DAL C AL C++

Chi possiede una discreta conoscenza del linguaggio C sa che la definizione di «linguaggio evoluto» non è poi così vera. I costrutti di cui dispone privilegiano soprattutto la velocità e l'efficienza, ma non rendono certo «piacevole» il lavoro di programmazione quando si tratta di implementare dei dati astratti (data abstraction). In questo senso, si può addirittura ritenere un linguaggio relativamente a «basso livello», poiché tale lavoro è lasciato quasi interamente all'abilità del programmatore. Molti lo apprezzano e lo preferiscono ad altri linguaggi più evoluti proprio per queste sue caratteristiche.

Ciò nonostante, l'esigenza di un linguaggio dotato della possibilità di una maggiore astrazione dai dati primitivi, che consentisse di definire nuovi tipi di dati astratti in modo molto semplice, adatto ad affrontare con facilità problemi di simulazione, ma dotato della stessa efficienza e portabilità del C, fu la molla della nascita del C++.

In questo capitolo saranno riepilogate le principali differenze nei costrutti comuni ai due linguaggi e illustrate le prime novità non Object Oriented introdotte dal C++.

10.1 Introduzione

Il *C++* fu originariamente ideato nel 1979 dal danese *Bjorne Stroustrup* allo scopo di rendere più efficienti alcuni programmi di simulazione scritti in *SIMULA67*.

Anziché sviluppare un linguaggio del tutto nuovo, Stroustrup scelse la strada di utilizzare come linguaggio base il *C*, per le sue affermate doti di versatilità, portabilità ed efficienza. Questa scelta avrebbe implicitamente offerto il vantaggio di rendere il nuovo linguaggio compatibile con un notevolissimo numero di programmi e librerie di funzioni già esistenti favorendone, altresì, la diffusione. Inoltre un numero notevolissimo di programmatori con esperienza di *C* non avrebbe dovuto imparare i fondamenti di un nuovo linguaggio di programmazione, ma solo le sue estensioni.

La prima versione di questo «*C* esteso» prese il nome di «*C* con classi», dove le classi costituiscono uno strumento linguistico, comune ad altri linguaggi precedenti, come *ADA* e *MODULA-2*, che consente quella maggiore «*data abstraction*» che era stata auspicata; ovvero la possibilità di astrarre nuovi tipi di dati che meglio potessero modellizzare oggetti reali.

Dopo alcuni anni di lavoro la nuova versione del «*C* con classi» aveva assorbito alcuni concetti fondamentali della programmazione orientata agli oggetti (***Objet Oriented Programming***).

Nel 1986 il «nuovo» linguaggio è ormai noto come *C++* e viene accolto nel mondo universitario e della ricerca con grande entusiasmo.

Durante gli anni '90 il *C++* diventa il linguaggio di programmazione dominante in applicazioni complesse e sofisticate e nei più disparati campi, da quello finanziario a quello delle telecomunicazioni e della progettazione assistita da computer (*CAD*).

Nato e sviluppato sotto il sistema operativo *UNIX*, il *C++* è attualmente utilizzato da milioni di programmatori un po' in tutti gli ambienti operativi, dai potentissimi supercomputer agli usuali *PC*. Numerosi sono i compilatori sviluppati sotto *DOS*, *Windows* e *Linux*.

Nel 1998 è stato approvato dagli istituti ***ISO/IEC*** (*International Standards Organization* e *International Engineering Consortium*) lo standard attuale, riconosciuto in tutto il mondo: ***International Standard ISO/IEC 14882-1998(E) Programming languages – C++***.

10.2 Stream standard di input e output

Il *C++*, al pari del *C*, non possiede parole chiave che permettono la comunicazione o l'acquisizione dei dati, tuttavia mette a disposizione diverse funzioni di libreria che consentono di assolvere questi compiti. In particolare, l'*header file* **IOSTREAM.H** contiene la definizione di due ***stream*** di I/O **cout** e **cin**, che consentono di interagire con gli *standard output* (video) e *input* (tastiera) in modo molto semplice.

10.2.1 cout

cout serve per inviare messaggi e dati sull'uscita standard. Il programma seguente, ad esempio:

```
#include <iostream.h>
void main()
{
  cout << "Il linguaggio C++" << endl;
}
```

invia la stringa `"Il linguaggio C++"` allo *stream standard* di uscita, tramite lo *stream* **cout**. Il **manipolatore endl** termina la linea introducendo il carattere **\n** e svuota il *buffer* di uscita. L'**operatore di inserzione** **<<** «scrivi su» indica la direzione del flusso di dati, in questo caso verso **cout** ([1]).

Tramite **cout** è possibile inviare all'uscita standard stringhe e dati numerici, contemporaneamente, separati fra loro dall'operatore **<<** ([2]):

```
#include <iostream.h>
void main()
{
  double Numero = 123.5;
  cout << "Il valore è: " << Numero << endl;
}
```

La stringa visualizzata sul video sarà:

```
Il valore è: 123.5
```

10.2.2 cin

In modo del tutto analogo è possibile leggere dei dati da tastiera. L'esempio seguente converte in °F una temperatura acquisita dallo *standard input* (tastiera) in °C, e ne invia il valore allo *standard output* (video):

```
#include <iostream.h>
double Temperatura;
```

([1]) Si noti che lo stesso operatore conserva ancora il significato di *shift* a sinistra per dati di tipo integrale; il compilatore è in grado di discriminarne il significato a seconda dell'operando di sinistra: se è uno *stream* avrà il significato di «scrivi su» altrimenti di *shift* a sinistra.

([2]) Per un *output* formattato *vedi* l'Approfondimento 3.8 *C++: controllo del formato dei dati con* `cout`.

```
void main()
{
    cout << "Temperatura in gradi centigradi: ";
    cin >> Temperatura;
    cout << "Temperatura in gradi Fahrenheith: "
         << 32 + Temperatura * 9 / 5 << endl;
}
```

In analogia con gli esempi precedenti, è semplice comprendere il funzionamento del programma. Lo *stream* **cin** consente di leggere i dati dall'ingresso standard, mentre **>>** indica la direzione del loro flusso, in questo caso verso la variabile `Temperatura`. Successivamente, tale valore è convertito in °F e visualizzato sul video.

In questo contesto **>>** rappresenta l'**operatore di estrazione** («leggi da»).

10.3 Commenti

Il *C++* supporta il formato dei commenti tipico del *C*, ma ne prevede anche un altro che inizia con la coppia di caratteri **//**. Un commento di questo tipo termina alla fine della riga in cui compare e, come nel caso precedente, è considerato uno «spazio bianco» dal compilatore.

I commenti che seguono, in *C++*, sono entrambi consentiti:

```
int Num1, Num2; // Dichiarazione di variabili intere

/*
Fine main
e
fine programma
*/
```

L'uso del doppio *slash* **//** può risultare molto utile per inserire brevi commenti a lato di un'espressione o per «mascherare» (nascondere) sia la sequenza **/*** sia ***/**; analogamente, il simbolo **/*** può mascherare un **//**.

10.4 Dichiarazione di variabili

Tutte le variabili di un programma devono essere dichiarate per poi essere utilizzate, ma in *C++* è possibile dichiararle ovunque. Ciò consente di dichiarare le variabili nel punto in cui servono, contribuendo a migliorare la leggibilità del programma.

Si esamini, ad esempio, questa istruzione `for`:

```
for (int Cont = 0; Cont < 10; Cont++)
```

Chi legge il programma non può avere dubbi: la variabile `Cont` è utilizzata in questo ciclo come contatore.

10.5 Tipi `bool`

Per rappresentare il valore di un'espressione condizionale nello standard del *C++* è stato introdotto il nuovo tipo primitivo **bool** che può avere solo due valori **true** e **false**.

La relazione tra `true` e `false` è la seguente:

```
!true == false
!false == true
```

I valori delle espressioni condizionali sono di tipo `bool`, tuttavia, per compatibilità con il *C* e il codice *C++* prodotto prima dell'avvento dello standard, è ancora possibile utilizzare valori integrali. Il compilatore li convertirà automaticamente nei corrispondenti valori *booleani*.

È possibile anche la conversione opposta: `false` diventerà `0` e `true` `1`.

A una variabile di tipo `bool` può essere applicato l'operatore postfisso o prefisso **++** (ma non **--**). In questo caso il risultato sarà sempre `true`, indipendentemente dal valore iniziale della variabile.

10.6 Tipi `struct`, `union` ed `enum`

In *C++* ogni definizione `struct`, `union` o `enum` diventa essa stessa un tipo (senza usare la parola chiave `typedef`) e il loro nome (*tag*) è utilizzabile, anche da solo, per dichiarare variabili dello stesso genere.

Volendo definire un nuovo tipo `enum` di nome `STATO`, è possibile scrivere semplicemente:

```
enum STATO {OFF, ON};
```

e successivamente dichiarare le variabili di questo tipo nel seguente modo:

```
STATO Lampeggio;
```

Analogamente, anche i *tag* di strutture e unioni sono considerati nomi di tipi e, nelle successive dichiarazioni, non è necessario specificare le parole chiave `struct` e `union`.

In *C++* le strutture e le unioni possono essere anonime, in altre parole dichiarate senza un identificatore, come nell'esempio seguente:

```
struct {
  char Descrizione[30];
  union {
    char   TabellaConversione[13];
    double FattoreConversione;
  };
  double CostanteTempo;
} Trasduttore;
```

L'accesso a un campo della `union` può essere fatto come a un qualsiasi altro membro della struttura. Ad esempio:

```
Trasduttore.FattoreConversione = 40.44e-6;
```

Diversamente dalle `union` (o `struct`) con nome, lo spazio di visibilità (*scope*) di ogni membro della `union` (o `struct`) anonima è quello del blocco che la contiene e, pertanto, i loro nomi devono essere distinti da quelli degli altri membri.

In ogni caso, né `struct`, né `union` anonime possono essere impiegate come argomenti o valori di ritorno di una funzione.

Diversamente dal *C*, in *C++* per assegnare un valore intero a un elemento di una enumerazione è necessario ricorrere a un *cast* di tipo. Ad esempio, se `Colore` è una variabile di tipo `enum SFONDO` e `Nero` è uno degli enumeratori, allora:

```
int i = Nero; /* Ammesso sia in C che in C++ */
Colore = 3;   /* Non ammesso in C++! La versione corretta è: */
Colore = (SFONDO)3;
```

10.7 Operatore di *scope resolution* ::

Se una variabile locale ha lo stesso nome di una variabile globale, all'interno di quel blocco, in *C*, non è possibile accedere all'omonima variabile globale. In *C++* questo problema è stato risolto introducendo l'operatore **::**, detto di ***scope resolution***. Anteposto al nome della variabile informa il compilatore che si fa esplicito riferimento a una **variabile globale**. Ad esempio:

```
#include <iostream.h>
int x = 0;          //Variabile globale x inizializzata a zero
void main()
{
  int x = 5;        //Variabile locale x inizializzata a 5
  ::x = 4;          //Assegna alla variabile globale x il valore 4
  cout << x << endl;
  cout << ::x << endl;
}
```

il programma visualizzerà sul video:

```
5
4
```

L'operatore di scope resolution permette di accedere ***solo*** *alla variabile* ***globale*** *omonima.*

Nonostante l'operatore di *scope resolution* consenta una certa flessibilità nell'assegnare i nomi alle variabili, si consiglia di non abusarne inutilmente: se due variabili hanno scopi diversi è bene distinguerle anche nel nome.

10.8 Qualificatore `const`

Il qualificatore **const** consente di dichiarare un oggetto come «non modificabile», eccetto al momento della sua inizializzazione.

In *C++* una variabile **const** è utilizzabile ovunque sia consentito l'uso di un'espressione costante. Il segmento di programma che segue, in *C++*, è perfettamente lecito:

```
const unsigned DIMENSIONE = 100;
double Vett[DIMENSIONE];
```

In questo caso, un compilatore *C* emetterebbe una segnalazione di errore, poiché non è permesso utilizzare una variabile, anche se qualificata come `const`, per specificare le dimensioni di un vettore.

In *C++*, invece, non è **assolutamente** permesso l'assegnamento di puntatori a oggetti `const` a puntatori a oggetti ordinari([3]). Ciò esclude qualsiasi possibilità che un oggetto dichiarato come costante possa essere modificato:

```
const double R2 = 1.41;          // R2 è un double costante
const double *pR2 = &R2;         /* pR2 è un puntatore a un double
                                    costante inizializzato con
                                    l'indirizzo di R2 */

double *pRadice2;
pRadice2 = pR2;                  // Errore
```

([3]) In *C* si otterrebbe, al più, solo un *warning* in fase di compilazione.

L'errore è dovuto al fatto che assegnando `pR2` a `pRadice`, quest'ultima potrebbe, mediante l'operazione di indirezione `*`, modificare il valore di `R2`, dichiarata costante, vanificando le intenzioni del programmatore.

I puntatori a oggetti costanti sono spesso usati nelle dichiarazioni di funzione. Ad esempio:

```
char *strcpy(char *Target, const char *Source);
```

In questo caso, entrambi gli argomenti sono passati per riferimento, se `Source` non fosse dichiarata costante, la funzione sarebbe libera di modificarla.

10.9 Puntatori a tipi `void`

I puntatori a un qualunque oggetto possono essere implicitamente convertiti in puntatori **`void`**([4]). In *C++*, diversamente dal *C*, la conversione di un puntatore `void` a un tipo diverso deve invece avvenire attraverso un **esplicito** *cast* di tipo.

Ad esempio, facendo riferimento alla funzione della *libreria standard* `malloc` che ritorna un puntatore `void`, le assegnazioni a un puntatore a `char` possono essere le seguenti:

```
char *Stringa1, *Stringa2;
Stringa1 = malloc(80);          /* Ammessa in C, illegale in C++ */
Stringa2 = (char *)malloc(80);/* Ammessa in C e C++ */
```

10.10 Linkage

In *C* e *C++* l'unità di compilazione è il file, ma un programma può essere costituito da numerosi file sorgenti, compilati separatamente e collegati assieme dal *linker*([5]).

Il ***linkage*** determina la porzione di programma nel quale un identificatore può essere referenziato (*scope*)([6]). Esistono tre tipi di *linkage*, interno, esterno e nessun *linkage*.

Se un identificatore è visibile in tutto il file sorgente in cui è dichiarato, e la sua dichiarazione contiene lo specificatore di classe di memorizzazione `static`, si dice che ha ***linkage* interno**, è visibile, cioè, solo all'interno di quel file sorgente, ma non altrove. Se **non** è dichiarato come `static` ha, invece, ***linkage* esterno** ed è referenziabile da quel punto in tutto il resto del programma.

Se la dichiarazione di un identificatore all'interno di un blocco non contiene lo specificatore di classe di memorizzazione `extern`, si dice che quell'identificatore non ha **nessun *linkage*** ed è visibile solo all'interno di quel blocco.

Osservazioni

Se si decide di implementare una libreria di funzioni, quelle esportabili (richiamabili da ogni punto di un programma) dovranno avere linkage esterno, mentre quelle realizzate per scopi interni, e quindi non esportabili, linkage interno.

([4]) *Il linguaggio C++* può essere considerato un'estensione del *C*, ma l'evoluzione dell'uno ha spesso influenzato gli aggiornamenti dell'altro. Ad esempio, il puntatore `void *` e la nuova sintassi dichiarativa delle funzioni, inseriti nel *C* standard, furono implementati dapprima in *C++*. Ne segue che molti programmi, scritti secondo lo standard *C*, sono ricompilabili in *C++* senza apportarvi sostanziali modifiche.

([5]) *Vedi* l'Approfondimento 6.1 *Struttura di un programma.*

([6]) *Vedi* l'Approfondimento 6.2 *Periodo di vita e visibilità.*

10.10.1 Lo spazio dei nomi: **namespace**

Gli identificatori con *linkage* esterno condividono un'area di memoria definita **spazio globale**. In progetti complessi, a cui lavorano molte persone che elaborano numerosi file, si potrebbero **definire** ([7]), involontariamente, identificatori con lo stesso nome nello spazio globale. Tali evenienze, meno improbabili di quanto s'immagini, creano, al momento di richiamare il *linker*, conflitti di *linkage* non facilmente risolvibili. In questi casi il prezzo da pagare è un controllo minuzioso dei programmi già elaborati o una maggiore attenzione in fase di definizione delle specifiche di progetto. Si consideri, ad esempio, il seguente caso:

```
// Header file name1.h
...
int x = 2;
...
// Header file name2.h
...
int x = 3;
...
// Source file main.cpp
#include "name1.h"
#include "name2.h"
...
```

Ci sono due identificatori `x` con *linkage* esterno **definiti** nello spazio globale. Al momento del *linking* dei tre file sarà segnalato un errore del tipo:

```
'x' : redefinition; multiple initialization
```

Il *C++ standard* offre l'opportunità di suddividere lo spazio globale in più parti, ognuna delle quali è definita come «**spazio dei nomi**» o **namespace**. In questo modo possono convivere identificatori con lo stesso nome, purché definiti in `namespace` differenti.

La sintassi di un `namespace` è la seguente:

namespace [[*identifier*]] **{***namespace-body***}**

dove *identifier* specifica il nome (opzionale) del `namespace` e *namespace-body* è una lista di dichiarazioni di dati e funzioni.

Il problema dell'esempio precedente può adesso essere risolto in questo modo:

```
// Header file name1.h
namespace primo {
    int x = 2;
}
// Header file name2.h
namespace secondo {
    int x = 3;
}
// Source file main.cpp
#include "name1.h"
#include "name2.h"
...
```

([7]) Se si vogliono condividere degli identificatori in più file sorgenti è necessario ripetere la loro dichiarazione, ma la definizione, se esiste, deve essere una sola in tutto il programma.

Se necessario, è possibile accedere alle singole variabili tramite l'operatore di *scope resolution* `::` e l'identificatore che qualifica il `namespace` di appartenenza:

```
#include "name1.h"
#include "name2.h"
#include <iostream.h>

void main()
{
  cout << "Somma = " << primo::x + secondo::x << endl;
}
```

il programma visualizzerà il messaggio:

```
Somma = 5
```

10.10.2 Il **namespace** standard **std**

Tutti gli identificatori del *C++ standard* sono inseriti entro lo «spazio dei nomi» chiamato **std**. Per questi motivi, un programma scritto in *C++ standard* assume una forma leggermente differente da quella ormai nota:

```
#include <iostream>

void main(void)
{
  std::cout << "Il linguaggio C++ standard" << std::endl;
}
```

La prima differenza importante, rispetto alla versione precedente, è la mancanza dell'estensione «**.h**» nell'*header* file. Lo standard, infatti, propone questa nuova forma abbreviata nell'utilizzo dei nomi per i propri *header* file per consentire la convivenza con i «vecchi» *header* file delle librerie *C* (*vedi* l'Approfondimento 12.1).

L'oggetto `cout` e il manipolatore `endl` sono preceduti dal prefisso **std** che qualifica a quale `namespace` appartengono.

10.10.3 La dichiarazione/direttiva **using**

Osservando il programma precedente non è tanto facile comprenderne i vantaggi, sembrerebbe anzi molto noioso dover qualificare, ogni volta, funzioni, oggetti, costanti o manipolatori di un particolare `namespace`.

La **dichiarazione using**, consente di riferirsi a un certo identificatore tratto da un `namespace` semplicemente mediante il suo nome:

```
#include <iostream>

using std::cout;
using std::endl;

void main()
{
  cout << "Il linguaggio C++ standard" << endl;
}
```

Certo, per programmi costituiti da un unico file sorgente o nel caso di alcuni identificatori di uso frequente o, in ogni caso, quando si è sicuri che non si possono verificare conflitti, risulterebbe molto più comodo usare direttamente gli identificatori, come se appartenessero a un unico `namespace`.

La **direttiva `using`** può fornire una soluzione al problema: inserita prima di un identificatore o di un `namespace` ne modifica lo *scope* (visibilità). Nel listato che segue questa direttiva consente di importare tutti gli identificatori del `namespace` standard `std` in quello globale:

```
#include <iostream>

using namespace std;

void main()
{
  cout << "Il linguaggio C++ standard<< endl;
}
```

Si elimina così la necessità della qualificazione, e questo è un vantaggio, congestionando però tale spazio con tutti i nomi dell'*header* file, e questo è uno svantaggio.

Attenzione, quindi, a non abusarne. In questo modo si vanifica la funzione dei `namespace`, rendendo nuovamente possibili i conflitti di nome che proprio si volevano evitare.

Osservazioni

Nel testo, quasi tutti i programmi impiegano la forma sconsigliata, ma ***solo per semplicità e brevità*** *degli esempi proposti.*

10.10.4 Specifiche di *link*

Più volte si è affermato che il *C* può considerarsi un sottoinsieme del *C++*, ma esistono delle differenze nel modo in cui i due linguaggi implementano il meccanismo di chiamata di funzione.

Questa incompatibilità può diventare un problema nel caso in cui, all'interno di un programma *C++*, sia necessario utilizzare delle funzioni appartenenti a una libreria *C* che non è possibile (o non si desidera) «portare» sotto *C++*.

Per ovviare a questo problema senza che sia necessario riconvertire tutto il codice *C* già esistente, è prevista una dichiarazione particolare che prende il nome di **specifica di *link***, la cui sintassi è:

`extern "C"` ([8]) [[**{** *elenco dichiarazioni* **}**]]

La dichiarazione **`extern "C"`** segnala al *C++* che ogni funzione dichiarata tra le parentesi graffe deve essere chiamata seguendo le convenzioni stabilite dal linguaggio *C*. Se le parentesi sono omesse, la specifica di *link* è applicata solo alle dichiarazioni presenti sulla stessa riga di programma dove appare `extern "C"`.

Si supponga, ad esempio, di voler utilizzare delle funzioni contenute in una libreria personalizzata *C* e i cui prototipi sono elencati nell'*header* file `UTIL.H`:

```
extern "C"
{
  #include "util.h"
}
...
```

([8]) In effetti, la specifica di *link* è stata ideata per consentire il *link* di file scritti in diversi linguaggi. Il nome *"C"* è supportato dal compilatore *Microsoft C++*, ma potrebbe essere differente in altre implementazioni.

`extern "C"` specifica che tutte le funzioni dichiarate in `UTIL.H` sono state compilate con un compilatore *C*.

La specifica di link non è necessaria per le funzioni appartenenti alla libreria standard del C.

Un altro caso in cui è necessario l'uso della specifica di *link* è quello in cui una funzione, contenuta all'interno di un file sorgente *C++*, è chiamata da una funzione *C*.

Si consideri, ad esempio, la funzione della *libreria standard* `qsort` che consente di ordinare gli elementi di un vettore: uno dei parametri di questa funzione è un puntatore a un'altra funzione che realizza il confronto tra due elementi. `qsort` è una funzione *C* e, di conseguenza, anche la funzione di confronto deve essere una funzione che soddisfa le specifiche di chiamata del linguaggio *C*.

Per comunicarlo al compilatore *C++* si usa ancora una volta la dichiarazione `extern "C"`, come nel file `EXTERNC.CPP`.

Prog. ExternC.cpp: Uso di funzioni di linguaggi differenti all'interno di un programma *C++*

```
#include <iostream>
#include <stdlib.h>

//* Prototipo di funzione dichiarata con le specifiche di link del C
extern "C" int Confronta(const void *a, const void *b);

void main()
{
  const int Numero = 5;
  int Dati[5] = {5, 4, 3, 2, 1};

  // Chiamata alla funzione C qsort che richiama Confronta
  qsort(Dati, Numero, sizeof(int), Confronta);

  // Visualizza gli elementi del vettore in ordine ascendente
  for (register i = 0; i < Numero; i++)
       std::cout << Dati[i] << std::endl;
}

// Funzione compilata secondo le specifiche di link del C
extern "C"
{
  int Confronta(const void *a, const void *b)
  {
    return *(int *)a - *(int *)b;
  }
}
```

`qsort` è dichiarata come funzione *C* nell'*header file* `STDLIB.H`.

In questo caso, `extern "C"` è richiesto sia per il prototipo sia per la definizione della funzione `Confronta`.

10.11 Prototipi di funzione

Il *C++* richiede **espressamente** che ogni chiamata a funzione sia preceduta dalla corrispondente dichiarazione o dal prototipo. Il prototipo di una funzione, come nel *C standard*, è una dichiarazione in cui sono stabiliti il nome, il tipo di ritorno della funzione e il numero e il tipo dei suoi eventuali parametri formali.

Questa informazione, inserita nel programma sorgente prima della chiamata della corrispondente funzione, consente al compilatore di controllare la coerenza del numero e del

tipo degli argomenti passati, anche se la definizione della funzione compare successivamente o non compare del tutto.

Per compatibilità con il *C standard*, la parola chiave **void** è utilizzabile al posto della lista dei parametri per specificare una funzione che non ha argomenti, ma non è obbligatoria.

In *C++* la sintassi di dichiarazione:

```
double f(void); è equivalente a double f();
```

10.12 Argomenti di *default* delle funzioni

In *C++* è possibile attribuire dei valori di *default* ai parametri formali di una funzione. Al momento della chiamata, se non vengono passati gli argomenti corrispondenti, la funzione assumerà come tali quelli di *default*.

Considerando la dichiarazione:

```
void Ordine(int Primo = 8, double Secondo = 7.1);
```

la chiamata della funzione, in base a quanto affermato, può assumere differenti forme, tutte ugualmente lecite:

```
Ordine();
Ordine(2);
Ordine(21, 56.1);
```

Nella prima forma, la totale mancanza di argomenti fa sì che il compilatore usi gli argomenti di *default*; ovvero la chiamata equivale a:

```
Ordine(8, 7.1);
```

Nella seconda forma, priva di un argomento, l'istruzione equivale a:

```
Ordine(2, 7.1);
```

La terza forma non ha bisogno di commenti ed è perfettamente conforme alla sintassi del *C*.

Osservazioni

Se nella lista dei parametri formali esistono degli argomenti di default, devono essere elencati per ultimi e di seguito uno all'altro. La seguente dichiarazione, ad esempio, è illegale:

```
void Ordine (int Primo =8, double Secondo);
```

La versione corretta è:

```
void Ordine(double Secondo, int Primo = 8);
```

Un argomento di default non può essere ridefinito in una dichiarazione successiva, anche se la ridefinizione è identica all'originale. Il seguente codice:

```
// Prototipo della funzione Ordina
void Ordine(double Secondo, int Primo = 8);
...
// Definizione della funzione Ordina
void Ordine(double Secondo, int Primo = 8)
{
  ...
}
```

verrebbe interpretato come un tentativo di ridefinire la stessa funzione e produrrebbe un errore in fase di compilazione. Pertanto, se gli argomenti di default sono definiti nel prototipo, non possono esserlo nella definizione della funzione o viceversa:

```
// Prototipo della funzione Ordina
void Ordine(double Secondo, int Primo);
...
// Definizione della funzione Ordina
void Ordine(double Secondo, int Primo = 8)
{
  ...
}
```

In generale, se in un programma esistono diverse dichiarazioni della stessa funzione, gli argomenti di default devono essere specificati solo una volta.

10.13 Funzioni `inline`

Ogni volta che il testo in una direttiva `#define` è costituito da un'espressione, l'identificatore associato è detto **macro**([9]) e la sostituzione operata dal preprocessore è definita **espansione della macro**. Sostanzialmente, il preprocessore effettua una sostituzione di testo ricopiando al posto delle macro le espressioni corrispondenti e rimpiazzando i parametri formali con gli argomenti attuali. Nonostante ciò possa presentare innegabili vantaggi, rispetto alle equivalenti chiamate di funzioni, comporta anche alcuni inconvenienti: il preprocessore non effettua nessun controllo sul tipo dei dati passati alla macro come argomenti attuali e in alcuni casi (con operatori di autoincremento) si possono verificare gravi errori.

Ad esempio, definendo la macro:

```
#define MIN(x, y) ((x) > (y) ? (y) : (x))
```

l'istruzione:

```
Minimo = MIN(Val1++, Val2++)
```

è espansa dal preprocessore come:

```
Minimo = ((Val1++) > (Val2++) ? (Val2++) : (Val1++));
```

In questo modo il valore minimo, qualunque esso sia, è incrementato due volte, effetto probabilmente non desiderato né previsto dal programmatore.

Per evitare questo inconveniente ed effettuare un corretto controllo del tipo dei dati passati come argomenti, senza tuttavia perdere i vantaggi delle macro, il *C++* consente di **qualificare** certe funzioni diversamente tramite una nuova parola chiave: **`inline`**. Il compilatore sostituisce l'istruzione relativa alla chiamata di una funzione `inline` con tutto il codice del corpo della funzione stessa. La tecnica è molto simile all'espansione di una macro.

Eliminato il meccanismo di chiamata attraverso lo *stack*([10]), le funzioni `inline` (se contengono poche istruzioni) consentono di incrementare la velocità di esecuzione e, molte volte, di ridurre anche la dimensione del programma eseguibile.

([9]) *Vedi* anche l'Approfondimento 6.5.

([10]) *Vedi* anche l'Approfondimento 6.4 *Chiamate di funzione*. In alcuni casi, per conservare la semantica delle chiamate di funzione, il compilatore fa ugualmente uso di variabili locali.

Mediante l'uso di una funzione `inline`, l'esempio precedente può essere più efficacemente sviluppato nel seguente modo:

```
inline double Min(double x, double y) { return x > y ? y : x; }
```

L'istruzione:

```
Minimo = Min(Val1++, Val2++);
```

sarà valutata senza produrre gli errori indesiderati della macro precedente, in quanto gli argomenti attuali passati alla funzione (`Val1` e `Val2`) verranno incrementati solo successivamente alla chiamata.

Le funzioni `inline` rappresentano pertanto uno strumento di lavoro molto utile ed efficiente e rendono superfluo l'uso delle macro.

Perché l'uso di tali funzioni sia effettivamente vantaggioso è necessario che il corpo della funzione contenga un numero molto limitato di istruzioni e che le chiamate a funzioni `inline` non siano numerose.

Osservazioni

Il compilatore considera il qualificatore `inline` *solo come un'indicazione. Esistono dei casi in cui potrebbe ignorarlo:*

— *la funzione contiene troppe istruzioni;*
— *il compilatore incontra una chiamata a una funzione* `inline` *prima della sua definizione e non può, quindi, effettuarne l'espansione.*

10.14 *Overloading* delle funzioni

Negli esercizi di programmazione proposti, affrontando il problema dell'ordinamento dei valori di un vettore, ci si è imbattuti più volte nell'algoritmo di scambio di due variabili.

Volendo strutturare tale algoritmo sotto forma di funzione, dovremmo necessariamente, in relazione al tipo di dati da scambiare (`char`, `int`, `float`,...), scrivere più funzioni con nomi diversi, ma con identica struttura logico-funzionale. Per lo stesso motivo, nella libreria matematica del *C* proliferano funzioni di tipo matematico adatte a elaborare dati di tipo diverso: `sin`, `sinl`, `cos`, `cosl`, `sqrt`, `sqrtl` e molte altre. Certo sarebbe molto più comodo chiamare tutte le funzioni che eseguono le stesse operazioni con un unico nome, lasciando poi al compilatore il compito di richiamare la funzione adeguata a trattare il tipo di dati.

La scelta della funzione corretta dovrebbe essere effettuata in base al tipo di argomenti passati al momento della chiamata: se sono dei `double`, la funzione chiamata dovrà essere in grado di effettuare le operazioni desiderate sui `double`; se sono di tipo `int` dovrà operare sugli interi, e così via. Il meccanismo appena descritto prende il nome di ***function overloading***.

Il *C++* consente l'*overloading* delle funzioni, comprese quelle della *libreria standard* e quelle `inline`, in altre parole dà la possibilità di fornire a funzioni con lo stesso nome diversi significati.

Il programma `MINIMO.CPP` effettua l'*overloading* di due funzioni che ritornano entrambe il minimo fra due numeri, indipendentemente dal loro tipo.

Prog. Minimo.cpp: Overloading di due funzioni che operano su tipi differenti

```
#include <iostream>
using namespace std;
/*
Overloading delle funzioni Minimo: la prima tratta dati int, la
seconda double
```

```
*/
int    Minimo(int x, int y);
double Minimo(double x, double y);

void main()
{
  int    a = 18123, b = 22456;
  double x = 3.387, y = 2.981;
  /* Anche se le funzioni chiamate hanno lo stesso nome
     sono distinte */
  cout << "Il minimo tra a e b è: " << Minimo(a, b) << endl;
  cout << "Il minimo tra x e y è: " << Minimo(x, y) << endl;
}

// Funzione che ritorna il minimo fra due int
int Minimo(int x, int y)
{
  return x > y ? y : x;
}

// Funzione che ritorna il minimo fra due double
double Minimo(double x, double y)
{
  return x > y ? y : x;
}
```

In base al tipo dei loro argomenti, il compilatore è in grado di distinguere quale delle due funzioni chiamare: prima `Minimo` che opera su dati di tipo `int` e, successivamente, `Minimo` che opera su `double`.

Non è possibile scrivere funzioni che differiscono unicamente per il tipo del valore di ritorno, perché in questo caso, considerati i criteri di scelta, il compilatore non potrebbe discernere quale funzione chiamare.

È interessante osservare che l'utente può ampliare il numero delle funzioni della *libreria standard* effettuandone l'*overloading* con versioni personalizzate.

Osservazioni

Se nelle dichiarazioni di funzioni che sfruttano l'overloading si impiegano degli argomenti di default, occorre prestare attenzione a non creare forme di ambiguità:

```
void Punto()
void Punto(double x = 0, double y = 0);
...
Punto();?
...
```

In questo caso il compilatore non saprebbe quale delle due funzioni chiamare e segnalerebbe un errore.

10.15 *Reference*

La maggior parte dei linguaggi ad alto livello dispone di due meccanismi fondamentali per il passaggio di argomenti a una funzione o procedura:

— **per valore**: la funzione riceve una copia degli argomenti;
— **per riferimento**: la funzione riceve l'indirizzo degli argomenti.

Il primo meccanismo garantisce una notevole immunità da eventuali errori (i valori originali non possono essere modificati neanche involontariamente), ma nel caso in cui gli argomenti attuali sono dati aggregati di notevoli dimensioni, può risultare una tecnica onerosa in termini di tempo necessario per effettuarne la copia sui corrispondenti parametri formali.

Il secondo meccanismo è adottato sia per risolvere il problema dei dati aggregati di grandi dimensioni, ma soprattutto per modificare i valori originali degli argomenti. Nel linguaggio *C*, come noto, il meccanismo del passaggio per riferimento è ottenuto mediante i puntatori. Questa scelta fu, originariamente, dettata da esigenze di semplificazione del linguaggio e del relativo compilatore, ma rimane un costrutto poco elegante che introduce inevitabili complicazioni di ordine logico e sintattico.

Il *C++* consente di superare i limiti di questa dicotomia obbligata, tipica del *C*, introducendo un nuovo modo per eseguire il passaggio dei parametri per riferimento: il ***reference***. Un *reference* è sempre un contenitore d'indirizzi, ma si può adoperare come una normale variabile, è un buon compromesso tra affidabilità e chiarezza di sintassi.

Il nuovo costrutto richiede l'uso dell'operatore **&** (*ampersand*). In questo esempio:

```
int   Dato;
int&  RefDato = Dato;
```

`RefDato` è dichiarata come un *reference* della variabile `Dato`.

Qui, l'operatore **&** non ha il significato di «indirizzo di», ma quello di «riferimento a». Il compilatore *C++* è in grado di distinguere il significato dell'operatore in relazione al contesto: se è preceduto da un tipo (fase di dichiarazione) è interpretato come *reference*; in tutti gli altri casi è considerato un operatore «indirizzo di».

Un *reference* non è una copia di una variabile, ma è la stessa variabile sotto un altro nome (***alias***) ovvero un sinonimo. Nell'esempio precedente, la dichiarazione ha lo scopo di informare il compilatore che la variabile `Dato` possiede anche un secondo nome: `RefDato`.

Qualunque operazione effettuata su un *reference*, si riflette sulla variabile alla quale fa riferimento e viceversa. Il programma `REF.CPP` ha lo scopo di dimostrare l'interscambiabilità di una variabile con il suo *reference*:

Prog. Ref.cpp: Dichiarazione e uso di una reference

```
#include <iostream>
using namespace std;
void main()
{
  int Dato = 5;
  int& RefDato = Dato;
  cout << ++Dato << endl;
  cout << RefDato << endl;
  cout << ++RefDato << endl;
  cout << Dato << endl;
  cout << &Dato << endl;
  cout << &RefDato << endl;
}
```

L'uscita prodotta è la seguente:

```
6
6
7
7
0x0C98
0x0C98
```

Si noti come coincidono sia i valori delle variabili `Dato` e `RefDato` sia i corrispondenti indirizzi.

Un *reference*, tranne alcune eccezioni, deve essere riferito a una variabile in fase di dichiarazione e, in seguito, non può essere modificato. In altre parole: dopo l'inizializzazione, un *reference* non può essere associato a una nuova variabile.

L'inizializzazione di un *reference* è omessa solo in questi casi:

— se è dichiarato con lo specificatore di classe di memorizzazione `extern`;
— se è un parametro formale o rappresenta il valore di ritorno di una funzione;
— se è un membro di una classe([11]).

In ognuno dei tre casi, l'inizializzazione avverrà ugualmente in un secondo momento.

In nessun caso un *reference* può essere inizializzato con un valore costante:

```
int& RefDato = 5;
```

Un *reference* può essere anche visto come un tipo particolare di puntatore, utilizzabile senza l'operatore d'indirezione * («contenuto della locazione di memoria puntata da»).

Ad esempio:

```
int Dato = 5;
int *const PuntDato = &Dato([12]); // Puntatore costante a Dato
```

Questa dichiarazione consente alla variabile `PuntDato` di riferirsi (indirettamente) alla stessa locazione di memoria di Dato: ogni assegnamento a `*PuntDato` si rifletterà sulla variabile `Dato` e viceversa. Nell'esempio precedente, un *reference* consente di ottenere lo stesso risultato senza che occorra utilizzare l'operatore *.

I puntatori, a differenza dei *reference*, sono distinti rispetto alle variabili alle quali puntano. Di conseguenza, è possibile eseguire delle operazioni logiche e aritmetiche sui primi mentre le stesse operazioni applicate a dei *reference* coinvolgono solo gli oggetti a quali si riferiscono.

Per analogia con i puntatori, è possibile utilizzare il qualificatore `const` per dichiarare dei riferimenti a costanti. Ad esempio:

```
int Dato = 5;
const int& RefDato = Dato // Riferimento ad un intero costante
```

Dopo questa dichiarazione, il valore di `Dato` è modificabile solo intervenendo direttamente sulla variabile `Dato` e non attraverso il suo *alias*, `RefDato`, attraverso il quale è solo possibile leggerne il valore.

Con dichiarazioni analoghe, è possibile passare grandi strutture di dati a una funzione con l'efficienza dei passaggi per riferimento e la stessa protezione dei dati offerta dai passaggi per valore.

Come sintesi, l'esempio `REFEFUNZ.CPP` prende in considerazione i tre meccanismi di passaggio di argomenti a una funzione.

Si consideri una struttura che descrive la pagina di un libro.

Prog. RefeFunz.cpp: Passaggio di parametri a una funzione per valori, puntatori e *reference*

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

// Dichiarazione del tipo PAGINA
struct PAGINA {
```

([11]) Per i concetti di classe *vedi* il capitolo 11.
([12]) È priva di senso, invece una dichiarazione come:
`int& const RefDato = Dato // Riferimento costante a un intero`
perché ogni *reference* è costante per definizione.

```
  unsigned NumeroPagina;
  char Testo[2000];
};
// Viene passata l'intera variabile (più di 2000 byte)
void ValDisplay(PAGINA Var);
// Viene passato l'indirizzo (puntatore) della variabile
void PtrDisplay(PAGINA *pVar);
// Viene passato un alias (reference) della variabile
void RefDisplay(PAGINA& RefVar);
void main()
{
  PAGINA Prefazione = {1, "La programmazione in C++ è ..."};
  ValDisplay(Prefazione);          // Chiamata per valore
  PtrDisplay(&Prefazione);         // Chiamata per puntatore
  RefDisplay(Prefazione);          // Chiamata per riferimento
}
void ValDisplay(PAGINA Var)
{
  cout << Var.NumeroPagina << endl;
  cout << Var.Testo << endl;
}
void PtrDisplay(PAGINA *pVar)
{
  cout << pVar->NumeroPagina << endl;
  cout << pVar->Testo << endl;
}
void RefDisplay(PAGINA& RefVar)
{
  cout << RefVar.NumeroPagina << endl;
  cout << RefVar.Testo << endl;
}
```

In entrambe le chiamate per riferimento, `PtrDisplay` e `RefDisplay`, l'argomento attuale è costituito dall'indirizzo della variabile passata, ma nella chiamata per *reference* non è richiesto l'uso dell'operatore **&**. Inoltre, poiché il parametro formale è un *alias* della struttura originaria, è possibile fare riferimento ai suoi membri come se la variabile fosse stata passata per valore.

Si noti che le chiamate di `ValDisplay` e quella di `RefDisplay` sono identiche. Se non è possibile esaminare il prototipo di queste funzioni ciò potrebbe erroneamente far pensare che la variabile `Prefazione` sia passata per valore e non possa, di conseguenza, essere modificata dalla funzione chiamata.

Per evitare ogni genere di equivoci, si consiglia pertanto di seguire alcune semplici regole che consentono di aumentare la leggibilità e la facilità di comprensione di un programma:

— se gli argomenti attuali sono dati di tipo fondamentale, da non modificare, conviene utilizzare il meccanismo del passaggio per valore;
— se gli argomenti sono dati aggregati complessi, da non modificare, conviene utilizzare il meccanismo del passaggio degli argomenti tramite un *reference* a una costante;
— se la funzione deve modificare gli argomenti attuali, di qualunque dimensione siano, si utilizzino i puntatori.

Per coerenza con queste regole, la funzione `RefDisplay` dell'esempio precedente andrebbe quindi modificata come segue:

```
void RefDisplay(const PAGINA& RefVar);
```

In questo modo, il valore cui fa riferimento `RefVar` **non può** essere modificato dalla funzione, così come se fosse stato passato per valore.

Osservazioni

Non è possibile dichiarare vettori di reference. Se si desidera passare un vettore a una funzione, il parametro può essere dichiarato come un reference a un vettore, come nel seguente esempio:

```
...
int Vett[100];
...
Stampa(Vett);
...
void Stampa(int (&v)[100])
{
  ...
}
```

Le parentesi consentono di dichiarare `v` *come un reference a un vettore di* `100 int`. *Se si omettessero,* `v` *assumerebbe il significato di un vettore di* `100` *reference a* `int` *e pertanto porterebbe a una segnalazione di errore da parte del compilatore* ([13]).

10.16 Funzioni che ritornano un *reference*

In *C++* si può utilizzare un *reference* come valore di ritorno di una funzione. Come per il passaggio di *reference*, tale scelta si dimostra particolarmente vantaggiosa nel caso in cui i valori da ritornare siano costituiti da variabili di grosse dimensioni.

Poiché una funzione che torna un *reference* rappresenta, di fatto, una variabile (quella «ritornata»), è possibile assegnargli anche un valore. Il meccanismo è insolito, ma adottato a livello professionale da molti programmatori. Nel seguente esempio, la funzione `ReadWritePageNumber` consente, con la **stessa** funzione, di leggere e impostare il membro `NumeroPagina` della struttura `PAGINA` dell'esempio precedente:

```
...
unsigned& ReadWritePageNumber(PAGINA& p1)
{
  return p1.NumeroPagina;
}
cout << ReadWritePageNumber(Prefazione) << endl;
ReadWritePageNumber(Prefazione) = 1;
...
```

La prima chiamata alla funzione legge il numero di pagina della struttura `Prefazione`, la seconda l'imposta a 1.

Osservazioni

In questo caso l'esempio funziona correttamente, perché anche il parametro della funzione è un reference e quindi può essere modificato. In caso contrario, il valore assegnato alla funzione modificherebbe solo il parametro nell'espressione `return`, *eliminato automaticamente al termine dell'esecuzione della funzione.*

([13]) Si noti che è necessario specificare anche la dimensione che deve coincidere con quella del vettore passato.