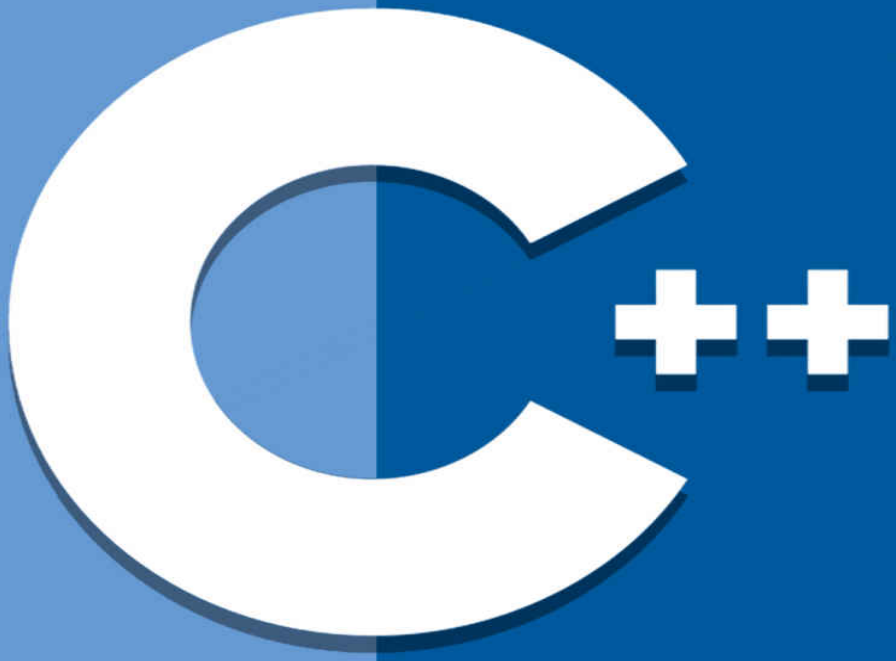
JACK FELLERS



INTRODUZIONE E FONDAMENTI AL LINGUAGGIO DI PROGRAMMAZIONE AD OGGETTI PIU' FAMOSO AL MONDO

WEBHAWK

C++

JACK FELLERS

INDICE

Caro lettore, per ringraziarti per la fiducia dimostratami acquistando il mio libro, ecco per te in **regalo**, una guida per fortificare ancora di più la tua conoscenza nella programmazione web!

Scansiona il codice o clicca sul link per riscattarlo in meno di un minuto:

Link alternativo al Qr code:
https://webhawk.tech/optin-it/

Buona Lettura!

INTRODUZIONE

C++ è un linguaggio di programmazione generico progettato per rendere la programmazione più piacevole per i programmatori. Fatta eccezione per dettagli minori, C++ è considerato un super-insieme del linguaggio di programmazione C infatti, oltre alle strutture fornite da C, C++ offre strutture flessibili ed efficienti per la definizione di nuovi tipi.

Un programmatore può suddividere un'applicazione in diverse parti definendo nuovi tipi che corrispondono strettamente ai concetti dell'applicazione. Questa tecnica per la costruzione di programmi è spesso chiamata **astrazione dei dati**. Gli oggetti di alcuni tipi definiti dall'utente contengono informazioni sul tipo. Tali oggetti possono essere utilizzati in modo comodo e sicuro in contesti in cui il loro tipo non può essere determinato al momento della compilazione. I programmi che usano oggetti di questo tipo sono spesso chiamati basati sugli oggetti. Se utilizzate bene, queste tecniche si traducono in programmi più brevi, più facili da capire e più facili da mantenere.

Il concetto chiave in C++ è la **classe** che sostanzialmente è un tipo definito dall'utente. Le classi forniscono incapsulamento dei dati, inizializzazione dei dati, conversione implicita dei tipi per tipi definiti dall'utente, tipizzazione dinamica, gestione della memoria controllata dall'utente e meccanismi per sovrascrivere gli operatori.

Se conosci già il linguaggio C, ti accorgerai che C++ offre strumenti di gran lunga migliori per la verifica del tipo e per esprimere la modularità rispetto a C. Contiene inoltre miglioramenti che non sono direttamente correlati alle classi tra cui costanti simboliche, argomenti di funzione predefiniti ecc.

C++, tuttavia, mantiene la capacità di C di gestire in modo efficiente gli oggetti fondamentali dell'hardware (bit, byte, parole, indirizzi, ecc.) e ciò consente d'implementare i tipi definiti dall'utente con un elevato grado di efficienza. C++ e le sue librerie standard sono progettate per la portabilità, infatti, le librerie C possono essere utilizzate da un programma C++ e la maggior parte degli strumenti che supportano la programmazione in C possono essere utilizzati con C++. Questo Ebook ha principalmente lo scopo di aiutare i programmatori seri a imparare il linguaggio e usarlo per

progetti non banali. Forniremo una descrizione completa ed essenziale di C++, con esempi completi e molti frammenti di codice.

1

LE BASI DI C++

Il linguaggio di programmazione C++ è stato creato da Bjarne Stroustrup e dal suo team presso i Bell Laboratories (AT&T, USA) per realizzare progetti di simulazione in modo efficiente e orientato agli oggetti. Le prime versioni, che in origine erano denominate "C con classi", risalgono al 1980. Come suggerisce il nome del linguaggio, C++ è stato derivato dal linguaggio di programmazione C: ++ è l'operatore d'incremento in C. Già dal 1989 è stato istituito il Comitato ANSI (American National Standards Institute) per standardizzare il linguaggio di programmazione C++.

L'obiettivo era far sì che il maggior numero possibile di produttori di compilatori e sviluppatori di software concordasse su una descrizione unificata del linguaggio per evitare la confusione causata da una moltitudine di varianti del linguaggio. C++ non è un linguaggio puramente orientato agli oggetti ma un ibrido che include le funzionalità del linguaggio di programmazione C. Ciò significa che hai tutte le funzionalità disponibili in C:

- Alta efficienza, vicino al linguaggio macchina;
- Programmi modulari e universalmente utilizzabili;
- Programmi portabili per varie piattaforme.

Proprio per questo motivo, le grandi quantità di codice sorgente C esistente possono essere utilizzate anche nei programmi C++. C++ supporta i concetti di programmazione orientata agli oggetti (anche detta OOP in breve), che sono:

- Astrazione dei dati, ovvero la creazione di classi per descrivere gli oggetti;
- Eredità mediante la creazione di classi derivate (tra cui classi derivate multiple);
- Incapsulamento dei dati per l'accesso controllato ai dati degli oggetti;
- Polimorfismo ovvero l'implementazione d'istruzioni che possono avere effetti diversi durante l'esecuzione del programma.

Vari elementi del linguaggio sono stati aggiunti a C++, come riferimenti, modelli e gestione delle eccezioni. Anche se questi elementi del linguaggio non sono caratteristiche di programmazione strettamente orientate agli oggetti, sono importanti per l'efficienza del programma.

Nella programmazione procedurale, i dati e le funzioni (subroutine, procedure) sono separate dai dati che elaborano. Ciò ha un effetto significativo sul modo in cui un programma gestisce i dati:

- Il programmatore deve assicurarsi che i dati siano inizializzati con valori adeguati prima dell'uso e che i dati appropriati vengano passati a una funzione quando invocata;
- Se la rappresentazione dei dati viene modificata, ad esempio se un record viene esteso, anche le funzioni corrispondenti devono essere modificate.

Entrambi questi punti possono causare errori e non sono il massimo per la manutenibilità del codice.

La programmazione orientata agli oggetti sposta l'attenzione sugli oggetti, ovvero sugli aspetti su cui è centrato il problema. Un programma progettato per funzionare con conti bancari deve usare concetti come saldi, limiti di credito, trasferimenti, calcolo d'interessi e così via. Un oggetto che rappresenta un conto corrente in un programma avrà proprietà e capacità importanti per la gestione del conto stesso.

Alla luce di ciò, gli oggetti OOP combinano dati (proprietà) e funzioni (capacità). Una classe definisce un determinato tipo di oggetto definendo sia le proprietà che le capacità degli oggetti di quel tipo. Gli oggetti comunicano inviandosi reciprocamente "messaggi", che a loro volta attivano le funzioni di un altro oggetto. La programmazione orientata agli oggetti offre numerosi importanti vantaggi allo sviluppo del software:

- Bassa necessità di manutenzione: un tipo di oggetto può modificare la propria rappresentazione interna dei dati senza richiedere modifiche all'applicazione;
- Facile riutilizzo: gli oggetti hanno una manutenzione migliore e possono quindi essere utilizzati come elementi costitutivi per altri programmi;

- Ridotta suscettibilità agli errori: un oggetto controlla l'accesso ai propri dati, più specificamente, un oggetto può rifiutare tentativi di accesso errati.

2

COME USARE C++

I seguenti tre passaggi sono necessari per creare e tradurre un programma in C++:

1. Innanzitutto, viene utilizzato un editor di testo per salvare il programma C++ in un file di testo. In altre parole, il codice sorgente viene salvato in un file sorgente. Nei progetti più grandi i programmatori utilizzano la programmazione modulare quindi il codice sorgente verrà archiviato in diversi file sorgente che vengono modificati e tradotti separatamente;
2. Il codice sorgente viene inserito in un compilatore per la traduzione. Se tutto funziona come previsto, viene creato un file oggetto composto da codice macchina. Il file oggetto viene anche definito modulo;
3. Infine, il linker combina il file oggetto con altri moduli per formare un file eseguibile. Questi ulteriori moduli contengono funzioni di librerie standard o parti del programma che sono state compilate in precedenza.

È importante utilizzare l'estensione di file corretta per il nome del file del codice sorgente. Sebbene l'estensione del file dipenda dal compilatore utilizzato, le estensioni di file più comunemente utilizzate sono .cpp e. cc.

Prima della compilazione, i file d'intestazione, denominati anche file d'inclusione, possono essere copiati nel file del codice sorgente. I file d'intestazione sono semplici file di testo contenenti informazioni necessarie a vari file, ad esempio definizioni di tipo o dichiarazioni di variabili e funzioni. I file d'intestazione possono avere l'estensione. h, ma potrebbero anche non avere alcuna estensione. La libreria standard C++ contiene funzioni predefinite e standardizzate disponibili per qualsiasi compilatore.

I compilatori moderni normalmente offrono un ambiente di sviluppo software integrato, che combina i passaggi precedentemente menzionati in un'unica attività. È disponibile un'interfaccia utente grafica per la modifica, la compilazione, il Linking e l'esecuzione dell'applicazione. Inoltre, è

possibile avviare strumenti aggiuntivi, come un Debugger (utile per identificare gli errori).

Oltre ai messaggi di errore, il compilatore può mostrare anche degli avvisi. Un avviso non indica un errore di sintassi ma semplicemente attira la tua attenzione su un possibile errore nella logica del programma, ad esempio l'uso di una variabile non inizializzata.

Un programma C++ è costituito da oggetti con le relative funzioni membro associate e funzioni globali, che non appartengono a nessuna singola classe. Ogni funzione svolge il proprio compito specifico e può anche chiamare altre funzioni. È possibile creare funzioni proprie o utilizzare funzioni già pronte e presenti nella libreria standard. Tuttavia, esiste un requisito: dovrai sempre scrivere tu stesso la funzione globale main () poiché ha un ruolo speciale da svolgere; in realtà è il programma principale.

Il breve esempio di programmazione che segue mostra due degli elementi più importanti di un programma C++:

```
#include <iostream>
using namespace std;
int main()
{
cout << "Mi piace C++!" << endl;
return 0;
}
```

IL PROGRAMMA CONTIENE SOLO la funzione main() e visualizza un messaggio. La prima riga inizia con il simbolo #, che indica che la riga è destinata al preprocessore.

Il preprocessore è solo uno degli step nella fase di traduzione e, in questa fase, non viene creato alcun codice oggetto. Puoi digitare #include <nomefile> per fare in modo che il preprocessore copi il file tra parentesi angolari in questa posizione nel codice sorgente. Ciò consente al programma di accedere a tutte le informazioni contenute nel file di intestazione.

Il file di intestazione iostream comprende alcune convenzioni per i flussi di input e output. La parola "stream" indica che le informazioni coinvolte verranno trattate come un flusso di dati.

I nomi predefiniti in C++ si trovano nel namespace std (standard). La direttiva using consente l'accesso diretto ai nomi del namespace std. L'esecuzione del programma inizia con la prima istruzione nella funzione main() ed è per questo che ogni programma C++ deve avere una funzione principale.

A parte il fatto che il nome non può essere modificato, la struttura di questa funzione non è diversa da quella di qualsiasi altra funzione C++. Nel nostro esempio la funzione main () contiene due istruzioni. La prima istruzione cout << "Mi piace C++!" << endl; genera la stringa di testo Mi piace C++! sullo schermo.

Il nome cout (output della console) indica un oggetto responsabile dell'output ed il simbolo <<, indica che i caratteri vengono "spinti" nel flusso di output, infine endl (fine riga) crea una nuova riga.

L'istruzione return 0; termina la funzione main () e anche il programma, restituendo il valore 0 come codice di uscita al programma chiamante. È una prassi comune utilizzare il codice di uscita 0 per indicare che un programma è stato terminato correttamente, un valore diverso da 0 indica che si è verificato un errore.

Nota bene che le istruzioni sono seguite da un punto e virgola (;) che serve al compilatore per capire quando e dove termina un'istruzione.

Vediamo un altro esempio:

```
/*************************************
Programma con funzioni e commenti
*************************************/
#include <iostream>
using namespace std;
void linea(), messaggio(); // Prototipi
int main()
{
cout << "Ciao! Il programma inizia nel main()." << endl;
linea();
messaggio();
linea();
cout << "Sono alla fine del main()." << endl;
return 0;
}
```

```
VOID LINEA() // Disegna una linea
    {
    cout << "---------------------------------" << endl;
    }


VOID MESSAGGIO() // Visualizza un messaggio.
    {
    cout << "Sono nella funzione messaggio()." << endl;
    }
```

IL RISULTATO di questo codice sarà:

```
Ciao! Il programma inizia nel main().
-----------------------------------
Sono nella funzione messaggio().
-----------------------------------
Sono alla fine del main().
```

L'esempio riportato mostra la struttura di un programma C++ contenente più funzioni. In C++, le funzioni non devono essere definite in alcun ordine prefissato. Ad esempio, è possibile definire prima la funzione messaggio (), seguita dalla funzione linea () e infine dalla funzione main (). Tuttavia, è più comune iniziare con la funzione main () poiché questa funzione controlla il flusso del programma. In altre parole, main () chiama le funzioni che devono ancora essere definite. Ciò è reso possibile fornendo al compilatore un prototipo di funzione che include tutte le informazioni necessarie per l'esecuzione.

Questo esempio introduce anche i commenti ovvero le stringhe racchiuse in /*...*/ o che iniziano con // vengono interpretati come commenti. Nei commenti a riga singola il compilatore ignora tutti i caratteri che seguono il simbolo // fino alla fine della riga. I commenti che si estendono su più righe sono utili durante la risoluzione dei problemi, in quanto è possibile utilizzarli per mascherare sezioni complete del programma.

Ogni tipo di commento può essere utilizzato per commentare l'altro tipo. Per quanto riguarda il layout dei file sorgenti, il compilatore analizza

sequenzialmente ogni file, suddividendo il contenuto in token, come nomi di funzioni e operatori.

I token possono essere separati da un numero qualsiasi di spazi bianchi, ovvero da spazi, tabulazioni o nuove righe.

L'ordine del codice sorgente è importante ma non è importante aderire ad un layout specifico, come la formattazione del codice in righe e colonne.

Potresti anche scrivere tutto il codice in due righe, per C++ si tratta sempre di un codice valido purché siano rispettate le sue regole.

Le direttive del preprocessore sono un'eccezione alla regola del layout poiché occupano sempre una sola riga. Il segno # all'inizio di una riga può essere preceduto solo da uno spazio o da un carattere di tabulazione.

Per migliorare la leggibilità dei tuoi programmi C++ devi adottare uno stile coerente, usando l’indentazione e le righe vuote per riflettere la struttura del tuo programma. Inoltre, fai un uso generoso dei commenti ma mai banale.

3

TIPI, COSTANTI E VARIABILI

Un programma può utilizzare diversi dati per risolvere un determinato problema, ad esempio caratteri, numeri interi o numeri in virgola mobile. Poiché un computer utilizza metodi diversi per l'elaborazione e il salvataggio dei dati, è necessario conoscere il tipo di dati. Il tipo definisce:

- La rappresentazione interna dei dati;
- La quantità di memoria da allocare.

Un numero come -1000 può essere memorizzato in 2 o 4 byte. Quando si accede alla parte della memoria in cui è memorizzato il numero, è importante leggere il numero corretto di byte. Inoltre, il contenuto della memoria, ovvero la sequenza di bit da leggere, deve essere interpretato correttamente come un intero con segno. Il compilatore C++ riconosce i tipi fondamentali, noti anche come tipi integrati, su cui si basano tutti gli altri tipi (vettori, puntatori, classi, ...).

Bool

Il risultato di un confronto o un'associazione logica che utilizza AND o OR è un valore booleano, che può essere vero o falso. C++ utilizza il tipo bool per rappresentare valori booleani. Un'espressione di tipo bool può essere true o false, dove il valore interno per true sarà rappresentato come valore numerico 1 e false da uno zero.

Char e wchar_t

QUESTI TIPI VENGONO UTILIZZATI per il salvataggio dei codici carattere. Un codice carattere è un numero intero associato a ciascun carattere. La lettera A è rappresentata dal codice 65, ad esempio. L'insieme di caratteri definisce quale codice rappresenta un determinato carattere. Quando si visualizzano i caratteri sullo schermo, vengono trasmessi i codici dei caratteri applicabili e il "ricevitore", ovvero lo schermo, è responsabile dell'interpretazione corretta dei codici.

Il linguaggio C++ non specifica alcun set di caratteri particolare, sebbene in generale venga utilizzato un set di caratteri che contiene il codice ASCII (American Standard Code for Information Interchange).

Questo codice a 7 bit contiene definizioni per 32 caratteri di controllo (codici 0 - 31) e 96 caratteri stampabili (codici 32 - 127). Il tipo char (carattere) viene utilizzato per memorizzare i codici carattere in un byte (8 bit). Questa quantità di spazio di archiviazione è sufficiente per un set di caratteri estesi, ad esempio il set di caratteri ANSI che contiene i codici ASCII e caratteri aggiuntivi.

Il tipo wchar_t comprende almeno 2 byte (16 bit) ed è quindi in grado di memorizzare caratteri Unicode moderni. Unicode è un codice a 16 bit utilizzato anche in Windows e contenente codici per circa 35.000 caratteri in 24 lingue.

VALORI interi

I TIPI SHORT, int e long sono disponibili per operazioni con numeri interi. Questi tipi si distinguono per i loro intervalli di valori. La tabella seguente mostra i tipi interi, che sono anche chiamati tipi integrali, con i loro requisiti di archiviazione e intervalli di valori tipici.

TIPO
Dimensione
Intervallo
char
1 byte

-128 a +127 o 0 a 255
unsigned char
1 byte
0 a 255
signed char
1 byte
-128 a +127
int
2 byte
-32768 a +32767

4 byte
-2147483648 a +2147483647
unsigned int
2 byte
0 a 65535

4 byte
0 a 4294967295
short
2 byte
-32768 a +32767
unsigned short
2 byte
0 a 65535
long
4 byte
-2147483648 a +2147483647
unsigned long
4 byte
0 a 4294967295

Il tipo int (intero) è fatto su misura per i computer e si adatta alla lunghezza del un registro del computer. Per i computer a 16 bit (molto datati), int è quindi equivalente a short, mentre per i computer a 32 bit int sarà equivalente a long.

C++ tratta i codici dei caratteri proprio come i normali numeri interi, ciò significa che è possibile eseguire calcoli con variabili appartenenti ai tipi char o wchar_t esattamente allo stesso modo delle variabili di tipo int.

Char è un tipo integrale con dimensione di un byte, l'intervallo di valori è quindi compreso tra -128 e +127 o tra 0 e 255, a seconda che il compilatore interpreti il tipo di carattere come con o senza segno. Il tipo wchar_t è un ulteriore tipo integrale ed è normalmente definito come short senza segno.

I tipi short, int e long sono normalmente interpretati come con segno infatti il bit più alto rappresenta il segno. Tuttavia, i tipi integrali possono essere preceduti dalla parola chiave unsigned. La quantità di memoria richiesta rimane inalterata ma l'intervallo di valori cambia a causa del bit più alto non più richiesto per il segno. La parola chiave unsigned può essere utilizzata come abbreviazione di unsigned int.

Anche il tipo char viene normalmente interpretato come con segno. Poiché questa è solo una convenzione e non è obbligatoria, la parola chiave signed è opzionale. Sono quindi disponibili tre tipi: char, signed char e unsigned char.

Float

I numeri reali sono indicati da un punto decimale in C++ e sono indicati come numeri in virgola mobile. Contrariamente agli interi, i numeri in virgola mobile devono essere memorizzati con una precisione definita. Per i calcoli che coinvolgono numeri in virgola mobile sono disponibili i seguenti tre tipi:

- Float per precisione semplice;
- Double per precisione doppia;
- Long double per precisione elevata.

L'intervallo di valori e l'accuratezza di un tipo sono derivati dalla quantità di memoria allocata e dalla rappresentazione interna del tipo. La precisione è espressa in decimali e ciò significa che "sei cifre decimali" consentono a un programmatore di memorizzare due numeri in virgola mobile che differiscono tra le prime sei cifre decimali come numeri separati. Al contrario, non vi è alcuna garanzia che 12.3456 e 12.34561 saranno distinti quando si lavora con una precisione di sei decimali. E ricorda, non si tratta della posizione del punto decimale, ma semplicemente della sequenza numerica. Se per il tuo programma è importante visualizzare numeri in virgola mobile con una precisione supportata da una macchina particolare, devi fare riferimento ai valori definiti nel file di intestazione cfloat.

La quantità di memoria necessaria per memorizzare un oggetto di un certo tipo può essere verificata usando l'operatore sizeof: sizeof(tipo) restituisce la dimensione di un oggetto in byte e il nome del parametro indica il tipo di oggetto o l'oggetto stesso. Ad esempio, sizeof(int) rappresenta un valore di 2 o 4 byte a seconda della macchina. Al contrario, sizeof(float) sarà sempre uguale a 4 byte.

NOMI VALIDI

ALL'INTERNO di un programma i nomi sono usati per designare variabili e funzioni. Le seguenti regole si applicano durante la creazione di nomi, noti anche come identificatori:

- Un nome contiene una serie di lettere, numeri o caratteri di sottolineatura (_). Le lettere accentate non sono valide, inoltre, C++ distingue tra maiuscole e minuscole; cioè le lettere maiuscole e minuscole sono considerate diverse;
- Il primo carattere deve essere una lettera o un trattino basso;
- Non ci sono restrizioni sulla lunghezza di un nome e tutti i caratteri nel nome sono significativi;
- le parole chiave C++ sono riservate e non possono essere utilizzate come nomi.

Di seguito sono indicate le parole chiave C++ e alcuni esempi di nomi validi e non validi:

asm
auto
bool
break
case
catch
char
class
const
const_cast
continue
default
delete
do
double
dynamic_cast
else
enum
explicit
extern
false
float
for
friend
goto
if
inline
int
long
mutable
namespace
new
operator
private
protected

public
register
reinterpret_cast
return
short
signed
sizeof
static
static_cast
struct
switch
template
this
throw
true
try
typedef
typeid
typename
union
unsigned
using
virtual
void
volatile

Esempi di nomi validi:

a
TEST
test
VOID
_variabile
ImpostaColore
B12
inizio_pagina
nome_molto_lungo123467890

ECCO, invece, alcuni nomi che non sono consentiti:

goto
598_test
variabile-test
TEST$
true
già

IL COMPILATORE C++ utilizza nomi interni che iniziano con uno o due trattini bassi seguiti da una lettera maiuscola. Per evitare confusione con questi nomi, evita l'uso del trattino basso all'inizio di un nome. In circostanze normali, il linker valuta solo un determinato numero di caratteri, ad esempio i primi 8 caratteri di un nome quindi i nomi degli oggetti globali, come le funzioni, dovrebbero essere scelti in modo che i primi otto caratteri siano significativi.

VARIABILI

È necessario definire una variabile prima di poterla utilizzare in un programma. Quando si definisce una variabile, viene specificato il tipo e riservata una quantità appropriata di memoria. Questo spazio di memoria è indirizzato facendo riferimento al nome della variabile. Una semplice definizione ha la sintassi seguente:

tipo nome1 [, nome2...];

Questo definisce i nomi delle variabili nell'elenco come variabili del tipo tipo. Le parentesi [...] nella descrizione della sintassi indicano che questa parte è facoltativa e può essere omessa. Pertanto, una o più variabili possono essere dichiarate all'interno di una singola istruzione.

In un programma, le variabili possono essere definite all'interno delle funzioni del programma o al di fuori di esse. Ciò ha il seguente effetto:

- una variabile definita all'esterno di ciascuna funzione è globale, ovvero può essere utilizzata da tutte le funzioni;
- una variabile definita all'interno di una funzione è locale, ovvero può essere utilizzata solo in quella funzione.

Le variabili locali vengono normalmente definite immediatamente dopo la prima parentesi graffa, ad esempio, all'inizio di una funzione. Tuttavia, possono essere definite ovunque sia consentita un'istruzione. Ciò significa che le variabili possono essere definite immediatamente prima di essere utilizzate dal programma.

Vediamo un esempio di dichiarazione singola e multipla:

char c**;**
int i**,** contatore**;**
double x**,** y**,** dimensione**;**

Una variabile può essere inizializzata, ovvero un valore può essere assegnato alla variabile durante la sua definizione. L'inizializzazione si ottiene posizionando quanto segue immediatamente dopo il nome della variabile:

- un segno di uguale (=) e un valore iniziale per la variabile

oppure

- parentesi tonde contenenti il valore della variabile.

Qualsiasi variabile globale non inizializzata viene esplicitamente impostata a zero. Al contrario, il valore iniziale per qualsiasi variabile locale che non si riesce ad inizializzare avrà un valore iniziale non definito.

COSTANTI

LA PAROLA chiave const viene utilizzata per creare un oggetto di "sola lettura". Poiché un oggetto di questo tipo è costante, non può essere

modificato in una fase successiva e deve essere inizializzato come segue:

```
CONST DOUBLE PI_GRECO = 3.1415947;
    pi_greco = pi_greco + 2.5; // non valido
```

PERTANTO, il valore di pi_greco non può essere modificato dal programma. In questo caso la seconda istruzione genererà un messaggio di errore.

4

FUNZIONI

Ogni nome, o identificatore, presente in un programma deve essere noto al compilatore o provocherà un messaggio di errore. Ciò significa che tutti i nomi diversi dalle parole chiave devono essere dichiarati, cioè introdotti nel compilatore, prima di essere utilizzati. Ogni volta che viene definita una variabile o una funzione, viene anche dichiarata. Al contrario, non tutte le dichiarazioni devono essere una definizione. Se è necessario utilizzare una funzione che è già stata introdotta in una libreria, è necessario dichiarare la funzione ma non è necessario ridefinirla.

Una funzione ha un nome e un tipo, molto simile a una variabile. Il tipo di funzione è definito dal valore restituito ovvero dal valore che la funzione restituisce al programma. Inoltre, è importante il tipo di argomenti richiesti da una funzione. Quando viene dichiarata una funzione, al compilatore devono quindi essere fornite informazioni su:

- nome e tipo della funzione;
- sul tipo di ciascun argomento.

Questo è anche indicato come prototipo di funzione, vediamone un esempio:

double potenza**(**double**,** double**);**

La funzione potenza() restituisce un tipo double e accetta due argomenti di tipo double che devono essere passati alla funzione quando viene chiamata. I tipi di argomenti possono essere seguiti da nomi (che consiglio sempre di usare), tuttavia i nomi vengono visualizzati solo come commento dal compilatore.

double potenza**(**double base**,** double esponente**);**

Dal punto di vista del compilatore, questi prototipi sono equivalenti. I prototipi di funzioni standard non devono essere dichiarati poiché sono già stati dichiarati nei file di intestazione standard. Se il file di intestazione è incluso nel codice sorgente del programma mediante la direttiva #include, la funzione può essere utilizzata immediatamente. In questo caso potremmo usare la funzione pow() della libreria cmath. Una chiamata o invocazione di

funzione è un'espressione dello stesso tipo della funzione e il cui valore corrisponde al valore restituito.

Il valore restituito viene comunemente passato ad una variabile adatta per quel tipo. Nel prossimo esempio la funzione pow() viene prima chiamata usando gli argomenti x e 3.0 e il risultato viene assegnato a y. Poiché la chiamata di funzione rappresenta un valore, sono anche possibili altre operazioni pertanto, la funzione pow() può essere utilizzata per eseguire calcoli per valori double.

y = pow(x, 3.0);

Qualsiasi espressione può essere passata a una funzione come argomento, può essere una costante o un'espressione aritmetica. Tuttavia, è importante che i tipi di argomenti corrispondano a quelli previsti dalla funzione.

Il compilatore fa riferimento al prototipo per verificare che la funzione sia stata chiamata correttamente e se il tipo di argomento non corrisponde esattamente al tipo definito nel prototipo, il compilatore esegue la conversione del tipo, se possibile.

y = pow(x, 3);

Il valore 3 di tipo int viene passato alla funzione come secondo argomento ma poiché la funzione prevede un valore double, il compilatore eseguirà la conversione del tipo da int a double. Se viene chiamata una funzione con un numero errato di argomenti o se la conversione del tipo risulta impossibile, il compilatore genera un messaggio di errore. Ciò consente di riconoscere e correggere gli errori causati dalla chiamata di funzioni in fase di sviluppo invece di causare errori di run-time.

È inoltre possibile scrivere funzioni che eseguono una determinata azione ma non restituiscono un valore alla funzione che le ha chiamate. Il tipo void è disponibile per funzioni di questo tipo, che vengono anche definite procedure in altri linguaggi di programmazione. La funzione standard srand() inizializza un algoritmo che genera numeri casuali. Poiché la funzione non restituisce un valore, è di tipo void. Un valore senza segno viene passato alla funzione come argomento per eseguire l'inizializzazione

del generatore di numeri casuali infatti il valore viene utilizzato per creare una serie di numeri casuali. Vediamo il prototipo della funzione:

```
void srand( unsigned int seed );
```

SE UNA FUNZIONE non prevede un argomento, il prototipo della funzione deve essere dichiarato void o le parentesi graffe che seguono il nome della funzione non devono racchiudere alcun valore.

```
int rand( void ); // o int rand();
```

I PROTOTIPI di funzione per srand() e rand() sono disponibili sia nei file di intestazione cstdlib che stdlib.h. Chiamare la funzione rand() senza aver precedentemente chiamato srand() crea la stessa sequenza di numeri come se fosse stata eseguita la seguente istruzione:

```
srand(1);
```

SE SI DESIDERA EVITARE di generare la stessa sequenza di numeri casuali ogni volta che si esegue il programma, è necessario chiamare srand() con un valore diverso per l'argomento ogni volta che si esegue il programma. È comune utilizzare l'ora corrente per inizializzare un generatore di numeri casuali.

5

LE CLASSI

I file di intestazione sono file di testo contenenti dichiarazioni e macro. Usando una direttiva #include queste dichiarazioni e macro possono essere rese disponibili a qualsiasi altro file sorgente, anche in altri file di intestazione. Prestare attenzione ai seguenti punti quando si utilizzano file di intestazione:

- i file di intestazione dovrebbero generalmente essere inclusi all'inizio di un programma prima di qualsiasi altra dichiarazione;
- è possibile nominare un solo file di intestazione per direttiva #include;
- il nome del file deve essere racchiuso tra parentesi angolate <...> o virgolette doppie "...".

I FILE di intestazione che accompagnano il compilatore tenderanno ad essere archiviati in una cartella a sé stante, normalmente chiamata include. Se il nome del file di intestazione è racchiuso tra parentesi angolari <...>, è comune cercare solo i file di intestazione nella cartella include.

La cartella corrente non viene utilizzata ai fini della ricerca per aumentare la velocità durante la ricerca di file di intestazione ma i programmatori C++ di solito scrivono i propri file di intestazione e li memorizzano nella cartella del progetto corrente. Per consentire al compilatore di trovare questi file di intestazione, la direttiva #include deve indicare il nome dei file di intestazione tra virgolette doppie come segue:

#include "project.h"

IN QUESTO CASO, il compilatore cercherà quindi anche la cartella corrente. Il suffisso del file .h viene normalmente utilizzato per i file di intestazione definiti dall'utente.

Oltre ai prototipi di funzioni standard, i file di intestazione contengono anche definizioni di classi standard. Quando viene incluso un file di

intestazione, le classi definite e tutti gli oggetti dichiarati nel file sono disponibili per il programma. Seguendo queste direttive, le classi istream e ostream possono essere utilizzate con i flussi cin e cout. L'oggetto cin fa parte della classe istream mentre cout è un oggetto della classe ostream.

I file di intestazione standardizzati per il linguaggio di programmazione C sono stati adottati per lo standard C++ e, pertanto, la completa funzionalità delle librerie C standard è disponibile per i programmi C++.

Gli identificatori dichiarati nei file di intestazione C sono globalmente visibili, ciò può causare conflitti di nomi in programmi di grandi dimensioni. Per questo motivo ogni file di intestazione C, ad esempio nome.h, è accompagnato in C++ da un secondo file di intestazione, cname, che dichiara gli stessi identificatori nel namespace std. Includere il file math.h equivale quindi a:

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

I FILE string.h o cstring devono essere inclusi nei programmi che utilizzano le funzioni standard per manipolare le stringhe C. Questi file di intestazione consentono l'accesso alla funzionalità della libreria di stringhe C e devono essere distinti dal file di intestazione di stringa che definisce la classe di stringhe. Ogni compilatore offre file di intestazione aggiuntivi per funzionalità dipendenti dalla piattaforma. Possono essere librerie grafiche o interfacce di database.

Diverse classi sono definite nella libreria standard C++, dalle classi per il flusso di input e output alle classi per rappresentare stringhe o gestire condizioni di errore. Ogni classe è un tipo con determinate proprietà e capacità. Come accennato in precedenza, le proprietà di una classe sono definite dai suoi membri e le capacità della classe sono definite dai suoi metodi. I metodi sono funzioni che appartengono a una classe e cooperano con i membri per eseguire determinate operazioni. I metodi vengono anche definiti funzioni membro.

Un oggetto è una variabile di un tipo di classe, detta anche istanza della classe. Quando viene creato un oggetto, la memoria viene allocata ai membri e inizializzata con valori adeguati come nell'esempio:

```
string s("Sono una stringa");
```

In questo esempio, l'oggetto s, un'istanza della stringa di classe standard (o semplicemente una stringa), viene definito e inizializzato con la costante di stringa che segue. Gli oggetti della classe string gestiscono lo spazio di memoria richiesto per la stringa stessa. In generale, ci sono diversi modi per inizializzare un oggetto di una classe. Tutti i metodi definiti come pubblici all'interno della classe corrispondente possono essere chiamati per un oggetto. Contrariamente alla chiamata di una funzione globale, viene sempre chiamato un metodo per un oggetto particolare. Il nome dell'oggetto precede il metodo ed è separato dal metodo da un punto, ad esempio:

```
s.length(); // oggetto.metodo();
```

Il metodo length() fornisce la lunghezza di una stringa, ovvero il numero di caratteri in una stringa. Ciò si traduce in un valore di 16 per la stringa s sopra definita.

Esistono funzioni definite globalmente per alcune classi standard, queste funzioni eseguono determinate operazioni per gli oggetti passati come argomenti. La funzione globale getline(), ad esempio, memorizza una riga di input da tastiera in una stringa. L'input della tastiera viene terminato premendo il tasto Invio per creare un carattere di nuova riga, '\n', che non verrà memorizzato nella stringa:

```
getline(cin, s);
```

6

GLI OPERATORI

Se un programma deve essere in grado di elaborare l'input di dati che riceve, è necessario definire le operazioni da eseguire per tali dati. Le operazioni eseguite dipenderanno dal tipo di dati, ad esempio è possibile aggiungere, moltiplicare o confrontare numeri. Tuttavia, non avrebbe alcun senso addizionare o moltiplicare le stringhe. Innanzitutto, viene fatta una distinzione tra operatori unari e binari. Un operatore unario ha un solo operando mentre un operatore binario ne ha due.

GLI ARITMETICI

GLI OPERATORI aritmetici vengono utilizzati per eseguire calcoli ed è necessario tenere presente quanto segue:

- Le divisioni eseguite con operandi integrali produrranno risultati integrali; per esempio, 7/2 restituirà 3 come risultato. Se almeno uno degli operandi è un numero in virgola mobile, anche il risultato sarà un numero in virgola mobile; ad esempio, la divisione 7.0 / 2 produce un risultato esatto di 3.5;
- La divisione con resto è applicabile solo agli operandi integrali e restituisce il resto di una divisione integrale. Ad esempio, il 7%2 restituisce il valore 1.

Un'espressione può essere usata come operando, infatti, nella sua forma più semplice un'espressione è composta da una sola costante, una variabile o una chiamata di funzione. Le espressioni possono essere utilizzate come operandi degli operatori per formare espressioni più complesse. Un'espressione tenderà generalmente ad essere una combinazione di operatori e operandi. Ogni espressione che non è un tipo void restituisce un valore. Nel caso delle espressioni aritmetiche, gli operandi definiscono il tipo dell'espressione.

int a**(**4**);** double x**(**7.9**);**

a * 512 // Tipo int
1.0 + sin(x) // Tipo double
x – 3 // Tipo double, perché uno degli operandi è di tipo double

LE REGOLE matematiche si applicano anche quando si valuta un'espressione, ovvero gli operatori *, /, % hanno una precedenza maggiore di + e -, tuttavia, è possibile utilizzare le parentesi per applicare un ordine diverso.

Esistono quattro operatori aritmetici unari: gli operatori di segno + e -, l'operatore di incremento ++ e l'operatore di decremento --.

- L'operatore di segno - restituisce il valore dell'operando ma ne inverte il segno mentre l'operatore di segno + non esegue alcuna operazione utile, restituendo semplicemente il valore del suo operando.
- L'operatore di incremento ++ modifica l'operando aggiungendo 1 unità al suo valore e ovviamente non può essere utilizzato con le costanti proprio perché ne modifica il valore. Consideriamo che i è una variabile, sia i++ (notazione postfissa) sia ++i (notazione prefisso) aumentano il valore di i di 1. In entrambi i casi viene eseguita l'operazione i = i + 1. Tuttavia, questi operatori sono due operatori diversi.

La differenza diventa evidente quando guardi il valore dell'espressione; ++i significa che il valore di i è già stato incrementato di 1 unità, mentre l'espressione i++ mantiene il valore originale di i.

Questa è una differenza importante se ++i o i++ fanno parte di un'espressione più complessa: con ++i viene prima incrementato il valore di i e poi viene applicato il nuovo valore di i, con i++ viene applicato il valore originale di i prima di incrementarla.

L'operatore -- modifica l'operando riducendo il valore dell'operando di 1 unità. Ecco un programma di esempio:

```
#include <iostream>
using namespace std;
int main()
```

```
{
int i(2), j(8);
cout << i++ << endl; // Output: 2
cout << i << endl; // Output: 3
cout << j-- << endl; // Output: 8
cout << --j << endl; // Output: 6
return 0;}
```

L'ASSEGNAZIONE

UN'ASSEGNAZIONE SEMPLICE UTILIZZA l'operatore di assegnazione = per assegnare il valore di una variabile a un'espressione. Nell'espressioni di questo tipo, la variabile deve essere posizionata a sinistra e il valore assegnato a destra dell'operatore di assegnazione. L'operatore di assegnazione ha una precedenza bassa infatti, di solito, prima viene valutato il lato destro dell'espressione e solo dopo viene assegnato il risultato alla variabile a sinistra.

Ogni assegnazione è un'espressione a sé stante e il suo valore è il valore assegnato. In questa assegnazione il numero 2.5 viene assegnato a x e quindi passato alla funzione come argomento:

```
sin(x = 2.5);
```

SONO ANCHE POSSIBILI ASSEGNAZIONI MULTIPLE, che vengono sempre valutate da destra a sinistra:

```
i = j = 9;
```

I COMPOSTI

OLTRE AI SEMPLICI operatori di assegnazione ci sono anche operatori di assegnazione composti che eseguono simultaneamente un'operazione

aritmetica e un'assegnazione, per esempio:

```
i += 3; // equivale a i = i + 3;
i *= j + 2; // equivale a i = i * (j+2);
```

IL SECONDO ESEMPIO mostra che le assegnazioni composte sono implicitamente poste tra parentesi, come dimostra il fatto che la precedenza dell'assegnazione composta è piuttosto bassa così come quella dell'assegnazione semplice. Gli operatori di assegnazione composti possono essere composti da qualsiasi operatore aritmetico binario, sono quindi disponibili i seguenti operatori composti: +=, -=, *=, /= e %=.

È possibile modificare una variabile durante la valutazione di un'espressione complessa mediante gli operatori ++, --, questa tecnica viene definita effetto collaterale (side effect) quindi ti consiglio di eseguire le operazioni singolarmente per evitare errori e per non compromettere la leggibilità del codice.

IL CONFRONTO

OGNI CONFRONTO in C++ è un'espressione di tipo bool con valore vero o falso, dove vero significa che il confronto è corretto e falso significa che il confronto è errato. Se le variabili lunghezza e spazio contengono lo stesso numero, il confronto è vero e il valore dell'espressione relazionale è vero. Se le espressioni contengono valori diversi, il valore dell'espressione sarà falso.

```
lunghezza == spazio // true o false
```

QUANDO SI CONFRONTANO i singoli caratteri, vengono confrontati, in realtà, i codici dei caratteri. Il risultato dipende quindi dal set di caratteri che si sta utilizzando. L'espressione seguente determina il valore true quando si utilizza il codice ASCII:

'A' < 'a' // true, perchè 65 < 97

GLI OPERATORI di confronto sono riepilogati nella seguente tabella:

Operatore	Effetto
==	Uguale a
>	Maggiore di
<	Minore di
>=	Maggiore o uguale a
<=	Minore o uguale a
!=	Diverso da

I LOGICI

GLI OPERATORI logici comprendono gli operatori booleani && (AND), || (OR) e ! (NOT). Possono essere utilizzati per creare condizioni composte ed eseguire l'esecuzione condizionale di un programma in base a più condizioni. Un'espressione logica produce un valore vero o falso, a seconda che l'espressione logica sia corretta o errata, proprio come un'espressione relazionale.

Gli operandi per operatori di tipo booleano sono di tipo booleano, tuttavia, è possibile utilizzare anche operandi di qualsiasi tipo che possono essere convertiti in bool, inclusi eventuali tipi aritmetici. In tal caso l'operando viene interpretato come falso o convertito in falso, se ha un valore pari a 0. Qualsiasi altro valore diverso da 0 viene interpretato come vero.

L'operatore OR (||) restituirà vero solo se almeno un operando è vero, quindi il valore dell'espressione

(spazio < 0.2**) || (**spazio > 9.8**)**

È vero se la variabile spazio è inferiore a 0,2 o maggiore di 9,8.

L'operatore AND (&&) restituirà vero solo se entrambi gli operandi sono veri, quindi l'espressione logica

(spazio < 0.2**) && (**spazio > 9.8**)**

SARÀ SEMPRE FALSA perché la stessa variabile non può essere contemporaneamente minore di 0,2 e maggiore di 9,8.

Una caratteristica importante degli operatori logici && e || è il fatto che esiste un ordine di valutazione fisso. L'operando di sinistra viene valutato per primo e se un risultato è già stato accertato, l'operando di destra non verrà valutato.

L'operatore NOT (!) restituirà vero solo se il suo operando è falso, vale anche il viceversa. Se la variabile contiene il valore false (o il valore 0), restituisce il valore booleano true. L'operatore && ha una precedenza superiore a || infatti la precedenza di entrambi questi operatori è superiore alla precedenza di un operatore di assegnazione, ma inferiore alla precedenza di tutti gli operatori precedentemente utilizzati. Questo è il motivo per cui era consentito omettere le parentesi negli esempi precedenti in questo capitolo. L'operatore NOT è un operatore unario e quindi ha una precedenza maggiore.

LE ISTRUZIONI DI CONTROLLO

Consideriamo il seguente programma:

```
#include <iostream>
using namespace std;
int main()
{
float x, y, min;
cout << "Inserisci due numeri diversi:\n";
if( cin >> x && cin >> y) // Se sono due numeri validi calcola il risultato
{
if( x < y )
min = x;
else
min = y;
cout << "\nIl numero più piccolo è: " << min << endl;
}
else
cout << "\nInput non valido!" << endl;
return 0;
}
```

IL RISULTATO di questo programma sarà:

Inserisci due numeri diversi:
3
6
Il numero più piccolo è: 3

L'istruzione if-else può essere utilizzata per scegliere tra due istruzioni condizionali. Quando viene eseguito il programma, l'espressione viene prima valutata e successivamente il controllo del programma viene deviato di conseguenza. Se il risultato è vero, viene eseguita l'istruzione dell'if altrimenti viene eseguita l'istruzione del blocco else in tutti gli altri casi, a condizione che esista un ramo else.

Se non c'è un ramo else e l'espressione è falsa, il controllo passa all'istruzione che segue l'istruzione if. È anche possibile nidificare più istruzioni if-else e non è detto che tutte le istruzioni if abbiano un ramo else. Un if può esistere senza un else ma un else non può esistere senza if, motivo per cui ogni istruzione o blocco else è sempre associato all'istruzione if precedente più vicina.

```
if( n > 0 )
if( n%2 == 1 )
cout << "Numero positivo dispari";
else
cout << "Numero positivo pari";
```

In questo esempio, il ramo else appartiene al secondo if, come indicato dal fatto che l'istruzione è stata indentata. Tuttavia, è possibile utilizzare un blocco di codice per ridefinire l'associazione di un ramo else.

```
if( n > 0 ) {
if( n%2 == 1 )
cout << "Numero positivo dispari";
else
cout << "Numero positivo pari";
}
else
cout << "Numero negativo o zero";
```

È possibile definire ed inizializzare una variabile all'interno di un'istruzione if. L'espressione è vera se la conversione del valore della variabile in un tipo bool risulta vera.

È anche possibile utilizzare una catena else-if per eseguire selettivamente una delle diverse opzioni. Una catena di questo tipo implica una serie di istruzioni if-else il cui layout è normalmente il seguente:

```
if ( espressione1 )
istruzione1
else if( espressione2 )
istruzione2
```

.
.
.
else if(espressione(n) **)**
istruzione**(**n**)**
[else istruzione**(**n+1**)]**

QUANDO VIENE ESEGUITA la catena else-if, espressione1, espressione2, ... vengono valutati nell'ordine in cui si verificano. Se una delle espressioni risulta vera, l'istruzione corrispondente viene eseguita e questo termina la catena else-if. Se nessuna delle espressioni è vera, viene eseguito il ramo else dell'ultima istruzione if. Se questo ramo else viene omesso, il programma esegue l'istruzione seguente alla catena else-if.

Proprio come la catena else-if, l'istruzione switch consente di scegliere tra più alternative. L'istruzione switch confronta il valore di un'espressione con più costanti come segue:

switch(espressione **)**
{
case const1**: [** istruzione **]**
[break;]
case const2**: [** istruzione **]**
[break;]
.
.
.
[default: istruzione **]**
}

INNANZITUTTO, viene valutata l'espressione nell'istruzione switch. Il risultato viene quindi confrontato con le costanti, const1, const2, ..., nelle etichette del case.

Le costanti devono essere diverse e possono essere solo tipi integrali (ricorda che anche i valori booleani e le costanti di caratteri sono tipi integrali). Se il valore di un'espressione corrisponde a una delle costanti del

case, il programma viene deviato nel ramo appropriato. In tal caso, il programma continua e le etichette dei case perdono il loro significato.

È possibile utilizzare la parola chiave break per uscire incondizionatamente dal blocco switch. Questa istruzione è necessaria per evitare di eseguire tutte le dichiarazioni contenute nelle etichette che seguono. Se il valore dell'espressione non corrisponde a nessuna delle costanti del case, il programma passa all'etichetta default, se disponibile. Se non si definisce un'etichetta default, non accade nulla.

L'etichetta default non deve essere per forza l'ultima etichetta; può anche essere seguita da ulteriori etichette case.

Questo costrutto richiama la catena else-if che è più versatile dell'istruzione switch. Ogni switch può essere codificato usando una catena else-if ma dovrai spesso confrontare il valore di un'espressione integrale con una serie di possibili valori. In questo caso (e solo in questo caso), è possibile utilizzare un'istruzione switch.

Come mostra l'esempio, un'istruzione switch viene letta più facilmente di una catena else-if equivalente; quindi, usa l'istruzione switch ogni volta che è possibile.

```
int comando = menu(); // La funzione menu() legge un comando
switch( comando ) // Valuta il comando
{
case 'a':
case 'A':
azione1(); // Esegue azione1
break;
case 'b':
case 'B':
azione2(); // Esegue azione2
break;
default:
cout << '\a' << flush; // Beep se input non valido
}
```

FINORA ABBIAMO VISTO operatori unari e binari ma esiste anche un operatore ternario in C++. L'operatore ternario composto da ? e : viene utilizzato per formare un'espressione che produce uno di due valori, a

seconda del valore di una condizione. Poiché il valore prodotto da tale espressione dipende dal valore di una condizione, si chiama espressione condizionale.

Contrariamente all'istruzione if-else, il meccanismo di selezione si basa sulle espressioni: viene selezionata una delle due espressioni possibili. Pertanto, un'espressione condizionale è spesso un'alternativa concisa a un'istruzione if-else.

Questa catena if-else:

```
if( a > 0 )
z = a;
else
z = -a;
```

equivale a:

```
z = (a >= 0) ? a : -a;
```

QUESTA ISTRUZIONE ASSEGNA il valore assoluto di a alla variabile z. Se a contiene il valore positivo di 12, il numero 12 viene assegnato a z. Ma se a contiene un valore negativo, ad esempio –8, il numero 8 viene assegnato a z.

C++ offre anche istruzioni per modificare il flusso di esecuzione, si tratta di break, continue e goto.

L'istruzione break esce immediatamente da uno switch o da un ciclo. È possibile utilizzare la parola chiave break per passare alla prima istruzione che segue lo switch o il ciclo.

L'istruzione continue può essere utilizzata nei cicli e ha l'effetto opposto di break, ovvero il ciclo successivo viene saltato immediatamente. Questa parola chiave è molto usata nei cicli in C++, infine è disponibile anche l'istruzione goto. Questa istruzione consente di saltare a un determinato punto contrassegnato da un'etichetta all'interno di una funzione. Ad esempio, è possibile uscire immediatamente da una struttura ad anello molto innestata o complessa.

8

I CICLI

I cicli vengono utilizzati per eseguire ripetutamente una serie di istruzioni. L'insieme di istruzioni da iterare è chiamato corpo del ciclo.

C++ offre tre elementi del linguaggio per formulare istruzioni di iterazione: while, do-while e for.

Il numero di volte in cui un ciclo viene ripetuto è definito da un'espressione di controllo. Nel caso di istruzioni while e for, questa espressione viene verificata prima dell'esecuzione del corpo del ciclo mentre in un ciclo do-while viene eseguito una volta prima del test dell'espressione. L'istruzione while ha il seguente formato:

while(espressione **)**
istruzione // corpo del ciclo

QUANDO SI ENTRA NEL CICLO, viene verificata l'espressione di controllo ovvero viene valutata l'espressione. Se questo valore è true, il corpo del ciclo viene quindi eseguito prima che l'espressione di controllo venga valutata ancora una volta. Se l'espressione di controllo è falsa, ovvero l'espressione restituisce false, il programma continua la sua esecuzione a partire dall'istruzione che segue il ciclo while.

È pratica comune posizionare il corpo del ciclo in una nuova riga del codice sorgente e indentare l'istruzione per migliorare la leggibilità del programma.

```
int contatore = 1; // Inizializzazione
while( contatore <= 10) // Controllo
{
cout << contatore << ". ciclo" << endl;
++contatore; // Reinizializzazione
}
```

COME ILLUSTRA QUESTO ESEMPIO, l'espressione di controllo è normalmente un'espressione booleana. Tuttavia, l'espressione di controllo potrebbe essere qualsiasi espressione che può essere convertita nel tipo

bool, comprese eventuali espressioni aritmetiche. Come abbiamo già appreso dalla sezione sugli operatori booleani, il valore 0 viene convertito in false e tutti gli altri valori vengono convertiti in true. Se è necessario ripetere più di un'istruzione in un ciclo del programma, è necessario posizionare le istruzioni in un blocco contrassegnato da parentesi {}. Un blocco è sintatticamente equivalente a un'istruzione, quindi è possibile utilizzare un blocco ovunque la sintassi richieda un'istruzione.

Un tipico ciclo utilizza un contatore che viene inizializzato, verificato dall'espressione di controllo e re-inizializzato alla fine del ciclo come abbiamo visto nell'esempio del ciclo while. Nel caso di un'istruzione for, gli elementi che controllano il ciclo si trovano nell'intestazione del ciclo stesso. L'esempio sopra può anche essere espresso come un ciclo for:

```
int contatore;
for( contatore = 1; contatore <= 10; ++contatore)
cout << contatore << ". ciclo" << endl;
```

QUALSIASI ESPRESSIONE PUÒ ESSERE UTILIZZATA per inizializzare e re-inizializzare il ciclo, un ciclo for si presenta quindi con questa struttura:

for(espressione1**;** espressione2**;** espressione3 **)**

L'espressione1 viene eseguita per prima e solo una volta per inizializzare il ciclo, mentre espressione2 è l'espressione di controllo, che viene sempre valutata prima di eseguire il corpo del ciclo:

- se espressione2 è falsa, il ciclo viene terminato;
- se espressione2 è vera, il corpo del ciclo viene eseguito.

Successivamente, il ciclo viene re-inizializzato eseguendo espressione3 mentre l'espressione2 viene nuovamente testata. Puoi anche definire il contatore del ciclo direttamente nell'espressione1. Ciò significa che il contatore può essere utilizzato all'interno del ciclo, ma non può essere utilizzato fuori dal ciclo stesso.

Contrariamente ai cicli while e for, che sono controllati dalle loro intestazioni, il ciclo do...while è controllato alla fine, ovvero l'espressione di controllo viene valutata dopo aver eseguito il corpo del ciclo. Ciò comporta l'esecuzione del corpo del ciclo almeno una volta.

Ecco la sua struttura:
do
istruzione
while(espressione**);**

QUANDO VIENE ESEGUITO un ciclo do...while, il corpo del ciclo viene elaborato per primo e solo dopo viene valutata l'espressione di controllo. Il corpo del ciclo viene ripetuto nuovamente se il risultato è vero, altrimenti il ciclo termina.

I cicli possono essere nidificati, ovvero il corpo di un qualsiasi ciclo può contenere un altro ciclo.

Lo standard ANSI prevede una profondità massima di 256 cicli annidati, tuttavia, è buona norma utilizzare al massimo due cicli per evitare la creazione di cicli infiniti. Innestando più cicli, infatti, è molto facile perdere il controllo quindi ti consiglio di eseguire un ciclo e salvarne il risultato, se possibile. In tal modo potrai eseguire un altro ciclo senza la necessità di doverli innestare.

9

I RIFERIMENTI

Un riferimento è un altro nome, o alias, per un oggetto che esiste già. La definizione di un riferimento non occupa memoria aggiuntiva e qualsiasi operazione definita per il riferimento viene eseguita con l'oggetto a cui si riferisce. I riferimenti sono particolarmente utili come parametri e valori di ritorno delle funzioni. Il carattere e commerciale, &, è usato per definire un riferimento. Considerato il tipo T, T& indica un riferimento a T.

```
float x = 10.7;
float& rx = x; // o float &rx = x;
```

RX È quindi un modo diverso di esprimere la variabile x e appartiene al tipo "riferimento a float". Operazioni con rx, come ad esempio:

```
--rx; // equivalente a --x;
```

INFLUENZERANNO AUTOMATICAMENTE LA VARIABILE X. Il carattere &, che indica un riferimento, compare solo nelle dichiarazioni e non è correlato all'operatore dell'indirizzo &. L'operatore & restituisce l'indirizzo di un oggetto e, se applicato ad un riferimento, restituisce l'indirizzo dell'oggetto referenziato.

```
&rx // Indirizzo di x, quindi è uguale a &x
```

UN RIFERIMENTO DEVE ESSERE INIZIALIZZATO quando viene dichiarato e non può essere modificato successivamente. In altre parole, non è possibile utilizzare il riferimento per indirizzare una variabile diversa in una fase successiva.

Un riferimento che indirizza un oggetto costante deve essere una costante, ovvero deve essere definito utilizzando la parola chiave const per evitare di modificare l'oggetto mediante riferimento. Tuttavia, è possibile utilizzare un riferimento ad una costante per indirizzare un oggetto non costante.

int a; const int& cref = a; // valido!

Il riferimento cref può essere utilizzato per l'accesso in sola lettura alla variabile a e si dice che è un identificatore di sola lettura. Un identificatore di sola lettura può essere inizializzato da una costante, in contrasto con un riferimento normale:

const double& pi = 3.1415927;

Poiché la costante non occupa spazio in memoria, il compilatore crea un oggetto temporaneo a cui viene fatto riferimento.

Un passaggio per riferimento può essere codificato utilizzando riferimenti o puntatori come parametri di funzione. Sintatticamente è più semplice usare i riferimenti, anche se non sempre è consentito. Un parametro di un tipo di riferimento è un alias per un argomento e quando viene chiamata una funzione, un parametro di riferimento viene inizializzato con l'oggetto fornito come argomento. La funzione può quindi manipolare direttamente l'argomento passato ad essa.

void test(int& a) { ++a; }

In questo caso l'istruzione:

test(var); // per una variabile var intera

incrementa la variabile var. All'interno della funzione, qualsiasi accesso al riferimento a accede automaticamente alla variabile fornita, var.

Se un oggetto viene passato come argomento usando il riferimento, l'oggetto non viene copiato.

Al contrario, l'indirizzo dell'oggetto viene passato internamente alla funzione, consentendo alla funzione di accedere all'oggetto con cui è stata chiamata.

CONTRARIAMENTE A UN NORMALE passaggio per valore, un'espressione, come a + b, non può essere utilizzata come argomento. L'argomento deve avere un indirizzo in memoria ed essere del tipo corretto.

L'uso dei riferimenti come parametri offre i seguenti vantaggi:

- gli argomenti non vengono copiati. Contrariamente al passaggio per valore, il tempo di esecuzione di un programma è inferiore, specialmente se gli argomenti occupano grandi quantità di memoria;
- una funzione può utilizzare il parametro di riferimento per restituire più valori alla funzione chiamante. Il passaggio per valore consente solo un risultato come valore di ritorno, a meno che non si ricorra all'uso di variabili globali.

Se è necessario leggere gli argomenti, ma non copiarli, è possibile definire un riferimento di sola lettura come parametro.

void mostra**(** const string**&** str**);**

LA FUNZIONE mostra() contiene una stringa come argomento, tuttavia, non genera una nuova stringa in cui viene copiata la stringa dell'argomento. Invece, str è semplicemente un riferimento all'argomento. Il chiamante può essere certo che l'argomento non verrà modificato all'interno della funzione, poiché str viene dichiarato come const. Il tipo restituito da una funzione può anche essere un tipo di riferimento. La chiamata di funzione rappresenta quindi un oggetto e può essere utilizzata proprio come un oggetto.

VEDIAMO QUESTO ESEMPIO:

string**&** messaggio**()** // Riferimento!

```
{
static string str = "Stringa di prova!";
return str;
}
```

QUESTA FUNZIONE RESTITUISCE un riferimento a una stringa statica, str. Presta attenzione nel restituire riferimenti: l'oggetto a cui fa riferimento il valore restituito deve esistere dopo aver eseguito la funzione. Sarebbe un errore critico dichiarare la stringa str come una normale variabile auto nella funzione messaggio(). Ciò distruggerebbe la stringa all'uscita dalla funzione e il riferimento punterebbe ad un oggetto che non esiste più.

10

CLASSI DERIVATE

Da Simula, C++ ha preso in prestito il concetto di classe come tipo definito dall'utente e il concetto di gerarchie di classi. Inoltre, ha preso in prestito l'idea per la progettazione del sistema in modo tale che le classi dovrebbero essere utilizzate per modellare concetti nel mondo dei programmatori e delle applicazioni. C++ fornisce costrutti di linguaggio che supportano direttamente queste nozioni di progettazione. Al contrario, l'uso delle funzionalità del linguaggio a supporto dei concetti di progettazione distingue l'uso efficace di C++. Usare i costrutti del linguaggio solo come orpelli di notazione per tipi di programmazione più tradizionali significa perdere i punti di forza chiave di C++. Un concetto non esiste isolatamente ma coesiste con concetti correlati e trae gran parte del suo potere dalle relazioni con concetti correlati. Ad esempio, prova a spiegare cos'è un'auto. Presto avrai introdotto le nozioni di ruote, motori, conducenti, pedoni, camion, ambulanze, strade, carburante, multe per eccesso di velocità ecc. Poiché utilizziamo le classi per rappresentare concetti, il problema diventa come rappresentare le relazioni tra concetti. Tuttavia, non possiamo esprimere relazioni arbitrarie direttamente in un linguaggio di programmazione. Anche se potessimo, non vorremmo. Le nostre classi dovrebbero essere definite in modo più ristretto riguardo ai nostri concetti quotidiani e in modo più preciso.

La nozione di classe derivata e i meccanismi linguistici associati sono utili per esprimere relazioni gerarchiche, cioè per esprimere comunanza tra classi. Ad esempio, i concetti di cerchio e triangolo sono correlati in quanto sono entrambi forme; cioè hanno in comune il concetto di forma. Quindi, dobbiamo definire esplicitamente la classe Cerchio e la classe Triangolo per avere la classe Forma in comune. Rappresentare un cerchio e un triangolo in un programma senza coinvolgere la nozione di forma significherebbe perdere qualcosa di essenziale. Questo capitolo è un'esplorazione delle implicazioni di questa semplice idea, che è alla base di quella che viene comunemente chiamata programmazione orientata agli oggetti. La presentazione delle caratteristiche e delle tecniche del linguaggio progredisce dal semplice e concreto al più sofisticato e astratto. Per molti programmatori, questa sarà anche una progressione dal familiare al meno noto. Non è un semplice viaggio dalle "vecchie tecniche scadenti" verso

"l'unico modo giusto", quando indico i limiti di una tecnica come motivazione per un'altra, lo faccio nel contesto di problemi specifici; per problemi diversi o in altri contesti, la prima tecnica può infatti essere la scelta migliore. Il software utile è stato costruito utilizzando tutte le tecniche presentate qui. L'obiettivo è aiutarti a raggiungere una comprensione sufficiente delle tecniche per poter fare scelte intelligenti ed equilibrate tra di esse di fronte a problemi reali. In questo capitolo, introduco innanzitutto le funzionalità di base del linguaggio a supporto della programmazione orientata agli oggetti. Successivamente, l'uso di tali funzioni per sviluppare programmi ben strutturati viene discusso nel contesto di un esempio più ampio.

Prendi in considerazione la creazione di un programma che abbia a che fare con le persone impiegate in un'azienda. Un tale programma potrebbe avere una struttura dati come questa:

```
struct Employee{
string first_name, family_name;
char middle_initial;
Date hiring_date;
short department;
/ / ...
};
```

SUCCESSIVAMENTE, potremmo provare a definire un manager come:

```
struct Manager{
Employee emp; / / manager’s employee record
set<Employee*> group; / / people managed
short level;
/ / ...
};
```

UN MANAGER È ANCHE un dipendente; i dati del dipendente sono archiviati nel membro emp di un oggetto Manager. Questo può essere ovvio per un umano, specialmente un lettore attento, ma non c'è nulla che dica al compilatore e ad altri strumenti che Manager è anche un dipendente. Un

Manager* non è un Dipendente*, quindi non si può semplicemente utilizzarne uno dove è richiesto l'altro. In particolare, non si può inserire un Dirigente in una lista di Dipendenti senza scrivere un codice speciale. Potremmo utilizzare la conversione esplicita del tipo su un Manager* o inserire l'indirizzo del membro emp in un elenco di dipendenti. Tuttavia, entrambe le soluzioni sono poco eleganti e possono essere piuttosto oscure. L'approccio corretto è indicare esplicitamente che un Manager è un Dipendente, con l'aggiunta di alcune informazioni:

```
struct Manager: public Employee{
set<Employee*> group;
short level;
/ / ...
};
```

Il Manager è derivato da Dipendente e, al contrario, Dipendente è una classe base per Manager. La classe Manager ha i membri della classe Dipendente (nome, età, ecc.) oltre ai propri membri (gruppo, livello, ecc.). La derivazione è spesso rappresentata graficamente da una freccia dalla classe derivata alla sua classe base, ciò indica che la classe derivata si riferisce alla sua base (piuttosto che viceversa).

Si dice spesso che una classe derivata erediti le proprietà dalla sua base; quindi, la relazione è anche chiamata ereditarietà. Una classe base è talvolta chiamata superclasse e una classe derivata è chiamata sottoclasse. Questa terminologia, tuttavia, confonde le persone che osservano che i dati in un oggetto di classe derivato infatti sono un super-set dei dati di un oggetto della sua classe base. Una classe derivata è più grande della sua classe base, nel senso che contiene più dati e fornisce più funzioni.

Un'implementazione popolare ed efficiente della nozione di classi derivate ha un oggetto della classe derivata rappresentato come un oggetto della classe base, con le informazioni appartenenti specificamente alla classe derivata aggiunte alla fine.

Facendo derivare Manager da Dipendente si rende Manager un sottotipo di Dipendente in modo che un Manager possa essere utilizzato ovunque sia accettabile un Dipendente. Ad esempio, possiamo creare un elenco di Dipendenti, alcuni dei quali sono Manager:

```
void f(Manager m1, Employee e1)
```

```
{
list<Employee*> elist;
elist.push_front(&m1) ;
elist.push_front(&e1) ;
/ / ...
}
```

Un Manager è (anche) un Dipendente, quindi un Manager* può essere utilizzato come Dipendente*. Tuttavia, un Dipendente non è necessariamente un Manager, quindi un Dipendente* non può essere utilizzato come Manager*. In generale, se una classe Derivata ha una classe base pubblica Base, allora una Derivata* può essere assegnata a una variabile di tipo Base* senza l'uso della conversione esplicita del tipo. La conversione opposta, da Base* a Derivata*, deve essere esplicita. Per esempio:

```
void g(Manager mm, Employee ee)
{
Employee* pe= &mm; // ok: every Manager is an Employee
Manager* pm= &ee; // error: not every Employee is a Manager
pm->level= 2; // disaster: ee doesn't have a 'level'
pm= static_cast<Manager*>(pe) ; // brute force: works because pe points to the Manager mm
pm->level= 2; // fine: pm points to the Manager mm that has a 'level'
}
```

In altre parole, un oggetto di una classe derivata può essere trattato come un oggetto della sua classe base quando manipolato tramite puntatori e riferimenti. Non è vero il contrario. Usare una classe come base equivale a dichiarare un oggetto (senza nome) di quella classe. Di conseguenza, è necessario definire una classe per poter essere utilizzata come base:

```
class Employee; // declaration only, no definition
class Manager: public Employee{ // error: Employee not defined
// ...
};
```

SEMPLICI STRUTTURE DI DATI, come Employee e Manager, non sono davvero così interessanti e spesso non particolarmente utili. Dobbiamo fornire le informazioni come un tipo proprio che fornisca un insieme adeguato di operazioni che presentino il concetto, e dobbiamo farlo senza legarci ai dettagli di una rappresentazione particolare. Per esempio:

```
class Employee{
string first_name, family_name;
char middle_initial;
// ...
public:
void print() const;
string full_name() const
{ return first_name+´´ + middle_initial+´´ + family_name; }
// ...
};
class Manager: public Employee{
// ...
public:
void print() const;
// ...
};
```

UN MEMBRO di una classe derivata può utilizzare i membri pubblici – e protetti – della sua classe base come se fossero dichiarati nella classe derivata stessa. Per esempio:

```
void Manager::print() const
{
cout<< "name is" << full_name() <<´\n´;
// ...
}
```

TUTTAVIA, una classe derivata non può utilizzare i nomi privati di una classe base:

```
VOID MANAGER::PRINT() const
    {
    cout<< " name is" << family_name<<´\n´; // error!
    // ...
    }
```

Questa seconda versione di Manager::print() non verrà compilata. Un membro di una classe derivata non ha un permesso speciale per accedere ai membri privati della sua classe base, quindi family_name non è accessibile a Manager::print(). Questo è sorprendente per alcuni, ma consideriamo l'alternativa: supponiamo che una funzione membro di una classe derivata potesse accedere ai membri privati della sua classe base. Il concetto di membro privato verrebbe reso privo di significato consentendo a un programmatore di accedere alla parte privata di una classe semplicemente derivando da essa una nuova classe. Inoltre, non si potrebbero più trovare tutti gli usi di un nome privato guardando le funzioni dichiarate come membri di quella classe. Si dovrebbe esaminare ogni file sorgente del programma completo per le classi derivate, quindi esaminare ogni funzione di quelle classi, quindi trovare ogni classe derivata da quelle classi, ecc. Questo è, nella migliore delle ipotesi, noioso e spesso poco pratico. Laddove accettabile, possono essere utilizzati membri protetti, anziché privati. Un membro protetto è come un membro pubblico per un membro di una classe derivata, ma è come un membro privato per altre funzioni. In genere, la soluzione più pulita è che la classe derivata utilizzi solo i membri pubblici della sua classe base. Per esempio:

```
    void Manager: :print() const
    {
    Employee: :print() ; // print Employee information
    cout<< level; // print Manager-specific information
    // ...
    }
```

NOTA CHE DEVE ESSERE UTILIZZATO :: perché print() è stato ridefinito in Manager. Questo riutilizzo dei nomi è tipico e i principianti potrebbero scrivere qualcosa di simile e trovare il programma coinvolto in una sequenza inaspettata di chiamate ricorsive:

```
void Manager: :print() const
    {
    print() ; // oops!
    // print Manager-specific information
    }
```

ALCUNE CLASSI derivate necessitano di costruttori. Se una classe base ha costruttori, allora deve essere invocato un costruttore. I costruttori predefiniti possono essere invocati in modo implicito. Tuttavia, se tutti i costruttori di una base richiedono argomenti, è necessario chiamare in modo esplicito un costruttore per quella base:

```
    class Employee{
    string first_name, family_name;
    short department;
    // ...
    public:
    Employee(const string& n, int d) ;
    // ...
    };
    class Manager: public Employee{
    set<Employee*> group; // people managed
    short level;
    // ...
    public:
    Manager(const string& n, int d, int lvl) ;
    // ...
    };
```

GLI ARGOMENTI per il costruttore della classe base sono specificati nella definizione di un costruttore della classe derivata. A questo riguardo, la classe base agisce esattamente come un membro della classe derivata. Per esempio:

```
Employee::Employee(const string& n, int d)
: family_name(n) , department(d) // initialize members
{
// ...
}
Manager::Manager(const string& n, int d, int lvl)
: Employee(n,d) , // initialize base
level(lvl) // initialize members
{
// ...
}
```

UN COSTRUTTORE di classi derivate può specificare inizializzatori solo per i propri membri e per le basi immediate; non può inizializzare direttamente i membri di una base. Per esempio:

```
Manager::Manager(const string& n, int d, int lvl)
: family_name(n) , // error: family_name not declared in manager
department(d) , // error: department not declared in manager
level(lvl)
{
// ...
}
```

QUESTA DEFINIZIONE CONTIENE TRE ERRORI: non riesce a richiamare il costruttore di Employee e tenta di inizializzare due volte direttamente i membri di Employee. Gli oggetti di classe sono costruiti dal basso verso l'alto: prima la base, poi i membri e poi la classe derivata stessa. Vengono distrutti nell'ordine opposto: prima la classe derivata, poi i membri e poi la base. I membri e le basi sono costruiti in ordine di dichiarazione nella classe ma sono distrutti nell'ordine inverso.

11

CLASSI ASTRATTE

Molte classi assomigliano alla classe Employee in quanto sono utili sia come se stesse che come basi per classi derivate. Per tali classi sono sufficienti le tecniche descritte nel capitolo precedente. Tuttavia, non tutte le classi seguono questo schema. Alcune classi, come la classe Shape, rappresentano concetti astratti per i quali gli oggetti non possono esistere. Una forma ha senso solo come base di una classe derivata da essa. Ciò può essere visto dal fatto che non è possibile fornire definizioni sensate per le sue funzioni virtuali:

```
class Shape{
public:
virtual void rotate(int) { error("Shape::rotate") ; } // inelegant
virtual void draw() { error("Shape::draw") ; }
// ...
};
```

CERCARE di creare una forma di questo tipo non specificato è sciocco ma consentito:

```
Shape s; // silly: ‘‘shapeless shape’’
```

È sciocco perché ogni operazione su s genererà un errore. Un'alternativa migliore è dichiarare le funzioni virtuali della classe Shape come pure funzioni virtuali. Una funzione virtuale è "resa pura" dall'inizializzatore = 0:

```
class Shape{ // abstract class
public:
virtual void rotate(int) = 0; // pure virtual function
virtual void draw() = 0; // pure virtual function
virtual bool is_closed() = 0; // pure virtual function
// ...
};
```

UNA CLASSE con una o più funzioni virtuali è anch'essa una classe astratta e non è possibile creare oggetti di quella classe astratta:

```
Shape s; // error: variable of abstract class Shape
```

UNA CLASSE astratta può essere utilizzata solo come interfaccia e come base per altre classi. Per esempio:

```
class Point{ /* ... */ };
class Circle: public Shape{
public:
void rotate(int) { } // override Shape::rotate
void draw() ; // override Shape::draw
bool is_closed() { return true; } // override Shape::is_closed
Circle(Point p, int r) ;
private:
Point center;
int radius;
};
```

UNA PURA FUNZIONE virtuale che non è definita in una classe derivata rimane una pura funzione virtuale; quindi, anche la classe derivata è una classe astratta. Questo ci permette di costruire implementazioni in più fasi:

```
class Polygon: public Shape{ // abstract class
public:
bool is_closed() { return true; } // override Shape::is_closed
// ... draw and rotate not overridden ...
};
Polygon b; // error: declaration of object of abstract class Polygon
class Irregular_polygon: public Polygon{
list<Point> lp;
public:
void draw() ; // override Shape::draw
void rotate(int) ; // override Shape::rotate
// ...
};
```

```
Irregular_polygon poly(some_points) ; // fine (assume suitable constructor)
```

UN USO importante delle classi astratte consiste nel fornire un'interfaccia senza esporre i dettagli di implementazione. Ad esempio, un sistema operativo potrebbe nascondere i dettagli dei suoi driver dietro una classe astratta:

```
class Character_device{
public:
virtual int open(int opt) = 0;
virtual int close(int opt) = 0;
virtual int read(char* p, int n) = 0;
virtual int write(const char* p, int n) = 0;
virtual int ioctl(int...) = 0;
virtual~Character_device() { } // virtual destructor
};
```

POSSIAMO quindi specificare i driver come classi derivate da Character_device e manipolare una varietà di driver attraverso quell'interfaccia.

Con l'introduzione delle classi astratte, abbiamo le strutture di base per scrivere un programma completo in modo modulare usando le classi come elementi costitutivi. Si consideri un semplice problema di progettazione: fornire un modo ad un programma per ottenere un valore intero da un'interfaccia utente. Questo può essere fatto in un numero sconcertante di modi. Per isolare il nostro programma da questa varietà, e anche per avere la possibilità di esplorare le possibili scelte di progettazione, iniziamo definendo il modello del nostro programma di questa semplice operazione di input. L'idea è di avere una classe Ival_box che sappia quale intervallo di valori di input accettare. Un programma può chiedere a un Ival_box il suo valore e richiederlo all'utente se necessario. Inoltre, un programma può chiedere a un Ival_box se un utente ha modificato il valore dall'ultima volta che il programma lo ha esaminato. Poiché ci sono molti modi per implementare questa idea di base, dobbiamo presumere che ci saranno molti

diversi tipi di Ival_box, come slider, caselle semplici in cui un utente può digitare un numero, quadranti e interazioni vocali. L'approccio generale consiste nel creare un "sistema di interfaccia utente virtuale" per l'applicazione. Può essere implementato su un'ampia varietà di sistemi per garantire la portabilità del codice dell'applicazione. Naturalmente, ci sono altri modi per isolare un'applicazione da un sistema di interfaccia utente. Ho scelto questo approccio perché è generale, perché mi permette di dimostrare una varietà di tecniche e compromessi progettuali, perché quelle tecniche sono anche quelle usate per costruire sistemi di interfaccia utente "reali" e, cosa più importante, perché queste tecniche sono applicabili a problemi ben oltre il dominio ristretto dei sistemi di interfaccia.

La nostra prima soluzione è una gerarchia di classi tradizionale, come si trova comunemente in Simula, Smalltalk e nei vecchi programmi C++. La classe Ival_box definisce l'interfaccia di base per tutte le Ival_box e specifica un'implementazione predefinita che tipi più specifici di Ival_box possono sovrascrivere con le proprie versioni. Inoltre, dichiariamo i dati necessari per implementare la nozione di base:

```
class Ival_box{
protected:
int val;
int low, high;
bool changed;
public:
Ival_box(int ll, int hh) { changed= false; val= low= ll; high= hh; }
virtual int get_value() { changed= false; return val; }
virtual void set_value(int i) { changed= true; val= i; } // for user
virtual void reset_value(int i) { changed= false; val= i; } // for application
virtual void prompt() { }
virtual bool was_changed() const{ return changed; }
};
```

L'IMPLEMENTAZIONE predefinita delle funzioni è piuttosto sciatta e viene fornita qui principalmente per illustrare la semantica prevista. Una classe realistica, ad esempio, fornirebbe un controllo dell'intervallo. Un programmatore potrebbe usare queste classi "ival" in questo modo:

```
void interact(Ival_box* pb)
{
pb->prompt() ; // alert user
// ...
int i= pb->get_value() ;
if(pb->was_changed()) {
// new value; do something
}
else{
// old value was fine; do something else
}
// ...
}
void some_fct()
{
Ival_box* p1= new Ival_slider(0,5) ; // Ival_slider derived from Ival_box
interact(p1) ;
Ival_box* p2= new Ival_dial(1,12) ;
interact(p2) ;
}
```

LA MAGGIOR PARTE del codice dell'applicazione è scritta in termini di (puntatori a) semplici Ival_box come lo è interact(). In questo modo, l'applicazione non deve conoscere il numero (potenzialmente elevato) di varianti di Ival_box. La conoscenza di tali classi specializzate è isolata nelle relativamente poche funzioni che creano tali oggetti. Questo isola gli utenti dalle modifiche nelle implementazioni delle classi derivate. La maggior parte del codice può ignorare il fatto che esistono diversi tipi di Ival_box. Per semplificare la discussione, non affronto i problemi di come un programma attende l'input. Forse il programma aspetta davvero l'utente in get_value(), forse il programma associa Ival_box a un evento e si prepara a rispondere a un callback, o forse il programma genera un thread per Ival_box e in seguito chiede informazioni sullo stato di quel thread. Tali decisioni sono cruciali nella progettazione di sistemi di interfaccia utente, tuttavia, discuterne qui il dettaglio distrarrebbe semplicemente dalla

presentazione delle tecniche di programmazione e delle strutture del linguaggio. Le tecniche di progettazione qui descritte e le strutture linguistiche che le supportano non sono specifiche per le interfacce utente. Si applicano a una gamma molto più ampia di problemi. I diversi tipi di Ival_box sono definiti come classi derivate da Ival_box. Per esempio:

```
class Ival_slider: public Ival_box{
// graphics stuff to define what the slider looks like, etc.
public:
Ival_slider(int, int) ;
int get_value() ;
void prompt() ;
};
```

I MEMBRI dati di Ival_box sono stati dichiarati protetti per consentire l'accesso da classi derivate. Pertanto, Ival_slider::get_value() può depositare un valore in Ival_box::val. Un membro protetto è accessibile dai membri di una classe e dai membri di classi derivate, ma non da utenti generici. Oltre a Ival_slider, definiremmo altre varianti di Ival_box, questi potrebbero includere Ival_dial, che ti consente di selezionare un valore ruotando una manopola; flashing_ival_slider, che lampeggia quando gli chiedi un prompt(); e popup_ival_slider, che risponde a prompt(), rendendo così difficile per l'utente ignorarlo. Da dove prendiamo la grafica? La maggior parte dei sistemi di interfaccia utente fornisce una classe che definisce le proprietà di base dell'essere un'entità sullo schermo. Quindi, se usiamo il sistema di "Big Bucks Inc.", dovremmo rendere ciascuna delle nostre classi Ival_slider, Ival_dial, ecc. una sorta di BBwindow. Ciò si ottiene semplicemente riscrivendo il nostro Ival_box in modo che derivi da BBwindow. In questo modo, tutte le nostre classi ereditano tutte le proprietà di una BBwindow. Ad esempio, ogni Ival_box può essere posizionato sullo schermo, obbedire alle regole dello stile grafico, essere ridimensionato, trascinato ecc., secondo lo standard impostato da BBwindow. La nostra gerarchia di classi sarebbe simile a questa:

```
class Ival_box: public BBwindow{ /* ... */ }; / / rewritten to use BBwindow
class Ival_slider: public Ival_box{ /* ... */ };
class Ival_dial: public Ival_box{ /* ... */ };
```

```
class Flashing_ival_slider: public Ival_slider{ /* ... */ };
class Popup_ival_slider: public Ival_slider{ /* ... */ };
```

QUESTO PROGETTO FUNZIONA BENE in molti casi e per molti problemi questo tipo di gerarchia è una buona soluzione. Tuttavia, ci sono alcuni dettagli imbarazzanti che potrebbero portarci a cercare design alternativi. Abbiamo adattato BBwindow come base di Ival_box. Questo non è del tutto corretto perché l'uso di BBwindow non fa parte della nostra nozione di base di Ival_box; è un dettaglio di attuazione. La derivazione di Ival_box da BBwindow ha elevato un dettaglio di implementazione a una decisione progettuale di primo livello. Può essere giusto, ad esempio, l'utilizzo dell'ambiente definito da "Big Bucks Inc." può essere una decisione chiave su come la nostra organizzazione conduce la propria attività. Tuttavia, se volessimo implementare anche le nostre Ival_boxes per i sistemi di "Imperial Bananas", "Liberated Software" e "Compiler Whizzes?" Dovremmo mantenere quattro versioni distinte del nostro programma:

```
class Ival_box: public BBwindow{ /* ... */ }; // BB version
class Ival_box: public CWwindow{ /* ... */ }; // CW version
class Ival_box: public IBwindow{ /* ... */ }; // IB version
class Ival_box: public LSwindow{ /* ... */ }; // LS version
```

AVERE più versioni potrebbe causare un incubo sul controllo di versione. Un altro problema è che ogni classe derivata condivide i dati di base dichiarati in Ival_box. Questi dati sono, ovviamente, un dettaglio di implementazione che si è insinuato anche nella nostra interfaccia Ival_box. Da un punto di vista pratico, in molti casi si tratta anche di dati sbagliati. Ad esempio, un Ival_slider non ha bisogno del valore memorizzato in modo specifico, può essere facilmente calcolato dalla posizione del cursore quando qualcuno esegue get_value().

In generale, mantenere due insiemi di dati correlati, ma diversi, è fonte di problemi perché prima o poi non si sincronizzeranno. Inoltre, l'esperienza mostra che i programmatori inesperti tendono a creare problemi con i dati protetti in modi che non sono necessari e che causano problemi di manutenzione. È meglio mantenere i dati privati in modo che gli autori di

classi derivate non possano pasticciarli. Meglio ancora, i dati dovrebbero trovarsi nelle classi derivate, dove possono essere definiti per soddisfare esattamente i requisiti e non possono complicare la vita di classi derivate non correlate. In quasi tutti i casi, un'interfaccia protetta dovrebbe contenere solo funzioni, tipi e costanti. La derivazione da BBwindow dà il vantaggio di mettere a disposizione degli utenti di Ival_box i servizi forniti da BBwindow. Sfortunatamente, significa anche che le modifiche alla classe BBwindow possono costringere gli utenti a ricompilare o addirittura a riscrivere il loro codice per recuperare tali modifiche. In particolare, il modo in cui funzionano la maggior parte delle implementazioni C++ implica che una modifica delle dimensioni di una classe base richiede una ricompilazione di tutte le classi derivate. Infine, il nostro programma potrebbe dover essere eseguito in un ambiente misto in cui coesistono finestre di diversi sistemi di interfaccia utente. Ciò potrebbe accadere perché due sistemi in qualche modo condividono uno schermo o perché il nostro programma deve comunicare con utenti su sistemi diversi.

12

TEMPLATES

I concetti indipendenti dovrebbero essere rappresentati in modo indipendente e dovrebbero essere raggruppati solo quando necessario. Quando questo principio viene violato, si raggruppano concetti non correlati o si creano dipendenze non necessarie. In ogni caso, ottieni un insieme di componenti meno flessibile da cui comporre i sistemi. I modelli (template) forniscono un modo semplice per rappresentare un'ampia gamma di concetti generali e modi semplici per combinarli. Le classi e le funzioni risultanti possono corrispondere a codice scritto a mano e più specializzato in termini di efficienza di runtime e spazio. I modelli forniscono un supporto diretto per la programmazione generica, ovvero la programmazione che utilizza i tipi come parametri. Il meccanismo del modello C++ consente a un tipo di essere un parametro nella definizione di una classe o di una funzione. Un modello dipende solo dalle proprietà che utilizza effettivamente dai suoi tipi di parametri e non richiede tipi diversi utilizzati come argomenti per essere esplicitamente correlati. In particolare, non è necessario che i tipi di argomento utilizzati per un modello provengano da una singola gerarchia di ereditarietà. Qui vengono introdotti i modelli con l'obiettivo principale di illustrare le tecniche necessarie per la progettazione, l'implementazione e l'uso della libreria standard.

La libreria standard richiede un grado maggiore di generalità, flessibilità ed efficienza rispetto alla maggior parte dei software. Di conseguenza, le tecniche che possono essere utilizzate nella progettazione e nell'implementazione della libreria standard sono efficaci ed efficienti per la progettazione di soluzioni per un'ampia varietà di problemi. Queste tecniche consentono ad un “implementatore” di nascondere le sofisticate implementazioni dietro interfacce semplici e di esporre la complessità all'utente solo quando l'utente ne ha una specifica esigenza.

Ad esempio, sort(v) può essere l'interfaccia per una varietà di algoritmi di ordinamento per elementi diversi e di tipi contenuti in una varietà di contenitori. Verrà scelta automaticamente la funzione di ordinamento più appropriata per la particolare v. Ogni principale astrazione di librerie standard è rappresentata come un modello (ad esempio, string, ostream, complex, list e map) e così sono le operazioni chiave (ad esempio, confronto tra stringhe, l'operatore di output <<, addizioni complicate,

ottenere l'elemento successivo da un elenco e sort()). Considera una stringa di caratteri. Una stringa è una classe che contiene caratteri e fornisce operazioni come pedice, concatenazione e confronto che di solito associamo alla nozione di "stringa". Vorremmo fornire questo comportamento per molti e diversi tipi di caratteri. Ad esempio, stringhe di caratteri con segno, stringhe di caratteri senza segno, di caratteri cinesi, di caratteri greci, ecc., sono utili in vari contesti. Pertanto, vogliamo rappresentare la nozione di "stringa" con una dipendenza minima da un tipo specifico di carattere. La definizione di una stringa si basa sul fatto che un carattere può essere copiato e poco altro. Quindi, possiamo creare un tipo stringa più generale e rendendo il tipo di carattere un parametro:

```
template<class C> class String{
struct Srep;
Srep*rep;
public:
String() ;
String(const C*) ;
String(const String&) ;
C read(int i) const;
// ...
};
```

IL PREFISSO template<class C> specifica che viene dichiarato un template e che nella dichiarazione verrà utilizzato un argomento di tipo C. Dopo la sua introduzione, C è usato esattamente come altri nomi di tipo. L'ambito di C si estende alla fine della dichiarazione preceduta da template<class C>. Nota che template<class C> dice che C è un nome di tipo; non è necessario che sia il nome di una classe. Il nome di un modello di classe seguito da un tipo racchiuso tra < > è il nome di una classe (come definito dal modello) e può essere utilizzato esattamente come altri nomi di classe. Per esempio:

```
String<char> cs;
String<unsigned char> us;
String<wchar_t> ws;
class Jchar{
// Japanese character
};
```

```
String<Jchar> js;
```

FATTA eccezione per la sintassi speciale del suo nome, String<char> funziona esattamente come se fosse stato definito utilizzando la definizione della classe String. Rendere String un modello ci consente di fornire le strutture che avevamo per String di char per stringhe di qualsiasi tipo di carattere. Ad esempio, se utilizziamo la mappa della libreria standard e il modello String, ecco un esempio di codice per il conteggio delle parole:

```
int main() // count the occurrences of each word on input
{
String<char> buf;
map<String<char>,int> m;
while(cin>>buf) m[buf]++;
// write out result
}
```

LA VERSIONE per il nostro tipo di carattere giapponese Jchar sarebbe:

```
int main() // count the occurrences of each word on input
{
String<Jchar> buf;
map<String<Jchar>,int> m;
while(cin>>buf) m[buf]++;
// write out result
}
```

LA LIBRERIA standard fornisce la classe template basic_string che è simile alla stringa template. Nella libreria standard, string è definito come sinonimo di basic_string<char>:

```
typedef basic_string<char> string;
```

QUESTO CI PERMETTE di scrivere il programma di conteggio delle parole in questo modo:

```
int main() // count the occurrences of each word on input
{
string buf;
map<string,int> m;
while(cin>>buf) m[buf]++;
// write out result
}
```

IN GENERALE, i typedef sono utili per abbreviare i nomi lunghi delle classi generate dai modelli. Inoltre, spesso preferiamo non conoscere i dettagli di come viene definito un tipo e un typedef ci consente di nascondere il fatto che un tipo viene generato da un modello.

Una classe generata da un modello di classe è una classe perfettamente ordinaria. Pertanto, l'uso di un modello non implica alcun meccanismo di runtime oltre a quello utilizzato per una classe equivalente "scritta a mano", né implica necessariamente una riduzione della quantità di codice generato. Di solito è una buona idea eseguire il debug di una classe particolare, come String, prima di trasformarla in un modello come String<C>. In questo modo, gestiamo molti problemi di progettazione e la maggior parte degli errori di codice nel contesto di un esempio concreto. Questo tipo di debugging è familiare a tutti i programmatori e la maggior parte delle persone affronta meglio un esempio concreto che un concetto astratto. In seguito, possiamo affrontare qualsiasi problema che potrebbe derivare dalla generalizzazione senza essere distratti da errori più convenzionali. Allo stesso modo, quando si cerca di comprendere un modello, è spesso utile immaginarne il comportamento per un particolare tipo di argomento come char prima di provare a comprendere il modello nella sua piena generalità. I membri di una classe modello vengono dichiarati e definiti esattamente come sarebbero stati per una classe non-modello. Non è necessario definire un membro del modello all'interno della classe del modello stessa. In tal caso, la sua definizione deve essere fornita da qualche altra parte, come per i membri della classe non modello.

I membri di una classe template sono essi stessi template parametrizzati dai parametri della loro classe template. Quando un tale membro è definito

al di fuori della sua classe, deve essere dichiarato esplicitamente un modello. Per esempio:

```
template<class C> struct String<C>::Srep{
C* s; // pointer to elements
int sz; // number of elements
int n; // reference count
// ...
};
template<class C> C String<C>::read(int i) const{ return rep->s[i] ; }
template<class C> String<C>::String()
{
p= new Srep(0,C()) ;
}
```

I PARAMETRI DEL MODELLO, come C, sono parametri anziché nomi di tipi definiti esternamente al modello. Tuttavia, ciò non influisce sul modo in cui scriviamo il codice del modello utilizzandoli. Nell'ambito di String<C>, la qualifica con <C> è ridondante per il nome del modello stesso, quindi String<C>::String è il nome del costruttore. Se preferisci, puoi essere esplicito:

```
template<class C> String<C>::String<C>()
{
p= new Srep(0,C()) ;
}
```

PROPRIO COME PUÒ ESSERCI SOLO una funzione che definisce una funzione membro di una classe in un programma, può esserci solo un modello di funzione che definisce una funzione membro di un modello di classe in un programma. Tuttavia, l'overloading è una possibilità solo per le funzioni, mentre la specializzazione ci consente di fornire implementazioni alternative per un modello. Non è possibile fare overload sul nome di un modello di classe; quindi, se un modello di classe è dichiarato in un ambito, nessun'altra entità può essere dichiarata in quell'ambito con lo stesso nome. Per esempio:

```
template<class T> class String{ /* ... */ };
class String{ /* ... */ }; // error: double definition
```

UN TIPO utilizzato come argomento del modello deve fornire l'interfaccia prevista dal modello. Ad esempio, un tipo usato come argomento per String deve fornire le normali operazioni di copia. Si noti che non è necessario che argomenti diversi per lo stesso parametro del modello debbano essere correlati per ereditarietà.

Il processo di generazione di una dichiarazione di classe da una classe template e da un argomento template è spesso chiamato istanziazione di un template. Allo stesso modo, viene generata una funzione ("istanziata") da una funzione modello più un argomento modello. Una versione di un modello per un particolare argomento del modello è chiamata specializzazione. In generale, è compito dell'implementazione, non del programmatore, garantire che le versioni di una funzione modello siano generate per ogni insieme di argomenti modello utilizzati. Per esempio:

```
String<char> cs;
void f()
{
String<Jchar> js;
cs= "It´s the implementation´s job to figure out what code needs to be generated";
}
```

PER QUESTO, l'implementazione genera dichiarazioni per String<char> e String<Jchar>, per i tipi Srep corrispondenti, per i loro distruttori e costruttori predefiniti e per l'assegnazione String<char>::operator=(char*). Altre funzioni membro non vengono utilizzate e non devono essere generate. Le classi generate sono classi perfettamente ordinarie e obbediscono a tutte le regole usuali per le classi.

Allo stesso modo, le funzioni generate sono funzioni ordinarie che obbediscono a tutte le normali regole per le funzioni. Ovviamente, i modelli forniscono un modo efficace per generare codice da definizioni relativamente brevi. Di conseguenza, è necessaria una certa cautela per

evitare di inondare la memoria con definizioni di funzioni quasi identiche. Un modello può accettare parametri di tipo, parametri di tipo ordinario come int e parametri di modello. Naturalmente, un modello può assumere diversi parametri. Per esempio:

```
template<class T, T def_val> class Cont{ /* ... */ };
```

COME MOSTRATO, un parametro del modello può essere utilizzato nella definizione dei parametri del modello successivi. Gli argomenti interi sono utili per fornire dimensioni e limiti. Per esempio:

```
template<class T, int i> class Buffer{
T v[i] ;
int sz;
public:
Buffer() : sz(i) {}
// ...
};
Buffer<char,127> cbuf;
Buffer<Record,8> rbuf;
```

CONTENITORI semplici e vincolati come Buffer possono essere importanti laddove l'efficienza e la compattezza del runtime sono fondamentali (impedendo così l'uso di una stringa o di un vettore più generale). Il passaggio di una dimensione come argomento del modello consente all'implementatore di Buffer di evitare l'uso gratuito del negozio.

Un argomento modello può essere un'espressione costante, l'indirizzo di un oggetto o di una funzione con collegamento esterno oppure un puntatore a membro senza overload. Un puntatore utilizzato come argomento del modello deve essere della forma &of, dove of è il nome di un oggetto o di una funzione, oppure della forma f, dove f è il nome di una funzione. Un puntatore a membro deve essere nel formato &X::of, dove of è il nome di un membro. In particolare, una stringa letterale non è accettabile come argomento del modello. Ecco un esempio che può chiarire il concetto:

```
void f(int i)
{
```

```
Buffer<int,i> bx; // error: constant expression expected
}
```

AL CONTRARIO, un parametro del modello non di tipo è una costante all'interno del modello, quindi tentare di modificare il valore di un parametro è un errore.

13

LE ECCEZIONI

L'autore di una libreria può rilevare errori di runtime ma in generale non ha idea di cosa fare al riguardo. L'utente di una libreria può sapere come far fronte a tali errori ma non può rilevarli, altrimenti sarebbero stati gestiti nel codice dell'utente e non sarebbero stati lasciati alla libreria. La nozione di eccezione viene fornita per aiutare ad affrontare tali problemi. L'idea fondamentale è che una funzione che trova un problema che non può affrontare genera un'eccezione, sperando che il suo chiamante (diretto o indiretto) possa gestire il problema. Una funzione che vuole gestire quel tipo di problema può indicare che è disposta a cogliere quell'eccezione. Questo stile di gestione degli errori si pone in modo favorevole con le tecniche più tradizionali. Ma considera le alternative. Dopo aver rilevato un problema che non può essere gestito localmente, il programma potrebbe:

[1] terminare il programma;

[2] restituire un valore che rappresenta "errore";

[3] restituire un valore valido e lasciare il programma in uno stato illegale;

[4] invocare una funzione fornita da chiamare in caso di "errore".

Caso [1], "terminare il programma" è ciò che accade per impostazione predefinita quando non viene rilevata un'eccezione. Per la maggior parte degli errori, possiamo e dobbiamo fare di meglio. In particolare, una libreria che non conosce l’ambito e la strategia generale del programma in cui è incorporata non può semplicemente invocare exit() o abort(). Una libreria che termina incondizionatamente non può essere utilizzata in un programma che non può permettersi di andare in crash. Un modo per visualizzare le eccezioni è quello di dare il controllo a un chiamante quando non può essere intrapresa nessuna azione significativa localmente.

Il caso [2], "restituire un valore di errore", non è sempre possibile perché spesso non esiste un "valore di errore" accettabile. Ad esempio, se una funzione restituisce un int, ogni int potrebbe essere un risultato plausibile. Anche quando questo approccio è fattibile, è spesso scomodo perché ogni chiamata deve essere controllata per il valore di errore. Questo può facilmente raddoppiare le dimensioni di un programma. Di

conseguenza, questo approccio è usato raramente in modo sufficientemente sistematico per rilevare tutti gli errori.

Il caso [3], "restituire un valore valido e lasciare il programma in uno stato illegale", presenta il problema che la funzione chiamante potrebbe non notare che il programma è stato messo in uno stato illegale. Ad esempio, molte funzioni standard della libreria C impostano la variabile globale errno per indicare un errore. Tuttavia, i programmi in genere non riescono ad essere testati in modo abbastanza coerente da evitare errori consequenziali causati dai valori restituiti da chiamate non riuscite. Inoltre, l'uso di variabili globali per la registrazione delle condizioni di errore non funziona molto bene in presenza di concorrenza.

La gestione delle eccezioni non ha lo scopo di gestire i problemi per i quali è rilevante il caso [4], "chiamare una funzione di gestione degli errori". Tuttavia, in assenza di eccezioni, una funzione di gestione degli errori ha esattamente gli altri tre casi come alternative per il modo in cui gestisce l'errore.

Il meccanismo di gestione delle eccezioni fornisce un'alternativa alle tecniche tradizionali quando sono insufficienti, poco eleganti e soggette a errori. Fornisce un modo per separare esplicitamente il codice di gestione degli errori dal codice "ordinario", rendendo così il programma più leggibile e più adattabile agli strumenti. Il meccanismo di gestione delle eccezioni fornisce uno stile più regolare di gestione degli errori, semplificando così la cooperazione tra frammenti di programma scritti separatamente. Un aspetto dello schema di gestione delle eccezioni che apparirà nuovo ai programmatori C e Pascal è che la risposta predefinita a un errore (specialmente a un errore in una libreria) è terminare il programma. La gestione delle eccezioni rende i programmi più "fragili", nel senso che è necessario prestare maggiore attenzione e sforzi per far funzionare un programma in modo accettabile. Ciò sembra preferibile, tuttavia, per ottenere risultati errati più avanti nel processo di sviluppo o dopo che il processo di sviluppo è stato considerato completo e il programma è stato consegnato a utenti innocenti. Laddove la risoluzione è inaccettabile, possiamo intercettare tutte le eccezioni o tutte le eccezioni di un tipo specifico. Pertanto, un'eccezione termina un programma solo se un programmatore gli consente di terminarlo. Questo è preferibile alla cessazione incondizionata che si verifica quando un tradizionale recupero incompleto porta a un errore catastrofico.

A volte le persone hanno cercato di alleviare gli aspetti poco attraenti del "rimuginare" scrivendo messaggi di errore, aprendo finestre di dialogo per chiedere aiuto all'utente, ecc. Tali approcci sono utili principalmente nelle situazioni di debug in cui l'utente è un programmatore familiare con la struttura del programma. Nelle mani di utenti finali e non sviluppatori, una libreria che chieda aiuto all'utente/operatore (possibilmente assente) è inaccettabile. Come minimo, il messaggio di errore potrebbe essere nella lingua naturale sbagliata (ad esempio, in finlandese per un utente inglese). Peggio ancora, il messaggio di errore si riferirebbe in genere a concetti di libreria completamente sconosciuti a un utente (ad esempio, "argomento errato per atan2", causato da un input errato in un sistema grafico).

Le eccezioni forniscono un modo per il codice che rileva un problema dal quale non è possibile eseguire il ripristino per trasmettere il problema a una parte del sistema che potrebbe essere in grado di eseguire il ripristino. Solo una parte del sistema che ha un'idea del contesto in cui viene eseguito il programma ha qualche possibilità di comporre un messaggio di errore significativo. Il meccanismo di gestione delle eccezioni può essere visto come un analogo in fase di esecuzione del controllo del tipo in fase di compilazione e dei meccanismi di controllo dell'ambiguità. Rende il processo di progettazione più importante e può aumentare il lavoro necessario per ottenere una versione iniziale e difettosa di un programma in esecuzione. Tuttavia, il risultato è un codice che ha maggiori possibilità di essere eseguito come previsto, di essere eseguito come parte accettabile di un programma più ampio, di essere comprensibile ad altri programmatori e di essere manipolabile dagli strumenti. Allo stesso modo, la gestione delle eccezioni fornisce funzionalità specifiche del linguaggio per supportare il "buono stile" nello stesso modo in cui altre funzionalità C++ supportano il "buono stile" che può essere praticato solo in modo informale e incompleto in linguaggi come C e Pascal.

Va riconosciuto che la gestione degli errori rimarrà un compito difficile e che il meccanismo di gestione delle eccezioni, sebbene più formalizzato delle tecniche che sostituisce, è ancora relativamente non strutturato rispetto alle caratteristiche del linguaggio che coinvolgono solo il flusso di controllo locale. Il meccanismo di gestione delle eccezioni C++ fornisce al programmatore un modo per gestire gli errori dove vengono gestiti in modo più naturale, data la struttura di un sistema. Le eccezioni rendono visibile la

complessità della gestione degli errori, tuttavia, le eccezioni non sono la causa di tale complessità.

"Eccezione" può avere un significato diverso per persone diverse. Il meccanismo di gestione delle eccezioni di C++ è progettato per supportare la gestione di errori e altre condizioni eccezionali (da cui il nome). In particolare, è destinato a supportare la gestione degli errori nei programmi composti da componenti sviluppati in modo indipendente. Il meccanismo è progettato per gestire solo le eccezioni sincrone, come i controlli dell'intervallo di array e gli errori di I/O. Gli eventi asincroni, come gli interrupt della tastiera e alcuni errori aritmetici, non sono necessariamente eccezionali e non vengono gestiti direttamente da questo meccanismo. Gli eventi asincroni richiedono meccanismi fondamentalmente diversi dalle eccezioni (come definite qui) per gestirli in modo pulito ed efficiente. Molti sistemi offrono meccanismi, come i signal, per gestire l'asincronia, ma poiché questi tendono a essere dipendenti dal sistema, non verranno approfonditi.

Il meccanismo di gestione delle eccezioni è una struttura di controllo non locale basata sulla rimozione dello stack che può essere vista come un meccanismo di restituzione alternativo. Esistono quindi usi legittimi di eccezioni che non hanno nulla a che fare con gli errori, tuttavia, l'obiettivo principale del meccanismo di gestione delle eccezioni è la gestione degli errori e il supporto della tolleranza agli errori. C++ standard non ha la nozione di thread o processo, di conseguenza, le circostanze eccezionali relative alla concorrenza non verranno esaminate.

Il meccanismo di gestione delle eccezioni C++ è stato progettato per essere efficace in un programma sincrono fintanto che il programmatore (o il sistema) applica regole di concorrenza di base, ad esempio bloccare correttamente una struttura di dati condivisa durante l'utilizzo. I meccanismi di gestione delle eccezioni C++ vengono forniti per segnalare e gestire errori ed eventi eccezionali. Tuttavia, il programmatore deve decidere cosa significa essere “eccezionale” in un determinato programma. Non è sempre facile, un evento che si verifica la maggior parte delle volte che viene eseguito un programma può essere considerato eccezionale? Un evento pianificato e gestito può essere considerato un errore? La risposta ad entrambe le domande è sì. "Eccezionale" non significa "non succede quasi mai" o "disastroso". È meglio pensare a un'eccezione come "qualche parte del sistema non ha potuto fare quello che gli è stato chiesto di fare".

Le throw di eccezioni dovrebbero essere rare rispetto alle chiamate di funzione altrimenti la struttura del sistema verrebbe oscurata, tuttavia, dovremmo aspettarci che la maggior parte dei programmi di grandi dimensioni generi e catturi almeno alcune eccezioni nel corso di un'esecuzione.

Un'eccezione è un oggetto di una classe che rappresenta un'occorrenza eccezionale. Il codice che rileva un errore (spesso una libreria) esegue la throw di un oggetto. Un pezzo di codice esprime il desiderio di gestire un'eccezione mediante una clausola catch. L'effetto di un throw è di rivedere lo stack fino a quando non viene trovato un catch adatto (in una funzione che ha invocato direttamente o indirettamente la funzione che ha generato l'eccezione). Spesso le eccezioni si riversano naturalmente nelle famiglie, ciò implica che l'ereditarietà può essere utile per strutturare le eccezioni e per facilitare la gestione delle eccezioni. Ad esempio, le eccezioni per una libreria matematica potrebbero essere organizzate in questo modo:

```
class Matherr{ };
class Overflow: public Matherr{ };
class Underflow: public Matherr{ };
class Zerodivide: public Matherr{ };
// ...
```

QUESTO CI CONSENTE di gestire qualsiasi Matherr senza preoccuparci di quale sia esattamente il tipo. Per esempio:

```
void f()
{
try{
// ...
}
catch(Overflow) {
// handle Overflow or anything derived from Overflow
}
catch(Matherr) {
// handle any Matherr that is not Overflow
}
}
```

QUI, viene gestito in modo specifico un Overflow. Tutte le altre eccezioni Matherr saranno gestite dal caso generale. Organizzare le eccezioni in gerarchie può essere importante per la robustezza del codice. Ad esempio, considera come gestiresti tutte le eccezioni da una libreria di funzioni matematiche senza un tale meccanismo di raggruppamento. Ciò dovrebbe essere fatto elencando in modo esaustivo le eccezioni:

```
void g()
{
try{
// ...
}
catch(Overflow) { /* ... */ }
catch(Underflow) { /* ... */ }
catch(Zerodivide) { /* ... */ }
}
```

QUESTO NON È SOLO NOIOSO, ma un programmatore può facilmente dimenticare di aggiungere un'eccezione all'elenco. Considera cosa sarebbe necessario se non raggruppassimo le eccezioni matematiche. Quando abbiamo aggiunto una nuova eccezione alla libreria, ogni pezzo di codice che ha tentato di gestire ogni eccezione matematica dovrebbe essere modificato. In generale, tale aggiornamento universale non è fattibile dopo il rilascio iniziale della libreria. Spesso non c'è modo di trovare ogni pezzo di codice rilevante. Anche quando c'è, non possiamo in generale presumere che ogni pezzo di codice sorgente sia disponibile o che saremmo disposti ad apportare modifiche se lo fosse. Questi problemi di ricompilazione e manutenzione porterebbero a una politica secondo cui nessuna nuova eccezione può essere aggiunta a una libreria dopo il suo primo rilascio; sarebbe inaccettabile per quasi tutte le librerie.

Questo ragionamento porta le eccezioni a essere definite come gerarchie di classi per libreria o per sottosistema. Si noti che né le operazioni matematiche integrate né la libreria matematica di base (condivisa con C) riportano errori aritmetici come eccezioni. Uno dei motivi è che il

rilevamento di alcuni errori aritmetici, come la divisione per zero, è asincrono su molte architetture di macchine pipeline.

14

GERARCHIA DI CLASSI

In generale, una classe è costruita da un reticolo di classi base. Poiché la maggior parte di tali reticoli storicamente sono stati alberi, un reticolo di classi è spesso chiamato gerarchia di classi. Cerchiamo di progettare classi in modo che gli utenti non debbano preoccuparsi eccessivamente del modo in cui una classe è composta da altre classi. In particolare, il meccanismo della chiamata virtuale assicura che quando chiamiamo una funzione f() su un oggetto, la stessa funzione sia chiamata qualunque classe nella gerarchia abbia fornito la dichiarazione di f() usata per la chiamata. Esaminiamo i modi per comporre i reticoli di classi e controllare l'accesso a parti di classi e sulle funzionalità per navigare tra i reticoli di classi in fase di compilazione e in fase di esecuzione. Una classe può avere più di una classe base diretta, cioè più di una classe specificata dopo il carattere : nella dichiarazione di classe. Si consideri una simulazione in cui le attività simultanee sono rappresentate da una classe Task e la raccolta e la visualizzazione dei dati è ottenuta attraverso una classe Displayed. Possiamo quindi definire una classe di entità simulate, classe Satellite:

```
class Satellite: public Task, public Displayed{
// ...
};
```

L'uso di più di una classe base immediata è generalmente chiamato ereditarietà multipla. Al contrario, avere una sola classe base diretta è chiamata ereditarietà singola. Oltre a tutte le operazioni definite ad hoc per un Satellite, può essere applicata l'unione di operazioni su Task e Displayed. Per esempio:

```
void f(Satellite& s)
{
s.draw() ; // Displayed::draw()
s.delay(10) ; // Task::delay()
s.transmit() ; // Satellite::transmit()
}
```

ALLO STESSO MODO, un Satellite può essere passato a funzioni che prevedono un Task o un Displayed. Per esempio:

```
void highlight(Displayed*) ;
void suspend(Task*) ;
void g(Satellite* p)
{
highlight(p) ; // pass a pointer to the Displayed part of the Satellite
suspend(p) ; // pass a pointer to the Task part of the Satellite
}
```

L'IMPLEMENTAZIONE di questo implica chiaramente alcune (semplici) tecniche di compilazione per garantire che le funzioni che prevedono un Task vedano una parte diversa di un Satellite rispetto alle funzioni che si aspettano un Displayed. Le funzioni virtuali funzionano normalmente. Per esempio:

```
class Task{
// ...
virtual void pending() = 0;
};
class Displayed{
// ...
virtual void draw() = 0;
};
class Satellite: public Task, public Displayed{
// ...
void pending() ; // override Task::pending()
void draw() ; // override Displayed::draw()
};
```

CIÒ GARANTISCE CHE SATELLITE::DRAW() e Satellite::pending() vengano chiamati per un Satellite trattato rispettivamente come Displayed e Task. Si noti che con l'ereditarietà singola, le scelte del programmatore per l'implementazione delle classi Displayed, Task e Satellite sarebbero limitate. Un Satellite potrebbe essere un Task o un Displayed, ma non

entrambi (a meno che Task non sia derivato da Displayed o viceversa). Entrambe le alternative comportano una perdita di flessibilità. Perché qualcuno dovrebbe volere una classe Satellite? Contrariamente alle congetture di alcuni, l'esempio di Satellite è reale. C'era davvero – e forse c'è ancora – un programma costruito sulla falsariga usata qui per descrivere l'ereditarietà multipla.

È stato utilizzato per studiare la progettazione di sistemi di comunicazione che coinvolgono satelliti, stazioni di terra, ecc. Data una tale simulazione, possiamo rispondere a domande sul flusso di traffico, determinare risposte adeguate a una stazione di terra bloccata da un temporale, considerare compromessi tra connessioni satellitari e connessioni terrestri, ecc. Tali simulazioni implicano una varietà di operazioni di visualizzazione e debug. Inoltre, è necessario memorizzare lo stato di oggetti come Satellite e i loro sottocomponenti per l'analisi, il debug e il ripristino degli errori.

Casi di ambiguità

Due classi base possono avere funzioni membro con lo stesso nome. Per esempio:

```
class Task{
// ...
virtual debug_info* get_debug() ;
};
class Displayed{
// ...
virtual debug_info* get_debug() ;
};
```

QUANDO SI UTILIZZA UN SATELLITE, queste funzioni devono essere disambiguate:

```
void f(Satellite* sp)
{
debug_info* dip= sp->get_debug() ; // error: ambiguous
dip= sp->Task: :get_debug() ; // ok
dip= sp->Displayed: :get_debug() ; // ok
}
```

TUTTAVIA, la disambiguazione esplicita è complicata, quindi di solito è meglio risolvere tali problemi definendo una nuova funzione nella classe derivata:

```
class Satellite: public Task, public Displayed{
// ...
debug_info* get_debug() // override Task::get_debug() and Displayed::get_debug()
{
debug_info* dip1= Task: :get_debug() ;
debug_info* dip2= Displayed: :get_debug() ;
return dip1->merge(dip2) ;
```

```
}
};
```

QUESTO LOCALIZZA le informazioni sulle classi base di Satellite. Poiché Satellite::get_debug() sovrascrive le funzioni get_debug() di entrambe le sue classi base, Satellite::get_debug() viene chiamato ovunque get_debug() venga chiamato per un oggetto Satellite. Un nome qualificato come Telstar::draw può riferirsi a un pareggio dichiarato in Telstar o in una delle sue classi base. Per esempio:

```
class Telstar: public Satellite{
// ...
void draw()
{
draw() ; // oops!: recursive call
Satellite::draw() ; // finds Displayed::draw
Displayed::draw() ;
Satellite::Displayed::draw() ; // redundant double qualification
}
};
```

IN ALTRE PAROLE, se un Satellite::draw non si risolve in Satellite, il compilatore guarda ricorsivamente nelle sue classi base; ovvero, cerca Task::draw e Displayed::draw. Se viene trovata esattamente una corrispondenza, verrà utilizzato quel nome. In caso contrario, Satellite::draw non viene trovato o è ambiguo.

La risoluzione dell'overload non viene applicata a diversi ambiti di classe. In particolare, le ambiguità tra funzioni di classi base diverse non vengono risolte in base ai tipi di argomento. Quando si combinano classi essenzialmente non correlate, come Task e Displayed nell'esempio Satellite, la somiglianza nella denominazione in genere non indica un ambito comune. Quando si verificano tali conflitti di nomi, spesso sono una vera sorpresa per il programmatore. Per esempio:

```
class Task{
// ...
```

```
    void debug(double p) ; // print info only if priority is lower than p
    };
    class Displayed{
    // ...
    void debug(int v) ; // the higher the ‘v,’ the more debug information is
printed
    };
    class Satellite: public Task, public Displayed{
    // ...
    };
    void g(Satellite* p)
    {
    p->debug(1) ; // error: ambiguous. Displayed::debug(int) or
Task::debug(double) ?
    p->Task::debug(1) ; // ok
    p->Displayed::debug(1) ; // ok
    }
```

E SE L'USO dello stesso nome in classi base diverse fosse il risultato di una decisione progettuale deliberata e l'utente volesse selezionare in base ai tipi di argomento? In tal caso, una dichiarazione using può portare le funzioni in un ambito comune. Per esempio:

```
    class A{
    public:
    int f(int) ;
    char f(char) ;
    // ...
    };
    class B{
    public:
    double f(double) ;
    // ...
    };
    class AB: public A, public B{
    public:
    using A::f;
```

```
using B::f;
char f(char) ; // hides A::f(char)
AB f(AB) ;
};
void g(AB& ab)
{
ab.f(1) ; // A::f(int)
ab.f('a') ; // AB::f(char)
ab.f(2.0) ; // B::f(double)
ab.f(ab) ; // AB::f(AB)
}
```

LE DICHIARAZIONI di utilizzo consentono a un programmatore di comporre un insieme di funzioni sovraccaricate dalle classi base e dalla classe derivata. Le funzioni dichiarate nella classe derivata nascondono funzioni che altrimenti sarebbero disponibili da una base. Le funzioni virtuali dalle basi possono essere sovrascritte come sempre.

Una dichiarazione using in una definizione di classe deve fare riferimento ai membri di una classe base. Una dichiarazione using non può essere utilizzata per un membro di una classe al di fuori di tale classe, le sue classi derivate e le relative funzioni membro. Una direttiva using potrebbe non apparire in una definizione di classe e potrebbe non essere utilizzata per una classe. Una dichiarazione using non può essere utilizzata per accedere a informazioni aggiuntive, è semplicemente un meccanismo per rendere le informazioni accessibili più comode da usare.

Con la possibilità di specificare più di una classe base, c'è anche la possibilità di avere una classe come base due volte. Ad esempio, se Task e Displayed fossero state derivate ciascuno da una classe Link, un Satellite avrebbe due Link:

```
struct Link{
Link* next;
};
class Task: public Link{
// the Link is used to maintain a list of all Tasks (the scheduler list)
// ...
};
```

```
class Displayed: public Link{
// the Link is used to maintain a list of all Displayed objects (the display list)
// ...
};
```

QUESTO NON CAUSA PROBLEMI. Per rappresentare i collegamenti vengono utilizzati due oggetti Link separati e i due elenchi non interferiscono l'uno con l'altro. Naturalmente non si può fare riferimento a membri della classe Link senza rischiare un'ambiguità. Gli esempi in cui la classe base comune non dovrebbe essere rappresentata da due oggetti separati possono essere gestiti utilizzando una classe base virtuale. Se tale base deve essere riferita da un punto in cui sia visibile più di una copia della base, il riferimento deve essere esplicitamente qualificato per risolvere l'ambiguità. Per esempio:

```
void mess_with_links(Satellite* p)
{
p->next= 0; // error: ambiguous (which Link?)
p->Link::next= 0; // error: ambiguous (which Link?)
p->Task::Link::next= 0; // ok
p->Displayed::Link::next= 0; // ok
// ...
}
```

QUESTO È ESATTAMENTE il meccanismo utilizzato per risolvere i riferimenti ambigui ai membri.

Una funzione virtuale di una classe base replicata può essere sovrascritta da una (singola) funzione in una classe derivata. Ad esempio, si potrebbe rappresentare la capacità di un oggetto di leggere sé stesso da un file e riscriversi in un file come questo:

```
class Storable{
public:
virtual const char* get_file() = 0;
virtual void read() = 0;
```

```
virtual void write() = 0;
virtual~Storable() { write() ; } // to be called from overriding destructors
};
```

NATURALMENTE, molti programmatori potrebbero fare affidamento su questo per sviluppare classi che possono essere usate indipendentemente o in combinazione per costruire classi più elaborate.

15

LIBRERIE E CONTAINER

Cosa dovrebbe essere una libreria standard in C++? L'idea è che un programmatore è in grado di trovare ogni classe, funzione, modello, ecc. significativa e ragionevolmente generale in una libreria. Tuttavia, la domanda qui non è "Cosa dovrebbe essere in una libreria?", ma "Cosa dovrebbe essere nella libreria standard?" La risposta "Tutto!" è una prima approssimazione ragionevole di una risposta alla prima domanda ma non alla seconda. Una libreria standard è qualcosa che ogni "implementatore" deve fornire in modo che ogni programmatore possa fare affidamento su di essa.

La libreria standard C++:

1. Fornisce supporto per le funzionalità del linguaggio, come la gestione della memoria e le informazioni sui tipi di runtime;
2. Fornisce informazioni sugli aspetti del linguaggio definiti dall'implementazione, come il valore float più grande;
3. Fornisce funzioni che non possono essere implementate in modo ottimale nel linguaggio stesso per ogni sistema, come sqrt() e memmove();
4. Fornisce servizi non primitivi su cui un programmatore può fare affidamento per la portabilità come liste, mappe, funzioni di ordinamento e flussi di I/O;
5. Fornisce un framework per estendere le funzionalità che fornisce, come convenzioni e strutture di supporto che consentono a un utente di fornire I/O di un tipo definito dall'utente nello stile di I/O per i tipi integrati.
6. Fornisce la base comune per altre librerie.

Inoltre, la libreria standard fornisce alcune funzionalità, come i generatori di numeri casuali, semplicemente perché è convenzionale e utile farlo. Il design della libreria è determinato principalmente dagli ultimi tre ruoli. Questi ruoli sono strettamente correlati. Ad esempio, la portabilità è di solito un importante criterio di progettazione per una libreria specializzata e tipi di contenitori comuni come liste e mappe sono essenziali

per una comoda comunicazione tra libreria sviluppate separatamente. L'ultimo ruolo è particolarmente importante dal punto di vista della progettazione perché aiuta a limitare l'ambito della libreria standard e pone vincoli alle sue strutture. Ad esempio, le funzionalità di stringa e lista sono fornite nella libreria standard. In caso contrario, le librerie sviluppate separatamente potrebbero comunicare solo utilizzando i tipi incorporati. Tuttavia, le funzionalità di pattern matching e grafica non sono presenti. Tali strutture sono ovviamente molto utili, ma non sono direttamente coinvolte nella comunicazione tra librerie sviluppate separatamente.

A meno che una struttura non sia in qualche modo necessaria per supportare questi ruoli, può essere lasciata ad una libreria al di fuori dello standard. Nel bene e nel male, lasciare qualcosa fuori dalla libreria standard apre l'opportunità a diverse librerie di offrire realizzazioni concorrenti di un'idea.

I ruoli di una libreria standard impongono diversi vincoli alla sua progettazione. Le strutture offerte dalla libreria standard C++ sono progettate per essere:

1. Inestimabili e convenienti essenzialmente per ogni studente e programmatore professionista, inclusi i costruttori di altre librerie.
2. Utilizzate direttamente o indirettamente da ogni programmatore per tutto ciò che rientra nell'ambito della libreria.
3. Abbastanza efficienti da fornire autentiche alternative a funzioni, classi e modelli codificati manualmente nell'implementazione di ulteriori librerie.
4. Senza criteri o con la possibilità di fornire all'utente criteri come argomenti.
5. Primitive in senso matematico. Cioè, un componente che svolge due ruoli debolmente correlati sarà sicuramente meno performante rispetto ai singoli componenti progettati per svolgere un solo ruolo.
6. Comode, efficienti e ragionevolmente sicure per usi comuni.
7. Complete in quello che fanno. La libreria standard può lasciare le funzioni principali ad altre librerie, ma se svolge un'attività, deve fornire funzionalità sufficienti in modo che i singoli utenti

o implementatori non debbano sostituirla per svolgere il lavoro di base.
8. Aumentare i tipi e le operazioni incorporate.
9. Digitare sicuro per dafeult.
10. Supportare gli stili di programmazione comunemente accettati.
11. Estendibili per gestire i tipi definiti dall'utente in modi simili a quelli in cui vengono gestiti i tipi incorporati e i tipi di librerie standard.

Ad esempio, la creazione dei criteri di confronto in una funzione di ordinamento è inaccettabile perché gli stessi dati possono essere ordinati in base a criteri diversi. Questo è il motivo per cui la libreria standard C qsort() prende una funzione di confronto come argomento piuttosto che basarsi su qualcosa di fisso, diciamo, l'operatore <. D'altra parte, l'overhead imposto da una chiamata di funzione per ogni confronto compromette qsort() come elemento costitutivo per l'ulteriore costruzione di librerie. Per quasi tutti i tipi di dati, è facile eseguire un confronto senza imporre l'overhead di una chiamata di funzione. Si tratta di un overhead importante? Nella maggior parte dei casi, probabilmente no. Tuttavia, l'overhead della chiamata di funzione può dominare il tempo di esecuzione per alcuni algoritmi e indurre gli utenti a cercare alternative. La tecnica di fornire criteri di confronto attraverso un argomento descritta poco fa, risolve questo problema. L'esempio illustra la tensione tra efficienza e generalità. Una libreria standard non è necessaria solo per svolgere i suoi compiti ma deve anche eseguirli in modo abbastanza efficiente da non indurre gli utenti a fornire i propri meccanismi. In caso contrario, gli implementatori di funzionalità più avanzate sono costretti a bypassare la libreria standard per restare competitivi. Ciò aggiungerebbe un onere allo sviluppatore della libreria e complicherebbe seriamente la vita degli utenti che desiderano rimanere indipendenti dalla piattaforma o utilizzare diverse librerie sviluppate separatamente.

I requisiti di "primitività" e "convenienza degli usi comuni" sembrano in conflitto. Il primo requisito preclude l'ottimizzazione esclusiva della libreria standard per i casi comuni. Tuttavia, i componenti che soddisfano esigenze comuni, ma non primitive, possono essere inclusi nella libreria standard in aggiunta alle strutture primitive, piuttosto che come sostituti. Il culto dell'ortogonalità non deve impedirci di rendere la vita comoda al

principiante e all'utente occasionale, né dovrebbe indurci a lasciare il comportamento predefinito di un componente pericoloso.

Le funzionalità della libreria standard sono definite nel namespace std e presentate come un insieme di intestazioni. Le intestazioni identificano le parti principali della libreria. Pertanto, elencarli fornisce una panoramica della libreria e fornisce una guida alla descrizione della libreria stessa. Un'intestazione standard con un nome che inizia con la lettera c, è equivalente a un'intestazione nella libreria standard C. Per ogni intestazione <cX> che definisce i nomi nel namespace std, esiste un'intestazione <X.h> che definisce gli stessi nomi nel namespace globale.

<vector>
Array di T a una dimensione
<list>
Lista a doppio link di T
<deque>
Coda a due estremità di T
<queue>
Coda di T
<stack>
Stack di T
<map>
Array associativo di T
<set>
Set di T
<bitset>
Array di booleani

I CONTENITORI associativi multimap e multiset si trovano rispettivamente in <map> e <set>. La priority_queue è dichiarata in <queue>.

Un utente o un implementatore della libreria non è autorizzato ad aggiungere o sottrarre dichiarazioni dalle intestazioni standard. Né è accettabile provare a modificare il contenuto delle intestazioni definendo macro prima che vengano incluse o provare a cambiare il significato delle dichiarazioni nelle intestazioni mediante dichiarazioni nel loro contesto. Qualsiasi programma o implementazione che riproduce tali stratagemmi non è conforme allo standard e i programmi che si basano su tali trucchi

non sono portabili. Anche se funzionano adesso, la prossima versione di qualsiasi parte di un'implementazione potrebbe renderli inutili. Per utilizzare una funzione di libreria standard, è necessario includere la sua intestazione. Scrivere personalmente le dichiarazioni pertinenti non è un'alternativa conforme agli standard. Il motivo è che alcune implementazioni ottimizzano la compilazione in base all'inclusione di intestazioni standard e altre forniscono implementazioni ottimizzate di funzionalità di libreria standard attivate dalle intestazioni. In generale, gli implementatori utilizzano le intestazioni standard in modi che i programmatori non possono prevedere e non dovrebbero conoscere. Un programmatore può, tuttavia, specializzare template di utilità, come swap(), per tipi definiti dall'utente di librerie non standard.

Container

Un contenitore (container) è un oggetto che contiene altri oggetti. Esempi sono liste, vettori e array associativi. In generale, puoi aggiungere e rimuovere oggetti ad un contenitore. Naturalmente, questa idea può essere presentata agli utenti in molti modi diversi. I contenitori della libreria standard C++ sono stati progettati per soddisfare due criteri: fornire la massima libertà nella progettazione di un singolo contenitore, consentendo allo stesso tempo ai contenitori di presentare un'interfaccia comune agli utenti. Ciò consente un'efficienza ottimale nell'implementazione dei contenitori e consente agli utenti di scrivere codice indipendente dal particolare contenitore utilizzato. I design dei contenitori in genere soddisfano solo l'uno o l'altro di questi due criteri di progettazione. Il contenitore e gli algoritmi parte della libreria standard (spesso chiamata STL) possono essere visti come una soluzione al problema di fornire contemporaneamente generalità ed efficienza.

L'approccio più ovvio per fornire un vettore e una lista è di definire ciascuno nel modo che ha più senso per l'uso previsto:

```
template<class T> class Vector{ // optimal
public:
explicit Vector(size_t n) ; // initialize to hold n objects with value T()
T& operator[](size_t) ; // subscripting
// ...
};
template<class T> class List{ // optimal
public:
class Link{ /* ... */ };
List() ; // initially empty
void put(T*) ; // put before current element
T* get() ; // get current element
// ...
};
```

OGNI CLASSE FORNISCE operazioni che sono quasi ideali per il loro utilizzo e per ogni classe possiamo scegliere una rappresentazione adatta senza preoccuparci di altri tipi di contenitori. Ciò consente alle implementazioni delle operazioni di essere quasi ottimali. In particolare, le operazioni più comuni come put() per un Lista e operator[]() per un Vector sono piccole e facilmente semplici.

Un uso comune della maggior parte dei tipi di contenitori è quello di esaminare il contenitore guardando gli elementi uno dopo l'altro. Questo viene fatto tipicamente definendo una classe iteratore appropriata al tipo di contenitore. Tuttavia, a un utente che esegue l'iterazione su un contenitore spesso non importa se i dati sono archiviati in List o in un Vector. In tal caso, l'iterazione del codice non dovrebbe dipendere dal fatto che sia stato utilizzato List o un Vector. Idealmente, lo stesso pezzo di codice dovrebbe funzionare in entrambi i casi. Una soluzione consiste nel definire una classe iteratore che fornisce un'operazione get-next-element che può essere implementata per qualsiasi contenitore. Per esempio:

```
template<class T> class Itor{ // common interface (abstract class §2.5.4, §12.3)
public:
// return 0 to indicate no-more-elements
virtual T* first() = 0; // pointer to first element
virtual T* next() = 0; // pointer to next element
};
```

ORA POSSIAMO FORNIRE implementazioni per Vector e List:

```
template<class T> class Vector_itor: public Itor<T> { // Vector implementation
Vector<T>& v;
size_t index; // index of current element
public:
Vector_itor(Vector<T>& vv) :v(vv) , index(0) { }
T* first() { return(v.size()) ? &v[index=0] : 0; }
T* next() { return(++index<v.size()) ? &v[index] : 0; }
};
template<class T> class List_itor: public Itor<T> { // List implementation
```

```
List<T>& lst;
List<T>: :Link p; // points to current element
public:
List_itor(List<T>&) ;
T* first() ;
T* next() ;
};
```

LA STRUTTURA interna dei due iteratori è abbastanza diversa, ma per gli utenti non importa. Possiamo ora scrivere codice che itera su qualsiasi cosa per cui possiamo implementare un Itor. Per esempio:

```
int count(Itor<char>& ii, char term)
{
int c= 0;
for(char* p= ii.first() ; p; p=ii.next()) if(*p==term) c++;
return c;
}
```

C'È UN INTOPPO, tuttavia. Le operazioni su un iteratore Itor sono semplici, ma comportano il sovraccarico di una chiamata di funzione (virtuale). In molte situazioni, questo sovraccarico è minore rispetto a quello che viene avviene in realtà, tuttavia, l'iterazione attraverso un semplice contenitore è un'operazione critica in molti sistemi ad alte prestazioni e una chiamata di funzione è molte volte più costosa dell'aggiunta di interi o del dereferenziamento del puntatore che implementa next() per un vettore e una lista.

Di conseguenza, questo modello non è adatto, o almeno non è l'ideale, per una libreria standard. Tuttavia, questo modello di contenitore e iteratore è stato utilizzato con successo in molti sistemi. Per anni è stato il mio preferito per la maggior parte delle applicazioni. I suoi punti di forza e di debolezza possono essere così riassunti:

- I singoli contenitori sono semplici ed efficienti.

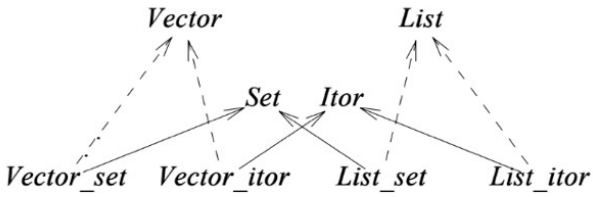
- È richiesta poca comunanza di contenitori. Iteratori e classi wrapper possono essere usati per adattare contenitori sviluppati indipendentemente in un framework comune.
- La comunanza d'uso è fornita tramite iteratori (piuttosto che tramite un tipo di contenitore generale)
- È possibile definire iteratori diversi per soddisfare esigenze diverse per lo stesso contenitore.
- I contenitori sono per impostazione predefinita sicuri e omogenei (ovvero, tutti gli elementi in un contenitore sono dello stesso tipo). Un contenitore eterogeneo può essere previsto come contenitore omogeneo di puntatori a una base comune.
- I contenitori non sono intrusivi (ovvero, un oggetto non ha bisogno di una classe base speciale o di un campo di collegamento per essere un membro di un contenitore). I contenitori non intrusivi funzionano bene con i tipi incorporati e con le strutture con layout imposti dall'esterno.

Passiamo adesso ai difetti:

- Ogni accesso all'iteratore comporta il sovraccarico di una chiamata di funzione virtuale. Il sovraccarico può essere molto alto rispetto alle semplici funzioni di accesso.
- Una gerarchia di classi di iteratori tende a complicarsi.
- Non c'è niente in comune per ogni contenitore e niente in comune per ogni oggetto in ogni contenitore. Ciò complica la fornitura di servizi universali come la persistenza e l'I/O di oggetti.

Considero particolarmente importante la flessibilità fornita dagli iteratori. Un'interfaccia comune, come Itor, può essere fornita molto tempo dopo la progettazione e l'implementazione dei container (in questo caso, Vector e List). Quando progettiamo, in genere inventiamo prima qualcosa di abbastanza concreto. Ad esempio, progettiamo un array e inventiamo una lista. Solo più tardi scopriamo un'astrazione che copre sia gli array che le liste in un dato contesto. In effetti, possiamo fare questa "astrazione tardiva" più volte. Supponiamo di voler rappresentare un insieme. Un set è

un'astrazione molto diversa da Itor, tuttavia possiamo fornire un'interfaccia Set a Vector e List nello stesso modo in cui ho fornito a Itor come un'interfaccia per Vector e List:



Pertanto, l'astrazione tardiva utilizzando classi astratte ci consente di fornire diverse implementazioni di un concetto anche quando non vi è alcuna somiglianza significativa tra le implementazioni. Ad esempio, liste e vettori hanno alcune cose in comune ovviamente, ma potremmo facilmente implementare un Itor per un istream. Logicamente, gli ultimi due punti della lista sono i principali punti deboli dell'approccio. Cioè, anche se l'overhead della chiamata di funzione per iteratori e interfacce simili ai contenitori fosse nullo (come è possibile in alcuni contesti), questo approccio non sarebbe l'ideale per una libreria standard. I contenitori non intrusivi comportano un piccolo sovraccarico di tempo e spazio per alcuni contenitori rispetto ai contenitori intrusivi. Non ritengo che questo sia un problema, qualora dovesse diventarlo, un iteratore come Itor può essere reso compatibile per un contenitore intrusivo.

16

ALGORITMI E FUNZIONI

Un contenitore di per sé non è davvero così interessante. Per essere veramente utile, un contenitore deve essere supportato da operazioni di base come trovare le dimensioni, iterare, copiare, ordinare e cercare elementi. Fortunatamente, la libreria standard fornisce algoritmi per soddisfare le esigenze più comuni e degli utenti base per i contenitori. Questo capitolo riassume gli algoritmi standard e fornisce alcuni esempi dei loro usi, una presentazione dei principi chiave e delle tecniche usate per esprimere gli algoritmi in C++ e una spiegazione di alcuni algoritmi chiave. Gli oggetti funzione forniscono un meccanismo attraverso il quale un utente può personalizzare il comportamento degli algoritmi standard. Gli oggetti funzione forniscono le informazioni chiave di cui un algoritmo ha bisogno per operare sui dati di un utente. Di conseguenza, l'enfasi è posta sul modo in cui gli oggetti funzione possono essere definiti e utilizzati.

A prima vista, gli algoritmi della libreria standard possono sembrare tantissimi, tuttavia, ce ne sono solo 60. Ho visto classi con più funzioni membro. Inoltre, molti algoritmi condividono un comportamento di base comune e uno stile di interfaccia comune che ne facilita la comprensione. Come per le funzionalità linguistiche, un programmatore dovrebbe utilizzare gli algoritmi effettivamente necessari, e solo quelli. Non ci sono premi per l'utilizzo del maggior numero di algoritmi standard in un programma, né ci sono premi per l'utilizzo di algoritmi standard nel modo più intelligente e oscuro. Ricorda, uno degli obiettivi principali della scrittura del codice è chiarirne il significato alla prossima persona che lo legge – e quella persona potrebbe essere te stesso tra qualche anno. D'altra parte, quando si esegue qualcosa con gli elementi di un contenitore, considera se tale azione può essere espressa come un algoritmo nello stile della libreria standard. Quell'algoritmo potrebbe già esistere. Se non consideri il lavoro in termini di algoritmi generali, ti troverai a dover re-inventerare la ruota. Ciascun algoritmo è espresso come una funzione modello o un insieme di funzioni modello. In questo modo, un algoritmo può operare su molti tipi di sequenze contenenti elementi di vari tipi. Di conseguenza, gli algoritmi che restituiscono un iteratore utilizzano generalmente la fine di una sequenza di input per indicare un errore. Per esempio:

```
void f(list<string>& ls)
{
list<string>::const_iterator p= find(ls.begin() ,ls.end() ,"Fred") ;
if(p== ls.end()) {
// didn’t find "Fred"
}
else{
// here, p points to "Fred"
}
}
```

GLI ALGORITMI non eseguono il controllo dell'intervallo sul loro input o output. Gli errori sul range devono essere prevenuti con altri mezzi. Quando un algoritmo restituisce un iteratore, quell'iteratore è dello stesso tipo di uno dei suoi input. In particolare, gli argomenti di un algoritmo controllano se restituisce un const_iterator o un iteratore non const. Per esempio:

```
void f(list<int>& li, const list<string>& ls)
{
list<int>::iterator p= find(li.begin() ,li.end() ,42) ;
list<string>::const_iterator q= find(ls.begin() ,ls.end() ,"Ring") ;
}
```

GLI ALGORITMI nella libreria standard coprono le operazioni generali più comuni sui contenitori come lettura sequenziale, ordinamento, ricerca, inserimento e rimozione di elementi. Gli algoritmi standard sono tutti nello namespace std e le loro dichiarazioni si trovano in <algorithm>. È interessante notare che la maggior parte degli algoritmi realmente comuni sono così semplici che le funzioni del template sono tipicamente inline. Ciò implica che i loop espressi dagli algoritmi beneficiano di un'ottimizzazione aggressiva per funzione. Anche gli oggetti funzione standard si trovano nel namespace std, ma le loro dichiarazioni si trovano in <functional>. Gli oggetti funzione sono progettati per essere facilmente inline.

La maggior parte degli algoritmi consente all'utente di specificare l'azione effettiva eseguita per ciascun elemento o coppia di elementi.

Questo rende gli algoritmi molto più generali e utili di quanto appaiano a prima vista. In particolare, un utente può fornire i criteri utilizzati per l'uguaglianza e la differenza. Ove ragionevole, l'azione più comune e utile viene fornita come impostazione predefinita.

Ogni buon design mostra tracce dei tratti personali e degli interessi del suo designer. I contenitori e gli algoritmi nella libreria standard riflettono chiaramente una forte preoccupazione per le strutture dati classiche e la progettazione di algoritmi. La libreria standard fornisce non solo il minimo indispensabile di contenitori e algoritmi necessari essenzialmente a ogni programmatore. Include anche molti degli strumenti utilizzati per fornire quegli algoritmi, necessari per estendere la libreria. L'enfasi qui non è sulla progettazione di algoritmi, nemmeno sull'uso di qualsiasi algoritmo, ma è sull'uso del più semplice e ovvio. Questo focus consente a chi conosce gli algoritmi di utilizzare bene la libreria e di estenderla con lo spirito per cui è stata costruita.

È un buon principio generale che l'uso più comune di qualcosa dovrebbe essere anche il più breve, il più facile da esprimere e il più sicuro. La libreria standard viola questo principio in nome della generalità. Per una libreria standard, la generalità è essenziale. Ad esempio, possiamo trovare le prime due occorrenze di 42 in una lista come questa:

```
void f(list<int>& li)
{
list<int>::iterator p= find(li.begin() ,li.end() ,42) ; // first occurrence
if(p!= li.end()) {
list<int>::iterator q= find(++p,li.end() ,42) ; // second occurrence
// ...
}
// ...
}
```

SE FIND() FOSSE STATO ESPRESSO come un'operazione su un container, avremmo avuto bisogno di qualche meccanismo aggiuntivo per trovare la seconda occorrenza. È importante sottolineare che generalizzare un tale "meccanismo aggiuntivo" per ogni contenitore e ogni algoritmo è difficile. Al contrario, gli algoritmi di libreria standard funzionano su sequenze di elementi cioè, l'input di un algoritmo è espresso come una coppia di

iteratori che delineano una sequenza. Il primo iteratore si riferisce al primo elemento della sequenza e il secondo si riferisce ad un punto oltre l'ultimo elemento. Tale sequenza è chiamata "semiaperta" perché include il primo valore menzionato e non il secondo. Una sequenza semiaperta consente di esprimere molti algoritmi senza rendere la sequenza vuota un caso speciale.

Una sequenza, in particolare una sequenza in cui è possibile l'accesso casuale, è spesso chiamata intervallo o range. Le notazioni matematiche tradizionali per un intervallo semiaperto sono [first,last) e [first,last[. È importante sottolineare che una sequenza può essere composta dagli elementi di un contenitore o da una sottosequenza di un contenitore. Inoltre, alcune sequenze, come i flussi di I/O, non sono contenitori. Tuttavia, gli algoritmi espressi in termini di sequenze funzionano decisamente bene.

Scrivere x.begin(),x.end() per esprimere "tutti gli elementi di x" è comune, noioso e può anche essere soggetto a errori. Ad esempio, quando vengono utilizzati più iteratori, è troppo facile fornire un algoritmo con una coppia di argomenti che non costituiscono una sequenza:

```
void f(list<string>& fruit, list<string>& citrus)
{
typedef list<string>::const_iterator LI;
LI p1= find(fruit.begin() ,citrus.end() ,"apple") ; // wrong! (different sequences)
LI p2= find(fruit.begin() ,fruit.end() ,"apple") ; // ok
LI p3= find(citrus.begin() ,citrus.end() ,"pear") ; // ok
LI p4= find(p2,p3,"peach") ; // wrong! (different sequences)
// ...
}
```

In questo esempio sono presenti due errori. Il primo è ovvio (una volta che si sospetta un errore), ma non è facilmente rilevabile da un compilatore. Il secondo è difficile da individuare nel codice reale anche per un programmatore esperto. Ridurre il numero di iteratori espliciti usati, riduce questo problema. L'idea chiave è di essere espliciti sul prendere una sequenza come input. Per esempio:

```
template<class In, class T> In find(In first, In last, const T& v) // standard
{
```

```
while(first!=last&& *first!=v) ++first;
return first;
}
template<class In, class T> In find(Iseq<In> r, const T& v) // extension
{
return find(r.first,r.second,v) ;
}
```

In generale, l'overloading consente di preferire la versione della sequenza di input di un algoritmo quando viene utilizzato un argomento Iseq. Naturalmente, una sequenza di input viene implementata come una coppia di iteratori:

```
template<class In> struct Iseq: public pair<In,In> {
Iseq(In i1, In i2) : pair<In,In>(i1,i2) { }
};
```

Possiamo rendere esplicitamente necessario l'Iseq per invocare la seconda versione di find():

```
LI p= find(Iseq<LI>(fruit.begin() ,fruit.end()) ,"apple") ;
```

Tuttavia, è ancora più noioso che chiamare direttamente l'originale find(). Semplici funzioni di supporto migliorano il nostro lavoro. In particolare, l'Iseq di un contenitore è la sequenza di elementi dal suo begin() al suo end():

```
template<class C> Iseq<C::iterator_type> iseq(C& c) // for container
{
return Iseq<C::iterator_type>(c.begin() ,c.end()) ;
}
```

Questo ci permette di esprimere algoritmi su contenitori in modo compatto e senza ripetizioni. Per esempio:

```
void f(list<string>& ls)
{
list<string>::iterator p= find(ls.begin() ,ls.end() ,"standard") ;
list<string>::iterator q= find(iseq(ls) ,"extension") ;
// ..
}
```

È facile definire versioni di iseq() che producono Iseq per array, flussi di input, ecc. Il vantaggio principale di Iseq è che rende esplicita la nozione di sequenza di input. L'effetto pratico immediato è che l'uso di iseq() elimina gran parte della noiosa ripetizione (soggetta ad errori) necessaria per esprimere ogni sequenza di input come una coppia di iteratori. Tuttavia, è meno semplice e meno utile nell'immediato rispetto alla nozione di sequenza di input.

Oggetti funzione

Molti algoritmi operano su sequenze utilizzando solo iteratori e valori. Ad esempio, possiamo trovare il primo elemento con il valore 7 in una sequenza come questa:

```
void f(list<int>& c)
{
list<int>::iterator p= find(c.begin() ,c.end() ,7) ;
// ...
}
```

PER FARE cose più interessanti vogliamo che gli algoritmi eseguano il codice che forniamo. Ad esempio, possiamo trovare il primo elemento in una sequenza con un valore inferiore a 7 in questo modo:

```
bool less_than_7(int v)
{
return v<7;
}
void f(list<int>& c)
{
list<int>::iterator p= find_if(c.begin() ,c.end() ,less_than_7) ;
// ...
}
```

CI SONO molti usi ovvi per le funzioni passate come argomenti: predicati logici, operazioni aritmetiche, operazioni per estrarre informazioni dagli elementi, ecc. Non è né conveniente né efficiente scrivere una funzione separata per ogni uso, né una funzione è logicamente sufficiente per esprimere tutto ciò che vorremmo esprimere. Spesso, la funzione chiamata per ogni elemento deve mantenere i dati tra le chiamate e restituire il risultato di molte applicazioni. Una funzione membro di una classe soddisfa tali esigenze meglio di una funzione indipendente perché il suo oggetto può contenere dati. Inoltre, la classe può fornire operazioni per l'inizializzazione

e l'estrazione di tali dati. Considera come scrivere una funzione, o meglio una classe simile a una funzione, per calcolare una somma:

```
template<class T> class Sum{
T res;
public:
Sum(T i= 0) : res(i) { } // initialize
void operator()(T x) { res+= x; } // accumulate
T result() const{ return res; } // return sum
};
```

CHIARAMENTE, Sum è progettata per tipi aritmetici per i quali sono definite l'inizializzazione per 0 e +=. Per esempio:

```
void f(list<double>& ld)
{
Sum<double> s;
s= for_each(ld.begin() ,ld.end() ,s) ; // invoke s() for each element of ld
cout<< "the sum is" << s.result() <<'\n';
}
```

QUI, for_each() invoca Sum<double>::operator()(double) per ogni elemento di ld e restituisce l'oggetto passato come terzo argomento. Il motivo principale per cui funziona è che for_each() in realtà non presuppone che il suo terzo argomento sia una funzione. Presuppone semplicemente che il suo terzo argomento sia qualcosa che può essere chiamato con un argomento appropriato. Un oggetto opportunamente definito serve come – e spesso è meglio di – una funzione. Ad esempio, è più facile incorporare l'operatore dell'applicazione di una classe piuttosto che incorporare una funzione passata come puntatore a una funzione. Di conseguenza, gli oggetti funzione spesso vengono eseguiti più velocemente delle normali funzioni. Un oggetto di una classe con un operatore applicativo è chiamato oggetto simile a una funzione (function-like object), funtore (functor) o semplicemente oggetto funzione.

LE STRINGHE

Una stringa è una sequenza di caratteri. La stringa della libreria standard fornisce operazioni di manipolazione delle stringhe come pedice, assegnazione, confronto, aggiunta, concatenazione e ricerca di sottostringhe. Lo standard non fornisce alcuna funzione di sottostringa generale, quindi ne viene fornita una qui come esempio di utilizzo di stringhe standard. Una stringa standard può essere una stringa composta essenzialmente da qualsiasi tipo di carattere. L'esperienza dimostra che è impossibile progettare la stringa perfetta. I gusti, le aspettative e le esigenze delle persone differiscono troppo per questo obiettivo. Quindi, la stringa della libreria standard non è l'ideale. Avrei preso alcune decisioni di progettazione in modo diverso, e probabilmente l'avresti fatto anche tu. Tuttavia, soddisfa bene molte esigenze, le funzioni ausiliarie per soddisfare ulteriori bisogni sono fornite facilmente ed è generalmente noto e disponibile std::string. Nella maggior parte dei casi, questi fattori sono più importanti di qualsiasi miglioramento minore che potremmo fornire. Scrivere classi di stringhe ha un grande valore educativo, ma per il codice destinato ad essere ampiamente utilizzato, la stringa della libreria standard è quella da usare. C++, ha ereditato da C la nozione di stringhe come array di char con terminazione zero e un insieme di funzioni per manipolare tali stringhe in stile C.

Un carattere è di per sé un concetto interessante. Considera il carattere C, la C che vedi come una linea curva sulla pagina (o sullo schermo), l'ho digitata sul mio computer molti mesi fa. Lì, vive come valore numerico 67 in un byte a 8 bit. È la terza lettera dell'alfabeto latino, la consueta abbreviazione del sesto atomo (Carbon) e, come saprai, il nome di un linguaggio di programmazione. Ciò che conta nel contesto della programmazione con stringhe è che vi sia una corrispondenza tra gli scarabocchi con significato convenzionale, chiamati caratteri e valori numerici. Per complicare le cose, lo stesso carattere può avere valori numerici diversi in diversi set di caratteri, non tutti i set di caratteri hanno valori per ogni carattere ed è comune trovare molti set di caratteri diversi.

Un set di caratteri è una mappatura tra un carattere (un simbolo convenzionale) e un valore intero. I programmatori C++ di solito presumono che sia disponibile il set di caratteri americano standard

(ASCII), ma C++ tiene conto della possibilità che alcuni caratteri possano mancare nell'ambiente di un programmatore. Ad esempio, in assenza di caratteri come [e {, è possibile utilizzare parole chiave e digrafi. I set di caratteri con caratteri non in ASCII offrono una difficoltà maggiore infatti lingue come cinese, danese, francese, islandese e giapponese non possono essere scritte correttamente utilizzando solo ASCII. Peggio ancora, i set di caratteri utilizzati per queste lingue possono essere reciprocamente incompatibili. Ad esempio, i caratteri usati per le lingue europee che utilizzano alfabeti latini rientrano quasi in un set di 256 caratteri. Sfortunatamente, set diversi vengono ancora utilizzati per lingue diverse e alcuni caratteri diversi usano con lo stesso valore intero. Ad esempio, il francese (usando latin1) non convive bene con l'islandese (che quindi richiede latin2). I tentativi ambiziosi di presentare tutti i caratteri conosciuti dall'uomo in un unico set di caratteri hanno fatto progressi, ma anche set di caratteri a 16 bit, come Unicode, non sono sufficienti per soddisfare tutti. I set di caratteri a 32 bit che potrebbero, per quanto ne so, contenere tutti i caratteri non sono ampiamente utilizzati.

Fondamentalmente, l'approccio C++ consiste nel consentire a un programmatore di utilizzare qualsiasi set di caratteri come tipo di carattere nelle stringhe. È possibile utilizzare un set di caratteri esteso o una codifica numerica portatile. Una stringa può, in linea di principio, utilizzare qualsiasi tipo con operazioni di copia appropriate come tipo di carattere, tuttavia, l'efficienza può essere nettamente migliorata e le implementazioni possono essere semplificate per i tipi che non hanno operazioni di copia definite dall'utente. Di conseguenza, la stringa standard richiede che un tipo utilizzato come tipo di carattere non disponga di operazioni di copia definite dall'utente. Questo aiuta anche a rendere l'I/O di stringhe semplice ed efficiente. Le proprietà di un tipo di carattere sono definite dai suoi char_traits. Un char_trait è una specializzazione del template:

```
template<class Ch> struct char_traits{ };
```

TUTTI I CHAR_TRAITS sono definiti in std e quelli standard sono presentati in <string>. Lo stesso char_traits generico non ha proprietà; solo le specializzazioni di char_traits le hanno per un dato tipo di carattere. Considera char_traits<char>:

```
template<> struct char_traits<char> {
```

```
    typedef char char_type; // type of character
    static void assign(char_type&, const char_type&) ; // = for char_type
    // integer representation of characters:
    typedef int int_type; // type of integer value of character
    static char_type to_char_type(const int_type&) ; // int to char conversion
    static int_type to_int_type(const char_type&) ; // char to int conversion
    static bool eq_int_type(const int_type&, const int_type&) ; // ==
    // char_type comparisons:
    static bool eq(const char_type&, const char_type&) ; // ==
    static bool lt(const char_type&, const char_type&) ; // <
    // operations on s[n] arrays:
    static char_type* move(char_type* s, const char_type* s2, size_t n) ;
    static char_type* copy(char_type* s, const char_type* s2, size_t n) ;
    static char_type* assign(char_type* s, size_t n, char_type a) ;
    static int compare(const char_type* s, const char_type* s2, size_t n) ;
    static size_t length(const char_type*) ;
    static const char_type* find(const char_type* s, int n, const char_type&) ;
    // I/O related:
    typedef streamoff off_type; // offset in stream
    typedef streampos pos_type; // position in stream
    typedef mbstate_t state_type; // multi-byte stream state
    static int_type eof() ; // end-of-file
    static int_type not_eof(const int_type& i) ; // i unless i equals eof(); if not any value!=eof()
    static state_type get_state(pos_type p) ; // multibyte conversion state of character in p
    };
```

L'IMPLEMENTAZIONE del modello di stringa standard, basic_string, si basa su questi tipi e funzioni. Un tipo usato come tipo di carattere per basic_string deve fornire una specializzazione char_traits che li fornisca tutti. Affinché un tipo sia un char_traits, deve essere possibile ottenere un valore intero corrispondente a ciascun carattere. Il tipo di quel numero intero è int_type e la conversione tra esso e char_type viene eseguita da

to_char_type() e to_int_type(). Per un char, questa conversione è banale. Sia move(s,s2,n) che copy(s,s2,n) copiano n caratteri da s2 a s usando assign(s[i] ,s2[i]). La differenza è che move() funziona correttamente anche se s2 è nell'intervallo [s,s+n[. Pertanto, copy() può essere più veloce. Questo rispecchia le funzioni standard della libreria C memcpy() e memmove(). Una chiamata assign(s,n,x) assegna copie di x in s usando assign(s[i] ,x).

La funzione compare() usa lt() ed eq() per confrontare i caratteri. Restituisce un int, dove 0 rappresenta una corrispondenza esatta, un numero negativo significa che il suo primo argomento viene lessicograficamente prima del secondo e un numero positivo significa che il suo primo argomento viene dopo il secondo. Ciò rispecchia la funzione standard della libreria C strcmp().

Le funzioni relative agli I/O vengono utilizzate dall'implementazione di I/O di basso livello. Un carattere wide – cioè un oggetto di tipo wchar_t – è come un char, tranne per il fatto che occupa due o più byte. Le proprietà di un wchar_t sono descritte da char_traits<wchar_t>:

```
template<> struct char_traits<wchar_t> {
typedef wchar_t char_type;
typedef wint_t int_type;
typedef wstreamoff off_type;
typedef wstreampos pos_type;
// like char_traits<char>
};
```

UN WCHAR_T VIENE in genere utilizzato per contenere i caratteri di un set di caratteri a 16 bit come Unicode.

Le funzionalità di stringa della libreria standard si basano sul modello basic_string che fornisce tipi di membri e operazioni simili a quelle fornite dai contenitori standard:

```
template<class Ch, class Tr= char_traits<Ch>, class A= allocator<Ch>
>
class std::basic_string{
public:
// ...
};
```

QUESTO MODELLO e le sue strutture associate sono definiti nel namespace std e presentati da <string>. Due typedef forniscono i nomi convenzionali per i tipi di stringhe comuni:

```
typedef basic_string<char> string;
typedef basic_string<wchar_t> wstring;
```

UN BASIC_STRING È SIMILE A VECTOR, tranne per il fatto che basic_string fornisce alcune operazioni tipiche sulle stringhe, come la ricerca di sottostringhe, invece dell'insieme completo di operazioni offerte da vector. È improbabile che una stringa venga implementata da un semplice array o vettore. Esistono implementazioni per molti usi comuni delle stringhe che riducono al minimo la copia, non utilizzano un archivio libero per stringhe brevi, consentono la semplice modifica di stringhe più lunghe, ecc. Il numero di funzioni delle stringhe riflette l'importanza della manipolazione delle stringhe e anche il fatto che alcune macchine forniscono istruzioni hardware specializzate per la manipolazione delle stringhe. Tali funzioni sono più facilmente utilizzate da un implementatore di libreria se esiste una funzione di libreria standard con semantica simile.

Come altri tipi di libreria standard, un basic_string<T> è un tipo concreto senza funzioni virtuali. Può essere utilizzato come membro durante la progettazione di classi di manipolazione del testo più sofisticate, ma non è inteso come base per classi derivate.

Spesso le stringhe vengono semplicemente lette, scritte, stampate, memorizzate, confrontate, copiate, ecc. Questo non causa problemi o, nel peggiore dei casi, problemi di prestazioni. Tuttavia, una volta che iniziamo a manipolare singole sottostringhe e caratteri per comporre nuovi valori di stringa da quelli esistenti, prima o poi commettiamo errori che potrebbero farci scrivere oltre la fine di una stringa. Per l'accesso esplicito ai singoli caratteri, at() controlla se proviamo ad accedere oltre la fine della stringa e lancia out_of_range(); [] non ha lo stesso comportamento.

La maggior parte delle operazioni sulle stringhe prende una posizione di carattere più un numero di caratteri. Una posizione più grande della dimensione della stringa genera un'eccezione di tipo out_of_range. Un

numero di caratteri "troppo grande" viene semplicemente considerato equivalente a "il resto" dei caratteri. Per esempio:

```
void f()
{
string s= "Snobol4";
string s2(s,100,2) ; // character position beyond end of string: throw out_of_range()
string s3(s,2,100) ; // character count too large: equivalent to s3(s,2,s.size()– 2)
string s4(s,2,string::npos) ; // the characters starting from s[2]
}
```

PERTANTO, LE POSIZIONI "TROPPO GRANDI" devono essere evitate, ma i conteggi dei caratteri "troppo grandi" sono utili. In effetti, npos è in realtà solo il valore più grande possibile per size_type. Potremmo provare a dare una posizione negativa o un conteggio dei caratteri come segue:

```
void g(string& s)
{
string s5(s,-2,3) ; // large position!: throw out_of_range()
string s6(s,3,-2) ; // large character count!: ok
}
```

TUTTAVIA, il size_type utilizzato per rappresentare posizioni e conteggi è un tipo senza segno, quindi un numero negativo è semplicemente un modo confuso per specificare un numero positivo. Si noti che le funzioni utilizzate per trovare sottostringhe di una stringa restituiscono npos se non trovano nulla, pertanto, non generano eccezioni. Cosa che non succede, utilizzando in seguito npos come una posizione del carattere.

Una coppia di iteratori è un altro modo per specificare una sottostringa. Il primo iteratore identifica una posizione e la differenza tra due iteratori è il conteggio dei caratteri. Come al solito, gli iteratori non sono controllati dall'intervallo. Laddove viene utilizzata una stringa in stile C, il controllo dell'intervallo è più difficile. Quando viene assegnata una stringa in stile C (un puntatore a char) come argomento, la funzione basic_string presuppone

che il puntatore non sia 0. Quando vengono fornite posizioni dei caratteri per le stringhe in stile C, presuppongono che la stringa in stile C sia abbastanza lunga da consentire alla posizione di essere valido.

Stai attento! In questo caso, stare attenti significa essere paranoici, tranne quando si usano caratteri letterali. Tutte le stringhe hanno length() < npos. In alcuni casi, come l'inserimento di una stringa in un'altra, è possibile (sebbene non probabile) costruire una stringa troppo lungo per essere rappresentata. In tal caso, viene generato un length_error. Per esempio:

string s**(**string**: :**npos**,** ´a´**) ;** // throw length_error()

CONCLUSIONE

Alla luce di tutto ciò e con una miriade di linguaggi disponibili sorge una domanda: perché usare ancora C++? Si usa ancora per via delle possibilità che fornisce rispetto agli altri linguaggi.

Le motivazioni principali per cui preferire il C++ rispetto a linguaggi come Swift, Java oppure Python sono:

- Le prestazioni: La principale filosofia dietro al linguaggio C++ è di lasciare al programmatore il pieno controllo sulle risorse utilizzate, la loro allocazione e de-allocazione in memoria. In questo campo non fornisce alcuna funzionalità di gestione automatica delle risorse, infatti al contrario di Java, non fornisce alcun algoritmo di Garbage Collection. Inoltre, la semantica di base, così a basso livello, consente un controllo diretto su come i dati vengono maneggiati e gestiti a livello hardware dalla macchina su cui si sta eseguendo il proprio codice, ad esempio, attraverso l'utilizzo di costrutti come i puntatori, che nei linguaggi di più alto livello, come ad esempio il Java, viene demandata al compilatore/interprete. C++ è un linguaggio che si è dimostrato in grado di garantire altissime prestazioni, irraggiungibili dalle controparti, oltre che una versatilità notevole. Tutto questo lo rende, assieme al suo fratello maggiore il C, l'unica scelta fattibile quando si necessità di scrivere software a latenza molto bassa (software real-time) oppure si deve lavorare con hardware che forniscono risorse limitate.
- La sua semantica di basso livello: C++ è stato costruito, inizialmente, come un super-insieme del C, con un obbiettivo ben preciso: quello di fornire un linguaggio ben orientato ai sistemi, che tuttavia permettesse l'utilizzo del paradigma OOP. Sicuramente, ciò ha influenzato notevolmente la scelta di usare il C come punto di partenza, poiché era un linguaggio particolarmente indirizzato per la programmazione di sistemi ed anche largamente impiegato verso tal scopo. Questo rende C++ un linguaggio incredibilmente pratico anche per lo sviluppo di software di livello più basso e orientato nativamente verso la

programmazione di sistemi, ma comunque munito della potenza di un moderno linguaggio ad oggetti, perciò, permette anche un approccio di più alto livello.
- Linguaggio General Purpose: C++ è costruito per essere un linguaggio adatto in svariati campi applicativi e consente di farci praticamente ogni cosa. Questa caratteristica, unita alla sua semantica di basso livello e al suo approccio multi-paradigmatico, lo rende un linguaggio incredibilmente potente, quasi ineguagliabile dalle controparti citate precedentemente.

Questi sono gli aspetti di rilievo per capire il perché usare ancora C++, infatti, non solo è ancora largamente utilizzato, ma è tutt'oggi un linguaggio incredibilmente potente e insostituibile. La naturale maggior difficoltà del linguaggio, non è altro che una conseguenza degli aspetti appena descritti.

Tuttavia, il miglior modo per imparare un linguaggio consiste nella pratica quindi cerca di realizzare un progetto in C++ e vedrai che le tue abilità miglioreranno giorno dopo giorno.