Sistemi Intelligenti Naturali Artificiali

Corso di programmazione C/C++

Lorenzo Natale (nat@liralalab.it)
DIST, Università di Genova

Matteo Brunettini (matteo@liralab.it)
DIST, Università di Genova

INDICE

Introduzione

Il linguaggio C nasce negli anni settanta come linguaggio di sviluppo di Unix.
Caratteristiche:

- è un linguaggio di medio livello: permette di scrivere codice efficiente, in grado di accedere all'hardware in maniera rapida e flessibile
- è standard: il codice scritto in C può essere facilmente compilato su piattaforme diverse
- largamente diffuso: è molto facile trovare programmi scritti da altri e riutilizzarli

Svantaggi:

- è facile scrivere, per errore, programmi che portano al crash di sistema o del programma stesso
- permette di scrivere programmi molto compatti ma anche di difficile comprensione

Il C++ include ed estende il C.

Commenti

E' possibile inserire commenti all'interno del codice. I commenti vengono eliminati automaticamente al momento della compilazione, ma servono per spiegare parti di codice ad altri (dove per "altri" si intende anche se stessi qualche mese dopo).

In C, un commento può estendersi per diverse righe, inizia con /* e termina con */.
In C++ un commento può essere inserito semplicemente con // ma non può essere più lungo di una riga.

Esempio:

```
/* questo è un
commento in C */
// questo è un commento C++
```

Tipi di dato

Il C definisce 5 tipi di dato diversi e quattro modificatori.
1) Tipi:

- *void* : non-valore, utilizzato nelle dichiarazioni di funzioni e per definire puntatori generici (il suo utilizzo sarà più chiaro nel seguito)
- *char* : variabile di tipo intero, può assumere valori tra -128 e 127 (1 byte) e viene utilizzata per rappresentare caratteri con la codifica ASCII
- *int* : variabile di tipo intero, permette di rappresentare numeri interi con segno. Il tipo di codifica utilizzata per rappresentare numeri negativi dipende dall'architettura *hardware* della macchina che si utilizza (spesso complemento a due). Il numero di bit che costituiscono un *int* dipende ancora dall'*hardware* e dal compilatore. Sia 'n' tale valore. Un *int* può contenere

numeri interi nell'intervallo compreso tra -2^{n-1} a $2^{n-1}-1$ (un bit viene utilizzato per il segno).
- *float* : numeri in virgola mobile, singola precisione (6 cifre decimali)
- *double* : numeri in virgola mobile, doppia precisione (10 cifre decimali)

Nota sul tipo *char*: dal nome sembra che il tipo *char* rappresenti caratteri. Questo fatto può essere fuorviante. Si ricordi che all'interno di un calcolatore è possibile rappresentare solamente numeri. Il tipo *char*, infatti, non è altro che un numero (appunto tra -128 e 127), che in casi particolari viene convertito in un carattere utilizzando una opportuna tabella di conversione (ASCII). Questa conversione avviene, per esempio, quando si esegue dell'io su console o su file (in modalità testo) oppure nel caso in cui si decida di farlo esplicitamente all'interno del codice.
Es:

```
char a;
a = 10;   // corretto,' a' vale 10
a= 'c';   // 'a' è esattamente uguale al valore corrispodente,
          // nella tabella
          // ASCII, al carattere 'c'
```

Infatti a = '0' e a = 0 sono diversi in quanto il primo assegna il valore ASCII corrispondente a al carattere '0' (in decimale 48 oppure in esadecimale 0x30) alla variabile a, mentre ll secondo vi assegna il valore ASCII 0

2) Modificatori:
unsigned, che può essere applicato a *int* o *char* per ottenere:
- *unsigned int*: numeri interi positivi. In questo caso il range di numeri rappresentabili va da 0 a 2^n-1 poiché nessun bit è utilizzato per il segno.
- *unsigned char*: numeri interi positivi, tra 0 e 255 (tutti e otto i bit sono utilizzati per rappresentare il numero).

short, *long* applicabili a *int*. La specifica del C impone solamente le seguenti condizioni:
#bit short int ≥ 16
#bit long int ≥ 32
#bit short int $\leq$ *#bit int* $\leq$ *#bit long int*

short int e *long int* possono essere abbreviati rispettivamente con *short* e *long*.

Modificatore di accesso *const*, serve per specificare una variabile il cui valore non sarà mai cambiato. Se utilizzato con puntatori, impedisce la modifica delle variabili *puntate* dal puntatore.

Esempio:
Nel caso del compilatore Microsoft Visual C++:
int = 32 bit, nel range $\left[-2^{31}, 2^{31}-1\right]$ *ossia* $\left[-2147483648, 2147483647\right]$
short int = 16 bit, tra $\left[-2^{15}, 2^{15}-1\right]$ *ossia* $\left[-32768, 32767\right]$

long int = 32 bit (si veda int)

float = 32 bit
double = 64 bit

3) Il C++ aggiunge i tipi:

- *bool*: variabile di tipo booleano, può assumere valori *true* o *false*
- *wchar_t*: 16 bit, caratteri estesi, utilizzato per rappresentare caratteri di lingue particolari per le quali non bastano 256 valori.

Gli header *limits.h* e *float.h* definiscono costanti simboliche relative alle ampiezze dei tipi fondamentali. Es: *INT_MAX*, *INT_MIN* relativamente il valore massimo e minimo di una variabile di tipo *int*, FLT_MAX, FLT_MIN per il tipo *float* ecc…
In C++ le stesse costanti si trovano definite nell'header <*climits*> .

Operatore sizeof()

Per rendere il codice portabile è opportuno utilizzare l'operatore *sizeof(tipo)* il quale restituisce la dimensione in # byte del tipo di dato specificato tra parentesi.
Esempio:

```
a = sizeof(int);    // ritorna il numero di byte occupati da una
                    // variabile int
b = sizeof(bool);   // ritorna il numero di byte occupati da una
                    // variabile di tipo bool
```

L'operatore può essere utilizzato anche con variabili.
Esempio:

```
int c;
a = sizeof(c); // ritorna il numero di byte occupati da una
               // variabile int,
               // nel caso MS Visual C++, tale valore è 4
```

Dichiarazione di variabili

Prima di poter utilizzare una variabile è obbligatorio dichiararla. Per dichiarare una variabile occorre scrivere il tipo della variabile seguita dal nome. Nota: non tutti i caratteri sono ammessi all'interno del nome di una variabile, consultare un libro.
Es:

```
int a;          // a è una variabile di tipo int
char v;         // v è una variabile di tipo char
short s;        // s è una variabile di tipo short int
```

Una variabile dichiarata ma non inizializzata assume un valore non prevedibile (benchè certi compilatori inizializzino automaticamente le variabili a zero).

```
int a = 10;     // dichiara che a è una variabile di tipo int e
                // inizializza il suo
                // valore a 10.
char c = 'a';   // dichiara che c è una variabile char a la
                // inizializza al valore
                // corrispondente alla costante 'a'
```

Le variabili di tipo *const* vanno obbligatoriamente inizializzate (ovviamente, visto che non è più possibile cambiarne il valore).

```
const int costante = 900;
```

Costanti

Per costante si intende qualunque valore all'interno del codice che non sia una variabile.
Es:

```
int a = 10;    // 10 qui è una costante intera
float f= 10.0; // 10.0 qui è una costante di tipo floating point
char c = 'h';  // 'h' è una costante char
```

Per scrivere un valore esadecimale è sufficiente far precedere al numero il prefisso 0x, mentre nel caso in cui si voglia scrivere un valore ottale occore utilizzare il prefisso 0.
Es:

```
int a = 0xFF;  // a vale '255'
int b = 010;        // 10 in ottale, ossia 8 in decimale
```

Sequenze di escape: sono costanti che rappresentano caratteri particolari:
'\n': new line, sposta il cursore all'inizio di una nuova riga
'\t': tabulazione, inserisce una tabulazione
'\0': carattere nullo, rappresenta un byte avente tutti i bit a zero

Notare che i precedenti caratteri speciali sono solamente simboli utilizzati per convensione dal linguaggio e internamente sono rappresentati dal computer per mezzo di un solo byte.

Operatore di assegnamento

L'operatore '=' permette di copiare il valore di una variabile all'interno di un'altra.
Es:

```
int a = 10;
int b = 0;
b = a;         // b vale ora 10
```

Occorre fare attenzione quando le variabili sono di tipo diverso.
Es:

```
char c = 30;
int b = 10;

c = b;         // ??
```

Il compilatore, quando trova tipi di dato diversi in un'operazione di assegnamento, esegue una *conversione di tipo*. Il valore della variabile di destra viene convertito in un valore compatibile col tipo della variabile di sinistra. Questo è possibile solamente nel caso in cui il tipo della variabile di destra sia in grado di contenere una quantità di informazione superiore (spesso se è costituita da un numero di bit maggiore). In caso

contrario la conversione avviene ugualmente ma può portare ad una perdita di dati (il compilatore avverte con un messaggio di *warning*).

Es:
```
char c = 10;
int i = 30;
float f = 2.0;
double d = 3.0;

i = c;    // nessun problema un intero può contenere qualunque
          // char, i vale ora 10
c = i;    // warning: possibile loss of data (non è detto che un
          //char possa
          // contenere un int)
d = i;    // nessun problema, un double può contenere qualunque
          // intero, d vale ora 30.0
d = f;    // nessun problem un double può contenere qualunque
          // float, d vale ora 2.0

f = i;    // corretto o non corretto. Dipende dall'implementazione.
```

Operatori aritmetici

'+', '-', '*', '/'
Si spiegano da soli, vedere un qualunque libro.

'++' e '–' incremento e decremento.
Es:
```
int a = 0;
a++;      // a vale 1
a--;      // a vale nuovamente 0
```

C'è differenza tra l'uso degli operatori '++' e '–' cosiddetto postfisso e prefisso. Nel caso prefisso l'incremento della variabile avviene prima dell'esecuzione dell'istruzione che la contiene. Viceversa, nel caso postfisso, l'istruzione viene eseguita e solo successivamente la variabile è incrementata.
Es:
```
int a = 10;
int b = 0;
b = ++a;  // prefisso, b vale 11, a vale 11
b = a++;  // postfisso, b vale sempre 11 a vale 12
```

Stessa cosa per l'operatore '—'.

Operatore '+='.
Es:
```
a += b;   // coincide con a = a + b;
a += 4;   // coincide con a= a + 4;
```
Analogamente per '-=', '/=', '*='.

Operatori relazionali:
'<', '>', '>=', '<=', '==' rispettivamente minore, maggiore, minoreuguale, maggioreuguale, uguale

Operatori logici:
'&&', '||', '!' rispettivamente and, or e not

Da non confondere con gli operatori bit a bit '&', '|' per i quali si rimanda ad un libro.

Promozione di tipo nella valutazione delle espressioni

Si consideri il seguente esempio:

```
char c = 0;
int i = 0;
float f = 0.0;
double d = 0.0;
double res = 0;

res = (c*i) + (f+i) + (f*d);  // espressione che coinvolge
                              // variabili di tipo differente
```

Tutte le volte che il compilatore incontra un'espressione esegue la cosidetta "promozione di tipo". Le variabili sono convertite al tipo "più grande" tra quelli che figurano nell'espressione; la conversione è esattamente uguale a quella già descritta nel caso dell'operatore di assegnamento.

Nel caso particolare:

```
res = (c*i) + (f+i) + (f*d);  // tutte le variabili sono promosse
                              // (e convertite) a double
```

Conversione cast

E' possibile forzare il compilatore ad eseguire una conversione di tipo (*cast*). La forma generale è la seguente:
(tipo) variabile

Es:

```
int b;
float a;
a = (float) b;
```

In alcuni casi la conversione *cast* è di particolare utilità.
Si consideri il seguente esempio, nel quale si vuole stampare la percentuale di avanzamento di un contatore:

```
const int max = 1550;
for (int i = 1; i<=max; i++)
{
     double p;
     p = i/max * 100;    //calcola la percentuale di i rispetto a
                         //max
```

```
     printf("Percentuale: %lf\n", p);   // stampa su schermo il
                                        // valore di p
}
```

Se si prova ad eseguire il codice precedente non si ottiene il risultato desiderato ma una riga di zeri seguita dal valore 100 alla fine. Questo accade poiché la divisione tra 'i' e 'max' avviene tra numeri interi (la conversione a *double*, dovuta all'operatore '=' sarà eseguita solo dopo la moltiplicazione x 100) e come tale produce un risultato che viene troncato della parte decimale. Finché 'i' rimane minore di 'max' il valore inserito in 'p' è uguale a 0; l'ultima iterazione del ciclo *for*, quando i == max, produce 'p = 100'.

Per ovviare al precedente inconveniente è possibile sfruttare la promozione di tipo nelle espressioni. E' sufficiente modificare:

```
p = (double) i/max * 100;
```

In questo caso si dice esplicitamente al compilatore di eseguire una conversione di tipo; 'i' viene convertito in un valore *double*, di conseguenza anche 'max' e 100 vengono convertiti a dei valori *double* (rispettivamente 1550.0 e 100.0). La divisione tra variabili *floating point* produce ora il risultato desiderato.

Considerare anche il seguente esempio:

```
int a = 10;
double d;
d = a/20;      // d vale 0
d = (double) a/20;  // d vale 0.5
```

Nel caso in cui si abbia a che fare con delle costanti è sufficiente specificare la costante nel seguente modo:

```
d = a/20.0;    // qui il secondo membro della divisione è una
               // variabile di tipo double
```

Controllo di flusso

Nota: si ricorda che qualunque espressione in C viene considerata vera se è diversa da zero e viceversa.

```
if (espressione)
     istruzione_1
else
     istruzione_2
```

dove la parte relativa ad "else" è opzionale. L'*espressione* viene valutata; se ha un valore non nullo (ossia se è vera) si esegue *istruzione_1*, altrimenti *istruzione_2*.

```
while (espressione)
     istruzione
```

Il ciclo continua ad eseguire *istruzione*, e termina quando *espressione* è uguale a zero (cioè quando è falsa).

```
do istruzione
while (condizione)
```

Molto simile al *while*; in questo caso il controllo della condizione di uscita avviene dopo l'esecuzione del ciclo.

```
for(esp_1; esp_2; esp_3)
     istruzione
```

In genere *esp_1* è un'istruzione di inizializzazione, *esp_2* è la condizione di uscita del ciclo, *esp_3* è un'espressione che viene eseguita al termine del ciclo stesso. *istruzione* viene eseguita finché *esp_2* non diventa falsa.

In tutti i casi precedenti, *istruzione* può essere una singola istruzione terminata da ";" oppure un blocco di istruzioni.

Espressioni condizionali:
```
espr_1 ? espr_2 : espr_3
```

Si consideri il seguente esempio:
```
z = (a>b) ? a:b;
```
La variabile z assume il valore di a nel caso in cui a sia maggiore di b; in caso contrario (se a < b) z viene posta uguale a b. La precedente espressione è esattamente uguale al seguente *if*:
```
if (a>b)
     z = a;
else
     z = b;
```

In realtà l'utilizzo del precedente costrutto può essere più generale; si rimanda a qualunque libro di C per una trattazione più completa.

Istruzione break

Permette di terminare l'esecuzione di un loop in maniera forzata.
Es:
```
for(int i = 1; i < 100; i++)
{
     printf("%d\n", i);
     if (i == 10)
          break;
}
```
In questo esempio vengono eseguite solo 10 iterazioni del ciclo *for*. L'esecuzione del loop viene interrotta quando *i* è uguale a 10.
Si consideri anche il seguente esempio:
```
while (true)
{
     cout << "Inserire un carattere e premere invio ('e' per
uscire)\n";
     char c;
```

```
     cin >> c;
     if (c == 'e')
        break;
     cout << "Hai premuto il tasto: " << c << "\n";
}
```

Blocco di istruzioni

Un blocco di istruzioni è costituito da un insieme di istruzioni racchiuse da parentesi graffe {}.
Ogni blocco di istruzioni definisce un nuovo *scope*. Variabili dichiarate all'interno di un blocco di istruzioni sono valide solamente dentro a tale blocco e svaniscono alla fine del blocco stesso. Blocchi interni vedono le stesse variabili del blocco esterno più quelle eventualmente definite localmente all'interno del blocco; eventuali variabili locali al blocco nascondono variabili più esterne aventi lo stesso nome.
Si consideri il seguente esempio.

```
int main()
{
     int a = 0;
     int b = 0;
     {    // nuovo blocco di istruzioni
          int a = 10;    // questa variabile nasconde 'a' definita
                         // all'esterno
          int c;         // questa variabile esiste solo in questo
                         // blocco
          a++;      // a vale ora 11
          b++;      // b è quella definita nel main e vale ora 1
     }
     a++;
     // a vale 1
     // b vale 1
     // c non esiste più
}
```

Struttura di un programma C

Un programma C è costituito dalle seguenti sezioni:

```
// direttive preprocessore, #define, #include

// dichiarazione di variabili globali

// prototipi delle funzioni, se ne esistono
tipo_1 funzione1 (parametri_1);
tipo_2 funzione2 (parametri_2);

// funzione main, deve esisterne almeno una
int main()
{
     // dichiarazione variabili locali
     // codice
}
```

```
// implementazione delle funzioni
tipo_1 funzione1 (parametri_1)
{
     // dichiarazione variabili locali
     // codice
}
tipo_2 funzione2 (parametri_2)
{
     // dichiarazione variabili locali
     // codice
}
```

Preprocessore

Il preprocessore costituisce la prima fase della compilazione. Le due funzionalità di uso più frequente sono *#include* e *#define*.
L'istruzione *#include "nome_file" oppure #include <nome_file>* viene sostituita dal preprocessore con il contenuto del file *nome_file*, il quale può contenere dichiarazioni di funzioni (es. per l'uso di librerie) oppure ulteriori *#define*. C'è una lieve differenza tra i due casi *<nome_file>* e *"nome_file",* quale ?
La definizione nella forma *#define nome testo* dice al preprocessore di sostituire tutte le occorrenze di *nome* con *testo.*
Es:

```
#define MAX 500     //  la parola MAX all'interno del programma
                    // sarà sostituita con 500
```

Nota: è preferibile l'uso del modificatore *const* per evitare quest'uso di *#define*. In questo modo si definisce il tipo della variabile.
Es:

```
const int MAX = 500;
```

L'istruzione *#define* è spesso utilizzata per definire macro come la seguente:

```
#define errore printf("errore")    // la parola errore viene
                                   // sostituita dall'istruzione
                                   // printf("errore")
```

oppure:

```
#define max(A,B)    (A) > (B) ? (A) : (B)
```

Funzioni

Spesso è possibile, e opportuno, dividere il codice presente nel *main* in più funzioni. Questo permette di organizzare il codice in maniera maggiormente leggibile e di evitare di riscrivere parti di codice di utilizzo comune. (es: abbiamo già usato in precedenza la funzione *printf()*, appartenente alla libreria standard del C).

Dichiarazione:
`tipo_ritornato nome_funzione(parametri);`
Specifica al compilatore il nome della funzione, il tipo dei parametri e il tipo della variabile di ritorno. La dichiarazione di una funzione deve necessariamente precedere il suo utilizzo.

Dopo la dichiarazione della funzione occorre scrivere l'implementazione (questa può anche trovarsi su un file separato).
Implementazione:

```
tipo_ritornato nome_funzione(parametri)
{
     // codice
}
```

Di solito il prototipo di una funzione viene inserita in un file .h mentre l'implementazione va in un file .cpp (o .c). Qualora si voglia utilizzare una funzione è necessario includere il file .h che ne contiene il prototipo e compilare/"linkare" il relativo .cpp.

Puntatori

Un puntatore è una variabile che contiene l'indirizzo di un'altra variabile.
Es:

```
char *p;  // p è una variabile che può contenere l'indirizzo di una
          // variabile di tipo char
int *i;   // i è una variabile che può contenere l'indirizzo di una
          // variabile di tipo int
```

Gli operatori che vengono utilizzati quando si parla di puntatori sono i seguenti:

- "*" → accede alla variabile puntata dal puntatore
- "&" → restituisce l'indirizzo di una variabile

Es:

```
int i;
int y;
int *p;
i = 10;
p = &i;   // p ora punta a i
y = (*p); // y è uguale a i (ossia 10)
```

Organizzazione della memoria

In molti casi un programmatore non deve curarsi dei dettagli di come è organizzata la memoria; questo per due fondamentali ragioni:

1) la gestione della memoria può essere un'operazione complessa che può portare facilmente a errori
2) l'organizzazione della memoria varia al variare della piattaforma (calcolatore, sistema operativo)

Tuttavia è utile avere almeno un'idea di come è organizzata la memoria.
Qualunque variabile all'interno di un programma occupa una ben precisa porzione della memoria; la sua posizione può essere determinata attraverso *l'indirizzo*. Un indirizzo è un numero che identifica in modo univoco una cella di memoria. Il numero di bit utilizzati per rappresentare un indirizzo definisce la quantità massima di memoria che può essere gestita all'interno di un programma; tale valore dipende da fattori quali l'hardware della macchina e il sistema operativo. Molto frequentemente un indirizzo è costituito da 32 bit, valore che permette di indirizzare fino a 2^{32} celle di memoria (4 GB). Si ricorda che tale valore rappresenta il *massimo* numero di celle di

memoria che possono essere gestite dal programma (ossia la sua massima memoria *virtuale*); tale valore è sicuramente inferiore alla quantità di memoria effettivamente presente sul calcolatore (si rimanda ad un testo di sistemi operativi per i dettagli – es. per l'architettura Windows si veda *"Advanced Windows"*, J.Richter, Microsoft Press, 3rd ed.).

La memoria può essere vista come un foglio diviso in celle da 8 bit ciascuna, ognuna associata ad un numero (il suo indirizzo), ossia:

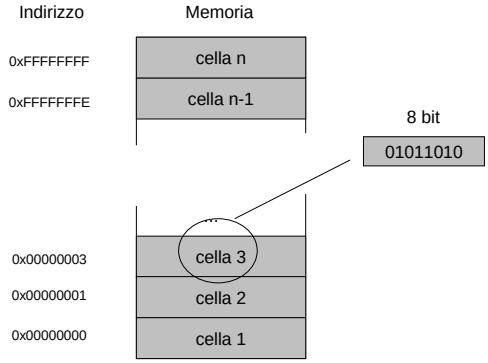


Figura 1

dove si suppone che un indirizzo sia costuito da 32 bit. Per praticità gli indirizzi sono stati scritti in notazione esadecimale. Si consideri questa rappresentazione solo a titolo di esempio. Nella realtà infatti le cose sono leggermente diverse, poichè alcune parti della memoria sono riservate e non direttamente accessibili dal programma.
Ecco come potrebbe essere la memoria durante l'esecuzione del seguente programma (si suppongono interi a 32 bit):

```
char c;
int i;
int *pi;
```

Si noti che le variabili non sono state inizializzate e contengono quindi valori non definiti.

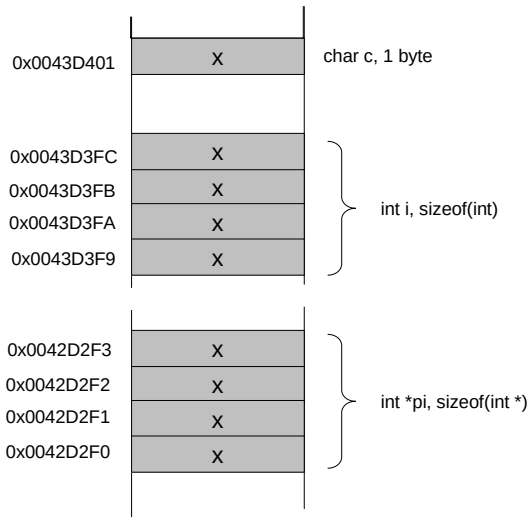


Figura 2

ed ecco come risulta la stessa zona di memoria dopo le seguenti istruzioni:

```
c = 200;
i = 350;
pi = &i;
```

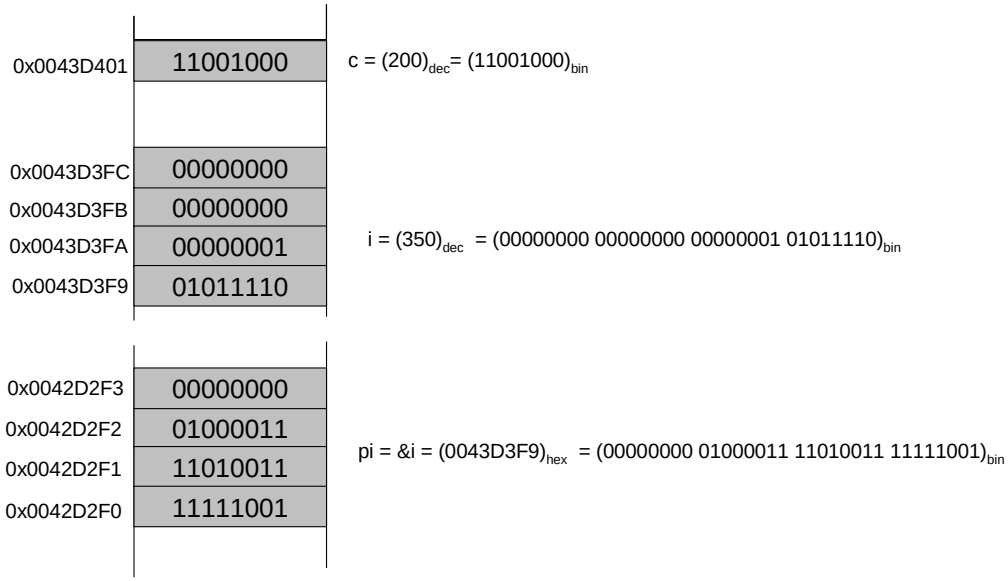


Figura 3

Nota: nell'esempio precedente (e nel seguito) si suppone che la notazione utilizzata per rappresentare gli interi è quella cosiddetta *little-endian* dove il primo byte ad esser inserito in memoria (byte avente indirizzo minore) è quello meno significativo. Questa notazione viene utilizzata dai microprocessori Intel e si distingue da quella *big-endian* che prevede il contrario.

Notare che il puntatore all'intero 'i' contiene solamente l'indirizzo del *primo* byte che costituisce 'i'. Risulta pertanto ovvio che i puntatori, indipendentemente dal tipo della variabile a cui puntano, occupano sempre la stessa quantità di memoria; tale quantità di memoria è esattamente uguale a quella necessaria a contenere un indirizzo (nel nostro caso 32 bit). A questo punto potrebbe sorgere la seguente domanda: a cosa serve conoscere il tipo della variabile puntata dal puntatore ?
La prima risposta è che il tipo della variabile puntata dal puntatore serve al compilatore per controllare che le operazioni eseguite sulla variabile "puntata" siano consistenti col tipo stesso. Ad esempio:

```
char a,c;
int i;
char *pc;
pc = &c;
a = *pc;       // corretto *pc e a sono dello stesso tipo
*pc = i;       // "warning" possibile perdita di informazione, i e
               // *pc non sono dello stesso tipo
```

La seconda risposta alla precedente domanda è l'algebra dei puntatori.

Algebra dei puntatori

Noto il tipo di puntatore il compilatore è in grado di incrementare un puntatore in modo che punti alla variabile successiva.
Es:

```
char *pc; // puntatore a una variabile di tipo char
int *pi;  // puntatore a una variabile di tipo int
int i
char c;
pi = &i;
pc = &c;
pc = pc+1;// incrementa pc in modo che punti al char
          // immediatamente successivo (ossia pc viene incrementato
          // di 1)
pi = pi+1;// incrementa pi in modo che punti all'intero
          // immediatamente successivo (ossia pi viene incrementato
          // di sizeof(int), nel nostro caso 4)
```

Nell'esempio di Figura 3 'pc' e 'pi' dopo l'incremento sarebbero rispettivamente 0x0043D402 e 0x0042D3FD.

In altre parole, l'operazione d'incremento di un puntatore si comporta in modo diverso al variare del tipo della variabile puntata. L'algebra dei puntatori è uno strumento molto potente e permette di eseguire svariate operazioni in memoria senza esplicitamente tener contro dei dettagli dell'implementazione. Questo vuol dire che nello scrivere *pi = pi+1* il programmatore non deve curarsi se un intero occupa 32 o 16 bit; per la stessa ragione codice scritto in modo da funzionare su una macchina che implementa gli interi utilizzando 32 bit funzionerà correttamente anche se compilato su una macchina dove gli interi sono rappresentati con soli 16 bit. E' il compilatore che in entrambi i casi tiene conto delle differenze.

Non tutte le operazioni algebriche sono permesse tra puntatori. Come regola generale sono possibili solo le operazioni che hanno "senso" tra indirizzi di memoria.
In breve è lecito sommare e sottrarre valori interi ad un puntatore (operatori ++, --, +=, -=, -, +), ma non è possibile eseguire somma e prodotto tra due puntatori, poiché il risultato di tale operazione non avrebbe senso. Sottrarre due puntatori tra loro è un'operazione consentite e fornisce la distanza tra i due indirizzi di memoria.

Notare infine che l'algebra dei puntatori vale anche per puntatori a tipi definiti dall'utente (struct in C e classi in C++).

Vettori

Un vettore (*array*) è un insieme di variabili (in C++ oggetti) dello stesso tipo disposti in memoria in modo contiguo. Gli array si indicizzano a partire da 0 fino a n-1 (dove n è la lunghezza dell'array).
Es:

```
int A[10];                // definisce un array di 10 interi, senza
                          // inizializzarlo
int B[] = {1,2,3,4,5};    // definisce e inizializza un array di 5
                          // interi

// il seguente ciclo for inizializza gli elementi di A
for(int i = 0; i <10;i++)
     A[i] = 0.0;
```

La seguente figura descrive come viene memorizzato l'array A[].
Notare che non esiste nessuno tipo di controllo da parte del linguaggio sull'indice dell'array. Nessuno pertanto ci vieta di accedere ad A[10] – corrispondente all'undicesimo elemento dell'array – e di uscire dai limiti dell'array stesso. Eventuali operazioni di scrittura al di fuori di un array possono portare alle più strane conseguenze (comportamento imprevedibile o crash del programma, terminazione da parte del sistema operativo, crash del sistema...).

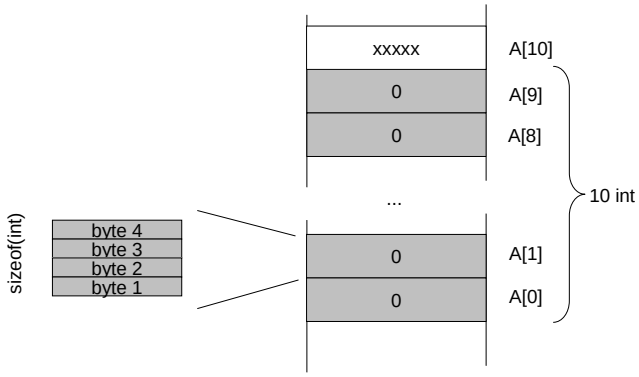


Figura 4

Stringhe C

In C le stringhe non esistono ... In realtà le stringhe non sono altro che array di *char* terminati dal carattere nullo ('\0').

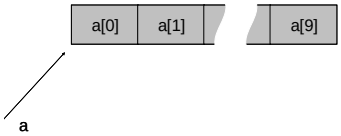
Quindi:
```
char nome[] = {'b','a','r','t','\0'};
```

ossia un array di 5 char terminato dal carattere '\0' è una stringa e coincide con:
```
char nome[] = "bart";
```
oppure
```
char nome[5] = "bart";
```

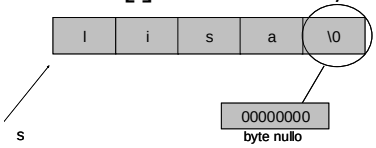
Nota: l'uso delle stringhe C appena descritte è ancora largamente diffuso ma ha delle pesanti limitazioni. In particolare non è possibile eseguire le seguenti operazioni:
```
char s1[80], s2[80], s3[80];
s1 = "homer";  //errore
s2 = "marge";  //errore
s3 = s1;       //errore
s3 = s1 + s2;       //errore
```
Tutto questo perchè in C la stringa non è un tipo tra quello predefiniti nel linguaggio. Per queste ragioni nella libreria stl del C++ è stata introdotta la classe *string*.

Vettori e puntatori.

Agli effetti pratici vettori e puntatori in C sono la stessa cosa.
Sia:
```
int a[10];          // a è un puntatore al primo int dell'array
```



analogamente:
```
char s[] = "lisa";  // s è un puntore al primo char della stringa
```



Le seguenti istruzioni sono pertanto pefettamente lecite:
```
char *p;
p = &s[0];     // copia ancora i puntatori, equivalentente
               //alla precedente
p = s;         // copia i puntatori
```

Alla luce di quanto appena detto è evidente che la notazione '*s[i]*' è perfettamente equivalente – sebbene molto più pratica – a '***(s+i)***'. E spiega perchè gli indici degli array in C partono da 0 ...

Si consideri infine il seguente esempio:
Es:
```
int array[10];
int *p;
p = array;
```

```
for (int i = 0; i<10;i++)
{
     *p = 0;
     p++;
}
```

Nota: esiste un'importante differenza tra puntatori ed array. Gli array infatti *non* sono variabili. Pensare agli array come a dei puntatori quindi non è sbagliato purché li si consideri come puntatori costanti.
In altre parole non è possibile scrivere:

```
int array[10];
for (int i = 0; i<10;i++)
{
     *array = 0;
     array++;       // errore
}
```

Passaggio di parametri nelle funzioni

In C gli argomenti vengono passati per *valore*; questo vuol dire che la funzione non ha modo diretto per alterare una variabile della funzione chiamante. Es:

```
double pow(double x);         // dichiarazione funzione
int main()
{
     double t = 10;
     double u;
     u = pow(t);    // il valore di 't', 10, viene copiato nella
                    // variabile 'x' locale alla funzione
                    // t in questo caso vale sempre 10
                    // u vale qui 100
     return 0;
}

double pow(double x)
{
     x = x*x;  // la funzione modifica  la variabile x
     return x;
}
```

All'uscita della funzione *pow()* il valore di *t* all'interno di *main* è invariato. Questo perchè la funzione crea localmente una variabile 'x' di tipo *double*. All'inizio dell'esecuzione della funzione questa variabile viene riempita con il valore del parametro passato dal main (in questo caso 10). Tuttavia questa variabile è solo una copia locale, e la sua modifica non influenza in alcun modo il valore di 't'. In questo caso il comportamento è desiderabile poiché altrimenti il valore di 't' risulterebbe inspiegabilmente modificato.

E' possibile fare in modo che una funzione possa modificare i suoi parametri. Si consideri il seguente esempio:

```
void swap(int *px, int *py)   // prototipo e implementazione
{
```

```
     int tmp;
     tmp = *px;
     *px = *py;
     *py = tmp;
}

int main()
{
     int x = 10;
     int y = 20;
     swap(&x, &y);
     // x ora vale 20, y vale 10
     return 0;
}
```

Passando i puntatori delle variabili alla funzione siamo riusciti a modificarne il valore. La situazione è la seguente:

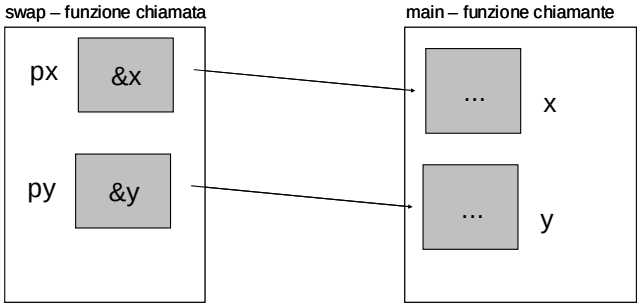


Figura 5

In questo modo la funzione chiamata riesce ad accedere, attraverso i puntatori, alle versioni originali delle variabili e a modificarle.

In C++ è possibile utilizzare una notazione diversa che permette di semplificare molto la sintassi precedentemente utilizzata (passaggio parametri *by reference* o per riferimento, si veda più avanti).

Funzioni, stringe ed array

Array e stringhe si passano sempre come puntatori. Si vedano i seguenti esempi:

Es1:

```
// moltiplica un vettore per una costante
void molt(int s, int *v, int d)
{
     int i = 0;
     while (i < s)
     {
          v[i] = v[i] * d;
          i++;
     }
}
```

```
int main()
{
     int v[] = {1,2,3};
     molt(3, v, 2);
     // v vale {2,4,6}
}
```

Es2

```
// conta le occorrenze di un carattere all'interno di una stringa
int  count(const char *str, char b)
{
     int i = 0;
     int ret = 0;
     while(str[i]!='\0')
     {
          if (str[i] == b)
               ret++;
          i++;
     }
     return ret;
}

int main()
{
     int n;
     char str[] = "lenny";
     n = count(str, 'n') ;
     // n vale 2
}
```

Funzioni "in linea" (*inline*)

L'uso delle funzioni aumenta la leggibilità, permette il riutilizzo del codice e limita gli errori. Tuttavia l'esecuzione di una funzione porta ad una certa perdita di tempo (*overhead*) necessario all'esecuzione del salto e alla creazione dello stack della funzione. Tutto questo può non essere trascurabile, soprattutto nel caso in cui la funzione è molto breve. Potrebbe accadere infatti che si stia perdendo più tempo nella chiamata della funzione che nell'esecuzione della stessa. In C++, per ovviare a tutto questo, è possibile istruire il compilatore affinché inserisca (espanda) il codice della funzione nel punto nel quale avviene la chiamata. Il prezzo da pagare per l'aumento di velocità di esecuzione è in termini di aumento delle dimensioni dell'eseguibile.
Es:

```
inline double pow(double x)
{
     return x*x;
}
```

In questo caso è buona norma mettere l'implementazione all'interno dell'header della funzione.

Occorre tenere presente che non è per niente garantito che il compilatore esegua l'inline di una funzione. Questo accade soprattutto se il corpo della funzione che si vuole rendere inline è molto complicato o se la funzione è ricorsiva. Nel caso in cui non sia possibile eseguire un "inlining" il compilatore compila il codice normalmente. Le condizioni affinché l'inlining sia possibile dipendono molto dal compilatore che si usa.

Allocazione dinamica

La memoria può essere allocata in due modi diversi.

- Allocazione *statica*. Il numero e la dimensione delle variabili viene definita durante la compilazione e non può venir modificata. Le variabili statiche si trovano nello *stack*, la loro creazione e distruzione è automatica.
- Allocazione *dinamica*. Avviene in quella parte di memoria denominata *heap*; in questo caso la dimensione della memoria può essere variata durante l'esecuzione del programma. Allocazione e deallocazione sono a carico del programma (ossia del programmatore).

L'allocazione statica è stata ampiamente sfruttata in precedenza e riguarda tutte le variabili locali e globali, i vettori, le stringhe e i parametri delle funzioni.

Le funzioni C per l'allocazione dinamica della memoria sono le seguenti:

```
void *malloc(n_byte)
```

Alloca n_byte consecutivi e restituisce un puntatore al primo di essi; il puntatore deve essere convertito al tipo desiderato per mezzo di un cast.

```
free(void *p)
```

Dealloca una zona di memoria a partire dal puntatore 'p' che ne identifica il primo elemento.

Es:

```
int  *p;
p = (int *) malloc(sizeof(int)*100);    // alloca dinamicamente 100
                                        // int
free(p);            // libera la memoria precedentemente allocata
```

Notare che la funzione malloc alloca solamente "bytes", è pertanto necessario calcolare la quantità di memoria richiesta (nell'esempio precedente *sizeof(int)*100*).

Il C++ definisce due altre funzioni per l'allocazione dinamica.

var_p new *tipo*

Alloca un oggetto di tipo 'tipo' e ne restituisce il puntatore.

delete *var_p*

Dealloca l'oggetto puntato dal puntatore 'var_p'.

Es:

```
int *p;
p = new int;   // alloca un intero
delete p;      // dealloca l'intero precedentemente allocato
```

E' possibile allocare array di oggetti in maniera dinamica, semplicemente specificandone il numero nella chiamata a *new*. In questo caso la realativa chiamata a *delete* è leggermente diversa. Si veda il seguente esempio:

```
int  *p;
p = new int [100];  // alloca dinamicamente 100 int
delete [] p;        // libera la memoria precedentemente allocata
```

Alcune note:

- In entrambi i casi l'allocazione dinamica fallisce qualora non sia disponibile abbastanza memoria per soddisfare la richiesta. In questo caso il puntatore ritornato dalle funzioni (sia *new* che *malloc*) non è valido (NULL).
- Le chiamate del C di allocazione e deallocazione sono ancora supportante dal C++ per mantenere la compatibilità con i vecchi programmi scritti in C e per la stessa ragione è opportuno conoscerne l'utilizzo. E' possibile quindi utilizzare sia *new* che *malloc* all'interno di un programma perché si utilizzi *delete* per deallocare memoria allocata con *new* e *free* per deallocare memoria allocata con *malloc*. E' comunque preferibile utilizzare *new* e *delete*.
- In C e C++ la memoria dinamica deve essere deallocata esplicitamente dal programma. Le zone di memoria allocate dinamicamente vengono assegnate dal sistema operativo al processo che ne ha fatto richiesta (con la chiamata a *new*) e risultano occupate finchè il processo stesso non le rilascia esplicitamente (con una chiamata a *delete*). E' importante ricordarsi di deallocare sempre la memoria non più necessaria. Regola generale: ad ogni chiamata a *new* deve corrispondere nel flusso del programma una relativa chiamata alla funzione *delete*.

CAST tra puntatori

Come visto in precedenza il tipo di un puntatore ne identifica l'algebra. L'operazione di *cast* applicata ad un puntatore, cambiandone il tipo, ne modifica le regole dell'algebra.
Es:

```
int a[5];
char *b,
b = (char *) a;     // cast tra puntatori
b++;           // b punta al secondo byte che costituisce a[0]
b++;           // b punta al terzo byte che costituisce a[0]
```

ecc...

Allocazione di array bidimensionali

Allocazione statica.

```
int A[10][20]; // alloca una matrice di interi 10x20
A[i][j];       // accede alla i-esima riga, j-esima colonna della
               // matrice
```

Allocazione dinamica.
Si è visto che un array viene identificato in memoria a partire dal puntatore al primo elemento. Nel caso in cui si voglia creare una matrice bidimensionale occorre quindi allocare una sequenza di vettori, ciascuno rappresentante una riga della matrice. I

puntatori alle righe sono organizzati a loro volta in un vettore (di puntatori). L'allocazione di una matrice *N righe x M colonn*e risulterà quindi nell'allocazione di *N* vettori di *M* elementi ciascuno:

Si supponga di voler allocare una matrice NxM di interi.

Es:
```
int ** B; // puntatore al vettore di puntatori alle righe

B = new (int *)[N]; // alloca il vettore di puntatori alle righe

for(int i = 0; i <N; i++)
     B[i] = new int[M];  // alloca le righe
```

La struttura di memoria allocata è rappresentata in Figura 6.
La deallocazione della matrice deve essere fatta in modo inverso:

```
for(i = 0; i < N; i++)
     delete [] B[i];// dealloca le righe

delete [] B;        // dealloca il vettore di puntatori alle righe
```

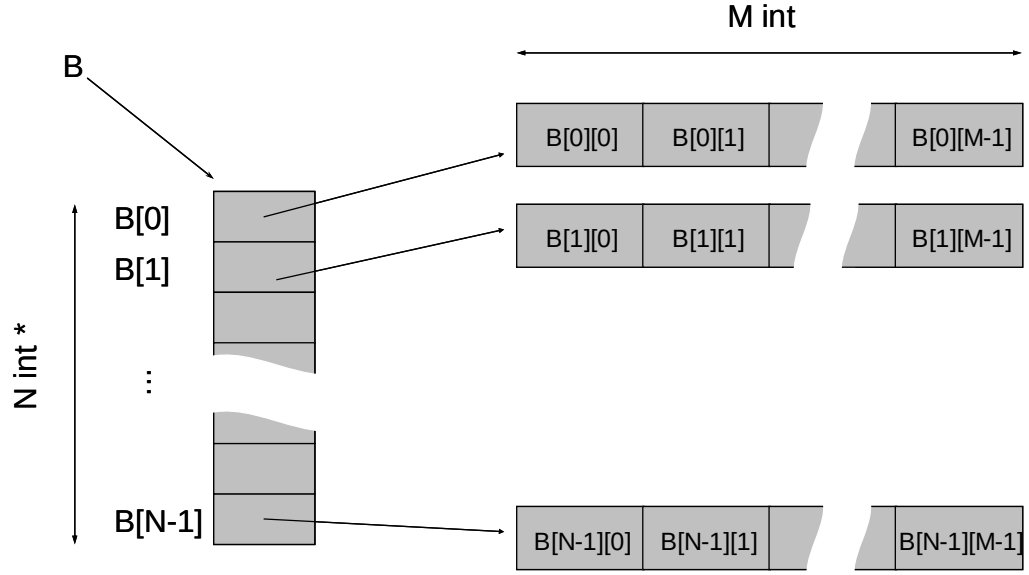


Figura 6

Strutture

Una struttura è un insieme di più variabili di uno o più tipi raggruppate da un nome comune. Le strutture sono utili per organizzare dati complessi poiché permettono di trattare più variabili come un solo oggetto. Per prima cosa occorre dichiarare la struttura, utilizzando la parola chiave *struct*. All'interno della dichiarazione si specificano le variabili che compongono la struttura, ossia i *membri* (o *campi*).

Es:
```
// definizione della struttura my_time
```

```
struct my_time {
     int hours;
     int minutes;
     int seconds;
};

struct my_time app;      // dichiara una variabile 'app' di tipo
                         // 'my_time' (sintassi C)
my_time app;        // dichiara una variabile 'app' di tipo
                    // 'my_time' (sintassi C++)
```

Il compilatore a questo punto ha allocato lo spazio necessario a contenere tutti gli elementi della struttura (in questo caso 3 interi).

Per accedere ai membri di una struttura è sufficiente utilizzare l'operatore '.':

```
app.hours = 10;
app.minutes  = 30;
app.seconds = 0;

my_time y;
y = app;  // copia le due strutture (copia bit a bit);
```

L'operatore *sizeof()* restituisce la quantità di memoria occupata dalla struttura.
Es:

```
int a = sizeof(my_time); // c vale sizeof(int)*3 (ossia 12)
```

Passaggio di strutture a funzioni

E' possibile passare strutture come parametri a funzioni; analogamente una funzione può restituire una struttura.

Es:

```
print_time(my_time t)
{
     printf("Ore:%d\n", t.hours);
     printf("Minuti:%d\n", t.minutes);
     printf("Secondi:%d\n", t.seconds);
}

main()
{
     my_time app;
     app.hours = 22;
     app.minutes = 10;
     app.seconds = 0;

     print_time(app);
}
```

Puntatori e strutture

E' possibile utilizzare puntatori a strutture. Per semplificare l'uso dei puntatori con le strutture è stato introdotto l'operatore '->'.

Es:
Si supponga di voler creare una funzione per inizializzare a zero tutti gli elementi della struttura.

```
inizializza_t (my_time *t)
{
     t->hours = 0;            // analogo a *t.hours = 0;
     t->minutes = 0;          // analogo a *t.minutes = 0;
     t->seconds = 0;          // analogo a *t.seconds = 0;
}

main()
{
     my_time t1;
     inizializza_t(&t1);
}
```

Libreria standard

Lo standard ANSI del C definisce la libreria standard dove è possibile trovare molte funzioni di uso comune.
Es:
<stdio.h> // input output
<math.h> // funzioni matematiche
<string.h> // funzioni di utilità per le stringhe
<time.h> // funzioni relative a data e ora

Per utilizzare le funzioni della libreria occorre:

1) includere il relativo .h
2) 'Linkare' la rispettiva libreria (.lib)

Alcuni esempi.
<stdio.h>
printf
fopen // si veda il paragrafo relativo all'i/o
fclose // si veda il paragrafo relativo all'i/o
fprintf
fscanf

<string.h>
strcpy(s, ct) // copia la stringa 'ct' in 's'
strcmp(cs, ct) // confronta cs e ct
strlen(s) // restituisce la lunghezza della stringa (escluso il carattere nullo)

<math.h>
sin(x)
cos(x)
sqrt(x)

Per una trattazione più approfondita si rimanda ai testi consigliati.

I/O da console in C

Le più semplici funzioni di I/O da console del linguaggio C sono *getchar()* e *putchar()* (entrambe contenute nella libreria **stdio**): la prima legge un carattere da tastiera e la seconda stampa un carattere sullo schermo. I prototipi di queste funzioni sono

```
int getchar(void);
int putchar(int c);
```

getchar() attende la pressione di un tasto e restituisce il valore corrispondente, mentre putchar() scrive un carattere sullo schermo alla posizione corrente del cursore. Come si può vedere dal prototipo il parametro ritornato da getchar() è un intero, anziché un char, ma questo può benissimo essere assegnato ad una variabile char in quanto il valore prelevato da tastiera si troverà nel byte meno significativo, lo stesso vale per il parametro passato alla funzione putchar (ovvero **c** il carattere da stampare) e per il valore che quest'ultima funzione ritorna ovvero il carattere stampato, utilizzato come controllo in quanto, in caso di errore, la funzione ritornerà EOF.

Esempio:

```
#include <stdio.h>

int main(void)
{
     char ch_in, ch_out;
     // preleviamo un carattere da tastiera
     ch_in = getchar();
     // assegnamo alla variabile ch_out il carattere a e
     // stampiamolo
     ch_out = 'a';
     putchar(ch_out);
     return 0;
}
```

L'uso di getchar() presenta un problema, poiché tale funzione solitamente viene implementata in modo da bufferizzare l'input fino alla pressione del tasto INVIO. Questo significa che la funzione non legge i caratteri premuti ed il programma non prosegue eseguendo l'istruzione successiva sino alla pressione del tasto INVIO. Inoltre, poiché getchar() *legge un solo carattere*, tutti i caratteri inseriti dopo il primo (quello che sarà restituito) sino alla pressione di INVIO, rimarranno in attesa nella coda di input, in attesa di essere prelevati da successive funzioni di input da console e questo può rappresentare un problema in ambienti interattivi.

Inoltre scrivere o leggere un carattere alla volta non è molto comodo, per questo esistono alternative a queste due funzioni, che permettono l'input e l'output formattato da console:

```
int printf(const char *control_string,..);
int scanf(const char *control_string,..);
```

Per I/O *formattato* si intende la lettura e la scrittura di diversi dati (appartenenti a diversi tipi) in vari formati.
Per quanto riguarda printf() la *control string* è formata da due tipi di oggetti: il primo è una stringa che contiene la formattazione dell'output che verrà stampato a video, mentre il secondo contiene i dati veri e propri da stampare.
Esempio:

```
char c = 'a';
int num = 9;
printf("%d è un numero, %c è un carattere e %s", num, c, "io sono
una stringa");
```

che fornisce il seguente output:

```
9 è un numero, a è un carattere e io sono una stringa
```

Come si vede dall'esempio la prima parte della control string viene stampata a video sostituendo i codici indicati con % (chiamati *specificatori di formato*) con i dati specificati, in maniera ordinata, nella seconda parte della control string, che può essere formata sia da variabili che da costanti. Così printf sostituisce al primo specificatore di formato il contenuto della varibile *num*, interpretandolo come un intero, al secondo il contenuto della variabile c, interpretandolo come un carattere ed al terzo la costante "*io sono una stringa*" interpretandola come una stringa (appunto!). L'interpretazione (e quindi il formato) che viene data ad una varibile (o a una costante) presente nella seconda parte della control string viene specificata dal corrispondente specificatore di formato, presente nella prima parte.

%c	Carattere	%s	Stringa di caratteri
%d	Intero decimale con segno	%u	Intero decimale senza segno
%i	Intero decimale con segno	%x	Esadecimale senza segno
%e	Notazione scientifica (e minuscola)	%p	Puntatore
%f	Numero decimale	%o	Ottale senza segno
/n	Nuova linea, manda a capo	%%	Stampa il carattere %

Esistono altri specificatori, per i quali si rimanda ad uno dei testi di riferimento per una descrizione esaustiva.
La funzione scanf() è la routine di input da console di utilizzo più generale: questa può leggere diversi tipi di dati e ne effettua automaticamente la conversione nel tipo richiesto.

Anche in questo caso la control string è divisa in due parti principali, la prima specifica quali sono i dati da prelevare da console, mentre la seconda specifica in quali variabili immagazzinare tali dati
Esempio:

```
int num;
scanf("%d", &num);
```

queste istruzioni prelevano un intero da console e ne assegnano il valore alla variabile num. Anche scanf() è bufferizzata, quindi la lettura dei dati da console non avviene sicché non viene premuto il tasto INVIO.
Possono essere prelevati anche diversi dati (da immagazzinare in diverse variabili) con un'unica istruzione:

```
scanf("%d %d %d %c", &day, &month, &year, %char);
```

Uno spazio vuoto nella control string fa si che scanf() salti uno o più spazi presenti nel buffer di ingresso (cioè nei dati immagini da console), lo stesso vale per i caratteri diversi dallo spazio, come ad esempio una virgola, la cui presenza nella stringa di controllo fa si che scanf() ignori e salti un virgola presente nel buffer di ingresso.

I/O su file in C

Nella libreria standard del C sono incluse alcune funzioni che permettono di scrivere/leggere su/da file. La seguente funzione può essere utilizzata per aprire un file:

```
FILE *fopen(char *, char *);
```

Al termine dell'utilizzo del file occorre chiuderlo con la seguente funzione:

```
fclose(FILE *);
```

Al momento dell'apertura del file occorre specificare se il file verrà utilizzato in *lettura*/*scrittura* o in modalità *binaria* o *testo*.

- Modalità testo

Un file aperto in modalità testo permette operazioni di scrittura e lettura in maniera equivalente a quelle su schermo.
Es:

```
#include <stdio.h>
main()
{
     FILE *fp;
     fp = fopen("output.txt", "wt");    // apre il file output.txt
                                        // in lettura e
                                        // modalità testo

     double tmp = 0.3;
     fprintf(fp, "%lf\n", tmp);         // scrive il contenuto
                                        // della variabile tmp su
                                        // file
```

```
     fclose(fp);                          // chiude il file
}
```

Per leggere da file occorre invece utilizzare la funzione *fscanf* (per esempio). La funzione *feof()* in questo caso può essere utilizzata per controllare quando si raggiunge la fine del file.

Es:

```
#include <stdio.h>
main()
{
     FILE *fp;
     fp = fopen(“input.txt”, “rt”);     // apre il file input.txt
                                        // in scrittura
                                        // e modalità testo

     double d_tmp = 0.0;
     while (!feof(fp))   // esegue il loop fino alla fine del file
     {
          fscanf(fp, "%lf\n", &d_tmp);  // legge una variabile di
                                        // tipo double da file
          printf(“Carattere corrente: %lf\n”, d_tmp);
     }

     fclose(fp);    // ricordarsi di chiudere il file
}
```

- Modalità binaria

La modalità binaria viene utilizzata quando si vogliono leggere o scrivere direttamente i byte. In questo caso si utilizzano le seguenti funzioni: *fputc, fgetc*, *fwrite*, *fread*.

Es:

```
#include <stdio.h>
main()
{
     FILE *fp;
     fp= fopen(“input.bin”, "wb"); // apre il file input.bin in
                                   // scrittura, modalità binaria

     char c_tmp = 0xFF;
     fputc(c_tmp, fp);             // scrive il valore 255 su file

     char c_vector[5] = {0,0,0,0,0};
     fwrite(c_vector, 5,  sizeof(char), fp); // scrive 5  elementi
                                             // di dimensione
                                             // sizeof(char) su
                                             // file

     fclose(fp);
}
```

I/O in C++

Una delle grosse differenze del C++ rispetto al C è la gestione dell'I/O.
Le classi cin e cout possono essere utilizzate per leggere e scrivere da/su console.
Es:

```
double c = 10.0;
cout << "c vale: " << c;
cin >> c; // legge una variabile di tipo double
```

cin e *cout* sono definite all'interno di <*iostream*>

Namespace e nuovi header C++

In C quando si usa una funzione appartenente ad una libreria occorre includere il relativo file header (.h). Questo avviene tramite l'istruzione *#include*. La stessa cosa è stata mantenuta nelle prime versioni del C++ ed è tuttora utilizzata per mantenere la compatibilità e per gestire gli header creati dal programmatore. Tuttavia lo standard C++ ha creato un nuovo tipo di header per la libreria standard. I nuovi header C++ non sono file e, come tali, non hanno estensione .h. Es:

```
#include <iostream>
#include  <fstream>
```

Visto che in nuovi header sono stati introdotti solo recentemente, esistono ancora vecchi programmi che non li utilizzano. Per mantenere la compatibilità con questi programmi il C++ supporta ancora i vecchi file header. Es:

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

In realtà esiste ancora una differenza: il contenuto dei nuovi header è inserito all'interno del *namespace std*. Senza entrare troppo nei dettagli, si sappia semplicemente che il contenuto di un namespace è interamente nascosto finché non specificato diversamente. In altre parole questo significa che il contenuto di un header (come per esempio <iostream>) rimane nascosto finché non viene incontrata l'istruzione:

```
using namespace std;
```

Dopo tale istruzione non vi è differenza tra l'uso dei vecchi e dei nuovi header.

Classi

Una classe ha una sintassi del tutto simile ad una struttura. In realtà, diversamente alle strutture C, le classi possono contenere anche funzioni. Tali funzioni – dette anche funzioni membro o primitive – servono per eseguire operazioni sui dati che compongono la classe.

```
class nome-classe {
     dati e funzioni privati
public:
     dati e funzioni pubblici
};
```

L'accesso ai membri di una classe avviene utilizzando gli operatori '.' e '->'.
Se non diversamente specificato, tutti i membri di una classe sono privati. Questo significa che sono visibili solamente dalle funzioni membro della stessa classe (*incapsulamento*). Per permettere l'accesso di un membro anche da parte di funzioni che non appartengono alla classe occorre dichiararlo esplicitamente come *public*. Analogamente è possibile utilizzare la parola chiave *private* per dichiarare, dopo la lista di parametri e funzioni pubbliche, un nuovo elenco di membri di tipo privato.

Si consideri il seguente esempio:
```
class impiegato {
     char nome[80]; // private
public:
     void set_name(char *n);       // funzioni membro pubbliche
     void get_name(char *n);
     void set_stipendio(double s);
private:
     double stipendio;   // private
}

main()
{
     impiegato imp;      // istanzia un oggetto di tipo impiegato
     imp.stipendio = 0.0;     // errore ! la variabile membro
                              // stipendio è privata

     imp.set_stipendio(0.0);  // lecito, set_stipendio è una
                              // funzione pubblica
     imp.set_name("Homer");   // lecito, set_nome è una funzione
                              // pubblica
}
```

L'implementazione delle funzioni avviene nel seguente modo:
```
void impiegato::set_name(const char *n)
{
     strcpy(nome, n);
}
void impiegato::get_name(char *n)
{
     strcpy(n, nome);
}
```
ecc…

Costruttori e distruttori

In alcuni casi possono essere necessarie alcune operazioni al momento dell'istanziazione della classe (ad esempio per inizializzare le variabili membro, oppure per allocare dinamicamente la memoria). Analogamente potrebbe esserci la necessità di eseguire alcune operazioni al termine del ciclo di vita dell'oggetto (spesso deallocazione di memoria). Esistono due particolari funzioni membro che servono a questo scopo: il *costruttore* e il *distruttore*. Il costruttore è una funzione avente lo

stesso nome della classe che viene eseguita al momento della creazione dell'oggetto. Il costruttore può ricevere in ingresso dei parametri. Il distruttore ha il nome della classe preceduto dal simbolo ~ e viene eseguito automaticamente al termine dello scope della classe (ossia al momento della sua distruzione).

Modifichiamo la classe *impiegato* in modo che contenga un array di *char* dinamico. A questo punto è sicuramente consigliabile aggiungere costruttore e distruttore in modo da essere sicuri che la memoria dinamica sia correttamente allocata e deallocata.

```
class impiegato {
     char *nome;         // private
     double stipendio;   // private
public:
     impiegato();        // costruttore
     ~impiegato();            // distruttore
     void set_name(char *n);       // funzioni membro pubbliche
     void get_name(char *n);
     void set_stipendio(double s);
}

impiegato::impiegato(){
     nome = new char [80];
}
impiegato::~impiegato() {
     delete [] nome;
}
```

Overload

In alcuni casi può essere utile poter passare dei parametri alla classe direttamente nel costruttore.
Es:

```
// costruttore con parametro
impiegato(char *n){
     nome = new char [80];
     strcpy(nome, n);
}

oppure:
impiegato(char *n, double s) {
     nome = new char [80];
     strcpy(nome,n);
     stipendio = s;
}
```

Se opportunamente inseriti nella classe *impiegato* i precedenti costruttori ci permettono di istanziare oggetti nei seguenti modi:

```
impiegato imp1;     // uso il costruttore senza parametri
impiegato imp2("Homer"); // uso il costruttore con un parametro
```

```
impiegato imp3("Marge", 10.5);    // uso il costruttore con due
parametri
```

In realtà tutti e tre i precedenti costruttori possono coesistere all'interno della classe. Più funzioni in C++ possono infatti avere lo stesso nome, purché differiscano nei parametri di ingresso. In base alla lista dei parametri il compilatore è in grado di scegliere quale funzione utilizzare. Questa operazione è detta *overload*. Ovviamente l'overload può essere applicato a qualunque funzione (e non solo ai costruttori). Si consideri il seguente esempio:

```
struct  complex {
     double Re;
     double Im;
};

// esegue il quadrato di un double
double  quadrato(double a) {
     return a*a;
}
// esegue il quadrato di complex
complex  quadrato(complex a) {
     complex tmp;
     tmp.Re = a.Re*a.Re – a.Im*a.Im;
     tmp.Im = 2*a.Re*a.Im;
     return tmp;
}
```

Notare che non è possibile eseguire l'overload sulla base del tipo di ritorno. La seguente funzione infatti andrebbe in conflitto con la precedente.

```
double quadrato(complex a) {
     double tmp = sqrt(a.Re*a.Re + a.Im*a.Im);
     return tmp;
}
```

Parametri di default

E' possibile specificare all'interno di una funzione uno o più parametri di *default*. Durante la chiamata alla funzione, il valore dei parametri eventualmente mancanti assume quello di default (se presente, altrimenti si genera un errore).

Es:

```
class cubo
{
     int a,b,c;
public:
     cubo (int i = 0, int j = 0, int k=0);   // parametri di
                                             // default per il
                                             // costruttore
     int volume() {
     return a*b*c;
     }
};
```

```
cubo::cubo(int i, int j, int k) {
     a = i;
     b = j;
     c = k;
}
```

Grazie alla presenza dei parametri di default nel costruttore è possibile inizializzare oggetti di tipo cubo nei seguenti modi:

```
cubo a;        // equivale a cubo a(0,0,0)
cubo b(2);     // equivale a cubo b(2,0,0);
cubo c(2,2,2);

int vb= b.volume();      // vb vale 0
int vc = c.volume();     // vc è pari a 8
```

Nota: limitazioni nell'uso dei parametri di default:

- i parametri di default devono essere necessariamente in fondo alla lista
- non devono esistere ambiguità con eventuali overloading della stessa funzione

Notare infine che i parametri di default devono essere inseriti nel prototipo della funzione.

L'overloading delle funzioni non è limitato ai soli costruttori (ovviamente). Si consideri il seguente esempio:

```
void stampa_errore(char *string = NULL);          // dichiarazione

void stampa_errore(char *string)             // implementazione
{
     if (string == NULL)
          printf("Errore\n");
     else
          printf("Errore: %s\n", string);
}
```

Overload di Operatori

Gli operatori in C++ sono funzioni e come tali possono essere soggetti ad overload.
Si consideri la seguente classe:

```
class complex {
public:
     double Re;
     double Im;
     complex (double r = 0.0, double i = 0.0){
     Re = r;
     Im = i;
     }
}

main()
{
     complex A;
```

```
     complex B(0,4);
     complex C(-1);
     A = B+ C;      // ??
}
```

E' possibile eseguire l'overload dell'operatore '+' in modo che possa funzionare anche con il tipo *complex*.

Nel caso particolare la sintassi per l'overloading dell'operatore + è la seguente:

```
complex operator+ (complex b);          // prototype
```

L'implementazione è la seguente:

```
complex complex::operator+(complex b)
{
     complex temp;
     temp.Re = Re + b.Re;          // somma la parte reale
     temp.Im = Im + b.Im;          // somma della parte immaginaria
     return temp;
}
```

L'esecuzione dell'espressione *A = B+C* corrisponde ad una chiamata alla funzione *(B+)* con parametro *'C'*.

Altro esempio è l'overloading dell'operatore ++.
Es:

```
complex a,b;
a = b++;
```

```
E' sufficiente definire:
complex operator++ ()
{
     Re++;
     Im++;
     return *this;       // ritorna l'oggetto stesso
}
```

In realtà il precedente corrisponde all'operatore prefisso *++b*. Nel caso postfisso la sintassi è lievemente differente:

```
complex operator++ (int tmp)
{
     complex tmp;
     tmp.Re = Re;
     tmp.Im = Im;
     Re++;
     Im++;
     return tmp;         // ritorna l'oggetto tmp
}
```

Si noti che in questo caso eseguire una copia dell'oggetto *prima* dell'incremento e restituirla alla fine della funzione. L'overload dell'operatore postfisso in questo caso ne soddisfa solamente la sintassi; è compito del programmatore garantire che ne sia rispettata la semantica.

E' possibile fare l'overload di vari operatori:
"+", "-", "++", "—", "*", "->", ">", "<" ecc…
ma non:
".", ".:", ".*", "?"

Ereditarietà

L'ereditarietà consente di definire una classe generale (chiamata classe base) le cui caratteristiche saranno "ereditate" dalle classi figlio (o *derivate*). Le classi derivate che ereditano da una, o più, classi base ne ereditano le variabili e funzioni membro, con la loro implementazione. Questo consente di stabilire una gerarchia fra le diverse classi ed al contempo evita la di riscrivere codice uguale per classi simili. Ogni classe derivata può a sua volta essere ereditata da un'altra classe derivata, ripetendo così il meccanismo di ereditarietà all'infinito.
L'ereditarietà delle classi ha la seguente forma generale:

```
class nome_classe_derivata : tipo_di_accesso nome_classe_base
{
     // corpo della classe
};
```

Lo specificatore *tipo_di_accesso* indica come la classe derivata erediterà variabili e funzioni della classe base. Esistono tre diversi specificatori di accesso:

- *public* : tutti i membri public della classe base verranno ereditati come membri public, mentre quelli protected verranno ereditati come tali.
- *private* : tutti i membri della classe base saranno ereditati come private
- *protected* : i membri public e quelli protected verranno ereditati come private.

In ogni caso un membro private non sarà accessibile alle classi derivate.
Esempio:

```
#include <iostream>

class baseC
{
     protected:
          int i;
     public:
          void set_i_Value(int val) { i = val; }
          int get_i_Value(void) { return i; }

          char c;
};

class derivedC : protected baseC
{
```

```
     private:
          int k;
     public:
          void set_k_Value(int val) { k = val; }
          int get_k_plus_i_Value(void) { return k+i; }
};

int main()
{
     derivedC obj;

     obj.set_k_Value(3);
     obj.set_i_Value(7);
     obj.c = 'a';

     cout << obj.get_i_value();
}
```

Come si può vedere la classe derivata opera con i membri della classe base come se appartenessero alla classe derivata stessa.
Da notare che, siccome la classe baseC è stata eredita come *private* il suo membro i è accessibile dalla classe derivata, ma non lo è dall'esterno, come se fosse *private* all'interno di derivedC.

È possibile ereditare da più di una classe, semplicemente separando l'elenco delle classi base con i relativi specificatori di accesso, nella dichiarazione della classe derivata.

Per quanto riguarda le funzioni costruttore e distruttore della classe base, queste vengo chiamate quando si istanza o si distrugge, rispettivamente, un oggetto della classe derivata, con un ordine preciso:
creazione

1) costruttore classe base
2) costruttore classe derivata

distruzione

1) distruttore della classe derivata
2) distruttore della classe base

Nel caso in cui il costruttore della classe base richieda dei parametri, questi devono essere specificati nella dichiarazione della classe derivata, in questo modo:

```
costruttore_classe_derivata(elenco_argomenti_derivata) :
classe_base1(elenco_argomenti_base1),
..,
clase_baseN(elenco_argomenti_baseN)
{
     // corpo del costruttore della classe derivata
}
```

Duplicazione degli oggetti

Il problema della duplicazione degli oggetti si ha nei seguenti casi:

1) inizializzazione
2) assegnamento

Inizializzazione di oggetti e il costruttore di copia

L'inizializzazione di un oggetto avviene nei seguenti casi:

i) passaggio di oggetti per valore all'interno di una funzione
ii) inizializzazione
iii) creazione di oggetti temporanei (es: in corrispondenza di *return*)

Si consideri una versione modificata della classe *cubo*.

```
class cubo
{
     int *x;
     int n_elem;
     public:
          cubo(int n = 3, int v = 0)
          {
               n_elem = n;
               x = new int [n_elem];
               for(int i = 0;  i < n_elem; i++)
                    x[i] = v;
          }
          ~cubo ()
          {
               delete [] x;
          }
          int volume ()
          {
               int tmp = 1;
               for(int i = 0;  i < n_elem; i++)
                    tmp = tmp *x[i];
               return  tmp;
          }
};
```

Supponiamo ora di voler implementare una funzione che stampa su schermo il volume del cubo:

```
void stampa_volume(cubo c)
{
     printf("Volume = %d\n", c.volume());
}

main()
{
     cubo t(3,2);
     stampa_volume(t);
```

```
}
```

Il problema si verifica quando *t* viene passata alla funzione *stampa_volume*. Si ricordi infatti che i parametri di una funzione sono variabili locali inizializzate allo stesso valore delle variabili passate alla funzione stessa. In questo caso particolare, ad esempio, al momento dell'esecuzione di *stampa_volume* viene creato un oggetto di tipo *cubo* che è inizializzato al valore di *t*. In mancanza di altra informazione l'inizializzazione avviene attraverso la copia membro a membro (bit a bit). La situazione risultante è rappresentata in Figura 7.

Il problema a questo punto è il seguente: *t* e *c* sono due classi *distinte* ma condividono la stessa area di memoria dinamica. Al termine della funzione *stampa_volume* la classe *c* cessa di esistere e viene lanciato il suo distruttore; come conseguenza la memoria dinamica interna a *c* viene distrutta (*delete [] x*). All'uscita dalla funzione, *t.x* non è più un puntatore valido: la zona di memoria alla quale punta è stata distrutta. Questo può provocare dei malfunzionamenti nel caso in cui si tenti di utilizzare la classe, e generare un errore quando, alla fine del *main*, il distruttore di *t* tenta nuovamente di deallocare la memoria puntata da *x*.

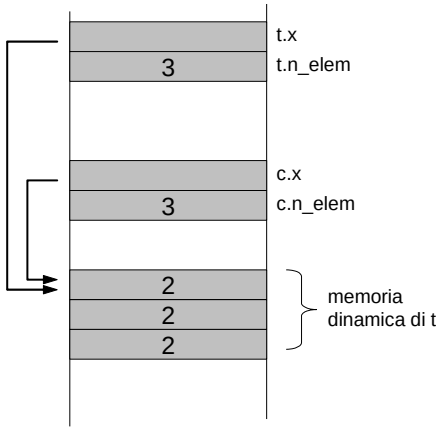


Figura 7. Situazione risultante al momento dell'esecuzione della funzione *stampa_volume*. L'oggetto *c* viene inizializzato al valore di *t* per mezzo della copia bit a bit. Il risultato è che *c.x* punta alla stessa zona di memoria di *t.x*.

E' possibile ovviare a questo inconveniente in tre modi:

1) Passare il puntatore della classe invece che la classe stessa. Ossia:

```
void stampa_volume(cubo *c)
{
     printf("Volume = %d\n", c->volume());
}

main()
{
     cubo t(3,2);
     stampa_volume(&t);
}
```

In questo modo non viene eseguita nessuna duplicazione di *t* perché la funzione riceve solamente un puntatore all'oggetto. Questa soluzione non è preferibile perché la funzione, per mezzo del puntatore, può modificare *t*; la sintassi del programma risulta appesantita dall'uso degli operatori & e ->.
2) Il C++ introduce il passaggio di parametri per riferimento *(by reference*). E' sufficiente modificare il prototipo della funzione nel seguente modo:

```
void stampa_volume(cubo &c)
{
     printf("Volume = %d\n", c.volume());
}
main()
{
     cubo t(3,2);
     stampa_volume (&t);
}
```

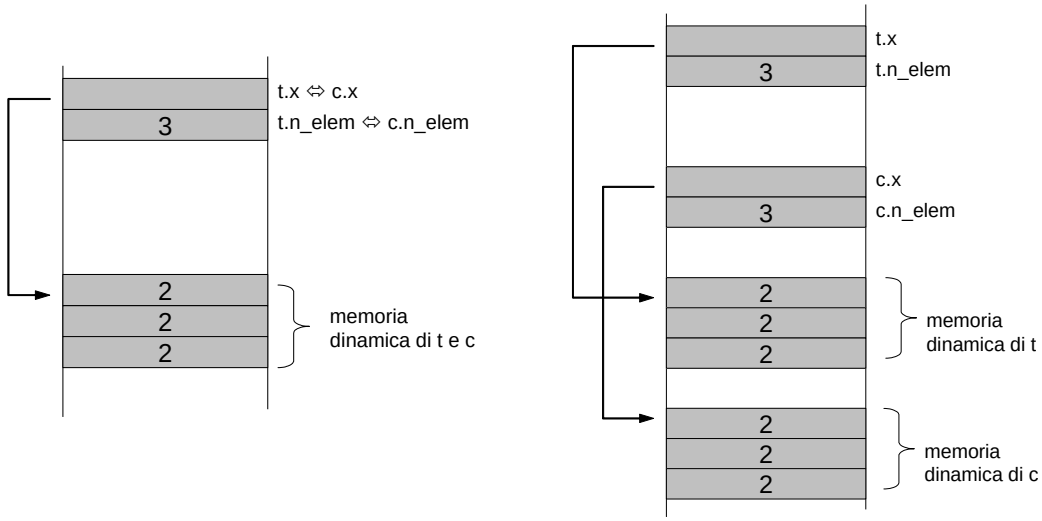
In questo modo *c* e *t* *sono lo stesso oggetto* e non viene eseguita nessuna copia (si veda Figura 8, a). Per impedire che la funzione cambi accidentalmente il contenuto di *t* (attraverso *c*) è possibile utilizzare il modificatore *const* nel prototipo:

```
void stampa_volume(const cubo &c)
{..}
```

Il compilatore garantisce che le variabili membro di *c* non siano modificate; per questo motivo sarà impossibile invocare funzioni membro che non sono dichiarate esplicitamente come *const*. Per una trattazione maggiormente approfondita dell'argomento si veda uno dei testi di riferimento.

Infine è possibile istruire il compilatore in modo che sia in grado di eseguire la copia dell'oggetto cubo.
3) Overload del costruttore di copia.
Il problema in esame nasce dal fatto che il compilatore non sa come eseguire la copia di un oggetto definito dal programmatore. In mancanza di informazione migliore esegue la copia bit a bit dei due oggetti. Scrivendo esplicitamente l'overload del costruttore di copia è possibile istruire il compilatore in modo che sia in grado di eseguire correttamente tale operazione. Il costruttore di copia è un costruttore avente come parametro un reference ad un oggetto dello stesso tipo della classe a cui appartiene.
Nel nostro esempio è quindi necessario aggiungere la seguente funzione al prototipo della classe:

```
cubo(const cubo &t)
{
     n_elem = t.n_elem;
     x = new int [n_elem];
     for (int i = 0; i < n_elem; i++)
          x[i] = t.x[i];
}
```



a) passaggio *by reference*

b) overload del costruttore di copia

Figura 8. a) Passaggio by reference: *t* e *c* coincidono (anche se *c* è dichiarato di tipo *const* e pertanto non può essere modificato). b) Overload del costruttore di copia: *c* e *t* sono due oggetti identici ma distinti .

Il costruttore di copia esegue l'inizializzazione dell'oggetto e successivamente provvede alla copia dei dati contenuti in *t*. L'aggiunta del costruttore di copia permette di passare la classe per valore ad una funzione (Figura 8, b).

Si noti che, benché molto elegante, quest'ultima soluzione costringe all'esecuzione del codice di copia ogni volta che la classe è passata come parametro. Il peso computazionale di questa operazione (*overhead*) può non essere trascurabile: il modo più efficiente (e più usato) di passare oggetti come parametro alle funzioni è quindi quello *by reference* (soluzione numero 2).

Assegnamento

Un problema del tutto analogo nasce al momento della copia di due oggetti.
Es:

```
cubo a(3,2)
cubo b;
b = a;    ??
```

Il C++ esegue automaticamente l'overload dell'operatore "=" e negli esempi appena descritti abbiamo già fatto uso di tale operatore anche in presenza di classi. La differenza in questo caso è ancora nel fatto che la classe *cubo* utilizza memoria allocata in modo dinamico. Eseguendo la semplice copia bit a bit durante l'assegnamento tra *a* e *b* la situazione risultante è del tutto analoga a quella descritta in Figura 7. Anche in questo caso i due oggetti finiscono per condividere l'area di memoria puntata da *x* con la conseguenza che:

i) eventuali modifiche su *a* influenzano anche *b*
ii) i distruttori di *a* e *b* tentano di deallocare la stessa area di memoria: uno dei due restituisce un errore

Per ovviare a questo secondo inconveniente è sufficiente eseguire l'overload esplicito dell'operatore "=":

```
void  operator = (const cubo &t)
{
     delete [] x;   // dealloca la memoria attualmente in uso dalla
                    // classe
     n_elem = t.n_elem;
     x = new int [n_elem];
     for(int i = 0; i < n_elem; i++)
          x[i] = t.x[i];
}
```

Per far sì che sia possibile concatenare tra loro successive operazioni di assegnamento (in modo cioè che sia possibile eseguire $a = b = c$) occorre modificare l'operatore '=' in modo che restituisca un oggetto di tipo *cubo*:

```
cubo operator = (const cubo &t)
{
     delete [] x;   // dealloca la memoria attualmente in uso dalla
                    // classe
     n_elem = t.n_elem;
     x = new int [n_elem];
     for(int i = 0; i < n_elem; i++)
          x[i] = t.x[i];

     return *this;
}
```

L'ultima riga fa in modo che al termine dell'assegnamento sia restituito una copia dell'oggetto stesso. Si noti che questo rientra nel primo problema (creazione di oggetti temporanei); in questo caso è quindi necessario fare anche l'overload del costruttore di copia. In alternativa è ancora possibile (preferibile) ritornare un semplice reference all'oggetto:

```
cubo &operator = (const cubo &t)
{
     [..]
     return *this;
}
```

Duplicazione di oggetti: nota finale

Il problema della duplicazione degli oggetti si presenta in due forme diverse: inizializzazione e assegnazione. Si noti che tali problemi sono simili ma del tutto differenti. Il primo si ha al momento della *creazione e inizializzazione* di un oggetto e si risolve quindi con l'overload del costruttore di copia. Il secondo si ha quando si assegna il valore di un oggetto ad un secondo oggetto *precedentemente allocato* e si risolve eseguendo l'overload dell'operatore '='. Nell'esempio riportato questo fatto è reso particolarmente evidente dalla presenza all'interno della funzione di una *delete* prima della *new*.

Testi di riferimento

"Linguaggio C", B.W. Kernighan D.M. Ritchie, Jackson.
"C++", H. Schildt , Mac Graw Hill.
"The C++ Programming language", Bjarne Stroustrup, Addison Wesley, 3d ed.
"Advanced Windows", J.Richter, Microsoft Press, 3rd ed.