Il linguaggio C++

Sommario

Introduzione	1
Elementi di base	3
Programmazione ad oggetti	113
Programmazione generica	225
Gestione degli errori	243
Libreria standard	269
Esercizi	311

Introduzione

Obiettivo

introdurre alla programmazione in C++, spiegando sintassi e semantica dei suoi costrutti anche con l'ausilio di opportuni esempi. Verranno inizialmente trattati gli aspetti basilari del linguaggio (tipi, dichiarazioni di variabili, funzioni,...), per poi passare all'esame degli aspetti peculiari del linguaggio (classi, template, eccezioni...); alla fine analizzeremo (almeno in parte) l'input/output tramite stream e la libreria standard del linguaggio.

Caratteristiche generali

linguaggio di programmazione “general purpose", ovvero adatto alla realizzazione di qualsiasi tipo di applicazione, da quelle real time a quelle che operano su basi di dati, da applicazioni per utenti finali a sistemi operativi.

compatibilita` (quasi assoluta) con C, il suo diretto antenato, da cui eredita la sintassi e la semantica per tutti i costrutti comuni, oltre alla notevole flessibilita` e potenza; ne estende le caratteristiche, rimediando almeno in parte alle carenze del suo predecessore (che manca soprattutto di un buon sistema dei tipi). In particolare l'introduzione di costrutti quali i Template e le Classi;

Portabilità: Possibilita` di portare facilmente le applicazioni verso altri sistemi;

Complessità; grande lavoro in fase di progettazione e una maggiore attenzione ai particolari in fase di realizzazione (implementazione), pena una valanga di errori spesso subdoli e difficili da individuare;

Problemi di ottimizzazione: il compilatore nella stragrande maggioranza dei casi si limita ad eseguire le ottimizzazioni tradizionali, sostanzialmente valide in linguaggi come il C, ma spesso inadatte a linguaggi pesantemente basati sulla programmazione ad oggetti.

2

Hello world!

```
//File: hello.cpp

#include <iostream>
using namespace std;

void main()
{
  cout << "Hello World!" << endl;
}
```

3

Elementi di base

0

Elementi lessicali

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

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

Commenti;

Identificatori;

Parole riservate;

Costanti letterali;

Segni di punteggiatura e operatori;

1

Commenti

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

secondo lo stile C ovvero racchiudendoli tra i simboli /* e */

facendoli precedere dal simbolo //

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

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


Commenti

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

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

Commenti

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

```
a + c       //* commento */ su
            una sola riga
```

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

4

Identificatori

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

Variabili: contenitori di valori di un qualche tipo; ogni variabile può contenere un singolo valore che può cambiare nel tempo, il tipo di questo valore viene comunque stabilito una volta per tutte e non può cambiare;

Costanti simboliche: valori che non cambiano nel tempo, non possono essere considerate dei contenitori, ma solo un nome per un valore;

Etichette: nomi il cui compito e` quello di identificare una istruzione del programma e sono utilizzate dall'istruzione di salto incondizionato *goto*.

Tipi definiti dal programmatore: insiemi di valori e di operazioni definiti su questi valori; ogni linguaggio (o quasi) fornisce un certo numero di *tipi primitivi* (cui e` associato un identificatore predefinito) e dei meccanismi per permettere la costruzione di nuovi tipi (a cui il programmatore deve poter associare un nome) a partire da questi.

Funzioni: sottoprogrammi (subroutines).

5

Identificatori

In effetti potremmo considerare una sesta categoria di identificatori, gli identificatori di macro; una macro e` sostanzialmente un alias per un frammento di codice. Le macro comunque, come vedremo in seguito, non sono trattate dal compilatore ma da un *preprocessore* che si occupa di eseguire alcune elaborazioni sul codice sorgente prima che questo venga effettivamente sottoposto a compilazione.

Un identificatore deve iniziare con una lettera o con carattere di underscore (_) che possono essere seguiti da un numero qualsiasi di lettere, cifre o underscore; viene fatta distinzione tra lettere maiuscole e lettere minuscole. Tutti gli identificatori presenti in un programma devono essere diversi tra loro, indipendentemente dalla categoria cui appartengono.

Benché il linguaggio non preveda un limite alla lunghezza massima di un identificatore, e` praticamente impossibile non imporre un limite al numero di caratteri considerati significativi, per cui ogni compilatore distingue gli identificatori in base a un certo numero di caratteri iniziali tralasciando i restanti; il numero di caratteri considerati significativi varia comunque da sistema a sistema.

Parole riservate

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

asm auto bool break case catch char class const continue const_cast default delete do double dynamic_cast else enum explicit extern false float for friend goto if inline int long mutable namespace new operator private protected public register reinterpret_cast return short signed sizeof static static_cast struct switch template this trow true try typedef typeid typename union unsigned using virtual void volatile wchar_t while

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

Costanti letterali

All'interno delle espressioni e` possibile inserire direttamente dei valori, questi valori sono detti costanti letterali. La generica costante letterale può essere:

un carattere racchiuso tra apice singolo,
una stringa racchiusa tra doppi apici,
un intero
un numero in virgola mobile.

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

8

Costanti letterali

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

Nella rappresentazione in esadecimale, oltre alle cifre decimali, e` consentito l'uso delle lettere da "A" a "F" e da "a" a "f".
Si noti che un segno puo` essere espresso solo in base 10, negli altri casi esso e` sempre +:

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

9

Costanti letterali

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

Base 10:

il piu` piccolo atto a contenerla tra int, long int e unsigned long int

Base 8 o 16:

il piu` piccolo atto a contenerla tra int, unsigned int, long int e unsigned long int

10

Costanti letterali

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

```
20          // intero in base 10
024         // 20 in base 8
0x14        // 20 in base 16
0x20ul      // forza unsigned long
0x20l       // forza long
0x20u       // forza unsigned
```

11

Costanti letterali

Un valore in virgola mobile e` costituito da:

Intero decimale, opzionalmente con segno;

Punto decimale

Frazione decimale;

e o E e un intero decimale con segno;

L'uso della lettera E indica il ricorso alla notazione scientifica.

E` possibile omettere uno tra l'intero decimale e la frazione decimale, ma non entrambi. E` possibile omettere uno tra il punto decimale e la lettera E (o e) e l'intero decimale con segno, ma non entrambi.

12

Costanti letterali

Il tipo scelto per rappresentare una costante in virgola mobile e` double, se non diversamente specificato utilizzando i suffissi F o f per float, o L o l per long double. Esempi:

```
.0          // 0 in virgola mobile
110E+4  // 110 * 10000
.14e-2      // 0.0014
-3.5e+3     // -3500.0
3.5f        // forza float
3.4L        // forza long double
```

13

Segni di punteggiatura e operatori

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

[ ] ( ) { } + - * % ! ^ &

= ~ | \ ; ' : " < > ? , .

Anche le seguenti combinazioni di simboli sono operatori:

++ -- -> .* ->* << >> <= >= == != &&

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

14

Assegnamento

Il C++ e` un linguaggio pesantemente basato sul paradigma imperativo, questo vuol dire che un programma C++ e` sostanzialmente una sequenza di assegnamenti di valori a variabili. E` quindi naturale iniziare parlando proprio dell'assegnamento.

L'operatore di assegnamento e` denotato dal simbolo = (uguale) e viene applicato con la sintassi:

< lvalue > = < rvalue >;

Il termine lvalue indica una qualsiasi espressione che riferisca ad una regione di memoria (in generale un identificatore di variabile), mentre un rvalue e` una qualsiasi espressione la cui valutazione produca un valore.

15

Assegnamento

Ecco alcuni esempi:

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

16

Assegnamento

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

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

17

Assegnamento

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

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

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

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

cioe` viene prima assegnato 5 alla variabile Pippo e il risultato di tale assegnamento (il valore 5) viene poi assegnato alla variabile Clarabella.

18

Assegnamento

Esistono anche altri operatori che hanno come effetto collaterale l'assegnazione di un valore, la maggior parte di essi sono comunque delle utili abbreviazioni, eccone alcuni esempi:

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

si tratta cioe` di operatori derivanti dalla concatenazione dell'operatore di assegnamento con un altro operatore binario.

L'uso di queste forme sintetiche e' da preferire per due ragioni:

1) maggiore chiarezza del codice

2) maggiore efficienza

19

Operatori di incremento/decremento

Gli altri operatori che hanno come effetto collaterale l'assegnamento sono quelli di autoincremento e autodecremento, ecco come possono essere utilizzati:

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

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

20

Operatori di incremento/decremento

```
Clarabella = ++Pippo;

 /* equivale a */

 Pippo++;
 Clarabella = Pippo;

 /* invece */

 Clarabella = Pippo++;

 /* equivale a */

 Clarabella = Pippo;
 Pippo++;
```

21

Altri operatori

Le espressioni, per quanto visto sopra, rappresentano un elemento basilare del C++, tant'e` che il linguaggio fornisce un ampio insieme di operatori. Nella seguente tabella di precedenza degli operatori del C++, questi sono suddivisi in 16 categorie.

La prima categoria ha la precedenza massima, la seconda (operatori unari) quella immediatamete successiva e cosi via fino all'operatore virgola, che ha la precedenza minima. Gli operatori di ciascuna categoria hanno la stessa precedenza.

Gli operatori unari (categoria 2), condizionali (categoria 14) e di assegnamento (categoria 15) vengono associati da destra a sinistra; tutti gli altri da sinistra a destra.

Altri operatori

#	Categoria	Associatività	Operatore	
1.	Massima	S -> D	()	Chiamata di funzione
			[]	Indicizzazione di array
			->	C++ selettore indiretto di component
			::	C++ accesso al campo d'azione
			.	C++ selettore diretto di componente
2.	Unari	S <- D	!	Negazione logica (NOT)
			~	Complemento ad 1 dei bit (Op. bit a bit)
			+	Più unario
			-	Meno unario
			++	Pre o post-incremento (applicabili solo a variabili e non ad espressioni)
			--	Pre o post-decremento

Altri operatori

			&	Indirizzo
			*	Indirezione (puntatori)
			sizeof	(dimensione in byte dell'operando)
			new	(C++ allocazione dinamica di memoria)
			delete	(C++ deallocazione di memoria)
			(tipo)	Casting
3.	Moltiplicativi	S -> D	*	Moltiplicazione
			/	Divisione
			%	Resto (modulo)
4.	Accesso ai membri	S -> D	.*	C++ dereferenza
			->*	C++ dereferenza
5.	Additivi	S -> D	+	Somma
			-	Sottrazione
6	Shift	S -> D	<<	Shift sinistro

24

Altri operatori

			>>	Shift destro
7.	Relazionali	S -> D	<	Minore
			<=	Minore od uguale
			>	Maggiore
			>=	Maggiore od uguale
8.	Eguaglianza	S -> D	==	Uguale
			!=	Diverso
9.	Op. bit a bit	S -> D	&	AND tra bit
10.	Op. bit a bit	S -> D	^	XOR tra bit
11.	Op. bit a bit	S -> D	|	OR tra bit
12. *	Op. logico	S -> D	&&	AND logico
13. *	Op. logico	S -> D	||	OR logico (per 12 e 13, la valutazione si blocco non appena si determina la verità o la falsità dell'intera espressione)
14. *	Condizionali	S <- D	?:	(a ? x : y significa "if a then x else y")

25

Altri operatori

15. Assegnamento S <- D = Assegnamento
*= Assegna prodotto
/= Assegna quoziente
(esp1 op= esp2) <=> %= Assegna resto (modulo)
(esp1 = esp1 op esp2) += Assegna somma
-= Assegna differenza
&= Assegna AND tra bit
^= Assegna XOR tra bit
|= Assegna OR tra bit
<<= Assegna shift sinistro
>>= Assegna shift destro
16. * Virgola S -> D , Valuta

26

Altri operatori

L'ordine di valutazione delle sottoespressioni che compongono una espressione piu` grande non e` definito, ad esempio nell'espressione

Pippo = 10 * 13 + 7 * 25;

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

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

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

27

Altri operatori

<Cond> ? <Expr1> : <Expr2>

Cond puo` essere una qualunque espressione che produca un valore booleano, se essa e` verificata il risultato di tale operatore e` la valutazione di Expr1, altrimenti il risultato e` Expr2.

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

28

Vero e falso

Prima che venisse approvato lo standard, il C++ non forniva un tipo primitivo (vedi tipi primitivi) per rappresentare i valori booleani. Esattamente come in C i valori di verita` venivano rappresentati tramite valori interi: 0 (zero) indicava falso e un valore diverso da 0 indicava vero. Cio` implicava che ovunque fosse richiesta una condizione era possibile mettere una qualsiasi espressione che producesse un valore intero (quindi anche una somma, ad esempio). Non solo, dato che l'applicazione di un operatore booleano o relazionale a due sottoespressioni produceva 0 o 1 (a seconda del valore di verita` della formula), era possibile mescolare operatori booleani, relazionali e aritmetici.

Il comitato per lo standard ha tuttavia approvato l'introduzione di un tipo primitivo appositamente per rappresentare valori di verita`. Come conseguenza di cio`, la` dove prima venivano utilizzati i valori interi per rappresentare vero e falso, ora si dovrebbero utilizzare il tipo bool e i valori true (vero) e false (falso), anche perche` i costrutti del linguaggio sono stati adattati di conseguenza.

29

Vero e falso

Comunque sia per compatibilita` con il C ed il codice C++ precedentemente prodotto e` ancora possibile utilizzare i valori interi, il compilatore converte automaticamente ove necessario un valore intero in uno booleano e viceversa (true viene convertito in 1):

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

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

30

IF-ELSE

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

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

oppure

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

L'else e` quindi opzionale, ma, se utilizzato, nessuna istruzione deve essere inserita tra il ramo if e il ramo else. Vediamo ora la semantica di tale istruzione.

In entrambi i casi se Condizione e` vera viene eseguita Istruzione1, altrimenti nel primo caso non viene eseguito alcunche`, nel secondo caso invece si esegue Istruzione2.

31

IF-ELSE

Si osservi che Istruzione1 e Istruzione2 sono istruzioni singole (una sola istruzione), se e` necessaria una sequenza di istruzioni esse devono essere racchiuse tra una coppia di parentesi graffe { }, come mostra il seguente esempio (si considerino X, Y e Z variabili intere):

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

32

IF-ELSE

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

Per risolvere eventuali ambiguita` il compilatore lega il ramo else con la prima occorrenza libera di if che incontra tornando indietro (si considerino Pippo, Pluto e Topolino variabili intere):

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

viene interpretata come:

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

l'else viene cioe` legato al secondo if.

33

WHILE & DO-WHILE

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

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

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

Esaminiamo ora l'altro costrutto:

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

Nuovamente, Istruzione indica una istruzione singola, se e` necessaria una sequenza di istruzioni essa deve essere racchiusa tra parentesi graffe.

34

WHILE & DO-WHILE

Il do while differisce dall'istruzione while in quanto prima si esegue Istruzione e poi si valuta Condizione, se essa e` vera si riesegue il corpo altrimenti l'istruzione termina; il corpo del do while viene quindi eseguito sempre almeno una volta. Ecco un esempio:

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

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

35

IL CICLO FOR

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

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

Inizializzazione puo` essere una espressione che inizializza le variabili del ciclo o una dichiarazione di variabili (nel qual caso le veriabili dichiarate hanno scope e lifetime limitati a tutto il ciclo); Condizione e` una qualsiasi espressione booleana; e Iterazione e` una istruzione da eseguire dopo ogni iterazione (solitamente un incremento). Tutti e tre gli elementi appena descritti sono opzionali, in particolare se Condizione non viene specificata si assume che essa sia sempre verificata.

36

IL CICLO FOR

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

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

Una eventuale istruzione continue (vedi di seguito) in Istruzione causa un salto a Iterazione nel caso del ciclo for, nel while invece causa un salto all'inizio del ciclo.
Ecco come usare il ciclo for per calcolare il fattoriale:

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

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

37

BREAK & CONTINUE

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

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

38

SWITCH

L'istruzione switch consente l'esecuzione di uno o piu` frammenti di codice a seconda del valore di una espressione:

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

Espressione e` una qualunque espressione capace di produrre un valore intero; Valore1...ValoreN sono costanti a valori interi; Istruzione e` una qualunque sequenza di istruzioni (non racchiuse tra parentesi graffe).

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

39

SWITCH

Ecco alcuni esempi:

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

```
switch (Pluto) {
        case 5 :
          Pippo = 3;
        case 6 :
          Pippo = 5;
        case 10 :
          Orazio = 20;
          Tip = 7;
       }  // niente caso default
```

40

SWITCH

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

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

41

GOTO

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

L'istruzione goto prevede che l'istruzione bersaglio del salto sia etichettata tramite un identificatore utilizzando la sintassi

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

che serve anche a dichiarare Etichetta.

Il salto ad una istruzione viene eseguito con

```
goto <Etichetta> ;
```

42

GOTO

ad esempio:

```
if (Pippo == 7) goto PLUTO;
  Topolino = 5;
  /* ... */

PLUTO : Pluto = 7;
```

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

43

Dichiarazioni

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

consente di stabilire la quantita` di memoria necessaria alla memorizzazione di un oggetto;

determina l'interpretazione da attribuire ai vari bit che compongono la regione di memoria utilizzata per memorizzare l'oggetto, l'insieme dei valori che puo` assumere e le operazioni che possono essere fatte su di esso;

permette l'esecuzione di opportuni controlli per determinare errori semantici;

fornisce eventuali suggerimenti al compilatore;

nel caso di funzioni, invece una dichiarazione:

determina numero e tipo dei parametri e il tipo del valore restituito;

consente controlli per determinare errori semantici;

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

44

Tipi primitivi

Un tipo e` una coppia < V, O >, dove V e` un insieme di valori e O e` un insieme di operazioni per la creazione e la manipolazione di elementi di V.

In un linguaggio di programmazione i tipi rappresentano le categorie di informazioni che il linguaggio consente di manipolare. Il C++ fornisce sei tipi fondamentali (o primitivi):

bool

char

wchar_t

int

float

double

45

Tipi primitivi

Abbiamo gia visto (vedi Vero e falso) il tipo bool e sappiamo che esso serve a rappresentare i valori di verita`; su di esso sono definite sostanzialmente le usuali operazioni logiche (&& per l'AND, || per l'OR, ! per la negazione...) e non ci soffermeremo oltre su di esse, solo si faccia attenzione a distinguerle dalle operazioni logiche su bit (rispettivamente &, |, ~...).

Il tipo char e` utilizzato per rappresentare piccoli interi (e quindi su di esso possiamo eseguire le usuali operazioni aritmetiche) e singoli caratteri; accanto ad esso troviamo anche il tipo wchar_t che serve a memorizzare caratteri non rappresentabili con char (ad esempio i caratteri unicode).

int e` utilizzato per rappresentare interi in un intervallo piu` grande di char.

Infine float e double rappresentano entrambi valori in virgola mobile, float per valori in precisione semplice e double per quelli in doppia precisione.

46

Tipi primitivi

Ai tipi fondamentali e` possibile applicare i qualificatori signed (con segno), unsigned (senza segno), short (piccolo) e long (lungo) per selezionare differenti intervalli di valori; essi tuttavia non sono liberamente applicabili a tutti i tipi: short si applica solo a int, signed e unsigned solo a char e int e infine long solo a int e double. In definitiva sono disponibili i tipi:

- bool
- char
- wchar_t
- short int
- int
- long int
- signed char
- signed short int
- signed int
- signed long int
- unsigned char
- unsigned short int
- unsigned int
- unsigned long int
- float
- double
- long double

47

Tipi primitivi

Il tipo int e` per default signed e quindi e` equivalente a tipo signed int, invece i tipi char, signed char e unsigned char sono considerate categorie distinte. I vari tipi sopra elencati, oltre a differire per l'intervallo dei valori rappresentabili, differiscono anche per la quantita` di memoria richiesta per rappresentare un valore di quel tipo (che pero` puo` variare da implementazione a implementazione). Il seguente programma permette di conoscere la dimensione di alcuni tipi come multiplo di char (di solito rappresentato su 8 bit), modificare il codice per trovare la dimensione degli altri tipi e` molto semplice e viene lasciato per esercizio:

48

Tipi primitivi

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

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

49

Tipi primitivi

Una veloce spiegazione sul listato:

le prime due righe permettono di utilizzare una libreria (standard) per eseguire l'output su video; la libreria iostream dichiara l'oggetto cout il cui compito e` quello di visualizzare l'output che gli viene inviato tramite l'operatore di inserimento <<.

L'operatore sizeof(<Tipo>) restituisce la dimensione di Tipo, mentre endl inserisce un ritorno a capo e forza la visualizzazione dell'output. L'ultima istruzione serve a terminare il programma. Infine main e` il nome che identifica la funzione principale, ovvero il corpo del programma, parleremo in seguito e piu` in dettaglio di main().

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

50

Variabili e costanti

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

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

Ad esempio:

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

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

La prima riga dichiara quattro variabili di tipo int, mentre la seconda una di tipo signed char. La terza dichiarazione e` un po' particolare in quanto apparentemente manca la keyword int, in realta` poiche` il default e` proprio int essa puo` essere omessa; in conclusione la terza dichiarazione introduce una variabile di tipo unsigned short int. Gli identificatori che seguono il tipo sono i nomi delle variabili, se piu` di un nome viene specificato essi devono essere separati da una virgola.

51

Variabili e costanti

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

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

La dichiarazione delle costanti e` identica a quella delle variabili eccetto che deve sempre essere specificato un valore e la dichiarazione inizia con la keyword const:

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

52

Scope e lifetime

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

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

53

Scope e lifetime

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

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


Scope e lifetime

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

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

Scope e lifetime

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

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

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


Scope e lifetime

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

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

Scope e lifetime

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

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

58

Costruire nuovi tipi

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

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

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

59

Array

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

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

introduce con il nome Array 10 variabili di tipo int (anche se solitamente si parla di una variabile di tipo array); il tipo di Array e` array di 10 int(eri).
La sintassi per la generica dichiarazione di un array e`

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

60

Array

NomeTipo puo` essere primitivo o definito dal programmatore, Identificatore e` un nome scelto dal programmatore per identificare l'array, mentre NumeroElementi deve essere un intero positivo e indica il numero di singole variabili che compongono l'array.
Il generico elemento dell'array viene selezionato con la notazione Identificatore[Espressione], dove Espressione puo` essere una qualsiasi espressione che produca un valore intero; il primo elemento di un array e` sempre Identificatore[0], e di conseguenza l'ultimo e` Identificatore[NumeroElementi-1]:

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

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

61

Array

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

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

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

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


Array

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

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

int Quo[4][3][2]   = { {{1, 2}, {3, 4}, {5, 6}},
                       {{7, 8}, {9, 10}, {11, 12}},
                       {{13, 14}, {15, 16}, {17, 18}},
{{19, 20}, {21, 22}, {23, 24}}
                     };
float Minni[ ]     = { 1.1, 3.5, 10.5 };
long Manetta[ ][3] = { {5, -7, 2}, {1, 0, 5} };
```

Array

La prima dichiarazione e` piuttosto semplice, dichiara un array di 5 elementi e per ciascuno di essi indica il valore iniziale a partire dall'elemento 0. Nella seconda riga viene dichiarata una matrice bidimensionale e se ne esegue l'inizializzazione, si noti l'uso delle parentesi graffe per raggruppare opportunamente i valori; la terza dichiarazione chiarisce meglio come procedere nel raggruppamento dei valori, si tenga conto che a variare per primo e` l'ultimo indice cosi` che gli elementi vengono inizializzati nell'ordine Quo[0][0][0], Quo[0][0][1], Quo[0][1][0], ..., Quo[3][2][1].

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

64

Array

Gli array consentono la memorizzazione di stringhe:

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

La dimensione dell'array e` pari a quella della stringa "investigatore" + 1, l'elemento in piu` e` dovuto al fatto che in C++ le stringhe per default sono tutte terminate dal carattere nullo ('\0')che il compilatore aggiunge automaticamente.

L'accesso agli elementi di Topolino avviene ancora tramite le regole viste sopra e non e` possibile eseguire un assegnamento con la stessa metodologia dell'inizializzazione:

65

Array

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

E` possibile inizializzare un array di caratteri anche nei seguenti modi:

```
char Minnie[ ]  = { 'M', 'i', 'n', 'n', 'i', 'e' };
char Pluto[5]   = { 'P', 'l', 'u', 't', 'o' };
```

In questi casi pero` non si ottiene una stringa terminata da '\0', ma semplici array di caratteri il cui numero di elementi e` esattamente quello specificato; a meno che non si inserisca un '\0' in ultima posizione nell'inizializzazione:

```
char Minnie[ ]  = { 'M', 'i', 'n', 'n', 'i', 'e', '\0' };
```

66

Strutture

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

67

Strutture

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

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

68

Strutture

La sintassi per la dichiarazione di una struttura e`

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

Si osservi che la parentesi graffa finale deve essere seguita da un punto e virgola, questo vale anche per le unioni, le enumerazioni e per le classi.

I singoli campi di una variabile di tipo struttura sono selezionabili tramite l'operatore di selezione . (punto), come mostrato nel seguente esempio:

69

Strutture

```
struct Persona {
  char Nome[20];
  unsigned short Eta;
  char CodiceFiscale[7];
};
Persona Pippo          = { "Pippo", 40, "PPP718" };
Persona AmiciDiPippo[2] = { {"Pluto", 40, "PLT712"},
                            {"Minnie", 35, "MNN431"}
                          };
```

Innanzi tutto viene dichiarato il tipo Persona e quindi si dichiara la variabile Pippo di tale tipo; in particolare viene mostrato come inizializzare la variabile con una inizializzazione aggregata del tutto simile a quanto si fa per gli array, eccetto che i valori forniti devono essere compatibili con il tipo dei campi e dati nell'ordine definito nella dichiarazione. Viene mostrata anche la dichiarazione di un array i cui elementi sono di tipo struttura, e il modo in cui eseguire una inizializzazione fornendo i valori necessari all'inizializzazione dei singoli campi di ciascun elemento dell'array.

Strutture

```
// esempi di uso di strutture:

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

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

Strutture

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

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

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

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

72

Strutture

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

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

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

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

73

Strutture

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

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

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

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


Unioni

Un costrutto sotto certi aspetti simile alle strutture e quello delle unioni. Sintatticamente l'unica differenza e` che nella dichiarazione di una unione viene utilizzata la keyword union invece di struct:

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

Come per i tipi struttura, la selezione di un dato campo di una variabile di tipo unione viene eseguita tramite l'operatore di selezione . (punto).

Unioni

Vi e` tuttavia una profonda differenza tra il comportamento di una struttura e quello di una unione: in una struttura i vari campi vengono memorizzati in indirizzi diversi e non si sovrappongono mai, in una unione invece tutti i campi vengono memorizzati a partire dallo stesso indirizzo. Cio` vuol dire che, mentre la quantita` di memoria occupata da una struttura e` data dalla somma delle quantita` di memoria utilizzata dalle singole componenti, la quantita` di memoria utilizzata da una unione e` data da quella della componente piu` grande (Stringa nell'esempio precedente).

Dato che le componenti si sovrappongono, assegnare un valore ad una di esse vuol dire distruggere i valori memorizzati accedendo all'unione tramite una qualsiasi altra componente.

Le unioni vengono principalmente utilizzate per limitare l'uso di memoria memorizzando negli stessi indirizzi oggetti diversi in tempi diversi. C'e` tuttavia un altro possibile utilizzo delle unioni, eseguire "manualmente" alcune conversioni di tipo. Tuttavia tale pratica e` assolutamente da evitare (almeno quando esiste una alternativa) poiche` tali conversioni sono dipendenti dall'architettura su cui si opera e pertanto non portabili, ma anche potenzialmente scorrette.

Enumerazioni

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

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

Esempio:

Enumerazioni

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

Elemento Atomo = Idrogeno;
```

78

Enumerazioni

Gli identificatori Idrogeno, Elio, Carbonio e Ossigeno costituiscono l'intervallo dei valori del tipo Elemento. Si osservi che come da sintassi, i valori di una enumerazione devono essere espressi tramite identificatori, non sono ammessi valori espressi in altri modi (interi, numeri in virgola mobile, costanti carattere...), inoltre gli identificatori utilizzati per esprimere tali valori devono essere distinti da qualsiasi altro identificatore visibile nello scope dell'enumerazione onde evitare ambiguita`.

Il compilatore rappresenta internamente i tipi enumerazione associando a ciascun identificatore di valore una costante intera, cosi` che un valore enumerazione puo` essere utilizzato in luogo di un valore intero, ma non viceversa:

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

79

Enumerazioni

```
};

Elemento Atomo = Idrogeno;
int Numero;

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

Nell'ultima riga dell'esempio si verifica un errore perche` non esiste un operatore di conversione da int a Elemento, mentre essendo i valori enumerazione in pratica delle costanti intere, il compilatore e` in grado di eseguire la conversione a int.

80

Enumerazioni

E` possibile forzare il valore intero da associare ai valori di una enumerazione:

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

Non e` necessario specificare un valore per ogni identificatore dell'enumerazione, non ci sono limitazioni di segno e non e` necessario usare valori distinti (anche se cio` probabilmente comporterebbe qualche problema). Si puo` utilizzare anche un identificatore dell'enumerazione precedentemente definito.

La possibilita` di scegliere i valori da associare alle etichette (identificatori) dell'enumerazione fornisce un modo alternativo di definire costanti di tipo intero.

81

La keyword typedef

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

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

Il listato seguente mostra alcune possibili applicazioni:

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


La keyword typedef

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

E` possibile dichiarare tipi anonimi solo per i costrutti struct, union e enum e sono utilizzabili quasi esclusivamente nelle dichiarazioni (come nel caso di typedef oppure nelle dichiarazioni di variabili e costanti).

La keyword typedef e` utile per creare abbreviazioni per espressioni di tipo complesse, soprattutto quando l'espressione di tipo coinvolge puntatori e funzioni.

Funzioni

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

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

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

Funzioni

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

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

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

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

```
// funzione che non riceve parametri
// e restituisce un int (default)
Funny();            // ORRORE!!!
// oppure
Funny2(void);     // ORRORE!!!
```

Funzioni

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

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

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

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

86

Funzioni

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

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

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

87

Funzioni

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

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

La funzione Sum e` costituita da una sola istruzione che calcola la somma degli argomenti e restituisce tramite la keyword return il risultato di tale operazione. Inoltre, benche` non evidente dall'esempio, la keyword return provoca l'immediata terminazione della funzione;

88

Funzioni

ecco un esempio non del tutto corretto, che pero` mostra il comportamento di return:

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

Se il divisore e` 0, la prima istruzione return restituisce (erroneamente) una stringa (anziche` un intero) e provoca la terminazione della funzione, le successive istruzioni della funzione quindi non verrebbero eseguite.
Concludiamo questo paragrafo con alcune considerazioni:

89

Funzioni

La definizione di una funzione non deve essere seguita da ; (punto e virgola), cio` tra l'altro consente di distinguere facilmente tra prototipo (dichiarazione) e definizione di funzione poiche` un prototipo e` terminato da ; (punto e virgola), mentre in una definizione la lista di argomenti e` seguita da { (parentesi graffa aperta). Ogni funzione dichiarata non void deve restituire un valore, ne segue che da qualche parte nel corpo della funzione deve esserci una istruzione return con un qualche argomento (il valore restituito), in caso contrario viene segnalato un errore; analogamente l'uso di return in una funzione void costituisce un errore, salvo il caso in cui la keyword sia utilizzata senza argomenti (provocando cosi` solo la terminazione della funzione);

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

Non e` possibile definire una funzione all'interno del corpo di un'altra funzione.

Funzioni

Ecco ancora qualche esempio relativo alla seconda nota:

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

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

Funzioni

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

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

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

92

Funzioni

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

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

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

93

Funzioni

```
  /* ... */
  Sleep(X);
  Result = Sum(X, Y);
  Sum(X, 8);            // Ok!
  Result = Sleep(1000);  // Errore!
  return 0;
}
```

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

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

94

passaggio di parametri

I parametri di una funzione si comportano all'interno del corpo della funzione come delle variabili locali e possono quindi essere usati anche a sinistra di un assegnamento:

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

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

95

passaggio di parametri

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

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

 int main(int, char* []) {
  int X = 5;
  int Y = 10;
```

96

passaggio di parametri

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

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

  cout << "X = " << X << endl;
  cout << "Y = " << Y << endl;
  return 0;
 }
```

97

passaggio di parametri

L'esempio appena visto e` perfettamente funzionante e se eseguito mostrerebbe come la funzione Assign, pur eseguendo una modifica ai suoi parametri formali, non modifichi i parametri attuali. Questo comportamento e` perfettamente corretto in quanto i parametri attuali vengono passati per valore: ad ogni chiamata della funzione viene cioe` creata una copia di ogni parametro localmente alla funzione stessa; tali copie vengono distrutte quando la chiamata della funzione termina ed il loro contenuto non viene copiato nelle eventuali variabili usate come parametri attuali.

In alcuni casi tuttavia puo` essere necessario fare in modo che la funzione possa modificare i suoi parametri attuali, in questo caso e` necessario passare non una copia, ma un riferimento o un puntatore e agire su questo per modificare una variabile non locale alla funzione. Per adesso non considereremo queste due possibilita`, ma lo faremo appena avremo parlato di puntatori e reference.

parametri e argomenti di default

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

Ecco come fare:

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

parametri e argomenti di default

Quella che abbiamo appena visto e` la definizione della funzione Sum ai cui argomenti sono stati associati dei valori di default (in questo caso 0 per entrambi gli argomenti), ora se la funzione Sum viene chiamata senza specificare il valore di a e/o b il compilatore genera una chiamata a Sum sostituendo il valore di default (0) al parametro non specificato. Una funzione puo` avere piu` argomenti di default, ma le regole del C++ impongono che tali argomenti siano specificati alla fine della lista dei parametri formali nella dichiarazione della funzione:

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

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

100

parametri e argomenti di default

```
 }                    // Errore!

 void Foo3(int a, float f, int c = 4) {
   /* ... */
 }                    // Ok!
```

La dichiarazione di Foo2 e` errata poiche` quando viene specificato un argomento con valore di default, tutti gli argomenti seguenti (in questo caso f) devono possedere un valore di default; l'ultima definizione mostra come si sarebbe dovuto definire Foo2 per non ottenere errori.

101

parametri e argomenti di default

La risoluzione di una chiamata di una funzione con argomenti di default naturalmente differisce da quella di una funzione senza argomenti di default in quanto sono necessari un numero di controlli maggiori (tutti i controlli comunque sono eseguiti al momento della compilazione, non temete: non c'e' alcun overhead run-time!). Sostanzialmente se nella chiamata per ogni parametro formale viene specificato un parametro attuale, allora il valore di ogni parametro attuale viene copiato nel corrispondente parametro formale sovrascrivendo eventuali valori di default; se invece qualche parametro non viene specificato, quelli forniti specificano il valore dei parametri formali secondo la loro posizione e per i rimanenti parametri formali viene utilizzato il valore di default specificato (se nessun valore di default e` stato specificato, viene generato un errore):

```
// riferendo alle precedenti definizioni:
  Foo(1, 'b');     // chiama Foo con argomenti 1 e 'b'
  Foo(0);          // chiama Foo con argomenti 0 e 'a'
```

102

parametri e argomenti di default

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

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

103

parametri del main

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

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

104

parametri del main

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

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

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

105

La funzione main()

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

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

```
< Dichiarazioni globali e funzioni >

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

106

La funzione main()

Un programma e` dunque costituito da un insieme (eventualmente vuoto) di dichiarazioni e di definizioni globali di costanti, variabili... ed un insieme di dichiarazioni e definizioni di funzioni (che non possono essere dichiarate e/o definite localmente ad altre funzioni); infine il corpo del programma e` costituito dalla funzione main, il cui prototipo per esteso è quello appena mostrato.

Nello schema main ritorna un valore di tipo int (che generalmente e` utilizzato per comunicare al sistema operativo la causa della terminazione). I vecchi compilatori non standard spesso lasciavano ampia liberta` circa il prototipo di main, alcuni consentivano di dichiararla void, ora a norma di standard main deve avere tipo int e se nel corpo della funzione non viene inserito esplicitamente una istruzione return, il compilatore inserisce automaticamente una “return 0;”.

Inoltre main puo` accettare opzionalmente due parametri: il primo e` di tipo int e indica il numero di parametri presenti sulla riga di comando attraverso cui e` stato eseguito il programma; il secondo parametro (si comprendera` in seguito) e` un array di stringhe terminate da zero (puntatori a caratteri) contenente i parametri, il primo dei quali (argv[0]) e` il nome del programma come riportato sulla riga di comando.

107

Funzioni inline

Le funzioni consentono di scomporre in piu` parti un grosso programma facilitandone sia la realizzazione che la successiva manutenzione. Tuttavia spesso si e` indotti a rinunciare a tale beneficio perche` l'overhead imposto dalla chiamata di una funzione e` tale da sconsigliare la realizzazione di piccole funzioni. Le possibili soluzioni in C erano due:

Rinunciare alle funzioni piccole, tendendo a scrivere solo poche funzioni corpose;

Ricorrere alle macro;

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

108

Funzioni inline

Il problema principale e` che questo sistema rende difficoltoso ogni controllo statico di tipo poiche` gli errori divengono evidenti solo quando la macro viene espansa; in C tutto sommato cio` non costituisce un grave problema perche` il C non e` fortemente tipizzato.

Al contrario il C++ possiede un rigido sistema di tipi e l'uso di macro costituisce un grave ostacolo allo sfruttamento di tale caratteristica. Esistono poi altri svantaggi nell'uso delle macro: rendono difficile il debugging e non sono flessibili come le funzioni (e` ad esempio difficile rendere fattibili macro ricorsive). Per non rinunciare ai vantaggi forniti dalle (piccole) funzioni e a quelli forniti da un controllo statico dei tipi, sono state introdotte nel C++ le funzioni inline.

Quando una funzione viene definita inline il compilatore ne memorizza il corpo e, quando incontra una chiamata a tale funzione, semplicemente lo sostituisce alla chiamata della funzione; tutto cio` consente di evitare l'overhead della chiamata e, dato che la cosa e` gestita dal compilatore, permette di eseguire tutti i controlli statici di tipo.

109

Funzioni inline

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

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

La keyword inline informa il compilatore che si desidera che la funzione Sum sia espansa inline ad ogni chiamata; tuttavia cio` non vuol dire che la cosa sia sempre possibile: molti compilatori non sono in grado di espandere inline qualsiasi funzione, tipicamente sono molto difficili da trattare funzioni ricorsive (viene fatto l'inlining di un numero finito, piccolo, di iterazioni).

Quando l'espansione inline della funzione non e` possibile solitamente si viene avvisati da un warning.

110

Funzioni inline

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

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

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

111

Funzioni inline

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

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

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


Funzioni inline

Un'altra cosa da tener presente e` che il codice che costituisce una funzione inline deve essere disponibile prima di ogni uso della funzione, altrimenti il compilatore non e` in grado di espanderla (non sempre almeno!). Una funzione ordinaria puo` essere usata anche prima della sua definizione, poiche` e` il linker che si occupa di risolvere i riferimenti (il linker del C++ lavora in due passate); nel caso delle funzioni inline, poiche` il lavoro e` svolto dal compilatore (che lavora in una passata), non e` possibile risolvere correttamente il riferimento. Una importante conseguenza di tale limitazione e` che una funzione puo` essere inline solo nell'ambito del file in cui e` definita, se un file riferisce ad una funzione definita inline in un altro file (come, lo vedremo piu` avanti), in questo file (il primo) la funzione non potra` essere espansa; esistono comunque delle soluzioni al problema.

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

Overloading delle funzioni

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

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

Ritorniamo per un attimo alla nostra funzione Sum...

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

Overloading delle funzioni

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

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

Overloading delle funzioni

Nel nostro esempio ci siamo limitati solo a dichiarare piu` volte la funzione Sum, ogni volta con un significato diverso (uno per ogni possibile caso di somma in cui possono essere coinvolti, anche contemporaneamente, interi e reali); e` chiaro che poi da qualche parte deve esserci una definizione per ciascun prototipo (nel nostro caso tutte le definizioni sono identiche a quella gia` vista, cambia solo l'intestazione della funzione).

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

Overloading delle funzioni

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

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

Comunque si tratta di una pratica obsoleta che infatti non e` prevista nello standard.

Overloading delle funzioni

Le funzioni sovraccaricate si utilizzano esattamente come le normali funzioni:

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

 /* Dichiarazione ed implementazione delle varie Sum */

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

118

Overloading delle funzioni

```
   float r = 0.5;

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

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

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

119

Overloading delle funzioni

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

   return 0;
 }
```

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

120

Overloading delle funzioni

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

La seconda dichiarazione e` errata perche`, per scegliere tra la prima e la seconda versione della funzione, il compilatore si basa unicamente sui tipi dei parametri che nel nostro caso coincidono; la soluzione e` mostrata con la terza dichiarazione, ora il compilatore e` in grado di distinguere perche` il primo parametro anzicche` essere un int e` un float.

121

Overloading delle funzioni

Infine le ultime due dichiarazioni non sono in conflitto per via delle regole che il compilatore segue per scegliere quale funzione applicare; in linea di massima e secondo la loro priorita`:

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

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

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

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

Overloading delle funzioni

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

Se nessuna di queste regole puo` essere applicata, si genera un errore (funzione non definita!). La piena comprensione di queste regole richiederebbe la conoscenza del concetto di conversione di tipo.

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

volatile

La keyword volatile serve ad informare il compilatore che una certa variabile cambia valore in modo aleatorio e che di conseguenza il suo valore va riletto ogni volta che esso sia richiesto:

```
volatile int ComPort;
```

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

Puntatori

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

Vediamo alcune esempi di dichiarazione di puntatori:

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

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

Puntatori

Si osservi che un puntatore a un certo tipo puo` puntare solo a oggetti di quel tipo, (non e` possibile ad esempio assegnare l'indirizzo di una variabile di tipo float a un puntatore a char, come mostra il codice seguente), o meglio in molti casi e` possibile farlo, ma viene eseguita un cast:

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

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

```
Puntatore = 0;
```

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

126

Puntatori

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

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

127

Puntatori

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

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

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

128

Puntatori

Ritorniamo all'esempio da cui siamo partiti.

La terza riga mostra come dichiarare un puntatore a un intero costante, attenzione non un puntatore costante; la dichiarazione di un puntatore costante e` mostrata nella penultima riga. Un puntatore a una costante consente l'accesso all'oggetto da esso puntato solo in lettura (ma cio` non implica che l'oggetto puntato sia effettivamente costante), mentre un puntatore costante e` una costante di tipo puntatore (a ...), non e` quindi possibile modificare l'indirizzo in esso contenuto e va inizializzato nella dichiarazione. L'ultima riga mostra invece come combinare puntatori costanti e puntatori a costanti per ottenere costanti di tipo puntatore a costante (intera, nell'esempio).

Attenzione: anche const, se utilizzato per dichiarare una costante puntatore, si applica ad un solo nome (come *) e valgono quindi le stesse raccomandazioni fatte sopra.

129

Puntatori

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

```
void* PuntatoreGenerico;
```

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

130

Operazioni sui puntatori

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

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

131

Operazioni sui puntatori

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

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

132

Operazioni sui puntatori

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

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

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

133

Operazioni sui puntatori

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

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


Operazioni sui puntatori

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

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

Operazioni sui puntatori

Val e` stata modificata tramite P:
*P = -10
Val = -10

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

136

Operazioni sui puntatori

L'operazione di dereferenzazione puo` essere eseguita su un qualsiasi puntatore a condizione che questo non sia stato dichiarato void. In generale infatti non e` possibile stabilite il tipo dell'oggetto puntato da un puntatore void e il compilatore non sarebbe in grado di trattare tale oggetto.

Quando si dereferenzia un puntatore bisogna prestare attenzione che esso sia stato inizializzato correttamente; la dereferenzazione di un puntatore inizializzato a 0 e` sempre un errore, la dereferenzazione di un puntatore non inizializzato causa errori non definiti (e potenzialmente difficili da scovare). Quando possibile comunque il compilatore segnala eventuali tentativi di dereferenziare puntatori che potrebbero non essere stati inizializzati tramite un warning.

Per i puntatori a strutture (o unioni) e` possibile utilizzare un altro operatore di dereferenzazione che consente in un colpo solo di dereferenziare il puntatore e selezionare il campo desiderato:

137

Operazioni sui puntatori

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

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

La terza riga dell'esempio dereferenzia Puntatore e contemporaneamente seleziona il campo Eta (il tutto tramite l'operatore ->) per eseguire un assegnamento a quest'ultimo. Nell'ultima riga viene mostrato come utilizzare -> per ottenere il valore di un campo dell'oggetto puntato.

Operazioni sui puntatori

Sui puntatori e` definita una speciale aritmetica composta da somma e sottrazione. Se P e` un puntatore di tipo T, sommare 1 a P significa puntare all'elemento successivo di un ipotetico array di tipo T cui P e` immaginato puntare; analogamente sottrarre 1 significa puntare all'elemento precedente. E` possibile anche sottrarre da un puntatore un altro puntatore (dello stesso tipo), in questo caso il risultato e` il numero di elementi che separano i due puntatori:

```
int Array[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int* P1 = &Array[5];
int* P2 = &Array[9];
cout << P1 - P2 << endl; // visualizza 4
cout << *P1 << endl;     // visualizza 5
P1+=3;                   // equivale a P1 = P1 + 3;
cout << *P1 << endl;     // visualizza 8
cout << *P2 << endl;     // visualizza 9
P2-=5;                   // equivale a P2 = P2 - 5;
cout << *P2 << endl;     // visualizza 4
```

Operazioni sui puntatori

Sui puntatori sono anche definiti gli usuali operatori relazionali:

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

140

Puntatori vs array

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

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

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

141

Puntatori vs array

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

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

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

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

142

Puntatori vs array

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

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

143

Puntatori vs array

In C++ le dichiarazioni char Array[ ] = "Una stringa" e char* Ptr = "Una stringa" hanno lo stesso effetto, entrambe creano una stringa (terminata dal carattere nullo) il cui indirizzo e` posto rispettivamente in Array e in Ptr, e come mostra l'esempio un char* puo` essere utilizzato esattamente come un array di caratteri.

Esistono tuttavia profonde differenze tra puntatori e array: un puntatore e` una variabile a cui si possono applicare le operazioni viste sopra e che puo` essere usato come un array, ma non e` vero il viceversa, in particolare il nome di un array non e` un puntatore a cui e` possibile assegnare un nuovo valore (non e` cioe` modificabile: e' un puntatore const). Ecco un esempio:

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

144

Puntatori vs array

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

In definitiva un puntatore e` piu` flessibile di quanto non lo sia un array, anche se a costo di un maggiore overhead.

145

Uso dei puntatori

I puntatori sono utilizzati sostanzialmente per quattro scopi:
Realizzare strutture dati dinamiche (es. linked lists);
Realizzare funzioni con effetti collaterali sui parametri;
Ottimizzare il passaggio di parametri di grosse dimensioni;
Rendere possibile il passaggio di parametri di tipo funzione.

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

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

146

Uso dei puntatori

```
// Una lista e` composta da tanti elementi collegati
// tra di loro; ogni elemrnto  contiene un valore
// e un puntatore al successivo.

struct TCell {
  float AFloat;  // per memorizzare un valore
  TCell* Next;  // puntatore all'elemento successivo
};
```

147

Uso dei puntatori

```
// La lista viene realizzata tramite questa
  // struttura contenente il numero corrente di celle
  // della lista e il puntatore alla prima cella

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

  int main(int, char* []) {
```


Uso dei puntatori

```
    TList List;        // Dichiara una lista
    List.Size = 0;     // inizialmente vuota
    int FloatToRead;
    cout << "Quanti valori vuoi immettere? " ;
    cin >> FloatToRead;
    cout << endl;

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

Uso dei puntatori

```
for(int i=0; i < FloatToRead; ++i) {
    TCell* Temp = List.First;
    cout << "Creazione di una nuova cella..." << endl;
    List.First = new TCell;    // new vuole il tipo di
                               // variabile da creare
    cout << "Immettere un valore reale " ;

    // cin legge l'input da tastiera e l'operatore di
    // estrazione >> lo memorizza nella variabile.
    cin >> List.First -> AFloat;
    cout << endl;
    List.First -> Next = Temp; // aggiunge la cella in
                               // testa alla lista
    ++List.Size;       // incrementa la
                       // dimensione della lista
  }
```

150

Uso dei puntatori

```
  // il seguente ciclo calcola la somma
  // dei valori contenuti nella lista;
  // via via che recupera i valori,
  // distrugge le relative celle
  float Total = 0.0;
  for(int j=0; j < List.Size; ++j) {
    Total += List.First -> AFloat;

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

151

Uso dei puntatori

```
    List.First = List.First -> Next;

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

152

Uso dei puntatori

Il programma sopra riportato programma memorizza in una lista un certo numero di valori reali, aggiungendo per ogni valore una nuova cella; in seguito li estrae uno ad uno e li somma restituendo il totale; via via che un valore viene estratto dalla lista, la cella corrispondente viene distrutta. Il codice e` ampiamente commentato e non dovrebbe essere difficile capire come funziona. La creazione di un nuovo oggetto avviene allocando un nuovo blocco di memoria (sufficientemente grande) dalla heap-memory (una porzione di memoria riservata all'avvio di un programma per operazioni di questo tipo), mentre la distruzione avviene deallocando tale blocco (che ritorna a far parte della heap-memory); l'allocazione viene eseguita tramite l'operatore new cui va specificato il tipo di oggetto da creare (per sapere quanta ram allocare), la deallocazione avviene invece tramite l'operatore delete, che richiede come argomento un puntatore all'aggetto da deallocare (la quantita` di ram da deallocare viene calcolata automaticamente).

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

153

Uso dei puntatori

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

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

La dimensione massima di strutture dinamiche e` unicamente determinata dalla dimensione della heap memory che a sua volta e` generalmente limitata dalla quantita` di memoria del sistema.

Un altro importante aspetto degli oggetti allocati dinamicamente e` che essi non ubbidiscono alle normali regole di scoping statico, solo i puntatori in quanto tali sono soggetti a tali regole, un oggetto allocato dinamicamente puo` quindi essere creato in un certo scope ed essere acceduto in un altro semplicemente trasmettendone l'indirizzo (il valore del puntatore).

154

Uso dei puntatori

Consideriamo ora il secondo uso che si fa dei puntatori.

Si tratta del "passaggio di parametri per indirizzo" e consente la realizzazione di funzioni con effetti collaterali sui parametri attuali:

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

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

155

Uso dei puntatori

```
int A = 10;

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

l'output che il precedente codice produce e`:

```
A = 10
Chiamata a Change(int*)...
Ora A = 5
```

156

Uso dei puntatori

Quello che nell'esempio accade e` che la funzione Change riceve l'indirizzo della variabile A e tramite esso e` in grado di agire sulla variabile stessa.

L'uso dei puntatori come parametri di funzione non e` comunque utilizzato solo per consentire effetti collaterali, spesso una funzione riceve parametri di dimensioni notevoli e l'operazione di copia del parametro in un'area privata della funzione ha effetti deleteri sui tempi di esecuzione della funzione stessa; in questi casi e` molto piu` conveniente passare un puntatore che occupa pochi byte:

157

Uso dei puntatori

```
void Func(BigParam parametro);
// funziona, ma e` meglio quest'altra dichiarazione

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

Il secondo prototipo e` piu` efficiente perche` evita l'overhead imposto dal passaggio per valore, inoltre l'uso di const previene ogni tentativo di modificare l'oggetto puntato e allo stesso tempo comunica al programmatore che usa la funzione che non esiste tale rischio.

Infine quando l'argomento di una funzione e` un array, il compilatore passa sempre un puntatore, mai una copia dell'argomento; in questo caso inoltre l'unico modo che la funzione ha per conoscere la dimensione dell'array e` quello di ricorrere ad un parametro aggiuntivo, esattamente come accade con la funzione main().

158

Uso dei puntatori

Ovviamente una funzione puo` restituire un tipo puntatore, in questo caso bisogna pero` prestare attenzione a cio` che si restituisce, non e` raro infatti che un principiante scriva qualcosa del tipo:

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

159

Uso dei puntatori

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

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

Uso dei puntatori

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

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

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

Uso dei puntatori

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

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


Uso dei puntatori

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

int main(int, char* []) {
    for(int i=0; i<10; ++i) {
      cout << "Immetti un intero: ";
      int A;
      cin >> A;
      cout << endl << "Immetti un altro intero: ";
```

Uso dei puntatori

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


Uso dei puntatori

La typedef dice che Eval e` un tipo "funzione che prende due interi e restituisce un bool", quindi conformemente al tipo Eval definiamo due funzioni Max e Min dall'evidente significato. Si definisce quindi una funzione Check che riceve quattro parametri: un puntatore a Eval, una stringa e due interi. La funzione Check usa Func per eseguire la chiamata alla funzione puntata e ricavarne il valore restituito. Si noti che la chiamata alla funzione puntata viene eseguita come se Func fosse esso stesso la funzione (ovvero utilizzando l'operatore () e passando normalmente i parametri).

Si noti infine che la funzione main ricava l'indirizzo di Max e Min senza ricorrere all'operatore &, analogamente a quanto si fa con gli array.

Reference

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

Cosi` come un puntatore viene indicato nelle dichiarazioni dal simbolo *, un reference viene indicato dal simbolo &:

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

166

Reference

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

Le ultime due righe dichiarano rispettivamente un riferimento a int e uno a float che vengono subito inizializzati usando le due variabili dichiarate prima. Un riferimento va inizializzato immediatamente, e dopo l'inizializzazione non puo` essere piu` cambiato; si noti che non e` necessario utilizzare l'operatore & (indirizzo di) per eseguire l'inizializzazione.

167

Reference

Dopo l'inizializzazione il riferimento potra` essere utilizzato in luogo della variabile cui e` legato, utilizzare l'uno o l'altro sara` indifferente:

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


Reference

Ecco l'output del precedente codice:

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

Dall'esempio si capisce perche`, dopo l'inizializzazione, un riferimento non possa essere piu` associato ad un nuovo oggetto: ogni assegnamento al riferimento si traduce in un assegnamento all'oggetto riferito.

Reference

Un riferimento puo` essere inizializzato anche tramite un puntatore:

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

int& IntRef = *IntPtr;
```

Si noti che il puntatore va dereferenziato, altrimenti si legherebbe il riferimento al puntatore (in questo caso l'uso del riferimento comporta implicitamente una conversione da int* a int).

Reference

Ovviamente il metodo puo` essere utilizzato anche con l'operatore new:

```
double& DoubleRef = *new double;

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

DoubleRef = 7.3;
// Di nuovo, e` compito del programmatore
```

Reference

```
// distruggere l'oggetto crato con new

delete &DoubleRef;

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

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

Uso dei reference

I riferimenti sono stati introdotti nel C++ come ulteriore meccanismo di passaggio di parametri (per riferimento).
Una funzione che debba modificare i parametri attuali puo` ora essere dichiarata in due modi diversi:

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

oppure in modo del tutto equivalente

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

Uso dei reference

Naturalmente cambierebbe il modo in cui chiamare la funzione:

```
long double Var = 0.0;
long double* Ptr = &Var;
// nel primo caso avremmo
Esempio(&Var);
// oppure
Esempio(Ptr);
// nel caso di passaggio per riferimento
Esempio(Var);
// oppure
Esempio(*Ptr);
```


Uso dei reference

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

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

Uso dei reference

Puntatori e reference possono essere liberamente scambiati, non esiste differenza eccetto che non e` necessario dereferenziare un riferimento e che i riferimenti non possono associati ad un'altra variabile dopo l'inizializzazione.

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

```
&A + &B
```

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

176

Puntatori vs reference

Visto che per le funzioni e` possibile scegliere tra puntatori e riferimenti, come decidere quale metodo scegliere? I riferimenti hanno un vantaggio sui puntatori, dato che nella chiamata di una funzione non c'e` differenza tra passaggio per valore o per riferimento, e` possibile cambiare meccanismo senza dover modificare ne` il codice che chiama la funzione ne` il corpo della funzione stessa. Tuttavia il meccanismo dei reference nasconde all'utente il fatto che si passa un indirizzo e non una copia, e cio` puo` creare grossi problemi in fase di debugging.

Quando e` necessario passare un indirizzo e` quindi meglio usare i puntatori, rendono esplicito il modo in cui il parametro viene passato. Esiste comunque una eccezione nel caso dei tipi definiti dall'utente tramite il meccanismo delle classi. In questo caso vedremo che l'incapsulamento garantisce che l'oggetto passato possa essere modificato solo da particolari funzioni (funzioni membro e funzioni amiche), e quindi usare i riferimenti e` piu`conveniente perche` non e` necessario dereferenziarli, migliorando cosi` la chiarezza del codice; le funzioni membro e le funzioni amiche, in quanto tali, sono invece autorizzate a modificare l'oggetto e quindi quando vengono usate l'utente sa gia` che potrebbero esserci effetti collaterali.

177

Puntatori vs reference

Non si tratta comunque di una regola generale, come per tante altre cose, i progettisti del linguaggio hanno pensato di non limitare l'uso dei costrutti con rigide regole e schemi predefiniti, ma di lasciare al buon senso del programmatore il compito di decidere quale fosse di volta in volta la soluzione migliore.

178

Linkage

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

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

// File b.cpp
```

179

Linkage

```
extern int a;

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

Il primo file dichiara una variabile intera e la inizializza, il secondo (trascuriamone per ora la prima riga di codice) dichiara una funzione che ne restituisce il valore. La compilazione del primo file non e` un problema, ma nel secondo file GetVar() deve utilizzare un nome dichiarato in un altro file; perche` la cosa sia possibile bisogna informare il compilatore che tale nome e` dichiarato da qualche altra parte e che il riferimento a tale nome non puo` essere risolto se non quando tutti i file sono stati compilati, solo il linker quindi puo` risolvere il problema collegando insieme i due file. Il compilatore deve dunque essere informato dell'esistenza della variabile al fine di non generare un messaggio di errore; tale operazione viene effettuata tramite la keyword extern.

Linkage

In effetti la riga extern int a; non definisce una nuova variabile, ma dice "La variabile intera a e` definita da qualche altra parte, lascia solo lo spazio per risolvere il riferimento". Se la keyword extern fosse stata omessa il compilatore avrebbe interpretato la riga come una nuova dichiarazione e avrebbe risolto il riferimento in GetVar() in favore di tale definizione; in fase di linking comunque si sarebbe verificato un errore perche` a sarebbe stata definita due volte (una per file), il perche` di tale errore sara` chiaro piu` avanti.

Naturalmente extern si puo` usare anche con le funzioni (anche se come vedremo e` ridondante):

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

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

Linkage

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

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

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

Linkage

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

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

// oppure per risparmiare tempo
```

Linkage

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

La presenza di "C" serve a indicare che bisogna adottare le convenzioni del C sulla codifica dei nomi (in quanto il compilatore C++ codifica internamente i nomi degli identificatori in modo assai diverso).

Linkage

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

```
extern int Func2(int);

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

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

Linkage

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

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

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


Linkage

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

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

Linkage

I nomi che sono visibili all'esterno di un file sono detti avere linkage esterno; tutte le variabili globali hanno linkage esterno, cosi` come le funzioni globali non inline; le funzioni inline, tutte le costanti e le dichiarazioni fatte in un blocco hanno invece linkage interno (cioe` non sono visibili all'esterno del file); i nomi di tipo non hanno alcun linkage.

188

Linkage

E` possibile forzare un identificatore globale ad avere linkage interno utilizzando la keyword static:

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

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


 // File b.cpp
```

189

Linkage

```
extern int f(int);

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

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

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

190

File header

Purtroppo non esiste un meccanismo analogo alla keyword static per forzare un linkage esterno, d'altronde i nomi di tipo non hanno linkage (e devono essere consistenti) e le funzioni inline non possono avere linkage esterno per ragioni pratiche (la compilazione e` legata al singolo file sorgente). Esiste tuttavia un modo per aggirare l'ostacolo: racchiudere tali dichiarazioni e/o definizioni in un file header (file solitamente con estensione .h) e poi includere questo nei files che utilizzano tali dichiarazioni; possiamo anche inserire dichiarazioni e/o definizioni comuni in modo da non doverle ripetere.

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

191

File header

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

const int Max = 1000;

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


File header

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

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

/* codice modulo */
```

File header

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

In alcuni esempi abbiamo gia` utilizzato la direttiva per poter eseguire input/output, in quei casi abbiamo utilizzato le parentesi angolari (< >) al posto dei doppi apici (" "); la differenza e` che utilizzando i doppi apici dobbiamo specificare (se necessario) il path in cui si trova il file header, con le parentesi angolari invece il preprocessore cerca il file in un insieme di directory predefinite.

Si noti inoltre che questa volta e` stato specificato l'estensione del file (.h), questo non dipende dall'uso degli apici, ma dal fatto che ad essere incluso e` l'header di un file di libreria (ad esempio quando si usa la libreria iostream), infatti in teoria tali header potrebbero non essere memorizzati in un normale file.

File header

Un file header puo` contenere in generale qualsiasi istruzione C/C++ (in particolare anche dichiarazioni extern) da condividere tra piu` moduli:

```
// Esempio2.h

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

// o dichiarazioni extern comuni ai moduli
extern "C" {              // Inclusione di un
```

File header

```
  #include "HeaderC.h"     // file header C
}
extern "C" {
  int CFunc1(int, float);
  void CFunc2(char*);
}
extern int a;
extern double* Ptr;
extern void Func();
```

196

Librerie di funzioni

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

```
// Date.h
struct Date {
  unsigned short dd;    // giorno
  unsigned short mm;    // mese
  unsigned yy;          // anno
```

197

Librerie di funzioni

```
  unsigned short h;     // ora
  unsigned short m;     // minuti
  unsigned short s;     // secondi
};

void PrintDate(Date);
```


Librerie di funzioni

ed ecco come sarebbe il file che la implementa:

```
// Date.cpp
#include "Date.h"
#include < iostream >
using namespace std;

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

A questo punto la libreria e` pronta, per distribuirla basta compilare il file Date.cpp e fornire il file oggetto ottenuto ed il file header Date.h. Chi deve utilizzare la libreria non dovra` far altro che includere nel proprio programma il file header e linkarlo al file oggetto contenente le funzioni di libreria.

Librerie di funzioni

Esistono tuttavia due problemi, il primo e` illustrato nel seguente esempio:

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

/* altre dichiarazioni */

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


Librerie di funzioni

```
/* altre dichiarazioni */

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

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

int main(int, char* []) {
  /* codice funzione */
}
```

Librerie di funzioni

Si tratta cioe` di un programma costituito da piu` moduli, quello principale che contiene la funzione main() e altri che implementano le varie routine necessarie. Piu` moduli hanno bisogno di una stessa libreria, in particolare hanno bisogno di includere lo stesso file header (nell'esempio iostream) nei rispettivi file header.

Per come funziona il preprocessore, poiche` il file principale include (direttamente e/o indirettamente) piu` volte lo stesso file header, il file che verra` effettivamente compilato conterra` piu` volte le stesse dichiarazioni (e definizioni) che daranno luogo a errori di definizione ripetuta dello stesso oggetto (funzione, costante, tipo...). Come ovviare al problema?

Librerie di funzioni

La soluzione ci e` fornita dal preprocessore stesso ed e` nota come compilazione condizionale; consiste cioe` nello specificare quando includere o meno determinate porzioni di codice. Per far cio` ci si avvale delle direttive #define SIMBOLO, #ifndef SIMBOLO e #endif: la prima ci permette di definire un simbolo, la seconda e` come l'istruzione condizionale e serve a testare un simbolo (la risposta e` positiva se SIMBOLO non e` definito, negativa altrimenti), l'ultima direttiva serve a capire dove finisce l'effetto della direttiva condizionale. Le ultime due direttive sono utilizzate per delimitare porzioni di codice; se #ifndef e verificata il preprocessore lascia passare il codice (ed esegue eventuali direttive) tra l'#ifndef e #endif, altrimenti quella porzione di codice viene nascosta al compilatore.

Ecco come tali direttive sono utilizzate (l'errore era dovuto all'inclusione multipla di iostream):

Librerie di funzioni

```
// Contenuto del file iostream.h
 #ifndef IOSTREAM_H
 #define IOSTREAM_H
 /* contenuto file header */
 #endif
```

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

204

Librerie di funzioni

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

Il preprocessore inizia a elaborare il file per produrre un unico file compilabile;

Viene incontrata la direttiva #include < iostream > e il file header specificato viene elaborato per produrre codice;

A seguito delle direttive contenute inizialmente in iostream, viene definito il simbolo IOSTREAM_H e prodotto il codice contenuto tra #ifndef IOSTREAM_H e #endif;

Si ritorna al file Main.cpp e il preprocessore incontra #include < Modulo1.h > e quindi va ad elaborare Modulo1.h;

La direttiva #include < iostream > contenuta in Modulo1.h porta il precompilatore ad elaborare di nuovo iostream, ma questa volta il simbolo IOSTREAM_H e` definito e quindi #ifndef IOSTREAM_H fa si` che nessun codice venga prodotto;

205

Librerie di funzioni

Si prosegue l'elaborazione di Modulo1.h e viene generato l'eventuale codice;

Finita l'elaborazione di Modulo1.h, la direttiva #include < Modulo2.h > porta all'elaborazione di Modulo2.h che e` analoga a quella di Modulo1.h;

Elaborato anche Modulo2.h, rimane la funzione main() di Main.cpp che produce il corrispondente codice;

Alla fine il preprocessore ha prodotto un unico file contenente tutto il codice di Modulo1.h, Modulo2.h e Main.cpp senza alcuna duplicazione e contenente tutte le dichiarazioni e le definizioni necessarie;

Il file prodotto dal preprocessore e` passato al compilatore per la produzione di codice oggetto;

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

206

I namespace

Il secondo problema che si verifica con la ripartizione di un progetto in piu` file e` legato alla necessita` di utilizzare identificatori globali unici. Quello che spesso accade e` che al progetto lavorino piu` persone ognuna delle quali si occupa di parti diverse che devono poi essere assemblate. Per quanto possa sembrare difficile, spesso accade che persone che lavorano a file diversi utilizzino gli stessi identificatori per indicare funzioni, variabili, costanti...

Pensate a due persone che devono realizzare due moduli ciascuno dei quali prima di essere utilizzato vada inizializzato, sicuramente entrambi inseriranno nei rispettivi moduli una funzione per l'inizializzazione e molto probabilmente la chiameranno InitModule() (o qualcosa di simile). Nel momento in cui i due moduli saranno linkati insieme (e sempre che non siano sorti problemi prima ancora), inevitabilmente il linker segnalera` errore.

207

I namespace

Naturalmente basterebbe che una delle due funzioni avesse un nome diverso, ma modificare tale nome richiederebbe la modifica anche dei sorgenti in cui il modulo e` utilizzato. Molto meglio prevenire tale situazione suddividendo lo spazio globale dei nomi in parti piu` piccole (i namespace) e rendere unicamente distinguibili tali parti, a questo punto poco importa se in due namespace distinti un identificatore appare due volte... Ma vediamo un esempio:

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


I namespace

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

In una situazione di questo tipo non ci sarebbe piu` conflitto tra le definizioni dei due file, perche` per accedere ad esse e` necessario specificare anche l'identificatore del namespace:

I namespace

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

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

  Areal *= Pi;      // Errore!

  SamLib::Close();
}
```

210

I namespace

L'operatore :: e` detto risolutore di scope e indica al compilatore dove cercare l'identificatore seguente. In particolare l'istruzione MikeLib::Init(); dice al compilatore che la Init() cui vogliamo riferirci e` quella del namespace MikeLib. Ovviamente perche` non ci siano conflitti e` necessario che i due namespace abbiano nomi diversi, ma e` piu` facile stabilire pochi nomi diversi tra loro, che molti.

Si noti che il tentativo di riferire ad un nome senza specificarne il namespace viene interpretato come un riferimento ad un nome globale esterno ad ogni namespace e nell'esempio precedente genera un errore perche` nello spazio globale non c'e` alcun Pi.

211

I namespace

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

Un namespace puo` essere creato solo nello scope globale;

Se nello scope globale di un file esistono due namespace con lo stesso nome (ad esempio i due namespace sono definiti in file header diversi, ma inclusi da uno stesso file), essi vengono fusi in uno solo;

E` possibile creare un alias di un namespace con la sintassi: namespace < ID1 > = < ID2 >;

E` possibile avere namespace anonimi, in questo caso gli identificatori contenuti nel namespace sono visibili al file che contiene il namespace anonimo, ma essi hanno tutti automaticamente linkage interno. I namespace anonimi di file diversi non sono mai fusi insieme.

212

La direttiva using

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

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

/* ... */
```

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

213

La direttiva using

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

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


La direttiva using

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

 /* ... */
```

La direttiva using

Naturalmente se dopo la using ci fosse una nuova definizione di identificatore del namespace importato, quest'ultima nasconderebbe quella del namespace. L'identificatore del namespace sarebbe comunque ancora raggiungibile qualificandolo totalmente:

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

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

int a = Pi;             // Riferisce al precedente Pi

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

216

La direttiva using

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

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

```
#include "SamLib.h"
#include "MikeLib"
using namespace MikeLib;
using SamLib::Sum(int, int);
void F() {
  PiType a = Pi;        // Riferisce a MikeLib
  int r = Sum(5, 4);    // SamLib::Sum(int, int)
}
```

217

Programmazione ad oggetti

0

Strutture e campi funzione

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

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

1

Strutture e campi funzione

In C++ cio` puo` essere realizzato tramite le strutture:

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


Strutture e campi funzione

Cio` che sostanzialmente cambia, rispetto a quanto visto, e` che una struttura puo` possedere campi di tipo funzione (detti funzioni membro oppure metodi) che costituiscono insieme ai campi ordinari (membri dato o attributi) l'insieme dei messaggi (interfaccia) a cui quel tipo e` in grado di rispondere. L'esempio non mostra come implementare le funzioni membro, per adesso ci basta sapere che esse vengono definite da qualche parte fuori dalla dichiarazione di struttura in modo pressoché identico alle ordinarie funzioni.

Una funzione dichiarata come campo di una struttura puo` essere invocata ovviamente solo se associata ad una istanza della struttura stessa, dato che quello che si fa e` inviare un messaggio ad un oggetto. Cio` nella pratica si fa tramite la stessa sintassi utilizzata per selezionare un qualsiasi altro campo (solo che ora ci sono anche campi funzione):

Strutture e campi funzione

```
Complex A;
Complex* C;

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

4

Strutture e campi funzione

Nell'esempio viene mostrato come inviare un messaggio: la quarta riga invia il messaggio Print() all'oggetto A, l'ultima invece invia il messaggio Abs() all'oggetto puntato da C e assegna il valore ottenuto alla variabile FloatVar. Anche la terza riga invia un messaggio ad A, in questo caso il messaggio richiede dei parametri che vengono forniti nello stesso modo in cui vengono forniti alle funzioni.

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

5

Strutture e campi funzione

// Con riferimento agli esempi riportati sopra:

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

6

Sintassi della classe

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

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

7

Sintassi della classe

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

public: le dichiarazioni che seguono questa keyword sono visibili sia alla classe che a cio` che sta fuori della classe e l'invocazione (selezione) di uno di questi campi e` sempre possibile;

private: tutto cio` che segue e` visibile solo alla classe stessa, l'accesso ad uno di questi campi e` possibile solo dai metodi della classe stessa;

come mostra il seguente esempio:

```
Complex A;
Complex * C;
A.Re = 10.2;          // Errore!
C -> Im = 0.5;        // Ancora errore!
A.Print();           // Ok!
C -> Print()          // Ok!
```

8

Sintassi della classe

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

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

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

9

Sintassi della classe

In realta` esiste una terza categoria di visibilita` definibile tramite la keyword protected (che pero` analizzeremo quando parleremo di ereditarieta`); la sintassi per la dichiarazione di classe e` dunque:

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

Non ci sono limitazioni al tipo di dichiarazioni possibili dentro una delle tre sezioni di visibilita`: definizioni di variabili o costanti (attributi), funzioni (metodi) oppure dichiarazioni di tipi (enumerazioni, unioni, strutture e anche classi), l'importante e` prestare attenzione a evitare di dichiarare private o protected cio` che deve essere visibile anche all'esterno della classe, in particolare le definizioni dei tipi di parametri e valori di ritorno dei metodi public.

Definizione delle funzioni membro

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

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

Definizione delle funzioni membro

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

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

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

12

Definizione delle funzioni membro

I due metodi non sono comunque del tutto identici: nel primo caso implicitamente si richiede una espansione inline del codice della funzione, nel secondo caso se si desidera tale accorgimento bisogna utilizzare esplicitamente la keyword inline nella definizione del metodo:

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

13

Definizione delle funzioni membro

Se la definizione del metodo Print() e` stata studiata con attenzione, il lettore avra` notato che la funzione accede ai membri dato senza ricorrere alla notazione del punto, ma semplicemente nominandoli: quando ci si vuole riferire ai campi dell'oggetto cui e` stato inviato il messaggio non bisogna adottare alcuna particolare notazione, lo si fa e basta (i nomi di tutti i membri della classe sono nello scope di tutti i metodi della stessa classe)!

La domanda corretta da porsi e` come si fa a stabilire dall'interno di un metodo qual e` l'effettiva istanza cui ci si riferisce. Il compito di risolvere correttamente ogni riferimento viene svolto automaticamente dal compilatore: all'atto della chiamata, ciascun metodo riceve un parametro aggiuntivo, un puntatore all'oggetto a cui e` stato inviato il messaggio e tramite questo e` possibile risalire all'indirizzo corretto; cio` inoltre consente la chiamata di un metodo da parte di un altro metodo:

Definizione delle funzioni membro

```
class MyClass {
  public:
    void BigOp();
    void SmallOp();
  private:
    void PrivateOp();
    /* altre dichiarazioni */
};
/* definizione di SmallOp() e PrivateOp() */
void MyClass::BigOp() {
  /* ... */
  SmallOp();   // messaggio all'oggetto a cui e` stato inviato BigOp()
  /* ... */
  PrivateOp(); // anche questo!
  /* ... */
}
```

Definizione delle funzioni membro

Ovviamente un metodo puo` avere parametri e/o variabili locali che sono istanze della stessa classe cui appartiene (il nome della classe e` gia` visibile all'interno della stessa classe), in questo caso per riferirsi ai campi del parametro o della variabile locale si deve utilizzare la notazione del punto:

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

void MyClass::Func(MyClass A, /* ... */ ) {
  /* ... */
  BigOp();    // questo messaggio arriva all'oggetto
```


Definizione delle funzioni membro

```
            // cui e` stato inviato Func(MyClass)
  A.BigOp();  // questo invece arriva al parametro.
  /* ... */
}
```

In alcuni rari casi puo` essere utile avere accesso al puntatore che il compilatore aggiunge tra i parametri di un metodo, l'operazione e` fattibile tramite la keyword this, che in pratica e` il nome del parametro aggiuntivo, ed e' un puntatore all'instanza cui si riferisce.

Costruttori

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

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

ma il compilatore rifiutera` di accettare tale codice. Il motivo e` semplice, stiamo definendo un tipo e non una variabile (o una costante) e non e` possibile inizializzare i membri di una classe (o di una struttura) in quel modo... E poi in questo modo ogni istanza della classe sarebbe sempre inizializzata con valori prefissati, e la situazione sarebbe sostanzialmente quella di prima.

Costruttori

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

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

Costruttori

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

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

La definizione appena vista introduce un oggetto C di tipo Complex che viene inizializzato chiamando il costruttore con gli argomenti specificati tra le parentesi. Si noti che il costruttore non viene invocato come un qualsiasi metodo (il nome del costruttore non e` cioe` invocato esplicitamente ma e` implicito nel tipo dell'istanza); un sistema alternativo di eseguire l'inizializzazione sarebbe:

Costruttori

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

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

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

In alcuni casi, alcune operazioni possono richiedere la certezza assoluta che tutti o parte dei campi dell'oggetto che si vuole creare siano subito inizializzati prima ancora che incominci l'esecuzione del corpo del costruttore; la soluzione in questi casi prende il nome di lista di inizializzazione.

Costruttori

La lista di inizializzazione e` una caratteristica propria dei costruttori e appare sempre tra la lista di argomenti del costruttore e il suo corpo:

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

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

22

Costruttori

L'ultima riga dell'esempio implementa il costruttore della classe Complex; si tratta esattamente dello stesso costruttore visto prima, la differenza sta tutta nel modo in cui sono inizializzati i membri dato: la notazione Attributo(< Espressione >) indica al compilatore che Attributo deve memorizzare il valore fornito da Espressione; Espressione puo` essere anche qualcosa di complesso come la chiamata ad una funzione.

Nel caso appena visto l'importanza della lista di inizializzazione puo` non essere evidente, lo sara` di piu` quando parleremo di oggetti composti e di ereditarieta`.

23

Costruttori

Una classe puo` possedere piu` costruttori, cioe` i costruttori possono essere overloaded, in modo da offrire diversi modi per inizializzare una istanza; in particolare alcuni costruttori assumono un significato speciale:

il costruttore di default ClassName::ClassName();

il costruttore di copia ClassName::ClassName(ClassName& X);

altri costruttori con un solo argomento;

Il costruttore di default e` particolare, in quanto e` quello che il compilatore chiama quando il programmatore non definisce esplicitamente un costruttore nella dichiarazione di un oggetto:

Costruttori

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

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

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

Costruttori

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

 cout << "definizione di C... ";
  Trace C;       // costruttore di default chiamato!
  return 0;
}
```


Costruttori

Eseguendo tale codice si ottiene l'output:

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

Ma l'importanza del costruttore di default e` dovuta soprattutto al fatto che se il programmatore della classe non definisce alcun costruttore, automaticamente il compilatore ne fornisce uno (che pero` non da` garanzie sul contenuto dei membri dato dell'oggetto). Se non si desidera il costruttore di default fornito dal compilatore, occorre definirne esplicitamente uno (anche se non di default).

Costruttori

Il costruttore di copia invece viene invocato quando un nuovo oggetto va inizializzato in base al contenuto di un altro; modifichiamo la classe Trace in modo da aggiungere il seguente costruttore di copia:

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

e aggiungiamo il seguente codice a main():

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

28

Costruttori

Cio` che viene visualizzato ora, e` che per D viene chiamato il costruttore di copia.

Se il programmatore non definisce un costruttore di copia, ci pensa il compilatore. In questo caso il costruttore fornito dal compilatore esegue una copia bit a bit (non e` proprio cosi`, ma avremo modo di vederlo in seguito) degli attributi; in generale questo e` sufficiente, ma quando una classe contiene puntatori e` necessario definirlo esplicitamente onde evitare problemi di condivisione di aree di memoria.

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

29

Costruttori

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

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

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

30

Costruttori

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

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

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

31

Costruttori

```
int main(int, chra* []) {
    cout << "definizione di A... " << endl;
    Trace A(1,2);
    cout << "definizione di B... " << endl;
    Trace B(2,4);
    cout << "definizione di C... " << endl;
    Trace C = A;
    cout << "assegnamento a C... " << endl;
    C = B;
    return 0;
  }
```

Eseguendo questo codice si ottiene il seguente output:

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

32

Costruttori

Restano da esaminare i costruttori che prendono un solo argomento.
Essi sono a tutti gli effetti dei veri e propri operatori di conversione di tipo che convertono il loro argomento in una istanza della classe. Ecco una classe che fornisce diversi operatori di conversione:

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

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

33

Costruttori

```
int main(int, char* []) {
   MyClass A(1);
   MyClass B = 5.5;
   MyClass D = (MyClass) 7;
   MyClass C = Complex(2.4, 1.0);
   return 0;
 }
```


Costruttori

Le prime tre dichiarazioni sono concettualmente identiche, in tutti e tre i casi convertiamo un valore di un tipo in quello di un altro; il fatto che l'operazione sia eseguita per inizializzare degli oggetti non modifica in alcun modo il significato dell'operazione stessa.

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

Costruttori

Si consideri questo caso:

```
class Vector {
  public:
    Vector(int s) { size=s; array=new int[size]; }
    /* ... */
  private:
    int size;
    int *array;
}
Vector v(10); // OK, chiamata esplicita del costruttore con 1 argomento int
v = 18;          // ???
```


Costruttori

Cosa fa l'ultima istruzione? Semplice: a sinistra dell'uguale c'e' una variabile di tipo Vector, a destra un int; poiche' esiste un costruttore di Vector a partire da un int, si ha una conversione da int a Vector; quindi e' come se avessimo scritto

v=Vector(18);

Viene allora utilizzato il costruttore di copia per copiare il Vector temporaneo Vector(18) su v, distruggendo il contenuto precedente di v.

Probabilmente non e' quello che si voleva!

In casi come questo, non si desidera che un costruttore con un solo argomento induca una conversione attraverso una chiamata implicita a quel costruttore; si puo' allora utilizzare la keyword explicit nella dichiarazione del costruttore per indicare che il costruttore potra' essere impiegato solo in una chiamata esplicita:

Costruttori

```
class Vector {
  public:
    explicit Vector(int s);
  /* ... */
};

Vector v(10); // ok, chiamata esplicita
v=18;         // syntax error
```


Costruttori

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

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

Distruttori

Poiche` ogni oggetto ha una propria durata (lifetime) e` necessario disporre anche di un metodo che permetta una corretta distruzione dell'oggetto stesso, un distruttore.

Un distruttore e` un metodo che non riceve parametri, non ritorna alcun tipo (neanche void) ed ha lo stesso nome della classe preceduto da ~ (tilde):

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


Distruttori

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

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

Distruttori

Si noti che nell'esempio non c'e` alcuna chiamata esplicita al distruttore, e` il compilatore che lo chiama alla fine del blocco applicativo (le istruzioni racchiuse tra { }) in cui la variabile e` stata dichiarata (alla fine del programma per variabili globali e statiche). Poiche` il distruttore fornito dal compilatore non tiene conto di aree di memoria allocate tramite membri puntatore, e` sempre necessario definirlo esplicitamente ogni qual volta esistono membri puntatori; come mostra il seguente esempio:

Distruttori

```
#include < iostream >
using namespace std;
class Trace {
  public:
    /* ... */
    Trace(long double);
    ~Trace();
  private:
    long double * ldPtr;
};
```

Distruttori

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

In tutti gli altri casi, spesso il distruttore di default e` piu` che sufficiente e non occorre scriverlo.

Solitamente il distruttore e` chiamato implicitamente dal compilatore quando un oggetto termina il suo ciclo di vita, oppure quando un oggetto allocato con new viene deallocato con delete:

44

Distruttori

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

45

Distruttori

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

Facciamo un esempio:

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

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

Distruttori

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

Distruttori

L'uso del puntatore permette di capire se la variabile e` stata deallocata (nel qual caso il puntatore viene posto a 0) oppure no (puntatore non nullo).

Comunque si tenga presente che i casi in cui si deve ricorrere ad una tecnica simile sono rari e spesso (ma non sempre) denotano un frammento di codice scritto male (quello in cui si vuole chiamare il distruttore) oppure una cattiva ingegnerizzazione della classe cui appartiene la variabile

Si noti che poiche` un distruttore non possiede argomenti, non e` possibile eseguirne l'overloading; ogni classe cioe` possiede sempre e solo un unico distruttore.

48

Membri static

Normalmente istanze diverse della stessa classe non condividono direttamente risorse di memoria, l'unica possibilita` sarebbe quella di avere puntatori che puntano allo stesso indirizzo, per il resto ogni nuova istanza riceve nuova memoria per ogni attributo. Tuttavia in alcuni casi e` desiderabile che alcuni attributi fossero comuni a tutte le istanze. Si pensi ad esempio ad un contatore che indichi il numero di istanze di una certa classe in un certo istante...

Per rendere un attributo comune a tutte le istanze occorre dichiararlo static:

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

49

Membri static

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

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

Il motivo e` abbastanza ovvio, qualunque operazione sul membro static nel corpo del costruttore verrebbe eseguita ogni volta che si istanzia la classe, una inizializzazione eseguita tramite costruttore verrebbe quindi ripetuta piu` volte rendendo inutili i membri statici.

Membri static

L'inizializzazione di un attributo static va eseguita successivamente alla sua dichiarazione ed al di fuori della dichiarazione di classe:

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

int MyClass::Counter = 0;
```

Membri static

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

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

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


Membri static

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

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

   private:
     static int Counter = 0;
     /* ... */
 };

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

Membri static

Si noti che nella definizione della funzione membro statica la keyword static non e` stata ripetuta, essa e` necessaria solo nella dichiarazione (anche in caso di definizione inline).

54

Membri static

Ci si puo` chiedere quale motivo ci possa essere per dichiarare un metodo static, ci sono essenzialmente tre giustificazioni:

maggiore controllo su possibili fonti di errore: dichiarando un metodo static, chiediamo al compilatore di accertarsi che il metodo non acceda ad altre categorie di attributi;

minor overhead di chiamata: i metodi non static per sapere a quale oggetto devono riferire, ricevono dal compilatore un puntatore all'istanza di classe per la quale il metodo e` stato chiamato; i metodi static per loro natura non hanno bisogno di tale parametro e quindi non richiedono tale overhead;

i metodi static oltre a poter essere chiamati come un normale metodo, associandoli ad un oggetto (con la notazione del punto), possono essere chiamati come una normale funzione senza necessita` di associarli ad una particolare istanza, ricorrendo al risolutore di scope come nel seguente esempio:

55

Membri static

```
MyClass Obj;

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

Si noti che nell'esempio vogliamo poter conoscere il numero di istanze di MyClass in qualsiasi momento; se dovessimo avere almeno una istanza per chiamare GetCounterValue su quell'istanza, come nel primo caso dell'esempio qui sopra, non potremmo mai avere come risposta zero. Invece, anche nel caso in cui non vi sia alcuna istanza, la chiamata MyClass::GetCounterValue() e' possibile e dara' la risposta corretta.

Non e` possibile dichiarare static un costruttore o un distruttore.

56

Membri const e mutable

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

```
class MyClass {
  public:
    MyClass(int a, float b);
    /* ... */

  private:
```

57

Membri const e mutable

```
    const int ConstMember;
    float AFloat;
};

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

Il motivo per cui bisogna ricorrere alla lista di inizializzazione e` semplice: l'assegnamento e` una operazione proibita sulle costanti, l'operazione che si compie tramite la lista di inizializzazione e` invece concettualmente diversa (anche se per i tipi primitivi e` equivalente ad un assegnamento), la cosa diverra` piu` evidente quando vedremo che il generico membro di una classe puo` essere a sua volta una istanza di una generica classe.

58

Membri const e mutable

E` anche possibile avere funzioni membro const analogamente a quanto avviene per le funzioni membro statiche. Dichiarando un metodo const si stabilisce un contratto con il compilatore: la funzione membro si impegna a non accedere in scrittura ad un qualsiasi attributo della classe e il compilatore si impegna a segnalare con un errore ogni tentativo in tal senso. Oltre a cio` esiste un altro vantaggio a favore dei metodi const: sono gli unici a poter essere eseguiti su istanze costanti (che per loro natura non possono essere modificate). Per dichiarare una funzione membro const e` necessario far seguire la lista dei parametri dalla keyword const, come mostrato nel seguente esempio:

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

59

Membri const e mutable

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

    void ChangeFloat(float b) {
      AFloat = b;
    }

  private:
    const int ConstMember;
    float AFloat;
 };
```


Membri const e mutable

```
int main(int, char* []) {
   MyClass A(1, 5.3);
   const MyClass B(2, 3.2);

   A.GetConstMember();     // Ok!
   B.GetConstMember();     // Ok!
   A.ChangeFloat(1.2);     // Ok!
   B.ChangeFloat(1.7);     // Errore!
   return 0;
}
```

Membri const e mutable

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

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


Membri const e mutable

```
int GetConstMember() const;

    /* ... */
};

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

Avremmo dovuto cioe` esplicitare nuovamente il const (cosa che non avviene con le funzioni membro static).
Come per i metodi static, non e` possibile avere costruttori e distruttori const (sebbene essi vengano utilizzati per costruire e distruggere anche le istanze costanti).

Membri const e mutable

Talvolta puo` essere necessario che una funzione membro costante possa accedere in scrittura ad uno o piu` attributi della classe, situazioni di questo genere sono rare ma possibili (si pensi ad un oggetto che mappi un dispositivo che debba trasmettere dati residenti in ROM attraverso una porta hardware, solo metodi const possono accedere alla ROM...). Una soluzione potrebbe essere quella di eseguire un cast per rimuovere la restrizione del const, ma una soluzione di questo tipo sarebbe nascosta a chi usa la classe.

Per rendere esplicita una situazione di questo tipo e` stata introdotta la keyword mutable, un attributo dichiarato mutable puo` essere modificato anche da funzioni membro costanti:

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


Membri const e mutable

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

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

Membri const e mutable

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

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

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

Costanti vere dentro le classi

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

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

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

perche` non si puo` stabilire a tempo di compilazione il valore di Size. Le possibili soluzioni al problema sono due.

La soluzione tradizionale viene dalla keyword enum; se ricordate bene, e` possibile stabilire quali valori interi associare alle costanti che appaiono tra parentesi graffe al fine di rappresentarle.

Costanti vere dentro le classi

Nel nostro caso dunque la soluzione e`:

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

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

Costanti vere dentro le classi

Questa soluzione, pur risolvendo il nostro problema, soffre di una grave limitazione: possiamo avere solo costanti intere. Una soluzione definitiva al nostro problema la si trova utilizzando contemporaneamente le keyword static e const:

```
class Array {
  public:
    /* ... */
  private:
    static const int Size = 20;
    char String[Size];          // Ok!
};
```

Essendo static, Size viene inizializzata prima della creazione di una qualunque istanza della classe ed essendo const il suo valore non puo` essere modificato e risulta quindi prefissato gia` a compile time.

Costanti vere dentro le classi

Le costanti dichiarate in questo modo possono avere tipo qualsiasi e in questo caso il compilatore puo` non allocare alcuna memoria per esse, si ricordi solo che non tutti i compilatori potrebbero accettare l'inizializzazione della costante nella dichiarazione di classe, in questo caso e` sempre possibile utilizzare il metodo visto per gli attributi static:

```
class Array {
  public:
    /* ... */
  private:
    static const int Size;
    char String[Size];          // Ok!
};
const int Array::Size = 20;
```

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

70

Membri volatile

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

```
volatile int Var;

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

In tal modo il compilatore non ottimizza gli accessi a tale risorsa e ogni tentativo di lettura di quella variabile e` tradotto in una effettiva lettura della locazione di memoria corrispondente.

71

Membri volatile

Gli oggetti volatile sono normalmente utilizzati per mappare registri di unita` di I/O all'interno del programma e per essi valgono le stesse regole viste per gli oggetti const; in particolare solo funzioni membro volatile possono essere utilizzate su oggetti volatile e non si possono dichiarare volatile costruttori e distruttori (che sono comunque utilizzabili su tali oggetti):

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

  private:
    volatile long int ComPort;
```


Membri volatile

```
    const int PortID;
    /* ... */
};

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

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

Membri volatile

Si noti che volatile non e` l'opposto di const: quest'ultima indica al compilatore che un oggetto non puo` essere modificato indipendentemente che sia trattato come una vera costante o una variabile a sola lettura, volatile invece dice che l'oggetto puo` cambiare valore al di fuori del controllo del sistema; quindi e` possibile avere oggetti const volatile. Ad esempio unita` di input, come la tastiera, possono essere mappati tramite oggetti dichiarati const volatile:

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

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


Dichiarazioni friend

In taluni casi e` desiderabile che una funzione non membro possa accedere direttamente ai membri (attributi e/o metodi) privati di una classe. Tipicamente questo accade quando si realizzano due o piu` classi, distinte tra loro, che devono cooperare per l'espletamento di un compito complessivo e si vogliono ottimizzare al massimo le prestazioni, oppure semplicemente quando ad esempio si desidera eseguire l'overloading degli operatori ostream& operator<<(ostream& o, T& Obj) e istream& operator>>(istream& o, T& Obj) per estendere le operazioni di I/O alla classe T che si vuole realizzare (in questo caso pero' e' in generale possibile, e probabilmente preferibile, definire funzioni pubbliche const che non fanno altro che restituire le variabili private necessarie).

In situazioni di questo genere, una classe puo` dichiarare una certa funzione friend (amica) abilitandola ad accedere ai propri membri privati.

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

Dichiarazioni friend

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

class MyClass {
  public:
    /* ... */

  private:
    float F1, F2;
    char C;
    void Func();
    /* ... */
```

76

Dichiarazioni friend

```
friend ostream& operator<<(ostream& o, MyClass& Obj);
  };

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

  // essendo stato dichiarato friend dentro MyClass, il
  // seguente operatore puo` accedere ai membri privati
  // della classe come una qualunque funzione membro.
  ostream& operator<<(ostream& o, MyClass& Obj) {
    o << Obj.F1 << ' ' << Obj.F2 << ' ' << Obj.C;
    return o;
  }
```

77

Dichiarazioni friend

in tal modo diviene possibile scrivere:

```
MyClass Object;
/* ... */
cout << Object;
```

L'esempio comunque risultera` meglio comprensibile quando parleremo di overloading degli operatori, per adesso e` sufficiente considerare ostream& operator<<(ostream& o, MyClass& Obj) alla stessa stregua di una qualsiasi funzione.

La keyword friend puo` essere applicata anche a un identificatore di classe, abilitando cosi` una classe intera

Dichiarazioni friend

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

in tal modo qualsiasi membro di AnotherClass puo` accedere ai dati privati di MyClass.

Si noti infine che deve essere la classe proprietaria dei membri privati a dichiarare una funzione (o una classe) friend e che non ha importanza la sezione (pubblica, protetta o privata) in cui tale dichiarazione e` fatta.

Si noti che l'*amicizia* tra classe non gode della proprieta' commutativa (cioe', se la classe A dichiara di essere amica della classe B, non vale automaticamente il contrario, a meno che anche B dichiari esplicitamente di essere amica di A) ne' della proprieta' transitiva (cioe', se A e' amica di B e B di C, A non e' automaticamente amica di C).

Riutilizzo delle classi

È possibile creare nuove classi utilizzando classi preesistenti attraverso due meccanismi:

Composizione

- Gli oggetti delle classi preesistenti vengono creati all'interno del nuovo oggetto che risulterà così "composto" da tali oggetti.

Ereditarietà

- La nuova classe viene dichiarata come appartenente alle tipologie di classi preesistenti in modo da "ereditarne" le proprietà ed eventualmente estenderle.
- Si tratta di uno dei cardini della programmazione ad oggetti.

Composizione

Benche` non sia stato esplicitamente mostrato, non c'e` alcun limite alla complessita` di un membro dato di un oggetto; un attributo puo` avere sia tipo elementare che tipo definito dall'utente, in particolare un attributo puo` a sua volta essere un oggetto.

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

    /* ... */

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

Composizione

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

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


Composizione

L'esempio mostrato suggerisce un modo di reimpiegare codice gia` pronto quando si e` di fronte ad una relazione di tipo Has-a, in cui una entita` piu` piccola e` effettivamente parte di una piu` grossa. In questo caso la composizione e` servita per modellare una proprieta` della classe Lavoratore, ma sono possibili casi ancora piu` complessi:

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

Composizione

```
class Matrix {
   public:
     Matrix();
     Matrix operator+(Matrix &);
     /* ... */

   private:
     Complex Data[10][10];
 };
```


Composizione

In questo secondo esempio invece il reimpiego della classe Complex ci consente anche di definire le operazioni sulla classe Matrix in termini delle operazioni su Complex (un approccio matematicamente corretto).
Tuttavia la composizione puo` essere utilizzata anche per modellare una relazione di tipo Is-a, in cui invece una istanza di un certo tipo puo` essere vista anche come istanza di un tipo piu` "piccolo":

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

   private:
```

Composizione

```
    const char* Name;
    unsigned int Age;
};

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


Composizione

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

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

Composizione

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

/* ... */
```

In sostanza la composizione puo` essere utilizzata anche quando vogliamo semplicemente estendere le funzionalita` di una classe realizzata in precedenza (esistono tecnologie basate su questo approccio).

Composizione

Esistono due tecniche di composizione:

Contenimento diretto;

Contenimento tramite puntatori.

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

Nel caso del contenimento tramite puntatori:

L'uso di puntatori permette di modellare relazioni 1-n, altrimenti non modellabili se non stabilendo un valore massimo per n;

Non e` necessario conoscere il modo in cui va costruito una componente nel momento in cui l'oggetto che la contiene viene istanziato;

E` possibile che piu` oggetti contenitori condividano la stessa componente;

Il contenimento tramite puntatori puo` essere utilizzato insieme all'ereditarieta` e al polimorfismo per realizzare classi di oggetti che non sono completamente definiti fino al momento in cui il tutto (compreso le parti accessibili tramite puntatori) non e` totalmente costruito.

Composizione

L'ultimo punto e` probabilmente il piu` difficile da capire e richiede la conoscenza del concetto di ereditarieta` che sara` esaminato in seguito. Sostanzialmente possiamo dire che poiche` il contenimento avviene tramite puntatori, in effetti non possiamo conoscere l'esatto tipo del componente, ma solo una sua interfaccia generica (classe base) costituita dai messaggi cui l'oggetto puntato sicuramente risponde. Questo rende il contenimento tramite puntatori piu` flessibile e potente (espressivo) del contenimento diretto, potendo realizzare oggetti il cui comportamento puo` cambiare dinamicamente nel corso dell'esecuzione del programma (con il contenimento diretto invece oltre all'interfaccia viene fissato anche il comportamento ovvero l'implementazione del componente). Pensate al caso di una classe che modelli un'auto: utilizzando un puntatore per accedere alla componente motore, se vogliamo testare il comportamento dell'auto con un nuovo motore non dobbiamo fare altro che fare in modo che il puntatore punti ad un nuovo motore. Con il contenimento diretto la struttura del motore (corrispondente ai membri privati della componente) sarebbe stata limitata e non avremmo potuto testare l'auto con un motore di nuova concezione (ad esempio uno a propulsione anziche a scoppio). Come vedremo invece il polimorfismo consente di superare tale limite. Tutto cio` sara` comunque piu` chiaro in seguito.

Composizione

Consideriamo ora i principali vantaggi e svantaggi del contenimento diretto:

L'accesso ai componenti non deve passare tramite puntatori;

La struttura di una classe e` nota gia` in fase di compilazione, si conosce subito l'esatto tipo del componente e il compilatore puo` effettuare molte ottimizzazioni (e controlli) altrimenti impossibili (tipo espansione delle funzioni inline dei componenti);

Non e` necessario eseguire operazioni di allocazione e deallocazione per costruire le componenti, ma e` necessario conoscere il modo in cui costruirle gia` quando si istanzia (costruisce) l'oggetto contenitore.

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

Costruttori per oggetti composti

L'inizializzazione di un oggetto composto richiede che siano inizializzate tutte le sue componenti. Abbiamo visto che un attributo non puo` essere inizializzato mentre lo si dichiara (infatti gli attributi static vanno inizializzati fuori dalla dichiarazione di classe; la stessa cosa vale per gli attributi di tipo oggetto:

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

  private:
```


Costruttori per oggetti composti

```
    unsigned int Attr = 5;    // Errore!
    Component Elem(10, 5);    // Errore!
    /* ... */
};
```

Il motivo e` ovvio, eseguendo l'inizializzazione nel modo appena mostrato il programmatore sarebbe costretto ad inizializzare la componente sempre nello stesso modo; nel caso si desiderasse una inizializzazione alternativa, saremmo costretti ad eseguire altre operazioni (e avremmo aggiunto overhead inutile).

La creazione di un oggetto che contiene istanze di altre classi richiede che vengano prima chiamati i costruttori per le componenti e poi quello per l'oggetto stesso; analogamente ma in senso contrario, quando l'oggetto viene distrutto, viene prima chiamato il distruttore per l'oggetto composto, e poi vengono eseguiti i distruttori per le singole componenti.

Costruttori per oggetti composti

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

```
#include < iostream >
using namespace std;
class SmallObj {
  public:
    SmallObj() {
      cout << "Costruttore SmallObj()" << endl;
    }
    SmallObj(int a, int b) : A1(a), A2(b) {
```


Costruttori per oggetti composti

```
      cout << "Costruttore SmallObj(int, int)" << endl;
    }
    ~SmallObj() {
       cout << "Distruttore ~SmallObj()" << endl;
    }

  private:
    int A1, A2;
};
```

Costruttori per oggetti composti

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


Costruttori per oggetti composti

```
  private:
    SmallObj Obj;
    char B;
};

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

Costruttori per oggetti composti

il cui output sarebbe:

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

98

Costruttori per oggetti composti

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

Si poteva pensare di realizzare il costruttore nel seguente modo:

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

99

Costruttori per oggetti composti

ma benche` funzionalmente equivalente al precedente, non genera lo stesso codice. Infatti poiche` un costruttore per SmallObj non e` esplicitamente chiamato nella lista di inizializzazione e poiche` per costruire un oggetto complesso bisogna prima costruire le sue componente, il compilatore esegue una chiamata a SmallObj::SmallObj() e poi passa il controllo a BigObj::BigObj(char, int, int). Conseguenza di cio` e` un maggiore overhead dovuto a due chiamate di funzione in piu`: una per SmallObj::SmallObj() (aggiunta dal compilatore) e l'altra per SmallObj::operator=(SmallObj&) (dovuta alla prima istruzione del costruttore).

Il motivo di un tale comportamento potrebbe sembrare piuttosto arbitrario, tuttavia in realta` una tale scelta e` dovuta alla necessita` di garantire sempre che un oggetto sia inizializzato prima di essere utilizzato.

Ovviamente poiche` ogni classe possiede un solo distruttore, in questo caso non esistono problemi di scelta!

Costruttori per oggetti composti

In pratica possiamo riassumere quanto detto dicendo che:

la costruzione di un oggetto composto richiede prima la costruzione delle sue componenti, utilizzando le eventuali specifiche presenti nella lista di inizializzazione del suo costruttore; in caso non venga specificato il costruttore da utilizzare per una componente, il compilatore utilizza quello di default. Alla fine viene eseguito il corpo del costruttore per l'oggetto composto;

la distruzione di un oggetto composto avviene eseguendo prima il suo distruttore e poi il distruttore di ciascuna delle sue componenti;

In quanto detto e` sottointeso che se una componente di un oggetto e` a sua volta un oggetto composto, il procedimento viene iterato fino a che non si giunge a componenti di tipo primitivo.

Costruttori per oggetti composti

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

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

102

Ereditarietà

Il meccanismo dell'ereditarieta` e` per molti aspetti simile a quello della composizione quando si vuole modellare una relazione di tipo Is-a.

L'idea e` quella di dire al compilatore che una nuova classe (detta classe derivata) e` ottenuta da una preesistente (detta classe base) "copiando" il codice di quest'ultima nella classe derivata eventualmente sostituendone una parte qualora una qualche funzione membro venisse ridefinita:

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

103

Ereditarietà

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

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


Ereditarietà

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

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

Con la composizione ciascuna istanza della classe contenitore possiede al proprio interno una istanza della classe componente; con l'ereditarieta` le istanze della classe derivata formalmente non contengono nessuna istanza della classe base, le definizioni fatte nella classe base vengono "quasi" immerse tra quelle della classe derivata senza alcuno strato intermedio (il "quasi" e` giustificato dal seguente punto;

Un oggetto composto puo` accedere solo ai membri pubblici della componente, l'ereditarieta` permette invece di accedere direttamente anche ai membri protetti della classe base (quelli privati rimangono inaccessibili alla classe derivata).

Accesso ai campi ereditati

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

```
class Person {
  public:
    Person();
    ~Person();
    void PrintData();
    void Sleep();

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


Accesso ai campi ereditati

```
/* Definizione dei metodi di Person */


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

  private:
```

Accesso ai campi ereditati

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

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

Il codice ereditato continua a comportarsi nella classe derivata esattamente come si comportava nella classe base: se Person::PrintData() visualizzava i membri Name e Age della classe Person, il metodo PrintData() ereditato da Student continuera` a fare esattamente la stessa cosa, solo che riferira` agli attributi propri dell'istanza di Student su cui il metodo verra` invocato.

108

Accesso ai campi ereditati

In molti casi e` desiderabile che una certa funzione membro, ereditata dalla classe base, si comporti diversamente nella classe derivata. Come alterare dunque il comportamento (codice) ereditato? Tutto quello che bisogna fare e` ridefinire il metodo ereditato; c'e` pero` un problema, non possiamo accedere direttamente ai dati privati della classe base. Come fare? Semplice: riutilizzando il metodo che vogliamo ridefinire:

```
class Student : Person {
  public:
    Student();
    ~Student();
    void DoNothing();
    void PrintData();    // ridefinisco il metodo

  private:
```

109

Accesso ai campi ereditati

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

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

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

110

Accesso ai campi ereditati

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

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

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

111

Accesso ai campi ereditati

Il linguaggio non pone alcuna limitazione circa il modo in cui PrintData() (o una qualunque funzione membro ereditata) possa essere ridefinita, in particolare avremmo potuto eliminare la chiamata a Person::PrintData(), ma avremmo dovuto trovare un altro modo per accedere ai campi privati di Person. Al di la` della fattibilita` della cosa, non sarebbe comunque buona norma agire in tal modo, non e` bene ridefinire un metodo con una semantica differente. Se Person::PrintData() aveva il compito di visualizzare lo stato dell'oggetto, anche Student::PrintData() deve avere lo stesso compito. Stando cosi` le cose, richiamare il metodo della classe base significa ridurre la possibilita` di commettere un errore e risparmiare tempo e fatica.

Accesso ai campi ereditati

E` per questo motivo infatti che non tutti i membri vengono effettivamente ereditati: costruttori, distruttore, operatore di assegnamento e operatori di conversione di tipo non vengono ereditati perche` la loro semantica e` troppo legata alla effettiva struttura di una classe (il compilatore comunque continua a fornire per la classe derivata un costruttore di default, uno di copia e un operatore di assegnamento, esattamente come per una qualsiasi altra classe e con una semantica prestabilita); il codice di questi membri e` comunque disponibile all'interno della classe derivata (nel senso che possiamo richiamarli tramite il risolutore di scope ::).

Naturalmente la classe derivata puo` anche definire nuovi metodi, compresa la possibilita` di eseguire l'overloading di una funzione ereditata (naturalmente la versione overloaded deve differire dalle precedenti per tipo e/o numero di parametri, altrimenti si ha ridefinizione del metodo ereditato, non overloading). Infine non e` possibile ridefinire gli attributi (membri dato) della classe base.

Costruttori per classi derivate

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

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

Si deve pero` considerare che non si puo` accedere ai campi privati della classe base, e non e` neanche possibile scrivere codice simile:

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

114

Costruttori per classi derivate

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

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

Nel modo appena visto si chiede al compilatore di costruire e inizializzare i membri ereditati utilizzando un certo costruttore della classe base con i parametri attuali da noi indicati. Se nessun costruttore per la classe base viene menzionato il compilatore richiama il costruttore di default, generando un errore se la classe base non ne possiede uno.

Se il programmatore non specifica alcun costruttore per la classe derivata, il compilatore ne fornisce uno di default che richiama quello di default della classe base. Considerazioni analoghe valgono per il costruttore di copia fornito dal compilatore (richiama quello della classe base).

115

Tipi di ereditarietà

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

Con l'ereditarieta` pubblica i membri ereditati mantengono lo stesso grado di protezione che avevano nella classe da cui si eredita (classe base immediata): i membri public rimangono public e quelli protected continuano ad essere protected;

Con l'ereditarieta` protetta i membri public della classe base divengono membri protected della classe derivata; quelli protected rimangono tali.

La sintassi completa per l'ereditarieta` diviene dunque:

Tipi di ereditarietà

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

dove Qualifier e` opzionale e puo` essere uno tra public, protected e private; se omesso si assume private.

Lo standard ANSI consente anche la possibilita` di esportare singolarmente un membro in presenza di ereditarieta` privata o protetta, con l'ovvio limite di non rilasciare il grado di protezione che esso possedeva nella classe base:

Tipi di ereditarietà

```
class MyClass {
  public:
    void PublicMember(int, char);
    void Member2();
    /* ... */

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

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


Tipi di ereditarietà

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

    MyClass::ProtectedMember;  // Errore!
    /* ... */
};
```

Tipi di erditarietà

```
class Derived2 : private MyClass {
   public:
     MyClass::PublicMember;     // Ok!

   protected:
     MyClass::ProtectedMember;  // Ok!
     /* ... */
 };

 class Derived3 : private MyClass {
```


Tipi di erditarietà

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

L'esempio mostra sostanzialmente tutte le possibili situazioni, compresa il caso di un errore dovuto al tentativo di far diventare public un membro che era protected. Si noti la notazione utilizzata, non e` necessario specificare niente piu` del semplice nome del membro preceduto dal nome della classe base e dal risolutore di scope. Un metodo alternativo e` dato dall'uso della direttiva using per importare nel namespace della classe derivata, un nome appartenente al namespace della classe base.

Tipi di erditarietà

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

Il tentativo di utilizzare una funzione non esportata viene segnalato solo a run-time;

E` una operazione che costringe il programmatore a lavorare di piu` aumentando la possibilita` di errore e diminuendone la produttivita`.

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

L'introduzione di queste funzioni inoltre potrebbe generare overhead run-time, cosa che non avviene nella gestione degli accessi ai membri fatta dal compilatore.

122

Tipi di erditarietà

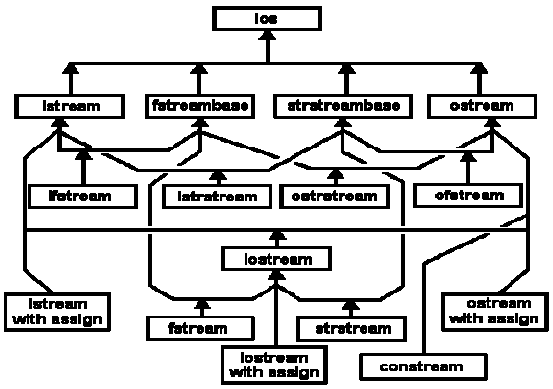
I vari "tipi" di derivazione (ereditarieta`) hanno conseguenze che vanno al di la` della semplice variazione del livello di protezione di un membro.

Con l'ereditarieta` pubblica si modella effettivamente una relazione di tipo Is-a poiche` la classe derivata continua ad esportare l'interfaccia della classe base (e` cioe` possibile utilizzare un oggetto derived come un oggetto base); con l'ereditarieta` privata questa relazione cessa, in un certo senso possiamo vedere l'ereditarieta` privata come una sorta di contenimento. L'ereditarieta` protetta e` invece una sorta di ibrido ed e` scarsamente utilizzata.

123

Ereditarieta` multipla

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



124

Ereditarieta` multipla

come si puo` vedere esistono diverse classi ottenute per ereditarieta` multipla, iostream ad esempio che ha come classi basi istream e ostream.

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

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

L'ereditarieta` multipla comporta alcune problematiche che non si presentano in caso di ereditarieta` singola, quella a cui si puo` pensare per prima e` il caso in cui le stesse definizioni siano presenti in piu` classi base (name clash):

125

Ereditarieta` multipla

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

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

126

Ereditarieta` multipla

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

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

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

non puo` che generare un errore perche` il compilatore non sa a quale membro si riferisce l'assegnamento. Si noti che l'errore viene segnalato al momento in cui si tenta di chiamare il metodo e non al momento in cui Derived eredita, il fatto che un membro sia ereditato piu` volte non costituisce di per se alcun errore.

127

Ereditarieta` multipla

Rimane comunque il problema di eliminare l'ambiguita` nella chiamata di Foo(), la soluzione consiste nell'utilizzare il risolutore di scope indicando esplicitamente quale delle due Foo():

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

in questo modo non esiste piu` alcuna ambiguita`.

Alcune osservazioni:
quanto detto vale anche per i membri dato;

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

Ereditarieta` multipla

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

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

Ereditarieta` multipla

Nuovamente SecondDerived presenta lo stesso problema, e` cioe` sufficiente che la stessa definizione giunga attraverso classi basi indirette (nel precedente esempio BaseClass2 e` una classe base indiretta di SecondDerived);

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

Il problema diviene piu` grave quando una o piu` copie della stessa definizione sono nascoste dalla keyword private nelle classi basi (dirette o indirette), in tal caso la classe derivata non ha alcun controllo su quella o quelle copie (in quanto vi accede indirettamente tramite le funzioni membro ereditate) e il pericolo di inconsistenza dei dati diviene piu` grave.

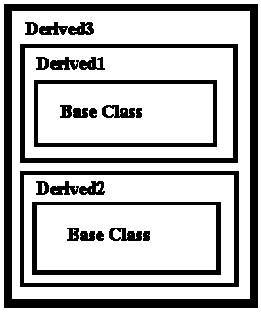
Ereditarieta` multipla

Il problema dell'ambiguita` che si verifica con l'ereditarieta` multipla, puo` essere portato al caso estremo in cui una classe ottenuta per ereditarieta` multipla erediti piu` volte una stessa classe base:

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

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

Ereditarieta` multipla

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

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



Di nuovo quello che succede e` che alcuni membri (in particolare tutta una classe) sono duplicati nella classe Derived3.

Consideriamo l'immagine in memoria di una istanza della classe Derived3, la situazione che avremmo sarebbe la seguente:

132

Ereditarieta` multipla

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

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

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

Supponiamo ad esempio di avere una classe Person e di derivare da essa prima una classe Student e poi una classe Employee al fine di modellare un mondo di persone che eventualmente possono essere studenti o impiegati; dopo un po' ci accorgiamo che una persona puo` essere contemporaneamente uno studente ed un lavoratore, cosi` tramite l'ereditarieta` multipla deriviamo da Student e Employee la classe Student_Employee. Il problema e` che la nuova classe contiene due istanze della classe Person e queste due istanze vengono accedute (in lettura e scrittura) indipendentemente l'una dall'altra...

Cosa accadrebbe se nelle due istanze venissero memorizzati dati diversi? Una gravissima forma di inconsistenza!

Anche se non vi fosse inconsistenza, non e' corretto che venga impiegato il doppio della memoria necessaria per mantenere le informazioni relative a Person.

133

Ereditarieta` multipla

Si noti che questo non e' semplicemente un problema di ambiguita': infatti, in molti casi e' proprio sbagliato che la classe di base comune sia “duplicata”, o ereditata due volte. L'esempio seguente chiarisce il concetto.

class Person;
class Sudent : public Person;
class Employee : public Person;
class Student_Employee : public Student, public Employee;

E' chiaro che, se uno studente lavoratore e' sia uno studente che un lavoratore, non per questo diventa due persone, ma resta una persona sola.

Ereditarieta` multipla

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

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

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

class Employee : virtual public Person {
```

Ereditarieta` multipla

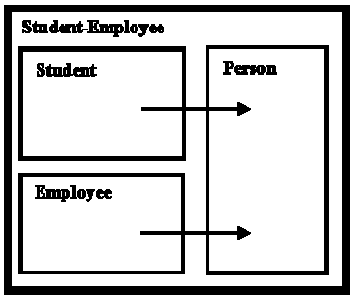
```
  /* ... */
};

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

Si tratta di un esempio che nella pratica non avrebbe alcuna validita`, ma ottimo da un punto di vista didattico.

Ereditarieta` multipla

Vediamo piu` in dettaglio cosa e cambiato e come virtual opera. Quando una classe eredita tramite la keyword virtual il compilatore non si limita a copiare il contenuto della classe base nella classe derivata, ma inserisce nella classe derivata un puntatore ad una istanza della classe base. Quando una classe eredita (per ereditarieta` multipla) piu` volte una classe base virtuale (e` questo il caso di Student_Employee che eredita piu` volte da Person), il compilatore inserisce solo una istanza della classe virtuale e fa si che tutti i puntatori a tale classe puntino a quell'unica istanza.

La situazione in questo caso e` illustrata dalla seguente figura:



Ereditarieta` multipla

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

Il puntatore alla classe base virtuale non e` visibile al programmatore, non bisogna tener conto di esso poiche` viene aggiunto dal compilatore a compile-time, tutto il meccanismo e` completamente trasparente, semplicemente si accede ai membri della classe base virtuale come si farebbe con una normale classe base.

Ereditarieta` multipla

Il vantaggio di questa tecnica e` che non e` piu` necessario definire la classe Student_Employee derivandola da Student (al fine di eliminare la fonte di inconsistenza) e aggiungendo a mano le definizioni di Employee, in tal modo si risparmiano tempo e fatica riducendo la quantita` di codice da produrre e limitando la possibilita` di errori.

C'e` pero` un costo da pagare: un livello di indirezione in piu` perche` l'accesso alle classi base virtuali (nell'esempio Person) avviene tramite un puntatore.

L'ereditarieta` virtuale risolve dunque l'ambiguita` di cui sopra, nelle classi derivate le definizioni di una classe base virtuale sono presenti una volta sola, tuttavia il problema dell'ambiguita` non e` del tutto risolto, esistono ancora situazioni in cui il problema si ripropone. Supponiamo ad esempio che una delle classi intermedie ridefinisca una funzione membro della classe base:

Ereditarieta` multipla

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

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


Ereditarieta` multipla

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

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

Ereditarieta` multipla

Se Student_Employee non ridefinisce il metodo DoSomething(), la situazione seguente presenterebbe ancora ambiguita`:

```
Student_Employee Caio;
/* ... */
Caio.DoSomething();       // Ambiguo!
```

perche` la classe Student_Employee eredita nuovamente due diverse definizioni del metodo DoSomething().
Esiste anche un caso apparentemente ambiguo e simile al precedente:

```
class Person {
  public:
```

142

Ereditarieta` multipla

```
    void DoSomething();
  /* ... */
};

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

143

Ereditarieta` multipla

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

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


Ereditarieta` multipla

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

```
Student_Employee Caio;
/* ... */
Caio.DoSomething();
```

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

Ritorniamo a parlare a proposito della classe base virtuale.

Ereditarieta` multipla

Nei vari costruttori delle classi derivate c'e` implicitamente o esplicitamente una chiamata al costruttore della classe base virtuale, ma ora abbiamo una sola istanza di tale classe e non possiamo certo inizializzarla piu` volte.

Nel nostro esempio la classe base virtuale Person e` inizializzata sia da Student che da Employee, entrambe le classi hanno il dovere di eseguire la chiamata al costruttore della classe base, ma quando queste due classi vengono fuse per derivare la classe Student_Employee il costruttore della nuova classe, chiamando i costruttori di Student e Employee, implicitamente chiamerebbe due volte il costruttore di Person.

E` necessario stabilire un criterio deterministico che stabilisca chi deve inizializzare la classe virtuale. Lo standard stabilisce che il compito di inizializzare la classe base virtuale spetta alla classe massimamente derivata. La classe massimamente derivata e` quella che noi stiamo istanziando: se vogliamo creare un oggetto di tipo Student la classe massimamente derivata e` in questo caso Student, se invece stiamo istanziando Student_Employee allora e` quest'ultima la classe massimamente derivata.

Ereditarieta` multipla

E` dunque il costruttore della classe massimamente derivata che inizializza la classe virtuale.

Il seguente codice

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

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

Ereditarieta` multipla

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

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

/* ... */
```


Ereditarieta` multipla

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

opportunamente completato, produrrebbe il seguente output:

Ereditarieta` multipla

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

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

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

150

Metodi virtuali

```
Definizione di Bruto:
 Costruttore Person invocato...
 Costruttore Student invocato...
 Costruttore Employee invocato...
 Costruttore Student_Employee invocato...
```

Come potete osservare il costruttore della classe Person viene invocato una sola volta, per verificare poi da chi viene invocato basta tracciare l'esecuzione con un debugger simbolico.

Naturalmente ci sarebbe un problema simile anche con il distruttore, bisogna evitare che si tenti di distruggere la classe base virtuale piu` volte; nuovamente e` il compilatore che si assume l'onere di fare in modo che l'operazione venga eseguita una sola volta

151

Metodi virtuali

Il meccanismo dell'ereditarieta` e` stato gia` di per se una grande innovazione nel mondo della programmazione, tuttavia le sorprese non si esauriscono qui. Esiste un'altra caratteristica tipica dei linquaggi a oggetti (C++ incluso): la possibilita` di avere oggetti capaci di "recitare" di volta in volta il ruolo piu` appropriato, ma andiamo con ordine.

L'ereditarieta` pone nuove regole circa la compatibilita` dei tipi, in particolare se Ptr e` un puntatore di tipo T, allora Ptr puo` puntare non solo a istanze di tipo T ma anche a istanze di classi derivate da T (sia tramite ereditarieta` semplice che multipla). Se Td e` una classe derivata (anche indirettamente) da T, istruzioni del tipo

Metodi virtuali

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

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

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

Metodi virtuali

Purtroppo la cosa cosi` com'e` non puo` funzionare poiche` le funzioni sono linkate staticamente dal linker. Anche se tutte le classi della gerarchia possiedono un metodo Paint(), noi sappiamo solo che Ptr punta ad un oggetto di tipo T o T-derivato, non conoscendo l'esatto tipo una chiamata a Ptr->Paint() non puo` che essere risolta chiamando Ptr->T::Paint() (che non fara` in generale cio` che vorremmo). Il compilatore non puo` infatti rischiare di chiamare il metodo di una classe derivata (quale?), poiche` questo potrebbe tentare di accedere a membri che non fanno parte dell'effettivo tipo dell'oggetto (causando inconsistenze o un crash del sistema), chiamando il metodo della classe T al piu` il programma non fara` la cosa giusta, ma non mettera` in pericolo la sicurezza e l'affidabilita` del sistema (perche` un oggetto derivato possiede tutti i membri della classe base).

Si potrebbe risolvere il problema inserendo in ogni classe della gerarchia un campo che stia ad indicare l'effettivo tipo dell'istanza:

Metodi virtuali

```
enum TypeId { T_Type, Td_Type };

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

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

Metodi virtuali

e risolvere il problema con una istruzione switch:

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

Una soluzione di questo tipo funziona ma e` macchinosa, allunga il lavoro e una dimenticanza puo` costare cara, e soprattutto ogni volta che si modifica la gerarchia di classi bisogna modificare anche il codice che la usa.

La soluzione migliore e` invece quella di far in modo che il corretto tipo dell'oggetto puntato sia automaticamente determinato al momento della chiamata della funzione e rinviando il linking di tale funzione a run-time.

156

Metodi virtuali

Per fare cio` bisogna dichiarare la funzione membro virtual:

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

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

La definizione del metodo procede poi nel solito modo:

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

157

Metodi virtuali

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

158

Metodi virtuali

Le funzioni virtuali hanno il grande vantaggio di consentire l'aggiunta di nuove classi alla gerarchia e di renderle immediatamente e correttamente utilizzabili dal vostro programma senza doverne modificare il codice (ovviamente il programma dovra` comunque essere modificato in modo che possa istanziare le nuove classi, ma il codice che gestisce che lavorava sul generico supertipo non avra` bisogno di modifiche), il late binding fara` in modo che siano chiamate sempre le funzioni corrette senza che il vostro programma debba curarsi dell'effettivo tipo dell'istanza che sta manipolando.

L'invocazione di un metodo virtuale e` piu` costosa di quella per una funzione membro ordinaria, tuttavia il compilatore puo` evitare tale overhead risolvendo a compile-time tutte quelle situazioni in cui il tipo e` effettivamente noto. Ad esempio:

```
Td Obj1;
T* Ptr = 0;
```

159

Metodi virtuali

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

La prima chiamata al metodo Paint() puo` essere risolta in fase di compilazione perche` il tipo di Obj1 e` sicuramente Td, nel secondo caso invece non possiamo saperlo (anche se un compilatore intelligente potrebbe cercare di restringere le possibilita` e, in caso di certezza assoluta, risolvere staticamente la chiamata).

Metodi virtuali

Se poi volete avere il massimo controllo, potete costringere il compilatore ad una "soluzione statica" utilizzando il risolutore di scope:

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

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

Polimorfismo

Il meccanismo delle funzioni virtuali e` alla base del polimorfismo: poiche` l'oggetto puntato da un puntatore puo` appartenere a tutta una gerarchia di tipi, possiamo considerare l'istanza puntata come un qualcosa che puo` assumere piu` forme (tipi) e comportarsi sempre nel modo migliore "recitando" di volta in volta il ruolo corretto, in realta` pero` un'istanza non puo` cambiare tipo, e' solo il puntatore che ha la possibilita` di puntare a tutta una gerarchia di classi.

Limiti e regole del polimorfismo

Esistono limitazioni e regole connesse all'uso dei metodi virtuali legate piu` o meno direttamente al funzionamento del polimorfismo; vediamole in dettaglio.

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

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

Limiti e regole del polimorfismo

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

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


Limiti e regole del polimorfismo

e` lo stesso che scrivere

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

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

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

Limiti e regole del polimorfismo

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

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

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

166

Limiti e regole del polimorfismo

```
};

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

167

Limiti e regole del polimorfismo

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

  int main(int, char* []) {
    Base* BasePtr = new Derived1;
    Derived* DerivedPtr = new Derived2;
```


Limiti e regole del polimorfismo

```
    BasePtr -> Foo();
    DerivedPtr -> Foo();
    return 0;
  }
```

Eseguendo il precedente codice, otterreste questo output:

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

Limiti e regole del polimorfismo

Essendo BasePtr un puntatore alla classe base, il compilatore utilizzera' il metodo della classe di base. Nel secondo caso, poiche` DerivedPtr puo` puntare solo a istanze di Derived1 o sue sottoclassi, viene eseguito regolarmente il late binding perche` il metodo Foo() sara` sempre virtual.

Da questo comportamento deriva un suggerimento ben preciso: se una classe potrebbe in futuro essere derivata (se pensate cioe` che ci siano validi motivi per specializzarla ulteriormente), e` bene che i metodi dell'interfaccia (potenzialmente soggetti a ridefinizione) siano sempre virtual, onde evitare potenziali errori; d'altronde se non si usa il polimorfismo e` comunque possibile forzare la risoluzione del binding a compile time.

Questo e' un limite del polimorfismo rispetto all'ereditarieta': mentre si puo' sempre derivare una classe da qualsiasi altra, indipendentemente da come questa e' fatta, per poter applicare il polimorfismo occorre che la classe di base sia fatta in un certo modo.

170

Limiti e regole del polimorfismo

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

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

171

Limiti e regole del polimorfismo

Consideriamo il seguente esempio:

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

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


Limiti e regole del polimorfismo

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

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

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

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

Limiti e regole del polimorfismo

La costruzione di un oggetto Derived richiede che sia prima eseguito il costruttore di Base. Al momento in cui viene eseguita la chiamata a Foo() contenuta nel costruttore della classe base il puntatore alla VTABLE puo` al piu` puntare alla VTABLE della classe Base perche` il costruttore della classe derivata non e` ancora stato eseguito e non e` quindi possibile chiamare Derived::Foo(), si puo` chiamare solo la versione locale di Foo(). La situazione e` analoga nel caso dei distruttori, al momento in cui viene eseguito il distruttore della classe base, il distruttore della classe derivata e` gia` stato eseguito ed il puntatore alla VTABLE non e` piu` valido; di conseguenza si puo` invocare solo Base::Foo().

Il suggerimento e` quindi quello di evitare per quanto possibile la chiamata di metodi virtuali all'interno di costruttori e distruttori, il risultato che potreste ottenere molto probabilmente non sarebbe quello desiderato.

174

Limiti e regole del polimorfismo

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

```
class T {
  public:
    virtual void Foo();
    virtual void Foo2();
    void DoSomething();

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

175

Limiti e regole del polimorfismo

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

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


Limiti e regole del polimorfismo

```
class Td : public T {
  public:
    virtual void Foo2();
    void DoSomething();

  private:
    /* ... */
};

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

Limiti e regole del polimorfismo

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

Si tratta di una situazione pericolosa: la classe Td ridefinisce un metodo non virtuale (ma poteva anche essere virtuale), ma non uno virtuale da questo richiamato. Di per se non si tratta di un errore, la classe derivata potrebbe non aver alcun motivo per ridefinire il metodo ereditato,

Limiti e regole del polimorfismo

tuttavia puo` essere difficile capire cosa esattamente faccia il metodo Td::DoSomething(), soprattutto in un caso simile:

```
class Td2 : public Td {
  public:
    virtual void Foo();

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

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

Limiti e regole del polimorfismo

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

viene chiamato il metodo Td::DoSomething() ereditato, ma in effetti questo poi chiama Td2::Foo() per via del linking dinamico.
Il risultato in queste situazioni e` che il comportamento che una classe puo` avere e` molto difficile da controllare e potrebbe essere potenzialmente errato; l'errore legato a situazioni di questo tipo e` noto in letteratura come fragile class problem e` puo` essere causa di forti inconsistenze.

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

180

Classi astratte

Il meccanismo dell'ereditarieta` e quello del polimorfismo possono essere combinati per realizzare delle classi il cui unico scopo e` quello di stabilire una interfaccia comune a tutta una gerarchia di classi:

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

181

Classi astratte

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

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

Classi astratte

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

Classi astratte

```
class TPoint : public TShape {
  public:
    TPoint(int x, int y) : X(x), Y(y) {}

  private:
    int X, Y;                // coordinate del punto
};

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

184

Classi astratte

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

Non e` possibile creare istanze della classe TShape, ma la classe TPoint ridefinisce tutte le funzioni virtuali pure e puo` essere istanziata e utilizzata dal programma; la classe TShape e` comunque ancora utile al programma, perche` possiamo dichiarare puntatori di tale tipo per gestire una lista di oggetti polimorfi che possiamo utilizzare senza preoccuparci del fatto che possano essere punti, triangoli o quant'altro.

185

Overloading degli operatori

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

Non e` possibile definire nuovi operatori, si puo` solamente eseguire l'overloading di uno per cui esiste gia` un simbolo nel linguaggio. Possiamo ad esempio definire un nuovo operatore *, ma non possiamo definire un operatore **.

Non e` possibile modificare la precedenza di un operatore e non e` possibile modificarne il numero di argomenti o l'associativita`, un operatore unario rimarra`sempre unario, uno binario dovra` applicarsi sempre a due operandi; analogamente uno associativo a sinistra rimarrà sempre associativo a sinistra.

Non e` concessa la possibilita` di eseguire l'overloading di alcuni operatori, ad esempio l'operatore ternario ? :, l'operatore sizeof e gli operatori di cast e in particolare l'operatore .* e l'operatore punto (per la selezione dei campi di una struttura).

Overloading degli operatori

E` possibile ridefinire un operatore sia come funzione globale che come funzione membro, i seguenti operatori devono tuttavia essere sempre funzioni membro non statiche: operatore di assegnamento (=), operatore di sottoscrizione ([ ]) e l'operatore ->.

A parte queste poche restrizioni non esistono molti altri limiti, possiamo ridefinire anche l'operatore virgola (,) e persino l'operatore chiamata di funzione (()); inoltre non c'e` alcuna restrizione riguardo il contenuto del corpo di un operatore: un operatore altro non e` che un tipo particolare di funzione e tutto cio` che puo` essere fatto in una funzione puo` essere fatto anche in un operatore.

Un operatore e` indicato dalla keyword operator seguita dal simbolo dell'operatore, per eseguirne l'overloading come funzione globale bisogna utilizzare la seguente sintassi:

Overloading degli operatori

```
< ReturnType > operator@( < ArgumentList > ) { < Body > }
```

ReturnType e` il tipo restituito (non ci sono restrizioni); @ indica un qualsiasi simbolo di operatore valido; ArgumentList e` la lista di parametri (tipo e nome) che l'operatore riceve, i parametri sono due per un operatore binario (il primo e` quello che compare a sinistra dell'operatore quando esso viene applicato) mentre e` uno solo per un operatore unario. Infine Body e` la sequenza di istruzioni che costituiscono il corpo dell'operatore.

188

Overloading degli operatori

Ecco un esempio di overloading di un operatore come funzione globale:

```
struct Complex {
  float Re;
  float Im;
};
Complex operator+(const Complex& A, const Complex& B) {
  Complex Result;
  Result.Re = A.Re + B.Re;
  Result.Im = A.Im + B.Im;
  return Result;
}
```

Si tratta sicuramente di un caso molto semplice, che fa capire che in fondo un operatore altro non e` che una funzione. Il funzionamento del codice e` chiaro e non mi dilunghero` oltre; si noti solo che i parametri sono passati per riferimento, non e` obligatorio, ma solitamente e` bene passare i parametri in questo modo (eventualmente utilizzando const come nell'esempio).

189

Overloading degli operatori

Definito l'operatore, e` possibile utilizzarlo secondo l'usuale sintassi riservata agli operatori, ovvero come nel seguente esempio:

```
Complex A, B;
/* ... */
Complex C = A + B;
```

L'esempio richiede che sia definito su Complex il costruttore di copia, ma come gia` sapete il compilatore e` in grado di fornirne uno di default. Detto questo il precedente esempio viene tradotto (dal compilatore) in

Overloading degli operatori

```
Complex A, B;
/* ... */
Complex C(operator+(A, B));
```

Volendo potete utilizzare gli operatori come funzioni, esattamente come li traduce il compilatore (cioe` scrivendo Complex C = operator+(A, B) o Complex C(operator+(A, B))), ma non e` una buona pratica in quanto annulla il vantaggio ottenuto ridefinendo l'operatore.

Quando un operatore viene ridefinito come funzione membro il primo parametro e` sempre l'istanza della classe su cui viene eseguito e non bisogna indicarlo nella lista di argomenti, un operatore binario quindi come funzione globale riceve due parametri ma come funzione membro ne riceve solo uno (il secondo operando); analogamente un operatore unario come funzione globale prende un solo argomento, ma come funzione membro ha la lista di argomenti vuota.

Overloading degli operatori

Riprendiamo il nostro esempio di prima ampliandolo con nuovi operatori:

```
class Complex {
  public:
    Complex(float re, float im);
    Complex operator-() const;    // - unario
    Complex operator+(const Complex& B) const;
    const Complex & operator=(const Complex& B);
  private:
    float Re;
    float Im;
};

Complex::Complex(float re, float im = 0.0) {
  Re = re;
  Im = im;
}
```

192

Overloading degli operatori

```
Complex Complex::operator-() const {
   return Complex(-Re, -Im);
}

Complex Complex::operator+(const Complex& B) const {
  return Complex(Re+B.Re, Im+B.Im);
}

const Complex& Complex::operator=(const Complex& B) {
  Re = B.Re;
  Im = B.Im;
  return B;
}
```

193

Overloading degli operatori

La classe Complex ridefinisce tre operatori. Il primo e` il -(meno) unario, il compilatore capisce che si tratta del meno unario dalla lista di argomenti vuota, il meno binario invece, come funzione membro, deve avere un parametro. Successivamente viene ridefinito l'operatore + (somma), si noti la differenza rispetto alla versione globale. Infine viene ridefinito l'operatore di assegnamento che come detto sopra deve essere una funzione membro non statica; si noti che a differenza dei primi due questo operatore ritorna un riferimento, in tal modo possiamo concatenare piu` assegnamenti evitando la creazione di inutili temporanei, l'uso di const assicura che il risultato non venga utilizzato per modificare l'oggetto. Infine, altra osservazione, l'ultimo operatore non e` dichiarato const in quanto modifica l'oggetto su cui e` applicato (quello cui si assegna), se la semantica che volete attribuirgli consente di dichiararlo const fatelo, ma nel caso dell'operatore di assegnamento (e in generale di tutti) e` consigliabile mantenere la coerenza semantica (cioe` ridefinirlo sempre come operatore di assegnamento, e non ad esempio come operatore di uguaglianza).

194

Overloading degli operatori

Ecco alcuni esempi di applicazione dei precedenti operatori e la loro rispettiva traduzione in chiamate di funzioni (A, B e C sono variabili di tipo Complex):

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

L'ultimo esempio e` errato poiche` quello che si vuole utilizzare e` il meno binario, e tale operatore non e` stato definito.

Passiamo ora ad esaminare con maggiore dettaglio alcuni operatori che solitamente svolgono ruoli piu` difficili da capire.

195

L'operatore di assegnamento

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

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

che e` equivalente a

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

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

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

196

L'operatore di assegnamento

Un'altra particolarita` di questo operatore lo rende simile al costruttore (oltre al fatto che deve essere una funzione membro): se in una classe non ne viene definito uno nella forma X::operator=(X&), il compilatore ne fornisce uno che esegue la copia bit a bit. Lo standard stabilisce che sia il costruttore di copia che l'operatore di assegnamento forniti dal compilatore debbano eseguire non una copia bit a bit, ma una inizializzazione o assegnamento a livello di membri chiamando il costruttore di copia o l'operatore di assegnamento relativi al tipo di quel membro. In ogni caso comunque è necessario definire esplicitamente sia l'operatore di assegnamento che il costruttore di copia ogni qual volta la classe contenga puntatori, onde evitare spiacevoli condivisioni di memoria.

Notate infine che, come per le funzioni, anche per un operatore e` possibile avere piu` versioni overloaded; in particolare una classe puo` dichiarare piu` operatori di assegnamento, ma e` quello di cui sopra che il compilatore fornisce quando manca.

197

L'operatore di sottoscrizione

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

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

dove T puo` essere anche un riferimento o un puntatore.
Restituendo un riferimento l'espressione Arg1[Arg2][Arg3] viene tradotta in Arg1.operator[](Arg2).operator[](Arg3).

198

L'operatore di sottoscrizione

Il seguente codice mostra un esempio di overloading di questo operatore, in un caso molto piu' semplice di indicizzazione monodimensionale:

```
class TArray {
public:
   TArray(unsigned int Size);
   ~TArray();
   int operator[](unsigned int Index);

 private:
   int* Array;
   unsigned int ArraySize;
};
```

199

L'operatore di sottoscrizione

```
TArray::TArray(unsigned int Size) {
  ArraySize = Size;
  Array = new int[Size];
}

TArray::~TArray() {
  delete[] Array;
}

int Tarray::operator[ ](unsigned int Index) {
 if (Index < Size) return Array[Index];
  else /* Errore */
}
```


L'operatore di sottoscrizione

Si tratta di una classe che incapsula il concetto di array per effettuare dei controlli sull'indice, evitando cosi` accessi fuori limite. La gestione della situazione di errore e` stata appositamente omessa, vedremo meglio come gestire queste situazioni quando parleremo di eccezioni.

Notate che l'operatore di sottoscrizione restituisce un int e non e` pertanto possibile usare indicizzazioni multiple, d'altronde la classe e` stata concepita unicamente per realizzare array monodimensionali di interi; una soluzione migliore, piu` flessibile e generale avrebbe richiesto l'uso dei template che saranno argomento del successivo capitolo.

Operatori && e ||

Anche gli operatori di AND e OR logico possono essere ridefiniti, tuttavia c'e` una profonda differenza tra quelli predefiniti e quelli che l'utente puo` definire. La versione predefinita di entrambi gli operatori eseguono valutazioni parziali degli argomenti: l'operatore valuta l'operando di sinistra, ma valuta anche quello di destra solo quando il risultato dell'operazione e` ancora incerto. In questi esempi l'operando di destra non viene mai valutato:

```
int var1 = 1;
int var2 = 0;

int var3 = var2 && var1;
var3 = var1 || var2;
```

In entrambi i casi il secondo operando non viene valutato poiche` il valore del primo e` sufficiente a stabilire il risultato dell'espressione.

Le versioni sovraccaricate definite dall'utente non si comportano in questo modo, entrambi gli argomenti dell'operatore sono sempre valutati (al momento in cui vengono passati come parametri).

202

Smart pointer

Un operatore particolarmente interessante e` quello di dereferenzazione -> il cui comportamento e` un po' difficile da capire.

Se T e` una classe che ridefinisce -> (l'operatore di dereferenzazione deve essere un funzione membro non statica) e Obj e` una istanza di tale classe, l'espressione

```
Obj -> Field;
```

e` valutata come

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

203

Smart pointer

Conseguenza di cio` e` che il risultato di questo operatore deve essere uno tra

un puntatore ad una struttura o una classe che contiene un membro Field;
una istanza di un'altra classe che ridefinisce a sua volta l'operatore. In questo caso l'operatore viene applicato ricorsivamente all'oggetto ottenuto prima, fino a quando non si ricade nel caso precedente;
In questo modo e` possibile realizzare puntatori intelligenti (smart pointer), capaci di eseguire controlli per prevenire errori disastrosi.
Pur essendo un operatore unario postfisso, il modo in cui viene trattato impone che ci sia sul lato destro una specie di secondo operando; se volete potete pensare che l'operatore predefinito sia in realta` un operatore binario il cui secondo argomento e` il nome del campo di una struttura, mentre l'operatore che l'utente puo` ridefinire deve essere unario.

204

L'operatore virgola

Anche la virgola e` un operatore (binario) che puo` essere ridefinito. La versione predefinita dell'operatore fa si` che entrambi gli argomenti siano valutati, ma il risultato prodotto e` il valore del secondo (quello del primo argomento viene scartato). Nella prassi comune, la virgola e` utilizzata per gli effetti collaterali derivanti dalla valutazione delle espressioni:

```
int A = 5;
int B = 6;
int C = 10;

int D = (++A, B+C);
```

In questo esempio il valore assegnato a D e` quello ottenuto dalla somma di B e C, mentre l'espressione a sinistra della virgola serve per incrementare A. A sinistra della virgola poteva esserci una chiamata di funzione, che serviva solo per alcuni suoi effetti collaterali. Quanto alle parentesi, esse sono necessarie perche` l'assegnamento ha la precedenza sulla virgola.
Questo operatore e` comunque sovraccaricato raramente.

205

Autoincremento e autodecremento

Gli operatori ++ e -- meritano un breve accenno poiche` esistono entrambi sia come operatori unari prefissi che unari postfissi.

Le prime versioni del linguaggio non consentivano di distinguere tra le due forme, la stessa definizione veniva utilizzata per le due sintassi. Le nuove versioni del linguaggi consentono invece di distinguere e usano due diverse definizioni per i due possibili casi.

Come operatori globali, la forma prefissa prende un solo argomento, l'oggetto cui e` applicato; la forma postfissa invece possiede un parametro fittizio in piu` di tipo int (c'e' una regola mnemonica: l'operatore postfisso richiede l'uso di una variabile temporanea, corrispondente all'argomento fittizio in piu'). I prototipi delle due forme di entrambi gli operatori per gli interi sono ad esempio le seguenti:

```
int operator++(int A);          // caso ++Var
int operator++(int A, int);     // caso Var++
```

206

Autoincremento e autodecremento

```
int operator--(int A);          // caso --Var
int operator--(int A, int);     // caso Var--
```

Il parametro fittizio non ha un nome e non e` possibile accedere ad esso.

Ridefiniti come funzioni membro, la versione prefissa non presenta nel suo prototipo alcun parametro (il parametro e` l'oggetto su cui l'operatore e` chiamato), la forma postfissa ha un prototipo con il solo argomento fittizio.

207

New e delete

Neanche gli operatori new e delete fanno eccezione, anche loro possono essere ridefiniti sia a livello di classe o addirittura globalmente.

Sia come funzioni globali che come funzioni membro, la new riceve un parametro di tipo size_t che al momento della chiamata e` automaticamente inizializzato con il numero di byte da allocare e deve restituire sempre un void*; la delete invece riceve un void* e non ritorna alcun risultato (va dichiarata void). Anche se non esplicitamente dichiarate, come funzioni membro i due operatori sono sempre static.

Poiche` entrambi gli operatori hanno un prototipo predefinito, non e` possibile avere piu` versioni overloaded di new e delete, e` possibile averne al piu` una unica definizione globale e una sola definizione per classe come funzione membro. Se una classe ridefinisce questi operatori (o uno dei due) la funzione membro viene utilizzata al posto di quella globale per gli oggetti di tale classe; quella globale definita (anch'essa eventualmente ridefinita dall'utente) sara` utilizzata in tutti gli altri casi.

208

New e delete

La ridefinizione di new e delete e` solitamente effettuata in programmi che fanno massiccio uso dello heap al fine di evitarne una eccessiva frammentazione e soprattutto per ridurre l'overhead globale introdotto dalle singole chiamate.

Ecco un esempio di new e delete globali:

```
void* operator new(size_t Size) {
  return malloc(Size);
}

void operator delete(void* Ptr) {
  free(Ptr);
}
```

209

New e delete

Le funzioni malloc() e free() richiedono al sistema (rispettivamente) l'allocazione di un blocco di Size byte o la sua deallocazione (in quest'ultimo caso non e` necessario indicare il numero di byte).

Sia new che delete possono accettare un secondo parametro, nel caso di new ha tipo void* e nel caso della delete e` di tipo size_t: nella new il secondo parametro serve per consentire una allocazione di un blocco di memoria ad un indirizzo specifico (ad esempio per mappare in memoria un dispositivo hardware), mentre nel caso della delete il suo compito e` di fornire la dimensione del blocco da deallocare (utile in parecchi casi). Nel caso in cui lo si utilizzi, e` compito del programmatore supplire un valore per il secondo parametro (in effetti solo per il primo parametro della new e` il compilatore che fornisce il valore).

New e delete

Ecco un esempio di new che utilizza il secondo parametro:

```
void* operator new(size_t Size, void* Ptr = 0) {
  if (Ptr) return Ptr;
  return malloc(Size);
}
int main() {
  // Supponiamo di voler mappare un certo
  // dispositivo hardware tramite una istanza di un apposito tipo
  const void* DeviceAddr = 0xA23;

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

New e delete

```
  /* ... */

  return 0;
}
```

Si noti che non c'e` una delete duale per questa forma di new (perche` una delete non puo` sapere se e come e` stata allocato l'oggetto da deallocare), questo vuol dire che gli oggetti allocati nel modo appena visto (cioe` fornendo alla new un indirizzo) vanno deallocati con tecniche diverse.

E` possibile sovraccaricare anche le versioni per array di questi operatori. I prototipi di new[] e delete[] sono identici a quelli gia` visti in particolare il valore che il compilatore fornisce come primo parametro alla new[] e` ancora la dimensione complessiva del blocco da allocare.

New e delete

Per terminare il discorso su questi operatori, bisogna accennare a cio` che accade quando una allocazione non riesce (generalmente per mancanza di memoria). In caso di fallimento della new, lo standard prevede che venga chiamata una apposita funzione (detta new-handler) il cui comportamento di default e` sollevare una eccezione di tipo std::bad_alloc che bisogna intercettare per gestire il possibile fallimento.

E` possibile modificare tale comportamento definendo e istallando una nuova new-handler. La generica new-handler deve essere una funzione che non riceve alcun parametro e restituisce void, tale funzione va installata tramite una chiamata a std::set_new_handler il cui prototipo e` dato dalle seguenti definizioni:

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

New e delete

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

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

int main(int, char* []) {
```


New e delete

```
  new_handler OldHandler = set_new_handler(NoMemory);

  char* Ptr = new char[1000000000];

  set_new_handler(OldHandler);

  /* ... */
}
```

Il precedente esempio funziona perche` la funzione standard abort() provoca la terminazione del programma, in realta` la new-handler viene richiamata da new finche` l'operatore non e` in grado di restituire un valore valido, per cui bisogna tenere conto di cio` quando si definisce una routine per gestire i fallimenti di new.

Ultime generalità sugli operatori

Per terminare questo argomento restano da citare gli operatori per la conversione di tipo e analizzare la differenza tra operatori come funzioni globali o come funzioni membro.

Solitamente non c'e` differenza tra un operatore definito globalmente e uno analogo definito come funzione membro, nel primo caso per ovvi motivi l'operatore viene solitamente dichiarato friend delle classi cui appartengono i suoi argomenti; nel caso di una funzione membro, il primo argomento e` sempre una istanza della classe e l'operatore puo` accedere a tutti i suoi membri, per quanto riguarda l'evaentuale secondo argomento puo` essere necessaria dichiararlo friend nell'altra classe. Per il resto non ci sono differenze per il compilatore, nessuno dei due metodi e` piu` efficiente dell'altro; tuttavia non sempre e` possibile utilizzare una funzione membro, ad esempio se si vuole permettere il flusso su stream della propria classe , e` necessario ricorrere ad una funzione globale, perche` il primo argomento non e` una istanza della classe (cio' deriva da come sono state definite le classi istream e ostream, e ci sono buone ragioni per averle definite in quella maniera):

Ultime generalità sugli operatori

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

  private:
    float Re, Im;
    friend ostream& operator<<(ostream& os,
                               Complex& C);
};
```

Ultime generalità sugli operatori

```
ostream& operator<<(ostream& os, Complex& C) {
  os << C.Re << " + i" << C.Im;
  return os;
}
```

Adesso e` possibile scrivere:

```
Complex C(1.0, 2.3);

cout << C;
```


Ultime generalità sugli operatori

In alcuni casi e' preferibile definire operatori binari che richiedano la commutativita' come operatori esterni, e non membro. La ragione e' evidente:

```
class Complex {
  public:
    Complex(doble r =0.0, double i=0.0);
    Complex operator -(const complex &o);
  /* ... */
};

Complex Complex::operator +(Complex a, Complex b);
```

Ultime generalità sugli operatori

```
Complex c = 0.5;  // Ok,conversione da chiamata implicita del costr. da  double
Complex e;
e = c +5.0;          // operator +(c,Complex(5.0))
e = 5.0 + c;         // operator +(Complex(5.0),c)
e = c – 5.0;         // c.operator -(c,Complex(5.0))
e = 5.0 – c;         // KO!!!
```

Nell'ultima riga si ha un errore in compilazione, in quanto l'espressione 5.0-c non puo' essere interpretata ne' come

```
operator -(5.0,c)
```

ne' come

```
5.0.operator -(c)
```

220

Programmazione generica

0

Classi contenitore

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

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

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

1

Classi contenitore

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


Classi contenitore

```
    TCell* FirstCell;
};

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

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

Classi contenitore

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

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

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

TList::~TList() {
  TCell* Iterator = FirstCell;
```

4

Classi contenitore

```
  while (Iterator) {
    TCell* Tmp = Iterator;
    Iterator = Iterator -> GetNextCell();
    delete Tmp;
  }
}

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

5

Classi contenitore

L'esempio mostra una parziale implementazione di una tale lista (che assume la proprieta` degli oggetti contenuti), nella realta` TInfo e/o TList molto probabilmente sarebbero diverse al fine di fornire un meccanismo per eseguire delle ricerche all'interno della lista e varie altre funzionalita`. Tuttavia il codice riportato e` sufficiente ai nostri scopi.

Una implementazione di questo tipo funziona, ma soffre di (almeno) un grave difetto: la lista puo` memorizzare tutta una gerarchia di oggetti, e questo e` utile e comodo in molti casi, tuttavia in molte situazioni siamo interessate a liste di oggetti omogenei e una soluzione di questo tipo non permette di verificare a compile time che gli oggetti memorizzati corrispondano tutti ad uno specifico tipo. Potremmo cercare (e trovare) delle soluzioni che ci permettano una verifica a run time, ma non a compile time. Supponete di aver bisogno di una lista per memorizzare figure geometriche e una altra per memorizzare stringhe, nulla vi impedisce di memorizzare una stringa nella lista delle figure geometriche (poiche` le liste memorizzano puntatori alla classe base comune TInfo). Sostanzialmente una lista di questo tipo annulla i vantaggi di un type checking statico.

Alcune osservazioni...

Classi contenitore

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

Tutto questo in C++ e` possibile tramite il meccanismo dei template.

Classi template

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

```
template < class T >
```

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

Classi template

```
template < class TInfo >
class TList {
  public:
    TList();
    ~TList();
    void Store(TInfo& Object);
    /* ... */
  private:
    class TCell {
      public:
        TCell(TInfo& Object, TCell* Next);
        ~TCell();
        TInfo& GetObject();
        TCell* GetNextCell();
      private:
```

Classi template

```
        TInfo& StoredObject;
        TCell* NextCell;
    };

    TCell* FirstCell;
};
```

Al momento l'esempio e` limitato alle sole dichiarazioni, vedremo in seguito come definire i metodi del template.

Intanto, si noti che e` sparita la dichiarazione della classe TInfo, la keyword template dice al compilatore che TInfo rappresenta un nome di tipo qualsiasi (anche un tipo primitivo come int o long double). Le dichiarazioni quindi non fanno piu` riferimento ad un tipo esistente, la` dove e` stato utilizzato il nome fittizio TInfo. Inoltre il contenitore non memorizza piu` tipi puntatore, ma riferimenti alle istanze di tipo.

10

Classi template

Supponendo di aver fornito anche le definizioni dei metodi, vediamo come istanziare la generica lista:

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

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

11

Classi template

```
ListOfStudent.Store(*Pippo);                    // Ok!
```

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

```
NomeTemplate < Tipo >
```

(si noti che Tipo va specificato tra i simboli < >).

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

12

Classi template

La definizione dei metodi di TList avviene nel seguente modo:

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

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

13

Classi template

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

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


Classi template

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

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

Classi template

```
  }
}

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

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

16

Classi template

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

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

17

Classi template

```
    Key KeyArray[Size];
    Value ValueArray[Size];
};

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

Questa soluzione non pretende di essere ottimale, in particolare soffre di un limite: la dimensione dell'array e` prefissata.

Classi template

Fortunatamente un template puo` ricevere come parametri anche valori di un certo tipo:

```
template < class Key, class Value, int Size =10>
class AssocArray {
  public:
    /* ... */
  private:
    Key KeyArray[Size];
    Value ValueArray[Size];
};
```

La keyword typename

Consideriamo il seguente esempio:

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

E` chiaro dall'esempio che l'intenzione era quella di utilizzare un tipo dichiarato all'interno di T per istanziarlo all'interno del template.

La keyword typename

Tale codice puo` sembrare corretto, ma in effetti il compilatore non produrra` il risultato voluto. Il problema e` che il compilatore non puo` sapere in anticipo se T::TId e` un identificatore di tipo o un qualche membro pubblico di T.
Per default il compilatore assume che TId sia un membro pubblico del tipo T e` l'unico modo per ovviare a cio` e` utilizzare la keyword typename introdotta dallo standard:

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

La keyword typename

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

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

    Alias Object
};
```

22

La keyword typename

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

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

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

23

Vincoli impliciti

Un importante aspetto da tenere presente quando si scrivono e si utilizzano template (siano essi classi template o, come vedremo, funzioni) e` che la loro istanziazione possa richiedere che su uno o piu` dei parametri del template sia definita una qualche funzione o operazione. Si consideri che le istanze dovranno presumibilmente essere costruite, copiate, assegnate, distrutte etc..., e che tali operazioni sono espresse da funzioni. Esempio:

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

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


Vincoli impliciti

```
/* Definizione della funzione First() */

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

Vincoli impliciti

la funzione Add tenta un confronto tra due valori di tipo T (parametro del template). La cosa e` perfettamente legale, solo che implicitamente si assume che sul tipo T sia definito operator <; il tentativo di istanziare tale template con un tipo su cui tale operatore non e` definito e` pero` un errore che puo` essere segnalato solo quando il compilatore cerca di creare una istanza del template.

Purtroppo il linguaggio segue la via dei vincoli impliciti, ovvero non fornisce alcun meccanismo per esplicitare assunzioni fatte sui parametri dei template, tale compito e` lasciato ai messaggi di errore del compilatore e alla buona volonta` dei programmatori che dovrebbero opportunamente commentare situazioni di questo genere.

Problemi di questo tipo non ci sarebbero se si ricorresse al polimorfismo, ma il prezzo sarebbe probabilmente maggiore dei vantaggi.

26

Funzioni template

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

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

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

27

Funzioni template

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

  int C = min(A, B);

  TMyClass D(/* ... */);
  TMyClass E(/* ... */);
  TMyClass F = min(D, E);

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


Funzioni template

Ogni qual volta il compilatore trova una chiamata alla funzione min istanzia (se non era gia stato fatto prima) la funzione template (nel caso delle funzioni l'istanziazione e` un processo totalmente automatico che avviene quando il compilatore incontra una chiamata alla funzione template producendo una nuova funzione ed effettuando una chiamata a tale istanza). In sostanza con un template possiamo avere tutte le versioni overloaded della funzione min che ci servono con un'unica definizione.

Template ed ereditarieta`

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

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

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

Template ed ereditarieta`

Un altro modo di combinare ereditarieta` e template e` dato dal seguente esempio:

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

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

Base<int> *Aptr = new Derived<int>        //OK
Base<int> *Bptr = new Derived<double> //Errore
```

Template ed ereditarieta`

in questo caso l'ereditarieta` e` stata utilizzata per estendere le caratteristiche della classe template Base. Tra le istanze dei template sussiste una relazione di discendenza per cui un puntatore a Base < T > può puntare a Derived < T >.

32

Templates vs ereditarietà

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

33

Gestione degli errori

0

Le eccezioni

Durante l'esecuzione di un applicativo possono verificarsi situazioni di errore, non verificabili a compile-time, che in qualche modo vanno gestite.

Le possibili tipologie di errori sono diverse ed in generale non tutte trattabili allo stesso modo. In particolare possiamo distinguere tra errori che non compromettono il funzionamento del programma ed errori che invece costituiscono una grave impedimento al normale svolgimento delle operazioni.

Tipico della prima categoria sono ad esempio gli errori dovuti a errato input dell'utente, facili da gestire senza grossi problemi. Meno facili da catturare e gestire e` invece la seconda categoria cui possiamo inserire ad esempio i fallimenti relativi all'acquisizione di risorse come la memoria dinamica; questo genere di errori viene solitamente indicato con il termine di eccezioni per sottolineare la loro caratteristica di essere eventi particolarmente rari e di comportare il fallimento di tutta una sequenza di operazioni.

1

Le eccezioni

La principale difficolta' connessa al trattamento delle eccezioni consiste nel fatto che in generale la parte di codice in cui l'eccezione puo' essere generata non coincide con la parte di codice in cui essa puo' essere gestita. Ad esempio, se in un metodo di una classe viene passato un argomento che fornisce la dimensione per un array da allocare con new, puo' succedere che l'allocazione dinamica fallisca. Il metodo che chiama new puo' individuare l'errore nel fatto che non c'e' abbastanza memoria, ma non puo' sapere come porvi rimedio: deve terminare il programma? Stampare un messaggio di errore? Riprovare ad allocare meno memoria?

In realta', solo la parte di codice che ha chiamato quel metodo con quel particolare argomento, o una porzione di codice ancora piu' a monte, conosce il modo corretto di gestire quell'errore.

2

Le eccezioni

La principale difficolta` connessa al trattamento delle eccezioni e` quindi quella di riportare lo stato dell'applicazione ad un valore consistente. Il verificarsi di un tale evento comporta infatti (in linea di principio) l'interruzione di tutta una sequenza di operazioni rivolta ad assolvere ad una certa funzionalita`, allo svuotamento dello stack ed alla deallocazione di eventuali risorse allocate fino a quel punto relativamente alla richiesta in esecuzione. Le informazioni necessarie allo svolgimento di queste operazioni sono in generale dipendenti anche dal momento e dal punto in cui si verifica l'eccezione e non e` quindi immaginabile (o comunque facile) che la gestione dell'errore possa essere incapsulata in un unico blocco di codice richiamabile indipendentemente dal contesto in cui si verifica il problema.

In linguaggi che non offrono alcun supporto, catturare e gestire questi errori puo` essere particolarmente costoso e difficile, al punto che spesso si rinuncia lasciando sostanzialmente al caso le conseguenze.

3

Le eccezioni

Il primo problema che bisogna affrontare quando si verifica un errore e` capire dove e quando bisogna gestire l'anomalia.

Poniamo il caso che si stia sviluppando una serie di funzioni matematiche, in particolare una che esegue la radice quadrata. Come comportarsi se l'argomento della funzione e` un numero negativo? Le possibilita` sono due:

Terminare il processo;

Segnalare l'errore al chiamante.

Probabilmente la prima possibilita` e` eccessivamente drastica, tanto piu` che non sappiamo a priori se e` il caso di terminare oppure se il chiamante possa prevedere azioni alternative da compiere in caso di errore (ad esempio ignorare l'operazione e passare alla successiva, oppure segnalare l'errore all'utente dell'applicazione). D'altronde neanche la seconda possibilita` sarebbe di per se` una buona soluzione, cosa succede se il chiamante ignora l'errore proseguendo come se nulla fosse?

E` chiaramente necessario un meccanismo che garantisca che nel caso il chiamante non catturi l'anomalia qualcuno intervenga in qualche modo.

Le eccezioni

Ma andiamo con ordine, e supponiamo che il chiamante preveda del codice per gestire l'anomalia.

Se al verificarsi di un errore grave non si dispone delle informazioni necessarie per decidere cosa fare, la cosa migliore da farsi e` segnalare la condizione di errore a colui che ha invocato l'operazione. Questo obiettivo viene raggiunto con la keyword throw:

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

Le eccezioni

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

La throw accetta un argomento come parametro che viene utilizzato per creare un oggetto che non ubbidisce alle normali regole di scope e che viene restituito a chi ha tentato l'esecuzione dell'operazione (nel nostro caso al blocco in cui Divide e` stata chiamata). Il compito di questo oggetto e` trasportare tutte le informazioni utili sull'evento.

L'argomento di throw puo` essere sia un tipo predefinito che un tipo definito dal programmatore.

Per compatibilita` con il vecchio codice, una funzione non e` tenuta a segnalare la possibilita` che possa lanciare una eccezione, ma e` buona norma avvisare dell'eventualita` segnalando quali tipologie di eccezioni sono possibili. Allo scopo, nel prototipo si usa ancora throw seguita da una coppia di parentesi tonde contenente la lista dei tipi di eccezione che possono essere sollevate:

Le eccezioni

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

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

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

Gestire le eccezioni

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

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

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

8

Gestire le eccezioni

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

9

Gestire le eccezioni

Utilizzando try e racchiudendo tra parentesi graffe (le parentesi si devono utilizzate sempre) il codice che puo` generare una eccezione si segnala al compilatore che siamo pronti a gestire l'eventuale eccezione.

Ci si potra` chiedere per quale motivo sia necessario informare il compilatore dell'intenzione di catturare e gestire l'eccezione, il motivo sara` chiaro in seguito, al momento e` sufficiente sapere che cio` ha il compito di indicare quando certi automatismi dovranno arrestarsi e lasciare il controllo a codice ad hoc preposto alle azioni del caso.

Il codice in questione dovra` essere racchiuso all'interno di un blocco catch che deve seguire il blocco try:

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

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

10

Gestire le eccezioni

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

11

Gestire le eccezioni

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

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

12

Gestire le eccezioni

```
catch(Type2 Id2) {
   /* ... */
  }

  /* Altre catch */

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

  /* Altro */
```

Ciascuna catch e` detta exception handler e riceve un parametro che e` il tipo di eccezione che viene gestito in quel blocco. Nel caso generale un blocco try sara` seguito da piu` blocchi catch, uno per ogni tipo di eccezione possibile all'interno di quel try. Si noti che le catch devono seguire immediatamente il blocco try.

13

Gestire le eccezioni

Quando viene generata una eccezione (throw) il controllo risale indietro fino al primo blocco try. Gli oggetti staticamente allocati (che cioe` sono memorizzati sullo stack) fino a quel momento nei blocchi da cui si esce vengono distrutti invocando il loro distruttore (se esiste). Nel momento in cui risalendo all'indietro si giunge ad un blocco try, anche gli oggetti staticamente allocati fino a quel momento dentro il blocco try vengono distrutti ed il controllo passa immediatamente dopo la fine del blocco.

Il tipo dell'oggetto creato con throw viene quindi confrontato con i parametri delle catch che seguono la try. Se viene trovata una catch del tipo corretto, si passa ad eseguire le istruzioni contenute in quel blocco, dopo aver inizializzato il parametro della catch con l'oggetto restituito con throw. Nel momento in cui si entra in un blocco catch, l'eccezione viene considerata gestita ed alla fine del blocco catch il controllo passa alla prima istruzione che segue la lista di catch (sopra indicato con "/* Altro */").

14

Gestire le eccezioni

Vediamo un esempio:

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

class Test {
   char Name[20];
 public:
   Test(char* name){
     Name[0] = '\0';
     strcpy(Name, name);
```

15

Gestire le eccezioni

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

16

Gestire le eccezioni

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

int main() {
```

17

Gestire le eccezioni

```
  try {
    Test g("try");
    int c = Div(10, 2);
    cout << "c = " << c << endl;
  }       // Il controllo ritorna qua
  catch(int exc) {
    cout << "exception catched" << endl;
    cout << "exception value is " << exc << endl;
  }
  return 0;
}
```

18

Gestire le eccezioni

La chiamata a Div all'interno della main provoca una eccezione nella Sub, viene quindi distrutto l'oggetto k ed il puntatore k2, ma non l'oggetto puntato (allocato dinamicamente). La deallocazione di oggetti allocati nello heap e` a carico del programmatore.

In seguito alla eccezione, il controllo risale a Div, ma la chiamata a Sub non era racchiusa dentro un blocco try e quindi anche Div viene terminata distruggendo l'oggetto h. L'oggetto h2 non e` stato ancora creato e quindi nessun distruttore per esso viene invocato.

Il controllo e` ora giunto al blocco che ha chiamato la Div, essendo questo un blocco try, vengono distrutti gli oggetti g e c ed il controllo passa nel punto in cui si trova il commento.

A questo punto viene eseguita la catch poiche` il tipo dell'eccezione e` lo stesso del suo argomento e quindi il controllo passa alla return della main.

Ecco l'output del programma:

```
Test constructor inside try
Div called
Test constructor inside Div
```

19

Gestire le eccezioni

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

20

Gestire le eccezioni

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

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

Nei casi in cui non interessa distinguere tra piu` tipologie di eccezioni, e` possibile utilizzare un unico blocco catch utilizzando le ellissi:

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

21

Gestire le eccezioni

In altri casi invece potrebbe essere necessario passare l'eccezione ad un blocco try ancora piu` esterno, ad esempio perche` a quel livello e` sufficiente (o possibile) fare solo certe operazioni, in questo caso basta utilizzare throw all'interno del blocco catch per reinnescare il meccanismo delle eccezioni a partire da quel punto:

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

In questo modo si puo` portare a conoscenza dei blocchi piu` esterni della condizione di errore.

Nota bene: in questo caso l'uso di throw senza argomenti serve per lanciare la stessa eccezione ricevuta, ma potrebbe benissimo essere lanciata una eccezione diversa relativa alla parte di errore ancora da gestire.

Casi particolari

Esistono ancora due problemi da affrontare

Cosa fare se una funzione solleva una eccezione non specificata tra quelle possibili;

Cosa fare se non si trova un blocco try seguito da una catch compatibile con quel tipo di eccezione;

Esaminiamo il primo punto.

Per default una funzione che non specifica una lista di possibili tipi di eccezione puo` sollevare una eccezione di qualsiasi tipo. Una funzione che specifica una lista dei possibili tipi di eccezione e` teoricamente tenuta a rispettare tale lista, ma nel caso non lo facesse, in seguito ad una throw di tipo non previsto, verrebbe eseguita immediatamente la funzione predefinita unexpected(). Per default unexpected() chiama terminate() provocando la terminazione del programma. Tuttavia e` possibile alterare tale comportamento definendo una funzione che non riceve alcun parametro e restituisce void ed utilizzando set_unexpected() come mostrato nel seguente esempio:

Casi particolari

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

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

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

24

Casi particolari

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

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

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

typedef void (* OldHandlerPtr) ();
```

25

Casi particolari

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

Il modo in cui terminate l'esecuzione non e` importante, quello che conta e` che la funzione non ritorni.

set_unexpected() infine restituisce l'indirizzo della unexpected precedentemente installata e che in talune occasioni potrebbe servire.

Casi particolari

Rimane da trattare il caso in cui in seguito ad una eccezione, risalendo i blocchi applicativi, non si riesca a trovare un blocco try oppure una catch compatibile con il tipo di eccezione sollevata.

Nel caso si trovi un blocco try ma nessuna catch idonea, il processo viene iterato fino a quando una catch adatta viene trovata, oppure non si riesce a trovare alcun altro blocco try. Se nessun blocco try viene trovato, viene chiamata la funzione terminate().

Anche in questo caso, come per unexpected(), terminate() e` implementata tramite puntatore ed e` possibile alterarne il funzionamento utilizzando set_terminate() in modo analogo a quanto visto per unexpected() e set_unexpected() (ovviamente la nuova terminate non deve ritornare).

set_terminate() restituisce l'indirizzo della terminate() precedentemente installata.

Eccezioni e costruttori

Il meccanismo di stack unwinding (srotolamento dello stack) che si innesca quando viene sollevata una eccezione garantisce che gli oggetti allocati sullo stack vengano distrutti via via che il controllo esce dai vari blocchi applicativi.

Ma cosa succede se l'eccezione viene sollevata nel corso dell'esecuzione di un costruttore? In tal caso l'oggetto non puo` essere considerato completamente costruito ed il compilatore non esegue la chiamata al suo distruttore, viene comunque eseguita la chiamata dei distruttori per le componenti dell'oggetto che sono state create:

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

class Component {
  public:
```

28

Eccezioni e costruttori

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

29

Eccezioni e costruttori

```
class Composed {
   private:
     Component A;

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

30

Eccezioni e costruttori

```
     }
};

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

31

Eccezioni e costruttori

Dall'output di questo programma:

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

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

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

Eccezioni e costruttori

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

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

Eccezioni e costruttori

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

class Composed {
```


Eccezioni e costruttori

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

Eccezioni e costruttori

```
catch(int) {
        cout << "Exception in Composed constructor...";
        cout << endl << "Cleaning up..." << endl;
        delete[] FloatArray;
        cout << "Rethrowing exception..." << endl;
        cout << endl;
        throw;
      }
    }
    ~Composed() {
      cout << "Composed distructor called..." << endl;
      delete[] FloatArray;
```


Eccezioni e costruttori

```
    }
};

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

Eccezioni e costruttori

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

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

Divide: throwing an exception...
```

38

Eccezioni e costruttori

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

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

Si noti che se la catch del costruttore della classe Composed non avesse rilanciato l'eccezione, il compilatore considerando gestita l'eccezione, avrebbe terminato l'esecuzione del costruttore considerando B completamente costruito. Cio` avrebbe comportato la chiamata del distruttore al termine dell'esecuzione della main con il conseguente errore dovuto al tentativo di rilasciare nuovamente la memoria allocata per FloatArray.

39

Eccezioni e costruttori

Per verificare cio` si modifichi il programma nel seguente modo:

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

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


Eccezioni e costruttori

```
}

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

Eccezioni e costruttori

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

42

Eccezioni e costruttori

```
      catch(int) {
        cout << "Exception in Composed constructor...";
        cout << endl << "Cleaning up..." << endl;
        delete[] FloatArray;
      }
    }
    ~Composed() {
      cout << "Composed distructor called..." << endl;
    }
};
```

43

Eccezioni e costruttori

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

eseguendolo otterrete il seguente output:

Eccezioni e costruttori

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

Divide: throwing an exception...

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

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

Eccezioni e costruttori

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

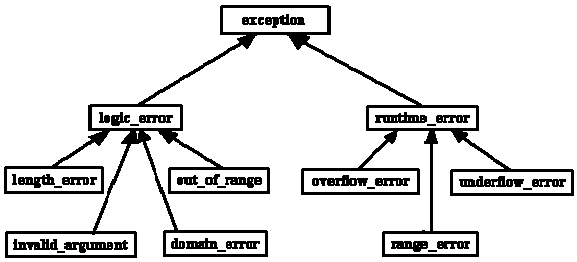
La realizzazione di un costruttore nella cui esecuzione puo` verificarsi una eccezione, e` dunque un compito non banale e in generale sono richieste due operazioni:

Eseguire eventuali pulizie all'interno del costruttore se non si e` in grado di terminare correttamente la costruzione dell'oggetto;
Se il distruttore non termina correttamente (ovvero l'oggetto non viene totalmente costruito), propagare una eccezione anche al codice che ha invocato il costruttore e che altrimenti rischierebbe di utilizzare un oggetto non correttamente creato.

La gerarchia exception

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

La gerarchia exception



48

La gerarchia exception

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

Dallo standard:

logic_error ha lo scopo di segnalare un errore che presumibilmente potrebbe essere rilevato prima di eseguire il programma stesso, come la violazione di una precondizione;

domain_error va lanciata per segnalare errori relativi alla violazione del dominio;

invalid_argument va utilizzato per segnalare il passaggio di un argomento non valido;

length_error segnala il tentativo di costruire un oggetto di dimensioni superiori a quelle permesse;

out_of_range riporta un errore di argomento con valore non appartenente all'intervallo di definizione;

runtime_error rappresenta un errore che puo` essere rilevato solo a runtime;

49

La gerarchia exception

range_error segnala un errore di intervallo durante una computazione interna;
overflow_error riporta un errore di overflow;
underflow_error segnala una condizione di underflow;
Eccetto che per la classe base che non e` stata pensata per essere impiegata direttamente, il costruttore di tutte le altre classi riceve come unico argomento un const string&; il tipo string e` definito nella libreria standard del linguaggio.

50

riflessioni sulle eccezioni

I meccanismi che sono stati descritti nei paragrafi precedenti costituiscono un potente mezzo per affrontare e risolvere situazioni altrimenti difficilmente trattabili. Non si tratta comunque di uno strumento facile da capire e utilizzare ed e` raccomandabile fare diverse prove ed esercizi per comprendere cio` che sta dietro le quinte. La principale difficolta` e` quella di riconoscere i contesti in cui bisogna utilizzare le eccezioni ed ovviamente la strategia da seguire per gestire tali eventi. Cio` che bisogna tener presente e` che il meccanismo delle eccezioni e` sostanzialmente non locale poiche` il controllo ritorna indietro risalendo i vari blocchi applicativi. Cio` significa che bisogna pensare ad una strategia globale, ma che non tenti di raggruppare tutte le situazioni in un unico errore generico altrimenti si verrebbe schiacciati dalla complessita` del compito.
In generale non e` concepibile occuparsi di una possibile eccezione al livello di ogni singola funzione, a questo livello cio` che e` pensabile fare e` solo lanciare una eccezione; e` invece bene cercare di rendere i propri applicativi molto modulari e isolare e risolvere all'interno di ciascun blocco quante piu` situazioni di errore possibile, lasciando filtrare una eccezione ai livelli superiori solo se le conseguenze possono avere ripercussioni a quei livelli.
Ricordate infine di catturare e trattare le eccezioni standard che si celano dietro ai costrutti predefiniti quali l'operatore globale ::new.

51

Libreria standard

0

La libreria standard

Fornisce:

Supporto per le caratteristiche del linguaggio, come la gestione della memoria e runtime type information (RTTI).

Informazioni sugli aspetti del linguaggio definiti dall'implementazione, come il più grande valore float.

Funzioni che non possono essere implementate in modo ottimale per ogni sistema nel linguaggio stesso, come sqrt () e memmove().

Funzionalità non primitive che un programmatore può utilizzare ai fini della portabilità, come liste, mappe, funzioni di ordinamento (sort) e I/O streams.

Un'infrastruttura per estendere le funzionalità che sono disponibili, come convenzioni e funzionalità di supporto che permettono a un utente di fornire I/O per un tipo definito dall'utente nello stile degli I/O per i tipi nativi.

La base comune per altre librerie.

La libreria standard

Le funzionalità della libreria standard sono definite nel namespace std e presentate come un insieme di headers. Gli headers identificano le parti principali della libreria. Quindi, un loro elenco rappresenta una panoramica della libreria e fornisce una guida per la descrizione della libreria.

Un header standard con un nome che inizia con la lettera c è equivalente a un header della libreria standard del C. Per ogni header <c X > che definisce un nome nel namespace std, c'è un header <X .h > che definisce gli stessi nomi nel namespace globale.

La libreria standard

Contenitori:

<vector > array unidimensionale di T

< list > lista bidirezionale di T

< deque > coda bidirezionale 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 and multiset si trovano rispettivamente in <map> and <set>, mentre priority_queue è dichiarata in <queue>.

La libreria standard

Utililità generali

< utility > operatori e *pairs*

< functional > oggetti funzione

< memory > allocatori per I contenitori

< ctime > data e tempo in stile C

L'header <memory> contiene inoltre il template *auto_ptr* usato principalmente per gestire più comodamente i puntatori nell'ambito delle eccezioni.

La libreria standard

Iterarori

< iterator> iteratori e realtivo supporto.

Gli ieratori forniscono il meccanismo per generalizzare gli algoritmi standard rispetto ai contenitori standard e agli altri tipi.

Algoritmi

< algorithm > algoritmi generali

< cstdlib> bsearch(), qsort()

Tipicamente un algoritmo generale può essere applicato a qualsiasi sequenza di qualsiasi tipo di elementi.

Le funzioni di libreria standard del C bsearch() and qsort () si applicano agli arrays nativi con elementi di tipi che non prevedono costruttori di copia ma solo distruttori definiti dall'utente.

La libreria standard

Diagnostica
< exception > classe exception
< stdexcept > eccezioni standard
< cassert > macro assert
< cerrno > gestione degli errori in stile C
Le asserzioni si basano sulle eccezioni.

La libreria standard

Stringhe
< string > stringa di T
< cctype > classificazione dei caratteri
< cwtype > classificazione estesa dei caratteri
< cstring > funzioni stringa in stile C
< cwchar > funzioni stringa per il set di caratteri esteso
< cstdlib > funzioni stringa in stile C
L'header <cstring> dichiara la famiglia di funzioni strlen(), strcpy(), ecc. .
<cstdlib> dichiara atof() e atoi() che convertono le stringhe C valori numerici.

La libreria standard

Input/Output

< iosfwd > dichiarazioni forward delle funzionalità di I/O

< iostream > oggetti standard iostream e operazioni

< ios > classi base di iostream

< streambuf > stream buffers

< istream > input stream template

< ostream > output stream template

< iomanip > manipolatori

< sstream > conversioni da/a streams/stringhe

< cstdlib > funzioni per la classificazione dei caratteri

< fstream > streams da/a files

< cstdio > printf() e simili

< cwchar> I/O per il set di caratteri esteso

La libreria standard

I manipolatori sono oggetti atti a manipolare lo stato di uno stream (es., cambiare il formato in virgola mobile dell'output) applicandoli allo stream stesso.

Localizzazione

< locale > per rappresentare le differenze culturali

< clocale> rappresenta le differenze in stile C

Un locale definisce le differenze come il formato di output delle date, i simboli usati per la moneta corrente, e i criteri di composizione delle stringhe che variano tra i diversi linguaggi naturali e culture.

La libreria standard

Supporto al linguaggio
< limits > limiti numerici
< climits > macro per i limiti degli scalari in stile C
< cfloat > macro per i limiti dei floating point in stile C
< new > gestione della memoria dinamica
< typeinfo > supporto al runtime type identification
< exception > support alla gestione delle eccezioni
< cstddef > supporto al C
< cstdarg > funzioni ad argomenti variabili
< csetjmp > stack unwinding in stile C
< cstdlib > terminazione del programma
< ctime > orologio di sistema
< csignal> gestione dei segnali in stile C

La libreria standard

L'header <cstddef > definisce il tipo dei valori restituiti da sizeof(), size_t, il tipo del risultato della differenza tra puntatori, ptrdiff_t, e la bistrattata macro NULL.

Numerica
< complex > numrica complessi e operazioni
< valarray > vettori numerica e operazioni
< numeric > operazioni numeriche generalizzate
< cmath > funzioni matematiche standard
< cstdlib > numeri casuali in C

Standard Template Library

La Standard Template Library (STL) è una libreria inclusa nella libreria standard del C++. Fornisce contenitori, iteratori ed algoritmi. Più specificamente, la libreria standard del C++ si basa sulla STL pubblicata da SGI (Silicon Graphics Incorporated). Entrambe includono qualche funzionalità non presente nell'altra. La STL di SGI è rigidamente defiinita nel suo insieme di headers, mentre l'ISO C++ non specifica il contenuto degli headers, lasciando la possibilità di fornire l'implementazione a livello di headers o di libreria vera e propria.

I contenitori standard non derivano da una base comune ma ogni contenitore implementa l'intera interfaccia. Analogamente non c'è una classe comune per gli iteratori. Non è previsto controllo di tipo implicito o esplicito a tempo di esecuzione nell'uso dei contenitori e degli iteratori standard.

L'importante e difficile compito di fornire servizi comuni a tutti i contenitori è svolto attraverso gli ''allocatori'' passati come argomenti dei templates invece che per mezzo di una base comune.

Standard Template Library

I singoli contenitori sono semplici ed efficienti (non proprio semplici come potrebbero veramente essere contenitori indipendenti ma almeno efficienti).

Ogni contenitore fornisce un insieme di operazioni standard con nomi e semantiche standard. Ulteriori operazioni sono fornite per particolari tipi di contenitori a seconda delle esigenze.

Inoltre ci sono classi wrapper che possono essere usate per inserire in un framework comune contenitori sviluppati indipendentemente.

Le modalità d'uso sono uniformate atrraverso gli iteratori piuttosto che con un tipo generale di contenitore.

Standard Template Library

Diversi iteratori possono essere implementati per svolgere diversi compiti sullo stesso contenitore.

Per default i contenitori sono sicuri rispetto al tipo ed omogenei (cioè gli elementi sono tutti dello stesso tipo, quello giusto). E' possibile avere un contenitore eterogeneo come contenitore omogeneo di puntatori ad una base comune.

I contenitori non sono intrusivi nel senso che un oggetto non deve avere una particolare classe base o un particolare attributo. Si adattano bene ai tipi primitivi e a qualsivoglia struttura.

Standard Template Library

Ogni accesso attraverso l'iteratore incorre nell'overhead della chiamata a un metodo virtuale, che può essere significativo rispetto all'esecuzione du una funzione inline.

La gerarchia di iteratori tende ad essere complicata.

Non c'è niente in comune tra tutti i contenitori e niente tra tutti gli oggetti dei vari contenitori. Questo complica la realizzazione di servizi universali come la persistenza e I/O.

Iteratori

Gli Iteratori vengono usati per accedere ai membri di una classe contenitore in maniera analoga ai puntatori. Ad esempio è possibile usare un iteratore per scorrere gli elementi di un vettore.

Ci sono diversi tipi di iteratori:

input_iterator: legge valori muovendosi in avanti. Può essere incrementato, confrontato e dereferenziato.

output_iterator: scrive valori muovendosi in avanti. Può essere incrementato e dereferenziato.

forward_iterator: legge e scrive valori muovendosi in avanti. Combina le funzionalità degli ieratori di input e output con la capacità di memorizzare il valore degli iteratori.

Iteratori

bidirectional_iterator: legge e scrive valori muovendosi avanti e indietro. Funziona come un forward iterator, ma può essere incrementato e decrementato.

random_iterator: legge e scrive ad accesso casuale. E' l'iteratore più potente che combina la capacità di un iteratore bidirezionale con il supporto all'aritmetica dei puntatori e al confronto tra puntatori.

reverse_iterator: Un random iterator o un bidirectional iterator che si muove all'indietro.

Ogni classe contenitore è associata a un tipo di iteratore e ogni algoritmo STL usa un certo tipo di iteratore. Ad esempio i vettori sono associati ad iteratori ad accesso casuale e quindi possono essere oggetto di algoritmi che richiedono l'accesso casuale. Dato che gli iteratori ad accesso casuale contemplano tutte le caratteristiche degli altri iteratori, i vettori possono usare anche gli algoritmi disegnati per gli altri iteratori.

Algoritmi e oggetti funzione

Un contenitore di per sè non è molto utile. Ciò che serve sono le operazioni base come l'individuazione della dimensione, l'iterazione, la copia, l'ordinamento e la ricerca degli elementi.

Fortunatamente la libreria standard fornisce gli algoritmi per svolgere i compiti base e più comuni sui contenitori.

Gli oggetti funzione cotituiscono un meccanismo per personalizzare il comportamento degli algoritmi standard. Forniscono agli algoritmi le informazioni chiave per operare sui dati dell'utente.

Algoritmi

Nonostante siano tanti, circa 60, e possano apparire complicati, gli algoritmi della STL condividono comportamenti di base e interfacce comuni che ne facilitano la comprensione.

Ogni algoritmo è espresso come una funzione template o come un insieme di funzioni template. In questo modo un algortmo può operare su diverse specie di sequenze contenenti elementi di vario tipo.

Gli algoritmi che restituiscono un iteratore come risultato in genere usano la fine di una sequenza di input per indicare una failure.

Gli algoritmi non controllano il range dei loro inputs o outputs. Gli errori di range vanno prevenuti in altro modo. Quando un algoritmo restituisce un iteratore, questo è dello stesso tipo di uno di quelli in input. In particolare, an algorithm's arguments control whether it returns a const_iterator or a nonconstiterator.

Gli algoritmi coprono la maggior parte delle operazioni sui contenitori, come lo scansione, l'ordinamento, la ricerca, l'inserimento e la rimozione degli elementi.

Algoritmi

Gli algoritmi standard sono definiti nel namespace std e le loro dichiarazioni sono disponibili in <algorithm>.

Molti degli algoritmi più comuni sono così semplici che le corrispondenti template functions sono inline. In questo modo i loops contenuti negli algoritmi traggono beneficio dalle optimizzazioni più aggressive sulle funzioni.

Algoritmi

Le operazioni che non modificano le sequenze vengono usate per estrarre informazioni da una sequenza o per trovare le posizioni degli elementi in una sequenza:

Nonmodifying Sequence Operations:

for_each Do operation for each element in a sequence.

find Find first occurrence of a value in a sequence.

find_if Find first match of a predicate in a sequence.

find_first_of Find a value from one sequence in another.

adjacent_find Find an adjacent pair of values.

count Count occurrences of a value in a sequence.

count_if Count matches of a predicate in a sequence.

mismatch Find the first elements for which two sequences differ.

equal True if the elements of two sequences are pairwise equal.

search Find the first occurrence of a sequence as a subsequence.

Algoritmi

find_end Find the last occurrence of a sequence as a subsequence.
search_ n Find the nth occurrence of a value in a sequence.

Molti algoritmi permettono all'utente di specificare l'azione da effetuare sugli elementi. Questo rende gli algoritmi molto più generali e più utili di quanto possano aparire di primo acchito. In particolare un utente può fornire i criteri di confronto. Quando possibile l'azione più comune ed utile è fornita di default.

Algoritmi

Modifying Sequence Operations
transform Apply an operation to every element in a sequence.
copy Copy a sequence starting with its first element.
copy_ backward Copy a sequence starting with its last element.
swap Swap two elements.
iter_swap Swap two elements pointed to by iterators.
swap_ranges Swap elements of two sequences.
replace Replace elements with a given value.

Algoritmi

replace_if Replace elements matching a predicate.
replace_copy Copy sequence replacing elements with a given value.
replace_copy_if Copy sequence replacing elements matching a predicate.
fill Replace every element with a given value.
fill_n Replace first n elements with a given value.
generate Replace every element with the result of an operation.
generate_n Replace first n elements with the result of an operation.
remove Remove elements with a given value.
remove_if Remove elements matching a predicate.

Algoritmi

remove_copy Copy a sequence removing elements with a given value.
remove_copy_i f Copy a sequence removing elements matching a predicate.
unique Remove equal adjacent elements.
unique_copy Copy a sequence removing equal adjacent elements.
reverse Reverse the order of elements.
reverse_copy Copy a sequence into reverse order.
rotate Rotate elements.
rotate_copy Copy a sequence into a rotated sequence.
random_shuffle Move elements into a uniform distribution.

Sequenze ordinate:
sort Sort with good average efficiency.
stable_sort Sort maintaining order of equal elements.
partial_sort Get the first part of sequence into order.

Algoritmi

partial_sort_copy Copy getting the first part of output into order.
nth_element Put the nth element in its proper place.
lower_bound Find the first occurrence of a value.
upper_bound Find the first element larger than a value.
equal_range Find a subsequence with a given value.
binary_search Is a given value in a sorted sequence?
Merge Merge two sorted sequences.
inplace_merge Merge two consecutive sorted subsequences.
partition Place elements matching a predicate first.
stable_partition Place elements matching a predicate first,
preserving relative order.

Algoritmi

Sets:
includes True if a sequence is a subsequence of another.
set_union Construct a sorted union.
set_intersection Construct a sorted intersection.
set_difference Construct a sorted sequence of elements
in the first but not the second sequence.
set_symmetric_difference Construct a sorted sequence of elements
in one but not both sequences.

Algoritmi

Le operazioni di tipo heap mantengono una sequenza in uno stato tale da renderne facile l'ordinamento:

make_heap Make sequence ready to be used as a heap.

push_heap Add element to heap.

pop_heap Remove element from heap.

sort_heap Sort the heap.

Algoritmi

La libreria fornisce pochi algoritmi per selezionare elementi in base a criteri di confronto:

min Smaller of two values.

max Larger of two values.

min_element Smallest value in sequence.

max_element Largest value in sequence.

lexicographical_compare() Lexicographically first of two sequences.

Algoritmi

Infine la libreria fornisce metodi per permutare le sequenze:

next_permutation Next permutation in lexicographical order.

prev_permutation() Previous permutation in lexicographical order.

Inoltre sono disponibili alcuni e generali algoritmi numerici in <numeric>.

Nella descrizione degli algoritmi i nomo dei parametri sono significativi:

In, Out, For, Bi, Ran = input, output, forward, bidirectional, random access iterator

Pred = unary predicate

BinPred = binary predicate

Cmp = comparison function

Op = unary operation

BinOp = binary operation

T = tipo contenuto in sequenza

Oggetti funzione

Gli oggetti funzione standard appartengono al namespace std ma le loro dichiarazioni sono in < functional >. Sono disegnati per essere messi inline facilmente. È infatti più semplice mettere inline l'operatore di una classe piuttosto che una funzione passata attraverso un puntatore.

Gli oggetti funzione sono quindi utilizzati per implementare o predicati logici, operazioni aritmetiche , sfruttando talvolta la capacità di conservare dati tra successive invocazioni.

```
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
};
```

La classe vector

I vettori contengono elementi contigui memorizzati come in un array. L'accesso agli elementi o l'aggiunta di elementi in coda richiede un tempo costante, mentre la ricerca di un valore o l'inserimento di elementi in una data posizione richiede un tempo lineare.

Costruttori:

```
vector();
vector( const vector& c );
vector( size_type num, const TYPE& val = TYPE() );
vector( input_iterator start, input_iterator end );
~vector();
```

Il costruttore di default non prevede argomenti e crea semplicemente un'istanza del vettore.

La classe vector

Il secondo costruttore è il costruttore di copia di default e può essere usato per creare un nuovo vettore come copia del vettore *c*.

Il terzo costruttore crea un vettore dimensionato per *num* oggetti, ciascuno dei quali assume il valore *val* (se presente). Per esempio, il seguente codice crea un vettore contenete 5 copie dell'intero 42:

```
vector<int> v1( 5, 42 );
```

L'ultimo costruttore crea un vettore inizializzato per contenere gli elemnti compresi tra *start* ed *end*.

La classe vector

Tutti I contenitori C++ possono essere confrontati e assegnati con gli operatori standard: ==, !=, <=, >=, <, >, and =. I singoli elementi di un vettore possono essere esaminati con l'operatore [].

Il confronto e l'assegnamento tra vettori richiede un tempo lineare. L'operatore [] impiega un tempo costante.

Due vettori sono uguali se la loro dimensione coincide, e ogni elemento in una data posizione in un vettore è uguale a quello nella stessa posizione dell'altro vettore.

Il confronto tra vettori è di tipo lessicografico.

Per esempio, il seguente codice usa l'operatore [] per accedere a tutti gli elementi del vettore:

```
vector<int> v( 5, 1 );
for( int i = 0; i < v.size(); i++ ) {
  cout << "Element " << i << " is " << v[i] << endl;
}
```

La classe vector

Metodi:

assign	assign elements to a vector
at	returns an element at a specific location
back	returns a reference to last element of a vector
begin	returns an iterator to the beginning of the vector
capacity	returns the number of elements that the vector can hold
clear	removes all elements from the vector
empty	true if the vector has no elements
end	returns an iterator just past the last element of a vector
erase	removes elements from a vector
front	returns a reference to the first element of a vector

La classe vector

insert	inserts elements into the vector
max_size	numero massimo di elementi che si possono memorizzare
pop_back	removes the last element of a vector
push_back	add an element to the end of the vector
rbegin	returns a reverse_iterator to the end of the vector
rend	returns a reverse_iterator to the beginning of the vector
reserve	sets the minimum capacity of the vector
resize	change the size of the vector
size	returns the number of items in the vector
swap	swap the contents of this vector with another

La classe vector: esempio

```
vector<string> words;
string str;

while( cin >> str ) words.push_back(str);

vector<string>::iterator iter;
for( iter = words.begin(); iter != words.end(); iter++ ) {
  cout << *iter << endl;
}
```

La classe vector: esempio

When given this input:

```
hey mickey you're so fine
```

...the above code produces the following output:

```
hey
mickey
you're
so
fine
```

La classe list

Le liste sono sequenze di elementi collegati tra loro. Rispetto ai vettori permettono inserimenti e cancellazioni con una certa rapidità, ma rusultano più lente nel'accesso casuale.

Costruttori:

```
list();
list( const list & c );
list( size_type num, const TYPE& val = TYPE() );
list( input_iterator start, input_iterator end );
~ list();
```

Il costruttore di default non prevede argomenti e crea semplicemente un'istanza della lista.

La classe list

Il secondo costruttore è il costruttore di copia di default e può essere usato per creare un nuova lista come copia della lista *c*.

Il terzo costruttore crea una lista dimensionato per *num* oggetti, ciascuno dei quali assume il valore *val* (se presente). Per esempio, il seguente codice crea una lista contenente 5 copie dell'intero 42:

```
list<int> l1( 5, 42 );
```

L'ultimo costruttore crea una lista inizializzata per contenere gli elementi compresi tra *start* ed *end*.

La classe list: operatori

Tutti i contenitori C++ possono essere confrontati ed assegnati mediante gli opearori standard: ==, !=, <=, >=, <, >, and =. Il confronto e l'assegnamento tra liste richiedono un tempo lineare.

Due liste sono uguali se hanno la stessa dimensione ed ogni elemnto in una data posizione di una lista è uguale a quello nella stessa posizione dell'altra lista.

I confronti vengono effettuati in ordine lessicografico.

La classe list: metodi

assign	assign elements to a list
back	returns a reference to last element of a list
begin	returns an iterator to the beginning of the list
clear	removes all elements from the list
empty	true if the list has no elements
end	returns an iterator just past the last element of a list
erase	removes elements from a list
front	returns a reference to the first element of a list
insert	inserts elements into the list
max_size	returns the maximum number of elements
merge	merge two lists
pop_back	removes the last element of a list
pop_front	removes the first element of the list

La classe list: metodi

push_back	add an element to the end of the list
push_front	add an element to the front of the list
rbegin	returns a reverse_iterator to the end of the list
remove	removes elements from a list
remove_if	removes elements conditionally
rend	returns a reverse_iterator to the beginning of the list
resize	change the size of the list
reverse	reverse the list
size	returns the number of items in the list
sort	sorts a list into ascending order
splice	merge two lists in constant time
swap	swap the contents of this list with another
unique	removes consecutive duplicate elements

La classe list: esempio

```
// Create a list of characters
  list<char> charList;
  for( int i=0; i < 10; i++ ) {
    charList.push_front( i + 65 );
  }
  // Display the list
  list<char>::iterator theIterator;
  for( theIterator = charList.begin(); theIterator != charList.end(); theIterator++ )
{
    cout << *theIterator;
  }
```

La classe stack

Lo stack C++ è un contenitore (adattatore) che implementa una struttura di tipo FILO (first-in, last-out).

empty	true if the stack has no elements
pop	removes the top element of a stack
push	adds an element to the top of the stack
size	returns the number of items in the stack
top	returns the top element of the stack

La classe stack: esempio

```
stack<int> s;
for( int i=0; i < 10; i++ )
  s.push(i);

while( !s.empty() ) {
  cout << s.top() << " ";
  s.pop();
}
```

La classe set

Il set C++ è un contenitore associativo che contiene un insieme ordinato di oggetti.

Costruttori:

```
set();
set( const set& c );
~set();
```

Il costruttore di default non ha argomenti e crea semplicemente una nuova istanza in un tempo costante. Il costruttore di copia di default impiega un tempo lineare e può essere usato per creare una copia del set *c*.

La classe set: operatori

Tutti i contenitori C++ possono essere confrontati ed assegnati mediante gli opearori standard: ==, !=, <=, >=, <, >, and =. Il confronto e l'assegnamento tra insiemi richiedono un tempo lineare.

Due insiemi sono uguali se hanno la stessa dimensione ed ogni elemento in una data posizione di un set è uguale a quello nella stessa posizione dell'altro set.

I confronti vengono effettuati in ordine lessicografico.

La classe set: membri

begin	returns an iterator to the beginning of the set
clear	removes all elements from the set
count	returns the number of elements matching a certain key
empty	true if the set has no elements
end	returns an iterator just past the last element of a set
equal_range	returns iterators to the first and just past the last elements matching a specific key
erase	removes elements from a set
find	returns an iterator to specific elements
insert	insert items into a set

La classe set: membri

key_comp	returns the function that compares keys
lower_bound a certain value	returns an iterator to the first element greater than or equal to
max_size hold	returns the maximum number of elements that the set can
rbegin	returns a reverse_iterator to the end of the set
rend	returns a reverse_iterator to the beginning of the set
size	returns the number of items in the set
swap	swap the contents of this set with another
upper_bound value	returns an iterator to the first element greater than a certain
value_comp	returns the function that compares values

La classe map

Le mappe C++ sono contenitori associativi ordinati che contengono coppie chiave/valore uniche. Per esempio, è possibile creare una mappa che associa una stringa a un intero, ed usarla per mettere in corrispondenza il numero di giorni di ciascun mese con il nome del mese.

Una map è una sequenza di coppie (key,value) che permette l'acesso rapido per chieve. Non più di un valore viene memorizzato per ogni chiave; in altre parole ogni chiave è unica nella mappa. Una map fornisce iteratori bidirectional.

La map richiede che per i tipi della chiave sia definita l'operazione di confronto "minore od uguale" e mantiene i propri elementi ordinati così che l'iterazione avvenga in modo ordinato. Per gli elementi per i quali è complesso definire un ordine o per i quali non è necessario mantenere il contenitore ordinato è bene considerare l'uso di una *hash_map*.

La classe map

Costruttori:
map();
map(const map& m);
map(iterator start, iterator end);
map(iterator start, iterator end, const key_compare& cmp);
map(const key_compare& cmp);
~map();

Il costruttore di default non ha argomenti e crea una nuova istanza di map in un tempo lineare. Il costruttore di copia di default impiega un tempo lineare e può essere usato per creare una nuova mappa copia di una data mappa *m*.

E' inoltre possibile creare una mappa contenente una copia degli elementi compresi tra *start* ed *end* o specificare una funzione di confronto *cmp*.

La classe map

Le mappe possono essere confrontate ed assegnate per mezzo degli opearatori standard: ==, !=, <=, >=, <, >, and =. I singoli elementi di una mappa possono essere esaminati con l'operatore [].

Il confronto e l'assegnamento tra mappe richiedono un tempo lineare.

Due mappe sono uguali se hanno la stessa dimensione e ogni elemento (coppia) in una data posizione (chiave) di una mappa è uguale a quello nella stessa posizione dell'altra mappa.

Il confronto viene effettuato secondo l'ordine lessicografico.

La classe map

begin	returns an iterator to the beginning of the map
clear	removes all elements from the map
count	returns the number of elements matching a certain key
empty	true if the map has no elements
end	returns an iterator just past the last element of a map
equal_range	returns iterators to the first and just past the last elements matching a specific key
erase	removes elements from a map
find	returns an iterator to specific elements
insert	insert items into a map

La classe map

key_comp	returns the function that compares keys
lower_bound	returns an iterator to the first element greater than or equal to a certain value
max_size	returns the maximum number of elements that the map can hold
rbegin	returns a reverse_iterator to the end of the map
rend	returns a reverse_iterator to the beginning of the map
size	returns the number of items in the map
swap	swap the contents of this map with another
upper_bound	returns an iterator to the first element greater than a certain value
value_comp	returns the function that compares values

La classe map

Il seguente codice usa il metodo find() per stabilire quante volte un utente ha digitato una certa parola:

```
 map<string,int> stringCounts;
 string str;

 while( cin >> str ) stringCounts[str]++;

 map<string,int>::iterator iter = stringCounts.find("spoon");
 if( iter != stringCounts.end() ) {
   cout << "You typed '" << iter->first << "' " << iter->second << " time(s)" << endl;
 }
```

La classe map

Con il seguente input:

my spoon is too big. my spoon is TOO big! my SPOON is TOO big! I am a BANANA!

...il codice precedente produce questo output:

You typed 'spoon' 2 time(s)

I/O

La libreria <iostream> definisce pochi oggetti standard:

cout, un oggetto della classe *ostream*, che visualizza dati sul dispositivo di standard output.

cerr, un altro oggetto della classe ostream che scrive “unbuffered” output sul dispositivo di standard error.

clog, è simile a cerr, ma usa “buffered” output.

cin, un oggetto della classe *istream* che legge dati dal dispositivo di standard input.

La libreria <fstream> permette di effettuare input e output da e su file attraverso le classi *ifstream* and *ofstream*.

E’ inoltre possibile effetuare input e output da e su stringhe mediante la classe *stringstream*.

Alcuni comportamenti degli streams di I/O (precisione, giustificazione, etc) possono essere modificati manipolando vari flags di formato.

I/O

```
fstream( const char *filename, openmode mode );
ifstream( const char *filename, openmode mode );
ofstream( const char *filename, openmode mode );
```

Gli oggetti fstream, ifstream e ofstream sono usati per il file I/O. Il modo è opzionale e definisce come il file deve essere aperto, come stabilito dai flags di modo. Anche il primo argomento è opzionale e specifica il nome del file da aprire ed associare allo stream.

I file streams di input and output possono essere usati in maniera analoga a quelli predefiniti per lo standard I/O, cin and cout.

I/O

Esempio:
Il seguente codice legge dati da un file, li trasforma e li appende a un altro file.

```
ifstream fin( "/tmp/data.txt" );
ofstream fout( "/tmp/results.txt", ios::app );
while( fin >> temp )
  fout << temp + 2 << endl;
fin.close();
fout.close();
```

Mode flags

I flags di modo degli streams di I/O stream permettoni di accedere ai files in diversi modi. I flags sono:

Modo	Significato
ios::app	appendere in output
ios::ate	raggiungere EOF all'apertura
ios::binary	aprire in modalità binaria
ios::in	aprire in lettura
ios::out	aprire in scrittura
ios::trunc	sovrascrivere

I/O

bad	true if an error occurred
clear	clear and set status flags
close	close a stream
eof	true if at the end-of-file
fail	true if an error occurred
fill	manipulate the default fill character
flags	access or manipulate io stream format flags
flush	empty the buffer
gcount	number of characters read during last input
get	read characters
getline	read a line of characters
good	true if no errors have occurred
ignore	read and discard characters
open	open a new stream

I/O

peek	check the next input character
precision	manipulate the precision of a stream
put	write characters
putback	return characters to a stream
rdstate	returns the state flags of the stream
read	read data into a buffer
seekg	perform random access on an input stream
seekp	perform random access on output streams
setf	set format flags
sync_with_stdio	synchronize with standard I/O
tellg	read input stream pointers
tellp	read output stream pointers
unsetf	clear io stream format flags

I/O

width	access and manipulate the minimum field width
write	write characters

la classe string

In C++ la classe string class è una rappresentazione standard di una stringa di testo. Questa classe supera molti dei problemi tipici delle stringhe C spostando l'onere della gestione della memoria dall'utente alla classe stessa. Questa classe inoltre prevede la costruzione implicita da stringhe C, conversione esplicita in stringhe C e un operatore di confronto cosicché le stringhe possano esere confrontate con == come per gli interi invece del farraginoso "strcmp(a, b) == 0" previsto per le stringhe C.

```
#include <iostream>
#include <cassert> // For assert().
#include <string>

int main() {
  std::string foo = "hi";
  using std::string;
```

la classe string

```
  // Now we can just say "string".
  string bar = "hi";
  assert(foo == bar); // Strings can be compared with operator==.
  std::cout << foo + bar << "\n"; // Prints "hihi".
  return 0;
}
```

Siccome la stringa è memorizzata per valore la complessità della copia è O(n) (proporzionale alla lunghezza dell stringa). Per questo motivo in genere le stringhe sono passate come riferimenti costanti, cioé:

```
void print_the_string(const std::string& str) {
  std::cout << str;
}
```

la classe string

Costruttori:

```
  string( const string& s );
  string( size_type length, const char& ch );
  string( const char* str );
  string( const char* str, size_type length );
  string( const string& str, size_type index, size_type length );
  string( input_iterator start, input_iterator end );
```

la classe string

I costruttori di stringa creano una nuova stringa contenente:
Nulla, una stringa vuota,
una copia di una data stringa *s*,
length copie di un carattere *ch*,
un duplicato di *str* (opzionalmente dei primi *length* caratteri),
una sottostringa di *str* dall'indice *index* e di lunghezza *length*
una stringa dei caratteri compresi tra *start* ed *end*

per esempio,

```
string str1( 5, 'c' );
string str2( "Now is the time..." );
```

la classe string

```
string str3( str2, 11, 4 );
cout << str1 << endl;
cout << str2 << endl;
cout << str3 << endl;
```

stampa:

```
ccccc
Now is the time...
time
```

Normalmente i costruttori di stringa impiegano un tempo lineare, tranne il costruttore vuoto che impiega un tempo costante.

la classe string

Le stringhe possono essere confrontate ed assegnate mediante gli operatori standard: ==, !=, <=, >=, <, >, and =. Il confronto e l'assegnamento tra stringhe richiedo un tempo lineare.

Due stringhe sono uguali se la loro lunghezza coincide e ogni carattere in una data posizione di una stringa è uguale a quello nella stessa posizione dell'altra.

I confronti sono effettuati in ordine lessicografico.

Le stringhe possono inoltre essere concatenate con l'opeartore + e inviate agli streams di I/O con gli opeartori << and >>.

la classe string

Il seguente codice concatena due stringhe e visualizza il risultato:

```
string s1 = "Now is the time...";
string s2 = "for all good men...";
string s3 = s1 + s2;
cout << "s3 is " << s3 << endl;
```

Inoltre, le stringhe possono assegnare valori come altre stringhe, arrays di caratteri o caratteri singoli. Esempio:

```
char ch = 'N';
string s;
s = ch;
```

I singoli caratteri possono essere esaminati con l'operatore [] che impiega un tempo costante.

la classe string

append append characters and strings onto a string
assign give a string values from strings of characters and other C++ strings
at returns an element at a specific location
begin returns an iterator to the beginning of the string
c_str returns a non-modifiable standard C character array version of the string
capacity returns the number of elements that the string can hold
clear removes all elements from the string
compare compares two strings

la classe string

copy copies characters from a string into an array
data returns a pointer to the first character of a string
empty true if the string has no elements
end returns an iterator just past the last element of a string
erase removes elements from a string
find find characters in the string
find_first_not_of find first absence of characters
find_first_of find first occurrence of characters

la classe string

find_last_not_of find last absence of characters
find_last_of find last occurrence of characters
getline read data from an I/O stream into a string
insert insert characters into a string
length returns the length of the string
max_size returns the maximum number of elements that the string can hold
push_back add an element to the end of the string
rbegin returns a reverse_iterator to the end of the string

la classe string

rend returns a reverse_iterator to the beginning of the string
replace replace characters in the string
reserve sets the minimum capacity of the string
resize change the size of the string
rfind find the last occurrence of a substring
size returns the number of items in the string
substr returns a certain substring
swap swap the contents of this string with another

String streams

Gli streams di stringhe sono simili a <iostream> e <fstream>, salvo che gli streams di stringhe permettono di effettuare I/O sulle stringhe invece che sugli streams. La libreria <sstream> fornisce funzionalità analoghe a sscanf() e sprintf() della libreria standard del C.

Principalmente tre classi sono disponibili in <sstream>:

stringstream - allows input and output

istringstream - allows input only

ostringstream - allows output only

Gli string streams attualmente sono sottoclassi degli iostreams, cosicché tutte le funzioni disponibili per gli I/O streams lo sono anche per gli string streams.

String streams

Costruttori:

```
stringstream()
stringstream( openmode mode )
stringstream( string s, openmode mode )
ostringstream()
ostringstream( openmode mode )
ostringstream( string s, openmode mode )
istringstream()
istringstream( openmode mode )
istringstream( string s, openmode mode )
```

String streams

Gli oggetti stringstream, ostringstream, and istringstream sono usati per input and output da e su stringhe. Si comportano in maniera simile agli oggetti fstream, ofstream e ifstream.

Il parametro *mode* è opzionale e definisce come il file deve essere aperto, come stabilito dai flags di modo.

Un oggetto ostringstream può essere usato per scrivere in una stringa in manieara analoga alla funzione C sprintf(). For example:

```
ostringstream s1;
int i = 22;
```

String streams

```
s1 << "Hello " << i << endl;
string s2 = s1.str();
cout << s2;
```

Un oggetto istringstream può essere usato per leggere da una stringa in maniera analoga alla funziona C sscanf(). For example:

```
istringstream stream1;
string string1 = "25";
stream1.str(string1);
int i;
stream1 >> i;
cout << i << endl;  // displays 25
```

String streams

E’ possibile specificare la stringa di input direttamente nel costruttore di istringstream come nell’esempio:

```
string string1 = "25";
istringstream stream1(string1);
int i;
stream1 >> i;
cout << i << endl;  // displays 25
```

Un oggetto stringstream può essere usato sia per input che per l’output da e su stringa come un oggetto fstream.

String streams

rdbuf get the buffer for a string stream
str get or set the stream's string

Letture consigliate

The C++ Programming Language (Third Edition and Special Edition), by Bjarne Stroustrup, http://www.research.att.com/~bs/3rd.html

Thinking in C++ 2nd Edition by Bruce Eckel, disponibile anche in formato elettronico (http://www.mindview.net/Books/TICPP/ThinkingInCPP2e.html)

Design Patterns - Elements of Reusable Object-Oriented Software, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.

Riferimenti Web

http://www.cppreference.com

http://www.cplusplus.com

http://www.boost.org, Boost provides free peer-reviewed portable C++ source libraries.

http://gcc.gnu.org, The GNU Compiler Collection

http://www.kdevelop.org, ambiente di sviluppo integrato (IDE) per KDE

http://www.bloodshed.net/devcpp.html, Bloodshed Dev-C++ is a full-featured Integrated Development Environment (IDE)

http://www.trolltech.com, Cross-platform C++ GUI framework

http://www.wxwindows.org, API for writing GUI applications on multiple platforms

Esercizi

Obiettivo

Implementare un dato di tipo matrice che supporti le seguenti funzionalità:

- Scelta del tipo degli elementi;
- Scelta delle dimensioni;
- Accesso in lettura/scrittura ai singoli elementi;
- Stampa in forma tabellare.

Mat =

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20

Si propongono varie implementazioni via via più evolute.

Procedurale

inizialmente il problema viene risolto in forma procedurale definendo il dato come struttura e fornendo le opportune funzioni di gestione (allocazione/deallocazione, accesso, stampa). Il tipo degli elementi è scelto dallo sviluppatore.

```
//main.cpp
#include "matrix.h"

int main(int argc, char *argv[])
{
  const int nrows = 4;
  const int ncols = 5;
  Matrix *mat;

  mat = MatrixCreate(nrows, ncols);
  for (int i = 0; i < nrows; i++)
    for(int j = 0; j < ncols; j++)
      MatrixSetValue(mat, i, j, 1 + j + i * ncols);

  MatrixPrint(mat);
  MatrixDestroy(mat);

  return 0;
}
```

2

A oggetti

Il dato viene definito come oggetto: le attività di allocazione/deallocazione vengono espletate nel costruttore/distruttore mentre le funzioni di accesso e stampa vengono tradotte nei corrispondenti metodi.

```
//main.cpp
#include "matrix.h"

int main(int argc, char *argv[])
{
  const int nrows = 4;
  const int ncols = 5;
  Matrix mat(nrows, ncols);

  for (int i = 0; i < nrows; i++)
    for(int j = 0; j < ncols; j++)
      mat.SetValue(i, j, 1 + j + i * ncols);

  mat.Print();
  return 0;
}
```

3

Con riferimenti

I metodi di accesso vengono unificati usando riferimenti agli elementi.

```
//main.cpp
#include "matrix.h"

int main(int argc, char *argv[])
{
  const int nrows = 4;
  const int ncols = 5;
  Matrix mat(nrows, ncols);

  for (int i = 0; i < nrows; i++)
    for(int j = 0; j < ncols; j++)
      mat.Value(i, j) = 1 + j + i * ncols;

  mat.Print();
  return 0;
}
```


Con overloading

I metodi di accesso e di stampa vengono semplificati sovraccaricando gli opportuni operatori.

```
//main.cpp
#include "matrix.h"

int main(int argc, char *argv[])
{
  const int nrows = 4;
  const int ncols = 5;
  Matrix mat(nrows, ncols);

  for (int i = 0; i < nrows; i++)
    for(int j = 0; j < ncols; j++)
      mat(i, j) = 1 + j + i * ncols;

  cout << mat;
  return 0;
}
```

Con classi derivate

L'oggetto matrice viene specializzato nella versione densa e sparsa. L'operatore di accesso diviene puramente virtuale. Gli elementi vengono memorizzati negli opportuni contenitori della libreria standard.

```
//main.cpp
#include "densematrix.h"
#include "sparsematrix.h"

int main(int argc, char *argv[])
{
  const int nrows = 4;
  const int ncols = 5;
  int i, j;

DenseMatrix densemat(nrows, ncols);
  for (i = 0; i < nrows; i++)
    for(j = 0; j < ncols; j++)
      densemat(i, j) = 1 + j + i * ncols;
  cout << "dense matrix\n" << densemat;

  SparseMatrix sparsemat(nrows, ncols);
  for (i = 0; i < nrows; i++)
    for(j = 0; j < ncols; j++)
      if (i % 2 - j % 2)
      sparsemat(i, j) = 1 + j + i * ncols;
   cout << "sparse matrix\n" <<
sparsemat;

  return 0;
}
```


e infine … parametriche

Le classi matrici vengono parametrizzate rispetto al tipo degli elementi in modo che questo possa essere scelto dall'utente.

```
//main.cpp
#include "densematrix.h"
#include "sparsematrix.h"

int main(int argc, char *argv[])
{
  const int nrows = 4;
  const int ncols = 5;
  int i, j;

  DenseMatrix<float> densemat(nrows,
ncols);

for (i = 0; i < nrows; i++)
    for(j = 0; j < ncols; j++)
      densemat(i, j) = 1 + j + i * ncols;
  cout << "dense matrix\n" << densemat;

  SparseMatrix<int> sparsemat(nrows,
ncols);
  for (i = 0; i < nrows; i++)
    for(j = 0; j < ncols; j++)
      if (i % 2 - j % 2)
      sparsemat(i, j) = 1 + j + i * ncols;

  cout << "sparse matrix\n" << sparsemat;

  return 0;
}
```