inoltre un parametro default può fare riferimento al nome di una variabile o costante extern.
- analogamente per gli operatori.

Notiamo che vi è un discreto margine soggettivo, ad esempio nell'ordine tra funzioni ed operatori, ma se provate a considerare dei layout alternativi vi accorgerete che non esistono possibilità radicalmente diverse. Ovviamente, qualunque struttura voi scegliate, talvolta potreste trovarvi a violare le vostre stesse regole; ciò non dovrebbe preoccupare, a meno che non accada così di frequente da suggerire che lo schema utilizzato non sia ben fondato. Se (ad esempio) un tipo enumerato va necessariamente inserito dopo la dichiarazione di una classe, mettete un commento che indichi a chi legge dove trovarlo. Notiamo che la classe va vista, a questo livello di macro-layout, come un blocco monolitico; vedremo in seguito come organizzare il micro-layout interno di una dichiarazione di classe.

Vi sono ancora alcuni punti che meritano un breve commento: è sempre opportuno scrivere il nome del file come parte del file stesso, in modo da evitare confusione con le copie stampate; ciò è particolarmente vero sui sistemi che impongono limiti molto stretti sul numero di caratteri ammessi in un nome di file. Analogamente, un breve commento esplicativo sul contenuto del file non richiederà molto del vostro tempo, e potrà semplificare la ricerca di una classe tramite strumenti come *grep*, quando sia noto il problema da risolvere ma non la classe che lo risolve: un caso

abbastanza frequente quando si abbiano librerie di classi molto vaste, non organizzate in sottoinsiemi omogenei.

Infine, ho implicitamente suggerito di spostare la definizione delle funzioni inline fuori dalla dichiarazione della classe: riprenderemo questo punto parlando del micro-layout di una dichiarazione di classe.

Raccomandazione 24

Definite uno standard per la sequenza delle dichiarazioni in un file header, ed aderitevi strettamente ogni volta che sia possibile.

File implementazione

Avendo definito un macro layout per i file header, il macro-layout più razionale per un file di implementazione segue in modo immediato: considerando che in pratica ogni file di implementazione è associato ad un header, dovremmo cercare di mantenere *esattamente* lo stesso layout, con l'ovvia differenza che nel file di implementazione troveremo anche i body delle funzioni. Ciò consentirà, avendo a disposizione il file header, di posizionarci "a colpo sicuro" all'interno del file di implementazione; inoltre, accoppiata alle regole di micro-layout per la dichiarazione delle classi, questa tecnica ci garantirà una manutenzione coerente del file di implementazione stesso.
Come vedremo, le regole di micro-layout per la dichiarazione delle classi ci aiuteranno a rendere l'header di più immediata lettura, e seguire lo stesso ordine all'interno del file di implementazione risulterà in una notevole coerenza tra i vari file: in genere, troveremo i costruttori all'inizio del file di implementazione, poi il distruttore, e così via: se vi siete mai sentiti "spaesati" cercando le varie funzioni all'interno di un file scritto da un vostro collega, riuscirete presto ad apprezzare i vantaggi che una sistematica coerenza tra il file header ed il file implementazione è in grado di darvi.
Due precisazioni sono comunque necessarie:

1. ovviamente, all'inizio del file di implementazione dovremo includere tutti gli header necessari, tra cui il file header associato; da questo punto, cercheremo di definire i vari elementi nello stesso ordine in cui sono stati dichiarati nel file header.

2. in genere il modulo "main" non ha un file header associato; in questo modulo, solitamente si ha solo la funzione main o poche altre funzioni accessorie o variabili globali. In tal caso, è importante rispettare lo stesso ordine dei file header (es. prima gli include, poi le variabili, poi le funzioni) e definire una posizione standard per la funzione *main*() (es. alla fine).

Raccomandazione 25

Il file di implementazione deve definire ogni elemento nell'esatto ordine di dichiarazione del file header associato.

Micro-layout

In questo paragrafo ci occuperemo del layout dei singoli elementi, come le dichiarazioni di variabile e di classe, le espressioni, gli statement condizionali, i cicli e così via. La sezione fondamentale è probabilmente quella relativa al layout della dichiarazione di classe, in quanto ha una influenza profonda sul layout complessivo del file header e quindi, per quanto visto, del file di implementazione.

Espressioni

Layout di un'espressione significa principalmente buona spaziatura; anche in questo caso, esistono certamente programmatori che non usano alcuna spaziatura all'interno delle espressioni, così come programmatori che tentano di usare le spaziature a scopi semantici. Entrambi gli stili hanno alcuni problemi, il primo di leggibilità vera e propria, sul quale non vale neppure la pena di insistere, mentre il secondo rischia di fornire al compilatore ed al lettore umano due informazioni diverse: vedere **Listato 19**.

Listato 19

```
x =  a+b * c+d ;
```

Chi legge è spesso ingannato e assume che l'espressione destra venga valutata come *(a+b)*(c+d)*, mentre ovviamente viene valutata come *a+(b*c)+d*. In tal senso, la spaziatura non riflette la struttura logica del codice: si tratta ovviamente di un errore involontario, tuttavia non sarebbe raro introdurlo in fase di manutenzione. È pertanto meglio usare sempre una spaziatura uniforme; in realtà, sarebbe opportuno avere una

convenzione di spaziatura anche sui singoli operatori: molti trovano che gli operatori . e -> a differenza degli altri operatori binari, siano più leggibili se *non* vengono separati dai loro operandi, e che gli operatori [] e () debbano essere attaccati all'operando di sinistra ma non a quelli di destra. Nuovamente, si tratta di un argomento troppo soggettivo per fornire una risposta definitiva in questa sede.

Raccomandazione 26

Usare una spaziatura uniforme tra gli operatori e gli operandi; vi sono sicuramente eccezioni, ma anche su queste cercate di essere consistenti con un vostro standard di layout.

Nel caso di espressioni complesse, non è insolito ritrovarsi con un elevato numero di parentesi, anche perché molti seguono il principio che "qualche parentesi in più non può far male". Se ciò è vero per il compilatore, non altrettanto si può dire degli esseri umani, che non sono affatto bravi ad eseguire il matching di parentesi: vedere il **Listato 20** per un piccolo esempio.

Listato 20

```
x = ((a+(b*(c+(d-3)+f)-d))) ;
// cosa moltiplica 'b'?

x = a + b * ( c + d - 3 + f ) - d ;
// molto più chiaro...
```

Tuttavia, ogniqualvolta l'associatività o l'ordine di valutazione non siano ovvii, è opportuno introdurre delle parentesi, in modo da rendere chiaro a chi legge il significato del codice; non assumete che chi legge il vostro codice conosca perfettamente la priorità e la associatività degli operatori, tranne che nei casi banali (come somma e moltiplicazione). Se non troppo nidificate, le parentesi possono rendere il codice effettivamente più semplice da capire, come nel **Listato 21**:

Listato 21

```
// versione "complicata" da leggere correttamente
if(d * e < f || g + h < i  && a + b < c)
  x = 0 ;

// versione di comprensione corretta più immediata
```

```
if( ( d * e < f ) || (( g + h < i ) && ( a + b < c )) )
  x = 0 ;
```

In realtà, tuttavia, di fronte ad espressioni booleane di una certa complessità la scelta migliore è semplicemente quella di spezzare l'espressione, assegnando il risultato di una sottoespressione significativa ad una variabile. A meno di non utilizzare un compilatore realmente scadente, non dovrebbero esserci variazioni significative nel codice generato. Il **Listato 22** mostra una possibile ristrutturazione del codice visto nel listato precedente.

Listato 22

```
BOOL inside = ( d * e < f ) || ( g + h < i ) ;
BOOL onTop = a + b < c ;
if( inside && onTop  )
  x = 0 ;
```

Notiamo come il codice non solo sia più leggibile, ma come siano anche presenti maggiori informazioni su ciò che il test dovrebbe verificare, informazioni che altrimenti avrebbero dovuto essere associate sotto forma di commento.

Ricordate in particolare che in C++ gli operatori hanno una gerarchia di priorità piuttosto articolata (per una tabella riassuntiva, potete consultare [Str97]. In particolare, se un'espressione utilizza sia operazioni su interi o float che operazioni bit-wise, è sempre opportuno specificare le parentesi: vedere il **Listato 23** per un esempio di possibile ambiguità nella lettura. Alcuni compilatori emettono un warning se questa regola non viene rispettata.

Listato 23

```
if( a & b < c )  // significa a&(b<c), non (a&b)<c
  x = 0 ;

if( a < b < c )  // significa (a<b)<c
  y = 0 ;
```

Ricordiamo infine che quando si effettua l'overloading di un operatore, l'associatività e la priorità dell'operatore stesso rimangono immutate. Pertanto, se la semantica standard dell'operatore è quella di un operatore bit-wise, come nel caso di << e >> (utilizzati anche per l'input/output su

stream), avremo in ogni caso una priorità alta associata all'operatore. Un errore tipico nell'uso degli stream in C++ è evidenziato nel **Listato 24**:

Listato 24

```
cout << a < b ;
// errore: interpretato come ( cout << a ) < b ;
```

Ciò sottolinea ancora di più l'importanza di utilizzare le parentesi quando si utilizzano operatori bit-wise, anche se overloaded.

Raccomandazione 27

Utilizzate le parentesi solo quando sono necessarie ai fini semantici, o per chiarire la priorità e/o l'associatività degli operatori.

Raccomandazione 28

Spezzate le espressioni troppo complesse, in particolar modo le espressioni booleane, assegnandone una sottoespressione significativa ad una variabile locale.

Raccomandazione 29

Utilizzate sempre le parentesi per chiarire il significato di espressioni che coinvolgono operatori bit-wise ed altri operatori. Questo è valido anche quando gli operatori bit-wise sono overloaded, come nel caso degli operatori << e >> per l'I/O su stream.

Funzioni

Nel caso delle funzioni abbiamo due importanti argomenti di layout: uno riguardante i parametri formali, ed uno riguardante la gestione di una lunga lista di argomenti.

In C++, i nomi dei parametri formali non devono essere necessariamente specificati nel prototipo della funzione; inoltre, i parametri formali non devono necessariamente avere gli stessi nomi se il prototipo viene ripetuto. Ciò è particolarmente rilevante per le funzioni che vengono dichiarate in un

header e definite in un file di implementazione: al riguardo, ho avuto modo di incontrare tre stili di codifica:

1. Nell'header non viene specificato il nome del parametro, ma solo il tipo; nell'implementazione viene specificato anche un nome.

2. Nell'header viene usato un nome lungo e descrittivo, nell'implementazione uno breve.

3. Sia nell'header che nell'implementazione viene usato lo stesso nome per i parametri formali.

Lo stile (1) è abbastanza inconsueto, per quanto abbia avuto modo di vederlo utilizzato nelle librerie di alcuni compilatori commerciali. Esistono certamente dei casi in cui il nome del parametro è abbastanza irrilevante all'interno dell'header (*sqrt(int)* non è meno chiaro di *sqrt(int x)*), tuttavia nella maggior parte dei casi il nome dei parametri formali è un elemento importante per la comprensibilità del codice. L'unico programmatore ad utilizzare tale stile che ho incontrato personalmente, ha portato come unica giustificazione: "in tal modo quando cambio il nome del parametro nell'implementazione non devo cambiarlo nell'header". Si tratta a mio parere di una giustificazione peregrina, in quanto il nome del parametro è un completamento dell'interfaccia astratta della funzione, che difficilmente cambierà durante il tempo; è più ragionevole che cambi il tipo, nel qual caso bisogna comunque mettere mano al file header. Ritengo che lo stile (1) sia più che altro motivato dalla pigrizia del programmatore.

Lo stile (2) è in un certo senso l'opposto di (1): in tal caso, il programmatore assume (giustamente) che il file header sia un elemento importante nella documentazione del codice, a cui altri programmatori faranno riferimento; pertanto, usa nomi lunghi e descrittivi al suo interno. Il file di implementazione viene visto come "privato" e nomi brevi vengono usati per "risparmiare tempo". Per quanto sia più giustificabile di (1), anche questa motivazione non sembra molto convincente: se un parametro formale appare un paio di volte nel body della funzione, non si risparmierà molto tempo a scriverne uno più corto; anzi, se si dedica un po' di tempo alla ricerca di una abbreviazione non ambigua, probabilmente non si risparmierà affatto. D'altra parte, se compare molte volte nel body della funzione allora il body è probabilmente molto lungo, il che significa che lo scope e la lifetime del parametro sono piuttosto estesi, ed in accordo a quanto visto nel capitolo 2 dovremmo comunque usare un nome lungo e

descrittivo. In conclusione, sembra nuovamente che la regola sia dettata dalla pigrizia più che da motivazioni di effettivo risparmio di tempo.

Lo stile (3) è probabilmente il più semplice da mettere in atto, non richiede lo sforzo di inventare abbreviazioni comprensibili, porta ad una perfetta consistenza tra l'header e l'implementazione, con il solo svantaggio che modificare l'uno richiede la modifica dell'altro: un prezzo che statisticamente pagheremo di rado. Si tratta pertanto dello stile più consigliabile in ogni situazione.

Raccomandazione 30

Specificare il nome dei parametri formali sia nel file header che nel file di implementazione, ed utilizzare gli stessi identificatori in entrambi i contesti.

Esiste una possibile eccezione al caso precedente, ovvero quando i parametri non sono utilizzati; ciò accade abbastanza di frequente con i metodi virtuali, che talvolta nelle classi base non utilizzano alcuni dei parametri: in tal caso, certi compilatori emettono un warning di "parametro non utilizzato".
Alcuni programmatori usano un #pragma (non portabile) per eliminare il warning[7], altri non mettono il nome dei parametri non utilizzati nel file implementazione per eliminare il warning in modo portabile. Il **Listato 25** mostra anche la soluzione suggerita: racchiudere in un commento il nome dei parametri non utilizzati; in tal modo, si ha comunque la massima portabilità, leggibilità, consistenza, e mantenibilità.

Listato 25

```
// BASE.H

class Base
  {
  virtual void f( int size ) ;
  // ...
  }
```

[7] Ciò impedisce anche al compilatore di segnalare il mancato uso di altri parametri che (forse) dovevano essere usati. Il #pragma non è di norma selettivo sui parametri.

```
// BASE.CPP

void Base :: f( int /* size */ )
  {
  }
```

Raccomandazione 31

Se un parametro formale non è utilizzato, racchiudere il suo nome in un commento nel file implementazione.

Infine, sia nel punto di dichiarazione che (più spesso) nel punto di chiamata, capita talvolta che la lunghezza della riga sia tale da consigliare di spezzarla; in tal caso, non sarebbe inopportuno introdurre una convenzione sul metodo da usare. Anche in questo caso, esistono numerose alternative: le più frequenti sono, per quanto mi risulta:

1. spezzare la funzione in modo che vi sia un solo parametro per riga

2. spezzare la funzione in modo da "bilanciare" la lunghezza delle righe, indentando le linee di "continuazione" come se fossero subordinate alla linea principale.

3. come (2), ma indentando le linee di continuazione in modo che i parametri inizino comunque dopo la parentesi aperta.

il **Listato 26** mostra il risultato delle diverse tecniche sulla stessa chiamata:

Listato 26

```
// (1) un parametro per riga:
x = functionWithManyParameters(  param1,
                                 param2,
                                 param3,
                                 param4 ) ;

// (1a), possibile variante
x = functionWithManyParameters(
                               param1,
                               param2,
                               param3,
                               param4
                               ) ;
```

```
// (2) righe bilanciate, indentazione standard
x = functionWithManyParameters(  param1, param2,
  param3, param4 ) ;


// (3) righe bilanciate, allineamento a sinistra
x = functionWithManyParameters( param1, param2,
                                param3, param4 ) ;
```

Vediamo rapidamente i benefici ed i difetti delle tecniche: la (1) è la più robusta rispetto alle variazioni sui nomi o sul numero dei parametri, in quanto non richiede un riaggiustamento per mantenere il layout. Per contro, se cambia il nome della funzione occorre spostare tutti i parametri se si vuole conservare la consistenza. La (2) è esattamente opposta, poiché richiede un riaggiustamento se cambia il numero o il nome dei parametri (per mantenere il bilanciamento) ma non se cambia il nome della funzione; risulta forse meno leggibile, soprattutto se posizionata all'interno di un nesting profondo, dove peraltro è più frequente la necessità di spezzare le chiamate. La (3) ha purtroppo i difetti di entrambe, in quanto richiede modifiche sia se cambia il nome della funzione che il nome/numero dei parametri. Per contro, è forse quella più piacevole a vedersi. La soluzione più robusta in assoluto sarebbe una combinazione di (1) e (2), che non mi è mai capitato di vedere in codice reale: un esempio è nel **Listato 27**, che ne mostra però la scarsa leggibilità.

Listato 27

```
// un parametro per riga, indentazione standard:
x = functionWithManyParameters( param1,
  param2,
  param3,
  param4 ) ;

// alternativa:
x = functionWithManyParameters
  (
  param1,
  param2,
  param3,
  param4
  ) ;
```

In definitiva, sembra che la soluzione perfetta non esista; considerando che il nome delle funzioni dovrebbe essere abbastanza stabile, certamente più di

quello delle variabili, la scelta più opportuna sarebbe la (1), che porta ad un codice leggibile senza troppi problemi di manutenzione. In ogni caso, vi sono troppi fattori soggettivi per poter esprimere una raccomandazione: scegliete uno stile pesando pregi e difetti, ed usatelo in modo consistente.

Variabili e Costanti

Vi sono tre punti fondamentali da considerare nel micro-layout della dichiarazione di variabile (o di costante: non distingueremo in quanto segue): allineamento, spaziatura, e molteplicità.

Talvolta alcune variabili sono "più correlate" di altre; ciò accade soprattutto quando si è parzialmente rinunciato all'applicazione sistematica dell'incapsulazione, e quindi vengono esplicitamente usati gruppi di variabili di tipo base anziché singole variabili strutturate in classi. Al di là di questa considerazione (che riprenderemo nei capitoli seguenti), in tal caso si utilizzano normalmente delle righe vuote per separare i gruppi di variabili correlate; alcuni programmatori amano rimarcare il raggruppamento allineando il nome di variabile a sinistra, come nel **Listato 28**:

Listato 28

```
Color  foreground ;
Color  background ;

long  x ;
long  y ;
int   w ;
int   h ;

List   edges ;
Node*  current ;
```

In effetti questo stile aggiunge una ulteriore caratterizzazione visiva ai gruppi di variabili omogenee; esistono però due obiezioni ben fondate contro questo stile, una pragmatica ed una propedeutica:

Richiede troppa cura nella manutenzione: se si aggiunge ad un gruppo una variabile con il nome del tipo più lungo del livello di allineamento corrente, occorre riallineare l'intero gruppo. Spesso non si fa, e lo stile decade nel disordine.

1. Questo stile incoraggia l'uso di variabili correlate, anziché la loro incapsulazione in una classe, ed incoraggia la dichiarazione all'inizio della funzione, anziché nel punto di inizializzazione (vedi oltre).

Per tali ragioni, è opportuno utilizzare uno stile più semplice, ma che non richiede manutenzione e non incoraggia particolarmente i raggruppamenti: utilizzare semplicemente un numero fisso di spazi (tipicamente uno) per separare i vari elementi della dichiarazione, e linee vuote per separare i gruppi.

Raccomandazione 32

Evitare l'allineamento nelle dichiarazioni di variabili: utilizzare semplicemente una spaziatura standard.

Puntatori e Reference

Nota: le osservazioni di questo paragrafo, quando non diversamente specificato, si applicano anche alla dichiarazione dei parametri formali delle funzioni, ed ai membri dato delle classi.

Esistono sostanzialmente quattro modi di dichiarare variabili di tipo puntatore o reference, in funzione della spaziatura utilizzata: **Listato 29**.

Listato 29

```
int*x ;   // (1) nessuna spaziatura

int* x ;  // (2) leggere "x è di tipo puntatore ad int"

int *x ;  // (3) leggere "il tipo di *x è int"

int * x ; // (4) spaziatura totale
```

Le versioni senza spaziatura o a spaziatura totale sono raramente utilizzate, sia perché meno leggibili, sia perché rinunciano a fornire ogni informazione aggiuntiva: come abbiamo detto, una buona spaziatura deve invece aggiungere informazione al testo del programma, isolandone gli elementi fondamentali.

Spesso viene consigliata la forma “int *x” (anche se la lettura associata è meno naturale della forma “int* x”) in quanto previene un problema tipico del C e del C++, visibile nel **Listato 30**:

Listato 30

```
int* x, y ;   // x è di tipo puntatore, ma y no!!!
```

Possiamo però osservare che in effetti il problema potrebbe essere risolto alla radice anche imponendo un’altra regola: non dichiarare più di una variabile in ogni statement di dichiarazione. Torneremo su questa importante osservazione più avanti.

Consideriamo ora uno degli altri aspetti fondamentali di uno standard di codifica: la *consistenza*. Se utilizziamo una regola di spaziatura per la dichiarazione dei tipi puntatore, dovremmo utilizzare la stessa regola per dichiarare i tipi reference, che hanno una sintassi analoga; tuttavia, questo nasconde una piccola insidia: **Listato 31**.

Listato 31

```
int& x ;      // leggere: “il tipo di x è int&”

int &y ;      // ATTENZIONE: non si può leggere come
              // “il tipo di &y è int”
```

È pertanto evidente che, se vogliamo effettivamente utilizzare le spaziature per fornire informazioni a chi legge il listato, e nello stesso tempo evitare uno degli errori tipici del C e del C++, è opportuno seguire le seguenti regole:

Raccomandazione 33

Dichiarare una sola variabile in ogni statement di dichiarazione.

Raccomandazione 34

Variabili di tipo puntatore (o di tipo riferimento) vanno dichiarate giustapponendo il simbolo * (o il simbolo &) al tipo dell'oggetto puntato (o referenziato), ovvero come nell'esempio che segue:

int* x ;

Esiste un'altra considerazione di layout a favore della **Raccomandazione 33** (più altre di ordine semantico che vedremo nei prossimi capitoli): porta a codice meno soggetto ad errori di manutenzione. Se vogliamo cambiare il tipo di una variabile, od il valore di inizializzazione, non dobbiamo preoccuparci di non modificare inavvertitamente il tipo di altre variabili, o di assegnare ad esse un valore iniziale scorretto.

Se poche righe di codice in più possono proteggerci da numerosi piccoli errori, è veramente poco sensato risparmiarle; purtroppo non pochi programmatori pensano che "non possa succedere a loro": in tal caso, consiglio l'illuminante lettura di [DC85].

Infine, se utilizzate tipi puntatore (specialmente puntatori a funzione) considerate sempre l'alternativa di usare un typedef per migliorare la leggibilità globale del codice. La sintassi C per il tipo dei puntatori a funzione non è esattamente un gioiello di leggibilità, e la sintassi C++ per puntatori a funzioni membro non è certo un miglioramento. Un typedef richiede una riga in più nel vostro codice, ma può risparmiare tempo prezioso a chi lo legge. Il **Listato 32** mostra un esempio lampante, in cui sia la funzione *f* che *g* restituiscono un puntatore a funzione, determinato da un parametro; la maggiore leggibilità del typedef è fuori discussione.

Listato 32

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

int sub( int x, int y )
{
  return x - y ;
}

// What !?
```

```
int (*g( int s ))( int, int )
{
  return s ? add : sub ;
}

typedef int (*BinOp)( int, int ) ;

// OK
BinOp f( int s )
{
  return s ? add : sub ;
}
```

Raccomandazione 35

Se utilizzate tipi complessi, ad esempio puntatori a funzione, dichiarate un tipo apposito con typedef.

Iterazione e Condizionale

Prima di discutere del micro-layout vero e proprio per le strutture di controllo, è importante dedicare alcune righe ad una riflessione che altrimenti non troverebbe spazio autonomo in questo libro: in C, e quindi in C++, le strutture di controllo sono molto liberali, ad esempio il *for* può essere facilmente usato come un *while* del Pascal. Questa libertà può essere molto utile in determinate occasioni, ma rappresenta nuovamente una scelta comunicativa abbastanza discutibile: se per il compilatore un *for* o un *while* del C sono molto simili, per chi legge il vostro codice non lo sono. Ogni struttura di controllo ha un suo uso schematizzato (un "cliché" secondo la terminologia di [RW90]) che comunica a chi legge non solo le azioni effettivamente eseguite, ma anche le intenzioni del programmatore: ad esempio, un loop su tutti gli elementi di un array è normalmente eseguito con un ciclo *for*, mentre la ricerca di un elemento in una linked list con un *while*. Rinunciare all'uso "standard" delle strutture di controllo del flusso spesso significa solo rendere il proprio codice meno leggibile: cercate sempre di comunicare non solo cosa state facendo, ma anche cosa *volevate fare*.

Raccomandazione 36

Utilizzate sempre la struttura di controllo che meglio esprime le azioni che intendete compiere ad un livello più astratto.

Uscire dai cicli

Esistono tre tecniche per uscire da un ciclo quando nel corpo del ciclo stesso si verifica una determinata condizione (trascurando il *goto*):

1. controllare un flag all'interno della condizione, e modificarlo nel body.
2. usare *break*.
3. usare *return*.

L'uso di (1) è tipico di chi proviene da linguaggi, come il Pascal, che non dispongono di uno statement di break; d'altra parte, non porta necessariamente a codice più semplice rispetto a (2), ed anzi in presenza di side-effect nella condizione può risultare anche più complesso da capire. Il metodo (2) è molto usato in C ed in C++, e deve comunque fare parte del bagaglio di chi programma in tali linguaggi poiché è un componente essenziale dello *switch/case*. La tecnica (3) è la meno consigliabile, perché è meno resiliente delle altre rispetto alle modifiche: se anziché restituire un valore doveste in seguito eseguire ulteriori operazioni su di esso, dovreste trasformare il codice in modo da usare la (1) o più probabilmente la (2). Inoltre usando (1) o (2) abbiamo un solo punto di uscita per la funzione, e ciò significa in genere che è più semplice eseguirne un testing accurato. Ovviamente, nel caso di funzioni banali come la ricerca di un elemento data una chiave, anche la tecnica (3) è più che adeguata.

Raccomandazione 37

Preferire il *break* o un flag al *return* per uscire dai loop.

Corpo vuoto

Data la flessibilità del *while*, non è raro trovare dei cicli in cui tutto il compito viene svolto nella condizione, ed il corpo è vuoto; l'esempio più famoso è la funzione *strcpy*, come da **Listato 33**:

Listato 33

```
void strcpy( char* s, const char* t )
  {
  while( ( *s++ = *t++ ) != '\0' )
    ;
  }
```

In questi casi, sarebbe sempre opportuno mettere il ';' su una riga separata (come sopra) magari accompagnato da un commento, se non addirittura introdurre un blocco vuoto, sempre su linee separate e con un commento al suo interno. Nuovamente, pochi caratteri in più in fase di scrittura potrebbero salvare parecchio tempo durante la manutenzione.

Raccomandazione 38

Se il body di uno statement è vuoto, posizionate il ';' o un blocco vuoto su una riga separata, ed annotatelo con un commento.

Limiti

La seguente norma è stata suggerita da molti autori che si sono occupati di "programming style": in un ciclo *for*, utilizzare un limite inferiore inclusivo ed un limite superiore esclusivo, come nel **Listato 34**:

Listato 34

```
for( int i = 0; i < 10; i++ )
  // ...
```

Vi sono due vantaggi importanti insiti in questo stile:

1. possiamo contare il numero di cicli con una semplice sottrazione dei due limiti.

2. possiamo usare la dimensione di un array come limite superiore, senza preoccuparci di sottrarre 1.

Non essendo in alcun modo limitativo, sarebbe pertanto opportuno adottare lo schema ogni volta che si utilizza un ciclo *for*.

Raccomandazione 39

Nei cicli *for*, utilizzate un limite inferiore inclusivo ed un limite superiore esclusivo.

"if" annidati

Consideriamo il **Listato 35**, che mostra un uso dello statement *if* con un notevole annidamento:

Listato 35

```
int state = f1() ;
if( state == OK )
  {
  state = f2() ;
  if( state == OK )
    {
    state = f3() ;
    if( state == OK )
      // ...
    }
  }
```

lo stesso codice può essere scritto come nel **Listato 36**:

Listato 36

```
int state = f1() ;

if( state == OK )
  state = f2() ;

if( state == OK )
  state = f3() ;

if( state == OK )
  // ...
```

La seconda versione è molto più semplice da capire, in quanto non dobbiamo affatto preoccuparci del livello di nesting; ovviamente, possiamo usarla solo se non esiste un *else*, ma ciò avviene in un numero sorprendentemente alto di casi. Il vantaggio della prima versione è che non è necessario eseguire tutti i test rimanenti se la condizione diventa falsa: in

genere, un ben misero risparmio di tempo. Il nesting, come le parentesi, va usato con discernimento, perché gli esseri umani non sono molto bravi a comprendere modelli con una struttura troppo ricca: non cedete sul fronte della comprensibilità per un trascurabile guadagno in efficienza.

Raccomandazione 40

Usare il nesting con discernimento: talvolta è possibile linearizzare la struttura del codice con un impatto trascurabile sulle prestazioni.

Switch

Anche nel caso dello statement *switch*, più che raccomandazioni di micro-layout vere e proprie sono qui riportate alcune note di pragmatica per prevenire futuri problemi di manutenzione.

La sintassi del C e del C++ prevede il fall-through automatico tra le clausole dello statement di switch, contrariamente ad altri linguaggi dove il fall-through non è permesso, ed il raggruppamento di più clausole deve essere esplicitamente dichiarato. Ciò porta ad un bug abbastanza frequente, ovvero la fine logica di una clausola non corrisponde alla sua fine fisica in quanto manca il relativo *break* o *return*; in effetti, la sintassi del C è abbastanza discutibile, in quanto il fall-through è piuttosto raro in programmi reali. In ogni caso, per rendere il codice più chiaro, ogni clausola dovrebbe terminare con un *break* o un *return*, salvo nei rari casi di fall-through che vanno *sempre* commentati, perché non è insolito che un programmatore "assuma" di aver visto un break prima della clausola successiva.

Raccomandazione 41

Ogni clausola di uno statement *switch* va terminata con *break* o con *return*, salvo i casi di fall-through intenzionale che devono *sempre* essere commentati.

Esiste un'ulteriore accortezza legata all'uso di *switch* che può risparmiare parecchio tempo in fase di manutenzione; considerate il **Listato 37**, dove uno *switch/case* è utilizzato su un tipo enumerato:

Listato 37

```
// TEST.H

#include <iostream.h>

enum Color { RED, GREEN, BLUE } ;

ostream& operator <<( ostream& s, Color c ) ;



// TEST.CPP

#include "test.h"


ostream& operator <<( ostream& s, Color c )
  {
  switch( c )
    {
    case RED :
      s << "red" ;
      break ;
    case GREEN :
      s << "green" ;
      break ;
    case BLUE :
      s << "blue" ;
      break ;
    }
  return( s ) ;
  }
```

Poiché abbiamo esaurito tutti i casi possibili, non è necessario avere anche una clausola *default*; in realtà, sarebbe molto consigliabile introdurne una che generi una violazione di asserzione, come nel **Listato 38**:

Listato 38

```
// TEST.CPP

#include <assert.h>
#include "test.h"

ostream& operator <<( ostream& s, Color c )
```

```
{
switch( c )
  {
  case RED :
    s << "red" ;
    break ;
  case GREEN :
    s << "green" ;
    break ;
  case BLUE :
    s << "blue" ;
    break ;
  default :
  assert( 0 ) ;
  // Meglio usare la propria macro di assert e FALSE.
  break ;
  }
return( s ) ;
}
```

La ragione è piuttosto semplice: se in fase di manutenzione aggiungiamo un elemento al tipo enumerato, l'asserzione segnalerà un errore che viceversa potrebbe passare inosservato. Inoltre, alcuni compilatori sono piuttosto liberali nella gestione degli enumerati, ed emettono solo un warning quando un qualunque intero è restituito da una funzione con risultato di tipo enumerato. In tal caso, la variabile *c* dell'esempio potrebbe in effetti assumere valori al di fuori dal range consentito staticamente; una asserzione permetterebbe di trovare la chiamata incriminata e di correggere l'errore.

Raccomandazione 42

Introdurre sempre una clausola *default* negli statement *switch*; se il controllo non deve mai raggiungere tale clausola, inserire una asserzione falsa come corpo, in modo che ogni violazione delle assunzioni fatte venga segnalata a run-time.

Classi

Il micro-layout delle classi è interessante principalmente dal lato della dichiarazione: seguendo la raccomandazione vista all'inizio del capitolo, il macro-layout dell'implementazione seguirà poi il micro-layout della dichiarazione (ovvero le varie funzioni verranno implementate nell'ordine di dichiarazione), rendendo il codice più semplice da navigare anche per

altri programmatori. Le classi rappresentano in sé una unità di incapsulazione, ed analogamente al modulo possono includere nella loro dichiarazione costanti, tipi, classi, funzioni e variabili; il loro spazio è però partizionato in tre sezioni di accessibilità (pubblica, protetta, privata).

La consistenza con il macro-layout del modulo suggerisce di organizzare *ogni* sezione come se fosse un vero e proprio file header: troveremo quindi prima i tipi annidati, poi gli enum, poi classi annidate, funzioni, eccetera. In realtà esistono alcuni elementi che nelle classi hanno un significato leggermente diverso (ad esempio le funzioni *static*), pertanto è necessario un minimo riassestamento rispetto al modello di layout per l'header; come vedremo, si tratta soprattutto di eliminare elementi che non dovrebbero comparire all'interno di una dichiarazione di classe (macro) e di aggiungere alcuni elementi tipici della classe: in nessun caso l'ordine è stato modificato per gli elementi comuni. Da notare che l'ordine si è sostanzialmente auto-preservato, in quanto basato sulle interdipendenze tra i diversi elementi, non su motivazioni di ordine estetico: se avete scelto un diverso macro-layout per l'header, questo è un buon punto per verificare la sua stabilità.

Il micro-layout proposto per ogni *sezione* di una dichiarazione di classe è:

[Typedef]

[Enum]

[Classi annidate]

[Const]

[Variabili membro statiche]

[Variabili membro]

[Funzioni membro statiche]

[Funzioni membro]

[Operatori membro]

Come vedete, è estremamente simile al macro-layout di un file header, salvo che le macro del preprocessore sono state eliminate e gli elementi

statici sono stati divisi da quelli di istanza; se dovete realmente inserire una macro all'interno di una classe potete ovviamente farlo, ma si tratta di casi così eccezionali da meritare sicuramente un commento.
Le motivazioni per il layout proposto sono in gran parte simili a quelle che giustificano il macro-layout di un header: la necessità di distinguere tra elementi statici e di istanza è ovvia, mentre l'ordine scelto (statici, d'instanza) ha una spiegazione "di principio" ed una pragmatica:

- Poiché i primi elementi (enum, classi, const) sono da considerare statici (ovvero riferiti all'intera classe), è più coerente mantenere gli elementi statici all'inizio della sezione.

- I costruttori ed il distruttore sono funzioni statiche, per quanto un po' particolari, ed era desiderabile che fossero tra le prime funzioni della classe. In tal senso, potremmo dire che il micro-layout della sezione [Funzioni membro statiche] è in realtà:

  [Funzioni membro statiche "comuni"]

  [Costruttori]

  [Distruttore]

Notiamo che ciò giustifica anche la scelta di avere le variabili, statiche e di istanza, prima delle funzioni; un ordinamento alternativo potrebbe essere variabili statiche, funzioni statiche, variabili di istanza, funzioni di istanza. Il difetto di tale schema è che separa il costruttore dalle altre funzioni, mentre ho osservato che il codice è più leggibile quando sono vicini. Nuovamente, vi è un certo spazio per riaggiustamenti personali, anche se dovete sempre cercare di basare il layout su considerazioni il più possibile oggettive.

Avendo discusso il micro-layout di ogni sezione, resta da vedere come organizzare il layout complessivo della dichiarazione: un semplice criterio di ordine rende evidente che si dovrebbero ordinare le sezioni come pubblica/protetta/privata o come privata/protetta/pubblica. Più difficile è la scelta tra una delle due alternative: alcuni suggeriscono la prima, così che chi legge la dichiarazione della classe trovi ciò che gli interessa (la parte pubblica) all'inizio, e che proseguendo nella lettura si "scoprano" più dettagli sull'implementazione della classe. In effetti ho avuto modo di constatare che in questo caso si tratta più che altro di una abitudine, e che programmatori che usavano sistematicamente la seconda struttura avevano

l'abitudine di leggere le dichiarazioni "dal basso", ovvero localizzavano rapidamente la sezione "public" e proseguivano verso il basso. Notiamo che dietro questo apparentemente futile argomento di layout si nasconde una vera e propria filosofia di sviluppo: da quanto ho potuto osservare, l'ordinamento public/protected/private è più comune tra chi pensa all'interfaccia che la classe deve esporre prima che ai dati che la classe deve contenere, e trova più naturale scrivere la parte pubblica inizialmente. Viceversa, l'ordinamento private/protected/public è più comune tra chi pensa ai dati (o all'implementazione) della classe prima che all'interfaccia, e trova quindi più naturale iniziare la stesura dichiarando le variabili di istanza private. Chi non ha alcun metodo di progetto, spesso non ha neppure un criterio di layout, ovvero scrive alcune classi con un certo ordine tra le sezioni, ed altre classi con un ordine diverso. Ovviamente, la scelta di un layout non pregiudica l'uso di una tecnica o dell'altra durante lo sviluppo: è più che altro la tecnica che porta invece in modo naturale all'adozione di un layout.

In ogni caso, dovendo suggerire una particolare sequenza ho basato la scelta sulle seguenti osservazioni:

- Il layout dell'implementazione (come visto prima) dovrebbe ricalcare il layout della dichiarazione.

- Sarebbe apprezzabile che i costruttori fossero normalmente all'inizio del file di implementazione.

- I costruttori sono di norma pubblici.

Le funzioni private e la variabili di istanza private possono essere di un tipo (esempio enumerato) dichiarato nella sezione pubblica, mentre non è vero il viceversa. In generale, elementi privati possono dipendere da elementi protetti o pubblici, ed elementi protetti dai pubblici, ma non il viceversa.

Le prime motivazioni sono abbastanza soggettive, e probabilmente le stesse obiezioni di "abitudine del programmatore" possono essere portate contro di esse; l'ultima è però inattaccabile, e dimostra la superiorità della sequenza public/protected/private.

Il micro layout suggerito per una dichiarazione di classe diventa quindi:

[Sezione pubblica]

[Sezione protetta]

[Sezione privata]

[Classi friend]

[Funzioni friend]

[Operatori friend]

Notiamo che i friend sono indipendenti dalla sezione in cui appaiono, e pertanto sono stati raggruppati al fondo della classe; nuovamente, potrebbero essere messi all'inizio, e l'unica ragione per cui suggerisco di metterli alla fine è che spesso si tratta di dettagli implementativi motivati dall'efficienza.

Raccomandazione 43

Strutturate la dichiarazione di una classe ordinando le sezioni di accessibilità come pubblica/protetta/privata, seguite da classi, funzioni, operatori friend.

Raccomandazione 44

All'interno di ogni sezione, seguite un layout il più vicino possibile a quello di un file header, con le dovute differenze per i membri statici.

Possiamo ora vedere alcune considerazioni di layout di granularità più fine, anche se sempre riferite alla dichiarazione di classe; innanzitutto va rimarcato che i criteri relativi al raggruppamento di elementi "vicini" dal punto di vista concettuale si applicano anche per la dichiarazione di classe. Anche se gli elementi di una classe dovrebbero essere tutti molto vicini dal punto di vista concettuale (ovvero, la classe dovrebbe essere altamente coesiva) spesso accade che alcune variabili o funzioni membro siano più simili tra loro che alle altre; se da una parte questo può suggerire una possibile astrazione in un'altra classe, in ogni caso potremmo comunque rendere tale vicinanza più evidente separandole con una riga vuota dalle altre variabili/funzioni. In generale, è meglio raggruppare oggetti con lo stesso livello di astrazione (breadth first) che cercare di raggruppare gli stessi aspetti di un singolo dettaglio (depth first).

Notiamo che lo stesso concetto si applica efficacemente anche ai dati, non solo alle funzioni; a tal proposito, se si vuole ottenere il massimo della consistenza tra la parte dichiarativa e la parte implementativa, potrebbe essere utile inizializzare le variabili di istanza, all'interno dei costruttori, nell'esatto ordine di dichiarazione, e distruggerle nell'ordine *opposto* all'interno del distruttore. Ciò porterebbe ad ordinare le "risorse" della classe nell'ordine in cui vengono acquisite, aumentando la comprensibilità della dichiarazione e risolvendo il problema di "dove mettere le nuove righe" in fase di manutenzione; si tratta comunque di finezze che non riassumerò in raccomandazioni, per non appesantire ulteriormente i criteri di layout sin qui esposti. Se il lettore ritiene opportuno seguirle, può facilmente integrare le sue "raccomandazioni personali" includendo quanto sopra.

Una buona norma di codifica, che meriterà invece una raccomandazione specifica, è relativa alle funzioni inline. In molti testi (talvolta anche in questo libro) per brevità le funzioni inline vengono definite nel punto di dichiarazione, all'interno della classe. Se ciò è giustificabile per un libro od un articolo, dove gli esempi devono spesso essere di lunghezza minima, non per questo rappresenta un buon esempio di layout: non solo la lettura della dichiarazione di classe diventa meno immediata, ma è anche più scomodo modificarla e cambiare la categoria di una funzione - da inline a non-inline o viceversa. Il codice è molto più chiaro e mantenibile se si sposta la definizione delle funzioni inline immediatamente dopo la dichiarazione della classe, come nel **Listato 39**:

Listato 39

```
class Stack
  {
  public :
    Stack() ;
    void Push( int x ) ;
    void Pop() ;
    int Top() ;
  private :
    int stack[ 100 ] ;
    int top ;
  } ;


inline void Stack :: Push( int x )
  {
```

```
  stack[ top++ ] = x ;
  }

// ...
```

Notiamo che non è strettamente necessario mettere lo specificatore *inline* nella dichiarazione della funzione: è sufficiente metterlo nella definizione. Se ciò costituisca o meno un corretto stile di programmazione è abbastanza controverso: non metterlo corrisponde ad un information hiding, quindi il problema è se sia o meno corretto nascondere tale informazione ad una prima lettura dell'header. Dal mio punto di vista, poiché la definizione delle funzioni inline si trova comunque nell'header, non mettere lo specificatore nella dichiarazione di classe significa astrarre da un dettaglio di efficienza che viene comunque ripreso in seguito a beneficio degli interessati. È comunque quest'ultima una visione troppo soggettiva per meritare una raccomandazione esplicita.

Raccomandazione 45

La definizione delle funzioni inline non va inserita all'interno della dichiarazione della classe, ma al di fuori di essa.

Commenti

Molti programmatori *non* commentano il loro codice: talvolta per mancanza di tempo, talvolta per trascuratezza, talvolta perché fa parte del loro stile (l'assenza di commenti è tipica nel codice degli "artisti"), talvolta perché sono convinti che la migliore documentazione sia il codice stesso.
Se avete avuto modo di parlare con un programmatore appartenente all'ultima categoria, avrete sicuramente "appreso" che i commenti creano problemi perché devono essere mantenuti, al pari del codice, e che in realtà un buon codice è comprensibile anche in assenza di commenti, che sono comunque meno precisi. Spesso anche programmatori con molta esperienza (talvolta proprio a cagione di tale esperienza) sono dei fieri sostenitori di questa posizione.

C'è ovviamente del vero in quelle parole: un commento in disaccordo con il codice diminuisce la comprensibilità di un programma, e una scelta attenta dei nomi (come quella auspicata in questo libro) diminuisce sensibilmente la necessità di commenti esplicativi.

In realtà, i problemi di “manutenzione” ed obsolescenza dei commenti sono normalmente associati ad un uso errato dei commenti stessi: il commento non deve (di norma) spiegare *cosa fa* il codice: deve spiegare *l’intenzione* del codice. In altri termini, dovrebbe dare una visione astratta dell’azione compiuta dal codice che segue: azione che ben di rado cambierà a seguito della manutenzione, perché le intenzioni cambiano con meno frequenza delle tecniche usate per realizzarle. I commenti dovrebbero pertanto convogliare una informazione a livello di design, più che spiegare cosa fa il codice (che effettivamente, se ben scritto dovrebbe essere comprensibile in sé).
Possiamo vedere alcuni esempi corretti o errati nel **Listato 40**:

Listato 40

```
// commenti inutili: ripetono il codice

i++ ;        // incrementa i

patient.SetFirstName( firstname ) ; // Modifica nome
patient.SetLastName( lastname ) ;   // Modifica cognome
patient.SetAge( age ) ;             // Modifica eta'



// commenti dannosi: contraddicono il codice

i-- ;        // incrementa i



// commenti utili: spiegano le intenzioni


// Aggiorna i dati del paziente

patient.SetFirstName( firstname ) ;
patient.SetLastName( lastname ) ;
patient.SetAge( age ) ;



// aggiunge i segmenti nel range
// [logicalStart, logicalEnd]

LineSegment bs ;
```

```
if(baselineDB->GetFirstOverlapping(logicalStart,
logicalEnd, bs))
  do
    Add( bs ) ;
  while( baselineDB->GetNextOverlapping( logicalStart,
logicalEnd,bs ) ) ;
```

Una caratteristica comune dei commenti "di intenzione" è quella di *precedere* il codice cui si riferiscono: ciò riflette il fatto che il programmatore conosce la sua intenzione *prima* di scrivere il codice stesso; spesso i commenti che "spiegano" il codice *seguono* il codice stesso, segno che dopo aver scritto alcune linee poco comprensibili il programmatore ha avuto un "pentimento" e le ha commentate.
Ovviamente, vi sono casi in cui non si può fare a meno di un commento esplicativo, vedi ad esempio il **Listato 41**; ciò avviene perché inevitabilmente alcune particolarità del codice non possono essere rese evidenti con l'uso di identificatori appropriati o con l'uso "pulito" dei costrutti del linguaggio.

Listato 41

```
void BaselineDatabase ::
SetBaseline( int channel,
             SampleNum startTime,
             SampleNum endTime,
             float y1,
             float y2 )
  {
  ASSERT( startTime <= endTime ) ;

  static Period newPeriod ;
  // static per non chiamare il costruttore
  // ad ogni ingresso nella funzione

  // ...
  }
```

In molti casi, tuttavia, i commenti "esplicativi" possono essere evitati scrivendo codice più comprensibile e scegliendo con accuratezza gli identificatori. Anche quando scrivete commenti esplicativi, cercate sempre di trasmettere le vostre intenzioni, più che ripetere quello che avete già scritto in modo più formale nel codice. Un buon candidato per un commento esplicativo potrebbe essere il codice che viola una delle raccomandazioni di questo libro.

Raccomandazione 46

Nei commenti, cercate di spiegare le vostre intenzioni, non come le state realizzando. Preferite i commenti introduttivi a quelli esplicativi, che spesso si possono evitare scrivendo codice più chiaro.

Infine, è sempre consigliabile indentare i commenti esattamente come il codice a cui si riferiscono: anche se alcuni programmatori amano "incorniciare" i commenti, facendo sì che occupino sempre l'intera riga, ciò provoca un fastidioso effetto di discontinuità nel layout globale del programma. Inoltre si rende più difficile l'associazione di un commento con il relativo codice, se il commento inizia sulla colonna 1 e il codice sulla colonna 32: si potrebbe facilmente pensare che il commento sia riferito allo statement di controllo precedente.

Raccomandazione 47

Indentare i commenti allo stesso livello del codice a cui si riferiscono.

5

Costanti

"None of us really understands what's going on with all these numbers."
David Allen Stockman

Le costanti, soprattutto di tipo numerico, non dovrebbero quasi mai apparire come tali nel codice, ma sempre attraverso un nome simbolico. Questo è uno dei punti chiave che permettono di distinguere immediatamente tra codice scritto in modo professionale, con la dovuta attenzione a trasmettere informazioni a chi legge, e codice scritto da non-professionisti, difficile da comprendere e da modificare in seguito.
Pensiamo infatti a costanti piuttosto comuni, come '2': se in fase di modifica del codice una di queste costanti dovesse cambiare valore, in quanti punti del codice troveremmo '2', ed in quanti punti dovremo effettivamente cambiarlo? Alcune occorrenze di '2' potrebbero essere legate alla modifica che dobbiamo apportare, altre totalmente scorrelate: usare un nome simbolico per le diverse costanti eviterebbe ogni problema sin dall'inizio, ed è una lezione che i programmatori esperti imparano presto.

Esistono diverse opzioni per dichiarare delle costanti in C++:

- usare la direttiva #define del preprocessore, come in:
  ```
  #define PI 3.14159
  ```

- dichiarare una costante del tipo opportuno, come in:
  ```
  const float PI = 3.14159 ;
  ```

- definire un tipo enumerato, come in:
  ```
  enum Colors { red, green, blue } ;
  ```

I tipi enumerati e le costanti possono anche essere dichiarati internamente ad una classe.

L'uso del #define, molto comune tra gli ex-programmatori C, dovrebbe essere evitato ogni volta che sia possibile, in quanto affetto da numerosi problemi:

1. La direttiva #define è gestita dal preprocessore, non dal compilatore. Il preprocessore è molto rudimentale, e non ha alcuna gestione dei tipi, dello scope, e così via. Pertanto, volendo ad esempio dichiarare una costante locale, come nel **Listato 42**, si ottiene comunque una dichiarazione di visibilità globale: ogni occorrenza di PI a partire dal punto di definizione verrà sostituita dal preprocessore.

Listato 42

```
float Area( float radius )
  {
  #define PI 3.14159

  return( radius * radius * PI ) ;
  }

float f()
  {
  // PI è visibile all'interno di f

  return( PI ) ;
  }
```

Inoltre, la mancanza di una dichiarazione di tipo esplicita può causare subdoli problemi di portabilità, in quanto il compilatore può assumere il tipo più idoneo (normalmente il tipo più piccolo che può contenere il valore) quando incontra una costante numerica: il codice del **Listato 43** può funzionare correttamente su macchine a 32 bit ma non su macchine a 16 bit.

Listato 43

```
#define BIG_CONST 63000

int f()
  {
  return( BIG_CONST ) ;
  }
```

Dichiarando la costante con la keyword *const*, dobbiamo invece dare un tipo esplicitamente, ad esempio come nel **Listato 44**:

Listato 44

```
const unsigned int BIG_CONST = 63000 ;

int f()
  {
  return( BIG_CONST ) ;
  // warning: restituiamo un unsigned int !!!
  }
```

2. Errori in fase di preprocessing possono essere difficili da individuare e riconoscere come tali durante la compilazione; consideriamo il **Listato 45**:

Listato 45

```
float Area( float radius )
  {
  #define PI 3.14159 ;

  return( radius * radius * PI ) ;
  }
```

Il compilatore segnalerà un errore nella linea

return(radius * radius * PI) ;

Il messaggio è dipendente dal compilatore stesso, ma in genere sarà ben poco esplicito rispetto al reale errore; un compilatore piuttosto diffuso genererà il seguente output:

Error test.cpp 5:) expected in function Area(float)
Warning test.cpp 5: Unreachable code in function Area(float)
Error test.cpp 5: Expression syntax in function Area(float)
Warning test.cpp 6: Function should return a value in function Area(float)
Warning test.cpp 6: Parameter 'radius' is never used in function Area(float)

Ovviamente, l'errore è invece sulla linea del #define, dove è stato aggiunto un ';' alla fine della riga stessa.

3. I simboli definiti con #define non sono normalmente disponibili in fase di debugging, dove quindi troveremo solo i valori numerici

corrispondenti; molti debugger possono invece visualizzare i nomi simbolici dei tipi enumerati ed in alcuni casi delle costanti dichiarate con *const*.

4. L'uso di #define è associato ad un errore piuttosto comune, dal quale i programmatori esperti si guardano in modo pressoché automatico, ma che può comunque causare bug abbastanza difficili da rintracciare: nel **Listato 46**, la mancanza di parentesi nella definizione di 2PI causa un errore nel punto di utilizzo.

Listato 46

```
#define PI  3.14159
#define 2PI  PI + PI

float Circumference( float radius )
  {
  return(  radius * 2PI ) ;
  // restituisce ( radius * PI ) + PI !!
  }
```

Possiamo ricavare da tutto ciò le seguenti regole di codifica:

Raccomandazione 48

Evitare di inserire esplicitamente valori numerici nel codice: definire invece delle opportune costanti.

Raccomandazione 49

Non utilizzare #define per definire le costanti, ma *const* o *enum*.

Const o enum?

È naturale chiedersi se sia più opportuno utilizzare *const* o *enum* per definire delle costanti simboliche. La soluzione è abbastanza immediata, se consideriamo che *enum* introduce un nuovo tipo, definito *estensionalmente*, ovvero per enumerazione dei suoi elementi. Tale tipo deve avere una sua coerenza interna, esattamente come una classe: avrebbe ben poco senso inserire in un tipo enumerato costanti scorrelate, come Rosso, Tavolo,

Lampadina[8]. Un tipo enumerato deve raggruppare oggetti omogenei, come Rosso, Verde e Blu. Negli altri casi, si utilizzino le costanti.

Raccomandazione 50

Usare *enum* per raggruppare costanti omogenee; utilizzare *const* per dichiarare costanti eterogenee.

Ricordate comunque che, mentre in ANSI C le costanti globali sono automaticamente di classe *extern*, ovvero visibili agli altri moduli, in C++ una costante globale è di classe *static*, ovvero di visibilità limitata al modulo di dichiarazione; se la costante è dichiarata in un include file, verrà creata una nuova costante per ogni modulo che lo includa, a meno che la costante non sia esplicitamente dichiarata *extern* nell'include file. Ciò è in genere abbastanza irrilevante, tranne che in due casi:

1. alcuni compilatori trattano le costanti come variabili, a meno che non vengano abilitate opportune ottimizzazioni; in questo caso, se non si dichiara la costante come *extern* nell'include file, il risultato è di richiedere più memoria del necessario per mantenere i valori delle costanti per ogni modulo.

2. in alcuni casi, ad esempio quando si prende l'indirizzo di una costante, il compilatore è costretto a trattarla come una variabile, e si ricade nel caso precedente. Inoltre, comparare l'indirizzo della stessa costante ma riferita a moduli diversi potrebbe dare risultati inattesi. In ogni caso, si tratta di un caso piuttosto estremo. Un esempio è comunque riportato nel **Listato 47**, che stampa in output '0', poiché ogni modulo ha una sua istanza statica della costante *GLOBAL*.

Listato 47

```
// CONST.H

const int GLOBAL = 10 ;

extern int IsGlobal( const int* p ) ;
```

[8] in effetti anche questi simboli potrebbero essere correlati, se descrivessero ad esempio gli oggetti presenti in una stanza; il significato della frase dovrebbe comunque essere chiaro.

```
// GLOB.CPP

#include "const.h"


int IsGlobal( const int* p )
  {
  return( p == &GLOBAL ) ;
  }




// MAIN.CPP

#include <iostream.h>
#include "const.h"


int main()
  {
  cout << IsGlobal( &GLOBAL ) ;

  return( 0 ) ;
  }
```

Incapsulare le costanti

Alcune costanti, ad esempio TRUE e FALSE, sono molto generali ed utilizzabili in molti contesti diversi; altre, come specifici codici di errore o le diverse opzioni per una famiglia di funzioni, dovrebbero essere utilizzate solo in contesti molto ristretti, ovvero la chiamata di apposite funzioni. Pertanto, sarebbe consigliabile rendere esplicita questa caratteristica delle costanti, direttamente nel codice del programma, senza lasciare all'interpretazione umana il compito di capire i contesti di chiamata corretti.

Sia che usiate *enum* che *const*, cercate sempre di identificare i casi in cui le costanti non hanno senso se non nel contesto di una classe: in tal caso, è largamente preferibile definire il tipo enumerato o le costanti stesse all'interno della classe. In tal modo si evitano possibili collisioni, e nel

punto di chiamata si arricchisce la costante con l'informazione della classe di appartenenza. Vedere anche il capitolo 9, al paragrafo "Argomenti ed Interfacce" per un esempio concreto di incapsulazione delle costanti.

Raccomandazione 51

Se una o più costanti sono utilizzate solo all'interno di una classe, o come parametri per le funzioni membro di una classe, le costanti vanno definite internamente alla classe, e dotate del giusto grado di visibilità (private, protected o public) a seconda dell'impiego che possono avere.

6

Variabili

"Keep yourself to yourself"
Charles Dickens

Tipi predefiniti

In molte occasioni, uno dei tipi predefiniti, come *float*, *int*, o *char*, è perfettamente adeguato per contenere l'insieme dei valori che una variabile deve assumere: ad esempio, potremmo rappresentare una frequenza od una distanza con un intero. In alcuni casi ciò è perfettamente lecito: dichiarare un tipo per la variabile indice di un *for* ha in genere ben poco senso, mentre il tipo *int* o *unsigned int* sarà più che adeguato. In molti altri casi, ad esempio per la frequenza o la distanza viste prima, l'uso dei tipi predefiniti, come nel **Listato 48**, è una pessima pratica di programmazione.

Listato 48

```
int maxFreq ; // la massima frequenza

int minWidth ; // minima lunghezza

int backgroundColor ; // colore di sfondo

float Area( int radius ) ;
```

L'uso diretto dei tipi predefiniti è una pratica molto diffusa in C, in gran parte per motivi storici; altri linguaggi, come il Pascal, fanno della definizione di tipi adeguati uno dei capisaldi nella pratica della buona programmazione. Si tratta in effetti di uno stile di codifica, e non di un diverso supporto da parte del linguaggio, che tuttavia ha contribuito alla diffusione dell'opinione che "il Pascal sia inerentemente più leggibile".

È abbastanza semplice capire perché utilizzare direttamente i tipi predefiniti sia talvolta deleterio: se i requisiti cambiano (ad esempio, la frequenza

potrebbe in seguito richiedere un float) è realmente molto difficile trovare tutte le espressioni che coinvolgono frequenze all'interno del codice e modificarle. Ciò vale sia per le variabili che per i risultati delle funzioni, ed è ulteriormente aggravato dal fatto che modificando solo alcune delle occorrenze necessarie di *int* in *float* avremo dei troncamenti dei risultati, in genere senza alcun warning dal compilatore.

Definire un tipo ha anche altri vantaggi immediati: se i valori ammissibili sono pochi e tra loro correlati, un tipo enumerato potrà essere vantaggiosamente utilizzato, rappresentando in un'unica soluzione il tipo ed i suoi elementi, consentendo l'overloading delle funzioni e una migliore gestione degli switch/case, senza la necessità di clausole *default* come sarebbe necessario per gli interi.
Inoltre, avere un tipo sin dall'inizio si rivelerà utile se, in fase di evoluzione del programma, un tipo "marginale" assumesse la dignità di classe. In tal caso, si tratterebbe sostanzialmente di modificare un typedef in una dichiarazione di classe, e definire gli opportuni operatori e funzioni, senza necessità di modificare il codice che fa uso del tipo. Molto diversa (e notevolmente peggiore) è la situazione se si è utilizzato un tipo base ovunque nel codice.

Alla luce di queste considerazioni, una versione migliore del **Listato 48** è quella del **Listato 49**

Listato 49

```
typedef int Frequency ;

typedef int Meter ;

typedef unsigned int SquareMeter ;

enum Color { RED, GREEN, BLUE } ;


Frequency maxFreq ;

Meters minWidth ;

Color backgroundColor ;

SquareMeter Area( Meter radius ) ;
```

Osserviamo che il codice è molto più chiaro per chi legge, in quanto i nomi dei tipi aggiungono significato al nome delle variabili; inoltre nel punto di dichiarazione delle variabili si ha un effettivo *information hiding*: mentre nel **Listato 48** era esplicita l'implementazione di una frequenza come un intero, qui è stata efficacemente nascosta dal tipo Frequency.

Infine, è abbastanza comune l'uso del tipo *int* anche in situazioni in cui un *unsigned* sarebbe più indicato, poiché la quantità non può assumere valori negativi: nel **Listato 49**, ad esempio, un'area non può essere negativa, e pertanto è definita come unsigned. Ricordate sempre che la precisione nella dichiarazione dei tipi aiuterà la comprensione del codice e permetterà al compilatore di segnalare opportuni warning nei casi sospetti (come l'assegnazione di una quantità negativa ad un'area).

Raccomandazione 52

Utilizzare sempre il tipo più opportuno per ogni variabile; non abusare dei tipi *int* o *float*: talvolta *unsigned* può essere più indicato.

Raccomandazione 53

Non utilizzare i tipi predefiniti esplicitamente qualora sia possibile (e significativo) nascondere l'implementazione definendo un opportuno tipo con *typedef* od *enum*.

Variabili correlate

Quando si esamina codice C++ scritto da programmatori con scarsa esperienza nel paradigma object oriented, non è insolito trovarsi di fronte a codice simile al **Listato 50**:

Listato 50

```
// Larghezza ed altezza per 100 rettangoli

int width[ 100 ] ;
int height[ 100 ] ;
```

Per chi ha una certa esperienza di OOP, simile codice è una vera e propria eresia; oltre all'uso di una costante numerica esplicita, variabili che sono strettamente correlate da un punto di vista concettuale si trovano

completamente slegate da un punto di vista implementativo. Una implementazione di gran lunga migliore è data nel **Listato 51**.

Listato 51

```
class Rectangle
  {
  public :
    Rectangle( int w, int h ) ;
    // ....
  private :
    int w ;
    int h ;
    }

const int MAX_RECT = 100 ;

Rectangle clippingAreas[ MAX_RECT ] ;
```

Osserviamo che potrebbe anche avere senso definire un opportuno tipo per l'altezza e la larghezza; in ogni caso, sono evidenti i vantaggi effettivi della seconda soluzione:

- Si ha un'unica struttura che rappresenta un oggetto "rettangolo", con tutti i vantaggi che ne conseguono.

- Mentre nella prima versione si avevano due array *height* e *width*, che in ogni caso non fornivano alcuna informazione sullo scopo dei rettangoli così implementati, nella seconda versione possiamo rendere esplicito nel codice l'utilizzo dell'unico array di rettangoli, ad esempio per rappresentare delle aree di clipping.

Raccomandazione 54

Non dichiarate diverse variabili logicamente correlate se definendo una opportuna classe è possibile riunirle in un'unica struttura.

Variabili locali

Ogni variabile dovrebbe avere la vita e lo scope il più brevi possibile: in tal senso, il peggior caso è rappresentato dalle variabili globali, visibili in ogni modulo, e la cui vita è la stessa del task, mentre il caso migliore è

rappresentato da variabili che vengono dichiarate ed utilizzate all'interno di un piccolo blocco di codice, non visibili all'esterno di esso, e la cui vita è in genere breve.

Esistono diverse motivazioni a supporto di una simile pratica di programmazione, tra cui vale la pena ricordare le seguenti:

- Variabili locali ad un breve blocco di codice rendono il codice stesso semplice da capire: non è necessario tornare indietro nel testo del programma per controllare il tipo della variabile, o quali valori può avere assunto dinamicamente, o se esistono degli alias riferiti alla variabile, né preoccuparsi del successivo uso della variabile stessa. Ci si può concentrare sulle poche righe dove la variabile è dichiarata, inizializzata, utilizzata e distrutta.

- Variabili con vita breve aumentano normalmente la *località* di un programma, ovvero corrispondono a codice che tende ad utilizzare, all'interno di ogni blocco, aree molto limitate di memoria. Questo può sensibilmente aumentare l'efficienza in ogni architettura hardware con cache memory.

- Usare prevalentemente variabili con vita breve significa anche evitare l'occupazione di memoria quando la variabile stessa non è più utile; per oggetti di dimensioni rilevanti, questo può avere un impatto decisivo sia sulle prestazioni del programma (in ambienti con memoria virtuale) che sulle richieste di memoria minima del programma (in ambienti senza memoria virtuale).

- Per contro, variabili globali richiedono una continua cautela nel momento della modifica, e possono portare a problemi non banali di sincronizzazione in ambienti paralleli o concorrenti.

- Il compilatore è in genere in grado di accorgersi se una variabile locale è dichiarata ma mai utilizzata, o se una variabile locale è usata prima di essere stata inizializzata (ovvero, se compare in una espressione destra prima di essere comparsa in una espressione sinistra, in almeno uno dei percorsi possibili). Questo controllo è normalmente impossibile per le variabili globali, ed in tal modo si perde un importante supporto automatico per l'eliminazione degli errori.

Un buon metodo per garantire una vita minima alle variabili è di dichiararle all'ultimo istante possibile, nel qual caso siamo normalmente in grado di

assegnare direttamente il valore iniziale alla variabile, anziché dichiararla prima ed inizializzarla in seguito: vedere **Listato 52** e **Listato 53** per un utile confronto.

Listato 52

```
// Stile sconsigliato

void LongFunction()
  {
  Complex c ;

  // ... molte linee di codice

  c = Complex( f, x ) ;

  // ...
  }
```

Listato 53

```
// Stile migliore

void LongFunction()
  {

  // ... molte linee di codice

  Complex c( f, x ) ;

  // ...
  }
```

Notiamo che nel **Listato 53** si ha anche un ulteriore vantaggio: il codice è più compatto e veloce. Nel primo caso, infatti, l'oggetto *c* viene costruito al momento della dichiarazione, tramite un costruttore di default, ed in seguito modificato tramite assegnazione, coinvolgendo probabilmente la creazione di un oggetto temporaneo (la creazione o meno dei temporanei è largamente lasciata al compilatore). Nel secondo caso, l'oggetto viene costruito ed il valore corretto assegnato in un'unica chiamata. La differenza può essere minima per oggetti piccoli come un *Complex*, ed ancora più contenuta per tipi base come *int* o *float*, tuttavia per oggetti di grandi dimensioni può essere realmente notevole. Non solo, talvolta la creazione di un oggetto ha dei side-effect, come l'apertura di una finestra sul video: in tal caso, ha ben poco senso creare l'oggetto prima di avere tutti i parametri

necessari alla sua inizializzazione, o comunque prima di avere realmente bisogno dell'oggetto stesso: è molto meglio attendere sino all'ultimo istante. Tale raccomandazione si può generalizzare, tranne i rari casi in cui non si può avere a disposizione un valore iniziale, in quanto dipendente da molte opzioni alternative che vengono gestite da sequenze di if o di switch/case. Anche in simili situazioni, tuttavia, ci si potrebbe chiedere se non sia meglio muovere la porzione di codice complesso in una funzione separata (vedere **Listato 54** e **Listato 55**) o utilizzare una tabella di valori precalcolati.

Listato 54

```
// Stile sconsigliato

void ComplexFunction( int x )
  {
  int y = 0 ;

  if( x % 2 )
    {
    if( x % 4 )
      y = 2 ;
    else
      y = 1 ;
     }
  else if( x % 3 )
    y = 3 ;
  else
    y = 4 ;

  // utilizza y, molte linee di codice
  }
```

Listato 55

```
// Stile migliore

int StrangeMap( int x )
  {
  if( x % 2 )
    {
    if( x % 4 )
      return( 2 ) ;
    else
      return( 1 ) ;
```

```
    }
  else if( x % 3 )
    return( 3 ) ;
  else
    return( 4 ) ;
  }


void ComplexFunction( int x )
  {
  int y = StrangeMap( x ) ;

  // utilizza y
  }
```

Infine, inizializzando la variabile al momento della dichiarazione, evitiamo il rischio di utilizzarla prima di averla inizializzata, un errore abbastanza pericoloso ma relativamente comune, specialmente nell'uso dei puntatori. Ed ovviamente, assegnare un valore fittizio al momento della dichiarazione ha ben poco senso se possiamo evitarlo: meglio quindi aspettare, e dichiarare la variabile in un punto in cui può essere correttamente inizializzata.

Per completezza, va detto che alcuni trovano migliore lo stile "alla Pascal", dove tutte le variabili vengono dichiarate all'inizio della funzione o procedura, ed in seguito inizializzate ed utilizzate. La critica che in tal caso viene mossa alla tecnica del "dichiarare all'ultimo momento" è che può essere difficile, da una riga qualunque del codice, rintracciare la dichiarazione delle variabili utilizzate (mentre se sono tutte dichiarate all'inizio della funzione, ciò è immediato). Tuttavia, trovarsi in una riga a caso è abbastanza inconsueto, tranne forse all'interno di debugger interattivi, ed in ogni caso, la buona norma di dare scope e vita minimi alle variabili dovrebbe proprio rendere semplice la ricerca della dichiarazione: dovrebbe trovarsi a poche righe di distanza dal punto di utilizzo. Se una variabile locale è utilizzata a centinaia di righe dalla sua dichiarazione, probabilmente lo stile di codifica è comunque molto scadente.

Raccomandazione 55

Dichiarare le variabili in modo da avere vita e scope minimi, ovvero all'ultimo istante possibile, e con il massimo grado di annidamento possibile.

Raccomandazione 56

Inizializzare le variabili al momento della dichiarazione ogni volta che è possibile; altrimenti, considerare la possibilità di astrarre parte del codice in una funzione.

Notiamo che al fine di poter inizializzare le variabili al momento della dichiarazione, è importante dichiarare una sola variabile per volta (altrimenti il valore dovrebbe essere comune a tutte le variabili, e rimanere tale in fase di manutenzione). Questo conferma la validità della **Raccomandazione 33** a proposito del layout delle dichiarazioni.

In C++ è infine possibile ridichiarare una variabile all'interno di un blocco applicativo: la variabile dichiarata internamente al blocco nasconde allora la variabile, avente lo stesso identificatore, visibile all'esterno del blocco stesso.

Tale pratica è però talvolta fonte di malfunzionamenti inattesi, soprattutto in fase di manutenzione, in quanto è possibile introdurre errori che il compilatore non ha modo di riconoscere come tali. Vedremo ora come alcune modifiche apparentemente innocenti apportate ad un programma corretto possano portare a programmi errati; nel primo caso, partiamo da un programma corretto (**Listato 56**), consistente in una porzione di un algoritmo di visita di un grafo.

Listato 56

```
void Graph :: Visit( int startNode )
  {
  for( int i = startNode; i < nodeNum; i++ )
    {
    if( ! node[ i ].Visited()  )
      {
      node[ i ].MarkAsVisited() ;

      for( int j = 0; j < nodeNum; j++ )
        if( node[ j ].Connected( node[ i ] )
           Visit( j ) ;
      }
    }
  }
```

Introduciamo ora alcune righe a scopo di debugging, riutilizzando la variabile *i* (**Listato 57**); poiché la variabile dichiarata all'interno del *for* è visibile nel blocco che include il *for* stesso, con l'introduzione del codice di debug abbiamo mascherato la variabile *i* dal punto di introduzione in avanti, sino alla fine del blocco. Ne consegue che il riuso della variabile nel codice di debug ha causato un malfunzionamento del programma.

Listato 57

```
void Graph :: Visit( int startNode )
  {
  for( int i = startNode; i < nodeNum; i++ )
    {
    if( ! node[ i ].Visited()  )
      {
      node[ i ].MarkAsVisited() ;

      // debugging code
      for( int i = 0; i < nodeNum; i++ )
        cout << node[ i ] ;
      // ATTENZIONE: ora i vale nodeNum!

      for( int j = 0; j < nodeNum; j++ )
        if( node[ j ].Connected( node[ i ] )
           Visit( j ) ;
      }
    }
  }
```

Ancora più subdolo è il caso in cui durante la manutenzione si elimini una variabile, intenzionalmente o per sbaglio, senza possibilità da parte del compilatore di segnalare l'errore; partiamo dal **Listato 58**, che assumiamo corretto ma che utilizza la tecnica "a rischio" di ridichiarare una variabile in-scope:

Listato 58

```
void f()
  {
  int i = 10 ;

  // ....

  if( condition )
    {
```

```
    int i = 14 ;
    // ... usa i
    }

  // ....

  }
```

eliminiamo ora la dichiarazione della variabile locale *i* internamente all'if (**Listato 59**): il programma è ora errato, ma il compilatore non può segnalare il problema.

Listato 59

```
void f()
  {
  int i = 10 ;

  // ....

  if( condition )
    {
    // ... usa i
    }

  // ....

  }
```

Notiamo che se avessimo utilizzato un identificatore non in scope (ad es. *j* nel caso in esame) anziché riutilizzarne uno in scope, a fronte della modifica avremmo ottenuto un messaggio di errore dal compilatore. È sempre una buona pratica di programmazione attenersi a regole che consentono di trovare il maggior numero di errori a compile-time.

È invece perfettamente lecito riutilizzare una variabile, anche all'interno di una stessa funzione, se non ci si trova nello scope della dichiarazione precedente (**Listato 60**).

Listato 60

```
void Graph :: Visit( int startNode )
  {
  for( int i = startNode; i < nodeNum; i++ )
    {
```

```
    if( ! node[ i ].Visited()  )
      {
      node[ i ].MarkAsVisited() ;

      for( int j = 0; j < nodeNum; j++ )
        if( node[ j ].Connected( node[ i ] )
           Visit( j ) ;
      }
    else
      {
      for( int j = 0; j < nodeNum; j++ )
      // nessun problema,
      // j non è in scope
      // ...
      }
    }
  }
```

Raccomandazione 57

All'interno dei blocchi, non dichiarare variabili aventi lo stesso identificatore di una variabile visibile nello stesso scope.

Variabili static

La dichiarazione di una variabile come *static* ha due effetti profondamente diversi a seconda del contesto: se stiamo definendo una variabile globale, indichiamo in tal modo che la visibilità della variabile è a livello del modulo; se stiamo definendo una variabile locale, indichiamo che la variabile stessa dovrà conservare il suo valore, ed il relativo distruttore *non* dovrà essere chiamato, anche quando usciamo dallo scope della variabile. Se rientriamo nello scope della variabile, il relativo costruttore *non* sarà ulteriormente chiamato dopo la prima volta (nel caso, ovviamente, che si tratti di una variabile istanza di una classe con costruttore).

Variabili locali statiche sono molto utili in C per evitare l'uso di variabili globali, quando si voglia comunque conservare il valore tra una chiamata e l'altra di una funzione. Sono invece un elemento ridondante in C++, ove esiste un mezzo molto più opportuno per esprimere lo stesso concetto, ovvero la definizione della variabile come un campo (membro) di un oggetto. Consideriamo il classico caso di utilizzo di una variabile locale

statica, ovvero una funzione che genera numeri casuali, e che conserva il risultato precedente in una variabile statica: **Listato 61**.

Listato 61

```
// Warning: NON è un buon generatore random!

int Random()
  {
  static int lastValue = 17 ;

  lastValue = ( lastValue + 31 ) % 13 ;
  return( lastValue ) ;
  }
```

Una versione di gran lunga più elegante e flessibile utillizza invece una classe RandomGenerator, ove *lastValue* appare come membro: **Listato 62**.

Listato 62

```
// Warning: NON è un buon generatore random!

class RandomGenerator
  {
  public :
    int Random() ;
    void Randomize( int seed ) ;

  private :
    int lastValue ;
  } ;


int RandomGenerator :: Random()
  {
  lastValue = ( lastValue + 31 ) % 13 ;
  return( lastValue ) ;
  }

void RandomGenerator :: Randomize( int seed )
  {
  lastValue = seed ;
  }
```

Vediamo ora i vantaggi della seconda soluzione:

- Avendo una classe RandomGenerator, il codice può facilmente evolvere in una gerarchia di generatori random, dichiarando Random() e Randomize() come virtuali e ridefinendoli nelle sottoclassi.

- Possiamo avere più oggetti indipendenti di classe RandomGenerator, che non interferiscono tra loro, mentre nel caso della variabile statica, ogni chiamata alla funzione altererà il contenuto della variabile.

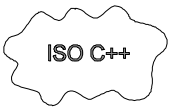
- In ambienti di programmazione che supportano il multi-threading, incapsulare in una classe le variabili locali statiche consente di mantenere un controllo sul codice che non è altrimenti possibile, in quanto le funzioni con dati statici non sono rientranti; lo stesso, come vedremo, si applica alle variabili globali. Come visto al punto precedente, con la tecnica di cui sopra possiamo invece utilizzare oggetti diversi nei diversi thread.

- Notiamo infine che, nel caso sia necessario condividere dati tra i diversi oggetti appartenenti alla classe RandomGenerator, possiamo sempre utilizzare membri dato statici all'interno della classe. Abbiamo così una maggiore flessibilità rispetto al caso della semplice variabile locale statica.

Raccomandazione 58

Considerate sempre le variabili locali *static* come candidati a divenire membri di una opportuna classe: se si trovano all'interno di funzioni membro, possono divenire membri della stessa classe; se si trovano in funzioni non-membro, dovrà essere introdotta una nuova classe, se corrisponde ad una astrazione significativa.

Variabili locali e strutture di controllo



Nella proposta di standard ISO per il C++, è stata inclusa la possibilità di dichiarare variabili locali come parte della *condizione* negli statement di *if*, *switch*, *while*, e *for*. Richiamando brevemente la sintassi:

<selection-statement> ::=
if(<condition>) <statement>
if(<condition>) <statement> else <statement>
switch(<condition>) <statement>

<iteration-statement> ::=
while(<condition>) <statement>
do <statement> while(<expression>) ;
for(<for-init-statement> <condition>$_{opt}$; <expression>$_{opt}$) <statement>

<condition> ::=
<expression>
<type-specifier> <declarator> = <expression>

si può vedere che nello standard la condizione consente la dichiarazione di variabili locali; lo scope di tali variabili va dal punto di dichiarazione sino alla fine dello statement controllato dalla condizione: vedere il **Listato 63** per un esempio.

Listato 63

```
if( int n = list.CountElems() )
  {
  //  n può essere usato qui

  // n NON può essere ridichiarata qui !!
  }
// qui siamo fuori dallo scope di n
```

Osserviamo che le variabili locali dichiarate nella condizione non possono essere ridichiarate nello statement controllato; ovviamente, possono essere ridichiarate in un blocco all'interno di esso, per quanto non sia una buona pratica di programmazione (**Raccomandazione 57**).

La possibilità di dichiarare variabili all'interno della condizione può essere utile per perseguire uno dei nostri scopi, ovvero dare ad ogni variabile la visibilità e la vita minime: non appena il vostro compilatore fornirà il necessario supporto, cercate quindi di farne l'uso adeguato.

Ulteriore delibera del comitato ANSI/ISO riguarda il trattamento delle variabili introdotte in un *branch* di selezione o in uno statement di

iterazione, che vengono implicitamente introdotte in uno scope locale, a differenza di quanto avviene in alcuni compilatori: vedere **Listato 64**. Da notare comunque che diversi compilatori oggi sul mercato implementano già questa regola; per verificare, provate a compilare il **Listato 65**: se (e solo se) il compilatore segnala un errore di "simbolo j non definito", l'interpretazione è corretta secondo il draft ISO.

Listato 64

```
// Interpretazione di alcuni compilatori
while( something )
  for( int i = 0; i < n; i++ )
    {
    // ...
    }
// i è visibile in questo punto

// Interpretazione ISO C++
//  viene introdotto uno scope locale
//  implicito dopo il while
while( something )
  {                                 // implicita
  for( int k = 0; k < n; k++ )
    {
    // ...
    }
  }                                 // implicita
// k non è visibile in questo punto
```

Listato 65

```
// test di aderenza allo standard ISO

#include <iostream.h>

int main()
  {
  int i = 0 ;
  if( i < 3 )
    for( int j = 0; j < 4; j++ )
      i++ ;
  cout << j ;

  return( 0 ) ;
  }
```

Raccomandazione 59

Non utilizzare una variabile dichiarata in uno statement di selezione o di iterazione, al di fuori dello statement stesso: il codice non rispetterà lo standard ISO C++.

Variabili globali

L'uso delle variabili globali è spesso deprecato dai programmatori, che tuttavia raramente sanno fornire delle motivazioni oggettive a supporto del loro (giusto) rifiuto. Vedremo qui di seguito alcune delle caratteristiche negative del codice che utilizza variabili globali:

- Problemi di rientranza: come già visto nel caso di variabili locali statiche, in ambienti multi-threading[9] la stessa funzione può essere chiamata da un differente thread prima che una precedente chiamata alla stessa funzione sia terminata. In questo caso, una funzione che modifichi dati globali al suo interno può trovarsi facilmente in situazioni inconsistenti, in quanto la prima invocazione della funzione può trovarsi con i dati modificati dalla seconda invocazione. Notiamo che ogni thread ha di norma un suo stack, pertanto simili problemi non coinvolgono le variabili locali.

- Problemi di aliasing: funzioni che operino in modo distruttivo sugli argomenti, e che facciano uso di variabili globali, dovrebbero sempre verificare che gli argomenti non siano puntatori ai dati globali, e gestire tali casi in un modo adeguato, ammesso che esista.

- Problemi nel riutilizzo del codice: routine che si appoggiano a variabili globali sono in genere difficili da riutilizzare in altri programmi, dove tali variabili non esistono, o peggio sono utilizzate per scopi o con criteri diversi.

- Accoppiamento tra le funzioni: ogni funzione che utilizzi una variabile globale è di fatto accoppiata a ogni altra funzione che utilizza la stessa variabile, in quanto la chiamata ad una qualunque di queste altre funzioni può alterare il risultato di una chiamata alla prima funzione.

[9] Ambienti multi-threading sono sempre più comuni, ed aumenteranno nel prossimo futuro, ed è quindi opportuno considerarne almeno le caratteristiche più rilevanti.

Questo ha un impatto estremamente negativo sulla comprensibilità del codice, in quanto si richiede la comprensione di tutti i possibili effetti incrociati.

Un suggerimento che viene spesso dato ai programmatori C++ è di creare una classe GlobalData ed introdurvi come membri tutte le variabili globali, creare un oggetto di tale classe, e fare riferimento ad esso dovunque si debba utilizzare un elemento globale.

Ciò può a prima vista sembrare una buona soluzione: in fondo, è sostanzialmente ciò che è stato suggerito nel caso delle variabili locali static. Occorre comunque chiedersi se tale soluzione risolva, almeno in parte, i problemi evidenziati sopra; a dire il vero, non solo non li risolve, ma ne aggiunge uno nuovo: l'uso di dati globali è meno esplicito e quindi chi legge il codice potrebbe non prestare la necessaria cautela: funzioni che prendano in input un puntatore a tale struttura potrebbero di fatto modificare ogni membro della struttura stessa, ovvero ogni variabile globale.

L'unica soluzione accettabile, una volta minimizzato il numero di variabili globali utilizzate, è di costruire una opportuna classe intorno ad ogni *singola* variabile globale (non un'unica struttura che le contenga tutte): in questo modo, trattiamo ogni variabile globale come una *risorsa* e forniamo un accesso controllato alla risorsa stessa. Le funzioni di accesso dovrebbero essere al più alto livello possibile, includendo quindi se necessario l'opportuno uso di semafori o altre strutture di locking.

Normalmente, tali classi possono essere costituite interamente da dati statici e funzioni statiche, avere costruttore e distruttore privati, e dichiarare un oggetto della classe come membro della classe stessa: in tal modo, si ottiene la creazione automatica dell'oggetto, e nessuna possibilità di creare istanze multiple della classe.

Prima di vedere un esempio, è necessario rimarcare per la seconda volta che per ogni variabile globale originaria occorre definire una classe: non bisogna assolutamente inserire più variabili globali come membri, a meno che non contribuiscano di fatto a definire un'unica risorsa (in qual caso, si dovrebbe riflettere sul perché non siano già strutturate come classe).

Consideriamo ora un semplice programma, con una funzione che dipende da una variabile globale (**Listato 66**):

Listato 66

```
char* screen ; // fornisce accesso diretto alla memoria
video

// svariate funzioni che operano su screen...

void Write( int x, int y, char ch )
  {
  // ...
  }

void GetCharEcho()
  {
  // ...
  }

// etc.
```

Una versione ristrutturata secondo la tecnica vista precedentemente è data nel **Listato 67**; osserviamo che non solo abbiamo incapsulato in una classe le funzioni di base, ed abbiamo quindi creato un framework adatto alle estensioni ed alla manutenzione, ma abbiamo in effetti creato una vera e propria risorsa "memoria video".

Listato 67

```
/* SCREENMEM.H */

class ScreenMemory
  {
  public :
    static void Write( int x, int y, char ch ) ;
    // altre primitive di accesso
  private :
    ScreenMemory() ;      // costruttore e distruttore
privati
    ~ScreenMemory() ;

    static char* screen ;
    static ScreenMemory initializer ;
    // forza richiamo automatico del costruttore
  } ;



/* SCREENMEM.CPP */
```

```
#include <assert.h>
#include "screenmem.h"

// Dichiarazioni dei dati statici

ScreenMemory ScreenMemory :: initializer ;

char* ScreenMemory :: screen = NULL ;


ScreenMemory :: ScreenMemory()
  {
  assert( screen == NULL ) ;
  // verifica che non venga mai richiamato
  // più di una volta.

  // ... inizializza screen ;
  }


ScreenMemory :: ~ScreenMemory()
  {
  // ...
  }


void ScreenMemory :: Write( int x, int y, char ch )
  {
  // usa la variabile screen
  }



/* USE.CPP */

#include "screenmem.h"

int main()
  {
  ScreenMemory :: Write( 10, 10, 'A' ) ;
  // accesso tramite classe, non tramite
  // oggetto globale!

  return( 0 ) ;
  }
```

Chi non avesse mai utilizzato una simile tecnica in C++ potrebbe aver bisogno di studiare il listato con attenzione, e magari eseguire dei test su qualche caso reale, inserendo dei comandi di output nel costruttore e nel distruttore per verificare l'ordine di esecuzione. Una variante della tecnica su esposta verrà discussa nel capitolo 7, nel paragrafo "Ordine di inizializzazione".

Come potete osservare, la seconda versione è considerevolmente più prolissa della prima; tuttavia, in questo caso la maggiore lunghezza va considerata come una connotazione positiva: il carattere deleterio delle variabili globali risiede in gran parte proprio nell'immediatezza, che porta facilmente all'abuso. Se l'introduzione di oggetti globali comporta il progetto e l'implementazione di una classe, i programmatori saranno molto più cauti nell'introdurre oggetti globali, riconsiderando eventuali decisioni prese con leggerezza, e (si spera) si limiteranno ad utilizzare la tecnica qui esposta nei casi in cui sia realmente necessario.

Raccomandazione 60

Minimizzare l'uso delle variabili globali.

Raccomandazione 61

Se è necessario introdurre un oggetto globale, costruire una opportuna classe che gestisca l'elemento come una risorsa, eventualmente con gli opportuni costrutti di locking.

7

Classi

"The system of private property is the most important guarantee to freedom..."
Friedrich August von Hayek

La classe è indubbiamente l'elemento fondamentale del C++, che non a caso nella sua prima incarnazione era chiamato "C with Classes"; essa rappresenta l'unità di incapsulazione, e fornisce la base per l'ereditarietà e per il polimorfismo. In questo capitolo ci occuperemo prevalentemente della classe *in sé*, tranne alcuni necessari cenni alle relazioni di ereditarietà tra classi, che verranno invece approfondite più oltre nel testo.

Visibilità: public, protected, private

Ad ogni membro, dato o funzione, di una classe può essere imposta una diversa visibilità dall'esterno: membri privati sono visibili solo all'interno della classe[10], membri protetti solo nella classe ed in quelle derivate, membri pubblici sono visibili ovunque.

Un dettaglio che passa molto spesso inosservato a chi non conosce altri linguaggi object oriented è che l'unità di protezione del C++ è la classe, non l'oggetto. Con riferimento al **Listato 68**, la funzione *f* modifica un dato privato di un oggetto *c* passato come parametro: se l'unità di protezione fosse l'oggetto, e non la classe, simile codice non sarebbe valido, ed una funzione membro potrebbe accedere solo ai dati privati di *this* e non di altri oggetti.

[10] con l'ovvia eccezione delle funzioni friend della classe.

Listato 68

```
class c
  {
  public:
     f( c* item ) { item->x = 10 ; }

  private :
    int x ;
  } ;
```

Alcuni ritengono che l'unità di protezione debba comunque essere l'oggetto e non la classe, e cercano quindi di evitare codice come quello del **Listato 68**; in generale, si tratta invece di una caratteristica che può tornare utile in parecchi casi, ed essendo parte integrante della definizione del linguaggio è piuttosto insensato escluderla dalla pratica della programmazione. L'unica raccomandazione potrebbe essere quella di commentarne l'uso se si ritiene che la comprensione possa essere non immediata, ma ciò vale in generale per ogni costrutto.

Ad uno sguardo più attento, appare evidente come il meccanismo di protezione segua in realtà regole più complesse se è coinvolta l'ereditarietà: nel **Listato 69** sia la classe *Derived* che la classe *OtherDerived* hanno accesso all'elemento protetto *x* della classe *Base*. Tuttavia, *OtherDerived* non può accedere all'elemento *x* di un oggetto di classe *Derived*: può solo accedere al membro *x* di un oggetto di classe *OtherDerived* o di una classe derivata da essa.

Listato 69

```
class Base
  {
  protected:
    int x ;
  } ;


class Derived : public Base
  {
  } ;


class OtherDerived : public Base
  {
```

```
  public :
    void f( OtherDerived* item ) { item->x = 10 ; }
    // OK
    void f( Derived* item ) { item->x = 10 ; }
    // Errore
  }
```

Nella release 1.0 del CFront, il codice del **Listato 69** era invece perfettamente legale, mentre ora ogni compilatore dovrebbe segnalare un errore di accesso al membro *x*. In ogni caso, come vedremo più avanti, l'uso di dati protetti è sconsigliabile come pratica di programmazione.

Un ulteriore punto che passa facilmente inosservato a chi si avvicina al C++ è che il concetto di *private* è piuttosto debole, in quanto funzioni virtuali private possono comunque essere ridefinite nelle classi derivate. Torneremo su questo punto nel capitolo 10; in ogni caso, osservando il **Listato 70** possiamo vedere come si possa non solo ridefinire una funzione virtuale privata in una classe derivata, ma addirittura modificarne la visibilità da privata a pubblica.

Listato 70

```
#include <iostream.h>


class Base
  {
  public :
    void Show() { cout << ClassName() ; }

  private :
    virtual const char* ClassName()
      { return( "base" ) ; }
  } ;


class Derived : public Base
  {
  public :
    virtual const char* ClassName()
      { return( "Derived" ) ; }
  } ;


int main() ;
  {
```

```
  Derived d ;
  Base* b = &d ;

  b->Show() ; // stampa “Derived”

  return( 0 ) ;
  }
```

Considerando che le funzioni private dovrebbero essere strettamente legate all’implementazione della classe, e non rilevanti al di fuori di essa, ci si dovrebbe chiedere se abbia o meno senso dichiararle come *virtual*. In effetti, nella maggior parte dei casi non ha senso, e può portare a comportamenti piuttosto strani, come visto poc'anzi. Funzioni *virtual* dovrebbero di norma essere protette o pubbliche; purtroppo, anche in questo caso, una classe derivata può cambiare la visibilità di una funzione virtuale, ad esempio da protetta a pubblica o a privata. Vi sono rari casi in cui ciò è opportuno: ad esempio, per impedire a classi derivate di accedere alla funzione; tuttavia, di norma dovrebbero essere esplorate altre soluzioni, come l’uso dell’ereditarietà privata o il contenimento, o commentare esplicitamente le ragioni della scelta.

Raccomandazione 62

Non dichiarare funzioni *private* come *virtual*.

Raccomandazione 63

Se si modifica la visibilità di una funzione in una classe derivata, motivare le ragioni della modifica con un opportuno commento.

Costruttori e Distruttori

Il costruttore ha il compito di inizializzare opportunamente un oggetto, ponendolo in uno stato valido; il distruttore dovrebbe gestire il rilascio delle risorse possedute dall'oggetto, ed eseguire ogni azione necessaria al momento della terminazione dell'oggetto stesso. In C++, i costruttori non possono avere altro nome che quello della classe cui appartengono, per cui l'unica possibilità di avere più costruttori è che abbiano parametri diversi. La possibilità di overloading per i costruttori è spesso abusata: citando Stroustrup da [Str91], "quando si progetta una classe è sempre presente la tentazione di attribuirle tutte le caratteristiche possibili [...] Decidere cosa sia realmente necessario richiede maggiore riflessione, ma conduce generalmente a programmi di minori dimensioni e più comprensibili". Particolare attenzione va pagata ai costruttori con un solo parametro, che costituiscono di fatto degli operatori di conversione: torneremo su questo punto nel capitolo 12, parlando del casting.

Un classico errore che praticamente ogni principiante commette è l'uso di funzioni virtuali all'interno di un costruttore. Ricordate sempre che, nel momento dell'invocazione del costruttore di una classe base, la tavola delle funzioni virtuali è quella della classe base, anche se stiamo creando un oggetto di classe derivata. Il **Listato 71** illustra il problema molto chiaramente:

Listato 71

```
#include <iostream.h>


class Base
  {
  public :
    Base() { Show() ; }    // chiama una funzione
virtuale
    virtual void Show() { cout << "base" ; }
  } ;


class Derived
  {
  public :
    virtual void Show() { cout << "derived" ; }
  } ;
```

```
int main()
  {
  Derived d ; // stampa “base” !!

  return( 0 ) ;
  }
```

La ragione per il comportamento, solo apparentemente strano, dei costruttori è in realtà molto semplice: quando costruiamo un oggetto di classe derivata, vengono dapprima invocati i costruttori per le classi base. Ciò significa che l’oggetto di classe derivata non si può realmente considerare costruito (ovvero, come detto sopra, “in uno stato valido”) finché siamo all’interno di uno dei costruttori per le classi base. Chiamare una funzione che operi sull’oggetto di classe derivata sarebbe pertanto pericoloso: pensate se, nell’esempio dato, la funzione Derived :: Show() accedesse ad alcuni membri della classe Derived; non essendo questi ancora inizializzati si potrebbero avere malfunzionamenti di varia gravità: basti pensare a membri di tipo puntatore.

Un comportamento del tutto analogo ha luogo nei distruttori, dove nuovamente le chiamate a funzioni virtuali vengono legate staticamente. Anche in questo caso la ragione va ricercata nella volontà di impedire la chiamata di funzioni su oggetti inconsistenti. La sequenza di distruzione è l'opposta di quella di costruzione[11] (viene quindi invocato prima il distruttore delle classi derivate, risalendo via via la gerarchia di ereditarietà). Se durante l'esecuzione del distruttore di classe base venisse chiamato un metodo di classe derivata, questo opererebbe su un oggetto inconsistente.

[11] La direzione opposta durante costruzione e distruzione è un elemento ricorrente nel design del C++: si pensi alla sequenza costruzione / distruzione di variabili locali, ecc. Per chi non ha mai avuto modo di soffermarsi sull'argomento, si tratta di uno spunto di riflessione interessante, che peraltro aiuterà a comprendere meglio alcuni paradigmi di programmazione C++ moderna come il RAII (Resource Acquisition Is Initialization).

Raccomandazione 64

Non eccedere nel numero di costruttori: introducete solo quelli realmente necessari. Attenzione ai costruttori con un solo parametro, che sono a tutti gli effetti operatori di conversione.

Raccomandazione 65

Non chiamare funzioni virtuali all'interno dei costruttori o del distruttore: non verranno in ogni caso legate dinamicamente, ma sempre legate staticamente. Se è necessario chiamarle, introducete comunque un commento esplicativo.

Normalmente i costruttori richiedono un certo numero di parametri per inizializzare correttamente l'oggetto: sarebbe buona norma non modificare il valore dei parametri passati ai costruttori tramite puntatore o per riferimento. A tal fine, è opportuno dichiarare detti parametri come const. Vi sono certamente delle eccezioni (ad esempio, la tecnica delle classi duali, [Ada94]), ma in genere la modifica dei parametri nei costruttori può portare a problemi in fase di manutenzione; consideriamo il **Listato 72**: poiché il costruttore di copia modifica il suo unico parametro, se scambiamo la posizione della dichiarazione di *b2* e *b3* all'interno del main, otteniamo valori diversi per i due oggetti.

Listato 72

```
class Bad
  {
  public :
    Bad() { x = 0 ; }
    Bad( Bad& b ) { x = b.x++ ; }
  private :
    int x ;
  } ;


int main()
  {
  Bad b1 ;

  Bad b2( b1 ) ;
  Bad b3( b1 ) ;
```

```
  return( 0 ) ;
  }
```

Simili effetti collaterali sono molto difficili da ricordare in fase di manutenzione, e richiederebbero in ogni caso un adeguato commento e maggiore attenzione durante le modifiche. Da notare che mentre in una comune chiamata di funzione, con parametri passati per valore o per riferimento, chi legge il codice fa normalmente attenzione a possibili effetti collaterali sui parametri, ciò spesso non accade per i costruttori, in quanto si assume spesso che l'ordine di dichiarazione di variabili dichiarate in modo strettamente consecutivo non sia rilevante ai fini della correttezza del programma.

Raccomandazione 66

Parametri di tipo puntatore o reference nei costruttori dovrebbero sempre essere const, in modo tale che il costruttore non modifichi i suoi parametri.

Distruttori virtuali

La conclusione del paragrafo precedente si può riassumere brevemente come "non esistono costruttori virtuali"[12]. Lo stesso non si può dire dei distruttori: consideriamo il codice del **Listato 73**: un oggetto viene costruito ed in seguito distrutto, tramite un puntatore ad una classe base. Poiché il distruttore non è dichiarato come *virtual*, verrà solo chiamato il distruttore della classe base, e non il distruttore della classe derivata: gli increduli possono eseguire il programma e verificarne l'output.

Listato 73

```
#include <iostream.h>


class Base
  {
  public :
    ~Base() { cout << "distruttore base" ; }
  } ;
```

[12]Per quanto sia possibile invece simularli, pur con qualche scomodità; vedere ad es. [Str91].

```
class Derived : public Base
  {
  public :
    ~Derived() { cout << “distruttore derived “ ; }
  } ;


int main()
  {
  Base* b = new Derived() ;
  delete b ;

  return( 0 ) ;
  }
```

Il problema nel caso in esame è che il distruttore non viene automaticamente inserito nella tavola delle funzioni virtuali, ma solo su richiesta del programmatore. Le ragioni di questa scelta sono da ricercare nel desiderio di non aggiungere alcun overhead, in termini di spazio o di tempo di esecuzione, se non vengono usate esplicitamente le funzioni virtuali. Tali motivazioni hanno profondamente influito sull'intero design del C++, come ben evidenziato da Stroustrup in [Str94], talvolta anche a costo di un compito più gravoso per il programmatore.

Non è infatti semplice decidere se dichiarare o meno *virtual* il distruttore; in generale, se abbiamo delle funzioni virtuali è opportuno dichiarare virtual anche il distruttore: l'overhead sarà minimo (una entry aggiuntiva nella tavola delle funzioni virtuali per la classe) ed in ogni caso la presenza di funzioni virtuali ci fa pensare che, nell'uso immediato o futuro, nuove classi verranno derivate dalla classe in questione. In tal caso, è importante che il distruttore sia dichiarato *virtual* per evitare i problemi visti sopra.
Anche in assenza di funzioni virtuali, il distruttore deve essere dichiarato virtual se vi sono classi derivate dalla classe in esame, viceversa ricadremo negli stessi problemi: in questo caso, l'overhead necessario va semplicemente accettato come inevitabile.
Solo in assenza di funzioni virtuali e di derivazione, il distruttore può essere dichiarato non-virtual, se desideriamo evitare l'overhead di un puntatore alla tavola delle funzioni virtuali per ogni oggetto. In questo caso, è caldamente consigliabile aggiungere un commento all'header della classe, specificando che la derivazione di una classe richiede la modifica del distruttore in *virtual*. Va detto che casi simili sono abbastanza rari, poiché si tratta di classi così coesive e poco accoppiate al resto del sistema da essere del tutto isolate. Un buon esempio potrebbe essere una classe per i numeri

complessi, che di norma non ha metodi virtuali e dalla quale difficilmente verranno derivate nuove classi: in tal caso, possiamo evitare lo spreco di spazio e di tempo e lasciare il distruttore non-vitrual.

Raccomandazione 67

Ogni classe avente funzioni virtuali, o utilizzata come classe base in una gerarchia di derivazione, deve dichiarare il distruttore come *virtual*. Negli altri casi, commentate opportunamente l'header della classe, così che l'introduzione di classi derivate coincida con la modifica del distruttore in *virtual.*

Oggetti composti

Quando un oggetto è formato dall'aggregazione di sotto-oggetti, abbiamo due opportunità per inizializzare le parti all'interno del costruttore, esemplificate nel **Listato 74**:

Listato 74

```
class Engine
  {
  public :
    Engine() ;
    Engine( HP power ) ;
  } ;


class Wheel
  {
  public :
    Wheel() ;
    Wheel( int diameter ) ;
  } ;


class Car
  {
  public :
    Car( HP power, int wheelDiameter )
  private :
    Engine engine ;
    Wheel wheel ;
  } ;
```

```
// Attenzione: le due alternative seguenti sono
// mutuamente esclusive: sono presentate nello stesso
// listato solo per comodita' tipografica.

// inizializzazione con assegnazione
Car :: Car( HP power, int wheelDiameter )
  {
  engine = Engine( power ) ;
  wheel = wheel( wheelDiameter ) ;
  }

// alternativa: inizializzazione con <whole> : <part>
Car :: Car( HP power, int wheelDiameter )  :
          engine( power ) ,
          wheel( wheelDiameter )
  {
  }
```

Nel primo caso (inizializzazione con assegnazione) viene dapprima richiamato il costruttore di default per i sotto-componenti *engine* e *wheel*, poi vengono creati due oggetti temporanei che vengono assegnati ai sottocomponenti. Nel secondo caso (inizializzazione con <whole> : <part>) vengono invece costruiti direttamente gli oggetti desiderati. La seconda tecnica è quindi da preferire in ogni caso, tranne quando non sia possibile avere a disposizione i valori corretti sin dall'inizio, ovvero quando i parametri per il costruttore non-default vengono calcolati in qualche punto all'interno del costruttore per l'oggetto contenitore. Spesso, in tal caso si può ugualmente utilizzare la tecnica <whole> : <part> definendo delle funzioni apposite per il calcolo di tali parametri: non dimentichiamo che è possibile utilizzare espressioni complesse anche all'interno di una inizializzazione.
Vale la pena di ricordare che, nel caso di tipi base, non vi è alcun overhead associato alla tecnica dell'assegnazione, che in tal caso è di norma preferibile per la sua maggiore leggibilità.

Raccomandazione 68

Nei costruttori di oggetti composti da più parti, inizializzate i sotto-oggetti con la tecnica <whole> : <part>, non tramite assegnazione. L'assegnazione è di norma preferibile per i sottocomponenti di tipo base.

Costruttori di copia

Quando un oggetto viene inizializzato tramite assegnazione, come nel **Listato 75**, viene utilizzato il cosiddetto costruttore di copia della classe.

Listato 75

```
class C
  {
  // ...
  }


int main()
  {
  C x ;

  C y = x ;

  return( 0 ) ;
  }
```

Se tale costruttore non è dichiarato nell'interfaccia della classe, il compilatore genera un codice di default per un costruttore di copia member-wise, corrispondente ad una copia bit-a-bit per ogni membro della classe che non disponga a sua volta di un costruttore di copia, e nella chiamata al costruttore di copia dei membri che lo definiscono[13].

Se una classe ha dei membri di tipo puntatore, dovrebbe sempre definire un costruttore di copia (e non solo, come vedremo oltre), viceversa ci si troverà quasi sicuramente in condizioni di aliasing imprevisti a run-time. Il **Listato 76** fornisce un esempio di codice errato.

Listato 76

```
#include <string.h>


class String
  {
  public:
    String( const char* s ) ;
```

[13]Nelle versioni del CFront precedenti alla 2.0, veniva sempre utilizzata la copia bit-a-bit; successivamente la regola è stata modificata come dal testo.

```
    ~String() ;
  private :
    char* cstring ;
  } ;

String :: String( const char* s )
  {
  int len = strlen( s ) ;
  cstring = new char[ len + 1 ] ;
  strcpy( cstring, s ) ;
  }

String :: ~String()
  {
  delete[] cstring ;
  }

int main()
  {
  String s( "prova" ) ;
  String c = s ;
  // quando s è distrutta, s :: cstring è rilasciato
  // quanto c è distrutta, c :: cstring è rilasciato,
  // ma è un alias per s :: cstring che è gia' stato
  // rilasciato!

  return( 0 ) ;
  }
```

Poiché non esiste un costruttore di copia, il C++ esegue una copia bit-wise dei puntatori; da questo momento, ogni modifica a *s* modificherà anche *c*, e viceversa. Peggio ancora, al momento della distruzione l'area puntata dal membro *cstring* di *s* e *c* verrà rilasciata due volte.

Una versione corretta del **Listato 76** è data nel **Listato 77** (in realtà manca ancora la gestione dell'operatore di assegnazione, discusso in questo stesso capitolo nel paragrafo "Operatore di assegnazione").

Listato 77

```
#include <string.h>


class String
  {
  public:
```

```
    String( const char* s ) ;
    String( const String& s ) ;
    ~String() ;
  private :
    void Init( const char* s ) ;
    char* cstring ;
  } ;

String :: String( const char* s )
  {
  Init( s ) ;
  }

String :: String( const String& s )
  {
  Init( s.cstring ) ;
  }

void String :: Init( const char* s )
  {
  int len = strlen( s ) ;
  cstring = new char[ len + 1 ] ;
  strcpy( cstring, s ) ;
  }

String :: ~String()
  {
  delete[] cstring ;
  }

int main()
  {
  String s( “prova” ) ;
  String c = s ;  // viene creata una nuova stringa

  return( 0 ) ;
  }
```

Talvolta è necessario impedire che nuovi oggetti possano essere creati come copia di altri oggetti: ad esempio, nel caso di risorse uniche che non possono essere duplicate, come porte di comunicazione, finestre video, eccetera; in tali situazioni, è sufficiente dichiarare il costruttore di copia come privato, e poi non definirlo affatto nel codice (fornendo un opportuno commento nell’header). Definendolo come privato, otteniamo un messaggio di errore dal compilatore se tentiamo di costruire un oggetto tramite copia; non definendolo, otterremmo comunque un messaggio di

errore dal linker se tentassimo erroneamente di utilizzarlo nel codice della classe, o all'interno di una funzione friend. Analogo trattamento, come vedremo, può essere riservato all'operatore di assegnazione. Il **Listato 78** propone un esempio di quanto sopra.

Listato 78

```
class CannotCopy
  {
  public :
    CannotCopy()
  private :
    CannotCopy( const CannotCopy& cc ) ;
    // NON IMPLEMENTATO!
  } ;


int main()
  {
  CannotCopy cc ;

  CannotCopy copy = cc ; // ERRORE di compilazione

  return( 0 ) ;
  }
```

Infine, notiamo che non è necessario che una classe implementi il costruttore di copia in modo tale che venga a tutti gli effetti creata una copia dei membri dinamici: gli oggetti allocati dinamicamente potrebbero, in determinate situazioni, essere tranquillamente condivisi (come avviene in molte implementazioni "copy-on-write" della classe String). In tal caso, si utilizza solitamente un reference count per evitare di deallocare prematuramente gli oggetti condivisi. Tuttavia è fondamentale definire il costruttore di copia, nel modo che si ritiene opportuno, per evitare sharing indesiderati ed incontrollati; spesso la condivisione totale (shallow copy) o la duplicazione totale (deep copy) sono troppo "estreme" per essere adeguate, ed in particolare la deep copy ha un overhead eccessivo quando sono coinvolti oggetti temporanei: una interessante (per quanto migliorabile) strategia, detta "delle classi duali" è stata presentata su [Ada95].

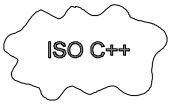
Raccomandazione 69

Una classe che abbia membri allocati dinamicamente deve definire un opportuno costruttore di copia.

Raccomandazione 70

Una classe che abbia membri allocati dinamicamente deve definire un opportuno distruttore.

Ordine di inizializzazione



Quando si dichiarano oggetti globali, occorre prestare estrema cautela alle interdipendenze tra essi (o da altri oggetti globali come *cout* e *cin*); al momento, in mancanza di uno standard più dettagliato del linguaggio, l'unica garanzia è che gli oggetti definiti in un modulo verranno inizializzati nell'ordine di definizione. Non è definito alcun ordine per l'inizializzazione degli oggetti definiti nei diversi moduli.

Una proposta per lo standard ISO, formulata da John Wilkinson di Silicon Graphics, consiste nell'inizializzare tutti gli oggetti globali definiti in un modulo nell'ordine di definizione, e garantire che tutti gli oggetti globali che appaiono in un modulo siano inizializzati prima che un qualunque oggetto o funzione definito nel modulo sia usato in altri moduli. Ciò corrisponde sostanzialmente ad inizializzare gli oggetti globali usando un misto di ordine di inizializzazione ed ordinamento topologico basato sull'utilizzo degli oggetti.

Va tuttavia tenuto presente che al momento non si hanno garanzie esplicite sull'ordine di inizializzazione tra i diversi moduli. Ciò rende particolarmente pericoloso l'uso di oggetti globali nei costruttori, in quanto se il costruttore in questione è usato per inizializzare un oggetto globale, potrebbe far riferimento ad oggetti non ancora inizializzati.

Esistono sostanzialmente due tecniche per risolvere il problema dell'ordine di inizializzazione, ovvero per garantire che alcune inizializzazioni siano state portate a termine prima di eseguire il codice di alcune funzioni membro di una classe. Il primo è di utilizzare un flag statico all'interno della classe (che in quanto statico è inizializzato a zero dal compilatore) ed aggiungere un test all'inizio di ogni funzione membro, come nel **Listato 79**. Purtroppo oltre ad essere molto scomodo, può avere un overhead significativo se le funzioni eseguono compiti molto semplici, come sarebbe invece auspicabile per una funzione.

Listato 79

```
class NeedInit
  {
  public :
    void f1() ;
    void f2() ;
    // ...
  private :
    void init() ;
    static int initialized ;
  } ;

void NeedInit :: f1()
  {
  if( ! initialized )
    {
    init() ;
    initialized = 1 ;
    }
  // "vero" codice di f1
  }

void NeedInit :: f2()
  {
  if( ! initialized )
    {
    init() ;
    initialized = 1 ;
    }
  // "vero" codice di f2
  }


// ecc
```

La seconda tecnica è più complessa da capire ma più breve, sicura ed efficiente da implementare. È stata introdotta inizialmente da Jerry Schwarz durante l'implementazione della libreria di I/O su stream, che presenta l'ovvio problema di richiedere l'inizializzazione di *cin* e *cout* prima di ogni altro oggetto definito nei moduli che la utilizzano.

L'idea di base è di definire una classe ausiliaria, il cui compito è di inizializzare la classe principale; per evitare inizializzazioni multiple, la classe ausiliaria utilizza un flag, come nel caso precedente. La differenza è che, nell'header della classe principale, non solo dichiariamo la classe principale e quella ausiliaria, ma definiamo anche una variabile statica di classe ausiliaria. Ogni modulo che includa l'header della classe principale definirà quindi una sua variabile statica di classe ausiliaria, che per la regola vista sopra verrà inizializzata prima delle variabili definite successivamente. Ciò garantisce che l'effettiva inizializzazione della classe primaria avvenga prima di ogni possibile uso all'interno del modulo che include l'header della classe (in sostanziale accordo con la proposta ISO, ma qui concretizzata dal programmatore).
La tecnica è illustrata nel **Listato 80** con riferimento agli stream globali di I/O.

Listato 80

```
// IOSTREAM.H

// oltre alla dichiarazione delle classi primarie e
degli oggetti globali
// (come extern), viene dichiarata una classe
ausiliaria:

class io_counter
  {
  public :
    io_counter() { if( count++ == 0 )
      { /* inizializza cin, cout, ecc */ } }
    ~io_counter() { if( --count == 0 )
      { /* clean-up cin, cout, ecc */ } }
  private :
    static int count ; // inizializzato a 0
  } ;

static io_counter io_init ;
  // richiede chiamata ad io_counter() e
  // garantisce inizializzazione in ogni modulo
```

Se non avete molta esperienza con il C++, l'esempio sopra può risultare un po' ostico da capire a fondo; potrebbe esservi allora utile provare ad implementare un meccanismo simile in un programma di prova, ed a tracciare l'ordine di esecuzione inserendo dei comandi di output o all'interno di un debugger.

Vi è purtroppo un problema non banale associato alla tecnica su esposta: ogni modulo che includa l'header file dichiarerà una variabile statica, che dovrà essere inizializzata al momento del caricamento del programma. Ciò significa che, in sistemi con caricamento dinamico del codice, gran parte del codice stesso dovrà essere caricato in memoria (e presumibilmente scaricato in seguito) durante la fase di inizializzazione; questo può avere conseguenze negative sui tempi di caricamento di programmi composti da molti moduli. Si tratta in genere di un costo accettabile, in quanto pagato solo al momento del caricamento, e che non intacca quindi in modo significativo la validità della tecnica.

Raccomandazione 71

Evitare, se possibile, l'uso di oggetti globali all'interno dei costruttori. Se è necessario, utilizzate la tecnica di Schwarz per garantire il corretto ordine di inizializzazione.

Costruttori e distruttori inline

L'uso delle funzioni inline è trattato più ampiamente nel capitolo 9, tuttavia è qui opportuno soffermarsi su alcune particolarità dei costruttori e dei distruttori.
A differenza delle altre funzioni membro, i costruttori ed i distruttori vengono infatti *concatenati* automaticamente dal compilatore: se costruiamo un oggetto di classe derivata, richiamiamo anche i costruttori per le classi genitore, e così via ricorsivamente sino alle classi base; analogamente per il distruttore. Ciò avviene sia nel caso in cui si specifichi la chiamata al costruttore delle classi genitore nella definizione del costruttore della classe derivata, sia che non la si specifichi, nel qual caso viene chiamato il costruttore di default per le classi genitore.

Definendo un costruttore o un distruttore come *inline*, il rischio di una esplosione nelle dimensioni del codice è abbastanza significativo e potenzialmente difficile da controllare, in quanto dipende dalla profondità

della gerarchia di derivazione, un elemento normalmente incognito quando sviluppiamo una classe base, che potrebbe essere riutilizzata in molti contesti diversi.

D'altro canto, in alcuni casi definire il costruttore ed il distruttore come *inline* può portare notevoli benefici senza alcun effetto collaterale: pensate a casi come *Complex*, una classe per numeri complessi, la quale difficilmente sarà la base di una gerarchia di derivazione. In tal caso, il codice potrebbe non solo essere più veloce se espanso inline, ma anche di dimensioni più ridotte: un costruttore per *Complex* richiede in genere solo un paio di assegnazioni, ed il distruttore è vuoto; in una simile situazione, l'overhead di una chiamata di funzione diventa significativo e va se possibile[14] evitato.

È quindi difficile, come spesso accade, dare regole di validità universale; forse la migliore strategia è di definire i costruttori ed i distruttori inline quando sono molto semplici, ed appartengono a classi non derivate e dalle quali difficilmente verranno derivate nuove classi. I costruttori ed i distruttori vuoti (considerandone il solo body, non l'eventuale chiamata ai costruttori delle classi base) possono sempre essere definiti inline, se siamo ragionevolmente sicuri che nella evoluzione della classe essi rimarranno comunque vuoti. In tutti gli altri casi, è opportuno definire costruttori e distruttori come funzioni non-inline.

Raccomandazione 72

I costruttori ed i distruttori non dovrebbero essere *inline*, tranne per classi base molto semplici, dalle quali non si deriverà in futuro, o nei casi in cui siano vuoti.

Errori a run-time nei costruttori

Uno dei classici problemi nella programmazione in C++ è "come gestire un costruttore che fallisce", in genere perché una risorsa (memoria, file, handle) non può essere acquisita o perché il costruttore di un sotto-componente fallisce a sua volta.

[14]Ricordate comunque che il compilatore non è tenuto ad espandere in linea le funzioni inline: come la keyword "register", anche "inline" rappresenta solo un suggerimento del programmatore, non una richiesta inderogabile.

Sino alla release 3.0 del CFront, non vi erano soluzioni complete ed eleganti all'interno del linguaggio: se non si volevano ignorare totalmente i possibili errori a run-time nei costruttori (il che avveniva più spesso di quanto non fosse piacevole ammettere), l'unica alternativa era di implementare un sistema a due fasi: nel costruttore, non veniva eseguita nessuna istruzione critica, tranne la memorizzazione dei parametri o altre attività che non richiedessero allocazione di risorse. La vera e propria inizializzazione era demandata ad un metodo di Init() che doveva essere chiamato dall'esterno, subito dopo aver costruito l'oggetto. Se Init() falliva, oltre a restituire un opportuno codice di errore, modificava un apposito flag di "valido" all'interno dell'oggetto, che veniva verificato all'interno di tutti i metodi della classe e/o nel punto di istanziazione.
Tale gestione, oltre ad essere inefficiente e prolissa, di fatto rendeva il meccanismo dei costruttori pressoché inutile, poiché il chiamante doveva comunque eseguire una ulteriore chiamata alla funzione Init().

Va osservato che il problema non era aggirabile con semplicità: i costruttori non possono restituire un valore, e modificare questa peculiarità del C++ avrebbe richiesto una sostanziale modifica della grammatica del linguaggio.

La vera soluzione è stata introdotta grazie alle *eccezioni* a run-time; in tal modo, un costruttore che fallisce può generare una opportuna eccezione, che verrà gestita dal chiamante o, in mancanza di una gestione apposita, propagata al livello di nesting precedente.

L'unica cautela riguarda la distruzione di questo oggetto semi-costruito: se il costruttore solleva una eccezione, il distruttore per l'oggetto *non* viene eseguito. Ne consegue che il costruttore deve essere in grado di liberare ogni risorsa allocata sino al punto in cui viene generata l'eccezione, ovviamente *prima* di sollevare l'eccezione stessa. La tecnica può divenire non banale da gestire correttamente, e questo è un ulteriore punto a favore dell'aggregazione per contenimento diretto anziché tramite puntatori (vedere anche il capitolo 14). Infatti, il problema in questione si pone solo per gli oggetti che allochiamo dinamicamente all'interno del costruttore, non per le sotto-parti dell'oggetto che vengono inizializzate con la sintassi <whole> : <part>. Non a caso, uno dei metodi classici per evitare di dover gestire la deallocazione delle risorse allocate nel costruttore, analizzato ad es. in [Str91], consiste nel creare classi wrapper ed aggregare oggetti di tali classi, anziché puntatori ad oggetti.

Membri dato pubblici e protetti

Il C++ consente di dichiarare i membri dato di una classe come privati, protetti o pubblici, stabilendo così differenti gradi di accessibilità dall'esterno. Tuttavia, mentre per le funzioni membro i diversi tipi di visibilità sono utili e vantaggiosi, per i membri dato possono rivelarsi dannosi: in effetti, dati *non privati* violano il principio di incapsulazione, uno dei cardini del paradigma object oriented. Da notare che l'uso di *protected* sposta il problema a livello delle classi derivate, ma non lo elimina completamente.

Violare l'incapsulazione dichiarando dati pubblici o protetti può facilmente portare a problemi di manutenzione: durante l'evoluzione della classe (ed ogni classe realmente utile subirà un processo di evoluzione) è normale trovarsi a riconsiderare alcune scelte implementative, e a dover quindi modificare alcune strutture dati (passando, ad esempio, da una semplice lista ad un albero binario o ad una hash table). Se tali dati sono pubblici o protected, ogni modifica rischia di compromettere le classi derivate o che in qualche modo accedono direttamente ai membri dato. Riprendendo l'esempio dell'evoluzione di una lista in un albero binario, va osservato che spesso l'*interfaccia* di una classe può essere mantenuta inalterata, anche a fronte di modifiche implementative: l'interfaccia derivante da un buon design è largamente insensibile all'implementazione. È l'esposizione diretta della struttura dati che può invece avere effetti deleteri.

Il problema dei dati pubblici e protetti è stato dapprima sollevato da Barbara Liskov in [Lis87], ed in seguito riconosciuto dallo stesso Stroustrup in [Str94], dove peraltro l'autore ricorda che Mark Linton, designer della famosa (e corposa) libreria Interviews, dopo anni di esperienza negativa in manutenzione ha eliminato ogni dato pubblico o protetto dalla libreria stessa.

L'alternativa immediata, spesso seguita dai principianti della programmazione object oriented, è allora di fornire un intero insieme di metodi di *get/set* per ogni membro dato. Tale soluzione ha un piccolo vantaggio, ovvero i nomi dei dati membro possono essere variati senza problemi, ed in alcuni casi anche i loro tipi (se sono compatibili); tuttavia ha in gran parte gli stessi difetti dell'esposizione diretta dei dati membro; è in ogni caso preferibile alla dichiarazione di dati *protected*, in quanto interpone comunque un ulteriore livello di disaccoppiamento tra i dati della classe base e l'implementazione della classe derivata.

La vera alternativa è di *pensare* in termini object oriented: fate svolgere all'oggetto le funzioni che gli sono proprie, senza richiedere ad altri l'accesso ai suoi dati. Vi sono rari casi in cui è necessario avere dei metodi di *get/set* pubblici per alcuni membri dato, ma in genere è solo indice di un cattivo design della classe.

Come detto poc'anzi (ma vale la pena di ripeterlo) metodi di *get/set* protetti sono invece utili al fine di evitare membri dato protetti, garantendosi comunque il piccolo vantaggio visto sopra; in questo caso, può essere preferibile dichiarare tali metodi come *inline*, evitando ogni overhead nel codice generato. Sarebbe utile soffermarsi comunque sul significato di tali get/set, e fornire delle astrazioni logiche dell'azione compiuta. Ad esempio, se una classe esporta dei metodi protetti per modificare dei membri ove sono memorizzate le coordinate di un oggetto, i metodi di get/set non dovrebbero necessariamente far riferimento ai nomi dei membri usati per contenere le coordinate, ma indicare semplicemente l'azione compiuta; in tal modo, si potrebbero in seguito fornire metodi overloaded per modificare le coordinate secondo diversi sistemi, ad es. coordinate polari o cartesiane, con le opportune conversioni gestite dai metodi di accesso. Se si fa riferimento al layout fisico della classe nei metodi di get/set, si rinuncia totalmente all'incapsulazione dei dati.

Raccomandazione 73

Non definire dati pubblici o protetti: tutti i dati devono essere privati.

Raccomandazione 74

Non definire metodi di get/set pubblici, e limitarsi il più possibile anche nel caso di metodi protected: cercate di identificare più chiaramente le responsabilità ed i compiti della classe, anziché esporne i dati. Se definite metodi di get/set protected, usate astrazioni a livello logico.

In questo senso, l'uso di *struct* dovrebbe essere limitato ai casi di interfacciamento con software scritto in C; diversi programmatori, con i quali ho avuto modo di discutere questo punto, hanno sostenuto con una certa veemenza l'utilità delle *struct* per memorizzare "dati senza operazioni associate". In effetti, osservandone con attenzione l'utilizzo abbiamo sempre notato che su tali dati venivano eseguite (come ovvio) delle operazioni, che tuttavia non erano state incapsulate, ma erano invece gestite esternamente. Anche se può sembrare una affermazione troppo

impegnativa, l'uso di *struct* è troppo spesso il rifugio di chi non vuole abbandonare le abitudini di programmazione nel paradigma imperativo (tipicamente, chi ha una lunga esperienza in C); se utilizzate delle *struct* nel vostro codice, provate a spendere un po' di tempo analizzandone gli schemi di utilizzo, e vedrete che in molti casi potrete incapsulare le operazioni che state eseguendo sui dati contenuti nelle *struct* stesse, o suddividere un oggetto monolitico, ma dal contenuto eterogeneo, in classi più piccole e coesive.

Raccomandazione 75

Limitare l'uso di *struct* all'interfacciamento con codice C; utilizzare le classi in tutti gli altri casi.

Valori di ritorno

L'accesso a membri dato delle classi non è solo negativo dal punto di vista della manutenzione del codice: può anche essere causa di problemi piuttosto difficili da identificare durante lo sviluppo, nonché di una fragilità nell'interfaccia fornita da una classe, tale da renderne l'uso stesso poco sicuro. Vedremo di seguito diverse situazioni in cui emergono tali problemi, e come possano invece essere risolti con una codifica più attenta.

In un progetto al quale ho lavorato tempo fa, un programmatore aveva implementato una classe secondo lo stile del **Listato 81**:

Listato 81

```
class Period
  {
  public :
    long& StartTime() { return( startTime ) ; }
    long& EndTime()  {  return( endTime ) ; }
    // ...

  private :
    long startTime ;
    long endTime ;
    // ...
  } ;
```

Anziché avere un insieme di metodi di Get/Set, vi era un solo metodo per ogni membro dato, che restituiva un reference al membro stesso; in tal modo, l'espressione *p.StartTime()*, ad esempio, poteva essere usata sia a sinistra che a destra di una assegnazione. Da un punto di vista pragmatico, la soluzione è persino migliore di una serie di metodi di Get/Set: almeno, ci si risparmia di doverli implementare entrambi (è invece peggiore se il membro è di tipo puntatore, ma poco importa ai fini della nostra discussione). Dovrebbe invece risultare ovvio che, in sostanza, il programmatore potrebbe aver direttamente dichiarato i membri come public, ottenendo sostanzialmente lo stesso grado di incapsulazione, cioé pressoché nullo.
Come abbiamo visto al paragrafo precedente, gli oggetti dovrebbero essere tali, e quindi in primo luogo dovrebbero incapsulare le funzionalità loro demandate.
Tra l'altro, nel caso in questione sia *startTime* che *endTime* avrebbero dovuto essere sempre positivi: tralasciando il fatto che sarebbe stato più opportuno dichiararli come unsigned long, è evidente che una buona interfaccia per la classe *Period* dovrebbe preoccuparsi di mantenere gli oggetti in stati consistenti. Fornire un puntatore o un reference ad un membro permette invece al chiamante di modificare a piacimento il valore del membro stesso.

Ovviamente, talvolta può essere necessario restituire un puntatore od un reference ad un campo, specialmente se restituirlo per valore comporta una sensibile penalizzazione delle prestazioni.
Tuttavia, in tal caso il puntatore od il reference dovrebbero essere di tipo *const*, in modo che il chiamante non possa farne uso per modificare lo stato interno dell'oggetto, o deallocare la zona di memoria puntata dal puntatore ottenuto.

Raccomandazione 76

Funzioni od operatori pubblici che restituiscano puntatori o reference a campi di una classe devono restituirli come puntatori/reference const.

Purtroppo la raccomandazione precedente non è sufficiente a proteggerci dagli errori: consideriamo come esempio una classe String, implementata in modo abbastanza semplice, sostanzialmente come un wrapper intorno ad un puntatore a carattere. Una possibile implementazione (incompleta) è data nel **Listato 82**:

Listato 82

```
class String
  {
  public:
    String( const char* s ) ;
    ~String() ;
    operator char*() { return( buffer ) ; }
  private :
    char* buffer ;
  } ;

String :: String( const char* s )
  {
  buffer = new char[ strlen( s ) + 1 ] ;
  strcpy( buffer, s ) ;
  }

String :: ~String()
  {
  delete[] buffer ;
  }
```

Chi ha implementato la classe String ha ceduto ad una facile tentazione: renderla totalmente compatibile con i puntatori a carattere, usati abitualmente in C, e spesso in C++, come implementazione delle stringhe. Il costruttore è un vero e proprio operatore di conversione da puntatore a carattere a String, ed esiste un operatore di conversione da String a puntatore a carattere. Notiamo che non viene restituito un puntatore ad un campo, ma un campo di tipo puntatore: non stiamo quindi violando la precedente raccomandazione.

Consideriamo ora un primo esempio di abuso della classe: **Listato 83**.

Listato 83

```
int main()
  {
  String s( “abcd” ) ;
  *( s + 2 ) = 'a' ;

  return( 0 ) ;
  }
```

Il problema è abbastanza evidente: poiché viene restituito un membro di tipo puntatore, possiamo liberamente manipolare i dati privati della classe attraverso il puntatore stesso; in fondo, restituire il right value di un membro di tipo puntatore non è diverso dal restituire un left value per un membro non puntatore, schema che abbiamo già identificato come pericoloso. In generale, non si dovrebbe mai restituire un membro di tipo puntatore o di tipo reference se non come puntatore/reference *const*. Anche in tal caso, come vedremo, occorre comunque una certa cautela.

Raccomandazione 77

Funzioni o operatori pubblici non devono restituire il valore di un membro di tipo puntatore o reference, se non sotto forma di puntatore const o reference const.

Nel **Listato 84** possiamo vedere una versione "riveduta e corretta" della classe String; con la nuova implementazione, il codice del **Listato 83** genererebbe un errore di compilazione.

Listato 84

```
class String
  {
  public:
    String( const char* s ) ;
    ~String() ;
    operator const char*() { return( buffer ) ; }
      // NB: const
  private :
    char* buffer ;
  } ;

String :: String( const char* s )
  {
  buffer = new char[ strlen( s ) + 1 ] ;
  strcpy( buffer, s ) ;
  }

String :: ~String()
  {
  delete[] buffer ;
  }
```

Si tratta di una cautela sufficiente? Purtroppo no, come mostra il **Listato 85**:

Listato 85

```
int main()
  {
  const char* p = NULL ;
  if( p == NULL )
    {
    String s( “abc” ) ;
    p = s ;
    }
  cout << p ;// possibile errore run-time

  return( 0 ) ;
  }
```

Il problema è abbastanza semplice da identificare, data la brevità del listato (e la conoscenza a priori dell'esistenza di un problema!): l'oggetto *s* viene distrutto all'uscita dal blocco if, ma il puntatore *p* conserva il suo valore; purtroppo, l'area di memoria puntata da *p* è già stata deallocata quando tentiamo di mandarla sullo stream di output. Un problema simile, connesso alla creazione di oggetti temporanei, verrà analizzato nel capitolo 9, al paragrafo “Oggetti temporanei”.

Non esiste, in generale, una strategia perfetta per gestire il problema del **Listato 85**; l'unica soluzione totalmente sicura sarebbe di non restituire mai puntatori a membri, o membri puntatori, neanche sotto forma const. Questa soluzione non è sempre percorribile: una classe String, ad esempio, *deve* fornire un metodo di conversione ad un puntatore a carattere; troppe funzioni di libreria, e troppe librerie, fanno uso di stringhe C-style, e una simile mancanza porterebbe ad abbandonare del tutto la classe String, piuttosto che le altre, indispensabili librerie. Dovremmo però sforzarci di rendere gli errori simili a quello del **Listato 85** più difficili da compiere: in tal senso, ad esempio, l'idea di un operatore di conversione da String a char* è totalmente errata. Infatti, se da una parte consente una più agevole scrittura del codice, dall'altra la rende *troppo* agevole: non occorre fermarsi a pensare se si sta richiedendo una funzione pericolosa, non occorre neppure *richiedere* la conversione “a rischio”: il compilatore la aggiungerà per noi.
Una soluzione di gran lunga migliore è di fornire una funzione di conversione, con un nome abbastanza significativo da ricordare al

programmatore cosa sta richiedendo: ad esempio, un nome come *ConvertToCStyleString*. Ulteriore cautela, che potrebbe evitare errori run-time, potrebbe essere di restituire un puntatore ad un membro statico della classe, dove la funzione in oggetto copia di volta in volta il buffer dei diversi oggetti; in tal modo, non si avrebbero più problemi di lifetime per gli oggetti, ma si correrebbe il rischio di non accorgersi di un difetto intrinseco del codice. Escluse rare condizioni particolari, è in genere meglio se il codice errato genera un errore, non se l'errore viene nascosto.

Raccomandazione 78

Non definite operatori di conversione pubblici che restituiscano puntatori/reference a membri o membri puntatore/reference: utilizzate invece funzioni membro dal nome esplicativo, che ricordino al programmatore la sua responsabilità nell'utilizzo del puntatore/reference ottenuto, che deve comunque essere const.

Funzioni Virtuali

Le funzioni virtuali sono una delle caratteristiche fondamentali del C++, alla base del meccanismo di binding dinamico. Il "lettore tipo" di questo libro avrà certamente usato le funzioni virtuali in numerose occasioni, tuttavia vale la pena di ricordare che esistono sostanzialmente due modelli di utilizzo per le funzioni virtuali:

1. La funzione virtuale viene definita come parte dell'interfaccia della classe, ed implementata dalla classe stessa; eventuali classi derivate possono ridefinire la funzione. La funzione *non* è utilizzata all'interno della classe stessa: solo il codice che utilizza la classe chiama tale funzione.

2. A differenza del punto 1, la funzione viene anche chiamata dal codice della classe stessa. Ciò significa che ridefinendo la funzione, una classe derivata può alterare il comportamento *anche* di altre funzioni della classe.

Un esempio dei due diversi schemi di utilizzo per le funzioni virtuali è dato nel **Listato 86**, dove *Print()* ricalca il modello (1), ovvero è implementata dalla classe ma non utilizzata dalla classe stessa, mentre ad esempio *PrintPage()* ricalca il modello (2), essendo chiamata da *Print()*.

Naturalmente, il fatto che *PrintPage()* sia protected e non public è solo una caratteristica dell'esempio dato, e non ha rilevanza generale.

Listato 86

```
class Document
  {
  public :
    virtual void Print() ;
  protected :
    virtual void PrintDocumentHeader() ;
    virtual void PrintPage() ;
    virtual void PrintPageHeader() ;

    unsigned GetCurrentPage() ;
  private :
    Page pages[ MAX_PAGES ] ;
    unsigned currentPage ;
    unsigned numOfPages ;
  } ;


void Document :: Print()
  {
  PrintDocumentHeader() ;
  for( currentPage = 0; currentPage < numOfPages;
       currentPage++ )
    {
    PrintPageHeader() ;
    PrintPage() ;  // entrambe si basano su currentPage
    }
  }


void Document :: PrintPage()
  {
  pages[ currentPage ].Print() ;
  }


// ...
```

Vi è una sottile differenza tra i due modelli, che stranamente non è nota anche a molti dei programmatori C++ più esperti: le funzioni virtuali del tipo (1) non creano praticamente mai problemi quando sono ridefinite in classi derivate, mentre le funzioni virtuali del tipo (2) sono la *vera* fonte del

problema noto come "fragile base class", che viene spesso attribuito (erroneamente) al meccanismo di ereditarietà tout-court.

Per chi non conoscesse il suddetto problema, vale la pena di illustrarlo brevemente: una delle caratteristiche dei linguaggi object oriented, basati sull'ereditarietà, dovrebbe essere il notevole riutilizzo del codice. Apparentemente, questo avviene in misura di gran lunga inferiore alle aspettative: per quanto alla base del fenomeno possano esservi fattori di tipo psicologico o di immaturità del mercato, è stato nondimeno riscontrato il comune bisogno di accedere al sorgente di una classe base per poterla efficacemente riutilizzare tramite ereditarietà.
Ciò comporta purtroppo un accoppiamento troppo elevato tra la classe base e la classe derivata, peggiorato dal fatto che si tratta di un accoppiamento concettuale, non visibile nel codice stesso; come vedremo tra breve, si tratta in realtà di un accoppiamento sul *contesto di chiamata*.
Le conseguenze di tale accoppiamento sono una eccessiva dipendenza delle classi derivate dall'implementazione -si badi bene, non dall'interfaccia- della classe base, cosicché cambiando l'implementazione della classe base si possono creare malfunzionamenti nelle classi derivate, anche se la modifica non ha variato l'interfaccia della classe base. Tale problema viene normalmente contrassegnato come intrinseco nel meccanismo di ereditarietà, e viene indicato come il problema della "fragile base class"[15].

Possiamo vedere un esempio del problema basandoci sul **Listato 86**; supponiamo di voler aggiungere il numero di pagina durante la stampa del nostro documento: tutto ciò che dobbiamo fare è derivare una nuova classe da Document, e ridefinire la funzione *PrintPageHeader()* in modo che stampi il numero di pagina, e poi richiami la funzione originale di Document. Quale numero di pagina deve stampare? Dobbiamo leggere il codice della classe Document per saperlo (o avere a disposizione una documentazione tecnica *molto* dettagliata della classe Document, spesso non disponibile, o perlomeno non al livello di precisione desiderato); in questo caso, il numero di pagina da stampare si trova nella variabile currentPage, e possiamo ottenerlo chiamando GetCurrentPage(). In realtà dobbiamo sommare 1 a tale valore, altrimenti il nostro documento comincerà a pagina zero. La nostra classe derivata è ora completa e funzionante.

[15]Altri meccanismi di riuso del codice, come la *delegation* o l'*aggregazione* (come definita nel Component Object Model) sono meno sensibili al problema, ma richiedono una maggiore accuratezza ed attenzione in fase di design dei componenti software.

In seguito, il responsabile della classe Document decide di modificarne l'implementazione: ad esempio, come mostrato nel **Listato 87**.

Listato 87

```
void Document :: Print()
  {
  PrintDocumentHeader() ;
  for( currentPage = 1;
       currentPage < numOfPages + 1;
       currentPage++ )
    {
    PrintPageHeader() ;
    PrintPage() ;   // entrambe si basano su pageNum
    }
  }


void Document :: PrintPage()
  {
  pages[ currentPage - 1 ].Print() ;
  }


// ...
```

La modifica è stata di poco conto: currentPage riflette ora il numero "logico" della pagina, anziché il suo indice "fisico" nell'array delle pagine. Tuttavia la nostra classe derivata non funzionerà più correttamente (un errore molto innocente, in questo piccolo esempio, ma potrebbe essere molto più grave in un caso reale). Ecco spuntare il problema della "fragile base class".

Si può dire molto a proposito del problema stesso; innanzitutto, la modifica può o meno aver violato l'interfaccia della classe: dipende da come era stata definita la semantica di *GetCurrentPage()*. Se era stata definita semplicemente come "restituisce il valore di *currentPage*", l'interfaccia della classe base non è stata violata. Resta il fatto che la modifica ha introdotto un errore nella classe derivata.

Esempi simili sono spesso usati dai detrattori dell'object oriented programming, o come spunto per proporre soluzioni alternative all'ereditarietà; il punto interessante è che, tra i molti articoli che trattano il

problema, nessuno sembra essersi concentrato sulle reali cause del problema stesso.

Non dovrebbe sorprendere, a questo punto, l'affermazione che tutto il problema sta nel fatto che *PrintPage()* è una funzione virtuale di tipo (2), ovvero è richiamata anche all'interno della classe che la definisce. Chi non fosse convinto, può cercare di ottenere un malfunzionamento simile ridefinendo la funzione di tipo (1) *Print()*.

Come avevo accennato, la vera ragione del problema è la *dipendenza dal contesto di chiamata*: per ridefinire una funzione utilizzata all'interno di una classe, dobbiamo capire il contesto in cui si aspetta di essere chiamata, in modo da gestirlo opportunamente nella versione ridefinita. Ciò comporta innanzitutto la necessità di accedere al sorgente della classe base, in quanto il contesto di chiamata non fa parte dell'interfaccia e non è normalmente specificato nella documentazione (grave, ma comune errore!).
Peggio ancora, ciò implica un accoppiamento implicito con l'implementazione della classe base, quindi un "contratto" più forte della semplice interfaccia; contratto che può facilmente, ed inavvertitamente, essere violato durante la manutenzione della classe base.

Come possiamo evitare il problema? Certamente cambiare paradigma di programmazione, come molti suggeriscono, è una soluzione; forse, però, è troppo drastica per essere intrapresa.
In realtà, la conoscenza del problema ci indica due buone strade per minimizzare le occasioni nel quale si presenta: quando progettiamo una classe, cerchiamo di evitare le funzioni virtuali di tipo (2); in molti casi, ciò è possibile senza grandi sforzi, perché è abbastanza comune dichiarare funzioni come *virtual* "giusto nel caso" debbano essere ridefinite. Quando non è possibile, è opportuno fornire una documentazione dettagliata sui diversi contesti in cui la funzione viene richiamata all'interno della classe stessa: ciò ridurrà al minimo la necessità di leggere il sorgente della classe per ereditare, e nel contempo costituirà un *impegno aggiuntivo*, ovvero una serie di vincoli che ci impegniamo a rispettare nella classe base. Nel caso precedente, ad esempio, avremmo dovuto specificare il vincolo che, al momento della chiamata di *PrintPage()*, la variabile *currentPage* contiene il numero logico della pagina meno 1. In tal senso, la modifica proposta alla classe base avrebbe violato il vincolo. Alternativamente, se tale vincolo non era specificato, o meglio ancora se era stato esplicitamente indicato come soggetto a variare in future implementazioni, allora era l'implementazione della classe derivata ad essere fragile, non la classe base.

Per completezza, come avremmo potuto implementare la classe derivata, se avessimo saputo di non poter far riferimento alla variabile *currentPage*, in quanto dipendente dall'implementazione? Vi sono diverse tecniche, e nel **Listato 88** riporto quella che ritengo migliore: come potete vedere, non vi è alcuna dipendenza dalla variabile *currentPage*, né replicazione di codice.

Listato 88

```
class DocWithPageNum : public Document
  {
  public :
    virtual void Print() ;
  protected :
    virtual void PrintPageHeader() ;
  private :
    unsigned pageNumber ;
  } ;


void DocWithPageNum :: Print()
  {
  pageNumber = 1 ;
  Document :: Print() ;
  }


void DocWithPageNum :: PrintPageHeader()
  {
  // stampa numero pagina: pageNumber

  pageNumber++ ;

  Document :: PrintPageHeader() ;
  }
```

In conclusione, il problema della "fragile base class" può essere gestito senza rinunciare all'ereditarietà: è sufficiente una maggiore cautela nel progetto delle classi, e dobbiamo fornire le adeguate informazioni a chi riutilizza il nostro codice.

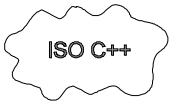
Raccomandazione 79

In ogni classe, limitate il numero di chiamate a funzioni virtuali dichiarate nella classe stessa.

Raccomandazione 80

Per ogni funzione virtuale dichiarata in una classe e richiamata nel codice della classe stessa, documentate adeguatamente nel file header della classe ogni contesto di chiamata. Specificate chiaramente quali vincoli sul contesto verranno rispettati in future versioni della classe (e rispettateli!) e quali possono essere variati senza preavviso.

Rilassamento sul tipo del risultato



Inizialmente il C++ permetteva di ridefinire le funzioni virtuali *solo* se sia i tipi degli argomenti che del risultato erano identici nella funzione derivata; nella proposta di standard, la regola è stata "rilassata" sul valore di ritorno. Se nella classe base una funzione ritorna un puntatore o un reference alla classe B, in una classe derivata possiamo ridefinire la funzione virtuale mantenendo gli stessi tipi per i parametri, ma dichiarando per il risultato una qualunque classe D derivata da B, posto che B sia accessibile da D (quindi D non può ad esempio essere derivata come *private*). Ciò consente in effetti di evitare alcuni type-cast nel codice, mantenendo comunque l'integrità del type-system. Il **Listato 89** illustra un esempio di uso, e nuovamente può essere utilizzato per verificare l'aderenza del vostro compilatore alla direttiva; attualmente, pressoché ogni compilatore di buon livello dovrebbe compilare il codice ed il programma generato dovrebbe produrre come output "derived ".

Listato 89

```
#include <iostream.h>


class Base
  {
  public :
    virtual Base* This() ;
  } ;

Base* Base :: This()
  {
  cout << "base " ;
  return( this ) ;
  }

class Derived : public Base
  {
  public :
    virtual Derived* This() ;
  } ;

Derived* Derived :: This()
  {
  cout << "derived " ;
  return( this ) ;
  }


int main()
  {
  Derived d ;
  Base* b = &d ;
  b->This() ;

  return( 0 ) ;
  }
```

“Super” o “Inherited”

È molto comune, nel body delle funzioni virtuali ridefinite nelle classi derivate, eseguire una chiamata alla stessa funzione nella classe base: ciò corrisponde, di norma, ad implementazioni delle funzioni derivate che

preservano la semantica della funzione, ed eseguono alcuni compiti addizionali. Un esempio di chiamata alla funzione della classe base è presente nella definizione di *DocWithPageNum* :: *Print*(), nel **Listato 88**. Il meccanismo per chiamare una funzione della classe base all'interno della classe derivata è molto semplice, e si basa sull'operatore di qualificazione :: (tale operatore esegue un *early binding* della chiamata, contrapposto al *late binding* che si avrebbe se non si qualificasse la classe di appartenenza per la funzione chiamata).

Apparentemente non vi è nulla di errato nell'uso dell'operatore di qualificazione; in realtà esiste un problema di manutenzione connesso al suo utilizzo, problema sul quale non è stato in effetti posto l'accento con sufficiente intensità nella letteratura sul C++.

Supponiamo infatti di voler inserire una nuova classe in una gerarchia esistente, e di doverla inserire non al termine della gerarchia, ma in una posizione intermedia. Non si tratta di un caso accademico, in quanto è piuttosto frequente, durante l'evoluzione di una libreria, sentire l'esigenza di astrarre alcuni dettagli di una o più classi in una classe di più alto livello, ma pur sempre derivata. Nell'esempio del **Listato 88**, potremmo voler inserire una classe *CustomDocument* tra la classe base *Document* e la classe derivata *DocWithPageNum.*

In tal caso, probabilmente la qualificazione esplicita all'interno della funzione *Print()* dovrebbe essere modificata, per riferirci sempre alla classe genitore e non alla classe radice, quindi a *CustomDocument* e non a *Document*. In altri linguaggi, come Smalltalk, esiste il meta-identificatore *Super* che indica sempre la superclasse immediatamente superiore; in C++, in gran parte a causa dell'ereditarietà multipla, non esiste un supporto equivalente, almeno non a livello del kernel del linguaggio. È però possibile simularlo, limitatamente ai casi di ereditarietà singola, seguendo una tecnica ideata da Michael Tienmann; poiché nella terminologia del C++ non si usano i termini "superclasse" e "sottoclasse", ritenuti poco chiari da Stroustrup, useremo l'identificatore *inherited* come nella formulazione originale.

Un esempio su una gerarchia di classi è visibile nel **Listato 90**:

Listato 90

```
class Base
  {
  public :
```

```
    virtual void Print() ;
  } ;



class Derived : public Base
  {
  public :
     virtual void Print() ;
  private :
    typedef Base inherited ;
  } ;

void Derived :: Print()
  {
  // ... fa qualcosa

  inherited :: Print() ;   // chiama Base :: Print()
  }



class DoubleDerived : public Derived
  {
  public :
     virtual void Print() ;
  private :
    typedef Derived inherited ;
  } ;

void DoubleDerived :: Print()
  {
  // ... fa qualcosa

  inherited :: Print() ;   // chiama Derived :: Print()
  }
```

L'idea di fondo è semplice: all'interno della dichiarazione di ogni classe derivata, definiamo il tipo *inherited* uguale alla classe genitore, ed usiamo l'identificatore *inherited* all'interno delle funzioni, ogni volta che dobbiamo qualificare esplicitamente la classe genitore. Se aggiungiamo una classe intermedia nella gerarchia, dobbiamo solo preoccuparci di cambiare la definizione di *inherited* all'interno della dichiarazione della classe: il resto è automatico. Infatti, come potete vedere, il body di *Derived :: Print()* e di *DoubleDerived :: Print()* è assolutamente identico, anche se in effetti eseguono un early binding a classi differenti.

Le regole di scope garantiscono che all'interno di ogni classe, *inherited* si riferisca sempre alla classe "giusta", ovvero alla classe genitore.

Osserviamo che tale tecnica può evitare errori grossolani durante il *merge* di più classi in una sola classe derivata: se due o più classi genitore definiscono un metodo, ad es. Print(), con due significati diversi, normalmente una classe derivata da entrambe dovrà ridefinire tale metodo in modo opportuno. Se tuttavia il merge avviene in fase di manutenzione, la classe derivata avrà già un early binding con una delle classi genitore, ed è possibile che non ci si accorga del problema. La presenza di un "typedef genitore inherited" nella dichiarazione della classe derivata dovrebbe invece farci riflettere nel momento in cui la classe acquisisce un nuovo genitore: la tecnica in questione diventa infatti inadatta ed è necessario ritornare nuovamente alla qualificazione esplicita della classe.

Raccomandazione 81

Nelle gerarchie di derivazione con ereditarietà singola, l'uso di *inherited ::* per qualificare la classe genitore può semplificare significativamente la manutenzione della gerarchia stessa.

Infine, osserviamo che, sia che si usi il nome della classe genitore, sia che si usi la tecnica su esposta, sarebbe opportuno che le qualificazioni esplicite con :: fossero limitate al codice della classe base o della classe derivata. Non dovrebbero essere presenti nel codice che utilizza le classi in questione, come nel **Listato 91** (che assume le dichiarazioni di classe del **Listato 88**).

Listato 91

```
int main()
  {
  DocWithPageNum dwpn ;

  dpwn.Document::Print() ; // Sconsigliato!
  }
```

Codice simile è ancor più difficile da mantenere, se cambia la gerarchia delle classi, proprio perché può essere localizzato in moduli che nulla hanno a che fare con la gerarchia che stiamo modificando: mentre può essere ragionevole, nel caso non si usi *inherited ::*, rivedere il codice delle classi derivate, è di norma improponibile la revisione di tutto il codice che utilizza le classi quando la gerarchia viene modificata. In effetti, è proprio

ciò che l'approccio object oriented vorrebbe evitare; alcuni costrutti del C++ vanno usati con cautela, proprio per non vanificare i vantaggi dell'approccio ad oggetti.

Raccomandazione 82

L'uso della qualificazione esplicita con :: dovrebbe essere limitato all'interno delle classi base (per ottenere l'early binding con il metodo della classe stessa) o nelle classi derivate (per richiamare i metodi delle classi base). Non dovrebbe essere usato al di fuori di esse, dove oggetti delle classi vengono usati.

Funzioni membro "const"

Una funzione membro viene definita *const* dichiarandola come nel **Listato 92**:

Listato 92

```
class Person
  {
  public :
    int GetAge() const ;
  // ...
  } ;
```

Una funzione membro *const* non altera lo stato dell'oggetto sul quale opera: spesso le funzioni *const* vengono chiamate *inspectors*, in quanto "ispezionano" -ma non modificano- l'oggetto stesso. Ogni tentativo di modificare l'oggetto all'interno della funzione const (es. **Listato 93**) viene indicato dal compilatore come un errore; torneremo su questo punto più avanti.

Listato 93

```
class Person
  {
  public :
    int GetAge() const
      { x = 0 ; return( 10 ) ; } // Errore !!!
  private :
    int x ;
  // ...
```

```
  } ;
```

È opportuno definire le funzioni come *const* ogni volta che ciò sia possibile, in quanto in tal modo forniamo al compilatore ulteriori informazioni, che possono essere utilizzate per ottimizzare il codice (vedi oltre), ma soprattutto perché dette informazioni rappresentano un utile mezzo per *comunicare,* a chi legge il codice, che la funzione in esame non opera alcuna modifica sullo stato dell'oggetto. Le funzioni *const*, inoltre, sono le uniche a poter essere chiamate su un oggetto costante: ulteriore ragione per dichiarare *const* le funzioni, quando è possibile. A tal proposito, ricordate che anche gli oggetti *const* sono modificabili all'interno della lifetime del costruttore, altrimenti non potrebbero essere inizializzati.

Infine, in C++ è possibile utilizzare l'overloading per fornire due versioni delle funzioni, una operante su oggetti *const* ed una che opera su oggetti "mutabili" (vedremo più oltre l'origine del termine), come nel **Listato 94**, dove un operatore di accesso agli elementi di un array è definito per restituire un const reference se applicato ad oggetti const, od un reference se applicato ad altri oggetti.

Listato 94

```
class Array
  {
  public :
    const int& operator[]( int index ) const ;
    int& operator[]( int index ) ;
  // ...
  } ;
```

Nella mia esperienza, ho osservato che molti programmatori tendono a non usare il *const*, o perlomeno a postporre la definizione delle funzioni *const* ad una successiva fase di "ripulitura" del codice.
Questa è una abitudine che si dovrebbe assolutamente evitare: non solo la definizione di una funzione come *const* o meno andrebbe presa al momento stesso della definizione dell'interfaccia per la classe, e non rimandata ad una manutenzione che spesso si finisce per non eseguire, ma aggiungere la keyword *const* in tempi successivi può richiedere una serie di modifiche al codice, ad esempio perché all'interno di una funzione const cerchiamo di chiamarne una non-const.
Problemi simili sono semplici da gestire all'inizio, ma modificare il codice in tal modo alla fine, è una delle tecniche migliori per introdurre bug e

perdere tempo, anche perché sarebbe buona norma rieseguire i test di unità e di integrazione dopo la modifica.

Raccomandazione 83

Funzioni che non modificano lo stato dell'oggetto vanno dichiarate *const*; ciò andrebbe fatto sin dalla fase di definizione dell'interfaccia della classe.

Osserviamo che la definizione di funzione *const* data dal C++ è piuttosto debole: è ad esempio possibile modificare parti dell'oggetto che siano accessibili tramite puntatori (vedere **Listato 95**) in quanto l'oggetto stesso non viene modificato dall'operazione (il puntatore resta immutato).

Listato 95

```
class Person
  {
  public :
    Person() { *x = 0 ; }
    int GetAge() const { return( (*x)++ ) ; }  // OK!!!
  private :
    int* x ;
  // ...
  } ;
```

Osserviamo che nel **Listato 95** *GetAge*() è legalmente dichiarabile come const, anche se due chiamate successive ad essa, senza altro codice intermedio, genereranno comunque valori diversi. Ciò potrebbe anche ingannare un compilatore ottimizzante troppo spinto, che tentasse di estrarre la chiamata da un ciclo *for*, o anche dalla condizione di un ciclo *for*, come nel **Listato 96**:

Listato 96

```
#include <iostream.h>

class Person
  {
  public :
    Person() { *x = 0 ; }
    int GetAge() const { return( ++(*x)  ) ; }  // OK!!!
  private :
    int* x ;
  // ...
```

```
  } ;

int main()
  {
  Person p ;
  // Il loop viene eseguito una sola volta se il
  // compilatore ottimizza (erroneamente) la
  // chiamata a GetAge(), altrimenti è un loop
  // infinito
  for( int i = 0; i < p.GetAge(); i++ )
    cout << "*" ;

  return( 0 ) ;
  }
```

Esistono a tal proposito due approcci all'uso del *const*:

1. Un approccio che potremmo chiamare *pragmatico* o *debole* (con riferimento alla permissività del C++). Secondo tale visione, se una funzione const viene correttamente compilata, essa è corretta. Modificare oggetti globali, eseguire input/output su file o sul video, modificare parti dell'oggetto tramite puntatori è interpretato come lecito.

2. Un approccio che potremmo chiamare *concettuale* o *forte*. Secondo tale visione, la chiamata ad una funzione const non deve a tutti gli effetti modificare l'oggetto, inteso anche nelle sue componenti accessibili tramite puntatori, nonché oggetti globali, inclusi gli stream, o eseguire I/O su video.

La scelta fra le due visioni è un punto fondamentale per l'impostazione di un progetto, meglio ancora se la si propaga ad ogni progetto al quale si lavora. Il primo approccio consente una maggiore versatilità, e permette di definire come *const* funzioni che secondo l'approccio *forte* non lo sono. In molti casi la visione *forte* porta a codice più resiliente rispetto alle modifiche, nel senso che se in futuro un membro dato di tipo puntatore venisse modificato, divenendo un sotto-oggetto senza l'indirezione del puntatore, codice che rispetta la visione *forte* non richiederebbe ulteriore modifiche, mentre codice *debole* non potrebbe essere ricompilato senza eliminare il *const* da alcune funzioni o riscriverne alcune porzioni in modo sostanziale.
A maggior ragione, modificare oggetti globali o lo stato della macchina, inteso come coppia (input, output), all'interno di una funzione *const* è quantomeno sospetto e potrebbe portare a effetti collaterali poco prevedibili

se le dichiarazioni e l'uso di oggetti costanti fossero riorganizzati - un evento che dovrebbe esssere innocuo nell'approccio forte.

Raccomandazione 84

Se possibile, non modificate oggetti globali o sotto-oggetti accessibili tramite puntatori all'interno di funzioni membro const, e non eseguite input-output all'interno di esse. Se dovete farlo, commentate adeguatamente il codice, e magari anche la dichiarazione della funzione.

Il "problema della cache"

Un classico problema che viene associato alla discussione delle "funzioni const" è rappresentato dalle situazioni in cui una funzione non modifichi l'oggetto da un punto di vista concettuale, ma si trovi a modificarlo per ragioni implementative: ad esempio, per mantenere una cache degli ultimi valori estratti da un database (da cui il nome del problema). Se la cache è un sotto-oggetto, non possiamo modificarlo all'interno di una funzione *const*: osserviamo che in questo caso la funzione è const secondo l'approccio concettuale, ma non sotto il profilo pragmatico. Abbiamo ora diverse opzioni:

1. Non dichiarare la funzione come const. Soluzione drastica e sconsigliata.

2. Non dichiarare un membro di classe Cache, ma un membro di tipo puntatore a Cache: a questo punto, per quanto visto sopra, possiamo modificare la cache senza problemi, magari con un commento come da **Raccomandazione 84**. Per altre ragioni (vedere capitolo 14) tale soluzione non è comunque ottimale.

3. Definire la cache come *mutable*: vedere **Listato 97**. La keyword *mutable* indica che il dato si intende modificabile ("mutabile") anche in oggetti costanti, e quindi modificabile all'interno di funzioni const. Questa è la soluzione perfetta: non vi sono trucchi di sorta, ed anche in fase dichiarativa rendiamo esplicito l'uso che vogliamo fare dei dati.

4. Usare const_cast per ottenere una versione non-const di *this* all'interno di una funzione membro: vedere **Listato 98**. In assenza di *mutable* (quindi utilizzando compilatori molto datati) questa soluzione va preferita per la *località* della soluzione stessa: viene nascosto

all'interno dell'implementazione della funzione ogni dettaglio di come la cache venga effettivamente gestita. La maggior parte dei compilatori supporta ormai const_cast. Usare in qualche modo la parola *mutable* all'interno del codice (come nel **Listato 98**) può essere utile per trovare tutte le occorrenze del "trucco" in futuro, quando il vostro compilatore supporterà direttamente *mutable*.

5. Se il compilatore (sempre molto datato) non supporta const_cast, usare la soluzione 4 effettuando un cast esplicito di *this* in modo da eliminare il *const*. È la soluzione che si può più facilmente portare alla 4 non appena il vostro compilatore supporterà const_cast.

Listato 97

```
class Database
  {
  private :
    //...
    mutable Cache localCache ;
  } ;
```

Listato 98

```
class Database
  {
  public :
    Person GetPerson( String name ) const ;
  private :
    //...
   Cache localCache ;
  } ;

Person Database :: GetPerson( String name ) const
  {
  // ...

  Database* mutableThis = const_cast<Database*>(this) ;
  (mutableThis->localCache).SetRecent( something ) ;
  // modifica la cache
  }
```

Raccomandazione 85

Dovendo modificare membri dato all'interno di un oggetto const, ammesso che ciò sia concettualmente corretto, dichiarate i dati come *mutable*; in alternativa, usate *const_cast* all'interno della funzione membro che deve modificare i dati, od un cast esplicito se *const_cast* non è disponibile.

Const e le ottimizzazioni

Dichiarare una funzione membro come *const* dovrebbe permettere al compilatore di ottimizzare maggiormente il codice (o, nei rari casi in cui ciò è necessario, porre gli oggetti in ROM; ciò si applica solo ad applicazioni di tipo embedded, che necessitano di compilatori particolari). In effetti, ciò in genere non avviene [Str94], e vi sono ottime ragioni: da una parte, come abbiamo visto, la definizione di *const* è così debole che il compilatore non può assumere che lo stato dell'oggetto non sia realmente cambiato dalla chiamata ad una funzione membro const. Una ottimizzazione possibile, nel caso di oggetti piccoli che stiano in alcuni registri della CPU, consiste nel non eseguire una copia registri-memoria dopo la chiamata alle funzioni const. Purtroppo, esiste comunque il problema dell'alias dei puntatori: se un oggetto const è accessibile attraverso un puntatore ad oggetto non-const, non può comunque essere accettata nessuna assunzione, anche in una visione *forte* del const. In effetti una analisi completa degli alias è intrattabile per il compilatore (alcuni hanno comunque un'opzione, sotto la responsabilità dell'utente, del tipo "assume no pointer aliasing") e questo significa che le poche ottimizzazioni corrette sono possibili in un numero di casi così esiguo[16] da risultare praticamente inesistenti, e probabilmente molti compilatori non eseguiranno alcuna ottimizzazione sulle funzioni const. Il reale vantaggio delle funzioni membro const è la migliore comunicazione di intenti tra chi scrive il codice e chi lo legge - uno dei punti fondamentali della buona programmazione.

[16] l'oggetto dovrebbe essere costruito nello scope della chiamata alla funzione const, in modo da verificare l'assenza di alias, inoltre ogni occorrenza dell'identificatore dell'oggetto dal momento della costruzione alla chiamata della funzione const da ottimizzare dovrebbe essere sotto forma const e tipata staticamente.

Overloading degli operatori

Come molte potenzialità del C++, l'overloading degli operatori può essere molto utile o molto dannoso per la comprensione del codice, in funzione della maturità del programmatore. Chiunque abbia letto codice in cui all'operatore + era stata assegnata una semantica tutt'altro che naturale, ha ben presenti i problemi derivanti da un utilizzo troppo disinvolto dell'overloading.

D'altra parte, definire la somma, moltiplicazione, sottrazione tra numeri complessi o matrici ridefinendo gli operatori + * -, o ridefinire l'operatore [] per accedere agli elementi di una classe Array, comporta un effettivo information hiding e può portare ad un codice più chiaro che non l'utilizzo di funzioni Add, Multiply, e così via.

La regola d'oro nell'uso dell'operator overloading è quindi di definire gli operatori in modo *naturale*: ad esempio il + dovrà sempre e comunque corrispondere ad un concetto di "somma", per quanto generalizzato. Ovviamente vi è un ampio margine discrezionale, sul quale è difficile intervenire: ad esempio, discussioni senza fine sono state intraprese per decidere se sia o meno chiaro utilizzare il + come operatore di concatenazione in una classe String.
Spesso un buon metodo per arrivare ad una conclusione è considerare la "famiglia" cui l'operatore appartiene (nel caso del +, la famiglia include certamente il -, e potrebbe o meno includere il * ed il /); se non è possibile associare una semantica adeguata agli altri operatori della famiglia, forse il significato che cerchiamo di attribuire al singolo operatore non è così immediato come potrebbe sembrare.
Naturalmente, è importante che le relazioni tra gli operatori vengano mantenute: se ridefiniamo gli operatori +, += ed =, la nuova semantica di *a = a + b* dovrebbe essere identica a quella di *a += b*. Analogamente per gli altri operatori "ridondanti", come il ++, --, e così via.

Una tecnica per garantire l'equivalenza semantica, ed evitare possibili problemi in fase di manutenzione (leggi: modifiche non propagate a tutti gli operatori) è definire il significato di un solo operatore, quello accoppiato all'assegnazione, e definire gli altri in termini di esso. Un esempio è visibile nel **Listato 99**; il difetto della tecnica è che si rinuncia alle possibili ottimizzazioni che si avrebbero definendo i singoli operatori: come sempre, la scelta tra semplicità di manutenzione ed efficienza del codice deve essere eseguita sui singoli casi concreti.

Listato 99

```
class Complex
  {
  // ...

  public :
    Complex& operator+=( const Complex& a ) ;
    // ...

  friend Complex operator+( const Complex& a,
    const Complex& b ) ;
  } ;


Complex operator+( const Complex& a, const Complex& b )
  {
  Complex result = a ;
  result += b ;
  // in tal modo dobbiamo solo definire la somma in +=
  return( result ) ;
  }
```

Raccomandazione 86

Se un operatore viene ridefinito, la sua semantica dovrebbe risultare il più possibile naturale; gli operatori della stessa famiglia logica dovrebbero essere a loro volta ridefiniti in modo che vengano preservate le usuali equivalenze semantiche.

Operatori && ed ||

Nel linguaggio C++, gli operatori condizionali && ed || eseguono il cosiddetto *short-circuit* (corto-circuito): se l'operando di sinistra è sufficiente a determinare il risultato dell'operazione (ovvero, se è false nel caso di &&, o true nel caso di ||) allora l'operando di destra non viene neppure valutato. Il corto-circuito risulta comodo in molte situazioni, come nel **Listato 100**, che sarebbe errato se l'ordine di valutazione degli operatori di && non fosse definito dal linguaggio, come avviene ad esempio per gli equivalenti operatori AND ed OR in Pascal, ove l'ordine di valutazione, nonché la possibilità di effettuare o meno lo short-circuit, sono lasciati al compilatore.

Listato 100

```
if( source != NULL && *source == 'a' )
  *source = 'A' ;
```

Esiste tuttavia un problema: il corto-circuito su && ed || **non** è applicato quando gli operatori vengono ridefiniti tramite overloading: ciò significa che i nostri operatori si comporteranno in modo leggermente (e subdolamente) diverso dagli operatori predefiniti. Pensate agli effetti che ciò avrebbe su una classe Bool, se il nuovo comportamento di && ed || non fosse chiaramente documentato! Il diverso comportamento è facilmente giustificabile, se consideriamo che un operatore overloaded altro non è che una funzione con notazione prefissa, infissa o postfissa, mentre gli operatori && ed || di default sono trattati in modo particolare dal compilatore, che ha sufficiente conoscenza del contesto per generare il codice opportuno[17]. Tuttavia si tratta di una sottigliezza che può facilmente sfuggire a chi utilizza gli operatori overloaded.

Raccomandazione 87

Non ridefinite gli operatori || ed && se non è veramente necessario; se dovete farlo, aggiungete sempre una linea di commento nel file header, richiamando esplicitamente la mancanza di short-circuit nella valutazione.

Operatore di assegnazione

Le cautele nella definizione di un operatore di assegnazione sono in gran parte simili a quelle necessarie nella definizione di un costruttore di copia. La somiglianza non dovrebbe stupire, considerando la vicinanza concettuale delle due operazioni (che in effetti spesso confondono i principianti).
Quando si esegue una assegnazione tra oggetti, come nel **Listato 101**, viene utilizzato l'apposito operatore di assegnazione della classe. Se tale operatore non è dichiarato, il compilatore genera un opportuno operatore di copia member-wise.

[17]Nel linguaggio Ada il problema è stato risolto considerando gli equivalenti di && ed || (ovvero "if then" ed "or else") distinti dagli operatori, e quindi non suscettibili di overloading. In C++, i programmatori devono invece tenere presente la differenza.

Listato 101

```
class C
  {
  // ...
  }


int main()
  {
  C x ;
  C y ;

  y = x ;

  return( 0 ) ;
  }
```

Se una classe ha dei membri di tipo puntatore, dovrebbe sempre definire un operatore di assegnazione, viceversa si avranno dei malfunzionamenti esattamente come nel caso del costruttore di copia, con l'ulteriore problema in questo caso del mancato rilascio della memoria. Il **Listato 102** illustra il problema dell'aliasing e del memory leak:

Listato 102

```
#include <string.h>

class String
  {
  public:
    String( const char* s ) ;
    ~String() ;
  private :
    char* cstring ;
  } ;

String :: String( const char* s )
  {
  int len = strlen( s ) ;
  cstring = new char[ len + 1 ] ;
  strcpy( cstring, s ) ;
  }

String :: ~String()
  {
```

```
  delete[] cstring ;
  }

int main()
  {
  String s( “prova” ) ;
  String c( “a” ) ;

  c = s ;  // c :: cstring non viene rilasciata!

  // quando s è distrutta, s :: cstring è rilasciato
  // quando c è distrutta, c :: cstring è rilasciato,
  // ma è un alias per s :: cstring che e' gia' stato
  // rilasciato!

  return( 0 ) ;
  }
```

Poiché non esiste un costruttore di copia, il C++ esegue una copia bit-wise dei puntatori; da questo momento, ogni modifica a *s* modificherà anche *c*, e viceversa. Inoltre, quando *s* viene assegnata a *c*, non viene deallocata la zona di memoria puntata da *c::cstring*, che non potrà più essere referenziata. Peggio ancora, al momento della distruzione l’area puntata dal membro *cstring* di *s* e *c* verrà rilasciata due volte.

Una versione corretta del **Listato 102** è data nel **Listato 103**, che include anche un opportuno trattamento del costruttore di copia.

Listato 103

```
#include <string.h>

class String
  {
  public:
    String( const char* s ) ;
    String( const String& s ) ;
    ~String() ;
    const String& operator =( const String& s ) ;
  private :
    void Init( const char* s ) ;
    char* cstring ;
  } ;

String :: String( const char* s )
  {
```

```
  Init( s ) ;
  }

String :: String( const String& s )
  {
  Init( s.cstring ) ;
  }


String :: ~String()
  {
  delete[] cstring ;
  }


const String& operator =( const String& s )
  {
  if( this != &s )
    {
    delete[] cstring ;
    Init( s.cstring ) ;
    }

  return( *this ) ;
  }


void String :: Init( const char* s )
  {
  int len = strlen( s ) ;
  cstring = new char[ len + 1 ] ;
  strcpy( cstring, s ) ;
  }


int main()
  {
  String s( "prova" ) ;
  String c( "a" ) ;
  c = s ;    // viene rilasciata la vecchia memoria e
             // viene allocata una nuova area

  return( 0 ) ;
  }
```

Osserviamo che l'operatore di assegnazione verifica che non si stia assegnando un oggetto a se stesso; ciò è fondamentale ogni volta che

l'operatore esegue operazioni distruttive sull'operando di sinistra, onde evitare di referenziare in seguito oggetti distrutti.

Talvolta è necessario impedire che gli oggetti appartenenti ad una classe possano essere assegnati ad altri oggetti: in analogia al caso dei costruttori di copia, possiamo citare risorse uniche che non possono essere duplicate, come porte di comunicazione, finestre video, eccetera; in tal caso, è sufficiente dichiarare l'operatore di assegnazione come privato, e poi non definirlo affatto nel codice (fornendo un opportuno commento). Definendolo come privato, otteniamo un messaggio di errore dal compilatore se tentiamo di assegnare un oggetto di tale classe; non definendolo, otterremmo comunque un messaggio di errore dal linker se tentassimo erroneamente di utilizzarlo nel codice della classe, o all'interno di una funzione friend. Il **Listato 104** propone un esempio di quanto sopra. Ovviamente, è in genere necessario utilizzare la tecnica congiuntamente per il costruttore di copia e per l'operatore di assegnazione.

Listato 104

```
class CannotAssign
  {
  public :
    CannotAssign()
  private :
    const CannotAssign& operator =(
      const CannotAssign& s ) ;
  } ;


int main()
  {
  CannotAssign ca1 ;
  CannotAssign ca2 ;

  ca1 = ca2 ; // ERRORE di compilazione

  return( 0 ) ;
  }
```

Come nel caso del costruttore di copia, non è necessario che una classe implementi l'operatore di assegnazione in modo tale che venga a tutti gli effetti creata una copia di tutti i membri dinamici: gli oggetti allocati dinamicamente potrebbero essere tranquillamente condivisi, come avviene in molte implementazioni "copy-on-write" della classe String. Vedere il

paragrafo “Costruttori di copia” in questo stesso capitolo per ulteriori dettagli.

Raccomandazione 88

Una classe che abbia membri allocati dinamicamente deve definire un opportuno operatore di assegnazione.

Raccomandazione 89

Un operatore di assegnazione che esegua operazioni distruttive sul suo operando di sinistra, deve per prima cosa verificare la non-identità dei due operandi.

Così come nei costruttori è opportuno che i parametri non vengano modificati, anche nelle assegnazioni sarebbe auspicabile che il parametro di destra non venisse modificato, in modo da evitare side-effects sempre difficili da gestire in fase di manutenzione del codice. Esistono rare eccezioni, tra cui il caso delle classi duali già citato, ma in generale il parametro destro di un operatore di assegnazione dovrebbe essere un reference const.

Raccomandazione 90

Il parametro di destra di un operatore di assegnazione dovrebbe essere const; nei rari casi in cui ciò non è possibile, commentare adeguatamente l’header della classe.

Ricordate sempre che l’operatore di assegnazione *non* viene ereditato dalle classi derivate; ciò è facilmente comprensibile, considerando che se una classe aggiunge membri dato propri, l’operatore di assegnazione della classe base non sarebbe idoneo ad assegnare elementi di classe derivata. Normalmente, l’operatore di assegnazione per la classe derivata dovrà chiamare l’operatore di assegnazione delle classi base.

Infine, consideriamo il valore di ritorno per un operatore di assegnazione; innanzitutto, non vi è restrizione sul tipo del risultato, anche se sembra logico mantenere consistente l’arità standard dell’operatore e restituire un reference ad un oggetto della classe. A quale oggetto, in ogni caso? Possiamo restituire un reference al parametro di destra o sinistra, ed un

reference const o non-const. Nessuna di queste considerazioni ha realmente valore, sino a quando l'operatore di assegnazione non viene utilizzato in contesti abbastanza discutibili, come nel **Listato 105**:

Listato 105

```
class C
  {
  // ...
  C& operator =( C& c ) ;
  } ;


int main()
  {
  C c1 ;
  C c2 ;
  C c3 ;

  ( c1 = c2 ) = c3 ;


  return( 0 ) ;
  }
```

In casi simili, il risultato dell'assegnazione multipla dipende dall'implementazione dell'operatore di assegnazione: se restituisce un reference al parametro di sinistra, l'assegnazione sarà equivalente al semplice *c1 = c3*, mentre se restituisce un reference al parametro di destra, l'assegnazione sarà equivalente a *c1 = c2; c2 = c3.*

È evidente che tali ambiguità nel punto di lettura sono altamente indesiderabili; esistono diverse alternative che consentono di risolvere il problema, in modo più o meno drastico.

1. dichiarare il risultato dell'operatore di assegnazione come *void.* In tal modo, ogni assegnazione multipla genera un errore di compilazione. Ciò costituirebbe un rinforzo di una norma di buona programmazione, ovvero che in ogni riga vi deve essere un solo statement.

2. dichiarare sempre il parametro destro come const, e restituire sempre il sinistro come non-const. Questo tuttavia corrisponde all'interpretazione meno utile del **Listato 105**, ovvero *c1 = c3*.

3. restituire un reference const; a questo punto, non vi è più differenza tra restituire un reference al parametro di sinistra o di destra, e l'esempio del **Listato 105** diventa nuovamente illegale. Rispetto all'alternativa (1) di un risultato void, questa tecnica consente la consueta assegnazione multipla, senza parentesi, *c1 = c2 = c3*, da leggersi *c1 = (c2 = c3)*.

Indubbiamente, una buona norma di programmazione richiede l'adozione di (1) o di (3) nella dichiarazione degli operatori di assegnazione. La scelta è largamente dipendente dal giudizio personale; essendo la (1) più limitativa, in genere molti preferiscono la (3).

Raccomandazione 91

Il risultato di un operatore di assegnazione dovrebbe essere di tipo void oppure un reference const alla classe per la quale l'operatore è definito.

Efficienza

Non vi è alcuna differenza tra l'efficienza di una funzione Add che prenda due parametri di classe (ad es.) Complex e ne restituisca la somma, ed un operatore + che implementi la stessa operazione. Inoltre, sia le funzioni che gli operatori possono essere espansi in linea quando necessario, evitando l'overhead della chiamata di funzione. L'unico vero punto a sfavore degli operatori definiti dall'utente, rispetto alle funzioni, è la creazione di temporanei che potrebbero essere evitati.

Consideriamo il **Listato 106**: l'assegnazione *c = a + b* può comportare la creazione di un oggetto temporaneo per mantenere il valore di *a + b* prima dell'assegnazione a *c*. Al contrario, sarebbe possibile definire una funzione AddAndAssign() con tre parametri, che somma i primi due e li assegna al terzo, e che non coinvolge la creazione di oggetti temporanei.

Listato 106

```
class Complex
  {
  public :
    Complex operator +( Complex a, Complex b ) ;
  // ...
  } ;
```

```
int main()
  {
  Complex a( 1, 0 ) ;
  Complex b( 0, 1 ) ;
  Complex c( 2, 2 ) ;

  // ...

  c = a + b ;

  // ...

  return( 0 ) ;
  }
```

A dire il vero, ciò non costituisce un grave problema; chi desidera o ha necessità di spremere l'ultima goccia di efficienza dal proprio codice, può sicuramente realizzare funzioni molto specializzate come AddAndAssign(), ma nei casi di utilizzo non pesante ci si può affidare alle capacità del compilatore di ottimizzare il codice. Osserviamo infatti che non è sempre necessario creare un temporaneo, e se il compilatore esegue una analisi anche conservativa degli alias, dovrebbe essere in grado di ottimizzare assegnazioni come $a = b + c$ ogni volta che b, c, l'operatore + e l'operatore = non dipendono dal valore di a. Ciò dovrebbe essere abbastanza comune nella buona programmazione, soprattutto considerando che, come vedremo nel capitolo 8, l'uso dei reference per creare alias è altamente sconsigliato, e che invece non è possibile l'overload degli operatori sui tipi puntatore.
Esistono comunque molte tecniche per evitare la creazione dei temporanei. Alcune sono a carico del compilatore, altre sotto il controllo del programmatore. Per una panoramica, si veda [Pes97].

Friend

Capita talvolta che una o più funzioni membro, concettualmente private, debbano essere chiamate da un'altra funzione, non appartenente alla classe stessa e neppure ad una classe derivata. Un caso abbastanza comune è il partizionamento di una entità logica in due classi, altamente coesive e degne di vita autonoma, che tuttavia devono condividere alcune funzionalità. Analoghe situazioni esistono per membri dato: ciò può accadere per diversi motivi, tra cui va spesso annoverata la velocità di esecuzione del codice [Str91].

Uno dei casi tipici, che useremo come esempio, è la definizione di operatori overloaded friend, come nel **Listato 107**:

Listato 107

```
class Complex
  {
  public :
    Complex( double r, double i = 0 ) ;
    friend Complex operator +( Complex, Complex ) ;
    friend Complex operator *( Complex, Complex ) ;

  private:
    double re ;
    double im ;
  } ;
```

In casi simili, l'operatore è dichiarato esternamente alla classe in modo da non dover definire ogni possibile combinazione di *Complex* e *double* per gli operatori + e *, sfruttando il costruttore di conversione (viceversa, non sarebbe stato possibile sommare un *double* ed un *Complex* direttamente). Deve tuttavia avere accesso all'implementazione di *Complex*, ed è quindi dichiarato friend della classe.

Dichiarare una funzione od una classe come *friend* è una forma di accoppiamento, non diversa dal derivare una classe da un'altra; si tratta tuttavia di uno strumento potente, che va quindi usato con moderazione, per evitare accoppiamenti eccessivi.
Anche in questo caso, se avete troppi *friend* nel vostro codice, è opportuno che vi fermiate a riconsiderare il design.
Tuttavia, se usato con cautela, il *friend* può sensibilmente aumentare l'incapsulazione, permettendo di isolare strutture dati che andrebbero altrimenti esposte, o evitare di fondere in un'unica classe, per ragioni implementative, due o più entità concettualmente separate.

In linguaggi più espressivi del C++, il concetto di *friend* è sostituito da un meccanismo più evoluto e flessibile: una classe può dichiarare delle *interfacce alternative*, permettendo così a classi diverse di avere una visione diversa, e quindi anche un accesso diverso, alla classe stessa.
Il grande vantaggio in tal caso è la selettività: mentre il friend rende visibile l'intera parte protetta e privata di una classe, una interfaccia alternativa espone solo la parte strettamente necessaria, aumentando così l'incapsulazione.

In C++, invece, deve essere il programmatore ad imporre un certo ordine nell'uso del friend, per evitare di avere classi troppo strettamente accoppiate; ricordando quanto detto in questo stesso capitolo nel paragrafo "membri dato pubblici e protetti", possiamo osservare che le stesse considerazioni si applicano a dati cui si acceda da una funzione/classe friend.
Un buon metodo per mantenere il controllo rispetto all'abuso del friend è di non accedere direttamente ai dati membro dalle funzioni friend, ma solo alle funzioni membro (per quanto private); l'accesso ai dati deve avvenire attraverso funzioni *private* inline di Get/Set. In tal modo, il numero di funzioni *private* di Get/Set (che sono inutili internamente alla classe) ci permetterà di controllare rapidamente il grado di accoppiamento con funzioni friend (semplicemente contandole, se si ha una convenzione sui nomi), ed il semplice fatto di dover scrivere una funzione extra scoraggerà l'uso "sporco" del friend, invitando invece il programmatore a riflettere su eventuali alternative.
Questo è uno dei pochi casi in cui una regola di buona programmazione incrementa deliberatamente il lavoro del programmatore, al fine di permettere un maggior controllo della qualità del codice.

Raccomandazione 92

Le funzioni friend non devono accedere ai dati privati della classe; se necessario, definire delle funzioni inline private di Get/Set, per accedere ai dati membro.

Controllo della derivazione

La dichiarazione di una funzione o classe come *friend* non è ereditata né transitiva; ciò in effetti riflette la vita reale, dove gli amici dei genitori non sono necessariamente amici dei figli, e gli amici degli amici non sono necessariamente nostri amici.
Questa particolarità può essere utilizzata per creare una classe dalla quale è impossibile derivare ulteriormente, una caratteristica che può risultare molto utile in alcuni casi. L'idea di base è dichiarare una classe con costruttore privato, derivare da essa la nostra classe "terminale" come *virtual* e dichiarare la classe terminale come *friend* all'interno della classe base.
Se deriviamo dalla classe terminale un'ulteriore classe, essa dovrà chiamare il costruttore della classe base (poiché la classe terminale è derivata come

virtual), ma essendo il costruttore privato e la nuova classe non-friend, il compilatore segnalerà un errore.
Un esempio è visibile nel **Listato 108**:

Listato 108

```
class BaseOfTerminal
  {
  private :
    BaseOfTerminal() ;
  friend Terminal ;
  } ;

class Terminal : public virtual BaseOfTerminal
  {
  Terminal() ;
  ~Terminal() ;
  // ...
  } ;

class Illegal : public Terminal
  {
  } ;

Terminal t ; // OK

Illegal i ;  // Errore!
```

La Legge di Demeter

La legge di Demeter è stata formulata alla Northeastern University [LH92], come tentativo di fornire una definizione, almeno parziale, di "buon stile di programmazione nel paradigma object oriented". In tal senso, è quindi molto interessante valutarne in questa sede l'impatto sulle regole di "buona codifica" in C++.

Esistono diverse formulazioni della legge di Demeter, più o meno formali e più o meno restrittive; alcune delle formulazioni hanno lo scopo di consentire una verifica automatica, da parte di un opportuno strumento, del rispetto della legge stessa. Tuttavia, per i nostri fini è sufficiente analizzare le conseguenze di una formulazione informale, ma semplice da capire, anche se non direttamente automatizzabile tramite un tool. Vedremo che tale formulazione rinforza alcune delle precedenti raccomandazioni e ne suggerisce delle nuove.

L'idea fondamentale della legge di Demeter è di *restringere* lo spazio dei metodi richiamabili all'interno di ogni singolo metodo; più precisamente, verrà definito un insieme di metodi "privilegiati" che possono essere chiamati all'interno di un altro metodo, mentre ogni chiamata ad altri metodi dovrebbe essere evitata. Il fine di tale restrizione è di strutturare e ridurre le "dipendenze comportamentali" tra le classi: quando leggiamo il codice per un metodo di una classe, è importante essere a conoscenza della struttura della classe stessa, e conoscere almeno l'interfaccia di ogni altra classe utilizzata all'interno del metodo (ad esempio, i parametri del metodo). Se il metodo in questione ottiene l'accesso a sottoparti di un parametro, dovremo conoscere almeno l'interfaccia di tale sottoparte per seguire il comportamento logico del codice, e così via. È evidente quindi che chiamare metodi di altre classi è una forma di *accoppiamento*, che se troppo elevato rischia di minare la comprensibilità del codice (nel peggiore dei casi, la comprensione del codice di un singolo metodo potrebbe richiedere la conoscenza del comportamento di ogni classe coinvolta nel sistema).

Osserviamo inoltre che, accedendo ad esempio alle sottoparti di un'altra classe all'interno di un metodo, stiamo cristallizzando nel codice del metodo alcune assunzioni circa la struttura di altre classi; mentre ciò può essere accettabile per classi la cui struttura è estremamente stabile (come una classe Complex formata da una parte reale ed una immaginaria), l'accoppiamento strutturale può essere estremamente complesso da mantenere durante l'evoluzione del software (per tale ragione lo abbiamo scoraggiato, impedendo l'accesso ai membri dato e suggerendo di evitare i metodi di Get/Set).

Raccomandazione 93
Legge di Demeter (formulazione "per oggetti")

Un metodo M della classe C dovrebbe richiamare solo metodi della classe C, metodi delle sottoparti *immediate* di C, metodi degli argomenti di M, metodi degli oggetti creati *internamente* ad M o metodi di oggetti globali.

La legge è molto semplice da capire: all'interno di ogni metodo, possiamo chiamare liberamente i metodi della stessa classe, i metodi delle sottocomponenti della classe (ma non scendere alle sottocomponenti delle sottocomponenti, e così via), i metodi di eventuali oggetti passati come parametro, i metodi di oggetti creati localmente (attenzione, devono essere effettivamente creati) oppure di oggetti globali. In effetti, come vedete,

apparentemente non si tratta di una legge molto restrittiva; spesso riesce comunque nel suo intento di rendere i programmi più comprensibili per chi legge.

Vediamo alcuni esempi di violazione della legge, per renderci conto di quali costrutti essa tenda a bandire: nel **Listato 109**, un computer è descritto in termini dei suoi componenti, ognuno dei quali ha un metodo *SelfTest*(); un componente integrato seriale/parallela ha invece due metodi per esporre i suoi costituenti. La parte interessante è il metodo *SelfTest*() per la classe *Computer*, che richiama l'omonimo metodo sulle sottoparti: mentre è lecito chiamare *SelfTest* sulle sottoparti elementari (es. *ram*), la chiamata sulle sottoparti del componente integrato *per* non rispetta la legge di Demeter. La soluzione nel caso in questione è banale (fornire di un metodo *SelfTest*() la classe *Peripheral*), così come la positiva influenza di tale modifica sulla mantenibilità del codice (se, ad esempio, *Peripheral* venisse estesa per gestire più di una seriale o parallela).

Listato 109

```
class RAM
  {
  public :
    int SelfTest() const ;
  // ...
  } ;


class CPU
  {
  public :
    int SelfTest() const ;
  // ...
  } ;


class Parallel
  {
  public :
    int SelfTest() const ;
  // ...
  } ;


class Serial
  {
```

```
  public :
    int SelfTest() const ;
  // ...
  } ;


class Peripheral
  {
  public :
    const Serial& GetSerial() const
      { return( ser ) ; }
    const Parallel& GetParallel() const
      { return( par ) ; }
  private :
    Serial ser ;
    Parallel par ;
 };


class Computer
  {
  public :
    int SelfTest()  const ;
  private :
    CPU cpu ;
    RAM ram ;
    Peripheral per ;
  } ;


int Computer :: SelfTest() const
  {
  int c = cpu->SelfTest() ;               // OK
  int r = ram->SelfTest() ;               // OK
  int s = per->GetSerial()->SelfTest() ;  // NO!!
  int p = per->GetParallel->SelfTest() ;  // NO!!

  int res = ...
  return res;
  }
```

Il secondo esempio è interessante perché pone l'accento sulla differenza tra il rispetto letterale della legge ed un suo rispetto concettuale. Nel **Listato 110**, un libro è formato da una copertina frontale, da una lista di capitoli, e da una copertina sul retro; sia le copertine che i capitoli hanno un metodo di *Print*(), e la lista di capitoli permette di accedere al singolo elemento della lista.

Listato 110

```
class Cover
  {
  public :
    void Print() const ;
  // ...
  } ;

class Chapter
  {
  public :
    void Print() const ;
  // ...
  } ;

class ChapterList
  {
  public :
    const Chapter& operator []( int i ) const ;
    int NumOfChapters() const ;
  // ...
  } ;

class Book
  {
  public :
    void Print() const ;
  private :
    Cover frontCover ;
    Cover backCover ;
    ChapterList chapters;
  } ;

void Book :: Print() const
  {
  frontCover.Print() ;                  // OK
  int chapterNum = chapters.NumOfChapters() ;
  for( int i = 0; i < chapterNum; i++ )
    chapters[ i ].Print() ;             // NO!!
  backCover.Print() ;                   // OK
  }
```

La violazione è nuovamente dovuta alla chiamata di un metodo per un sotto-componente di una delle componenti. Possiamo ora riportare il listato

entro i confini della legge di Demeter in due modi: dichiarando un metodo *Print(int i)* per la classe *ChapterList*, che stampa il singolo capitolo, o dichiarando per la stessa classe un metodo *Print()* che stampa l'intera lista di capitoli.

La prima soluzione è una traslazione diretta, semi-automatica, del codice originale; in questo senso, è una cattiva soluzione, poiché rispetta la legge a livello formale, ma non utilizza la legge come uno strumento concettuale per migliorare la struttura del codice. La seconda soluzione corrisponde a "pensare in termini object oriented": anziché accedere, sia pure in modo indiretto, al singolo capitolo, rendiamo la lista di capitoli un oggetto autonomo, in grado di stampare se stesso senza un controllo esterno.
In effetti, il difetto principale della legge di Demeter è proprio l'incoraggiamento della proliferazione di metodi di forwarding, ovvero metodi che propagano alcune chiamate alle sottoparti di una classe. In realtà, ciò accade maggiormente se si cerca di rispettare la legge a livello formale, anziché rispettarla a livello concettuale.

Avendo osservato su due esempi concreti le possibili violazioni della legge, e le possibilità di ricondurre il codice entro i limiti della legge stessa (si può peraltro dimostrare [LHR88] che *ogni* programma object oriented si può ricondurre entro la legge di Demeter), vediamo ora in maggiore dettaglio quali sono i principali vantaggi nel seguire la legge stessa:

- Controllo dell'accoppiamento: come abbiamo visto poc'anzi, il rispetto della legge di Demeter riduce l'accoppiamento dinamico e l'accoppiamento strutturale.

- Information hiding: come conseguenza del ridotto accoppiamento strutturale, abbiamo anche un effettivo occultamento della struttura. In questo senso, il rispetto delle raccomandazioni viste in precedenza (dati solo privati, riduzione dei metodi di get/set protetti, no get/set pubblici) porta comunque a codice più aderente ai principi della legge di Demeter.

Restrizione e localizzazione dell'informazione: come risultato del ridotto accoppiamento dinamico, si ha anche una riduzione dei punti in cui è necessario accedere a determinate informazioni, che quindi vengono utilizzate in modo più localizzato. Ciò è vantaggioso in termini di comprensione del codice, e può anche avere un impatto positivo sulle prestazioni, su sistemi a memoria virtuale.

Ovviamente, in determinati casi la legge può essere violata, sia per esigenze di efficienza, sia per la stabilità della classe (come nell'esempio già citato di una classe *Complex*); si tratta comunque di casi che meritano un commento esplicativo. In ogni caso, la "legge" è qui presentata, nello spirito del libro, come una "raccomandazione", che al pari delle altre ha soprattutto una valenza di suggerimento e guida, piuttosto che di dogma imprescindibile.

Esiste infine uno strumento [SHS93] che genera automaticamente le funzioni di forwarding per le varie classi, riducendo così la quantità di codice da scrivere. Tuttavia tale strumento suggerisce una soluzione pragmatica, più che concettuale, e se da un lato riduce il lavoro dello sviluppatore, dall'altro costituisce una via forse troppo semplice per evitare di risolvere il problema ad un più alto livello di astrazione.

8

Puntatori e Reference

"...By indirections find directions out"
William Sharkespeare

Puntatori e reference sono elementi concettualmente distinti, che in alcune occasioni possono essere utilizzati per gli stessi fini. È pertanto importante decidere in quali occasioni gli uni siano più adatti degli altri, o meglio quando sia più sicuro o corretto utilizzare i reference e quando i puntatori.

Informalmente, un valore di tipo puntatore è una locazione per il tipo puntato, mentre un reference è semplicemente un alias per la locazione con la quale viene inizializzato, ovvero un nome alternativo per raggiungere tale locazione. I puntatori possono cambiare valore, mentre gli alias no, come è evidenziato nel **Listato 111**:

Listato 111

```
int y = 3 ;
int z = 4 ;

int& ref = y ;
int* ptr = &y ;

ref = z ;    // assegna 4 ad y
ptr = &z ;   // assegna a ptr l'indirizzo di z
```

Osserviamo che riassegnando un valore al reference *ref* modifichiamo in realtà il contenuto della variabile *y* per cui *ref* è un alias, mentre riassegnando un valore al puntatore *ptr* modifichiamo di fatto il puntatore stesso.

Esistono tre casi fondamentali in cui i reference vengono utilizzati, ed in due di essi è altresì possibile utilizzare i puntatori:

1. Per ottenere il passaggio per riferimento dei parametri, come il VAR del Pascal. In C, dove non esiste il tipo reference, il passaggio dei parametri avviene unicamente per valore. Quando si debba passare un parametro che si vuole modificare, o la cui dimensione sconsiglia il passaggio per valore, si ricorre al passaggio (per valore) del puntatore all'oggetto. Ciò è particolarmente fastidioso nel caso non si voglia modificare l'oggetto stesso, in quanto all'interno del codice della funzione si dovrà comunque dereferenziare il puntatore. In tal senso, l'uso dei reference è migliore perché equivale ad un vero e proprio information hiding, sia nel punto di chiamata, dove non è necessario ottenere l'indirizzo del parametro attuale, sia all'interno della funzione, dove non è necessario dereferenziare il parametro formale[18]. In effetti, rendere evidente il passaggio per riferimento usando i puntatori è un retaggio del basso livello di astrazione del C; non dovrebbe essere necessario evidenziare dettagli implementativi, per quanto semplici, come l'uso di un puntatore nel passaggio per riferimento. L'unico problema è che non si dispone più del valore NULL sui reference; per questo aspetto, rimando al capitolo successivo.

2. Per restituire un *left value*, tipicamente come risultato di un operatore overloaded. Un caso molto comune è la definizione dell'operatore di output <<, come nel **Listato 112**. In questi casi è necessario restituire un left value per poter concatenare gli operatori, come in *cout << c1 << c2 ;* notiamo che in questo caso l'alternativa dei puntatori non è percorribile, in quanto non è possibile ridefinire gli operatori per i tipi puntatore. L'introduzione dell'overloading degli operatori è stata di fatto la vera ragione per l'introduzione dei reference nel C++.

Listato 112

```
class C
  {
  friend ostream& operator<<( ostream& os, C c ) ;
  // ...
  } ;
```

[18]In Ada, la struttura del linguaggio consente al compilatore di introdurre automaticamente la dereferenziazione dei puntatori, sgravando il programmatore e soprattutto chi legge il codice da dettagli di basso livello. Non esiste pertanto l'analogo dell'operatore '->', ma solo l'operatore '.'.

3. Per dichiarare un alias, ovvero un nome alternativo con cui referenziare una zona di memoria (come nel **Listato 111**). In casi simili, dovremmo assolutamente evitare di usare i reference, ed utilizzare invece i puntatori. Vi sono ragioni storiche e ragioni pragmatiche per evitare l'uso dei reference come alias; in generale, gli alias sono pericolosi e vanno tenuti sotto controllo. I programmatori C e C++ sono da sempre attenti agli alias tra puntatori: leggendo il codice, tengono automaticamente in considerazione la possibilità che due o più puntatori puntino alla stessa locazione. I reference, tuttavia, hanno una "storia" molto più breve, ed i programmatori non sono abituati a cautelarsi nei loro confronti; ovviamente, ogni stile di programmazione che aggiunga inutili fardelli al programmatore va evitato. Inoltre, mentre l'uso del puntatore per modificare il valore puntato è esplicito, l'uso dell'alias è implicito (**Listato 113**). È proprio nel momento della dereferenziazione di un puntatore (oltre che al momento della distruzione) che scatta il "campanello di allarme" dei programmatori a proposito dell'aliasing. Nel caso dei reference, invece, tutto avviene in modo pericolosamente trasparente: con riferimento al **Listato 113**, pensate quanto sarebbe semplice non notare che l'assegnazione a *ref* sta in realtà modificando *z* e non *ref*, se l'istruzione si trovasse a qualche dozzina di righe dalla inizializzazione di *ref*.

Listato 113

```
int y = 3 ;
int z = 4 ;

int* ptr = &y ;
*ptr = 5 ;          // dereferenziazione esplicita

int& ref = z ;
ref = 5 ;           // nessuna dereferenziazione
```

Raccomandazione 94

Non usate i reference per creare degli alias; se dovete creare alias, utilizzate i puntatori.

Raccomandazione 95

Per ottenere il passaggio per riferimento, utilizzate sempre i tipi reference e non i puntatori. Se il fine del passaggio per riferimento è di evitare la copia di un oggetto di grosse dimensioni, che non verrà comunque modificato, dichiarare il parametro come un reference const.

Aritmetica sui puntatori

Una delle caratteristiche negative (dal punto di vista della sicurezza del codice) che il C++ ha ereditato dal C è la conversione "liberale" di array in puntatori: in un contesto di espressione, il compilatore può convertire un array in un puntatore al suo primo elemento. Per quanto ciò possa risultare estremamente comodo in diverse occasioni, con i moderni compilatori è raramente necessario fare ricorso all'aritmetica sui puntatori per migliorare le prestazioni del codice; tranne casi particolari, l'uso degli array produce esattamente lo stesso codice. Considerando il **Listato 114**, qualunque compilatore con un minimo di capacità ottimizzanti produrrà codice identico per i due loop, sia che venga usata l'aritmetica sui puntatori sia che si utilizzi l'array come tale.

Listato 114

```
int main()
  {
  char a[ 10 ] = "123456789";

  // ...


  // usa aritmetica sui puntatori
  char* p = a ;
  while( *p )
    {
    // ...
    p++ ;
    }


  // usa l'array come array!
  int i = 0 ;
  while( a[ i ] )
    {
```

```
    // ...
    i++ ;
    }

  return( 0 ) ;
  }
```

Il vantaggio nel *non* usare l'aritmetica sui puntatori (oltre alla leggibilità, che per alcuni risulta maggiore) è che si riducono le possibilità di errori, come usare *delete* su un puntatore che non punta al primo elemento, e che è possibile modificare il codice per utilizzare una classe Array basata su template senza modifiche sostanziali al codice.
L'uso di una classe Array è fortemente incoraggiato, in quanto consente spesso controlli a compile-time ed a run-time non disponibili con gli array C-style, ed evita uno dei punti deboli del type-system del C++; una buona classe Array avrà spesso un impatto minimo sulle prestazioni del vostro codice, garantendo nel frattempo una sicurezza di gran lunga superiore all'array C-style.

Raccomandazione 96

Limitare l'uso dell'aritmetica sui puntatori: in molte occasioni, l'uso diretto degli array genererà codice più leggibile, sicuro, ed altrettanto efficiente.

Raccomandazione 97

Utilizzate una classe Array basata su template anziché gli array C-style.

Delete

La succitata conversione da array a puntatore è la causa di un errore abbastanza frequente per i programmatori C++: l'uso di *delete* anziché *delete[]* quando si vuole distruggere un array di oggetti; vedere **Listato 115**.

Listato 115

```
class X
  {
  // ...
  }
```

```
int main()
  {
  X* obj = new X() ;
  X* array = new X[ 10 ] ;

  delete obj ;      // OK
  delete array ;    // NO: usare delete[] array

  return( 0 ) ;
  }
```

L'uso della forma appropriata per *delete* è lasciato al programmatore: l'uso della forma errata ha risultato indefinito. Alcuni compilatori utilizzano lo stesso heap per l'allocazione di singoli oggetti ed array, nel qual caso se la classe non ha distruttori la presenza o meno di [] nella distruzione di un array è irrilevante. Tuttavia si tratta di un comportamento assolutamente particolare sul quale non dovete altrettanto assolutamente fare affidamento: mentre il comportamento del compilatore è legale ("indefinito" significa che l'operazione può anche andare a buon fine) tale uso del linguaggio non lo è, e il vostro codice non funzionerebbe su un compilatore che utilizzi, ad esempio, heap diversi per oggetti singoli ed array. Notiamo che nuovamente una classe Array ci proteggerebbe dall'errore, poiché la distruzione effettiva dell'array di oggetti (ammesso che esista) avverrebbe internamente al distruttore della classe, mentre dall'esterno noi dovremmo sempre e comunque distruggere un singolo oggetto (di classe Array).

Raccomandazione 98

Utilizzate sempre *delete[]* per distruggere array di oggetti, e *delete* per distruggere i singoli oggetti.

Un altro bug classico nella programmazione C++ è l'uso di un puntatore che punta a memoria deallocata: il **Listato 116** mostra un tipico esempio di codice C++ errato.

Listato 116

```
int main()
  {
  char* p = new char[ 100 ] ;

  // ...
```

```
delete[] p ;

// ...
*p = 0 ;


return( 0 ) ;
}
```

La probabilità di errori simili aumenta se il puntatore è un membro di una classe, che può essere distrutto come side-effect di alcuni metodi, mentre alcuni altri assumono che punti a memoria allocata. Ma è anche un classico bug introdotto nella manutenzione di funzioni troppo lunghe, o con un flusso del controllo troppo convoluto per essere efficacemente compreso. Il problema maggiore di questo tipo di errori è che talvolta non viene rilevato immediatamente, ma solo in seguito, quando la corruzione di altre strutture dati genera problemi tali da risultare in una violazione nell'accesso alla memoria; in tal senso, è un bug particolarmente insidioso perché può essere difficile replicare il problema, specialmente tra macchine e architetture diverse, e perché non causa un crash immediato del programma, che ne renderebbe più semplice l'individuazione con un debugger.
Una tecnica piuttosto banale per cautelarsi da questo tipo di errore è assegnare NULL al puntatore subito dopo la sua deallocazione: in tal modo, il primo uso illecito del puntatore genererà un'eccezione (sui sistemi con protezione della memoria). Ciò andrebbe sempre fatto per i puntatori che siano membri di una classe, tranne che nel distruttore della classe stessa; infatti, nel caso delle classi la allocazione, l'uso e la deallocazione dei dati membro avviene normalmente in funzioni diverse, ed è più complesso tenere traccia delle diverse azioni. Per le variabili locali, l'uso o meno della tecnica è largamente dipendente dalla lunghezza dello scope della variabile: se viene dichiarata, inizializzata, usata, distrutta ed esce dallo scope in poche righe, la probabilità di riutilizzarla dopo il ***delete*** è pressoché nulla; se viceversa lo scope della variabile è piuttosto lungo, e si estende per molte righe dopo la distruzione, può essere consigliabile assegnarle NULL dopo la deallocazione. Le variabili globali di tipo puntatore (comunque sconsigliate, vedere capitolo 6) dovrebbero anch'esse subire lo stesso trattamento dei puntatori membro, per le stesse ragioni.

Raccomandazione 99

Dopo la chiamata a *delete* o *delete[]*, assegnate il valore NULL alla variabile nel caso si tratti di un membro di una classe o di una variabile globale. Lo stesso vale se si tratta di una variabile locale il cui scope si estende per molte righe dopo la sua distruzione.

Puntatori a puntatori

Se i puntatori sono la croce e delizia dei programmatori C (e C++), i puntatori a puntatori non sono altrettanto amati: i principianti cercano di evitarli perché sono più complessi da maneggiare, gli esperti cercano di evitarli perché sono sensibilmente più lenti dei puntatori singoli. Esistono diverse situazioni tipiche ("pattern") di utilizzo dei puntatori a puntatori, che vedremo di seguito, insieme ad alcuni consigli su come evitarli:

- quando occorre un numero variabile di oggetti la cui classe non ha un costruttore di default: in tal caso si utilizza spesso la tecnica del **Listato 117**.

Listato 117

```
class X
  {
  public :
    X( int n ) ;
  // no costruttore di default
  // ...
  } ;

class Y
  {
  public :
    Y( int n ) ;
  private :
    X** x ;
  } ;


Y :: Y( int n )
  {
  x = new X*[ n ] ;
  for( int i = 0; i < n; i++ )
```

```
    x[ i ] = new X( n ) ;
  }
```

Notiamo che se la grandezza dell'array (qui simulato con doppi puntatori) fosse nota a compile-time, potremmo usare un array di puntatori, e che se la classe *X* avesse un costruttore di default, potremmo invece usare un puntatore singolo, facendolo puntare al primo elemento di un array allocato dinamicamente.

Va osservato che nuovamente è possibile usare una classe "Array indiretto", normalmente presente nelle librerie di classi contenitore, per risolvere il problema senza utilizzare i puntatori a puntatore; in molti casi, inoltre, ripensando meglio le strutture dati è possibile ottimizzare l'accesso: ad esempio, se sappiamo che l'accesso ad *x* avverrà sempre in modo sequenziale, possiamo organizzare i suoi elementi in una lista. Di fronte ad una variabile di tipo puntatore a puntatore, dovremmo sempre chiederci se "pensando in C++" non è possibile ridurre il numero di indirezioni, perlomeno apparenti.

- quando strutture dati complesse non sono incapsulate in classi: classico è l'esempio di una matrice. In tal caso, valgono le stesse osservazioni del punto precedente.

- quando un parametro puntatore deve essere modificato da una funzione: in tal caso, tuttavia, è preferibile passare il puntatore per riferimento, non un puntatore a puntatore (vedi raccomandazioni precedenti in questo stesso capitolo).

Raccomandazione 100

Minimizzate l'uso di puntatori a puntatore.

Smart pointers

In molte occasioni può essere utile definire una classe con un unico membro puntatore, e dare alla classe una semantica di "puntatore intelligente"; tali puntatori potrebbero eseguire automaticamente dei controlli di validità prima della dereferenziazione, oppure essere utilizzati come base di supporto per strutture più sofisticate, come gli oggetti persistenti o la garbage collection.

L'idea dei puntatori intelligenti è introdotta anche su [Str91], ove viene illustrata con un esempio simile al **Listato 118**, dove la classe puntatore non è "intelligente" rispetto al comune puntatore, ma dimostra abbastanza chiaramente le possibili tecniche (una possibile evoluzione potrebbe utilizzare un template).

Listato 118

```
class PtrY
  {
  public :
    PtrY( const Y* p ) { ptr = p ; }

    Y* operator ->() { return( ptr ) ; }
    Y& operator *() { return( *ptr ) ; }
    Y& operator []( int i ) { return( ptr[ i ] ) ; }

  private :
    Y* ptr ;
  } ;
```

Oltre ad aggiungere "l'intelligenza interna", una classe completa che implementi degli smart pointers dovrebbe ovviamente gestire l'incremento ed il decremento prefisso e postfisso, la differenza di puntatori, la somma tra puntatore ed intero, e così via. Vi è comunque un problema molto rilevante, non trattato in [Str91], che riguarda le conversioni di tipo.

Considerando i comuni puntatori C++, esiste una serie di conversioni implicite, che il compilatore può eseguire quando necessario, nonché una serie di priorità tra le possibili conversioni, che vengono utilizzate per discriminare situazioni ambigue. Un esempio di conversione è quella da puntatore ad una classe derivata a puntatore alla classe base, che insieme alle funzioni virtuali forma la base del polimorfismo in C++. Per la sua estrema importanza, una implementazione completa degli smart pointers dovrebbe fornire almeno questo tipo di conversione. Notiamo che non si ha alcuna conversione automatica tra le classi degli smart pointers, poiché tali classi non sono necessariamente derivate tra loro.

Esistono al proposito sostanzialmente quattro approcci per risolvere il problema; per semplificare l'esposizione, assumeremo che la classe degli smart pointers ad oggetti di classe Y sia chiamata PtrY, come nell'esempio precedente.

1. Mantenere ogni classe che implementa uno smart pointer disgiunta dalle altre, e fornire all'interno di PtrY degli operatori di conversione a PtrX, per ogni classe X da cui Y sia direttamente derivata.

2. Come (1), ma fornendo gli operatori di conversione ad ogni classe PtrX, per ogni classe X da cui Y sia direttamente *o indirettamente* derivata (nel caso, ovviamente, di gerarchie di derivazione con profondità superiore alle 2 classi).

3. Creare una gerarchia anche tra le classi degli smart pointer, derivando quindi PtrY da PtrX, per ogni X da cui Y sia direttamente derivata. Notiamo che in questo caso sfruttiamo, per simulare la conversione da Y* ad X*, la conversione da PtrY& a PtrX&.

4. Come (3), ma con derivazione virtuale tra le classi Ptr, e derivando tutti da una virtual base class che contiene l'unico membro di tipo puntatore. Notiamo che nel caso (3) si ha invece una replicazione del membro puntatore (*ptr* nell'esempio dato) per ogni classe derivata.

Ogni approccio ha alcuni problemi, che vedremo brevemente di seguito:

Il metodo (1) non consente la conversione tra smart pointers quando la prodondità della derivazione è superiore ad 1. Questo può costituire un forte fattore limitativo.

Il metodo (2) non soffre di tale problema, ma può invece portare ad ambiguità nelle chiamate a funzioni overloaded. Infatti, mentre per le conversioni implicite tra puntatori il C++ cerca di risolvere le eventuali ambiguità convertendo prima alle classi genitore, ed eventualmente in seguito a classi più lontane nella gerarchia di derivazione, nel caso di operatori di conversione definiti dall'utente non si ha la possibilità di specificare una priorità. Chiamate che sarebbero lecite usando i normali puntatori diventano quindi ambigue con gli smart pointers, se si utilizza la soluzione (2).

I metodi (3) e (4) non hanno alcuno dei problemi di (1) e (2), tuttavia sono inerentemente poco sicuri (il metodo (4) non supporta neppure l'ereditarietà multipla, ed il (3) ha alcuni problemi di efficienza, tuttavia qui affronteremo solo il fondamentale problema della sicurezza).

Infatti, poiché le classi degli smart pointers formano una gerarchia di ereditarietà, è possibile convertire puntatori (comuni) a smart pointers tra

loro. In tal modo, possiamo violare il type system, come dimostrato nel **Listato 119**; la violazione può anche essere peggiore, ovvero tra due classi sorelle anziché da base a derivata. Per tale motivo, la soluzione di replicare la gerarchia di derivazione non è normalmente percorribile quando si ricerchi la sicurezza del codice scritto.

Listato 119

```
class Base
  {
  // ...
  } ;


class Derived : public Base
  {
  // ...
  } ;


// smart pointer per Base
class PtrBase
  {
  // ...
  } ;


// smart pointer per Derived
class PtrDerived : public PtrBase
  {
  // ...
  } ;


void f( PtrBase* p1, PtrBase* p2 )
  {
  *p1 = *p2 ;
  }


int main()
  {
  Base b ;
  Derived d ;

  PtrBase pb( &b ) ;
```

```
  PtrDerived pd( &d ) ;

  f( &pd, &pb ) ;
  // assegna ad un PtrDerived un PtrBase !!!

  return( 0 ) ;
  }
```

In conclusione, chi desideri implementare delle classi per gli smart pointers dovrebbe seguire la soluzione (2), che appare la più completa e sicura, anche se in alcuni casi richiederà un cast esplicito per eliminare le ambiguità nella chiamata di funzioni overloaded.

In realtà il problema è più complesso, rispetto a quanto qui illustrato, se si desidera emulare anche la conversione da X* a const X*, da X* a volatile X*, da X* a const volatile X*. Chi fosse interessato a maggiori dettagli può consultare [Ede92]. Per un approccio alternativo agli smart pointers, detto degli *accessors*, vedere [Ken91]; gli accessors soffrono comunque dello stesso problema "di sicurezza" degli smart pointers con gerarchia di derivazione.

9

Funzioni

"Whatever is well conceived is clearly said,
And the words to say it flow with ease."
Nicolas Boileau-Despréaux

In questo capitolo vedremo alcune raccomandazioni generali sulla struttura delle funzioni; in diversi casi, i principi esposti sono di validità generale, e si applicano di conseguenza ad ogni linguaggio; altri sono specifici per il C++, e ad essi è dedicato un maggiore approfondimento.

Le osservazioni riportate di seguito si applicano sia alle funzioni membro che alle funzioni non-membro, se non viene esplicitamente definito un contesto più ristretto.

Mantenere le funzioni "semplici"

Una funzione dovrebbe fare *poche* cose e farle *bene*, ovvero in modo efficiente e privo di errori. Pertanto, funzioni "tuttofare" vanno sempre guardate con sospetto; talvolta sono indispensabili, per pure ragioni di efficienza, ma molto spesso sono semplicemente il risultato di un design affrettato. Una buona regola potrebbe essere: "se non si riesce a trovare un nome ragionevolmente breve, che indichi con sufficiente completezza il compito svolto dalla funzione, probabilmente la funzione è troppo complessa, e se possibile dovrebbe essere partizionata in più funzioni indipendenti". Ricordiamo sempre che funzioni elementari sono facilmente riutilizzabili, mentre blocchi monolitici di codice tendono ad essere troppo specializzati per essere realmente utili in molte situazioni; considerate le funzioni di libreria, che utilizzate così spesso: si tratta in genere di funzioni estremamente elementari, perlomeno dal punto di vista del compito svolto, anche se l'implementazione può essere complessa.
Il concetto di funzione che "svolge un solo compito" è stato formalizzato come *coesione funzionale*, inizialmente in [SMG74]; esistono anche dei tool automatici che misurano la coesione funzionale (per un saggio sui

principi di funzionamento di tali strumenti, vedere [BO94]). In generale, una funzione è coesiva se manipola dati concettualmente "vicini" e fornisce un unico risultato ben determinato: ad esempio, una funzione Print() che stampi un oggetto è coesiva, una funzione PrintMessageAndGetInput() che stampa un prompt e legge interattivamente un dato non è molto coesiva, e potrebbe essere efficacemente partizionata in due funzioni distinte.

Argomenti ed Interfacce

Vista dall'esterno, una funzione è definita dalla sua interfaccia: essa dovrebbe chiarire lo scopo della funzione, anche se letta nel punto di chiamata, e dovrebbe garantire che la funzione operi correttamente su parametri corretti, indipendentemente dal contesto di chiamata. Tuttavia, progettare una buona interfaccia è meno semplice di quanto possa sembrare: di seguito vedremo alcuni interessanti esempi di come un'interfaccia possa essere sensibilmente migliorata, prestando attenzione a semplici ma fondamentali criteri.

Consideriamo una funzione il cui compito è di scrivere un carattere in un file, ed opzionalmente eseguire un flush del buffer di scrittura; una possibile dichiarazione, e la corrispondente chiamata, sono rappresentate nel **Listato 120**:

Listato 120

```
// prototipo
void WriteChar( char c, FILE* file, BOOL flush ) ;

// esempio di chiamata
int main()
  {
   // ...
  WriteChar( 'x', outFile, TRUE ) ;
  // ...
  }
```

Indubbiamente molti programmatori giudicherebbero l'interfaccia di WriteChar perfettamente adeguata allo scopo, forse anche elegante nella sua essenzialità.
Tuttavia, osserviamo meglio la chiamata alla funzione WriteChar: lo scopo della chiamata stessa è realmente chiaro? Supponete di non avere molta confidenza con la funzione WriteChar (ciò è molto probabile se state

leggendo codice scritto da altri programmatori, e se non avete mai usato la funzione WriteChar nel vostro codice); è realmente così immediato comprendere la semantica associata a quel "TRUE"? Evidentemente NO: raramente parametri di tipo booleano sono realmente espressivi nel punto di chiamata. Certamente, esistono casi in cui un parametro booleano è la scelta migliore, ma in genere non lo è -specialmente se il nostro scopo è di scrivere codice comprensibile per gli esseri umani, e non solo per le macchine. L'interfaccia di una funzione incide in modo pesante sulla leggibilità delle chiamate.

Una alternativa di gran lunga migliore, per quanto richieda alcune righe di codice in più, è presentata nel **Listato 121**:

Listato 121

```
enum WriteOption { Flush, DontFlush } ;

// prototipo
void WriteChar(char c, FILE* file, WriteOption option);

// esempio di chiamata
int main()
  {
   // ...
  WriteChar( 'x', outFile, Flush ) ;
  // ...
  }
```

Questa è una chiamata realmente espressiva! Mentre il significato di TRUE nel **Listato 120** era pressoché incomprensibile nel punto di chiamata, il parametro Flush è praticamente autodocumentante. Un ulteriore vantaggio nell'uso di un tipo enumerato: è inerentemente più flessibile di un booleano; volendo aggiungere una ulteriore opzione, ad esempio la possibilità di eseguire il flush di una sola riga, usando gli enum dobbiamo semplicemente estendere il tipo, ed ovviamente gestire la nuova opzione. Se utilizziamo dei booleani, siamo costretti ad aggiungere un nuovo parametro alla funzione, od a ristrutturare il codice per usare un enumerato; in entrambi i casi, tutto il codice esistente che esegua chiamate a WriteChar

va modificato[19]. L'uso degli enumerati sin dall'inizio evita completamente tali problemi di manutenzione.

La tecnica presentata nel **Listato 121** ha un difetto, per quanto minore: lo spazio degli identificatori usati negli enum si può saturare abbastanza rapidamente, specialmente quando gli identificatori usati all'interno del tipo enumerato rappresentano concetti comuni, come on/off, left/right, e così via. Tuttavia, se la funzione WriteChar è una funzione membro di una classe, esiste una alternativa pressoché perfetta, presentata nel **Listato 122**:

Listato 122

```
class File
  {
  public :
    enum Option { Flush, DontFlush } ;
    void WriteChar( char ch, Option opt ) ;
  // ...
  } ;

int main()
  {
  File outFile ;
  outFile.WriteChar( 'x', File :: Flush ) ;
  }
```

Come potete osservare, non solo evitiamo di inserire troppi identificatori nello spazio delle enumerazioni, ed anche di definire troppi tipi enumerati visibili globalmente, ma aggiungiamo ulteriori informazioni al momento della chiamata, informazioni che possono risultare realmente utili a chi legge il nostro codice: *Button :: On* può essere molto più chiaro, in una chiamata di funzione, di un semplice *On*.
Una alternativa, per chi preferisce invece "nascondere" il fatto che Flush sia definito internamente alla classe File, è l'utilizzo dei *namespace* e della direttiva *using*.

[19]non è rarissimo trovare codice C dove viene sfruttato il range "fisico" di un booleano definito con un typedef, assegnandovi ad esempio 2 o 3, proprio perché in fase di manutenzione il range [TRUE, FALSE] si era rivelato troppo stretto.

Raccomandazione 101

Evitate l'uso di parametri booleani; definite invece un apposito tipo enumerato per le diverse opzioni. Il tipo può essere definito internamente ad una classe per evitare la saturazione dello spazio degli enumerati

Consideriamo ora un'altra funzione molto semplice, la relativa interfaccia, ed un contesto di chiamata: **Listato 123**.

Listato 123

```
// converte il numero nella sua rappresentazione
// esadecimale
const char* Hex( int n )
  {
  static char buffer[ sizeof( int ) + 1  ] ;
  //... converte il numero
  return( buffer ) ;
  }

int main()
  {
  const char* h1 = Hex( 10 ) ;
  // ...
  const char* h2 = Hex( 20 ) ;

  cout << h1 << "  " << h2 ;

  return( 0 ) ;
  }
```

La funzione Hex sembra nuovamente abbastanza ben studiata: usa un buffer statico interno, di grandezza sufficiente, e restituisce un puntatore const al primo elemento del buffer, che non può quindi essere modificato dall'esterno. La semplicità della funzione stessa è tale che il prototipo della funzione è estremamente semplice, e la sua chiamata è cristallina per chi legge il codice. Tuttavia il breve programma del **Listato 123** *non* funziona, o meglio non stampa la rappresentazione esadecimale di 10 e 20, ma stampa due volte quella di 20; la ragione, ovviamente, è che la seconda chiamata sovrascrive il nuovo risultato al vecchio nell'area condivisa. Al di là dell'esempio molto semplice, dove trovare la ragione del problema è quindi banale, se una funzione *f* restituisce un puntatore ad un buffer statico, è compito del chiamante garantire che il risultato non venga riutilizzato dopo una successiva chiamata alla funzione, o ad un'altra

funzione che a sua volta chiami *f*. Ciò richiede una conoscenza troppo dettagliata dell'implementazione di *f* e delle altre funzioni chiamate, per essere effettivamente accettabile in progetti reali. Notiamo che lo stesso problema si applica se la funzione restituisce l'indirizzo di un oggetto globale, anziché locale e statico.

Raccomandazione 102

Non restituite puntatori ad elementi statici o globali come risultato delle funzioni.

Una possibile alternativa al **Listato 123** è presentata nel **Listato 124**:

Listato 124

```
// converte il numero nella sua rappresentazione
// esadecimale
char* Hex( int n )
  {
  char* buffer = new char[ sizeof( int ) + 1  ] ;
  //... converte il numero
  return( buffer ) ;
  }

int main()
  {
  char* h1 = Hex( 10 ) ;
  // ...
  char* h2 = Hex( 20 ) ;

  cout << h1 << "  " << h2 ;

  return( 0 ) ;
  }
```

La differenza rispetto al precedente è che nuova memoria viene allocata ogni volta che la funzione viene chiamata, quindi non viene sovrascritta nessuna area globale o statica. In questo senso, il programma del **Listato 124** funziona, salvo che ho volutamente "dimenticato" di deallocare la memoria all'interno di main(). Nuovamente, anche se in un piccolo frammento di codice come l'esempio riportato la verifica della correttezza nell'allocazione/deallocazione della memoria è banale, su programmi di grandi dimensioni, dove i puntatori vengono spesso propagati tra diverse funzioni, la gestione della memoria dinamica è un problema molto sentito,

poiché la perfezione sta esattamente tra il memory leakage (quando non rilasciamo tutta la memoria) e le cancellazioni multiple, senza contare i pericoli dei dangling pointers. Nuovamente, quindi, l'interfaccia proposta per *Hex* è troppo debole.

Raccomandazione 103

Non restituite, come risultati di funzione, puntatori a memoria allocata all'interno della funzione, e che debba essere deallocata dal chiamante.

Una soluzione completa ed esente da problemi è visibile nel **Listato 125**:

Listato 125

```
const size_t HEX_SIZE = sizeof( int ) + 1 ;

// converte il numero nella sua rappresentazione
// esadecimale
void ConvertToHex( int n, char* buffer )
  {
  //... converte il numero
  }

int main()
  {
  char h1[ HEX_SIZE ]  ;
  ConvertToHex( 10, h1 ) ;
  // ...
  char h2[ HEX_SIZE ]  ;
  ConvertToHex( 20, h2 ) ;

  cout << h1 << "  " << h2 ;

  return( 0 ) ;
  }
```

Notiamo che esiste una apposita costante che contiene la dimensione richiesta per il buffer da passare alla funzione: l'allocazione e la deallocazione avviene nel punto di chiamata, ed è quindi localizzata e più semplice da gestire, tantopiù che si possono usare variabili automatiche come nell'esempio dato, o riutilizzare sotto la propria responsabilità la stessa area per chiamate multiple.
Ha un difetto residuo, ovvero non consente lo stile "dichiara e inizializza" per la variabile, e si deve rinunciare ad una visione più "funzionale" in

favore di una più “procedurale” (notiamo che anche il nome della funzione è stato cambiato, poiché ha ora una struttura di procedura). Inoltre se il chiamante commette un errore nell’allocare il buffer, possono esserci seri problemi a run-time.

Esiste quindi una “soluzione perfetta”? In effetti, considerando le varie proposte di cui sopra, possiamo vedere che esse sono praticamente scritte in C, senza sfruttare alcuno dei vantaggi del C++. La soluzione “perfetta” è invece di disporre di una classe *String* efficiente, ad esempio con un’implementazione copy-on-write, e strutturare la funzione come nel **Listato 126**:

Listato 126

```
String Hex( int n )
  {
  char buffer[ sizeof( int ) + 1  ] ;
  //... converte il numero
  return( String( buffer ) ) ;
  }

int main()
  {
  String h1 = Hex( 10 ) ;
  // ...
  String h2 = Hex( 20 ) ;

  cout << h1 << “  “ << h2 ;

  return( 0 ) ;
  }
```

Notiamo che non si hanno problemi di sovrapposizione o di mancato rilascio della memoria. Se necessario, la funzione *Hex* potrebbe essere resa più efficiente, creando sin dall’inizio con un apposito costruttore un oggetto String con un buffer interno di grandezza adeguata, e modificandone il contenuto.
Naturalmente, la tecnica su esposta è estendibile ad ogni tipo di dato, creando delle opportune classi wrapper, ed è probabilmente la tecnica migliore per restituire dati allocati dinamicamente.

Raccomandazione 104

Se una funzione deve restituire dati creati dinamicamente, considerate l'opportunità di creare una classe wrapper che gestisca l'allocazione/deallocazione dei dati.

Vediamo ora un ulteriore esempio di interfaccia debole, preso dall'API di un ambiente operativo molto diffuso: **Listato 127**.

Listato 127

```
// esegue il programma specificato in lpszCmdLine;
fuCmdShow
// stabilisce come deve essere visualizzata la finestra
// della applicazione.

// Il valore di ritorno identifica l'istanza
// dell'applicazione se la funzione ha successo,
// altrimenti è un codice di errore < 32

UINT WinExec( LPCSTR lpszCmdLine, UINT fuCmdShow ) ;
```

Tralasciando la leggibilità o meno della notazione hungarian, ed il fatto che "istanza dell'applicazione" è molto ambiguo nel contesto dell'API in questione, il vero problema è che il risultato della funzione può essere sia un valore lecito che un codice di errore. Questa pratica non è del tutto insolita in C: basti pensare a quante funzioni operanti su stringhe ritornano NULL per segnalare un errore o una terminazione; del resto, il C è stato a lungo criticato per un problema in parte simile, ovvero la memorizzazione del meta-dato '0' nello spazio dei dati, per rappresentare la terminazione di una stringa.

Ovviamente, il fatto che sia una pratica abbastanza in uso, e tutto sommato abbastanza comoda per chi scrive la funzione, non significa che sia una buona tecnica di programmazione; in effetti, mi è capitato molte volte di vedere codice che *non* controllava il valore di ritorno in casi simili, ed assumeva semplicemente che fosse corretto.
Il problema dell'interfaccia è che la sua stessa forma non incoraggia il programmatore a verificare eventuali errori, poiché non vi è la presenza esplicita di un "contenitore" per gli errori stessi; non a caso già in [DC85] si invitavano i programmatori ad un comportamento più responsabile verso i codici di errore. Mentre ciò è accettabile in caso di funzioni molto semplici, ad esempio la *strstr* che ritorna un puntatore ad una sottostringa o NULL se

tale sottostringa non esiste, se ci troviamo di fronte a funzioni che possono fallire in diversi modi, e per le quali è necessario un error handling più completo, è molto meglio richiedere un ulteriore parametro passato per riferimento, che conterrà il codice di errore, o l'uso delle eccezioni se di eccezioni si tratta.

In questo senso, possiamo vedere la differenza concettuale tra *strstr* e *WinExec*: mentre nella prima il risultato NULL, ovvero "sottostringa non presente", fa parte dei valori attesi in un funzionamento standard del programma (basti pensare alla ricerca/sostituzione di tutte le occorrenze di una sottostringa, operazione che dovrà comunque terminare quando non vi saranno più occorrenze), nella seconda un codice di errore rappresenta una vera e propria eccezione a run-time, come il tentativo di eseguire un programma non presente su disco.

Infine, notiamo che nel caso di *strstr* il valore di ritorno è comunque monomorfo, essendo NULL un puntatore spesso utilizzato per rappresentare la stringa vuota, mentre nel caso di *WinExec* è stato sfruttato un tipo scalare per memorizzare valori eterogenei, ovvero un codice di errore ed una non meglio precisata "istanza dell'applicazione" (in realtà, nel caso concreto, un handle per l'istanza dell'applicazione).
In un ambiente con un controllo sui tipi più stretto, e con interfacce studiate per essere robuste, una simile commistione sarebbe resa impossibile dal type checking del compilatore. Tra l'altro, alcuni dei 32 valori di ritorno sono "riservati" per futuri codici di errore, problema abbastanza tipico quando si utilizza un sottoinsieme dei valori ammissibili per rappresentare errori a run-time; se in futuro 32 valori non fossero sufficienti, ogni applicazione esistente dovrebbe essere modificata o ricompilata, viceversa un codice di errore verrebbe interpretato come una "instanza dell'applicazione" valida.

Raccomandazione 105

Non restituite valori eterogenei come risultato delle vostre funzioni; se necessario, utilizzate un ulteriore parametro per codici di errore, o generate una eccezione a run-time.

Un'ultima osservazione riguarda le funzioni con numero variabile di parametri, come la ben nota *printf*. Tali funzioni hanno molti difetti, il principale dei quali consiste nella mancanza di type checking (per ulteriori considerazioni, vedere il capitolo 13); poiché il type checking statico

rappresenta uno dei migliori supporti per una corretta programmazione, è sempre preferibile utilizzare tecniche che non scavalchino il type checking stesso.
In effetti, esistono ben poche ragioni per definire funzioni con ellipsis (...) in C++: meccanismi come l'overloading ed i parametri di default consentono di coprire i casi con un numero variabile di parametri, ma dove sia possibile stabilire in anticipo i *pattern* di chiamata (ad es., f(int), f(float), f(int, float) e così via).
Nel caso di funzioni in cui sia il tipo che il numero di parametri siano totalmente variabili, una buona soluzione è utilizzare un meccanismo simile a quello utilizzato per gli stream di I/O, che di fatto sostituiscono la succitata *printf*. Anziché avere una singola funzione in grado di accettare parametri qualunque, ed in numero variabile, definiamo una classe con un operatore overloaded, e definiamo una versione della funzione o dell'operatore per ogni singolo caso specifico; l'operatore restituirà un reference all'oggetto su cui è stato chiamato, consentendo così una concatenazione di chiamate.
L'effetto combinato dell'overloading e della concatenazione realizza di fatto un meccanismo del tutto simile a quello ottenibile con l'ellipsis, garantendo nello stesso tempo il type checking statico.

Asserzioni e programmazione difensiva

Una caratteristica irrinunciabile delle buone routine è certamente la correttezza: dato un input valido, devono restituire un output valido, in accordo con la specifica delle funzioni. Cosa possiamo dire a proposito di input non validi? Una vecchia legge della programmazione amatoriale recita "garbage in, garbage out", invitando a trascurare il comportamento delle routine su input non validi, o richieste di servizi che portano gli oggetti in stati non validi.
Consideriamo il **Listato 128**, che implementa un semplice stack di interi; ogni routine opera correttamente, se rispettiamo alcune condizioni al contorno, ovvero non dobbiamo mai inserire più di 100 elementi nè eseguire *Top*() o *Pop*() sullo stack vuoto. Tali eventi comporterebbero l'overrun di zone di memoria che non appartengono all'array *stack*, con conseguenze che potrebbero non manifestarsi immediatamente, ma che quasi certamente comporteranno dei malfunzionamenti del programma.

Listato 128

```
class Stack
  {
```

```
  public :
    Stack() ;
    void Push( int x ) ;
    void Pop() ;
    int Top() ;
  private :
    int stack[ 100 ] ;
    int top ;
  } ;

Stack :: Stack()
  {
  top = 0 ;
  }

void Stack :: Push( int x )
  {
  stack[ top++ ] = x ;
  }

void Stack :: Pop()
  {
  top-- ;
  }

int Stack :: Top()
  {
  return( stack[ top - 1 ] ) ;
  }
```

La domanda diventa quindi: “possiamo accettare che una routine chiamata con parametri errati o a partire da uno stato errato crei problemi non immediatamente verificabili?” e la risposta è ovviamente NO. In effetti, esiste una diffusa mentalità secondo la quale il “garbage in, garbage out” è pressoché inevitabile, se si vogliono avere buone prestazioni; ciò deriva spesso da un approccio semplicistico al problema degli errori, come quello del **Listato 129**.

Listato 129

```
class Stack
  {
  public :
    Stack() ;
    void Push( int x ) ;
    void Pop() ;
```

```
    int Top() ;
  private :
    int stack[ 100 ] ;
    int top ;
  } ;

Stack :: Stack()
  {
  top = 0 ;
  }

void Stack :: Push( int x )
  {
  if( top < 100 )
    stack[ top++ ] = x ;
  }

void Stack :: Pop()
  {
  if( top > 0 )
    top-- ;
  }

int Stack :: Top()
  {
  if( top > 0 )
    return( stack[ top - 1 ] ) ;
  else
    return( 0 ) ;
  }
```

Come vedete, in questa versione l'ipotetico programmatore si è "protetto" da tutte le situazioni che potevano generare un errore; ciò è stato ottenuto al prezzo di un leggero overhead (in senso assoluto: in senso relativo è molto alto), nonché esercitando un certo arbitrio nella funzione *Top*().
In senso assoluto, la seconda versione è migliore della prima, tuttavia è ancora lontana dalla soluzione ideale: le situazioni di errore all'interno di *Stack* sono in realtà degli errori nel punto di chiamata. Il codice chiamante è quindi errato, ma il programmatore che utilizza la classe *Stack* non ha modo di sapere che sta commettendo un errore, se non tracciando l'esecuzione all'interno di un debugger. Se invece il codice chiamante è esente da errori (almeno connessi all'uso di *Stack*) il codice "difensivo" rappresenta soltanto un overhead ingiustificato.

La soluzione migliore è di introdurre all'interno del codice, ovunque sia necessaria la verifica di una precondizione, ma anche di postcondizioni o di condizioni invarianti, una apposita asserzione, che generi un messaggio di errore se la condizione non è verificata. L'asserzione stessa deve essere rimovibile tramite compilazione condizionale: in tal modo, possiamo utilizzare la versione "con controlli" durante lo sviluppo, e la versione "senza controlli" e quindi senza overhead, per la release finale.

Un esempio di utilizzo corretto delle asserzioni è dato nel **Listato 130**; vedremo più avanti come definire opportunamente la macro ASSERT: per ora, possiamo pensare che generi un opportuno messaggio, indicando anche il file e l'esatta linea ove la condizione è stata violata.

Listato 130

```
class Stack
  {
  public :
    Stack() ;
    void Push( int x ) ;
    void Pop() ;
    int Top() ;
  private :
    int stack[ 100 ] ;
    int top ;
  } ;


Stack :: Stack()
  {
  top = 0 ;
  }


void Stack :: Push( int x )
  {
  ASSERT( top < 100 ) ;

  stack[ top++ ] = x ;
  }


void Stack :: Pop()
  {
  ASSERT( top > 0 ) ;
```

```
  top-- ;
  }


int Stack :: Top()
  {
  ASSERT( top > 0 ) ;

  return( stack[ top - 1 ] ) ;
  }
```

Come è semplice capire, in caso di errore nel codice chiamante riceveremo ora un chiaro messaggio che una condizione fondamentale per il corretto funzionamento di *Stack* è stata violata; sarà quindi necessario rivedere il codice della routine chiamante per eliminare la fonte dell'errore.
Attenzione: molti programmatori sono tentati di rimuovere l'asserzione se essa non è chiara; è una classica manifestazione di sfiducia nel lavoro altrui, o di eccessiva fiducia nel proprio, tuttavia se la condizione che state verificando non è di immediata comprensione, commentatela adeguatamente. In tal modo chi utilizza le vostre classi si convincerà più facilmente che l'errore non è nella vostra routine, ma da qualche parte nel codice chiamante.

Una macro di *assert* è normalmente disponibile per i vari compilatori, nell'header <assert.h>; nell'esempio precedente, ho usato una ASSERT completamente in maiuscole, perché in genere vi sono dei buoni motivi per non utilizzare la macro fornita dal compilatore. Essa infatti genera un messaggio di errore comprendente non solo il file e il numero di linea, ma anche la linea stessa; su programmi di grandi dimensioni, ciò comporta la definizione di un numero enorme di stringhe, anche di lunghezza notevole, che in determinati sistemi operativi può facilmente superare le dimensioni ammissibili per lo spazio dei dati costanti. D'altra parte, avendo il file ed il numero di riga, ogni altra informazione è ridondante, per cui si può spesso definire una macro più snella. Un ulteriore motivo è dato dalla diffusione di ambienti a finestre, che richiedono meccanismi di output ben diversi dal semplice *fprintf* su *stderr*; in tal caso, creare una propria macro di ASSERT è indispensabile. Nel **Listato 131** potete vedere una delle tante implementazioni possibili per ASSERT: in questo caso, se è definito il simbolo _DEBUG_ verrà incluso il codice di verifica, viceversa verrà escluso dalla compilazione. L'inclusione o meno del codice di debugging può essere realizzata a livello del singolo file (definendo o meno

DEBUG al suo interno) o più comunemente a livello di progetto, con un apposito switch di compilazione.

Listato 131

```
/* ASSERT.H */


#ifndef ASSERT_

#define ASSERT_


void Assert( char* file, LONG line ) ;


#ifdef _DEBUG_
  #define ASSERT( c )                          \
    if( !(c) )                                 \
      Assert( __FILE__, __LINE__ )
#else
  #define ASSERT( c ) NULL
#endif


#endif // ifndef ASSERT_


/* ASSERT.CPP */

void Assert( char* file, LONG line )
  {
  // stampa il messaggio, eventualmente chiede se
  // si vuole terminare l'esecuzione

  // dipendente dal sistema in uso
  }
```

Notiamo che la presenza di asserzioni nel codice è un buon aiuto per la comprensione del codice; talvolta, per renderlo ancora più chiaro, vengono definite tre macro, tutte associate allo stesso codice di ASSERT, che verificano rispettivamente:

1. Precondizioni, come il rispetto del range valido per i parametri, o lo stato iniziale; nel precedente esempio dello stack, tutte le asserzioni

sono precondizioni. Le precondizioni sono normalmente utilizzate per verificare che il chiamante utilizzi le routine nel modo corretto.

2. Postcondizioni: normalmente utilizzate per verificare che il nostro codice sia corretto, ovvero che il risultato rispetti la specifica. Ciò viene normalmente ottenuto verificando con un algoritmo alternativo (nel caso di operazioni fondamentali per l'applicazione) ma talvolta anche verificando solo alcune condizioni necessarie, ma non sufficienti (vedere oltre).

3. Invarianti: utilizzate nuovamente per verificare la correttezza del nostro codice, sono abitualmente inserite all'interno di loop, e verificano che alcune condizioni si mantengano valide all'interno del loop stesso.

Una ulteriore, utile modifica alla macro ASSERT potrebbe essere la gestione di una versione di debug, che come sopra stampa un messaggio ed eventualmente ferma il programma, di una versione beta (definendo BETA_ anziché _DEBUG_) che invece genera silenziosamente un log su file degli eventuali errori, ed ovviamente una versione senza controlli. La versione beta potrebbe essere utilizzata appunto in fase di beta-testing, richiedendo ai tester di inviare periodicamente il file di log degli errori.

È importante riprendere alcune norme fondamentali sull'uso delle asserzioni: esse vanno usate *solo* per verificare che non siano presenti errori di programmazione, non per gestire errori che potrebbero avvenire a run-time nella release finale. Usare una asserzione per verificare che non si ecceda la capacità di uno stack è un caso di buon utilizzo se lo stack è documentato come un fixed-size stack, utilizzarla per verificare che l'apertura di un file sia andata a buon fine non lo è, in quanto il secondo evento potrebbe facilmente verificarsi anche se il programma è corretto, dipendendo largamente dal sistema che fa girare il programma. Usare le asserzioni per verificare l'input dell'utente è parimenti errato.
Inoltre, ricordate sempre che il codice presente nell'asserzione *non* fa parte del programma (infatti verrà rimosso dalla compilazione nella versione finale): pertanto, di norma non dovrà generare side-effect ed il restante codice dovrà essere compilabile senza problemi sia in presenza che in assenza di esso.
Ovviamente, anche il codice scritto come parte di una asserzione può contenere errori, oppure la condizione posta può essere troppo restrittiva o troppo blanda; è quindi importante, nei casi non banali, commentare in modo dettagliato l'idea alla base dell'asserzione, in modo che questa possa essere eventualmente verificata anche a livello logico.

Infine, talvolta le asserzioni (specialmente le postcondizioni) non possono verificare una proprietà necessaria e sufficiente per la correttezza, ma solo una proprietà necessaria. Un caso emblematico è un algoritmo di sort: se volessimo verificarne la correttezza tramite una asserzione, dovremmo verificare che gli elementi siano effettivamente in ordine (semplice, complessità lineare) e che siano una permutazione degli elementi di partenza (più difficile, complessità $O(n \log(n))$. In casi simili, ci si può accontentare di verifiche meno restrittive, ad esempio verificando (nel caso in esame) che la somma sia la stessa, anziché verificare che siano una permutazione [SC94]. Esistono anche delle interessanti proposte per aggiungere un supporto più potente ed espressivo ai linguaggi di programmazione, direttamente o tramite tool esterni, che permettano un uso più efficace delle asserzioni; il lettore interessato può trovare in [CL90], [Cli90], e [Ros94] degli ottimi spunti, sia riferiti al C++ che ad altri linguaggi.

Raccomandazione 106

Inserite asserzioni nel codice per verificare precondizioni, postcondizioni ed invarianti. Le asserzioni vanno utilizzate per trovare errori di codifica, non per gestire errori run-time dovuti all'interazione con l'ambiente.

Raccomandazione 107

Non inserite codice che genera side-effect all'interno delle asserzioni: tale codice non verrà compilato come parte della release finale.

Raccomandazione 108

Se la condizione verificata all'interno di una asserzione non è di immediata comprensione, commentatela adeguatamente.

Lunghezza delle funzioni

Tempo fa, dovendo assumere un nuovo programmatore, ho avuto modo di incontrare e discutere con molti sviluppatori aventi una discreta esperienza, in quanto la posizione da ricoprire richiedeva una conoscenza piuttosto approfondita di diversi argomenti, tra cui il C++. Uno di essi, che in effetti poteva vantare l'esperienza di programmazione più lunga (ma non nel

paradigma object oriented), durante il colloquio affermò di avere l'abitudine di scrivere funzioni con un body "piuttosto lungo". Alla mia richiesta di chiarimenti, disse che considerava una lunghezza di duemila linee *non* eccessiva per una funzione, e si offrì di mostrarmi su un computer portatile un esempio di codice, sviluppato in ambito di controllo industriale, in cui vi era appunto un esempio di funzione con oltre duemila righe. Tale funzione eseguiva una molteplicità di compiti, implementando di fatto un algoritmo piuttosto complesso; l'algoritmo era tuttavia praticamente irriconoscibile, in quanto mancava una qualunque visione astratta delle operazioni. Parti fondamentali, eventualmente riutilizzabili, non erano state identificate ed astratte in funzioni separate: tutto il codice era stato inglobato in un blocco monolitico. Alla mia richiesta delle ragioni alla base di una tale struttura, il programmatore spiegò che "il flusso logico era comunque chiaro" e che "non vi era una reale necessità di spezzarla".

Eventi simili sono, nella mia esperienza, piuttosto rari; già funzioni lunghe più di un centinaio di righe sono considerate anomale tra gli sviluppatori professionisti. Perche? Vi sono molte ragioni, sulle quali vale la pena di soffermarsi:

- una funzione troppo lunga è difficile da leggere, comprendere e ricordare nella sua interezza. Gli esseri umani hanno limiti ben precisi sul grado di dettaglio che sono in grado di assimilare [Kla80].

- all'interno di una funzione troppo lunga è più difficile gestire gli errori, specialmente nel caso vi siano oggetti allocati dinamicamente. Deallocare opportunamente le risorse allocate centinaia (o anche solo decine) di righe prima può essere difficile, ed è invece semplice dimenticarsi di farlo.

- una funzione troppo lunga è difficile da modificare, sia perché è difficile da capire, sia perché è probabile introdurre degli errori, ad esempio modificando il valore di una variabile che viene riutilizzata in seguito.

- una funzione troppo lunga rappresenta spesso un'occasione di riutilizzo sprecata. Spesso all'interno di una funzione complessa si eseguono compiti più semplici che potrebbero essere di utilità comune, se venissero astratti in funzioni separate.

- una funzione molto lunga può essere molto difficile da testare, in quanto non è possibile fare un test di unità su sotto-funzioni ed in

seguito un test di integrazione nei punti di chiamata, ma ogni possibile percorso va testato all'interno di un blocco monolitico.

Esistono probabilmente altre ragioni per contenere la lunghezza delle funzioni; vi sono invece pochi dati oggettivi su quale sia una "lunghezza massima ideale" per le funzioni stesse. La lunghezza, peraltro, non è di per sé un indice sufficiente per la complessità: un criterio migliore potrebbe essere ad esempio la misura di complessità ciclomatica di McCabe [McC76]; non a caso, tuttavia, funzioni lunghe hanno in genere una complessità ciclomatica piuttosto alta, segno di bassa qualità.
Dovendo fissare dei limiti, potremmo dire che di norma una funzione di oltre 100-150 linee va considerata molto sospetta, ed una "buona" funzione non dovrebbe (comunemente) superare la cinquantina di righe. Se esistono degli ottimi motivi per scrivere una funzione molto lunga, essi sono anche degli ottimi candidati per un commento che li renda noti. Alcuni studi di comprensibilità del codice [CDS86], [LV89], [Jon86], [SB91] dimostrano che non vi è un reale guadagno a rendere le funzioni estremamente corte (es. 10 linee), e che in molti casi codice con funzioni di lunghezza sul centinaio di righe è perfettamente comprensibile e mantenibile (una dimensione "ideale" viene indicata intorno alle 25 linee). Gli stessi studi provano invece che oltre le 200 linee cominciano problemi di comprensione, ed in funzioni con oltre 500 linee si annidano probabilmente i bug più complessi da trovare ed eliminare.

Raccomandazione 109

Evitate di scrivere funzioni troppo lunghe; se vi sono importanti ragioni per non suddividere una funzione lunga in sottofunzioni, documentate le motivazioni con un opportuno commento.

Funzioni Inline

Le funzioni inline rappresentano uno degli strumenti di cui il programmatore C++ dispone per esercitare un controllo sulle prestazioni del codice generato. Capacità simili sono di rado fornite dai linguaggi di alto livello, dove il programmatore è isolato dalla generazione del codice, compito esclusivo del compilatore; il C++ rappresenta in tal senso un'eccezione, motivata dall'ampio spettro di applicazioni a cui si rivolge.

Avere strumenti potenti a disposizione significa però che è necessaria la giusta cautela nel loro utilizzo: usate con accortezza, le funzioni inline sono

un ausilio prezioso per la scrittura di programmi efficienti ed eleganti; il loro abuso può facilmente portare a codice più complesso da mantenere, con tempi di compilazione abnormi, ed in casi patologici ad un aumento delle dimensioni del codice tale da degradarne le prestazioni su sistemi a memoria virtuale.

I migliori candidati per l'espansione inline sono le funzioni così semplici che l'overhead della chiamata rappresenta una percentuale sensibile del loro tempo di esecuzione: un esempio classico sono le funzioni che eseguono il forwarding di una chiamata ad un sotto-componentente, funzioni che diventano tutt'altro che rare se si cerca di seguire la legge di Demeter (capitolo 7).
Ovviamente, una funzione può apparire semplice e non esserlo affatto in termini computazionali: se la nostra funzione ne richiama semplicemente un'altra, che tuttavia esegue un intenso I/O su disco, dichiararla inline non comporterà alcun vantaggio sensibile.

Viceversa, definire una funzione come inline significa esporne l'implementazione in un header file, e ciò comporta anche tempi di ricompilazione più lunghi in caso di modifica: tutti i file che includono l'header, quindi tutti i file che utilizzano la funzione inline o altre funzioni/classi definite nello stesso header file, dovranno essere ricompilate.

Funzioni "lunghe" non dovrebbero essere espanse inline, poiché normalmente l'overhead di chiamata è trascurabile: considerate anche che, oltre agli effetti negativi sui tempi di compilazione, un eccesso di espansione in linea del codice può portare a significativi aumenti delle dimensioni complessive del programma. Considerando che le architetture moderne degli elaboratori, basate su livelli di cache multipli, sono ottimizzate per programmi ad elevata località, un eccesso di espansione in linea potrebbe ottenere l'effetto opposto a quello desiderato, ovvero un peggioramento delle prestazioni; si tratta comunque di casi patologici, tutt'altro che normali.

Le funzioni inline vanno comunque preferite alle macro (che comunque condividono gli stessi problemi di ricompilazione ed esplosione del codice) in quanto inerentemente più sicure. Le argomentazioni sono in gran parte simili a quelle che portano a preferire le costanti alle macro costanti, ovvero:

1. Poiché le macro vengono trattate dal prepocessore prima della compilazione, errori nella definizione della macro possono essere difficili da individuare e riconoscere come tali durante la compilazione; considerando il **Listato 132**:

Listato 132

```
#define Square( x ) (x)*(x) ;


float Area( float radius )
  {
  const float PI = 3.14159 ;

  return( Square( radius ) * PI ) ;
  }
```

Il compilatore segnalerà un errore nella linea

return(Square(radius) * PI) ;

Il messaggio è dipendente dal compilatore stesso; un compilatore piuttosto diffuso genererà il seguente output:

Error test.cpp 9:) expected in function Area(float)
Warning test.cpp 9: Unreachable code in function Area(float)
Error test.cpp 9: Illegal use of floating point in function Area(float)
Error test.cpp 9: Statement missing ; in function Area(float)
Warning test.cpp 10: Function should return a value in function Area(float)

Ovviamente, l'errore è invece sulla linea del #define, dove è stato aggiunto un ';' alla fine della riga stessa.

2. le macro-funzioni definite con #define non sono disponibili per il trace a passo singolo in fase di debugging; in molti casi, invece, il compilatore supporta opportuni switch di compilazione per evitare l'espansione inline se necessario, consentendo così un più semplice debugging.

le macro-funzioni definite con #define sono ben note per il problema associato ai side effect: nel **Listato 133**, la chiamata di una macro-funzione con un argomento che genera un side-effect, comporta in realtà l'esecuzione del codice relativo al side-effect per due volte.

Listato 133

```
#define Square( x ) (x)*(x) ;


int f()
  {
  x = 1 ;
  y = Square( x++ ) ; // x viene incrementato 2 volte!
  return( x + y ) ;
  }
```

Possiamo quindi condensare quanto sopra in alcune raccomandazioni:

Raccomandazione 110

Utilizzare le funzioni inline anziché definire macro-funzioni con #define.

Raccomandazione 111

Utilizzate le funzioni inline solo se vi sono concreti motivi: il fatto che una funzione sia breve non giustifica di per sé l'espansione in linea.

Raccomandazione 112

Utilizzare le funzioni inline solo quando l'overhead di chiamata è significativo rispetto al tempo di esecuzione del corpo della funzione stessa. Funzioni di forwarding o funzioni protected di accesso ai membri dato sono normalmente buoni candidati all'espansione in linea.

Overloading

In una critica (a tratti un po' aspra) del C++ [Joy92], l'autore afferma che il supporto per l'overloading delle funzioni è una caratteristica negativa in un linguaggio di programmazione, in quanto "cose diverse devono avere nomi diversi". L'argomentazione è senza dubbio corretta, tuttavia la conclusione di bandire l'overloading è troppo radicale: invece di utilizzare una riflessione in senso negativo, possiamo utilmente trasformarla in una occasione costruttiva per migliorare la qualità del codice (ciò accade

spesso, ed in effetti molte delle idee in questo libro sono emerse proprio ascoltando con attenzione le critiche ed i commenti più severi dei detrattori del C++).

Letta in senso positivo, l'affermazione precedente diventa "tutte le versioni overloaded della stessa funzione dovrebbero fare la stessa cosa"; questo riflette l'esperienza comune nel caso più tipico di overloading: le classi con più costruttori, il cui risultato è sempre quello di costruire un oggetto della classe. Un altro esempio abbastanza comune è la ricerca o la memorizzazione in un database di oggetti eterogenei, per ognuno dei quali esiste un funzione overloaded che lo accetta come parametro.

Ricordate che l'overloading non è permesso tra classi base e sottoclassi: se una classe derivata dichiara una versione overloaded, ma diversa, di una funzione presente nella classe base, le corrispondenti funzioni della classe base verranno nascoste (alcuni compilatori emettono un warning in tal senso). Ciò corrisponde ad una precisa scelta progettuale del C++ [Str94], ovvero il rispetto delle regole di scope così come avviene per le variabili; ho avuto comunque modo di notare che molti programmatori lo considerano un difetto del linguaggio, ed in effetti in alcune occasioni si può rivelare fastidioso, soprattutto per chi non ne è a conoscenza. Il **Listato 134** mostra un esempio del problema: nonostante *f()* sia una funzione pubblica della classe base, essa non è richiamabile su un oggetto di classe derivata poiché è nascosta dalla versione overloaded.

Listato 134

```
class Base
  {
  public :
    void f() {}
  } ;


class Derived : public Base
  {
  public :
    void f( int x ) {}
  } ;


int main()
  {
  Derived d ;
```

```
  d.f( 1 ) ;    // OK
  d.f() ;       // NO: nascosta

  return( 0 ) ;
  }
```

L'unica soluzione, in simili casi, è di fornire una funzione di forwarding in *Derived* che chiami la corrispondente funzione in *Base*. Ciò non viola la raccomandazione di non ridefinire funzioni non virtuali della classe base, in quanto si tratta di un semplice forwarding, per di più inteso a scavalcare una particolarità abbastanza discutibile del linguaggio. Il codice in questione potrebbe comunque essere un buon candidato per un commento.

Raccomandazione 113

Tutte le versioni overloaded di una stessa funzione dovrebbero avere la stessa semantica, perlomeno ad un alto ma riconoscibile livello di astrazione.

Parametri di Default

La possibilità di avere parametri di default nelle funzioni è un'altra delle caratteristiche abbastanza criticate del C++: da una parte essa è infatti ridondante, potendo essere sostituita in larga misura dall'overloading o definendo apposite costanti, dall'altra si presenta poco chiara nel punto di lettura, dove semplicemente i parametri che devono assumere il valore di default non appaiono[20].

In effetti, esistono sostanzialmente due modi di usare i parametri di default: il primo, com'è ovvio, li utilizza per fornire dei valori validi per la maggior parte delle chiamate, e lo riprenderemo più avanti. Il secondo, utilizza i parametri di default come una forma di "overloading per programmatori pigri", e dovrebbe essere evitato. Un esempio della seconda tecnica è mostrato nel **Listato 135**: una funzione Put che esegue o meno il flush del buffer in funzione del valore del secondo parametro, cui è dato un valore di default conveniente per "l'azione più comune".

[20]Nel linguaggio Ada, per rendere più chiara la chiamata di funzioni con parametri di default, è possibile nominare esplicitamente i parametri formali nel punto di chiamata.

Listato 135

```
class IntStream
  {
  public :
    void Put( int x, Bool flush = FALSE ) ;
    // ...
  } ;
```

Non è raro vedere applicata questa tecnica, il che ha portato alla giusta critica sulla ridondanza dei parametri di default: nel caso in questione, una funzione overloaded sarebbe stata molto più chiara; a dire il vero, poiché la funzione fa due cose diverse, a seconda del parametro, sarebbe molto meglio scrivere due funzioni separate e dar loro due nomi distinti. Per esperienza, ho notato che non di rado si ricorre allo stratagemma del "flag di default" in fase di manutenzione, per modificare una funzione poco flessibile, mantenendo comunque la compatibilità con il codice esistente. Mentre in situazioni "di emergenza" questa tecnica può essere valida, un suo uso sistematico porta indubbiamente a codice molto difficile da leggere. Noterete comunque che eliminando l'uso dei parametri default per simulare l'overloading, l'utilizzo dei parametri di default stessi verrà ridotto in modo consistente.

Raccomandazione 114

Non utilizzare i parametri di default per simulare l'overloading in una sola funzione: una funzione dovrebbe svolgere un unico compito.

Riguardo l'uso più canonico dei valori di default, esso dovrebbe essere limitato ai casi in cui il valore di default è realmente significativo nella maggior parte delle chiamate. Ricordate che avere un default su un parametro costringe a fornire default per tutti i parametri seguenti al momento della dichiarazione, e ciò talvolta costringe a riorganizzare la sequenza di parametri in modo innaturale, o a fornire default privi di reale significato ad altri parametri.

Un esempio di abuso dei parametri di default che ho incontrato in un progetto reale è visibile nel **Listato 136**; l'oggetto rappresentato dalla classe *TimeScrollbar* è una particolare scrollbar, il cui cursore può essere allungato o accorciato tramite mouse per effettuare uno zoom, ed a cui è associata una scala (come in un righello) che evidenzia i tempi relativi ad ogni posizione. Essa è derivata da una classe più generale, una scrollbar con

la capacità di zoomare ma senza scale associate, ed il cui range non è limitato alla classe *Time*.
Notiamo che il costruttore di *TimeScrollbar* ha molti parametri, tuttavia ogni parametro è necessario per poter costruire un oggetto sensato: sostanzialmente, dobbiamo fornire la posizione, le dimensioni e la scala.

Listato 136

```
// Dichiarazione originale

class TimeScrollbar : public ZoomAndScrollBar
  {
  public :
    TimeScrollBar(  Window parent, int x, int y, int w,
      int h, Time min, Time max, ScaleRepr repr ) ;
    // ....

  private :
    // ....
  } ;



// Ridefinita da un programmatore
// (commento interno originale)

class TimeScrollbar : public ZoomAndScrollBar
  {
  public :
    TimeScrollBar( Window parent, int x = 0, int y = 0,
                   int w = 0, int h = 0,
                   Time min = 0, Time max = 0,
                  ScaleRepr repr = HMS) ;
    // Horray! Now featuring default arguments
    // ....

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

Durante l'attività di manutenzione, un programmatore ha modificato il costruttore come sopra, con valori di default ovviamente assurdi (ha ben poco senso creare una scrollbar con altezza, larghezza ed estensione temporale pari a zero): ovviamente ogni valore sarebbe stato ugualmente arbitrario. Altrettanto ovviamente, il programmatore li ha aggiunti perché

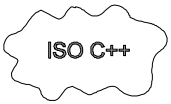
"la funzione aveva troppi parametri", trascurando il fatto che in nessuna chiamata i valori di default forniti potevano essere utilizzati come tali.

Un esempio di uso corretto dei parametri di default potrebbe invece essere una funzione che crea un bottone di vari tipi (*pushdown*, *radio*, *checkbox*, ecc) a seconda di un parametro, che può assumere come default il valore di *pushdown*, che rappresenta in effetti i bottoni di uso più comune.

Raccomandazione 115

Parametri di default devono essere forniti solo quando la maggior parte delle chiamate sfrutterà effettivamente tali parametri.

Il valore di default non deve essere necessariamente una costante: può essere una qualunque espressione, contenente anche variabili.



Il comitato ANSI/ISO C++ ha deliberato che in tal caso, per le funzioni non-membro i nomi che compaiono nell'espressione di default vengono legati (bound) nel punto di dichiarazione, ma vengono valutati ad ogni chiamata. Il type checking avviene nel punto di dichiarazione. Per le funzioni membro, il type checking ed il binding avvengono alla fine della dichiarazione di classe, per far sì che ogni nome definito all'interno della classe sia visibile nell'espressione; anche in questo caso la valutazione avviene ad ogni chiamata.

Con riferimento al **Listato 137**, l'output sarà "2 4 4", in quanto il nome '*k*' nella espressione di default è legato all'identificatore esterno '*k*' (visibile al momento della definizione di *f*), e viene rivalutato ogni volta: notiamo che la dichiarazione di una variabile locale '*k*' ***non*** influisce sul risultato, in quanto il binding (non la valutazione) avviene al momento della definizione di *f* e non al momento della chiamata. Il **Listato 137** può anche essere usato per verificare l'aderenza del vostro compilatore, anche se in questo caso, essendo la relativa delibera del comitato piuttosto datata, in pratica ogni compilatore commerciale dovrebbe essere già allineato col futuro standard.

Listato 137

```
#include <iostream.h>

int k = 0 ;

void f( int x = k )
  {
  cout << x << endl ;
  }

int main()
  {
  k = 2 ;
  f() ;

  k = 4 ;
  f() ;

  int k = 5 ;
  f() ;

  return( 0 ) ;
  }
```

Oggetti temporanei

Quando gli oggetti vengono passati per valore, o quando oggetti creati all'interno di una funzione vengono restituiti al chiamante, si ha normalmente la creazione di oggetti temporanei da parte del compilatore; consideriamo il **Listato 138**:

Listato 138

```
class BigObject
  {
  // ...
  } ;


BigObject f( BigObject b )
  {
  BigObject  newObject ;
  // ...
  return( newObject ) ;
  }
```

Quando *f* viene chiamata, il parametro attuale viene copiato in un temporaneo, che viene passato per valore alla funzione; al momento del return, viene eseguita una copia della variabile locale *newObject*. Per entrambi gli oggetti temporanei viene poi richiamato il distruttore nel blocco applicativo dove è avvenuta la chiamata di funzione.

L'inefficienza associata alla copia di oggetti di grandi dimensioni può essere risolta in diversi modi:

- Rinunciando al passaggio per valore, e passando invece i parametri per riferimento. Se il parametro non deve essere modificato, usare un const reference garantisce una sintassi identica al passaggio per valore (un perfetto information hiding) ed una efficienza di gran lunga superiore.

- Rinunciando a restituire copie di oggetti locali: ciò normalmente comporta modifiche sostanziali alla struttura della funzione. Ad esempio, potremmo passare un parametro per riferimento, e modificare il parametro stesso; a questo punto, sarebbe comunque più sensato trasformare la funzione in una funzione membro (in realtà una procedura) che modifichi l'oggetto al quale viene applicata. Non cadete invece nella facile tentazione di restituire puntatori ad oggetti dinamici allocati internamente (vedere il paragrafo "Argomenti ed Interfacce" in questo stesso capitolo).

- Modificando la struttura della classe, introducendo wrapper con un reference count o un meccanismo come le classi duali, in modo che gli oggetti temporanei vengano comunque generati, ma siano di piccole dimensioni. Questa soluzione, al costo di una struttura più complessa per le classi, garantisce di norma dei buoni risultati.

Raccomandazione 116

Limitare il passaggio per valore di oggetti di grandi dimensioni, e la restituzione per valore di oggetti di grandi dimensioni.

Lifetime dei temporanei

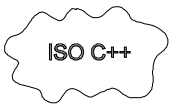
Consideriamo il codice del **Listato 139**, preso da [Str94]:

Listato 139

```
class String
  {
  public :
    friend String operator +( const String& s1,
      const String& s2 ) ;
    operator const char*() ;
    // returns a C-style string
    // ...
  } ;


void f( String s1, String s2 )
  {
  printf( "%s", (const char*)( s1 + s2 ) ) ;
  // ...
  }
```

all'interno della funzione *f*, viene creato un oggetto temporaneo di classe String per contenere il valore di *s1 + s2*; dopo che l'operatore di conversione a const char* è stato applicato a tale oggetto, esiste una garanzia che l'oggetto temporaneo viva abbastanza a lungo da non passare un "dangling pointer" alla funzione printf? Notiamo infatti che se il temporaneo viene distrutto *prima* della chiamata a printf, il puntatore ad una stringa C-style restituito dall'operatore di conversione punterà a memoria deallocata.

Sfortunatamente, in [ES90] la regola proposta per la lifetime dei temporanei è molto blanda: si richiede che il compilatore distrugga l'oggetto temporaneo tra il primo punto in cui il temporaneo è utilizzato e la fine del blocco applicativo in cui è stato generato. Ciò significa che il codice del **Listato 139** potrebbe funzionare correttamente su alcuni compilatori e non su altri, anche se entrambi i compilatori rispettano la regola di [ES90]; in effetti, molti compilatori tendono a postporre il più possibile la distruzione dei temporanei, in pratica deallocandoli tutti alla fine del blocco, ma non esiste al momento una regola generale.



La proposta ISO per la gestione dei temporanei è la seguente: i temporanei devono essere distrutti alla fine dell'*espressione completa* in cui appaiono,

dove con espressione completa si indica una espressione che non sia sotto-espressione di un'altra espressione: ad esempio, nel **Listato 140** l'espressione *s1 + s2* non è un'espressione completa, mentre *(p = s1 + s2) != NULL && p[0]* è un'espressione completa. Ciò significa anche che il codice del **Listato 140** è corretto, se e solo se il puntatore *p* non viene più utilizzato come espressione destra dopo la condizione dell'if; se ipoteticamente utilizziamo *p* all'interno del corpo dell'if, ad esempio per stampare la stringa, il codice è scorretto secondo la proposta ISO, mentre potrebbe o meno essere corretto, in funzione del compilatore, secondo la regola di [ES90].

Listato 140

```
class String
  {
  public :
    friend String operator +( const String& s1,
      const String& s2 ) ;
    operator const char*() ;
    // returns a C-style string
    // ...
  } ;


void f( String s1, String s2 )
  {
  const char* p = NULL ;

  if( ( p = s1 + s2 ) != NULL && p[ 0 ] )
    // ...
  }
```

Quali conclusioni possiamo trarre dalle osservazioni precedenti? Al momento, molti compilatori sono allineati con le regole di [ES90], il che significa che vi è una notevole incertezza nell'uso dei temporanei; d'altra parte, molte delle raccomandazioni in questo testo tendono proprio ad eliminare le situazioni potenzialmente pericolose, come quelle dei due listati precedenti: vedere ad esempio la **Raccomandazione 78**. Ritengo che uno stile di programmazione molto attento alla creazione ed all'uso dei temporanei sia una necessità anche utilizzando compilatori allineati alla proposta ISO: in fase di manutenzione, sarebbe troppo semplice violare la regola della "espressione completa" senza accorgersene.

Raccomandazione 117

Evitare codice che dipende dalla lifetime di oggetti temporanei; se ciò è necessario per ragioni di efficienza, commentare la porzione di codice in modo adeguato.

Risultati di operatori postfissi

Gli operatori postfissi ++ e -- sono molto particolari rispetto ai temporanei, poiché devono restituire una copia dell'oggetto *non* modificato, copia che ha durata temporanea, e che in alcuni (in realtà molti) casi non viene utilizzata e costituisce di fatto uno spreco di risorse.

Consideriamo il **Listato 141**, dove un'ipotetica classe Vector implementa un operatore postfisso di incremento, la cui semantica (sempre ipotetica) è di incrementare ogni elemento del vettore.

Listato 141

```
class Vector
  {
  public:
    Vector( int size ) ;
    Vector( const Vector& v ) ;
    ~Vector() ;
    const Vector& operator =( const Vector& v ) ;
    Vector operator ++( int dummy ) ;
    Vector operator +( const Vector& v ) ;
  private :
    int size ;
    int* data ;
  } ;


Vector :: Vector( int size )
  {
  Vector :: size = size ;
  data = new int[ size ] ;
  }


Vector :: Vector( const Vector& v )
  {
  size = v.size ;
```

```
  data = new int[ size ] ;
  for( int i = 0; i < size; i++ )
    data[ i ] = v.data[ i ] ;
  }


Vector :: ~Vector()
  {
  delete[] data ;
  }


const Vector& Vector :: operator =( const Vector& v )
  {
  ASSERT( size == v.size ) ;
  for( int i = 0; i < size; i++ )
    data[ i ] = v.data[ i ] ;
  }


Vector Vector :: operator++( int /* dummy */ )
  {
  Vector org( *this ) ;

  for( int i = 0; i < size; i++ )
    data[ i ]++ ;

  return( org ) ;
  }


Vector Vector :: operator +( const Vector& v )
  {
  ASSERT( size == v.size ) ;

  Vector sum( size ) ;
  for( int i = 0; i < size; i++ )
    size.data[ i ] = data[ i ] + v.data[ i ] ;

  return( sum ) ;
  }


int main()
  {
  Vector v1[ 100 ] ;
  Vector v2[ 100 ] ;
```

```
 // ...

v1++ ;  // il temporaneo non viene neppure utilizzato!

v2 = v2 + v1++ ;   // qui invece si usa il temporaneo

return( 0 ) ;
}
```

Purtroppo non possiamo applicare nessuna delle tecniche viste in precedenza per eliminare la creazione del temporaneo: l'arità dell'operatore è fisssata, così come la sua semantica, e non è quindi possibile ristrutturarlo in modo da evitare la creazione di un temporaneo.

La soluzione più immediata consiste nel *non* fornire gli operatori postfissi per oggetti di grandi dimensioni: notiamo infatti che gli operatori prefissi non soffrono dello stesso problema, potendo restituire l'oggetto modificato. Normalmente il codice che usa gli operatori postfissi si può modificare senza eccessivi problemi per utilizzare operatori prefissi, evitando così l'overhead di copia; per tale ragione, se i vostri oggetti sono molto grandi, è consigliabile non definire operatori postfissi nella classe corrispondente.
Notiamo infine che se stiamo semplicemente utilizzando una classe con oggetti di grande dimensione, non definita da noi, è comunque opportuno non applicare operatori postfissi ad instanze della classe: l'overhead che ne consegue raramente è giustificabile. In senso assoluto, un operatore postfisso che modifica l'oggetto cui è applicato sarà sempre meno efficiente di un operatore prefisso; per piccoli oggetti, la differenza può essere poco significativa, anche se all'interno di un loop potrebbe avere una sua rilevanza. Nel caso di oggetti di grandi dimensioni, anche se una libreria fornisce operatori postfissi, è comunque preferibile utilizzare sempre quelli prefissi.

Raccomandazione 118

Se lavorate con oggetti di grande dimensione, non utilizzare operatori postfissi; se create una classe con oggetti di grandi dimensione, non definite operatori postfissi.

10

Ereditarietà

"...It cannot be inherited, and if you want it
you must obtain it by great labor."
Thomas Stearns Eliot

L'ereditarietà è una delle basi del C++ e di ogni linguaggio object oriented; nel caso del C++, si tratta anche di uno dei punti dove la ricchezza del linguaggio, che consente ereditarietà pubblica, protetta, privata, e per ognuna di queste la scelta tra virtuale e non virtuale, è spesso poco approfondita dai programmatori. In questo capitolo tratteremo molti degli aspetti meno evidenti dell'ereditarietà, proponendo come sempre delle raccomandazioni a livello di codifica che consentano di evitare facilmente i problemi più frequenti.

Ereditarietà pubblica e privata

Da un punto di vista astratto, l'ereditarietà pubblica, così come l'ereditarietà protetta, stabilisce che ogni oggetto di classe derivata è-*un* oggetto di classe base. Da un punto di vista pragmatico, ciò è confermato dalla presenza, nell'interfaccia pubblica della classe derivata, di tutti i metodi pubblici della classe base; notiamo inoltre che la relazione è transitiva.
L'ereditarietà privata è concettualmente molto diversa, più simile in effetti al contenimento (vedremo più avanti le sottili differenze da quest'ultimo); ereditando in modo privato, non esponiamo i metodi della classe base, e pertanto non otteniamo una relazione del tipo è-*un*; all'interno della classe derivata, possiamo tuttavia chiamare i metodi della classe base, sfruttandone di conseguenza l'implementazione senza esporne l'interfaccia, come se un oggetto di classe base fosse contenuto in quello di classe derivata.

Ciò può essere molto conveniente, e di fatto l'ereditarietà privata è sottoutilizzata da molti programmatori C++, che impiegano invece l'ereditarietà pubblica anche in contesti in cui ciò è concettualmente errato. Consideriamo il **Listato 142**, dove è riportato un frammento del codice della libreria di classi fornita con un compilatore commerciale molto diffuso:

Listato 142

```
class Bag : public Collection
  {
  public :
    virtual void add( Object& o ) ;
    virtual Bool hasMember( const Object& o ) ;
    // ...
  } ;


class Set : public Bag
  {
  public :
    virtual void add( Object& o ) ;
    // ...
  } ;


void Set :: add( Object& o )
  {
  if( !( Bag :: hasMember( o ) ) )
    Bag :: add( o ) ;
  }
```

L'idea di base è di implementare un insieme (Set) derivandolo da una classe contenitore più generale (Bag) e di aggiungere un controllo sulla duplicazione degli elementi. Tuttavia il design della libreria è totalmente compromesso dall'uso errato dell'ereditarietà: nel listato in esame, si afferma che un insieme *è-un* contenitore generico, e ciò è palesemente errato. Chi non fosse convinto può esaminare il **Listato 143**: senza alcuna violazione del type system, e senza neppure utilizzare un cast esplicito, possiamo creare un "insieme" con due copie dello stesso elemento all'interno.

Listato 143

```
#include <set.h>

int main()
  {
  Set s ;
  s.add( 1 ) ;

  s.Bag::add( 1 ) ;  // inserisce un duplicato!

  return( 0 ) ;
  }
```

La ragione del malfunzionamento dovrebbe ormai essere evidente: l'uso di *Bag* per implementare un *Set* è semplicemente un dettaglio implementativo: un *Set* non è un contenitore generico, e quindi l'ereditarietà pubblica (o protected) non è utilizzabile. Ci si potrebbe chiedere perché il programmatore abbia quindi deciso di derivare come public; potrebbe essere stata una semplice svista, o una abitudine consolidata (in tutta la libreria viene utilizzata solo l'ereditarietà pubblica), dovuta forse ad una conoscenza parziale del linguaggio. Un'altra spiegazione è che alcune delle funzioni di *Bag* devono a tutti gli effetti far parte dell'interfaccia di *Set*: ad esempio, la funzione *hasMember* deve essere definita anche per *Set*.

L'uso corretto dell'ereditarietà privata richiede in effetti la scrittura di opportune funzioni di forwarding, nel caso alcune funzioni della classe base debbano essere comunque presenti nell'interfaccia della classe derivata; questo è comunque un piccolo prezzo rispetto alla correttezza concettuale e pragmatica dell'implementazione. Nel **Listato 144** possiamo vedere una versione corretta (almeno rispetto al problema esaminato) della classe *Set*; notiamo che la funzione di forwarding è dichiarata inline, per evitare ogni overhead: trattandosi di una funzione che difficilmente sarà sottoposta a manutenzione, i vantaggi della dichiarazione inline superano di gran lunga gli svantaggi connessi.

Listato 144

```
class Bag : public Collection
  {
  public :
    virtual void add( Object& o ) ;
    virtual Bool hasMember( const Object& o ) ;
```

```
    // ...
  } ;


class Set : private Bag
  {
  public :
    virtual void add( Object& o ) ;
    virtual Bool hasMember( const Object& o ) ;
    // ...
  } ;


void Set :: add( Object& o )
  {
  if( !( Bag :: hasMember( o ) ) )
    Bag :: add( o ) ;
  }

inline Bool Set :: hasMember( const Object& o )
  {
  return( Bag :: hasMember( o ) ) ;
  }
```

Se utilizzate l'ereditarietà privata, ricordate sempre che la relazione tra classe base e classe derivata è un dettaglio implementativo, e non va quindi esposto nell'interfaccia della classe: in futuro, potremmo scegliere di mantenere la stessa interfaccia, ma ereditare da un'altra classe, o da nessuna, o modificare l'ereditarietà privata in contenimento. Questo tipo di dettagli, esattamente come la struttura interna di una classe, dovrebbe essere modificabile liberamente (principio di incapsulazione) senza che siano necessarie modifiche laddove la classe viene utilizzata. Ad esempio, definire una funzione *Set :: CastToBag* per risolvere qualche problema implementativo (dovuto, indubbiamente, ad un cattivo design), permetterebbe di compromettere l'integrità della classe *Set* esattamente come nel caso dell'ereditarietà pubblica.

Raccomandazione 119

Utilizzate la derivazione pubblica o protetta solo se state effettivamente modellando una relazione del tipo è-*un*. Negli altri casi, utilizzate l'ereditarietà privata o il contenimento.

Raccomandazione 120

Non restituire mai un puntatore ad una classe base attraverso una funzione pubblica o protetta di una classe derivata con ereditarietà privata.

Purtroppo, l'ereditarietà privata in C++ non è "abbastanza privata" (vedere **Listato 70**), ed è possibile ridefinire funzioni private in una classe derivata, o ridefinire funzioni virtuali di un antenato non accessibile in una classe derivata: consideriamo una semplice modifica del **Listato 144**, unita ad una ulteriore derivazione di una classe dalla classe *Set*: **Listato 145**.

Listato 145

```
class Bag : public Collection
  {
  public :
    virtual void add( Object& o ) ;
    virtual Bool hasMember( const Object& o ) ;
    // ...
  } ;


class Set : private Bag
  {
  public :
    virtual void add( Object& o ) ;
    Bool isin( const Object& o ) ;
    // ...
  } ;


void Set :: add( Object& o )
  {
  if( ! hasMember( o )  )
    Bag :: add( o ) ;
  }

Bool Set :: isin( const Object& o )
  {
  return( hasMember( o ) ) ;
  }


class FakeSet : public Set
  {
  private :
```

```
    Bool hasMember( const Object& o ) { return( FALSE )
; }
  } ;
```

Le modifiche sono le seguenti: la classe *Set* non esporta più la funzione virtuale *hasMember*, sostituita da una funzione non virtuale *isin*; l'idea di base è di non rendere virtuale una funzione così fondamentale come l'appartenenza all'insieme. Tuttavia, l'implementazione di *isin* è basata su quella di *hasMember* della classe base: nell'intenzione del programmatore, la funzione chiamata dovrebbe essere *Bag :: hasMember*; in realtà, tale funzione è virtuale e può essere ridefinita in una sottoclasse di *Set*, anche se la classe in cui *hasMember* è stata dichiarata non è più accessibile alle sottoclassi di *Set*! La classe *FakeSet*, nuovamente senza alcuna violazione del type system o la presenza di cast espliciti o impliciti, permette di creare delle duplicazioni all'interno di un oggetto di classe *FakeSet* (e quindi di classe *Set*), che non dovrebbe permettere duplicazioni. L'errore fondamentale è stato quello di non eseguire un binding statico all'interno di *Set :: isin*: avremmo dovuto specificare esattamente che doveva essere chiamata la funzione *Bag :: hasMember*, senza risolvere dinamicamente la chiamata ad una funzione virtuale.

Raccomandazione 121

Se una funzione di una classe derivata come private chiama una funzione virtuale della classe base, è necessario legarla staticamente alla funzione della classe base, evitando la risoluzione dinamica della chiamata.

Purtroppo anche questa buona regola non è sufficiente a cautelarsi totalmente: esiste comunque il caso in cui una funzione virtuale sia chiamata all'interno della classe base, e non esportata dalla classe derivata come private: un esempio è dato nel **Listato 146**, dove per brevità ho utilizzato delle classi piuttosto schematiche rispetto ai listati precedenti:

Listato 146

```
#include <iostream.h>


class Base
  {
  public :
    void Show() { cout << ClassName() ; }
  protected :
```

```
    virtual const char* ClassName() { return( "Base" ) ;
}
  } ;


class PrivateDerived : private Base
  {
  public :
    void ShowBase() { Base :: Show() ; }
  } ;


class Fake : public PrivateDerived
  {
  private :
    const char* ClassName() { return( "Fake" ) ; }
  } ;


int main()
  {
  Fake f ;
  f.ShowBase() ;    // stampa "Fake"

  return( 0 ) ;
  }
```

Singolarmente, ogni componente sembra corretto: la classe base ha una funzione virtuale che restituisce il nome della classe (definirla come virtuale è pertanto necessario); la classe derivata ha una funzione, non virtuale, che stampa il nome della classe base, all'interno della quale la chiamata alla funzione della classe base è legata staticamente, una misura addirittura eccessiva se consideriamo che la funzioni *Show* non è virtuale. Tuttavia le misure prese non sono sufficienti, in quanto la classe *Fake* può facilmente alterare il comportamento della classe da cui è derivata. Sfortunatamente, non c'è nulla che si possa fare a livello di *DoubleDerived* per prevenire codice come quello di *Fake*: la debolezza è intrinseca nella struttura di *Base*; ciò non fa che rinforzare l'importanza della **Raccomandazione 79** e della **Raccomandazione 80**.
In ogni caso, la classe *Fake* è una classe "a rischio", poiché si basa su dettagli implementativi che potrebbero cambiare in futuro, se ad esempio *PrivateDerived* venisse modificata in modo da utilizzare il contenimento anziché l'ereditarietà privata. In tal senso, ridefinire funzioni virtuali di una classe non accessibile è una cattiva pratica di programmazione, che porta a codice con accoppiamenti tra classi più stretti del necessario. Ma anche

l'uso dell'ereditarietà privata con una classe base che chiama al suo interno funzioni virtuali è a rischio, in quanto non è possibile trattare la classe base come un semplice dettaglio implementativo: chi usa la classe derivata, potrà comunque alterare il funzionamento della classe base; in questo caso, considerate seriamente l'opportunità di usare il contenimento diretto, anziché l'ereditarietà privata.

Raccomandazione 122

Non ridefinire funzioni virtuali di una classe base non accessibile.

Raccomandazione 123

La derivazione privata da una classe base che chiama al suo interno funzioni virtuali dovrebbe essere sostituita dal contenimento diretto.

Ridefinire funzioni non virtuali

Consideriamo l'esempio del **Listato 147**, dove una classe derivata *public* ridefinisce una funzione *non* virtuale della classe base:

Listato 147

```
class Base
  {
  public :
    void Show() { cout << "Base " ; }
  } ;


class Derived : public Base
  {
  public :
    void Show() { cout << "Derived " ; }
  } ;


int main()
  {
  Derived d ;
  Base* bp = &d ;
  Derived* dp = &d ;
```

```
  bp->Show() ;  // stampa "Base"
  dp->Show() ;  // stampa "Derived"

  return( 0 ) ;
  }
```

Il risultato delle due chiamate alla funzione *Show* è diverso, anche se le chiamate sono applicate allo *stesso* oggetto, in quanto la funzione *Show* non è virtuale, e quindi chiamandola attraverso un puntatore alla classe base, si chiama in realtà la funzione definita nella classe base stessa.

Questo comportamento è già di per sé negativo: richiediamo al programmatore di sapere che *Show* non è virtuale, nonché di conoscere il tipo del puntatore, per poter correttamente comprendere il codice. Consideriamo ora un evento abbastanza comune nella programmazione in C++: è necessario derivare un'ulteriore classe da *Base*, tuttavia in questo caso la funzione *Show* deve essere virtuale affinché la nuova classe operi correttamente; la dichiarazione di *Show* nella classe *Base* andrà quindi modificata, aggiungendo la keyword *virtual*. Prima di poterlo fare, tuttavia, occorrerà trovare tutte le chiamate a tale funzione nel codice esistente, verificare il codice nelle varie situazioni, ed eventualmente modificarlo: viceversa, la modifica potrebbe introdurre errori in codice precedentemente corretto. Nel **Listato 147**, ad esempio, dichiarando *Show* virtuale l'output del programma cambierebbe da "Base Derived" a "Derived Derived". Si tratta di un comportamento molto insidioso, specie in programmi di grandi dimensioni, dove è probabile che simili errori passino inosservati durante la modifica del codice.

Osserviamo che nessuno dei problemi su esposti si applica se la funzione è dichiarata *virtual* sin dall'inizio, oppure se le funzioni *non* virtuali non vengono ridefinite nelle classi derivate. Osserviamo anche che, se la classe *Derived* fosse derivata tramite ereditarietà privata, il codice del **Listato 147** non sarebbe valido, in quanto sarebbe illegale convertire un puntatore a *Derived* in un puntatore a *Base*. Nuovamente, quindi, il problema non sussisterebbe sin dall'inizio. In effetti, forse l'unico caso in cui ha veramente senso ridefinire una funzione non virtuale è nell'ereditarietà di implementazione (ovvero privata), quando si voglia comunque rendere visibile una funzione della classe base. In ogni altro caso, un simile stile di programmazione può essere fonte di numerosi problemi in fase di comprensione e manutenzione del codice.

Raccomandazione 124

Non ridefinite le funzioni *non virtuali* delle classi base in classi derivate *public* o *protected* (ereditarietà di interfaccia); nel caso occorra farlo, verificate che non sia comunque più corretto derivare come *private* (ereditarietà di implementazione).

Ereditarietà multipla

In questo paragrafo considereremo solo le particolarità dell'ereditarietà multipla indipendente, ovvero quando una classe è derivata da più basi, ma non esistono due diversi cammini nel grafo di derivazione dalla classe derivata alla classe base. Tali situazioni, chiamate "a diamante" per la forma che il grafo di derivazione assume, e dette anche di "fork-join inheritance", verranno approfondite nel paragrafo successivo, relativo all'ereditarietà virtuale.

Nella derivazione multipla indipendente, in pratica la sola fonte di problemi è il name-clash tra due classi indipendenti, dovuto normalmente all'esistenza di omonimi nel linguaggio naturale, da cui vengono scelti i nomi delle funzioni; un esempio tipico, adattato da [Str94] che lo propone a proposito del renaming, è mostrato nel **Listato 148**: all'interno di *LotterySimulation*, dobbiamo sempre eseguire un bind statico delle chiamate a Draw(), viceversa il compilatore segnalerà una ambiguità nella chiamata. Spesso ciò si risolve ridefinendo la funzione localmente alla classe derivata, in modo che chiami entrambe le funzioni delle classi base: ovviamente, ciò è possibile solo quando la semantica è in qualche modo correlata, e ciò non avviene nell'esempio in esame.

Listato 148

```
class Lottery
  {
  // ...
  virtual void Draw() ;
  } ;


class GraphicalObject
  {
  // ...
  virtual void Draw() ;
```

```
  } ;


class LotterySimulation: public Lottery, public
GraphicalObject
  {
  // ...
  // Draw() NON e' ridefinita localmente
  void f() ;
  } ;


void LotterySimulation :: f()
  {
  Draw() ;                // Errore: chiamata ambigua
  Lottery :: Draw() ;     // OK
  }
```

Codice di questo tipo ha un difetto, già analizzato in precedenza, ovvero è poco resiliente alle modifiche nella gerarchia di classi: se introduciamo una nuova classe tra *LotterySimulation* e *Lottery*, il codice precedente potrebbe non chiamare la funzione corretta, a causa del binding statico. La soluzione proposta nel capitolo 7, ovvero l'uso di *inherited ::*, non è qui applicabile a causa dell'ereditarietà multipla. Esiste comunque una alternativa, visibile nel **Listato 149**, dove assumiamo la stessa definizione delle classi del listato precedente:

Listato 149

```
void LotterySimulation :: f()
  {
  // alternativa a Lottery :: Draw() ;
  this->*(&Lottery::Draw()) ;
  }
```

La sintassi è un po' contorta, ma l'idea è semplice: prendiamo l'indirizzo della funzione *Lottery::Draw* e la invochiamo su *this*. Osserviamo che per funzionare correttamente, la funzione *Draw* deve essere virtuale, viceversa non otterremmo benefici rispetto al binding statico. Questa soluzione è molto più resiliente rispetto alle modifiche nella gerarchia di classi, se confrontata con il binding statico; ha tuttavia il difetto di una scarsa leggibilità: se la userete sistematicamente, potrebbe valere la pena di definire una macro opportuna, che prenda come parametri il nome della classe e della funzione, e che nasconda la notazione un po' complessa del costrutto.

Raccomandazione 125

Se utilizzate il binding statico per eliminare l'ambiguità dovuta all'ereditarietà multipla, commentate adeguatamente il codice per prevenire problemi di manutenzione; considerate comunque l'alternativa di utilizzare i puntatori a funzioni membro.

Ereditarietà virtuale

L'ereditarietà virtuale è interessante soprattutto nelle situazioni di fork-join inheritance, dove quindi è coinvolta anche l'ereditarietà multipla: pertanto, in questo paragrafo ci occuperemo principalmente della fork-join inheritance. Alcune considerazioni riguardo l'ereditarietà virtuale singola sono presentate nel paragrafo successivo (esempi patologici).

Per snellire la discussione in seguito, è necessario definire un termine che non fa parte del vocabolario comune del paradigma object oriented: in caso di ereditarietà multipla, una classe può non essere accessibile tramite un percorso (a causa dell'ereditarietà ereditarietà privata) ma essere accessibile tramite un percorso alternativo, composto solo di ereditarietà pubblica o protetta. Diremo quindi che una classe è *accessibile* da un'altra se esiste un cammino composto da rami di derivazione pubblica o protetta da una classe all'altra. In tal caso, chiameremo *accessibile* anche l'intero cammino di derivazione.

Uno degli esempi più classici di fork-join inheritance è mostrato nel **Listato 150**, e riguarda il ben noto modello dello studente-lavoratore:

Listato 150

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

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

class Employee : public virtual Person
```

```
  {
  // ...
  } ;

class StudentEmployee :
public virtual Student, public virtual Employee
  {
  // ...
  } ;
```

L'uso della derivazione virtuale garantisce che *StudentEmployee* sia un'unica persona, ovvero che vi sia un unico sotto-oggetto di classe *Person* all'interno di ogni oggetto di classe *StudentEmployee*. Avrebbe senso usare la derivazione non-virtuale, ovvero definire lo studente-lavoratore come due persone? Ovviamente no.

Osserviamo ora il problema da un'ottica più generale: l'ereditarietà pubblica, così come quella protetta, rappresentano una relazione del tipo è-*un*: uno studente è-*una* persona; ciò significa che in tutte le situazioni in cui si ha ereditarietà fork-join *accessibile*, stiamo modellando una situazione in cui un oggetto di classe derivata è-*un* oggetto di classe base: ovvero non è-*due* (o più) oggetti di classe base. La conclusione è quindi molto semplice, anche se può risultare molto drastica: l'ereditarietà fork-join accessibile deve *sempre* essere virtuale. In ogni altro caso, stiamo utilizzando l'ereditarietà per modellare qualcosa d'altro (probabilmente il contenimento), o la stiamo usando per i risultati pratici che ci offre, dimenticandone gli aspetti concettuali, o ancora stiamo usando l'ereditarietà pubblica dove avremmo dovuto usare l'ereditarietà privata (notiamo infatti che in quanto sopra abbiamo parlato di ereditarietà accessibile).

Una violazione della "regola" di cui sopra viene da Stroustrup stesso, in [Str91]: nel **Listato 151**, la classe *satellite* è ottenuta derivando da *task* e da *displayed*, entrambe le quali sono derivate da una classe *link* per gestire una lista di oggetti; nell'esempio in questione, un oggetto *satellite* deve appartenere a due liste diverse perché il programma sia corretto.

Listato 151

```
class link
  {
  // ...
  } ;
```

```
class task : public link
  {
  // ...
  } ;

class displayed : public link
  {
  // ...
  } ;

class satellite : public task, public displayed
  {
  // ...
  } ;
```

Cerchiamo ora di leggere il testo del programma ad un più alto livello: un task è un link; un satellite è un task e quindi è un link, tuttavia nel suo layout attuale non è-*un* link, ma *ha-due* link. Se la differenza vi sembra troppo filosofica, provate a pensare alla differenza tra essere un cane o avere due cani!
In realtà, con buona pace di tutti, il **Listato 151** è un esempio errato di fork-join inheritance accessibile; potrebbe essere corretto in molti modi, ad esempio derivando le due classi *task* e *displayed* in modo privato da *link*, o (meglio) inserendo un *link* come sotto-componente di entrambe, o (meglio ancora) evitando di inserire la conoscenza dei link all'interno delle classi, ed usando invece delle classi o dei template come contenitore. Sicuramente, il codice così com'è funziona, ma non è concettualmente corretto e neppure facilmente modificabile: se una delle classi base richiedesse più di un link, occorrerebbe una ristrutturazione dell'intera gerarchia.

Raccomandazione 126

L'ereditarietà fork-join accessibile deve sempre essere virtuale.

Possiamo ora facilmente considerare il caso dell'ereditarietà fork-join *non accessibile*, ovvero dove sia stata usata l'ereditarietà privata; come abbiamo visto in precedenza, l'ereditarietà privata non rappresenta una relazione del tipo è-*un*, ma è molto più simile al contenimento, per quanto vi siano delle differenze a causa delle funzioni virtuali. Tuttavia rappresenta un dettaglio implementativo, e le classi ulteriormente derivate non dovrebbero esserne a conoscenza. Consideriamo allora l'esempio del **Listato 152**, ottenuto adattando leggermente la struttura del listato precedente; l'idea di base è di avere una lista di task, una lista di finestre, ed una classe che rappresenti le

applicazioni con una finestra, che sono sia task che finestre. Il collegamento tra gli oggetti in una lista è implementato con derivazione da una classe link, derivazione privata in quanto si tratta appunto di una scelta implementativa, non di una proprietà concettuale delle classi. È stata usata la derivazione virtuale per avere un'unica lista per le applicazioni con finestra, che collega quindi le finestre ed i task.

Listato 152

```
class link
  {
  // ...
  } ;

class task : private virtual link
  {
  // ...
  } ;

class window : private virtual link
  {
  // ...
  } ;

class windowedApplication : public virtual task, public
virtual window
  {
  // ...
  } ;
```

Se a prima vista vi sembra una buona soluzione, considerate che la classe *windowedApplication* sta violando il principio di incapsulazione: essa sfrutta un dettaglio implementativo delle classi *task* e *window* per creare un'unica lista per task e finestre. Il problema dell'ereditarietà privata e virtual risiede proprio nella possibilità che lascia ad altre classi derivate di manipolare dettagli implementativi delle classi base: se modifichiamo la classe *task* utilizzando il contenimento anziché l'ereditarietà privata, la classe *windowedApplication* non funzionerà più correttamente, e certamente non avremo una sola lista per task e finestre. Nuovamente, anche in questo caso esistono diverse soluzioni, e la più idonea resta comunque di avere una classe contenitore separata: certamente, l'ereditarietà fork-join *non* accessibile non dovrebbe mai essere virtuale.

Raccomandazione 127

L'ereditarietà fork-join non accessibile non deve mai essere virtuale.

Esempi patologici

L'usuale cautela va posta nell'uso dell'ereditarietà multipla, così come nella singola, quando alcune funzioni delle classi base chiamano a loro volta una funzione virtuale. Nel **Listato 153**, una classe *Component* è utilizzata da due classi derivate, poi fuse in una terza; la classe di base ha una funzione virtuale che svolge qualche tipo di azione, ed una funzione di debugging che esegue la stessa azione con qualche flag di debug abilitato. Le classi derivate modificano l'azione, e la classe join fa eseguire l'azione ad entrambe le sue componenti; l'ereditarietà è sempre privata (dettaglio implementativo) e non virtuale, in accordo a quanto sopra.

Listato 153

```
#include <iostream.h>


class Component
  {
  public :
    virtual void Action()
      {
      cout << " action " ;
      }
    virtual void Debug()
      {
      cout << " debug on " ;
      Action() ;
      cout << " debug off " ;
      }
  } ;


class Derived1 : private Component
  {
  public :
    virtual void Action()
      {
      cout << " action1 " ;
      Component :: Action() ;
```

```
      }
    void Debug()
      {
      Component :: Debug() ;
      }
  } ;


class Derived2 : private Component
  {
  public :
    virtual void Action()
      {
      cout << " action2 " ;
      Component :: Action() ;
      }
    void Debug()
      {
      Component :: Debug() ;
      }
  } ;


class Join : private Derived1, private Derived2
  {
  public :
    virtual void Action()
      {
      cout << " join-action " ;
      Derived1 :: Action() ;
      Derived2 :: Action() ;
      }
    virtual void DebugPart1()
      {
      Derived1 :: Debug() ;
      } ;
    virtual void DebugPart2()
      {
      Derived2 :: Debug() ;
      }
  } ;


int main()
  {
  Join j ;

  j.Action() ;             // OK
```

```
  cout << "\n" ;

  j.DebugPart1() ;  // NO!

  return( 0 ) ;
  }
```

Tuttavia, provando ad eseguire il debugging di una delle componenti del join, viene eseguita l'azione anche per l'altra componente; questo è un effetto "di esplosione" di anomalie che avevamo già notato in precedenza (con l'ereditarietà singola) quando la classe base chiama una sua funzione virtuale. La presenza dell'ereditarietà fork-join privata genera un effetto ancora più difficile da comprendere con immediatezza leggendo il codice, che potrebbe facilmente sembrare corretto ad una prima (e forse anche ad una seconda) lettura.

Abbiamo visto in precedenza che l'ereditarietà fork-join non accessibile non dovrebbe mai essere virtuale; in realtà, anche rimanendo all'interno dell'ereditarietà singola, l'accoppiata "private-virtual" è raramente vincente: il **Listato 154** può apparire corretto (e da alcuni test che ho eseguito, molti programmatori lo giudicano tale) ma sui compilatori che rispettano le regole di accessibilità per i costruttori, non verrà neppure compilato.

Listato 154

```
class Base
  {
  public :
    virtual void x() = 0 ;
    Base( int d ) {} ;     // non-default constructor
  } ;

class PrivateDerived: private virtual Base
  {
  public:
     PrivateDerived() : Base( 10 ) {} ;
     // Default constructor
  } ;

class DoubleDerived : public virtual PrivateDerived
  {
  public :
```

```
    DoubleDerived() : PrivateDerived() {} ;
    // Default constructor
    void x() {} ;
  } ;

int main()
  {
  DoubleDerived dd() ;  // errore!

  return( 0 ) ;
  }
```

Il problema è piuttosto semplice: anche nel caso dell'ereditarietà singola, il C++ richiede che il costruttore delle classi derivate da una virtual base class richiamino il costruttore della classe base esplicitamente. Ciò significa che, nonostante l'ereditarietà singola, chiamare il costruttore di *Base* all'interno del costruttore di *PrivateDerived* non è sufficiente: dovremmo chiamarlo anche dall'interno del costruttore di *DoubleDerived*. Tuttavia non possiamo, ovvero anche modificando il listato in modo tale da chiamarlo, otterremo un diverso errore di compilazione, in quanto il costruttore della classe base non è accessibile. Alcuni compilatori rilassano le restrizioni di accesso ai costruttori delle virtual base class per permettere a codice simile di essere compilato, almeno nei casi di ereditarietà singola; non si tratta comunque di una caratteristica standard del linguaggio.

Regole semplificate

Le raccomandazioni viste poc'anzi aiutano il programmatore che utilizza l'ereditarietà multipla ad organizzare il suo codice in modo concettualmente più corretto e pragmaticamente meno soggetto ad errori; tuttavia, non forniscono alcun suggerimento a chi vuole fornire una libreria di classi o a chi comunque debba creare degli oggetti che possano essere facilmente riutilizzati e specializzati da altri programmatori.

Il problema maggiore, in simili frangenti, è che non si può conoscere a priori se le nostre classi verranno o meno impiegate in situazioni di ereditarietà multipla, e non è quindi semplice decidere a priori che tipo di derivazione utilizzare.
Un esempio è visibile nel **Listato 155**: se forniamo una libreria di classi per la creazione di interfacce utente, molti oggetti saranno derivati da una classe di più alto livello *Window*; che tipo di derivazione dobbiamo usare: virtual o non-virtual? In effetti, in diverse librerie di classi per GUI il problema è risolto in modo drastico, imponendo che gli utilizzatori non

impieghino l'ereditarietà multipla; in altri casi, come nel piccolo listato di esempio, il problema è semplicemente ignorato, in favore dell'uso dell'ereditarietà pubblica non-virtual.

Listato 155

```
class Window
  {
  // ...
  }


class DialogBox : public Window
  {
   // ...
  } ;
```

Notiamo comunque che la definizione del listato precedente, per quanto apparentemente corretta, di fatto impedisce l'ereditarietà fork-join con radice in *Window* ed uno dei rami in *DialogBox*, per i problemi visti precedentemente.

L'obiezione più comune è, per l'appunto, che gli sviluppatori della libreria non possono sapere come la libreria stessa verrà utilizzata; con un minimo di sforzo speculativo, tuttavia, possiamo comunque estendere il criterio definito per l'ereditarietà fork-join anche al contesto dell'ereditarietà singola, e dare così delle buone linee guida anche a chi sviluppa delle librerie.

Riprendendo l'esempio del **Listato 155**, una finestra di dialogo è una finestra, ed è quindi corretto utilizzare l'ereditarietà pubblica (o protetta); una finestra non potrà mai *essere* due finestre (potrà invece contenerne molte) per cui non dovrebbe essere possibile ottenere, per ereditarietà multipla accessibile, una classe la cui istanza contenga due istanze ereditate di *Window*. Ne consegue che la derivazione più adatta è pubblica ma virtual, contrariamente al listato dato, e contrariamente a quanto avviene in molte librerie di classi commerciali.

Al di là dell'esempio, si tratta di un criterio molto generale, che solo in casi molto rari (spesso a dire il vero frutto di un design discutibile) può essere necessario trasgredire: l'ereditarietà pubblica o protetta dovrebbe essere virtual, l'ereditarietà privata non-virtual. Ciò corrisponde fortemente

all'idea che le relazioni *è-un* non consentono la moltiplicazione della base, mentre le relazioni *ha-un* la consentono.

Esistono degli svantaggi nell'uso dell'ereditarietà virtuale ogni volta che si deriva in modo pubblico o protetto? Purtroppo si: il **Listato 156**, che in assenza di ereditarietà virtuale sarebbe corretto, usando l'ereditarietà virtuale genera un errore di compilazione.

Listato 156

```
#include <iostream.h>


class Base
  {
  public :
    Base( int i ) { x = i ; }
    void Print() { cout << x ; }
  private :
    int x ;
  } ;


class Derived : public virtual Base
  {
  public :
    Derived( int i ) : Base( i ) {} ;
  } ;


class DoubleDerived : public virtual Derived
  {
  public :
    DoubleDerived( int i ) : Derived( i ) {} ;
  } ;


int main()
  {
  DoubleDerived dd( 5 ) ;
  dd.Print() ;

  return( 0 ) ;
  }
```

Il problema, come dovrebbe essere chiaro dal paragrafo precedente, è dovuto alle speciali regole di inizializzazione del C++ nel caso della derivazione virtuale: in particolare, le inizializzazioni intermedie sono ignorate, ed il costruttore della classe istanziata deve essere in grado di raggiungere i costruttori di tutte le basi virtuali, anche indirette, oppure devono esistere dei costruttori di default per le classi base virtuali.

Esistono comunque due soluzioni per il problema evidenziato; la prima, immediata soluzione richiede che le classi derivate siano comunque a conoscenza dell'intera gerarchia, sino alle classi base, e che i costruttori delle classi derivate chiamino opportunamente i costruttori delle classi base virtuali. Un esempio è dato nel **Listato 157**, dove modificando il solo costruttore di *DoubleDerived* rendiamo il programma nuovamente corretto.

Listato 157

```
#include <iostream.h>


class Base
  {
  public :
    Base( int i ) { x = i ; }
    void Print() { cout << x ; }
  private :
    int x ;
  } ;


class Derived : public virtual Base
  {
  public :
    Derived( int i ) : Base( i ) {} ;
  } ;


class DoubleDerived : public virtual Derived
  {
  public :
    DoubleDerived( int i ) : Base( i ), Derived( i ) {}
    // Modificato per chiamare anche il costruttore
    // della classe base indiretta
  } ;
```

```
int main()
  {
  DoubleDerived dd( 5 ) ;
  dd.Print() ;

  return( 0 ) ;
  }
```

Il difetto di tale tecnica risiede appunto nella duplicazione del codice di inizializzazione della base, che nell'esempio dato è presente sia nel costruttore di *Derived* che in quello di *DoubleDerived*; per quanto di norma tale codice sia sostanzialmente una chiamata al costruttore, come nell'esempio, si tratta in ogni caso di un più forte accoppiamento sull'interfaccia del costruttore e comunque di codice che dovrà essere parte del processo di manutenzione in seguito.

Una soluzione alternativa, che richiede alcune righe di codice in più inizialmente, ma che semplifica la manutenzione in seguito, è proposta nel **Listato 158**:

Listato 158

```
#include <iostream.h>

class Base
  {
  public :
    Base( int i )
      {
      Init( i ) ;
      }
    void Print() const
      {
      cout << x ;
      }

  protected :
    Base()
      {
      } ;
    void Init( int i )
      {
      x = i ;
      }

  private :
```

```
    int x ;
  } ;


class Derived : public virtual Base
  {
  public :
    Derived( int i )
      {
      Init( i ) ;
      }

  protected :
    Derived()
      {
      } ;
    void Init( int i )
      {
      Base :: Init( i ) ;
      // eventuali inizializzazioni locali
      }
  } ;


class DoubleDerived : public virtual Derived
  {
  public :
    DoubleDerived( int i )
      {
      Init( i ) ;
      }
    void Init( int i )
      {
      Derived :: Init( i ) ;
      // eventuali inizializzazioni locali
      }
  } ;


int main()
  {
  const DoubleDerived dd( 5 ) ;
  // funziona anche su oggetti const
  dd.Print() ;

  return( 0 ) ;
  }
```

In questo caso, il codice di ogni costruttore con parametri è spostato in una corrispondente funzione protetta *Init*(); il costruttore richiama direttamente tale funzione. Inoltre, aggiungiamo un costruttore di default protetto con corpo vuoto a tutte le classi che ne sono prive (nell'esempio, ad ogni classe). La funzione *Init*() per le classi derivate chiama anche la corrispondente funzione per le classi base dirette, realizzando di fatto lo stesso ordine di chiamata dei costruttori. In questo caso, il compilatore troverà un costruttore (protetto e vuoto) di default per la classe base virtuale, ed accetterà l'istanziazione della classe derivata; il costruttore della classe derivata, attraverso le funzioni *Init()*, eseguirà anche il codice di inizializzazione delle classi base virtuali.

Anche in questo caso esiste uno svantaggio, ovvero si viola la **Raccomandazione 68**, il che può portare in determinati casi alla riduzione delle prestazioni del programma. Pertanto, esiste una scelta fra tre tecniche nel caso di derivazione pubblica:

1. usare la derivazione pubblica *non* virtuale; ciò può portare a problemi piuttosto seri nel successivo riuso tramite ereditarietà multipla. Potrebbe essere necessario costringere gli utenti delle nostre classi ad usare solo l'ereditarietà singola.

2. usare la derivazione pubblica virtuale, replicando il codice di chiamata ai costruttori di tutte le classi base virtuali nel costruttore delle classi derivate; ciò permette l'uso dell'ereditarietà multipla, ma porta a codice potenzialmente più problematico da mantenere.

usare la derivazione pubblica virtuale, usando lo schema di Init() + costruttore di default protetto; ciò permette l'uso dell'ereditarietà multipla, ma può portare ad un degrado delle prestazioni se alcuni dei membri hanno una inizializzazione particolarmente onerosa.

La scelta della tecnica da usare è largamente dipendente dalle vostre reali esigenze e dalle classi che state sviluppando; è comunque importante sapere che non basta scrivere "class X : public class Y" per utilizzare in modo adeguato l'ereditarietà e permettere il riuso del codice.
Come norma, l'uso della tecnica (2) è spesso accettabile quando le classi base sono relativamente stabili, almeno per quanto riguarda l'interfaccia dei costruttori; viceversa, la tecnica (3) è preferibile, salvo i casi in cui il degrado delle prestazioni sia inaccettabile (ma dovreste misurarlo, non affidarvi all'intuizione) nel qual caso non resta che accettare i potenziali

problemi connessi all'uso dell'ereditarietà multipla fork-join ed utilizzare la tecnica (1).

Non esistono invece problemi connessi all'uso dell'ereditarietà non-virtuale in unione alla derivazione privata: in pratica, ogni utilizzo di "private virtual" rappresenta una violazione poco esplicita dell'information hiding, e va quindi evitato.

Raccomandazione 128

Quando è possibile, utilizzate l'ereditarietà virtuale ogni volta che derivate in modo pubblico o protetto.

Raccomandazione 129

L'ereditarietà privata dovrebbe sempre essere non-virtual.

11

Template

"...It is always the same shape, only very numerous."
Charlotte Perkins Gilman

I template, pur non essendo una delle estensioni più recenti per il C++, hanno visto un impiego abbastanza limitato, e ristretto in gran parte alle classi contenitore. Molto vi sarebbe da dire sull'uso dei template in fase di design, in quanto il C++ nasce come linguaggio tipato staticamente, e come tale mal si presta ad implementazioni che basano la genericità sull'inheritance, come è invece abitudine comune (ad esempio) in Smalltalk. Parte di queste osservazioni troveranno spazio nel capitolo 14, ma essendo questo libro dedicato alla fase di codifica, molte di esse esulano comunque dalla trattazione. Proprio per l'uso ristretto dei template, tuttavia, risulta difficile identificare delle regole di codifica che prevengano gli errori più comuni: in questo senso, l'uso limitato e molto schematizzato sulle classi contenitore ha in gran parte prevenuto molti dei problemi che probabilmente si nascondono nei template.

In questo capitolo vedremo comunque due dei problemi più comuni (dichiarazioni multiple e dimensioni del codice) e spenderemo alcune parole su una possibilità dei template largamente ignorata dagli sviluppatori, e che può invece tornare utile in molte situazioni.

Dichiarazioni multiple

Consideriamo il **Listato 159**: un template di classe fornisce due versioni della stessa funzione, una parametrizzata sull'argomento del template, ed una con tipo prefissato.

Listato 159

```
template <class T> class MultDecl
  {
  public:
```

```
    void f( int x ) {} ;
    void f( T a ) {} ;
  } ;


int main()
  {
  MultDecl< char > cc ;          // OK
  MultDecl< int > ci ;           // NO

  return( 0 ) ;
  }
```

Il template può essere istanziato in quasi tutti i casi (es. su *char*) ma se si tenta di istanziarlo su *int* si ottiene un errore di compilazione (normalmente "dichiarazione multipla") in quanto la stessa funzione è dichiarata due volte nella stessa classe. Il problema non è totalmente accademico, perché accade spesso che i programmatori tendano a fornire un'interfaccia "esuberante" per una classe, nell'intento di coprire tutti gli usi possibili, anziché concentrarsi sulle funzioni realmente importanti; ciò avviene con frequenza anche maggiore per i costruttori.
È evidente che in ogni caso dovremmo evitare simili costrutti, o quanto meno commentare adeguatamente l'header del template; tuttavia, esiste una tecnica spesso applicabile per risolvere il problema senza eliminare la funzione non parametrica, e senza cambiarle nome. La condizione necessaria è che l'implementazione della funzione non parametrica non sia dipendente dalla classe parametro, neppure indirettamente: non deve quindi accedere a membri della classe il cui tipo sia parametrizzato, o chiamare altre funzioni membro parametrizzate. In questo caso, è possibile ristrutturare il codice come da **Listato 160**:

Listato 160

```
class Base
  {
  public :
    void f( int x ) {} ;
  } ;


template <class T> class MultDecl : public Base
  {
  public:
    void f( T a ) {} ;
  } ;
```

```
int main()
  {
  MultDecl< char > cc ;
  MultDecl< int > ci ;

  return( 0 ) ;
  }
```

La tecnica consiste nel separare la parte indipendente dal parametro in una classe base, ove definiremo la nostra funzione, e nel derivare la classe template da tale base: in questo caso, istanziando il template su *int*, non avremmo più un conflitto dovuto a dichiarazioni multiple, in quanto la funzione definita nel template coprirà semplicemente quella della classe base.
Naturalmente, in funzione dell'uso che faremo della funzione, potrebbe essere più opportuno dichiararla come virtual; in questo caso, ricordate che se la funzione viene anche *richiamata* (non solo definita) nella classe base, potrebbe essere necessario eseguire un binding statico della funzione stessa con l'operatore *::* all'interno della classe base.

Raccomandazione 130

Attenzione all'overloading di funzioni parametrizzate e non parametrizzate nei template, che può impedire l'istanziazione con alcuni parametri. Preferibilmente, spostate le funzioni non parametriche in una classe base.

Anziché spostare una sola funzione in una classe base, per evitare problemi di dichiarazioni multiple, potremmo anche spostare l'intera parte di un template indipendente dai parametri in una classe base ai fini di ottenere codice più snello: tale tecnica è trattata nel prossimo paragrafo.

Dimensioni del codice

Una volta definita una classe template, il compilatore genererà una nuova istanza del codice relativo ogni volta che il template viene istanziato. Anche se in teoria potrebbe essere possibile, in determinati casi, generare "codice generico" che possa essere condiviso per una famiglia di istanziazioni, ciò comporta difficoltà molto elevate e problemi la cui soluzione è tutt'altro che immediata (basti pensare alle variabili membro o locali a funzioni membro e di classe static). La replicazione del codice, tuttavia, può creare

seri problemi di “esplosione” delle dimensioni dell’eseguibile, nel caso si utilizzino molte istanze del template; ciò probabilmente non avviene nell’uso di classi contenitore, ma non è infrequente se ad esempio si usano i template per fornire un allocatore di memoria alternativo. Consideriamo infatti un programma in cui si voglia decidere, per ogni classe, quale allocatore usare: ogni allocatore potrebbe essere ottimizzato per una particolare politica di accesso, o fornire supporti più sofisticati come garbage collection, allocazione in memoria condivisa, e così via.
Poiché è possibile che alcune classi siano utilizzate con diversi allocatori in diversi punti del programma, non è possibile basarsi solo sull’ereditarietà (o meglio è possibile, ma ciò porterebbe ad una esplosione del numero di classi derivate); potremmo però usare un template, come nel **Listato 161**:

Listato 161

```
template <class T> class SharedAlloc : public T
  {
  void* operator new( size_t sz ) ;
  // ...
  } ;


template <class T> class StackAlloc : public T
  {
  void* operator new( size_t sz ) ;
  // ...
  } ;


template <class T> class CollectAlloc : public T
  {
  void* operator new( size_t sz ) ;
  // ...
  } ;
```

Ogni allocatore definisce l’operatore *new* (e *delete*) per attuare una determinata politica di allocazione; volendo una instanza allocata su shared memory di una classe *Semaphore*, potremmo quindi usare *SharedAlloc*< *Semaphore* >, mentre una versione di *Semaphore* con allocazione a stack può essere ottenuta istanziando *StackAlloc*< *Semaphore* >. La tecnica è molto versatile perché consente di allocare ogni classe con ogni politica, e disaccoppia il codice della classe dalla particolare politica di allocazione.
Il difetto della tecnica è che ogni volta che instanziamo una classe basata su un template, l’intero codice del template viene replicato: se ogni classe

usata nel programma è istanza di un template che ne definisca la classe di allocazione, l'impatto delle varie versioni del codice di allocazione sulle dimensioni del programma sarà veramente notevole.

D'altra parte, con ogni probabilità gran parte del codice degli allocatori è indipendente dal tipo del parametro, e potremmo quindi sfruttare la tecnica vista al paragrafo precedente, ovvero spostarlo in una classe base, con lo scopo stavolta di ridurre le dimensioni del codice: un esempio è dato nel **Listato 162**:

Listato 162

```
class SharedAllocator
  {
  void* operator new( size_t sz ) ;
  // ...
  } ;


class StackAllocator
  {
  void* operator new( size_t sz ) ;
  // ...
  } ;


class CollectAllocator
  {
  void* operator new( size_t sz ) ;
  // ...
  } ;

template <class T> class SharedAlloc :
public T , private SharedAllocator
  {
  // ...
  } ;


template <class T> class StackAlloc :
public T , private StackAllocator
  {
  // ...
  } ;
```

```
template <class T> class CollectAlloc :
public T , private CollectAllocator
  {
  // ...
  } ;
```

La differenza fondamentale rispetto alla versione iniziale è che il codice del template è ora notevolmente più snello (in alcuni casi forse nullo) perché utilizziamo l'ereditarietà per ottenere una sola copia del codice, condivisa da tutte le istanze del template, mentre il template è utilizzato solo per la genericità che permette. In questo caso, ogni istanza del template produrrà solo una replicazione della (ridotta) quantità di codice che appartiene effettivamente al template.

Notiamo che astrarre il codice non generico in una classe base ha anche ulteriori vantaggi: permette di *non* distribuire il sorgente della classe base (un problema che molti sviluppatori associano ai template) e permette un più semplice testing e debugging, in quanto possiamo verificare il codice non generico una volta per tutte, indipendentemente dal parametro del template.

Raccomandazione 131

Dati e funzioni membro di un template che siano indipendenti dai parametri del template possono essere spostate in una classe base da cui il template è derivato.

Specifica

Una tendenza comune nell'uso dei template, sulle cui conseguenze sarebbe importante investigare più in profondità, è quella di sovra-generalizzare il codice[21]; nel caso dei template, la sovra-generalizzazione può portare a comportamenti inconsistenti del codice stesso, che possono essere difficili da individuare in programmi di grandi dimensioni.

[21]Il problema della sovra-generalizzazione è implicito in ogni tecnica di generalizzazione, anche nell'ereditarietà, nel qual caso può portare a problemi di consistenza all'interno di una catena di derivazioni [Bor90].

Un esempio molto semplice è presentato nel **Listato 163**: un template di funzione *min* che restituisce il minimo tra i suoi due argomenti, assumendo che il tipo degli argomenti fornisca l'operatore '<':

Listato 163

```
#include <iostream.h>

template< class T > T min( T a, T b )
  {
  return( a < b ? a : b ) ;
  }

int main()
  {
  int i1 = 2 ;
  int i2 = 1 ;

  char* c1 = "b" ; // su alcune macchine occorre
  char* c2 = "a" ; // invertire a, b
                   // per avere il malfunzionamento!

  cout << min( i1, i2 ) << endl ;
  cout << min( c1, c2 ) ;
  // NB: non e' portabile confrontare
  // puntatori a due array diversi!

  return( 0 ) ;
  }
```

Su molte macchine il programma fornirà il risultato errato (ovvero 'b' anziché 'a') nel caso delle stringhe; su altre occorrerà scambiare la dichiarazione di *c1* e di *c2*, oppure il comportamento potrebbe essere random. In effetti, nel caso delle stringhe ci troviamo a comparare due puntatori che non puntano all'interno dello stesso array, operazione dal risultato indefinito. In ogni caso, notiamo che l'errore è ancora a monte, ovvero non dovremmo affatto comparare i puntatori, ma gli oggetti puntati.
Esistono diversi approcci al problema, tra cui la definizione di una apposita classe *Comparable* da usare come argomento del template, e da cui vanno derivate le classi su cui vogliamo istanziare il template stesso; in realtà, ho utilizzato l'esempio principalmente per introdurre l'idea che, in determinati casi, sarebbe utile poter ridefinire il comportamento di un template su casi particolari: nel caso in esame, pur senza ottenere una soluzione completa, sarebbe sufficiente poter dire che, nel caso l'argomento sia di tipo char*,

esiste una versione più specifica della funzione, da utilizzare in luogo del template.

In effetti questa possibilità esiste, anche se ho constatato che molti programmatori non ne sono a conoscenza: un esempio è presentato nel **Listato 164**.

Listato 164

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


template< class T > T min( T a, T b )
  {
  return( a < b ? a : b ) ;
  }


// ridefinita nel caso di char*
char* min( char* a, char* b )
  {
  if( strcmp( a, b ) < 0 )
    return( a ) ;
  else
    return( b ) ;
  }


int main()
  {
  int i1 = 2 ;
  int i2 = 1 ;

  char* c1 = "b" ;
  char* c2 = "a" ;

  cout << min( i1, i2 ) << endl ;
  cout << min( c1, c2 ) ;

  return( 0 ) ;
  }
```

La stessa tecnica è utilizzabile per ridefinire l'intera istanziazione di una classe template, o la singola funzione membro di una istanziazione di

classe; si tratta in generale di una possibilità molto utile, qualora esista una funzione di libreria particolarmente ottimizzata che per un certo tipo di dato, che può essere sostituita in casi particolari al codice generico del template.

12

Casting

"Cast away care, he that loves sorrow..."
Thomas Dekker

Il casting -ovvero la conversione di un tipo in un altro- è una tra le caratteristiche più discusse che il C++ ha ereditato dal C; si tratta in effetti di una funzionalità talvolta utile, molto spesso abusata, e per quanto questa affermazione possa essere impopolare, generalmente indice di un design affrettato. Certamente, vi sono ambienti e librerie che incoraggiano o richiedono l'uso continuo dei cast, tuttavia ciò non è sufficiente a far considerare i cast stessi come elementi di buona programmazione: un cast è una violazione del type system, e come tale va considerata. Peggio ancora, la violazione passa del tutto inosservata al compilatore, e l'effetto a run-time è talvolta largamente dipendente dall'implementazione e dal sistema operativo. Codice che sembra funzionare in sistemi senza protezione della memoria genererà eccezioni su sistemi protetti, e codice che funziona correttamente su un processore può causare eccezioni su un'altra architettura, con diverse assunzioni circa gli indirizzi per locazioni di tipo diverso.

Purtroppo, in C i cast sono molto comuni, specie tra puntatori ed interi, ed hanno una sintassi molto semplice: *(T)expr* converte il tipo dell'espressione *expr* a *T*. Una sintassi così immediata che si tende ad eccedere nell'uso, e che li rende anche difficili da trovare con un semplice strumento, come il *find* di un editor o il comando *grep* (quando, ovviamente, si vogliano trovare tutti e soli i cast, di qualunque tipo, ad esempio per contarli come parte di una metrica).

In C++, la situazione è in un certo senso peggiore, poiché il cast tra classi indipendenti, ovvero non legate da legami di ereditarietà, porta in genere a risultati disastrosi; inoltre la semplice forma *(T)expr* può in realtà richiedere operazioni molto diverse:

- una reinterpretazione dei bit del valore di *expr*, come nella conversione puntatore - intero.

- una conversione di tipo aritmetico, ad esempio da int a float.

- operazioni aritmetiche sui puntatori, ad esempio per convertire un puntatore ad una classe derivata da due classi base in un puntatore ad una delle classi base.

- la modifica di attributi come *const* o *volatile*.

- un risultato dipendente dall'implementazione, come il cast ad una classe indipendente.

Il problema maggiore, se il nostro scopo è la chiarezza del codice, è che l'azione che stiamo compiendo non è evidente dal codice stesso; peggio ancora, se cambiano alcune premesse cambia anche l'azione compiuta, senza che il codice debba cambiare o che venga generato un messaggio di errore dal compilatore. Consideriamo il **Listato 165**:

Listato 165

```
class DoubleDerived: public FirstBase, public SecondBase
  {
  // ...
  } ;


void f( DoubleDerived* dd )
  {
  ((SecondBase*)dd)->SomeFunction() ;
  }
```

Nella funzione *f*(), il programmatore vuole chiamare la funzione *SomeFunction*() dell'oggetto puntato da *dd*, garantendosi che venga chiamata la funzione implementata nella classe *SecondBase*. Il codice di per sé è lecito, tuttavia se in seguito cambiamo la definizione di *DoubleDerived*, non ereditando più da *SecondBase*, il codice diventa logicamente errato, senza il minimo messaggio di errore dal compilatore, e con risultati indefiniti a run-time. Notiamo che il costrutto dell'esempio precedente, così comune nella pratica della programmazione, poteva essere implementato senza usare il cast (**Listato 166**).

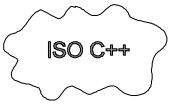
Listato 166

```
class DoubleDerived: public FirstBase, public SecondBase
  {
  // ...
  } ;


void f( DoubleDerived* dd )
  {
  dd->SecondBase::SomeFunction() ;
  }
```

In tal modo, se *DoubleDerived* non fosse derivata da *SecondBase*, il compilatore genererebbe un opportuno messaggio di errore.

Raccomandazione 132

Non usare un cast, di nessun tipo, quando è possible utilizzare l'operatore di qualificazione esplicita ::



Per le sue connotazioni negative, il cast C-style è stato considerato da Stroustrup troppo pericoloso per il C++, oltre ad essere in certi casi (che vedremo tra breve) anche insufficiente a risolvere alcune classi di problemi. Il comitato per lo standard ISO ha pertanto approvato nuovi operatori di cast che rimediano a diverse mancanze del cast C-style:

- esistono operatori diversi per i diversi significati che il cast può assumere.
- hanno una sintassi che li rende facilmente riconoscibili e facili da trovare e contare con tool di ricerca.
- viene eseguito un controllo statico a compile-time quando è sufficiente, o un controllo a run-time quando necessario, sfruttando il nuovo supporto per il Run-Time Type Identification.

- sono in grado di operare correttamente anche in casi in cui il cast C-style normalmente fallisce (ovvero genera un risultato errato).

In quanto segue vedremo i vari operatori ed indicheremo le circostanze di utilizzo.

L'operatore static_cast

Uno degli fini più comuni del cast in C++ è la conversione esplicita da un puntatore a classe derivata ad un puntatore a classe base, come nel **Listato 165**; nel caso specifico, abbiamo suggerito un'alternativa migliore, ovvero l'uso dell'operatore ::. In molti casi il cast da puntatore a classe derivata a puntatore a classe base è invece implicito, ed in seguito si desidera eseguire l'opposto: convertire il puntatore da classe base all'originale classe derivata. In opportuni contesti, ciò è corretto e può essere realizzato anche senza informazioni sui tipi a run-time, ovvero la conversione può essere risolta staticamente a compile-time.

Uno dei casi più noti di conversione base-derivata che non viola il type system è rappresentato dalle liste di oggetti omogenei, realizzate senza template. In tal caso, la lista generica memorizza puntatori ad oggetti di classe base, ed una seconda classe derivata si occupa di inserire (con cast implicito) e recuperare (con cast esplicito) puntatori ad oggetti di classe derivata. Tralasciando per il momento l'opportunità o meno di una simile soluzione, in questo contesto potremmo opportunamente utilizzare l'operatore *static_cast*, come nel **Listato 167**:

Listato 167

```
class Base
  {
  // ...
  } ;


class Derived : public Base
  {
  // ...
  } ;


class StackOfBase
  {
```

```
  public :
    void Push( Base* item ) ;
    Base* Top() ;
    // ...
  } ;


class StackOfDerived : private StackOfBase
  {
  public :
    void Push( Derived* item )
      { StackOfBase :: Push( item ) ; }
    Derived* Top()
      { return( static_cast< Derived* >(
          StackOfBase :: Top() ) ) ; }
    // ...
  } ;
```

In generale, ogni volta che esiste una conversione *implicita* dal tipo A al tipo B, è possibile richiedere il cast statico di B in A (con l'eccezione che il cast statico non può modificare l'attributo *const*). Ovviamente, nel caso precedente, se il puntatore a Base non punta effettivamente ad un oggetto di classe Derived, il risultato è indefinito, non eseguendo static_cast alcun controllo a run-time.

Raccomandazione 133

Se è realmente necessario convertire un puntatore a classe base ad un puntatore a classe derivata, o in generale invertire una conversione implicita, utilizzate l'operatore static_cast e non un cast C-style. Se vi è la possibilità che la conversione fallisca, utilizzate dynamic_cast.

L'operatore const_cast

La conversione da oggetto *const* a oggetto non-const è intrinsecamente pericolosa e raramente necessaria (vedere capitolo 7); per tale ragione, è stato deciso che l'operatore *static_cast* debba conservare l'attributo *const*. Codice come quello del **Listato 168** dovrebbe quindi generare un errore di compilazione; ho scritto dovrebbe, in quanto almeno un compilatore commerciale piuttosto diffuso compila il codice senza problemi.

Listato 168

```
class C
  {
  // ...
  } ;

int main()
  {
  const C constObject ;

  C* nonConstPtr = static_cast< C* >( &constObject ) ;
  // Errore

  return( 0 ) ;
  }
```

Potete pertanto utilizzare il **Listato 168** come verifica dell'aderenza del vostro compilatore al futuro standard. In ogni caso, se dovete necessariamente convertire un oggetto const in uno non-const, utilizzate l'operatore *const_cast*, come nel **Listato 169**.

Listato 169

```
class C
  {
  // ...
  } ;

int main()
  {
  const C constObject ;

  C* nonConstPtr = const_cast< C* >( &constObject ) ;

  return( 0 ) ;
  }
```

Raccomandazione 134

Nei rari casi in cui è realmente necessario convertire oggetti *const* in oggetti *non-const*, utilizzate l'operatore const_cast.

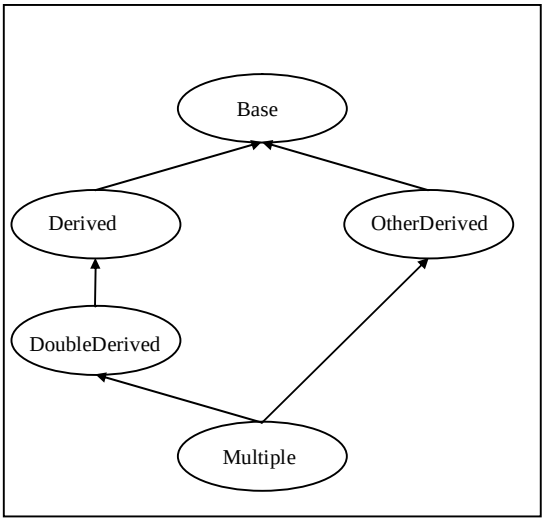
L'operatore dynamic_cast

L'operatore static_cast, così come il cast C-style, si basa esclusivamente su informazioni disponibili a compile-time; in tal senso, la conversione può portare a risultati inattendibili, ad esempio nel caso di ereditarietà multipla: vedere **Listato 170**.

Listato 170

```
class Base
  {
  // ...
  } ;

class Derived : public Base
  {
  // ...
  } ;

void f( Base* p )
  {
  Derived* dp = static_cast< Derived* >( p ) ;
  // ... usa dp ...
  }

class DoubleDerived : public Derived
  {
  // ...
  } ;

class OtherDerived : public Base
  {
  // ...
  } ;

class Multiple : public DoubleDerived, public
OtherDerived
  {
  // ...
  } ;

int main()
  {
  OtherDerived* dd = new Multiple ; // cast implicito
  f( dd ) ;  // cast implicito
  return( 0 ) ;
  } ;
```

Gerarchia di classi



L'esempio è abbastanza complesso e richiede una spiegazione dettagliata: con riferimento alla figura su riportata, dove è rappresentata la corrispondente gerarchia di classi, all'interno di main() creiamo un oggetto di classe Multiple, e richiediamo un cast implicito ad una delle sue classi base (OtherDerived); chiamiamo poi f(), richiedendo un ulteriore cast implicito a Base. Osserviamo che l'oggetto di classe Base che è stato passato ad f() *non è* un sotto-oggetto di Derived: è stato ottenuto dal cast di OtherDerived* a Base*, ovvero seguendo il percorso di destra nella figura. L'operatore static_cast non potrà pertanto dare un risultato corretto: semplicemente non dispone di sufficienti informazioni per convertire un puntatore ad una classe "sorella". Ovviamente, lo stesso accadrebbe utilizzando il cast C-style.

L'unica soluzione percorribile è di avere a disposizione, a run-time, sufficienti informazioni da navigare la gerarchia di classi e modificare opportunamente il puntatore. A tal fine è stato introdotto l'operatore dynamic_cast, avente la stessa sintassi dei nuovi operatori di cast. In accordo al draft ANSI/ISO, in una espressione *dynamic_cast< T >(v), T* deve essere un tipo puntatore o reference ad una classe già definita, oppure void*. Se *T* è un tipo puntatore a classe, e *v* è un puntatore ad una classe di

cui *T* è una classe base *accessibile*, allora il risultato è un puntatore all'unico sotto-oggetto di classe *T*. Analogamente nel caso dei reference. In tutti gli altri casi, ad esempio nel caso di classi "sorelle", *v* deve puntare ad un *tipo polimorfo*, ovvero avente almeno una funzione virtuale[22]. In questo caso, viene eseguito un controllo a run-time per verificare la convertibilità: in caso di fallimento, se si tratta di un cast di puntatori verrà restituito 0, se si tratta di un cast di reference verrà generata un'eccezione Bad_cast.

Notiamo che dynamic_cast può anche essere usato per convertire un puntatore ad una virtual base class in un puntatore ad una classe derivata, operazione impossibile con il cast C-style.

Raccomandazione 135

Nei rari casi in cui sia realmente necessario eseguire un cast che può fallire a run-time, utilizzate l'operatore dynamic_cast, non il cast C-style.

L'operatore reinterpret_cast

Alcuni dei cast espliciti C-style vengono eseguito tra puntatori a tipi base, ad esempio da char* ad int*, oppure tra classi scorrelate, ovvero non appartenenti ad una componente connessa nel grafo di ereditarietà. Tali conversioni sono inerentemente pericolose e dipendenti dall'implementazione, in quanto il valore di un puntatore a char potrebbe non essere valido come valore di puntatore a int (che in un particolare sistema potrebbe, ad esempio, dover assumere solo valori multipli di 32). In questi casi, l'apparente innocenza di un cast C-style può seriamente fuorviare il programmatore nella comprensione del codice: è invece opportuno usare il nuovo operatore *reinterpret_cast*, come nel **Listato 171**.

Listato 171

```
int main()
  {
  char* charBuffer = new char[ 100 ] ;
```

[22] l'esigenza nasce da ragioni implementative: le informazioni run-time sul tipo vengono mantenute in una struttura associata alla virtual-function table dell'oggetto; solo le classi con almeno una funzione virtuale dispongono di una virtual table, quindi sono le uniche a poter beneficiare del controllo a run-time sui tipi. In ogni caso il compilatore dovrebbe segnalare l'uso improprio di tipi non-polimorfi.

```
int* intBuffer = reinterpret_cast< int* >(charBuffer);
// usa intBuffer per accedere a coppie di caratteri,
// assume che un intero sia grande quanto due
// caratteri. Dipendente dall'implementazione!


return( 0 ) ;
}
```

Il risultato di reinterpret_cast è una brutale reinterpretazione degli stessi bit che compongono l'argomento, una esplicita violazione del type system. Uno dei pochi casi in cui può avere senso è per memorizzare temporaneamente un puntatore in "un'altra entità" (spesso un intero o un long) per poi riconvertirlo in seguito; ogni altro impiego è dipendente dall'implementazione.

Raccomandazione 136

Se è realmente necessario eseguire un cast tra tipi scorrelati, utilizzate l'operatore reinterpret_cast, non il cast C-style.

Cast impliciti

In realtà, un cast non è sempre conseguenza di una azione esplicita del programmatore: talvolta, il C++ esegue dei cast impliciti da un tipo ad un altro. Ciò non avviene solo nei casi più banali (ad esempio, la promozione di un *int* in un *long*), ma può coinvolgere anche operatori di conversione definiti dall'utente: notiamo che ogni costruttore con un solo parametro è di fatto anche un operatore di conversione, dal tipo dell'argomento al tipo della classe.

Supponiamo infatti di avere una funzione *f(class)*, e di chiamare la funzione con *f(1)*; se esiste un costruttore con un solo parametro di tipo intero per la classe *class*, il compilatore inserirà un cast implicito da *int* a *class*.
Ciò può essere molto comodo, ma può anche essere fonte di problemi abbastanza sottili, dovuti a possibili ambiguità o alla generazione di oggetti temporanei.

Ambiguità

Se il tipo del parametro attuale di una funzione non è uguale al tipo del'equivalente parametro formale nella dichiarazione, il C++ cerca di eseguire una conversione implicita: ad esempio, un puntatore a classe derivata può essere convertito implicitamente in un puntatore a classe base. Esiste una priorità nella scelta delle conversioni, che tuttavia non è sufficiente a proteggere da eventuali ambiguità: in questo caso, il compilatore segnalerà un errore, cui dovremo porre rimedio con un cast esplicito.
Il problema maggiore dei cast impliciti è che sono poco resilienti rispetto all'evoluzione delle classi: codice che funziona con una certa versione di una classe può generare errori di compilazione (dovuti ad ambiguità) se l'interfaccia della classe viene arricchita con nuove funzioni, anche se le vecchie rimangono inalterate. Un esempio è visibile nel **Listato 172**:

Listato 172

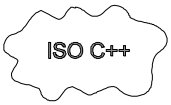
```
class Time
  {
  public :
    Time() ;
    Time( int sec ) ;
    operator int() const ;       // CAUSA AMBIGUITA'
    friend Time operator +( Time t1, Time t2 ) ;
  } ;


int main()
  {
  Time t ;

  t = t + 10 + 20 ;
  // errore: convesione implicita ambigua

  return( 0 ) ;
  }
```

L'errore è causato dalle due possibili conversioni implicite sulla riga indicata: convertire *(t + 10)* in *int* o *20* in *Time*. Notiamo che rimuovendo l'operatore di conversione ad intero dalla classe *Time* il codice viene compilato senza problemi: in effetti, tale operatore potrebbe essere stato aggiunto in fase di manutenzione, rendendo codice preesistente impossibile da compilare (con buona pace dell'incapsulazione e della mantenibilità del codice).

Esistono due buone norme per cautelarsi da simili problemi: limitare il numero di costruttori con un solo parametro (vedi **Raccomandazione 64**), nonché in genere il numero di operatori di conversione, ed evitare il più possibile di appoggiarsi al meccanismo di cast implicito, tranne in casi di stabilità evidente del codice. In questo senso, un cast esplicito in situazioni dubbie (come nel listato precedente) potrebbe essere meglio di un cast implicito, in quanto eviterà problemi in fase di manutenzione del codice.

Raccomandazione 137

In caso di possibile ambiguità, utilizzare un cast esplicito anziché appoggiarsi al meccanismo di cast implicito del linguaggio.



Temporanei

Un effetto della conversione implicita è anche la creazione di un oggetto temporaneo, che viene passato come argomento della funzione. Questo è particolarmente pericoloso se il parametro viene passato tramite reference, perché la funzione modificherà l'oggetto temporaneo e non il parametro attuale. In effetti, questo comportamento è stato escluso dallo standard ANSI/ISO, per cui il compilatore dovrebbe segnalare un errore o perlomeno un warning. Se volete verificare l'aderenza del vostro compilatore al futuro standard, provate a compilare il **Listato 173**: se viene compilato senza messaggi di errore, il compilatore non è ancora stato allineato.

Listato 173

```
void f( int& x )
  {
  x = 3 ;
  }


int main()
  {
  unsigned y = 0 ;
```

```
  f( y ) ;                  // Errore o almeno warning

  return( 0 ) ;
  }
```

Cast di array

Un cast da un puntatore a classe derivata ad un puntatore a classe base non è sempre corretto, anche se verrà sempre accettato dal compilatore: nel **Listato 174**, il codice viene compilato correttamente, in quanto il compilatore effettua un cast implicito da puntatore a Derived a puntatore a Base nella chiamata a GetX. Tuttavia quando la funzione GetX tenta di dereferenziare l'elemento dell'array, essa assume che la grandezza di un elemento sia pari alla grandezza di Base, ed in questo caso non è così. Il risultato dell'esecuzione sarà pertanto totalmente indefinito (potrebbe quindi anche essere '2', ma probabilmente non lo sarà).

Listato 174

```
#include <iostream.h>


class Base
  {
  public :
    Base() { x = 2 ; }
    int GetX() { return( x ) ; }
  private :
    int x ;
  } ;

class Derived : public Base
  {
  public :
    Derived() : Base() { y = 3 ; }
  private :
    int y ;
  } ;


int GetX ( Base* arrayOfBase, int index )
  {
  return( arrayOfBase[ index ].GetX() ) ;
  }
```

```
int main()
  {
  Derived arrayOfDerived[ 10 ] ;
  cout <<  GetX( arrayOfDerived, 9 ) ;

  return( 0 ) ;
  }
```

Ovviamente la causa del problema è la confusione tra “puntatore ad oggetto” e “puntatore al primo elemento di un array di oggetti”, la stessa che ci costringe ad utilizzare delete o delete[] a seconda dei casi. Vale la pena di notare che utilizzando una classe Array, ad esempio basata su template, l’errore di cui sopra sarebbe stato probabilmente intercettato dal compilatore: una ulteriore conferma dell’utilità di classi contenitore evolute.

Raccomandazione 138

Non richiedere mai un cast implicito o esplicito da un array di oggetti di classe derivata ad un array di oggetti di classe base.

13

Varie

"...Where order in variety we see,
And where, though all things differ, all agree."
Alexander Pope

Input/Output

Una delle funzioni più utilizzate del C è certamente *printf*, che per la sua versatilità e completezza appare praticamente in ogni programma, talvolta nella sua versione "su stringa" *sprintf*. Nel passaggio al C++, è naturale conservare l'abitudine di utilizzarla: tuttavia, si dovrebbe preferire l'uso degli stream di I/O ogni volta che ciò sia possibile (ovvero quasi sempre). Vi sono diverse ragioni per scegliere gli stream anziché *printf*:

- Type checking

La funzione *printf* non può effettuare un type checking dei suoi argomenti: si aspetta semplicemente di avere il numero giusto di parametri del tipo corretto, in accordo alla stringa di specifica del formato; pertanto, codice come quello del **Listato 175** verrà compilato senza errori, ma genererà un errore a run-time. L'uso degli stream consente al compilatore di eseguire un type checking sugli argomenti e di evitare così possibili errori a run-time.

Listato 175

```
#include <stdio.h>

int main()
  {
  printf( "%s\n", 1234 ) ;

  return( 0 ) ;
  }
```

- Estendibilità

Quando si definisce una nuova classe, è possibile definire un apposito operatore << per l'output su stream ed un operatore >> per l'input da stream. Tale flessibilità non è disponibile per *printf*. Inoltre i vostri operatori funzioneranno correttamente, ed in modo trasparente, su qualunque classe derivata da istream/ostream: se ottenete nuove classi stream, ad esempio per l'I/O su memory mapped file o su seriale, potrete direttamente utilizzarle con i vostri operatori.

- Velocità

La funzione *printf* contiene al suo interno un mini-parser che interpreta un linguaggio molto semplice (la stringa di formato), interpretazione che viene eseguita a run-time. Nel caso degli stream di I/O, la scelta dell'operatore corretto è eseguita a compile-time, in base al tipo dell'oggetto coinvolto. Se, come spesso avviene, l'I/O è all'interno di un loop, la differenza di velocità può essere sensibile.

Raccomandazione 139

Utilizzare gli stream di I/O anziché printf ovunque sia possibile.

Ouput per una gerarchia di classi

Anche se gli operatori friend non vengono ereditati, è possibile implementare in modo molto semplice un operatore di output che operi correttamente su una intera gerarchia di classi: l'idea di base è definire l'operatore una sola volta, nella classe base, in modo che richiami una funzione virtuale. Ridefinendo la funzione virtuale nelle classi derivate, si può specializzare l'output per le diverse classi. Un esempio è dato nel **Listato 176**:

Listato 176

```
#include <iostream.h>

class Base
  {
  protected :
    virtual ostream& OutOnStream( ostream& os ) const
      { return( os << "Base" ) ; }
```

```
  friend ostream& operator <<(ostream& os,const Base& b)
    { return( b.OutOnStream( os ) ) ; }
  } ;


class Derived : public Base
  {
  protected :
    virtual ostream& OutOnStream( ostream& os ) const
      { return( os << "Derived" ) ; }
      // potrebbe anche richiamare Base :: OutOnStream
      // in casi reali
  } ;


int main()
  {
  Derived d ;
  cout << d ;

  return( 0 ) ;
  }
```

Portabilità

Scrivere codice portabile non è sempre una necessità: talvolta i programmi sono inscindibilmente legati ad una architettura hardware o ad un sistema operativo. In molti casi, tuttavia, solo piccole porzioni del codice sono realmente dipendenti dal sistema, e molte altre potrebbero essere scritte in modo portabile; la portabilità non è molto sentita su personal computer, ma è spesso una esigenza molto forte nella produzione di software pacchettizzato, sia per il contenimento dei tempi di codifica che di testing e debugging.

Una trattazione completa della portabilità del codice esula purtroppo dagli obiettivi di questo libro; molti testi ed articoli sono stati pubblicati sulla portabilità del codice C (si vedano ad esempio [Lap87], [Jae88], [Koe89], [Hor90]), ed il lettore interessato troverà in essi molte informazioni pertinenti anche per lo sviluppo in C++. Ho ritenuto comunque importante presentare brevemente alcune delle tecniche fondamentali per garantire l'indipendenza del codice dall'architettura, pur senza pretendere di essere esaustivo o di giustificare in ogni punto le diverse scelte.

Iniziamo con le regole d'oro della portabilità: non pochi programmatori ritengono che il loro compilatore sia la pietra di paragone, ovvero che se con il loro compilatore un programma è compilato correttamente, allora il programma deve essere corretto anche per altri compilatori e piattaforme. Chiunque abbia una esperienza anche minima di porting, o chi semplicemente si sia fermato ad osservare quanti comportamenti del linguaggio siano "lasciati all'implementazione" o "indefiniti", sa che il singolo compilatore ha ben poco rilievo nel definire la portabilità del codice.
In molti casi, tuttavia, i compilatori possono emettere degli utili warning quando si usano costrutti poco portabili: per questa ragione (ed altre indipendenti dalla portabilità) sarebbe sempre opportuno abilitare *tutti* i warning del compilatore, ed eventualmente utilizzare anche tool esterni *lint*-style; a questo proposito, è importante notare che i warning andrebbero eliminati, in modo che la compilazione sia "silenziosa" il più possibile, ma l'eliminazione deve avvenire nel modo corretto, ovvero rimuovendo la causa del warning. Usare una direttiva #pragma per ignorare il warning non migliorerà la qualità del vostro codice, e neppure spargere cast espliciti ovunque anziché utilizzare uno stile di programmazione più adatto al type checking statico.

Raccomandazione 140

Non assumete che il vostro codice sia portabile verso un'altra piattaforma o semplicemente verso un altro compilatore *solo* perché funziona con il vostro sistema e compilatore.

Raccomandazione 141

Compilate sempre il vostro codice con il massimo numero di warning abilitati; eliminate i warning rimuovendone la causa, non zittendo il compilatore. Evitate l'uso dei #pragma.

Uno dei punti di maggiore scostamento tra le diverse architetture (ma anche tra i diversi sistemi operativi operanti sullo stesso hardware, e talvolta tra i diversi compilatori per lo stesso sistema operativo) è la dimensione dei tipi di dato primitivi. In particolare, bisognerebbe evitare ogni assunzione del tipo "un intero è grande la metà di un long" o "un intero è grande quanto un long". Considerazioni simili sono tipiche quando si cerca di "impaccare" più oggetti piccoli in uno più grande, normalmente a causa di un design poco flessibile; purtroppo, talvolta è il sistema operativo stesso ad

invogliare il programmatore all'uso di tali scorciatoie. Un problema analogo è costituito da routine che sfruttano l'overflow, o l'underflow, di una variabile all'interno del loro normale schema di funzionamento, assumendo di fatto una dimensione ed una rappresentazione prefissata dei tipi base. In questi casi, ammesso che abbia senso parlare di portabilità, è necessario rinunciare all'information hiding e definire dei tipi, possibilmente tramite compilazione condizionale, che espongano direttamente la loro grandezza: così un INT32 conterrà due INT16 su ogni architettura, ma su una di esse potranno essere rispettivamente un *long* ed un *int*, su un'altra un *int* ed uno *short*. Notiamo che questa pratica *non* è sempre consigliabile, anzi va usata solo nei casi di reale necessità: in molti casi il software è abbastanza indipendente dal range dei tipi base da non creare seri problemi di porting, specialmente se si programma con la dovuta cura. Solo nei casi in cui è necessario conoscere realmente le dimensioni dei singoli tipi è opportuno definirli in modo che la dimensione stessa sia visibile in modo esplicito, ed in modo tale che cambiando la definizione si possa portare il software su un'altra architettura.

Un altro errore molto comune è l'assunzione che puntatori ed interi (o long) abbiano la stessa dimensione, il che è falso in molte situazioni; poiché è lecito sottrarre due puntatori, esiste un tipo standard per l'intero con la minima precisione necessaria a memorizzare tale differenza: ptr_diff. Analogamente, la dimensione degli oggetti dovrebbe sempre essere nel range del tipo size_t (anche se in alcuni compilatori ciò non è vero). Non assumete che ptr_diff sia *int*, in quanto su alcune architetture deve essere *long*.

Raccomandazione 142

Non fate assunzioni arbitrarie sulle grandezze dei tipi, specialmente sulle grandezze relative di short, int, long e puntatori. Usate i tipi standard per contenere le differenze tra puntatori. Se è necessario fare riferimento alla dimensione di un tipo base, definite un apposito tipo utilizzando la compilazione condizionale.

Subito dopo la dimensione dei tipi, i problemi più comuni di portabilità vengono da assunzioni circa il layout degli oggetti in memoria. Problemi comuni sono il cast di puntatori a tipi base diversi (es. da char* a long*) per trattare stream di byte senza curarsi del loro tipo. Codice simile non funziona correttamente su architetture che assumono, ad esempio, che l'indirizzo di ogni long sia un multiplo di 32. Altro ben noto problema ha origine nella diversa posizione dei byte più significativi (little-endian o big-

endian): problemi di portabilità di questo tipo sono sempre dovuti alla manipolazione di elementi composti come se fossero sequenze di byte, o al tentativo di manipolare sequenze di byte come se fossero oggetti composti. Salvo particolari cautele e l'uso di compilazione condizionale, si tratta sempre di codice non portabile. Analoghe considerazioni valgono per i bit-fields: il loro uso è portabile solo se non si tenta mai di concatenare più di un campo per formare una singola unità.

Ricordate che non solo la grandezza, ma anche i singoli valori possono variare da una architettura ad un'altra: ad esempio, inizializzare un array di puntatori o di float con *memset* (ad esempio per mettere i puntatori a NULL o i float a 0) non è portabile, perché la rappresentazione bit-wise di NULL o di 0.0 può essere diversa da una sequenza di bit a posti a 0, ed essere diversa su macchine diverse.

Raccomandazione 143

Non assumete un particolare layout di memorizzazione per i tipi base.

Raccomandazione 144

Se usate i bit-field, non trattate mai una sequenza di campi come una singola unità.

Per quanto si tratti di un problema di portabilità più raro, talvolta il codice assume un particolare layout per i singoli oggetti di una classe: ad esempio, che i primi 32 bit siano un puntatore alla tavola delle funzioni virtuali. Codice simile è inerentemente legato non solo all'architettura, ma alla particolare versione del compilatore e talvolta anche alle opzioni di compilazione. Può essere necessario, in casi estremi, fare riferimento a simili dettagli implementativi; tuttavia, l'unica assunzione portabile è che in assenza di metodi virtuali, il layout di una classe (o di una struct) sia esattamente lo stesso che in C, ovvero una sequenza dei singoli campi con o senza byte di padding, in funzione del compilatore. Ogni altra assunzione sul layout degli oggetti è totalmente non portabile.

Raccomandazione 145

Non fate assunzioni sul layout in memoria degli oggetti.

Per ordine di frequenza, la prossima categoria di problemi di portabilità riguarda l'aritmetica sui puntatori. Errori tipici sono la comparazione con == o != di puntatori ad oggetti eterogenei, o l'uso (ad esempio) di < per comparare puntatori che non puntano all'interno dello stesso array. Gli operatori == e != sono definiti solo tra puntatori allo stesso tipo, e < , <=, >, >=, solo per puntatori che puntano all'interno dello stesso array, o al primo elemento dopo la fine dell'array stesso.

Raccomandazione 146

Non comparate puntatori a tipi diversi. Gli operatori <, <=, >, >= sono definiti solo per puntatori che puntano all'interno dello stesso array, o al primo elemento dopo la fine dell'array stesso.

L'ultima categoria di problemi di portabilità che prenderemo in considerazione riguarda l'ordine di esecuzione: si tratta in effetti di un errore che i programmatori esperti commettono di rado, in quanto assunzioni sull'ordine di valutazione degli operandi sono prive di senso in quasi ogni linguaggio. La particolarità del C e del C++ riguarda soprattutto gli operatori con side-effect (++, --): ogni espressione dove lo stesso identificatore appaia come argomento di un operatore con side effect *e* in un qualunque altro punto dell'espressione, ha risultato indefinito.

Raccomandazione 147

Non assumete che la chiamata di funzione comporti la valutazione degli argomenti in un ordine particolare.

Raccomandazione 148

Gli operandi di un operatore con side-effect non devono avere altre occorrenze nella stessa espressione.

14

Brevi note di Design Dettagliato

"...As if the design of all his words takes form
and frame from thinking and is realized."
Wallace Stevens

Scopo di questo libro è la stesura di un insieme di raccomandazioni e regole di codifica per il linguaggio C++: argomenti come il design architetturale di una applicazione complessa, o il design dettagliato della singola classe, esulano in larga misura dal contenuto previsto. Sono tuttavia argomenti molto interessanti, in quanto uno dei fondamentali vantaggi delle metodologie object oriented è il *continuum* tra analisi, design architetturale, design dettagliato, ed implementazione; pertanto, anche se una trattazione approfondita di tali argomenti richiederebbe un intero volume, presenterò di seguito alcune importanti considerazioni di design dettagliato, riguardanti argomenti talmente generali da essere rilevanti, in pratica, anche al momento della codifica.

Ereditarietà o Contenimento

Uno degli errori più comuni (a mio avviso l'errore piu comune) nella programmazione in C++ è l'uso dell'ereditarietà per modellare relazioni tutto-parti. L'errore viene più spesso commesso da programmatori che prestano scarsa attenzione ad argomenti "accademici" come il design, ed utilizzano i costrutti del linguaggio per quello che *fanno*, non per quello che *rappresentano*: in effetti, l'ereditarietà (singola o multipla) ha come effetto pratico quello di creare un'unione di oggetti che, ragionando ad un livello di astrazione troppo basso, può essere indistinguibile dal contenimento.

Alcuni programmatori sono addirittura convinti che l'ereditarietà sia più efficiente del contenimento diretto, perché non devono scrivere il nome del membro prima di richiamare la funzione: una classica reminiscenza dello stile (nuovamente errato) "codice più corto significa codice più efficiente" così popolare tra i programmatori C. In effetti, salvo casi di compilatori

inefficienti, non vi è alcuna differenza tra chiamare un metodo di una classe base o un metodo di un oggetto contenuto direttamente: in entrambi i casi, sarà necessario un "riaggiustamento" di *this* per ottenere un puntatore al sotto-oggetto, sia esso incluso tramite ereditarietà che tramite contenimento diretto.

Al di là delle leggende sull'efficienza, la ragione principale per l'abuso di ereditarietà sta nella scarsa attenzione agli aspetti semantici più astratti della gerarchia di classi; vi sono indubbiamente casi in cui è difficile decidere se sia più opportuno l'uso dell'una o dell'altra (quasi sempre il dubbio si può fugare considerando possibili evoluzioni future, e quasi sempre preferendo il contenimento), ma in generale dovremmo sempre osservare i seguenti punti:

- L'ereditarietà pubblica o protetta va usata solo per modellare relazioni *Is-A* (è-un). Ciò significa che non dobbiamo mai creare (attraverso ereditarietà multipla) due percorsi *accessibili* alla stessa base. Come abbiamo visto nel capitolo 10, quando è possibile l'ereditarietà pubblica o protetta dovrebbe essere anche virtuale.

L'ereditarietà privata può essere usata per modellare una relazione *Has-A* (ha-un): tuttavia come abbiamo visto nel capitolo 10 esistono in tal caso dei seri problemi dovuti alle funzioni virtuali. Dal mio punto di vista, l'ereditarietà privata dovrebbe essere limitata ai casi in cui la relazione è effettivamente del tipo *Is-A*, ma lo è per ragioni implementative, che non si vogliono rendere note, e che potrebbero in seguito suggerire di modificare la derivazione, ereditando da un'altra classe o usando il contenimento. Si tratta quindi di un insieme di casi più ristretto rispetto all'uso per modellare relazioni *Has*-A; un esempio tipico è la derivazione pubblica da una classe virtuale *Stack*, che definisce l'interfaccia, unitamente alla derivazione privata da una classe *Array*, che definisce una particolare implementazione, per ottenere una classe *FixedSizeStack*. In questo caso, un *FixedSizeStack* è a tutti gli effetti anche un *Array*, solo che non vogliamo renderlo noto ai clienti della classe: potremmo decidere di modificarne l'implementazione in un secondo tempo.

Raccomandazione 149

Non utilizzate l'ereditarietà per modellare relazioni del tipo tutto-parti, che vanno modellate con il contenimento.

Contenimento diretto o tramite puntatori

Supponiamo di avere a disposizione una classe Engine (motore) ed una classe Wheel (volante) e di voler costruire una classe Car (automobile) componendo le due classi (ovviamente un'automobile ha altri componenti che ignoreremo); tralasciando la possibilità (alquanto discutibile) di usare l'ereditarietà multipla, si hanno a disposizione due scelte fondamentali, ovvero "immergere" i sotto-oggetti direttamente nella classe composta (**Listato 177**), o immergerli attraverso puntatori (**Listato 178**).

Listato 177

```
// composizione diretta

class Car
  {
  Engine engine ;
  Wheel wheel ;
  // ...
  } ;
```

Listato 178

```
// composizione tramite puntatori

class Car
  {
  Engine* engine ;
  Wheel* wheel ;
  //...
  } ;
```

Il C++ è tra l'altro uno dei pochi linguaggi object oriented a permettere la prima soluzione: molti altri linguaggi, anche blasonati, consentono solo la composizione indiretta, tramite reference.

Esiste una diffusa opinione che la composizione diretta sia più efficiente in termini di velocità (ed in effetti è vero), tuttavia apparentemente pochi programmatori sanno individuare la *vera* ragione della maggiore efficienza (che non è, come si potrebbe immaginare, motivata dal non dover dereferenziare i puntatori).

Vediamo in ogni caso quali sono i reali vantaggi della composizione tramite puntatori:

- i sotto-componenti possono essere istanziati in un qualunque momento, anche se non hanno un costruttore di default ed i parametri necessari a costruirli non sono immediatamente disponibili. Questa è in realtà una istanza di un aspetto più generale, dato al punto seguente.

- tramite puntatori possiamo implementare relazioni dinamiche di tipo *1-n* tra il tutto e le parti; quando *n* è uguale a 0 in un certo momento nella vita dell'oggetto, ricadiamo nel caso precedente.

- tramite puntatori possiamo condividere lo stesso sotto-componente con altri oggetti; attenzione comunque a non abusare di questa tecnica senza le opportune precauzioni, come un reference counter nell'oggetto puntato. Al di là delle giuste cautele, questa è forse la ragione più importante per comporre tramite puntatori.

- talvolta è impossibile dichiarare un oggetto come automatico se la sua dimensione è troppo grande, tale da eccedere lo stack disponibile (ciò dipende in genere dal compilatore e dal sistema operativo, ma è una situazione abbastanza comune ad esempio su personal computer). Se le parti dell'oggetto sono allocate con l'operatore *new*, e quindi accedute tramite puntatori, il problema solitamente non sussiste (tranne nell'ovvio caso in cui non si abbia sufficiente memoria dinamica per l'oggetto).

Esistono quindi delle occasioni in cui l'uso dei puntatori è consigliabile o indispensabile: ad esempio, all'interno di classi "wrapper" che utilizzano artifici come il reference count per rendere trasparenti l'allocazione/deallocazione dinamica; in tutti gli altri casi, tuttavia, l'immersione diretta è da preferire. Al di là di un trascurabile risparmio di memoria (non dovendo memorizzare i puntatori, ma solo gli oggetti) i vantaggi dal punto di vista dell'efficienza sono i seguenti:

- L'accesso ai sotto-componenti non deve passare attravero il livello di indirezione creato dai puntatori. Per quanto in un utilizzo intensivo dei sotto-componenti questo possa degradare le prestazioni, il reale guadagno non è in genere molto sensibile.

- Non è necessario allocare/deallocare dinamicamente la memoria. Nel caso in cui l'oggetto contenitore sia automatico, di vita breve, questo può effettivamente portare a significative variazioni delle prestazioni. Non è insolito che applicazioni C++ siano fortemente "allocation-

bounded", ovvero che l'allocazione dinamica della memoria giochi un ruolo centrale nella performance dell'applicazione, anche a causa della frammentazione.

- Se non usiamo i puntatori, il compilatore conosce l'*esatta* classe del sotto-componente; se usiamo i puntatori, il compilatore conosce solo una classe base per i sotto-componenti. Il risultato è che non usando i puntatori, il compilatore può effettivamente espandere le funzioni inline virtuali per i sotto-componenti, mentre usando i puntatori ciò non è possibile. La differenza tra una chiamata di funzione virtuale ed una funzione espansa in linea è molto significativa, in genere tale da spiegare da sola la differenza di prestazioni tra l'uso dei puntatori o l'immersione diretta.

Esiste un ulteriore vantaggio, non direttamente legato all'efficienza, della soluzione per contenimento diretto, nel caso si usi il meccanismo di exception handling:

- Non usando i puntatori, non si deve gestire la deallocazione dei sotto-oggetti in caso di fallimento all'interno del costruttore: il meccanismo di gestione delle eccezioni richiamerà l'opportuno distruttore per i sotto-oggetti già costruiti.

Raccomandazione 150

Quando possibile, utilizzate la composizione diretta per i sotto-oggetti, non la composizione tramite puntatori.

Static o dynamic typing

Molti sviluppatori arrivano alla comprensione di un paradigma di programmazione attraverso lo studio e la pratica di un linguaggio che si basa su (o che supporta) quel paradigma. È raro che avvenga l'opposto, ovvero che il paradigma di programmazione venga studiato *in sé*, e che un linguaggio abbia un ruolo solo strumentale nella sperimentazione del paradigma: anche a livello accademico, non di rado si assume che lo studente sappia estrapolare il paradigma dallo studio di uno o più linguaggi. Purtroppo, questo processo induttivo è molto complesso, e si rischia facilmente di considerare alcune caratteristiche -o limitazioni- del linguaggio come caratteristiche del paradigma; così chi pensa di discutere di "programmazione strutturata" discute invece spesso di Pascal, e chi parla di OOP finisce spesso per parlare di Smalltalk o di C++.

In realtà questi due linguaggi, pur essendo entrambi orientati agli oggetti, hanno una storia, una filosofia di progetto, e di riflesso una cultura pragmatica molto diversa; uno dei punti di maggiore distacco è proprio il type checking, che in Smalltalk è eseguito a run-time, mentre in C++ esiste un type checking statico, ma che può essere bypassato dal programmatore.

Proprio questa caratteristica ha portato, specialmente nei primi tempi, ad utilizzare il C++ "alla Smalltalk": tutto sommato, per molti Smalltalk *era* (e per molti è tuttora) l'incarnazione stessa del paradigma object oriented, e quindi se Smalltalk eseguiva il type checking a run-time, non poteva esserci nulla di male nel fare lo stesso in C++.

In realtà, il C++ non supporta un vero type checking dinamico, anche se le nuove versioni includono informazioni sui tipi a run-time: il programmatore deve dapprima violare il type-system attraverso dei cast espliciti, ed in seguito verificare, sotto la propria responsabilità, la correttezza delle conversioni di tipo.

Uno schema classico è il seguente: ogni classe viene derivata da una radice *Object*, ed un puntatore ad ogni classe può quindi essere convertito in un puntatore ad *Object*; usando l'RTTI, o con vecchi compilatori definendo un metodo virtuale *Kind()*, si eseguono poi dei cast espliciti dalla classe base alla classe derivata che si suppone corretta per l'oggetto. Molte delle prime implementazioni delle classi contenitore, in mancanza dei template, seguivano una strategia di questo tipo, con vari meccanismi per cercare di tutelarsi da possibili violazioni del type system.

Proprio i template hanno radicalmente rimosso la necessità di simili acrobazie: il C++ basa ora la genericità sui template, che consentono il type-checking statico, non sull'ereditarietà, che richiede un type checking dinamico. Ciò non significa che il type checking dinamico in sé non sia adeguato[23]: significa solo che se volete un linguaggio con type checking a run-time, state sbagliando ad usare il C++.

Il beneficio maggiore del type checking statico in C++ è la riduzione del numero di errori che possono essere trovati solo a run-time: un argomento che dovrebbe avere la massima rilevanza per chi sia arrivato sino a questo punto del testo. Non solo, il type checking dinamico richiede l'introduzione forzata di una gerarchia di classi con un grafo connesso, dove ogni classe è-*un Object*. Ciò richiederebbe (vedere capitolo 10) un uso forzato dell'ereditarietà virtuale, ed in alcuni casi si passerebbe dall'ereditarietà virtuale indipendente a quella fork-join, con tutte le complicazioni del caso. I template non hanno alcuno di questi problemi, e se usati con le accortezze viste al capitolo 11 permettono di ottenere codice snello quanto quello basato sull'ereditarietà, spesso più efficiente, e senza violazioni del type system.

Raccomandazione 151

Non utilizzate schemi di type-checking dinamico per ottenere genericità: preferite invece l'uso dei template.

Isolamento

Nel capitolo 3 abbiamo accennato ad alcune tecniche per minimizzare i tempi di ricompilazione a fronte di modifiche. In molti casi, tuttavia, l'accoppiamento logico tra le classi (attraverso l'ereditarietà o il contenimento diretto) si riflette comunque in un accoppiamento fisico tra le implementazioni: modificare la struttura di una classe base o di una classe contenuta richiede la ricompilazione delle classi derivate o contenenti, anche se la modifica riguarda solo la parte privata della classe. La ricompilazione tende poi a propagarsi in modo combinatorio agli altri file, in mancanza di precauzioni adeguate.

[23] anche se molti, tra cui l'autore, ritengono che tutti i controlli eseguibili a compile-time *debbano* essere eseguiti a compile-time, e questo include il type checking.

In progetti molto grandi è necessario minimizzare non solo le dipendenze logiche, ma anche quelle fisiche: il termine che si usa in questo caso è *isolamento* delle classi, ovvero un meccanismo che non richieda la ricompilazione dei "clienti" di una astrazione (classe) a seguito di una modifica ad un dettaglio implementativo di tale astrazione. La necessità si presenta soprattutto per le classi che saranno più soggette a manutenzione nella loro struttura interna, pur mantenendo una interfaccia inalterata.

In C++ possiamo ottenere l'isolamento delle classi in diversi modi: il più banale, che tuttavia abbiamo sconsigliato pocanzi in questo stesso capitolo, si applica solo al caso del contenimento diretto, ed implica la trasformazione in contenimento indiretto. Come abbiamo visto nel capitolo 3, in questo caso non è necessario includere l'header della classe contenuta nell'header della classe contenente.

Tale soluzione è anche poco elegante, poiché richiediamo che il meccanismo di isolamento sia attuato dai clienti della classe: poiché la necessità di isolamento è invece una caratteristica della classe stessa, sarebbe molto più corretto se l'isolamento risultasse dalla struttura della classe, senza la cooperazione dei clienti.

Esistono a tal fine due tecniche, la prima delle quali è una evoluzione della strategia banale vista sopra. Entrambe richiedono la definizione di una nuova classe, il cui compito è di isolare i clienti dalla parte privata della classe originaria, che potrà quindi cambiare senza necessità di ricompilazione.

Data wrapping

La prima tecnica consiste nel creare una classe con una interfaccia identica alla classe base, e contenente un puntatore ad un oggetto di classe base. Ogni chiamata di funzione si risolve in un semplice forwarding della chiamata stessa attraverso il puntatore. In questo senso, è una evoluzione della tecnica banale di cui sopra, attuata tuttavia "lato server" anziché "lato client": i clienti della classe continuano ad usare il contenimento diretto o l'ereditarietà, senza alcuna conoscenza del meccanismo di isolamento. Di norma si esegue un renaming, cosicché la classe visibile ai clienti mantiene lo stesso nome mentre alla classe base si aggiunge (ad esempio) un suffisso "Data", in quanto è l'unica a manipolare direttamente i dati. Un esempio è visibile nel **Listato 179**:

Listato 179

```
// PERDATA.H

class PersonData
  {
  private :
    String firstName ;
    String lastName ;
    int height ;          // potrebbe cambiare in float
    int weight ;          // potrebbe cambiare in float
  public :
    PersonData() ;
    void Store() ;
    void InteractiveInput() ;
  } ;



// PERSON.H

class PersonData ;

class Person
  {
  private :
    PersonData* data ;
  public :
    Person() ;
    void Store() ;               // forward
    void InteractiveInput() ;    // forward
  } ;



// CLIENT1.H

#include “person.h”

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



// CLIENT2.H
```

```
#include "person.h"

class Car
  {
  private :
    Person owner ;
  // ...
  } ;
```

La classe *Person* gioca semplicemente il ruolo di classe di isolamento, ed esegue un forward di tutte le chiamate alla classe *PersonData*; notiamo che i clienti usano *Person* senza limitazioni, con ereditarietà o contenimento diretto, in quanto non devono avere conoscenza della strategia di isolamento. Se un dettaglio implementativo (ma non l'interfaccia pubblica) della classe originaria *Person* (ora *PersonData*) viene modificato, solo *PersonData* e la parte implementativa di *Person* devono essere ricompilate, ma non le classi client.

Classi interfaccia

La seconda tecnica non richiede il forwarding di funzioni attraverso un puntatore, ma si basa su una classe astratta che fa da interfaccia verso i clienti, e su una classe concreta da essa derivata per fornire l'implementazione. La classe astratta stabilisce il protocollo che i clienti possono utilizzare e che la classe concreta si impegna ad implementare: ogni dettaglio implementativo è però isolato nella classe concreta. Per isolare realmente le classi cliente, è però necessario che nell'header della classe interfaccia venga anche definita una funzione con il compito di creare oggetti di classe concreta (questa funzione viene normalmente detta class-factory): torneremo su questo punto dopo aver esaminato una versione con classe interfaccia dell'esempio precedente, visibile nel **Listato 180**:

Listato 180

```
// PERSON.H

class Person
  {
  public :
    virtual void Store() = 0 ;
```

```
    virtual void InteractiveInput() = 0 ;
  } ;


Person* CreatePerson() ;
// class factory; potrebbe essere un membro statico di
Person



// PERIMPL.H

class PersonImpl : public Person
  {
  private :
    String firstName ;
    String lastName ;
    int height ;          // potrebbe cambiare in float
    int weight ;          // potrebbe cambiare in float
  public :
    PersonImpl() ;
    virtual void Store() ;
    virtual void InteractiveInput() ;
  } ;



// PERSON.CPP

#include “perimpl.h”

Person* CreatePerson()
  {
  return( new PersonImpl() ) ;
  }




// CLIENT2.H

#include “person.h”

class Car
  {
  private :
    Person owner ;
  // ...
```

```
  } ;



// CLIENT2.CPP

// qui per creare una persona non si usa
//  new Person() che e' illegale, ma CreatePerson()
```

La tecnica è piuttosto semplice ma merita comunque qualche commento: la classe originaria *Person* viene trasformata in una virtual base class, il cui unico compito è di definire l'interfaccia per i clienti. L'implementazione vera e propria è fornita da una classe separata, *PersonImp*, della quale i clienti non sono neppure a conoscenza: in tal modo, i dettagli implementativi possono cambiare senza necessità di ricompilazione, poiché i clienti non includono neppure l'header della classe implementazione. Resta però il problema di creare delle persone, che non possono essere istanza di *Person* (che è una classe astratta), ma devono necessariamente essere istanze di *PersonImp*: a tal fine, l'header di *Person* esporta una funzione, detta class factory, che sostituisce l'operatore new e restituisce un puntatore a *Person* (in realtà un puntatore a *PersonImp*). Vi sono due note importanti a proposito della tecnica in questione:

1. Si può utilizzare nel caso i clienti usino il contenimento diretto ma non la derivazione: se dovessero ereditare da *Person* nello schema originario non isolato, con la tecnica della classe interfaccia dovrebbero ereditare da *PersonImp*, vanificando l'isolamento. È in realtà possibile mantenere due gerarchie distinte, una per le interfacce ed una per le implementazioni, il che consente l'uso della tecnica anche con la derivazione; occorre comunque una gestione molto disciplinata per non rendere nulli i vantaggi in termini di tempo di compilazione.

2. Poiché in principio si possono avere più class factory, possiamo avere diverse implementazioni per la stessa interfaccia, anche scelte a run-time. Un meccanismo simile, accoppiato con l'uso di librerie a caricamento dinamico, è alla base dei componenti detti in-process server di OLE 2.

Raccomandazione 152

Se i tempi di ricompilazione dopo la modifica di parti private delle classi sono troppo lunghi, minimizzate l'accoppiamento tra le implementazioni delle classi usando una tecnica di isolamento.

Bibliografia

[Ada94] Robert Adams: “Temporary Object Management Through Dual Classes”, C/C++ User’s Journal, May 1994.

[AL88] J.Aerts, J.Langhout: “C Coding Standards (Release 1.1)”, Nederlandse Philips, Centre for Software Technology, 1988.

[Ana88] N. Anand: "Clarify Function!" ACM SigPLAN Notices, Vol 23 No 6, 1988.

[ANS95] X3J11 ANSI Committee: “Working Paper for Draft Proposed American National Standard for Information Systems -- Programming Language C++”, 1995.

[Ben92] Johan Bengtsson: “C++, Without Exceptions”, Telia Research, Sweden, 1992.

[Bie91] J. M. Bieman: “Deriving Measures of Software Reuse in Object Oriented Systems”, Colorado State University, Technical Report, 1991.

[BM90] Backer, Marcus: “Human Factors and Typography for More Readable Programs”, Addison-Wesley, 1990.

[BO94] James M. Bieman, Linda M. Ott: “Measuring Functional Coesion”, IEEE Transactions on Software Engineering, Vol 20 No 6, 1994.

[Bor90] Alexander Borgida: “Modelling Class Hierarchies with Contradictions”, Rutgers University, Technical Report, 1990.

[Bro80] R. Brooks: "Studying Programmer Behavior Experimentally: The Problems of Proper Methodology," Communications of the ACM, Vol 23 No 4, 1980.

[BTBWS] Bell Labs, University of Toronto, UC Berkeley, University of Washington, Softquad Incorporated: “Recommended C Style and Coding Standards”, Techical Report.

[BZ88] Buhr, Zarnke: “Nesting in an object oriented language is not for the birds”, Proceedings of ECOOP’88 Conference.

[Car92] Thomas Cargill: “C++ Programming Style”, Addison-Wesley, 1992.

[CCL94] Carneiro, Covan, Lucena: “A rationale for both Nesting and Inheritance in Object-Oriented Design”, Technical Report, University of Waterloo, Canada, 1994.

[CGL94] Cowan, Germán, Lucena, von Staa: “Enhancing Code for Readability and Comprehension Using SGML”, Technical Report, University of Waterloo, Canada, 1994.

[CL90a] Marshall P. Cline, Doug Lea: “The Behaviour of C++ Classes”, Proceedings of the Symposium on Object Oriented Programming Emphasizing Practical Applications, Marist College, 1990.

[CL90b] Marshall P. Cline, Doug Lea: “Using Annotated C++”, Technical Report, Clarkson University, 1990.

[Cli95] Marshall P. Cline: “Frequently-Asked-Questions for comp.lang.c++”, USENET, 1995.

[CM90] Cheatham, Mellinger: “Testing Object Oriented Software Systems, Proceedings of the 1990 ACM SCS Conference”, 1990.

[Cop92] James O. Coplien: “Advanced C++ Programming Styles and Idioms”, Addison Wesley, 1992.

[CSD86] Conte, Dunsmore, Shen: “Software Engineering Metrics and Models”, Benjamin/Cummings, 1986.

[DC85] Ian Darwin, Geoff Collyer: “Can’t Happen or /* NOTREACHED */ or Real Programs Dump Core”, Usenix Conference, 1985.

[DK76] Frank DeRemer, Hans H. Kron: "Programming-in-the-Large Versus Programming-in-the-Small", IEEE Transactions on Software Engineering, Vol 2 No 2, June 1976.

[Ede92] Daniel Edelson: “Smart Pointers: They’re Smart, but They’re Not Pointers”, Proceedings of the 1992 Usenix C++ Conference, Usenix Association.

[ES90] Margareth Ellis, Bjarne Stroustrup: “The Annotated C++ Reference Manual”, Addison Wesley, 1990.

[Gol87] Adele Goldberg: “Programmer as a Reader”, IEEE Software Vol 4 No 5, 1987.

[Hau93] Franz J. Hauck: “Inheritance modeled with explicit bindings: an approach to typed inheritance”, Proc. OOPSLA 1993, SIGPLAN Notices Vol 28 No 10, ACM, 1993.

[Hen88] S. Henry: "A Technique for Hiding Proprietary Details While Providing Sufficient Information for Researchers; or, do you Recognize this Well-known Algorithm?" Journal of Systems and Software, Vol 8 No 1, 1988.

[Hor90] Mark Horton: " Portable C Software ", Prentice-Hall, 1990.

[Jae88] Rex Jaeschke: “Portability and the C language”, Hayden Books, 1988.

[Jal89] P. Jalote: “Functional refinement and nested objects for object-oriented design”, IEEE Transactions on Software Engineering, Vol 15, 1989.

[Jon86] Capers Jones: “Programming Productivity”, McGraw-Hill, 1986.

[Joy92] Ian Joyner: “C++?? A Critique of C++”, Internal Report, UNYSIS - ACUS, 1992.

[Kel90] Daniel Keller: “A Guide to Natural Naming”, ACM SigPLAN Notices Vol 25 No 5, 1990.

[Ken91] Brian Kennedy: “The features of the object-oriented abstract type hierarchy (OATH), Proceedings of the 1991 Usenix C++ Conference, Usenix Association.

[Kla80] R. Klatzky: “Human Memory”, Second Edition, Freeman & Co., 1980.

[Koe89] Andrew R. Koenig: “C Traps and Pitfalls”, Addison-Wesley, 1989.

[KP78] Brian W. Kernighan, P. J. Plauger: “The Elements of Programming Style”, Second Edition, McGraw-Hill, 1978.

[KS81] Kosslyn, Shwartz: “Empirical Constraints on Theories of Visual Mental Imagery”, Attention and Performance vol. IX, Erlbaum, 1981.

[Lak92] Al Lake: “A Software Complexity Metric for C++. Annual Oregon Workshop on Software Metrics, Silver Falls, Oregon, 1992.

[Lak93] John S. Lakos: "Designing-In Quality in Large C++ Projects", 10th Annual Pacific Northwest Software Quality Conference, Portland, Oregon, 1993.

[Lap87] J.E. Lapin: “Portable C and UNIX System Programming”, Prentice-Hall, 1987.

[LH92] Karl J. Lieberherr, Ian M. Holland: “Assuring Good Style for Object-Oriented Programs”, IEEE Software, September 1989. [Una versione aggiornata ed estesa è disponibile come Technical Report, Northeastern University, College of Computer Science, 1992].

[LHR88] Lieberherr, Holland, Riel: “Object-Oriented Programming: An objective sense of style”, SIGPLAN Notices Special Issue, 1988.

[Lis87] Barbara Liskov: “Data Abstraction and Hierarchy”, Addendum to Proceedings of OOPSLA ‘87, October 1987.

[LV89] Lind, Vairavan: “An Experimental Investigation of Software Metrics and Their Relationship to Software Development Effort”, IEEE Transactions of Software Engineering, Vol 15 No 5, 1989.

[Mad87] O. L. Madsen: “Block structure and object oriented languages”, in Research Directions in Object Oriented Languages, MIT Press, 1987.

[Mah90] M. L. Maher: “Process Models for Design Synthesis”, AI Magazine, Winter 1990.

[Mar92] Brian Marik: “A question catalog for code inspections”, Testing Foundations Group, University of Illinois at Urbana-Champaign, 1992.

[McC76] T.J. McCabe, A Complexity Measure, IEEE Transactions on Software Engineering, Vol 2 No 4, Dec. 1976.

[Mey88] Bertrand Meyer: “Object Oriented Software Construction”, Prentice Hall, 1988. Tradotto in Italiano come “La produzione del software object oriented”, Gruppo Editoriale Jackson, 1991.

[NH92] Erik Nyquist, Mats Henricson: “Programming in C++ Rules and Recommendations”, Ellemtel Telecommunication System Laboratories, 1992.

[Par72] David L. Parnas: “On the Criteria to be Used in Decomposing Systems into Modules”, Communications of the ACM Vol 15 No 2.

[Pau93] Mark C. Paul: “Comparing ISO 9001 and the Capability Maturity Model for Software”, Software Quality Journal, July 1994.

[Pes97] Carlo Pescio: “Return Value Optimization, Named Value Optimization e Costruttori Operazionali”, C++ Informer No. 1, Dicembre 1997.
http://www.eptacom.net/pubblicazioni/pub_it/iso_1.html

[Pes00] Carlo Pescio: “Underscore: cosa è legale, cosa no, e perchè”, C++ Informer No. 10, Gennaio 2000.
http://www.eptacom.net/pubblicazioni/pub_it/iso_10.html

[Plu84] Thomas Plum: “C Programming Guidelines”, Prentice-Hall, 1984.

[Ray91] Darrel Raymond: “Reading Source Code”, Proceedings of the 1991 CASCON, October 1991.

[Ros94] D. S. Rosenblum: “A Practical Approach to Programming With Assertions”, IEEE Transactions on Software Engineering, Vol 21 No 1, 1994.

[RW90] Charles Rich, Richard Waters: “The Programmer’s Apprentice”, ACM Press Frontiers Series, 1990.

[Sak88] Markku Sakkinen: “On the Darker Side of C++”, ECOOP’88 Proceedings, Springer-Verlag 1988 (LCNS 322).

[Sak92a] Markku Sakkinen: “A Critique or the Inheritance Principles of C++”, Computing Systems Vol. 5 No 1 Winter 1992.

[Sak92b] Markku Sakkinen: “The Darker Side of C++ Revisited”, Structured Programming, 1992.

[SB91] Selby, Vasili: “Analizyng Error-Prone System Structure, IEEE Transactions of Software Engineering, Vol 17 No 2, 1991.

[SBE83] Eliot Soloway, Jeffrey Bonar, Kate Ehrlich: “Cognitive Strategies and Looping Constructs: An Empirical Study”, Communications of ACM Vol 26 No ``, November 1983.

[SC94] N. R. Saxena, E.J. McCluskey: “Linear Complexity Assertions for Sorting”, IEEE Transactions of Software Engineering, Vol 20 No 6, 1994.

[Sch80] Ben Schneiderman: “Software Psychology: Human Factors in Computer and Information Systems”, Winthrop, 1980.

[Sch86] Ben Schneiderman: “Exploratory Experiments in Programmer Behaviour”, International Journal of Computing and Information Science, Vol 5, 1976.

[SE84] Eliot Soloway, Kate Ehrlich: “Empirical Studies of Programming Knowledge”, IEEE Transactions on Software Engineering, Vol 10, No 5, September 1984.

[SH91] Charles Simonyi, Martin Heller: “The Ungarian Revolution”, BYTE, August 1991.

[She81] B. A. Sheil, “The Psychological Study of Programming”, Computing Surveys Vol 13 No 1, March 1981.

[SHS93] Ignacio Silva-Lepe, Walter Hürsch, Greg Sullivan: “A Demeter/C++ Report”, Technical report, Northeastern University, College of Computer Science, 1993.

[Sim77] Charles Simonyi: “Meta-Programming: A Software Production Method”, PhD Thesis, Stanford University, 1977.

[SKC81] Shepard, Kruesi, Curtis: "The Effects of Symbology and Spacial Arrangement on Software Specification in a Coding Task", Proc. Trends and Applications 1981: Advances in Software Technology, IEEE.

[SMG74] Stevens, Myers, Constantine: “Structured Design”, IBM System Journal, Vol 13, No 2, 1974.

[Sny91] Alan Snyder: “Modeling the C++ Object Model: An Application of an Abstract Object Model”, ECOOP’91 Proceedings, Springer-Verlag 1991 (LNCS 512).

[SPC89] Software Productivity Consortium: “Ada Quality and Style: Guidelines for Professional Programmers”, Van Nostrand Reinhold, 1989.

[Str91] Bjarne Stroustrup: “The C++ Programming Language - Second Edition”, Addison-Wesley, 1991.

[Str94] Bjarne Stroustrup: “The Design and Evolution of C++”, Addison-Wesley, 1994.

[Str97] Bjarne Stroustrup: “The C++ Programming Language - Third Edition”, Addison-Wesley, 1997.

[Tas78] D. Van Tassel: “Program Style, Design, Efficiency, Debugging and Testing”, Second Edition, Prentice-Hall, 1978.

[The92] David Theilen: "No Bugs. Delivering error free code in C and C++.", Addison-Wesley, 1992

[TO90] E. Thomas, P. Oman: "A Bibliography of Programming Style Literature," ACM SIGPLAN Notices, Vol. 25 No 2, 1990.

[WD79] S. N. Woodfield, H. E. Dunsmore, V. Y. Shen: “The effects of modularisation and comments on program comprehension”, Proc. 5th Int. Conf. Soft. Eng., 1979.

[Wei71] Gerald M. Weinberg “The Psychology of Computer Programming”, Van Nostrand Reinhold, 1971.