

CAPITOLO I

I template

Obiettivi

- Imparare a utilizzare i template per creare modelli di funzioni
- Conoscere la differenza tra template di funzioni e istanze di template di funzioni
- Imparare a utilizzare i template di classi per creare un gruppo di tipi correlati
- Conoscere la differenza tra template di classe e istanze di template di classe
- Comprendere come effettuare l'overloading delle funzioni di template
- Capire le relazioni che legano template, funzioni friend, ereditarietà e membri statici

1.1 Introduzione

In questo capitolo parleremo di una delle caratteristiche più potenti e versatili del C++ che è presente in ben pochi linguaggi di programmazione orientati agli oggetti: i *template*. Questo meccanismo consente di specificare in un solo segmento di codice il modello per un vasto insieme di funzioni o classi correlate.

Possiamo scrivere, per esempio, un template di funzione per l'ordinamento di un array, ordinando al compilatore di generare automaticamente le diverse versioni della funzione che trattano i diversi tipi concreti (ad es. **int**, **float**, stringhe e così via). In un certo senso i template definiscono delle classi o delle funzioni generiche che si possono adattare a diversi tipi di dato.

Come ulteriore esempio, possiamo scrivere un template di una classe che rappresenti una pila, indicando al compilatore di generare in seguito le varie classi di template, come la classe pila-di-**int**, pila-di-**float**, pila-di-**string** e così via.

Abbiamo già parlato brevemente dei template di funzione nel Capitolo 3 del volume Fondamenti e in questo capitolo li introduciamo nuovamente, presentando ulteriori esempi del loro utilizzo.

C'è una distinzione tra i concetti di template di funzione (o di classe) e di istanza di template di funzione (o di classe): i template di funzione (o di classe) sono paragonabili a degli stampini con cui possiamo tracciare delle figure mentre le istanze di template di funzione (o di classe) sono invece come i diversi disegni, tracciati con lo stesso stampino, ma magari di colore (tipo) diverso.



Ingegneria del software 1.1

I template sono una delle caratteristiche del C++ che consente di scrivere codice riutilizzabile.

In questo capitolo presenteremo diversi esempi di template di funzione e di classe. Esamineremo anche le relazioni esistenti tra i template e le altre caratteristiche del C++, come l'overloading, l'ereditarietà, le funzioni friend e i membri statici.

I dettagli di progettazione dei template illustrati in questo capitolo si basano sul lavoro di Bjarne Stroustrup, così come è presentato nell'articolo "Parameterized Types for C++" (Tipi parametrici per il C++) pubblicato nella "Proceedings of the USENIX C++ Conference" che ha avuto luogo a Denver, in Colorado, nell'ottobre del 1988. Questo capitolo vuole essere soltanto un'introduzione all'argomento ricco e complesso dei template.

1.2 I template di funzione

Per effettuare lo stesso genere di operazioni su tipi di dati diversi si utilizza di norma l'overloading di una funzione. Se le operazioni sono identiche su ogni tipo di dato conviene scrivere un codice più compatto, che faccia uso dei template di funzione. Il compito di un programmatore in questo caso si esaurisce nello scrivere la definizione di un solo template poi, sulla base del tipo degli argomenti che vengono passati nelle chiamate a tale funzione, il compilatore genererà automaticamente nel codice oggetto le diverse funzioni che trattano ogni tipo di chiamata nel modo corretto. In C un compito simile è svolto dalle macro, in particolare dalle direttive al preprocessore **#define** che, però, presentano il rischio di effetti collaterali anche gravi e non permettono al compilatore di effettuare alcun controllo sui tipi di dato. I template funzionano un po' come le macro, ma consentono un controllo sui tipi.



Collaudo e messa a punto 1.1

I template di funzione, come le macro, facilitano il riutilizzo del software. A differenza delle macro, però, i template di funzione permettono di evitare un gran numero di errori, perché il compilatore può effettuare un controllo sui tipi di dato trattati.

La definizione di un template di funzione inizia con la parola chiave **template** e continua con la lista dei parametri formali del template racchiusa tra una coppia di *parentesi angolari* (< e > cioè esattamente i segni di maggiore e minore). Ogni parametro formale, che rappresenta un tipo, deve essere preceduto dalla parola chiave **class** (o dalla parola chiave **typename**), come in

```
template< typename T >
```

o in

```
template< class BorderType, class FillType >
```

I parametri formali possono servire per specificare il tipo di argomenti della funzione, per indicare il tipo restituito dalla funzione o per dichiarare variabili all'interno della funzione, proprio come i tipi predefiniti o definiti dall'utente. Dopo questa intestazione segue la definizione della funzione che è identica a quella di una funzione normale. Il significato della parola chiave **class** (o della parola chiave **typename**) che specifica i parametri di tipo è "qualsiasi tipo di dato predefinito o definito dall'utente".



Errore tipico 1.1

Se dimenticate la parola chiave **class** (o la nuova parola chiave **typename**) prima di ogni parametro formale di tipo nella definizione di un template di funzione, commetterete un errore.

Esaminiamo adesso il template della funzione **printArray** in Figura 1.1. Questo template viene utilizzato nel programma di Figura 1.2.

```

1  template< class T >
2  void printArray( const T *array, const int count )
3  {
4      for ( int i = 0; i < count; i++ )
5          cout << array[ i ] << " ";
6      cout << endl;
7  }
```

Figura 1.1 Un template di funzione.

Il template **printArray** dichiara un solo parametro formale, **T**, relativo al tipo di array che **printArray** deve visualizzare. **T** prende il nome di *parametro di tipo*; quando il compilatore incontra nel codice sorgente una chiamata a **printArray**, sostituisce **T** con il tipo del primo argomento di **printArray** in tutto il template. In questo modo il compilatore crea una funzione completa che visualizza un array che contiene il tipo specificato; essa viene poi compilata come qualsiasi altra funzione. In Figura 1.2 vengono istanziate tre funzioni **printArray**: la prima riceve un array di **int**, la seconda un array di **double** e la terza un array di **char**. Per esempio, l'istanza creata per il tipo **int** corrisponde al seguente codice:

```

void printArray( const int *array, const int count )
{
    for ( int i = 0; i < count; i++ )
        cout << array[ i ] << " ";
    cout << endl;
}
```

Ogni parametro formale deve comparire almeno una volta nella lista dei parametri della funzione. Il nome di ciascun parametro formale può essere utilizzato soltanto una volta nella lista di parametri che fa parte dell'intestazione di un template mentre può essere usato lo stesso nome in diversi template di funzioni. Inoltre un parametro formale deve essere univoco tra le funzioni di template.

La Fig 1.2 illustra l'uso del template di funzione **printArray**. Il programma inizia istanziando l'array di interi **a**, l'array di **double** **b** e l'array di **char** **c**, rispettivamente di dimensioni **5**, **7** e **6**. Ogni array viene poi visualizzato chiamando **printArray**, una prima volta con l'argomento **a** di tipo **int ***, la seconda con **b** di tipo **double *** e l'ultima volta con **c** di tipo **char ***. La chiamata

```
printArray( a, aCount );
```

informa il compilatore di istanziare il template di funzione di nome **printArray** per il quale il parametro **T** è **int** mentre la chiamata

```
printArray( b, bCount );
```

informa il compilatore di istanziare il template di funzione di nome **printArray** per il quale il parametro **T** è **double**. Infine, la chiamata

```
printArray( c, cCount );
```

indica al compilatore di istanziare il template di funzione di nome **printArray** per il quale il parametro **T** è **char**.

```

1 // Fig 1.2: fig1_02.cpp
2 // Utilizzo delle funzioni di template
3 #include <iostream.h>
4
5 template< class T >
6 void printArray( const T *array, const int count )
7 {
8     for ( int i = 0; i < count; i++ )
9         cout << array[ i ] << " ";
10
11     cout << endl;
12 }
13
14 int main()
15 {
16     const int aCount = 5, bCount = 7, cCount = 6;
17     int a[ aCount ] = { 1, 2, 3, 4, 5 };
18     double b[bCount] = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7};
19     char c[ cCount ] = "HELLO"; // sesta posizione per il
20                                 // terminatore nullo
21
22     cout << "Array a contains:" << endl;
23     printArray( a, aCount ); // istanza del template di
24                               // funzione per interi
25
26     cout << "Array b contains:" << endl;
27     printArray( b, bCount ); // istanza del template di
28                               // funzione per double
29
30     cout << "Array c contains:" << endl;
31     printArray( c, cCount ); // istanza del template di
32                               // funzione per caratteri
33
34     return 0;
35 }
```

```

Array a contains:
1 2 3 4 5
Array b contains:
1.1 2.2 3.3 4.4 5.5 6.6 7.7
Array c contains:
H E L L O
```

Figura I.2 Uso delle funzioni di template.

Nel prossimo esempio vedremo come i template possano risparmiarci l'onere di scrivere tre overloading diversi della stessa funzione, i cui prototipi sono:

```
void printArray( const int *, const int );
void printArray( const double *, const int );
void printArray( const char *, const int );
```



Obiettivo efficienza 1.1

I template hanno il pregio di facilitare il riutilizzo del software ma ricordate che, anche se scrivete un solo template, il compilatore istanzia diverse copie di funzioni e classi di template che possono occupare un notevole spazio in memoria.

1.3 L'overloading delle funzioni di template

Il concetto di template richiama quello di overloading: le funzioni correlate che sono generate da un template hanno tutte lo stesso nome, per cui il compilatore utilizza i meccanismi di risoluzione tipici dell'overloading per invocare la funzione giusta.

Lo stesso template di una funzione può subire overloading in molti modi diversi, ad esempio specificando funzioni con lo stesso nome che, però, ricevono parametri diversi. Per esempio, il template **printArray** in Figura 1.2 può subire overloading da parte di un altro template **printArray** che prende due parametri in più, **lowSubscript** (indice di partenza) e **highSubscript** (indice finale), che specificano la porzione di array da visualizzare (cfr. Esercizio 1.4).

Un template di funzione subisce overloading se nel programma sono presenti altre funzioni che abbiano nome identico ma argomenti diversi. Per esempio il template **printArray** in Figura 1.2 può subire overloading da parte di una versione non template della funzione che visualizzi un array di stringhe in formato tabulare (cfr. Esercizio 1.5).



Errore tipico 1.2

Se invocate un template con un tipo di classe definito dall'utente e se esso utilizza alcuni operatori (come ==, + e <=) sugli oggetti di tale classe, dovete prevedere anche l'overloading di tali operatori! Se lo dimenticate, il compilatore vi segnalerà un errore, perché genererà delle chiamate a operatori inesistenti.

Quando viene chiamata una funzione, il compilatore effettua una serie di confronti tra le varie funzioni disponibili per determinare quale di esse va chiamata. Per prima cosa, il compilatore cerca una corrispondenza precisa del nome della funzione e dei tipi degli argomenti tra funzione chiamata e funzioni disponibili. Se non esiste una corrispondenza esatta, continua la ricerca sui template disponibili, per vedere se ne esiste uno da cui può generare una funzione che tratti il tipo di argomenti in questione. Se trova lo trova, il compilatore genera l'istanza del template di funzione appropriato.

Fino a tempi relativamente recenti questo procedimento richiedeva una corrispondenza esatta del tipo di tutti gli argomenti e non venivano effettuate conversioni automatiche. Questo limite oggi è stato superato e, dunque, non occorre una corrispondenza precisa dei tipi, ma si possono applicare le tipiche regole dell'overloading.



Errore tipico 1.3

Quando incontra una chiamata di funzione nel codice sorgente, il compilatore effettua una serie di confronti per determinare quale funzione invocare. Se non trova una corrispondenza accettabile, o se ne trova più di una, il compilatore genera un errore.

1.4 I template di classe

È possibile comprendere che cosa sia una pila indipendentemente dal tipo di elementi che contiene: essa è una struttura dati dinamica in cui è possibile inserire gli elementi in un ordine ed estrarli in ordine inverso (si dice anche che essa è una struttura *LIFO*, dall'inglese *last-in-first-out*, ovvero l'ultimo elemento inserito è il primo a essere estratto). Se volete istanziare una pila, però, dovete necessariamente specificare il tipo di dato contenuto. Questo è un buon esempio per scrivere software riutilizzabile: tutto ciò di cui avete bisogno è un mezzo per descrivere il concetto di pila a un livello generale, tale da consentirvi di istanziare successivamente delle versioni specifiche. Questa funzionalità è offerta proprio dai template di classe del C++.



Ingegneria del software 1.2

I template di classe facilitano la scrittura di software riutilizzabile, permettendo di istanziare diverse versioni specifiche di una classe generica.

I template di classe sono detti *tipi parametrici* perché hanno bisogno di uno o più parametri per generare l'istanza del template di classe desiderato.

Per generare una collezione di classi, dunque, basta scrivere la definizione di un solo template di classe e, ogni volta che si ha bisogno di una nuova istanza specifica, è il compilatore che genererà il codice sorgente dell'istanza del template di classe richiesta. Un template di classe **Stack** (pila), per esempio, può essere la base di partenza per molte classi **Stack** come “**Stack di double**”, “**Stack di int**”, “**Stack di char**”, “**Stack di Employee**” e così via.

Osservate la definizione del template di classe **Stack** in Figura 1.3. Come vedete, assomiglia molto alla definizione di una classe convenzionale, tranne per il fatto che è preceduta dall'intestazione (linea 8)

```
template< class T >
```

che indica appunto che si tratta di un template. Il parametro di tipo **T** indica il tipo di classe **Stack** da creare. Naturalmente, potete utilizzare qualsiasi identificatore valido al posto di **T**. Il tipo di elemento da memorizzare in **Stack** è menzionato genericamente come **T** in tutta l'intestazione della classe **Stack** e nelle definizioni delle funzioni membro. Mostriamo adesso come **T** viene associato a un tipo specifico, come **double** o **int**.

```
1 // Figura 1.3: tstack1.h
2 // Il template di classe Stack
3 #ifndef TSTACK1_H
4 #define TSTACK1_H
5
6 #include <iostream.h>
```

Figura 1.3 Uso del template di classe Stack (continua)

```

7
8  template< class T >
9  class Stack {
10 public:
11     Stack( int = 10 ); // costruttore di default.
12     ~Stack() { delete [] stackPtr; } // distruttore
13     bool push(const T&); // inserisce un elemento sulla pila
14     bool pop( T& );      // recupera un elemento dalla pila
15 private:
16     int size;           // numero di elementi nella pila
17     int top;            // indice dell'elemento in cima alla pila
18     T *stackPtr;       // puntatore alla pila
19
20     bool isEmpty() const {return top == -1;} //f. di utilità
21     bool isFull() const { return top == size - 1; } // c.s.
22 };
23
24 // Costruttore con dimensione di default = 10
25 template< class T >
26 Stack< T >::Stack( int s )
27 {
28     size = s > 0 ? s : 10;
29     top = -1;                //Stack è inizialmente vuoto
30     stackPtr = new T[size]; //alloca spazio per gli elementi
31 }
32
33 // Spinge un elemento sulla pila e restituisce true se
34 // l'operazione riesce e false in caso contrario
35 template< class T >
36 bool Stack< T >::push( const T &pushValue )
37 {
38     if ( !isFull() ) {
39         stackPtr[ ++top ] = pushValue; // pone un elemento
40                                         // sullo Stack
41         return true; // operazione riuscita
42     }
43     return false; // operazione non riuscita
44 }
45
46 // Recupera un elemento dalla pila
47 template< class T >
48 bool Stack< T >::pop( T &popValue )
49 {
50     if ( !isEmpty() ) {
51         popValue = stackPtr[ top-- ]; // elimina l'elemento
52                                         // dallo Stack
53         return true; // operazione riuscita
54     }

```

Figura I.3 Uso del template di classe Stack (continua)

```

55     return false;      // operazione non riuscita
56   }
57
58   #endif

```

Figura I.3 Uso del template di classe Stack (continua).

Vediamo ora un programma di prova che fa uso del template di classe **Stack** (cfr. l'output in Figura 1.3). All'inizio essa istanzia l'oggetto **doubleStack** di dimensione **5** dichiarato come **Stack< double >** (si legge "**Stack** di **double**").

Il compilatore associa il tipo **double** al parametro di tipo **T** nel template per generare il codice sorgente di una classe **Stack** di tipo **double**. Anche se il codice non è presente esplicitamente nel programma, viene incluso durante la compilazione.

```

59   // Figura 1.3: fig1_03.cpp
60   // Programma di esempio per il template Stack
61   #include <iostream.h>
62   #include "tstack1.h"
63
64   int main()
65   {
66       Stack< double > doubleStack( 5 );
67       double f = 1.1;
68       cout << "Pushing elements onto doubleStack\n";
69
70       while ( doubleStack.push( f ) ) { // ha successo se
71                                           // restituisce true
72           cout << f << ' ';
73           f += 1.1;
74       }
75
76       cout << "\nStack is full. Cannot push " << f
77           << "\n\nPopping elements from doubleStack\n";
78
79       while ( doubleStack.pop( f ) ) // ha successo se
80                                           // restituisce true
81           cout << f << ' ';
82
83       cout << "\nStack is empty. Cannot pop\n";
84
85       Stack< int > intStack;
86       int i = 1;
87       cout << "\nPushing elements onto intStack\n";
88
89       while ( intStack.push( i ) ) { // ha successo se
90                                           // restituisce true
91           cout << i << ' ';
92           ++i;
93       }

```

Figura I.3 Uso del template di classe Stack (continua)

```

94
95     cout << "\nStack is full. Cannot push " << i
96         << "\n\nPopping elements from intStack\n";
97
98     while ( intStack.pop( i ) ) // ha successo se
99         // restituisce true
100         cout << i << ' ';
101
102     cout << "\nStack is empty. Cannot pop\n";
103     return 0;
104 }

```

```

Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Stack is full. Cannot push 6.6

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty. Cannot pop

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10
Stack is full. Cannot push 11

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty. Cannot pop

```

Figura 1.3 Programma di esempio per il template di classe Stack.

Il programma inserisce su **doubleStack** i valori **double 1.1, 2.2, 3.3, 4.4 e 5.5** in sequenza. Il ciclo che effettua questa operazione termina quando il programma tenta di inserire un sesto valore, in quanto **doubleStack** si è già riempito.

Il programma estrae quindi i cinque valori dalla pila (osservate l'ordine dei valori estratti). Il ciclo termina quando il programma tenta di estrarre un sesto valore e trova **doubleStack** vuoto.

Dopo queste operazioni, il programma istanzia la pila di interi **intStack** con la dichiarazione

```
Stack< int > intStack;
```

(si legge “**intStack** è un oggetto **Stack** di **int**”). Non essendo specificata alcuna dimensione si assume il valore di default **10**, come indicato nel costruttore di default (linea 11). Anche questa volta, il programma inserisce i valori in **intStack** finché questo non si riempie e li estrae finché non si svuota. Notate che l'ordine dei valori in output segue la regola *LIFO*.

Le definizioni delle funzioni membro che si trovano fuori del template di classe (linea 24) cominciano tutte con

```
template< class T >
```

Come abbiamo già detto, ogni definizione ricorda la definizione di una funzione convenzionale, tranne per il fatto che il tipo dell'elemento di **Stack** è sempre indicato genericamente con il parametro di tipo **T**. L'operatore binario di risoluzione dello scope utilizzato con il nome del template **Stack< T >** serve a legare la definizione di ogni funzione membro allo scope del template di classe. In questo caso, il nome della classe è **Stack< T >**. Quando viene istanziato l'oggetto di tipo **Stack< double >** di nome **doubleStack**, il costruttore di **Stack** utilizza **new** per creare un array di elementi di tipo **double**, che conterrà la rappresentazione della pila (linea 29). L'istruzione

```
stackPtr = new T[ size ];
```

nella definizione del template di classe **Stack**, diventa nell'istanza **Stack< double >**

```
stackPtr = new double[ size ];
```

Notate il codice della funzione **main** in Figura 1.3: è quasi identico per le operazioni su **doubleStack** nella prima metà della funzione e quelle su **intStack** nella seconda metà. Questo ci suggerisce che possiamo utilizzare ancora un template di funzione ed è proprio ciò che facciamo nel programma di Figura 1.4. Esso definisce il template di funzione **testStack** per effettuare le stesse operazioni di **main** in Figura 1.3, cioè inserisce una serie di valori su **Stack< T >** per poi estrarli. Il template di funzione **testStack** utilizza il parametro formale **T** per rappresentare il tipo di dati da memorizzare in **Stack< T >**. Il template prende quattro argomenti: un riferimento a un oggetto di tipo **Stack< T >**, un valore di tipo **T** che sarà il primo valore a essere spinto su **Stack< T >**, un valore di tipo **T** utilizzato per incrementare i valori spinti su **Stack< T >** e una stringa di caratteri di tipo **const char *** che rappresenta il nome dell'oggetto **Stack< T >**, che verrà utilizzata nell'output del programma. La nuova versione di **main** istanzia semplicemente un oggetto di tipo **Stack< double >** di nome **doubleStack** e un oggetto di tipo **Stack< int >** di nome **intStack**, e utilizza questi due oggetti nelle linee 37 e 38 come segue

```
testStack( doubleStack, 1.1, 1.1, "doubleStack" );
testStack( intStack, 1, 1, "intStack" );
```

Come vedete, l'output di Figura 1.4 è identico all'output di Figura 1.3.

```
1 // Figura 1.4: fig1_04.cpp
2 // Programma di esempio per il template Stack.
3 // La funzione main utilizza un template di funzione per
4 // manipolare gli oggetti di tipo Stack< T >.
5 #include <iostream.h>
6 #include "tstack1.h"
7
8 // Template di funzione che manipola Stack< T >
9 template< class T >
10 void testStack(
11     Stack< T > &theStack, // riferimento a Stack< T >
12     T value,             // valore iniziale da inserire
13     T increment,        // incremento per i valori successivi
14     const char *stackName ) // nome dell'oggetto Stack< T >
15 {
16     cout << "\nPushing elements onto " << stackName << '\n';
17 }
```

Figura 1.4 Passaggio di un oggetto template **Stack** a un template di funzione (continua).

```

18     while ( theStack.push( value ) ) { // ha successo se
19                                     // restituisce true
20         cout << value << ' ';
21         value += increment;
22     }
23
24     cout << "\nStack is full. Cannot push " << value
25           << "\n\nPopping elements from " << stackName << '\n';
26
27     while ( theStack.pop( value ) ) // ha successo se
28                                     // restituisce true
29         cout << value << ' ';
30
31     cout << "\nStack is empty. Cannot pop\n";
32 }
33
34 int main()
35 {
36     Stack< double > doubleStack( 5 );
37     Stack< int > intStack;
38
39     testStack( doubleStack, 1.1, 1.1, "doubleStack" );
40     testStack( intStack, 1, 1, "intStack" );
41
42     return 0;
43 }

```

```

Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Stack is full. Cannot push 6.6

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty. Cannot pop

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10
Stack is full. Cannot push 11

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty. Cannot pop

```

Figura 1.4 Passaggio di un oggetto template Stack a un template di funzione.

1.5 I template di classe e i parametri non di tipo

Il template di classe **Stack** della sezione precedente utilizzava un solo parametro di tipo nella sua intestazione. È possibile anche utilizzare parametri *non di tipo*; essi possono prevedere un argomento di default e vengono trattati come dati **const**.

Per esempio, l'intestazione del template può essere modificata in modo che preveda il parametro **int elements**, come segue:

```
template< class T, int elements > // parametro non di tipo
```

Quindi una dichiarazione come

```
Stack< double, 100 > mostRecentSalesFigures;
```

istanzierà in fase di compilazione una **Stack** di **100** elementi di nome **mostRecentSalesFigures**, che conterrà valori **double**. Il tipo di questa istanza sarà **Stack< double, 100 >**. L'intestazione della classe può contenere un dato membro privato con una dichiarazione di array, come

```
T stackHolder[ elements ]; // array che conterrà gli elementi
                          // della pila
```



Obiettivo efficienza 1.2

*Quando è possibile, specificate in fase di compilazione la dimensione di una classe container tramite un parametro non di tipo, perché ciò evita lo spreco di tempo richiesto da **new** in esecuzione.*



Ingegneria del software 1.3

*Quando è possibile, specificate la dimensione di un container in fase di compilazione con un parametro non di tipo, perché questo vi permetterà di evitare potenziali errori fatali durante l'esecuzione (ad esempio se **new** non riesce a ottenere lo spazio di memoria richiesto).*

Negli esercizi vi chiederemo di utilizzare un parametro non di tipo per creare un template della classe **Array** che abbiamo studiato nel Capitolo 8 del volume Fondamenti. Questo template permetterà di istanziare oggetti **Array** con un numero di elementi specificato e di un tipo specificato durante la compilazione, senza differire la creazione dell'oggetto in fase di esecuzione per mezzo di un'allocazione dinamica.

È possibile effettuare l'overriding del template per un dato tipo. Per esempio, anche se il template di classe **Array** può essere utilizzato per istanziare array di qualsiasi tipo, potreste volere un controllo diretto sull'istanza di **Array** di un tipo specifico, supponiamo per il tipo **Martian**. Per ottenere questo basta semplicemente ridefinire esplicitamente una nuova classe di nome **Array<Martian>**.

1.6 I template e l'ereditarietà

I template e l'ereditarietà sono in stretta correlazione:

- Un template di classe può derivare da un template di classe.
- Un template di classe può derivare da una classe non di template.
- Una istanza di template di classe può derivare da un template di classe.
- Una classe non di template può derivare da un template di classe.

1.7 I template e la relazione friend

Abbiamo visto che è possibile dichiarare funzioni e classi come **friend** di classi non di template. Con i template di classe è possibile dichiarare relazioni **friend** più naturali. Si può instaurare una relazione **friend** tra un template di classe e: una funzione globale, una funzione membro, una funzione membro di un'altra classe (anche una istanza di un template di classe) o persino un'intera classe. Le notazioni relative a queste relazioni non sono purtroppo sempre chiarissime.

In un template di classe per la classe **X** dichiarato con

```
template< class T > class X
```

una dichiarazione **friend** della forma

```
friend void f1();
```

rende **f1** funzione **friend** di ogni istanza del precedente template di classe.

In un template di classe per la classe **X** dichiarato con

```
template< class T > class X
```

una dichiarazione **friend** della forma

```
friend void f2( X< float > & );
```

rende la funzione **friend** solo della classe **X< float >**.

In un template di classe potete dichiarare che una funzione membro di un'altra classe è **friend** di ogni possibile istanza del template di classe generato. Basta indicare il nome della funzione membro dell'altra classe insieme con l'operatore binario di risoluzione dello scope. Per esempio, in un template di classe per la classe **X** dichiarato con

```
template< class T > class X
```

una dichiarazione **friend** della forma

```
friend void A::f4();
```

rende la funzione membro **f4** della classe **A** funzione **friend** di ogni possibile istanza del template di classe.

In un template di classe per la classe **X** dichiarato con

```
template< class T > class X
```

una dichiarazione **friend** della forma

```
friend void C< float >::f5( X< float > & );
```

rende la funzione membro una funzione **friend** della sola classe **X< float >**.

In un template di classe per la classe **X** dichiarato con

```
template< class T > class X
```

si può dichiarare una seconda classe **Y**

```
friend class Y;
```

rendendo ogni funzione membro della classe **Y** funzione **friend** di ogni istanza di template di classe generato dal template per **X**.

In un template di classe per la classe **X** dichiarato con

```
template< class T > class X
```

si può dichiarare una seconda classe **Z**

```
friend class Z< T >;
```

in modo tale che, quando viene istanziata una classe con un tipo particolare per **T** (ad es. **float**), tutti i membri della classe **Z< float >** diventino **friend** dell'istanza **X< float >**.

1.8 I template e i membri static

Che cosa possiamo dire a proposito dei dati membro **static**? Ricordate che in una classe non di template, tutti gli oggetti della classe condividono una sola copia del dato membro **static**, e questo va inizializzato con scope a livello di file.

Ogni istanza di un template di classe ha la propria copia di ogni dato membro **static** e tutti gli oggetti della stessa istanza condividono lo stesso dato membro **static**. Come accade per i dati membro **static** delle classi non di template, questi dati devono essere inizializzati con scope a livello di file. Ogni istanza di un template di classe ha la propria copia delle funzioni membro **static** del template di classe.

Esercizi di autovalutazione

1.1 Indicate quali sono le affermazioni vere e quali quelle false. Per quest'ultime date una breve spiegazione.

- Una funzione **friend** di un template di funzione deve essere un'istanza di un template di funzione.
- Se da un solo template di classe si generano diverse istanze con un solo dato membro **static**, ognuna di esse condividerà una sola copia del dato membro del template di classe.
- Una istanza di template di funzione può subire overloading da parte di un'altra istanza con lo stesso nome.
- Il nome di un parametro formale può essere utilizzato una volta sola nella lista dei parametri formali della definizione di un template. I nomi dei parametri formali devono essere univoci in tutte le definizioni di template.
- La parola chiave **class** o **typename** utilizzate con un parametro di tipo di un template significano "qualsiasi tipo di classe definito dall'utente".

1.2 Completate le seguenti affermazioni:

- Le definizioni dei template di funzione devono cominciare con la parola chiave _____ seguita dalla lista dei parametri formali del template racchiusa tra _____.
- Le funzioni generate da un template di funzione hanno tutte lo stesso nome, per cui il compilatore utilizza i meccanismi di _____ per invocare la funzione appropriata.
- I template di classe sono anche detti tipi _____.
- L'operatore _____ viene utilizzato con il nome di un'istanza di template di classe per legare la definizione di ogni funzione membro allo scope del template di classe.
- Come accade per i dati membro statici di classi non di template, i dati membro **static** delle classi di template devono essere inizializzati con lo scope a livello di _____.

Risposte agli esercizi di autovalutazione

1.1 (a) Falsa. Può essere anche una funzione non di template. (b) Falsa. Ogni istanza di un template di classe avrà una sua copia del dato. (c) Vera. (d) Falsa. I nomi dei parametri formali non devono essere univoci su tutte le funzioni di template. (e) Falsa. La parola chiave **class** in questo contesto consente l'utilizzo di un parametro di tipo predefinito.

1.2 (a) template, parentesi angolari (< e >). (b) overloading. (c) parametrici. (d) di risoluzione dello scope. (e) file.

Esercizi

1.3 Scrivete il template di funzione **bubbleSort** sulla base del programma di ordinamento in Figura 5.15 del volume Fondamenti. Scrivete un programma di prova che riceve in input un array di **int** e un array di **float**, li ordini e li visualizzi.

1.4 Effettuate l'overloading del template di funzione **printArray** della Figura 1.2 in modo tale che possa ricevere come argomenti altri due interi, **lowSubscript** (indice iniziale) e **highSubscript** (indice finale). Chiamando questa funzione visualizzerete soltanto la porzione di array prescelta. Verificate che i valori di **lowSubscript** e **highSubscript** siano validi e in caso contrario restituite il valore 0. Altrimenti restituite il numero di elementi effettivamente visualizzati. Infine modificate **main** per utilizzare entrambe le versioni di **printArray** sugli array **a**, **b** e **c**. Cercate di verificare tutte le caratteristiche delle due versioni.

1.5 Effettuate l'overloading del template di funzione **printArray** della Figura 1.2 con una versione non template che visualizza in modo specifico un array di stringhe in formato tabulare.

1.6 Scrivete un semplice template di funzione per la funzione booleana **UgualeA** che confronta i due argomenti con l'operatore di uguaglianza (**==**) e restituisce true se sono uguali e false se non lo sono. Utilizzatelo, poi, in un programma che chiami questa funzione per diversi tipi predefiniti. In seguito scrivete una versione distinta di **UgualeA** per una classe definita dall'utente senza effettuare l'overloading dell'operatore di uguaglianza. Che cosa succede quando cercate di eseguire questo programma? Effettuate l'overloading dell'operatore. Che cosa succede adesso?

1.7 Utilizzate il parametro di non tipo **numberOfElements** e il parametro di tipo **elementType** per creare un template per la classe **Array**, che abbiamo sviluppato nel Capitolo 8 del volume C++ Fondamenti di programmazione. Questo template consentirà di istanziare oggetti **Array** con un numero di elementi specificato e di tipo specificato, in fase di compilazione.

1.8 Scrivete un programma che faccia uso del template di classe **Array**. Effettuate l'overriding del template con una definizione specifica di un **Array** di elementi **float** (**class Array< float >**). Il programma di prova deve contenere l'istanza di un **Array** di **int** definita tramite il template, e di un **Array** di **float** definita tramite la definizione fornita in **class Array< float >**.

1.9 Indicate la differenza tra le locuzioni template di funzione e istanza di template di funzione.

1.10 Che cosa assomiglia di più a uno stampino, un template di classe o un'istanza di un template di classe? Spiegate la vostra risposta.

1.11 Che relazione c'è tra template di funzione e overloading?

1.12 Per quale motivo si preferisce un template di funzione a una macro?

1.13 Che influenza ha sulle prestazioni l'utilizzo di template di funzione e di classe?

1.14 Il compilatore effettua dei confronti per determinare quale istanza di un template di funzione chiamare quando viene invocata una funzione. In quali circostanze si ha un errore durante questa operazione del compilatore?

1.15 Perché si dice che un template di classe è un tipo parametrico?

1.16 Spiegate in che situazione usereste questa istruzione in un programma:

```
Array< Employee > workerList( 100 );
```

1.17 Rileggete la vostra risposta all'Esercizio 1.16. Spiegate ora in che situazione usereste questa istruzione:

```
Array< Employee > workerList;
```

1.18 Spiegate il significato di questa notazione:

```
template< class T > Array< T >::Array( int s )
```

1.19 Per quale ragione si utilizza tipicamente un parametro non di tipo con un template di classe di un container?

1.20 Descrivete come effettuare l'overriding di un template di classe per un dato tipo di dato con una classe specifica per quel tipo di dato.

1.21 Descrivete che relazione c'è tra template di classe ed ereditarietà.

1.22 Supponiamo che un template di classe abbia questa intestazione:

```
template< class T1 > class C1
```

Descrivete la relazione **friend** che si stabilisce ponendo ognuna delle seguenti dichiarazioni in questa intestazione. Gli identificatori che iniziano per "f" sono funzioni, quelli che iniziano per "C" sono classi e quelli che iniziano per "T" possono rappresentare qualsiasi tipo di dato, sia esso un tipo di dato predefinito o una classe.

- a) friend void f1();
- b) friend void f2(C1< T1 > &);
- c) friend void C2::f4();
- d) friend void C3< T1 >::f5(C1< T1 > &);
- e) friend class C5;
- f) friend class C6< T1 >;

1.23 Supponiamo che il template di classe **Employee** contenga il dato membro statico **count** e supponiamo di istanziarne tre classi. Quante copie abbiamo creato del dato membro statico? Quali restrizioni si applicano su ogni copia, ammesso che ce ne siano?