Codifica in memoria

Le variabili di un linguaggio evoluto sono porzioni della memoria che contengono i dati dell'utente, scritti in una qualche rappresentazione binaria.

I formati possibili sono:

- Interi con o senza segno
- "Reali" cioè numeri rappresentati in virgola mobile
- "Stringhe" di caratteri alfanumerici
- Variabili logiche
- Tipi composti
- Rappresentazioni definite dall'utente...

Rappresentazione dei numeri negativi

Dati N segnali digitali (N bits) è possibile rappresentare numeri interi negativi utilizzando N-1 bits per il modulo e 1 bit per il segno (0 = + 1 = -). Ad esempio, su 5 bits:

+3 = 0 0011 0011 = 1•2 + 1 = 3

-5 = 1 0101 0101 = 1•2^2 + 1 = 5

Il problema di questa tecnica consiste nel fatto che le regole da usare per sommare numeri positivi sono diverse da quelle che servono per i numeri negativi. Se ad esempio proviamo a sommare le rappresentazioni di +3 e –5 con le regole canoniche otteniamo:

```
 00011  +
 10101  =
-----
 11000  = -2^3 = -8
```

mentre la rappresentazione corretta di –2 è 10010.

Rappresentazione in complemento a due

L' idea della rappresentazione in complemento a due consiste nell'utilizzare i numeri positivi

$2^N-n \qquad 1 < n < 2^{N-1}$

Per rappresentare i numeri negativi $-n$.

A fianco è riportata la tavola di corrispondenza tra numeri e rappresentazioni.

I numeri che possono essere scritti su N bits sono

$-2^{N-1} < x < 2^{N-1} - 1$

$2^{N-1}-1$	=	011…11	$\Rightarrow$ $+(2^{N-1}-1)$
$2^{N-1}-2$	=	011…10	$\Rightarrow$ $+(2^{N-1}-2)$
		•	
		•	
		•	
3	=	000…11	$\Rightarrow$ +3
2	=	000…10	$\Rightarrow$ +2
1	=	000…01	$\Rightarrow$ +1
0	=	000…00	$\Rightarrow$ 0
2^N-1	=	111…11	$\Rightarrow$ –1
2^N-2	=	111…10	$\Rightarrow$ –2
2^N-3	=	111…01	$\Rightarrow$ –3
2^N-4	=	111…00	$\Rightarrow$ –4
		•	
		•	
		•	
$2^{N-1}+1$	=	100…01	$\Rightarrow$ $-(2^{N-1}-1)$
2^{N-1}	=	100…00	$\Rightarrow$ -2^{N-1}

Operazioni in complemento a due

Proviamo ora ad applicare le regole di somma ai numeri scritti in complemento a due; consideriamo due numeri, f e g, con $f>g>0$

Valore	Rappresentazione
$f+g = p$	$f+g \Rightarrow p$
$f-g = q$	$f+2^N-g = 2^N+(f-g)$
$g-f = -q$	$g+2^N-f = 2^N-(f-g) \Rightarrow -q$
$-f-g = -p$	$2^N+2^N-(f+g)$

q

-p

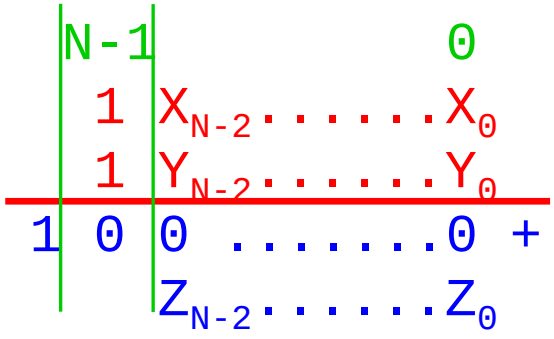
Come si vede, le rappresentazioni risultanti sono quelle dei risultati algebrici, o differiscono da queste per l'aggiunta di un termine 2^N non visibile su N bits:

$$2^N+x = \underbrace{100\ldots00}_{\text{N volte}} + x_{n-1}\ldots x_0 = 1\,\underbrace{x_{n-1}\ldots x_0}_{\text{N bits}}$$

Overflow e riporto

Quando si eseguono operazioni a numero di bits fissato, si deve fare attenzione a non eccedere i valori massimi e minimi ammessi. Se si lavora con i soli numeri positivi questo avviene quando si ha un riporto sul bit N. In complemento a due si osserva che:

- La somma di un positivo e un negativo è sempre nei limiti ammessi.
- La somma di due positivi è valida se non si ha riporto dal bit N-2 al bit N-1 (bit di segno).
- La somma di due negativi è valida se si ha riporto dal bit N-2 al bit N-1 (bit di segno).



Rappresentazioni in virgola mobile

La rappresentazione binaria (normale o in complemento a due) va bene per numeri interi o frazionari; questi ultimi vengono però dati con precisione fissa.

E' possibile rappresentare numeri "reali" (con precisione finita ma variabile) utilizzando una notazione del tipo

$$X = a\,10^{b} \quad -1 < a < 1 \quad b \text{ intero}$$

a (mantissa) e b (esponente) possono essere rappresentati in complemento a due. Su 32 bits si ottengono tipicamente sette cifre significative sulla mantissa e esponenti compresi tra –38 e +38. Su 64 bits si hanno 14 cifre di precisione ed esponenti tra -300 e 300.

È importante osservare che, a differenza del codice in complemento a due, in questo caso non tutte le possibili combinazioni di 32 bits formano un "floating" valido. Tali combinazioni illecite vengono segnalate come NaN (Not a Number). Esiste anche un codice speciale che si chiama Inf e che viene assegnato come risultato di overflow o di operazioni illecite come la divisione per zero.

Il codice ASCII

Per rappresentare il testo si usa un codice, detto codice ASCII, in cui a ogni combinazione di otto bits (1 byte) si fa corrispondere un simbolo; ad esempio:

“A” = 65	“a” = 97
“B” = 66	“b” = 98
“0” = 48	“(” = 40
“1” = 49	“+” = 43

Ci sono ovviamente 256 simboli possibili, solo in parte occupati da lettere, numeri e simboli vari.

Un insieme di caratteri ASCII si dice stringa.

Variabili Logiche

Le variabili logiche possono assumere solo i valori "vero" o "falso". In C++ si possono usare variabili intere come variabili logiche, con la convenzione che

0 = falso 1 = vero.

Le espressioni logiche sono comunque molto usate; nel comando:

```
if (a>5) {
  ...
}
```

a>5 un espressione logica che vale 0 o 1 (cioè

```
int i;
i = (a>5);
```

assegna a i il valore 0 o 1 a seconda del valore di a).

Esistono poi vere e proprie variabili logiche, di tipo bool, che possono assumere i valori true e false.

```
bool test = true;
...
if (test && a==0) {
    ...
}
```

Le variabili

Una variabile è una porzione della memoria che contiene dati (di ingresso o di uscita) codificati in un qualche modo.

Nei linguaggi evoluti, le variabili sono identificate da un nome, nei linguaggi macchina da un indirizzo. Posseggono le seguenti caratteristiche:

- Tipo: specifica il tipo di codifica usato
- Dimensione: indica la quantità di memoria occupata (dim = n_comp*dim_comp)
- Modo di allocazione: indica la strategia usata per riservare lo spazio in memoria
- Scope (visibilità): indica quali pezzi di un programma possono vedere (e usare) una variabile.
- Valore iniziale: il valore della variabile all'inizio dell'esecuzione del programma.

Il “tipo” di una variabile

La codifica utilizzata per rappresenare i dati nella memoria occupata da una variabile si dice tipo. In C sono disponibili diversi tipi predefiniti:

- int, unsigned int: intero con o senza segno, 4 bytes.
- float: floating point, 4 bytes.
- double: floating point, 8 bytes.
- char: carattere alfanumerico, 1 byte.

Il programmatore deve essere sicuro che tutte le parti di un programma che utilizzano una data variabile siano d’accordo sul suo tipo; in caso contrario i risultati saranno imprevedibili.

Dimensione di una variabile

E' possibile definire collezioni di variabili con uno o più indici, che possono essere manipolate come singoli oggetti nei programmi. In analogia con la notazione matematica, una variabile ad un indice si dice vettore, una a due o più indici matrice.

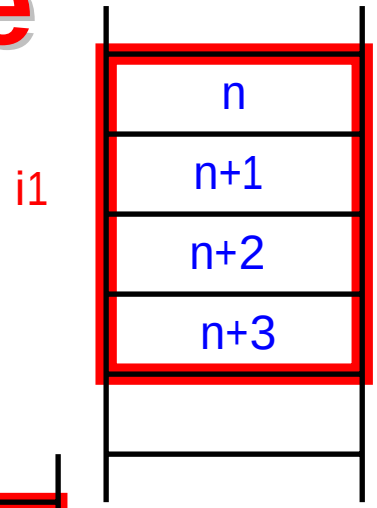
Il vantaggio dei vettori rispetto a gruppi di n variabili distinte, e che gli elementi di un vettore possono essere acceduti uno a uno per mezzo di un'altra variabile. Ad esempio:

```
int i;
float dati[10];
for (i=0; i<10; i++) {
   cout << "Il dato numero " << i << " vale " << dati[i] << endl;
}
```

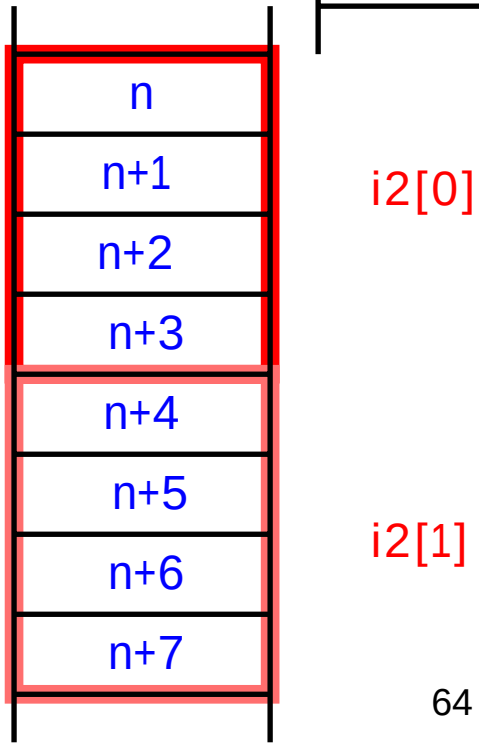
Gli indici vanno da 0 a n-1

Posizione in memoria degli elementi di un vettore

Uno scalare occupa sempre n bytes contigui in memoria (4 nel caso di un int);



Un vettore occupa un insieme di m bytes contigui, dove m = n*size, n è la lunghezza del vettore e size è la dimensione di un elemento.

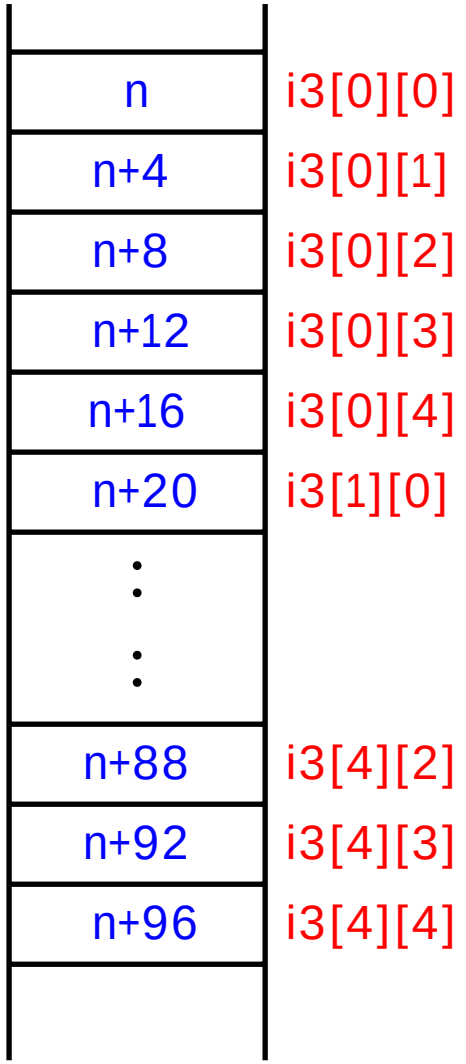


Anche in un vettore bi-dimensionale le variabili sono allocate in memoria in modo contiguo, facendo scorrere più rapidamente l'indice più a destra.

E' bene, quando si eseguono loop, rispettare l'ordine naturale delle matrici:

```
int a[1000][1000], b[1000][1000];
for (i=0; i<1000; i++) {
  for (j=0; j<1000; j++) {
    a[i][j] = a[i][j] + b[i][j]; /* corretto */
    /* a[j][i] = a[j][i] + b[j][i];  inefficiente */
  }
}
```

int i3[5][5];



Dichiarazione delle variabili in C++

In C++, ogni variabile utilizzata deve essere dichiarata esplicitamente all'inizio del blocco in cui viene utilizzata.

La linea che dichiara la variabile ne specifica, accanto al tipo, anche la dimensione ed eventualmente il valore iniziale. Ad esempio:

```
int i1, i2[10], i3[5][5];
float test = 0.24;
unsigned int counter[3] = { 1, 2, 3 };
string hello = "Ciao";
```

Costanti

In C si possono realizzare tre tipi di costanti:

- Costanti (alfa) numeriche esplicite, come nelle istruzioni

```
double C = 5./9.*(F-32);
char c = 'x';
stringa += " coda";
```

- "Variabili costanti" il cui contenuto non può essere modificato dopo l'assegnazione iniziale:

```
cons float pi = 3.14159;
```

- Simboli definiti a livello di precompilazione, che vengono sostituiti con il valore numerico al momento della compilazione:

```
#define COMPONENTS 12
double a[COMPONENTS];
```

Allocazione statica e automatica

La più semplice allocazione di memoria si dice statica: una variabile viene associata ad una locazione fissa all'inizio del programma e lì rimane per tutta l'esecuzione. Questa strategia comporta alcuni problemi:

- La memoria rimane sempre allocata anche se la variabile viene usata solo per brevi periodi.
- È necessario fissare a priori la dimensione delle variabili perché il compilatore deve sapere già al momento della compilazione quanto spazio riservare.

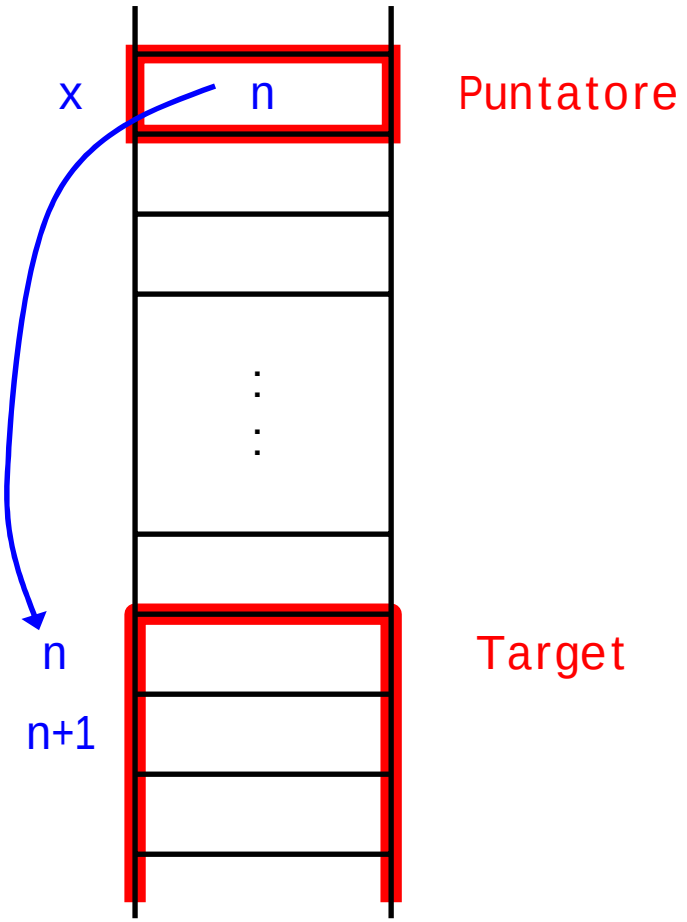
Il primo problema viene risolto dall'allocazione automatica: il compilatore fa in modo che la variabile venga allocata all'inizio del blocco che la usa e buttata via quando le istruzioni del blocco sono terminate. Le variabili all'interno di un blocco sono sempre allocate in modo automatico, a meno che non siano dichiarate con l'opzione static (es. "static float xx;").

Il concetto di puntatore

Il puntatore è lo strumento con cui si realizza l'allocazione dinamica della memoria. Si tratta di una variabile intera il cui valore rappresenta l'indirizzo di un'altra variabile (target).

Siccome il puntatore è una variabile, il suo contenuto può essere cambiato durante l'esecuzione del programma, il che equivale a modificare la posizione in memoria del target.

Inoltre un puntatore può puntare a scalari o vettori di qualunque dimensione, il che consente di modificare dinamicamente la dimensione del target.



Uso dei puntatori nei linguaggi

Per poter utilizzare la tecnica del puntatore è necessario che il linguaggio di programmazione fornisca due strumenti:

- Deve essere possibile costruire, a partire dal nome del puntatore, un riferimento simbolico al target.
- Deve essere disponibile uno strumento che consenta al programmatore di eseguire in modo esplicito l'allocazione di una porzione di memoria che costituirà il target di un puntatore.

Implementazione dei puntatori in C++

In C++ il puntatore è una variabile con un suo nome. Esiste un operatore * che, posto davanti al nome di un puntatore, simboleggia il target. Esiste anche un operatore & che data una variabile ne specifica l'indirizzo in memoria.

La sintassi per la definizione dei puntatori è del tipo:

```
int *a;    /* a è un puntatore a int */
float *b   /* b è un puntatore a float */
```

I puntatori sono specifici di ogni tipo, e possono essere usati per scalari o vettori. Ad esempio, se p1 e p2 sono puntatori

```
*p1     è il target di p1
p2[7]   è l'ottava componente del vettore puntato da p2
p1[0]   coincide con *p1
```

Se f è un vettore allocato staticamente, il nome senza parentesi è il puntatore a quel vettore.

Inizializzazione dei puntatori

Definire un puntatore **NON** significa associargli alcun target. Di conseguenza un puntatore appena creato è del tutto inutilizzabile in quanto non punta a niente di sensato. E' però buona norma assegnare ai puntatori un valore iniziale fisso (di solito 0 o il simbolo NULL) in modo da poter usare questo valore per testare se il puntatore è associato o meno a un target.

```
int *pt = NULL;

...
if (pt != NULL) {
   /* il puntatore pt ha un target definito */
}
```

Associazione a una variabile statica

Un puntatore può essere associato a una variabile statica (in principio anche a una automatica, ma la cosa è molto rischiosa perché quando la variabile viene distrutta il puntatore rimane definito in modo errato).

```
int *pt=NULL, i1, v1[10];

...

pt = &i1; /* da questo momento *pt coincide con i1 */

...

pt = v1; /* pt[i] coincide con v1[i] */

...
```

Allocazione dinamica

Con “allocazione dinamica” si intende la creazione di una variabile durante l’esecuzione. Ciò consente di creare variabili di dimensione voluta, pilotata ad esempio da un input ricevuto dal programma, o di modificare dinamicamente la dimensione di vettori o strutture.

In C++ le variabili allocate dinamicamente vengono accedute tramite puntatori, e lo spazio in memoria viene riservato con la funzione di sistema new e rilasciato quando non è più necessarie con la funzione delete.

Uso di new e delete

new è una funzione di sistema che ricerca in memoria un area di dimensioni specificate implicitamente dal tipo e dalla eventuale dimensione specificati di seguito, e ritorna l'indirizzo iniziale di tale area (o NULL se non c'è spazio in memoria).

La memoria necessaria sarà data in generale dal prodotto della dimensione della variabile da allocare per la dimensione del singolo elemento. Il calcolo è comunque eseguito in modo automatico.

La memoria allocata con new deve essere resa nuovamente disponibile al sistema quando non serve più. Per fare questo si chiama la funzione di sistema **delete addr** dando come parametro l'indirizzo di inizio dell'area. Se la memoria è stat allocata come un vettore si userà **delete[]**.

Vediamo come allocare dinamicamente un vettore di double:

```
int dim;
double *vect;
...
cout << “Dimmi la dimensione del vettore : “;
cin >> dim;
...
vect = new double[dim];
if (vect != NULL) {
    int i;
    for (i=0; i<dim; i++) {
       vect[i] = …
    }
    ...
    delete[] vect;
}
```

Controllo della dimensione

È importante ricordare che il C++ non esegue alcun controllo sulla corrispondenza tra la dimensione dei vettori (sia statici che dinamici) ed il loro effettivo utilizzo. Questo significa che nulla vieta di accedere alla posizione 100000 di un vettore definito con int v[10] (anzi, è una tecnica di hacking molto usata): il compilatore calcolerà l'indirizzo v+100000*sizeof(int) e vi farà accedere a quella locazione, qualunque cosa essa contenga. La cosa funziona perfettamente anche se v è un puntatore non inizializzato. Naturalmente i risultati sono imprevedibili, e specie in scrittura, si possono combinare notevoli disastri. Quindi, si deve fare attenzione, specie se si usa l'allocazione dinamica.

Conversioni implicite e casting

Abbiamo visto che in alcuni casi il compilatore esegue delle conversioni implicite tra tipi diversi.

Ad esempio, se a è un int e b un float, allora se si scrive b=a il compilatore convertirà implicitamente a in un float di ugual valore e lo assegnerà a b.

In alcuni casi è necessario richiedere esplicitamente una conversione, il che si ottiene specificando tra parentesi, prima dell'espressione, il tipo in cui l'espressione deve essere convertita:

```
int a; float b;
b = (float)a/5;
```

Un uso tipico del casting è quello di adattare il puntatore ritornato da malloc (che è un void*) al tipo richiesto.

Tipi composti

E' possibile definire gruppi di variabili dei tipi predefiniti in modo che, in determinati contesti, possano venire utilizzati come una variabile singola. Tali gruppi di dicono tipi composti o strutture.

```
struct studente {
    string nome;
    int matricola;
    float media;
};
```

Viene definita una struttura di nome studente con tre campi: una stringa per il nome, un intero per il numero di matricola e un float per la media.

La definizione non riserva alcuno spazio in memoria, ma serve a definire i campi che compongono la struttura. Per creare una variabile st1 di tipo "studente" ed un vettore vstud a 100 componenti useremo:

```
struct studente st1, vstud[100];
```

Si possono utilizzare vettori o matrici dentro le strutture:

```
Struct studente {
    string nome;
    int matricola;
    float media;
    float voti[50];
};
```

Come pure si possono usare altre strutture

```
struct corso {
    string nome;
    int numero esami;
    struct studente inscritti[1000];
};
```

Accesso ai campi di una struttura

Per accedere ai singoli elementi di una struttura si usa la sintassi:

st1.matricola

La matricola di st1

cl[1].iscritti[120].nome

Il nome del 119º iscritto al secondo corso

cl[6].iscritti[14].voti[2]

Il voto del terzo esame del 15º iscritto al settimo corso

Puntatori a tipi composti

Si possono creare puntatori a tipi composti:

```
struct aa {
    int i1;
    float r1;
};
struct aa *ptr;
```

Per accedere ai campi si puo usare (*ptr).r1 (l'operatore . ha priorita maggiore di *) oppure

```
ptr->r1
```

Si possono definire vettori di puntatori:

```
float *rptr[10];  /* un vettore di 10 puntatori a  float   */
*rptr[2]          /* il target della 3ª componente di rptr  */
rptr[3][4]        /* la 5ª componente del vettore puntato
                     dalla 4ª componente di rptr   */
```