

Programmazione II – Mega Riassunto

Sommario

Capitolo 1 – Operatori logici e bitwise	7
Tipo di dato	7
Differenze tra gli operatori logici e bitwise	8
Operatori bitwise di shifting	8
Applicazione degli operatori bitwise	8
Visualizzazione in output degli 8 bit di una variabile di tipo char	8
Creazione dei due gruppi	9
Conversione di ciascuno dei due gruppi	9
Uso degli operatori bitwise per effettuare moltiplicazioni e divisioni intera per 2	9
Scambio del contenuto di 2 variabili senza variabile di appoggio mediante bitwise	9
Azzeramento dei bit meno significativi di una variabile mediante operatori bitwise	10
Uso degli operatori bitwise per estrarre bit da una variabile	10
Costruzione di maschere per l'estrazione dei bit meno significativi	10
Costruzione di maschere per l'estrazione dei bit più significativi	11
Function per la stampa dei bit di un numero di qualunque tipo	11
Capitolo 2 – Tipo numerico intero	13
Cambiamenti di base	13
Da base diversa da 10 alla base 10	13
Da base 10 a una base diversa da 10 (parte intera)	13
Da base 10 a una base diversa da 10 (parte frazionaria)	13
Conclusione sulle conversioni da base 10 a base x	14
Da base 2 a una base di potenza di 2	14
Da una base potenza di 2 a base 2	14
Operazioni aritmetiche elementari tramite operatori bitwise	14
Il tipo intero nei linguaggi di programmazione	15
Sistema Aritmetico Intero	15
Aritmetica modulo m	16
Aritmetica modulo 2^N	16
Tipi di rappresentazione	17
Rappresentazione degli interi per segno e modulo	17
Rappresentazione degli interi per complemento a 2	17
Rappresentazione per eccesso B (biased) degli interi	17
Approfondimento sulle rappresentazioni per complemento a 2 e biased	18
Determinare la rappresentazione per complemento a 2 di un numero	18

Operazioni aritmetiche in complemento a 2	18
Rappresentazione per complemento alla base	18
Operazioni aritmetiche in rappresentazione B-biased	19
Range degli interi in C	19
Overflow un intero in C	19
Capitolo 3 – Tipo numerico reale	20
Normalizzazione della mantissa	20
Il sistema aritmetico reale floating-point (concetti teorici)	20
Formati dell'IEEE Standard 754	20
Campi del formato BASIC	21
Rappresentazione sull'asse reale dei numeri floating-point	21
Oggetti del Sistema Aritmetico Floating-Point	21
Sistema Aritmetico Standard IEEE 754 (concetti pratici)	22
Estrazione di alcuni bit dal contenuto di una variabile	22
Visualizzazione esadecimale di variabili reali	23
Relazione tra singola e doppia precisione. Alcune considerazioni	24
Estrazione dei campi segno, esponente e mantissa da un float	24
Schemi di Rounding	27
Errori di Roundoff	28
Misure di errore	28
Precisione decimale equivalente	29
Accuratezza statica	30
U.L.P. – Unit in the Last Place	30
Interpretazione geometrica dell'epsilon macchina	30
Risultato di massima accuratezza	30
Capitolo 4 – Stringhe	32
Differenze tra tipo carattere e stringa di caratteri in C	32
I/O di caratteri e di stringhe	32
Getchar e Putchar	32
Gestione delle stringhe mediante puntatori: allocazione dinamica	33
Function della libreria <string.h>	34
Capitolo 5 – Allocazione dinamica in C	35
Funzioni C per l'allocazione dinamica	35
Allocazione dinamica di matrici in C	35
Richiami sul tipo matrice in C	35
Allocazione in memoria di matrici dinamiche	36

Matrice memorizzata per righe	36
Matrice memorizzata per colonne	36
Esempio C: aggiunta di una riga in una matrice dinamica.....	37
Esempio C: eliminazione di una riga in una matrice dinamica	38
Le matrici come parametri dei sottoprogrammi	39
LDA (Leading Dimension Array).....	39
Capitolo 7 – Gestione dei file sequenziali in C	42
File in C	42
Apertura e chiusura di un file	42
File binari in C e funzioni di I/O	43
Funzioni di I/O per file di testo.....	43
Fine di un file di testo e binario	44
Capitolo 8 – Strutture dati dinamiche lineari	45
Richiami sulle strutture dati statiche	45
Tipo di dato strutturato	45
Tipo di dato primitivo e tipo di dato derivato	45
Richiami sul tipo array	45
Richiami sul tipo record	45
Generalità sulle strutture dati dinamiche.....	46
Classificazione delle strutture dati dinamiche.....	46
Strutture dinamiche lineari	46
Operazioni sulle strutture dinamiche lineari	46
Principali strutture dinamiche lineari: pila e coda.....	47
Struttura LIFO: la pila (stack)	47
Struttura FIFO: la coda (queue)	49
Principali strutture dinamiche lineari: lista	50
Struttura sequenziale: la lista lineare (linked list)	50
Operazioni su lista lineare: visita	50
Operazioni su lista lineare: eliminazione	51
Operazioni su lista lineare: inserimento	52
Implementazione C di una lista lineare: fondamentali	52
ADT (Abstract Data Type)	52
Struttura autoriferente statica	53
Struttura autoriferente dinamica.....	53
Particolari organizzazioni dei dati per una lista lineare	54
Nodo sentinella	54

Lista lineare generica	54
Applicazione delle liste ad altre strutture dati	56
Pila e coda mediante la struttura dati lista lineare	56
Lista circolare e lista bidirezionale	56
Liste multiple: rappresentazione di matrici sparse	57
Capitolo 9 – Strutture dati dinamiche gerarchiche.....	58
Generalità sulla struttura dati albero	58
Definizioni	58
Grado e sottoalbero di un nodo	58
Alberi binari.....	58
Algoritmi di visita di un albero binario.....	59
Parsing.....	59
Strutture dati per la rappresentazione di un albero binario.....	60
Rappresentazione di un albero binario mediante array	60
Rappresentazione di un albero binario mediante lista multipla	60
Algoritmi di visita (con stack esplicito) di un albero binario	61
Alberi tramati.....	61
Alberi binari di ricerca (BST)	61
Costruzione di un albero binario di ricerca (BST)	62
Bilanciamento	62
Struttura dati Heap	63
Memorizzazione di un Heap	63
Traduzione di un array in un heap – costruzione top-down	63
Costruzione bottom-up di un heap	64
Capitolo 10 – Strutture dati dinamiche reticolari	67
Struttura dati grafo	67
Rappresentazione in memoria di un grafo	67
Matrice di adiacenza.....	67
Lista di adiacenze	69
Algoritmi di visita di una struttura reticolare	69
Algoritmo DFS (Depth-First-Search) per la visita di un grafo	69
Versione iterativa dell’algoritmo DFS	69
Miglioramenti dell’algoritmo DFS	70
Algoritmo BFS (Breadth-First-Search, visita in ampiezza)	71
Confronto algoritmi BFS-DFS	72
Capitolo 11 – Tecniche di programmazione ricorsiva.....	73

Ricorsione	73
Struttura generale di una procedura ricorsiva	73
Tipi di ricorsione: classificazione	73
Tipi di ricorsione: ricorsione lineare	73
Tipi di ricorsione: ricorsione binaria	74
Tipi di ricorsione: ricorsione non lineare	74
Tipi di ricorsione: ricorsione indiretta	75
Analisi di funzioni ricorsive	75
Gestione della ricorsione da parte del compilatore	75
Profondità di ricorsione	75
Analisi della profondità di ricorsione	76
Altri esempi di algoritmi ricorsivi	76
Algoritmo di Horner	76
Costruzione ricorsiva di una lista lineare	77
Visita ricorsiva di un albero binario	78
Ricerca ricorsiva su un albero binario di ricerca	79
Capitolo 12 – Algoritmi di ordinamento a complessità quadratica	80
Classificazione degli algoritmi di ordinamento	80
Selection Sort	80
Analisi della complessità	81
Bubble Sort	82
Considerazioni sul bubble sort	83
Analisi della complessità	83
Insertion Sort	84
Analisi della complessità	84
Miglioria dell’insertion sort	85
Capitolo 13 – Algoritmi di ordinamento della classe “Divide et Impera”	86
Merge Sort	86
Richiamo: merge di array ordinati	86
Idea dell’algoritmo	86
Analisi della complessità	86
Quick Sort	88
Analisi della complessità	88
Scelta del partizionatore (pivot)	88
Heap Sort	89
Richiami sulla struttura dati Heap	89

Idea sull'algoritmo	90
C++	93
Operatori logici e bitwise	93
Variabili logiche in C++	93
Stringhe	94
Allocazione dinamica della memoria	95
Variabili puntatore	95
Variabili reference	95
Allocazione dinamica in C++	95
La classe vector	96
Classi e oggetti	97
Costruttori e distruttori	99
Function overloading	100
Ereditarietà	100
Upcasting	101
Virtual functions e polymorphism	102
Overload vs override	103
Pure virtual function e abstract class	104
File e Stream	105
I/O formattato	106
File in C++	107
Template	109
Sintassi e uso dei Template	110
Namespace e Using	111
Overload di operatori	111
Member Initializer List	112

Capitolo 1 – Operatori logici e bitwise

Tipo di dato

Per tipo di dato si intende il criterio di rappresentazione in memoria dei dati e la definizione delle operazioni consentite sul dato stesso. I tipi di dati scalari predefiniti sono:

- Logico e booleano
- Tipo carattere o stringa
- Numerico (intero o reale)

È noto che le informazioni si trovano nella memoria del calcolatore codificate in forma binaria, cioè sottoforma di sequenze di 0 e di 1 (**rappresentazione binaria**).

L'argomento della funzione **sizeof** è un unico tipo di dato di cui viene restituito il numero di byte necessari per la sua rappresentazione binaria in memoria. Il valore restituito dalla funzione **sizeof** per vari valori del suo argomento:

- Per un dato di tipo char occorre 1 byte
- Per un dato di tipo short int occorrono 2 byte
- Per un dato di tipo long int occorrono 4 byte
- Per un puntatore occorrono 4 byte

Tuttavia, il valore di una qualsiasi variabile viene interpretato come un valore di verità con questa convenzione:

- Il valore 0 corrisponde al valore falso
- Un valore diverso da 0 corrisponde al valore vero.

Anche il C fornisce la possibilità di usare degli operatori logici, che sono quelli tipici dell'algebra di Boole:

op. in C	OPERATORE LOGICO	Tavole di definizione					
!A	not (\neg = negazione)	!		&&		 	
A&&B	and (\wedge = congiunzione)	F	V	F	F	F	V
A B	or (\vee = disgiunzione)	V	F	V	F	V	V

La congiunzione è vera solo se i due operandi sono contemporaneamente veri.

La disgiunzione risulta falsa solo se i due operandi sono falsi.

La negazione è un operatore unario, ossia agisce su un solo operando.

Gerarchia:

- Negazione !
- Congiunzione &&
- Disgiunzione ||

Il C permette di utilizzare il costruttore di tipo per creare nuovi tipi. Mediante il TYPEDEF è quindi possibile creare il tipo logical avente due valori: True e False.

Il linguaggio C consente di intervenire direttamente sui singoli bit di una variabile ricorrendo agli operatori bitwise (bit a bit). [\sim si pronuncia tilde, & si pronuncia umperstand]

in C	OPERATORE	in C	OPERATORE
$\sim A$	NOT (complemento)	$A \ll n$	shift a sinistra di n bit
$A \& B$	AND	$A \gg n$	shift a destra di n bit
$A B$	OR (OR inclusivo)		
$A \wedge B$	XOR (OR esclusivo)		

Differenze tra gli operatori logici e bitwise

- Rappresentazione simbolica
- Gli operandi degli operatori logici sono le voci intere, valore associato alla variabile.
- Gli operandi dagli operatori bitwise sono i singoli bit della voce associata ad una variabile.

Operatore booleano: dobbiamo calcolare il valore di verità delle due variabili; l'operazione sarà vero && vero.

Operatori bitwise di shifting

L'operatore di shift non fa altro che prendere i bit di una variabile e shiftarli verso destra o verso sinistra di una quantità di bit, andando a riempire gli spazi vuoti creatosi con dei bit 0.

L'istruzione $A \ll 4$ produce lo spostamento di tutti i bit della variabile intera A di 4 posizioni verso sinistra. Vengono introdotti quattro bit 0 per riempire i "vuoti".

L'istruzione $B \gg 3$ produce lo spostamento di tutti i bit della variabile intera B di 3 posizioni verso destra. Vengono introdotti tre bit 0 per riempire i "vuoti".

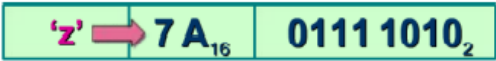
Applicazione degli operatori bitwise

Visualizzazione in output degli 8 bit di una variabile di tipo char

Sappiamo che una variabile di tipo char è rappresentata in C con 8 bit, cioè mediante 1 byte. Utilizzando il codice di formato:

- %c si visualizza il contenuto della variabile come carattere.
- %d si visualizza il contenuto della variabile come valore intero decimale.
- %x si visualizza il contenuto della variabile come valore esadecimale.

Quest'ultimo caso dimostra che è possibile rappresentare un carattere in formato esadecimale mediante due cifre, ciascuna delle quali corrisponde al gruppo di 4 bit più o meno significativo che

compongono la rappresentazione binaria in memoria del suddetto carattere, che ricordiamo essere di 8 bit. $z='z';$ 

Ad esempio, il carattere 'z' è rappresentato in binario mediante i bit '0111 1010' e in esadecimale mediante i due valori '7A' dove il 7 corrisponde al primo gruppo di 4 bit ($7 = 0111$) e la A corrisponde al secondo gruppo di 4 bit ($A = 1010$). I passi necessari per visualizzare la rappresentazione binaria di una variabile di tipo char:

- 1) Si separano gli 8 bit in due gruppi da 4 bit
- 2) Si converte ciascuno dei due gruppi nella stringa di bit equivalente
- 3) Si usa il valore ottenuto come indice nell'array bit di stringhe

Creazione dei due gruppi

Abbiamo un carattere costituito da 8 bit, di questi vogliamo prevalere i 4 bit più significativi e i 4 bit meno significativi. Questa operazione la si realizza mediante gli operatori bitwise di shifting:

- Per estrarre i 4 bit più significativi è sufficiente applicare alla variabile uno shift verso destra di 4 bit; in questo modo si eliminano i 4 bit meno significativi spostando quello più significativo sulla destra.
- Per estrarre i 4 bit meno significativi prima si shifta di 4 bit a sinistra (cancellando i 4 bit più significativi) e poi shifta di 4 bit a destra (per riportare i 4 bit sulla destra)

Conversione di ciascuno dei due gruppi

Una volta estratti i due gruppi di 4 bit, si salvano i valori numerici interi corrispondenti in due variabili, ad esempio A e B. Questi valori possono servire da indici nell'area bit. Questo è un array di stringhe composto da 6 componenti che rappresentano tutte le possibili configurazioni di 4 bit (da 0000 a 1111).

Uso degli operatori bitwise per effettuare moltiplicazioni e divisioni intera per 2

Nel sistema decimale moltiplicare un numero intero per la base 10 equivale ad aggiungere uno zero al numero originale. Nel sistema binario moltiplicare per 2 corrisponde ad ottenere una configurazione binaria identica all'originale ma con un nuovo zero nella posizione meno significativa. Questa operazione consiste in uno shift verso sinistra di una posizione.

Stessa cosa si può dire per la divisione per 2: consiste nell'aggiunta di uno zero nella posizione più significativa. Questa operazione consiste in uno shift verso destra di una posizione.

Scambio del contenuto di 2 variabili senza variabile di appoggio mediante bitwise

In programmazione è noto che effettuare lo scambio del contenuto di due variabili senza usarne una terza di appoggio provoca la perdita del valore memorizzato in una delle due. Per ovviare a questo inconveniente solitamente si usa una variabile di "appoggio" in cui si salva il contenuto che

altrimenti andrebbe inevitabilmente perso. Usando gli operatori bitwise, però, è possibile evitare l'uso di una terza variabile.

$X \wedge 1 = \sim X$ $X \wedge 0 = X$ In breve, si sfrutta la proprietà dell'operatore XOR di commutare un bit della variabile X se il secondo operando è 1 e di lasciarlo invariata se invece è 0.

Azzeramento dei bit meno significativi di una variabile mediante operatori bitwise

Per azzerare i bit meno significativi di una variabile è necessario predisporre una costante che abbia una configurazione di bit appropriata affinché l'applicazione dell'operatore bitwise & (AND) azzeri effettivamente i bit voluti.

Se si vogliono azzerare 6 bit, allora la costante assumerà valore ottale pari a 077 (costante ottale).

Se si vogliono azzerare 8 bit, allora la costante assumerà valore esadecimale pari a 0xff (costante esadecimale).

Lo 0 davanti ad un valore identifica, in C, una costante ottale. Lo 0x identifica una costante esadecimale.

Uso degli operatori bitwise per estrarre bit da una variabile

Gli operatori bitwise sono frequentemente utilizzati nell'estrazione di particolari bit della configurazione binaria di una variabile. Ciò si ottiene solitamente mediante bitwise (e alla loro proprietà), da "selettori di bit". Le proprietà degli operatori bitwise sono:

$$\begin{array}{ll} X \& 1 = X & X \& 0 = 0 \\ X | 1 = 1 & X | 0 = X \\ X \wedge 1 = \sim X & X \wedge 0 = X \end{array}$$

Supponiamo che la variabile X sia di tipo short e supponiamo di voler estrarre i bit compresi tra il quinto e l'ottavo. Per fare ciò si costruisce una variabile **mask** che contiene tutti i bit 0 tranne quei 4 bit che cogliamo estrarre. Dopodiché si fa un AND bitwise tra la variabile X e il valore calcolato dalla maschera.

Costruzione di maschere per l'estrazione dei bit meno significativi

Le tecniche di costruzione di maschere sono svariate. Di seguito elenchiamo le più comuni, considerando l'estrazione dei 5 bit meno significativi di una variabile. Nel **primo metodo** per costruire una maschera applico la formula $a^n - 1$. Nel **secondo metodo** si usano costanti che possono essere decimali, ottali ed esadecimali. Le ottali e le esadecimali si ottengono facilmente una volta scritta la sequenza di bit della maschera che si vuole costruire, mentre quella decimale la si ottiene calcolando il valore della formula precedente. Nel **terzo metodo** si usano gli operatori bitwise, si inizializza la maschera a 10, poi si itera per un numero di volte pari al numero di bit che si vogliono estrarre. In ogni iterazione viene introdotto un bit 1 nella posizione del bit meno significativo ed inoltre si effettua uno shift verso sinistra di 1 posizione in modo tale che gli 1 inseriti avanzino di posto.

Costruzione di maschere per l'estrazione dei bit più significativi

Banalmente si tratta di costruire dapprima la maschera secondo la modalità che abbiamo descritto in precedenza e successivamente operare un opportuno shift verso sinistra affinché gli 1 che la compongono diventino i bit più significativi. Il problema principale consiste nel determinare il numero di shift da effettuare, ossia di quante posizioni occorre spostare gli 1 perché diventino i bit più significativi. Ciò si può realizzare mediante la funzione SIZEOF. Occorre ricordare che questa funzione restituisce il numero di byte e non di bit necessari a rappresentare un certo tipo, quindi è necessario moltiplicare questo numero per 8 per ottenere il numero di bit necessari per la rappresentazione della variabile: $mask \ll (sizeof(type) * 8 - n)$;

Function per la stampa dei bit di un numero di qualunque tipo

Per realizzare una function del genere occorre usare il costrutto UNION. La dichiarazione del costrutto è molto simile a quella delle struct in C. La differenza sostanziale che intercorre tra i due costrutti riguarda come sono allocati i byte della memoria per ciascun tipo di dato, in particolare:

- Il size di una struct è dato risultante dalla somma dei size dei singoli campi.
- Il costrutto UNION alloca la stessa area di memoria per le variabili comprese tra parentesi. Per quanto riguarda lo spazio allocato in memoria, la UNION alloca 32 bit.

Codice

```
//estrae_bit.c – operatori bit a bit
#include <stdio.h>
#include <stdlib.h>
#include <math.h> //per usare pow(x, y)
#define MAX_LEN 32 //numero bit di intero long
void bit_show (short, char[], short[]);
void main() {
    short menu, bit[MAX_LEN];
    union word32bit {
        float F;
        long L;
        short S[2];
        char C[4];
    } word;
    do {
        puts("seleziona"); puts("[0] uscita programma");
        puts("[1] rappresentazione binaria di intero char");
        puts("[2] rappresentazione binaria di intero short");
        puts("[3] rappresentazione binaria di intero long");
        fflush(stdin); scanf("%hd", &menu);
        switch(menu) {
            case 0 : exit(0);
            case 1 :
```

```

len = sizeof(char);
printf("immettere intero char C");
fflush(stdin); scanf("%d", &(word.C[0]));
puts("char in decimale, esadecimale e binario");
printf("C=%+10hd, hex=%02x", word.C[0], word.C[0]);
bit_show(sizeof(char), word.C, bit); break;
case 2 :
len = sizeof(short);
printf("immettere intero short S"); //o S=pow(2,5);
fflush(stdin); scanf("%hd", &(word.S[0]));
puts("short in decimale, esadecimale e binario");
printf("S=%+10hd, hex=%04hx", word.S[0], word.S[0]);
bit_show(sizeof(short), word.C, bit); break;
case 3 :
len = sizeof(long);
printf("immettere intero long L"); //o L=pow(2,5);
fflush(stdin); scanf("%ld", &(word.L));
puts("long in decimale, esadecimale e binario");
printf("L=%+10d, hex=%081x", word.L, word.L);
bit_show(sizeof(long), word.C, bit); break;
default : exit(1);
}
//visualizza i bit
for (k=8*len-1; k>0; k--)
    (k%4 == 0) ? printf("%1d", bit[k]) : printf("%1d", bit[k]);
} while (menu != 0);
}

void bit_show(short len, char ch[], short bit[MAX_LEN]) {
    short j, jc; char c;
    for (j=0; j < MAX_LEN; j++)
        bit[j] = 0;
    //estrae i bit
    for (jc=0; jc<len; jc++) {
        c = ch[jc];
        for (j=0; j<8; j++) {
            bit[j+8*jc]=c&1;
            c=c>>1;
        }
    }
}

```

Capitolo 2 – Tipo numerico intero

Il sistema di numerazione decimale al quale siamo abituati utilizza la **rappresentazione posizionale**. In questa rappresentazione le cifre hanno un peso diverso in funzione della posizione che occupano. Si può dire che un qualsiasi numero, rappresentato usando un sistema di numerazione posizionale, appare scindibile come la somma dei prodotti tra coefficienti e potenze della base del sistema di numerazione stesso, ossia è a tutti gli effetti concepibile come una **combinazione lineare**, cioè un'espressione del tipo:

$$a_1b_1 + a_2b_2 + \dots + a_nb_n$$

Questa espressione specifica una combinazione lineare di elementi $\{b_1, b_2, \dots, b_n\}$ con coefficienti $\{a_1, a_2, \dots, a_n\}$. Si può notare che la somma di ogni coppia di cifre sia uguale a "base - 1".

Cambiamenti di base

Da base diversa da 10 alla base 10

Per scrivere un numero in base diversa da 10 in un numero in base 10 si deve in primo luogo scriverlo sottoforma polinomiale. Riportiamo un esempio che risulta di più facile comprensione rispetto a una spiegazione adatta. Per trasformare il numero $(1010)_2$ dalla base 2 alla base 10:

- Scriviamo il numero sottoforma polinomiale: $1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$.
- Eseguiamo le operazioni indicate: $8 + 0 + 2 + 0$
- Otteniamo 10 in base 10. Quindi $(1010)_2 = (10)_{10}$

Da base 10 a una base diversa da 10 (parte intera)

Per trasformare un numero in base 10 nella corrispondente scrittura in una base diversa da 10, operiamo con divisioni successive e in particolare bisogna tenere in considerazione i quozienti (Q, come dato in input) e il resto (R, come dato output) di ogni divisione. Esempio:

Si vuole trasformare il numero 428, appartenente al sistema decimale, nel numero corrispondente in base 6

- Dividiamo 428 per 6: otteniamo $q = 71, r = 2$
- Dividiamo il quoziente 71 per 6: otteniamo $q_1 = 11, r_1 = 5$
- Dividiamo il quoziente 11 per 6: otteniamo $q_2 = 1, r_2 = 5$
- Dividiamo il quoziente 1 per 6: otteniamo $q_1 = 0, r_1 = 1$

Scrivendo i resti ottenuti a partire dall'ultimo fino al primo otteniamo il numero $(1552)_6$ che è il numero cercato.

Da base 10 a una base diversa da 10 (parte frazionaria)

Per trasformare la parte frazionaria di un numero decimale in quella corrispondente ma in una scrittura con base diversa occorre operare in modo speculare a quanto visto per le divisioni successive, ossia procedendo per "moltiplicazioni successive". I passi fondamentali per effettuare la conversione:

- Si moltiplica la parte frazionaria del numero decimale per la base del sistema di numerazione di arrivo.
- Del nuovo numero la parte intera rappresenta una singola cifra del numero finale convertito mentre quella frazionaria viene adoperata in una nuova moltiplicazione ritornando al punto 1.
- A seconda del grado di precisione voluto si fissa un numero massimo di moltiplicazioni successive raggiunto il quale, il processo ha termine.

Conclusione sulle conversioni da base 10 a base x

Usando l'algoritmo delle divisioni successive e l'algoritmo delle moltiplicazioni successive si può generare la rappresentazione di un qualsiasi numero razionale rispetto ad una qualunque base di numerazione.

Da base 2 a una base di potenza di 2

Per esprimere numeri binari in sistemi di numerazione che hanno per base una potenza di 2, occorre per prima cosa suddividere il numero in gruppi di n bit dove n è pari a \log_2 della base del nuovo sistema di numerazione. Nel caso in cui, in un gruppo, siano presenti meno bit di quanti ne occorrono ai fini della conversione, si aggiungeranno zeri non significativi fino a raggiungere il numero richiesto. Suddivisi i bit in gruppi si procede semplicemente col sostituire ad ogni gruppo la corrispondente cifra del sistema di numerazione di arrivo.

Da una base potenza di 2 a base 2

In questo caso la conversione si effettua procedendo a ritroso nell'algoritmo visto per la conversione da base 2 ad una base potenza di 2. Cioè, si sostituisce ad ogni cifra del numero di partenza la stringa di n bit corrispondenti, dove n è determinato come $\log[a]$ della base del sistema di numerazione di partenza. Infine, si eliminano tutti gli zero non significativi.

Operazioni aritmetiche elementari tramite operatori bitwise

+	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	0 ^(*)
2	2	3	4	5	6	7	8	9	0 ^(*)	1 ^(*)
3	3	4	5	6	7	8	9	0 ^(*)	1 ^(*)	2 ^(*)
4	4	5	6	7	8	9	0 ^(*)	1 ^(*)	2 ^(*)	3 ^(*)
5	5	6	7	8	9	0 ^(*)	1 ^(*)	2 ^(*)	3 ^(*)	4 ^(*)
6	6	7	8	9	0 ^(*)	1 ^(*)	2 ^(*)	3 ^(*)	4 ^(*)	5 ^(*)
7	7	8	9	0 ^(*)	1 ^(*)	2 ^(*)	3 ^(*)	4 ^(*)	5 ^(*)	6 ^(*)
8	8	9	0 ^(*)	1 ^(*)	2 ^(*)	3 ^(*)	4 ^(*)	5 ^(*)	6 ^(*)	7 ^(*)
9	9	0 ^(*)	1 ^(*)	2 ^(*)	3 ^(*)	4 ^(*)	5 ^(*)	6 ^(*)	7 ^(*)	8 ^(*)

Come si può intuire, il risultato della somma tra due cifre del sistema decimale, viene riportato nella cella corrispondente all'intersezione di riga e colonna corrispondenti alle cifre coinvolte. Nelle celle

azzurre sono riportate le somme che generano un riporto. Consultando la tabella, è possibile effettuare operazioni di somma senza conoscere l'aritmetica decimale, facendo attenzione ad individuare eventuali generazioni di riporti. Il meccanismo visto può applicarsi naturalmente a qualunque sistema di numerazione, ossia quello binario. Nel caso del sistema binario, si preferisce scindere la tabella della somma da quella dei riporti. Queste considerazioni, apparentemente banali, permettono di **implementare un algoritmo di addizione binaria utilizzando operatori bitwise**. Si noti infatti come quella somma tra cifre binarie e quella dei riporti binari, corrispondono proprio alle tabelle di verità degli operatori XOR ed AND "bitwise". Questo nuovo algoritmo prevede l'applicazione "parallela" degli operatori bitwise alle voci che compongono gli operandi. **L'idea è quella di calcolare dapprima i riporti adoperando l'operatore bitwise "AND" e di shiftarli opportunamente per "incolonnarli" con la somma dei due operandi (che si realizza mediante l'operatore "XOR" tra i due operandi).**

La somma mediante XOR definisce il primo operando dell'operazione successiva. Il calcolo dei riporti mediante AND definisce il secondo operando dell'operazione successiva. **Il processo ha termine quando il vettore dei riporti è costituito da soli bit 0. Il risultato è dato dall'ultima applicazione dell'operatore XOR di bit dei due operandi.**

Anche per la sottrazione torna utile la tabella di verità dall'operatore bitwise XOR, mentre per il prestito non si riconosce immediatamente una descrizione equivalente. Si può giungere alla soluzione attraverso la combinazione di più operatori bitwise in diversi modi. Due possibili metodi sono rappresentati nelle figure che seguono:

$$\text{NOT}(\text{OR}(\text{op1}, \text{NOT}(\text{op2}))) \\ \text{AND}(\text{NOT}(\text{op1}), \text{op2})$$

Il tipo intero nei linguaggi di programmazione

In informatica, col termine "sistema aritmetico" si intende un criterio di rappresentazione in memoria di dati di tipo numerico e delle operazioni aritmetiche definite su di essi. Il Sistema Aritmetico di un calcolatore è costituito da due sottosistemi: il S.A. intero e S.A. reale. La finitezza della memoria di un calcolatore impone che gli insiemi numerici rappresentabili siano finiti e discreti.

Sistema Aritmetico Intero

Il seguente programma chiede di inserire da tastiera un intero, la variabile S di tipo short, e poi il valore di tale variabile viene copiato nella variabile U di tipo unsigned short. Infine, vengono stampati a video i valori di entrambe le variabili.

```
#include <stdio.h> #include <stdlib.h>
void main() {
    short s; unsigned short u;
    printf("immetti intero...");
    scanf("%hd", &s); u=s;
    printf("signed = %d\nunsigned = %d\n", s, u); }
```

Nella prima immissione, viene inserito il valore 13, che viene stampato correttamente in entrambi i casi. Nella seconda immissione, viene inserito il valore -13, che viene stampato correttamente solo quando si stampa la variabile S (signed), mentre quando si stampa la variabile U (unsigned) viene stampato un altro valore: 65523. È possibile spiegare la relazione che lega quest'ultimo valore con -13 conoscendo la **rappresentazione interna dei dati interi**.

Essa si basa sulla rappresentazione binaria posizionale. Inoltre, va detto che avendo a disposizione N bit, possono essere rappresentati tutti i numeri compresi nell'intervallo $[0; 2^N - 1]$. Quindi esiste un minimo ed un massimo tra gli interi rappresentabili oltre i quali, per definizione, si va in **overflow**. Ciò è dovuto alla limitatezza della memoria. Questa fa sì che solo un determinato intervallo di valori possa essere rappresentato.

Aritmetica modulo m

Nell'aritmetica modulo m l'insieme dei numeri naturali viene fatto corrispondere con un insieme a cardinalità finita, cioè costituito da un numero finito di numeri, precisamente da 0 a m-1. Questa corrispondenza, che si indica con ϕ_m , associa ad ogni elemento k appartenente all'insieme dei numeri naturali N un'immagine $\phi_m(k)$, che indicheremo con $k \bmod m$, che appartiene all'insieme $[0; M - 1]$. La scrittura $k \bmod m$ indica il resto della divisione intera tra k ed m. Ecco il motivo per cui il codominio è formato dagli elementi che vanno da 0 a m-1, in quanto solo questi possono essere i resti della divisione intera tra k ed m.

Gli m numeri (da 0 a m-1) sono i rappresentanti di **classi di equivalenza** dove ogni classe contiene tutti i numeri naturali che divisi per m danno luogo allo stesso resto. Due numeri naturali che appartengono alla stessa classe di equivalenza si dicono **congrui**. Cioè, se si usa fuori dal range $[0; 255]$ il risultato sarà proprio il rappresentante della corrispondente classe di equivalenza. Le variabili utilizzate sono 4:

- K di tipo unsigned char
- H, m e mod di tipo short.

I valori di queste variabili si ottengono nei seguenti modi:

- Il valore di **h** viene fissato mediante una classica assegnazione
- Il valore di **k** viene ricavato attraverso il cast del valore della variabile
- Il valore di **mod** viene ricavato attraverso la divisione intera (%) tra le variabili **h ed m**
- Il valore di **m** viene fissato mediante una classica assegnazione ed è uguale a $a^8 = 256$

Aritmetica modulo 2^N

Questo tipo di aritmetica si presta a facili implementazioni, soprattutto quando l'hardware a disposizione può rappresentare un numero abbastanza limitato di cifre.

Supponiamo che $n = 4$, quindi si hanno a disposizione soli 4 bit. Supponiamo di voler realizzare la somma tra 6 ed 11. Il risultato è ovviamente 17, ma in binario viene prodotto un bit "di troppo" rispetto ai 4 che si hanno a disposizione nell'aritmetica modulo 2^4 . Il bit in questione rappresenta un contenuto informatico che viene inevitabilmente perso e il risultato dell'operazione è – in binario – 1 codificato su 4 bit. Per spiegare cosa rappresentano i 4 bit rappresentati, affermiamo che i valori

1 e 17 sono **congrui** tra loro modulo 16. Nell'aritmetica modulare questo risultato è in effetti corretto, il resto delle divisioni tra 1 e 16 e tra 17 e 16 è proprio 1.

Tipi di rappresentazione

I tipi di rappresentazione utilizzati dai S.A. interi dei computer sono:

- Rappresentazione per segno e modulo
- Rappresentazione per complemento a 2
- Rappresentazione biased

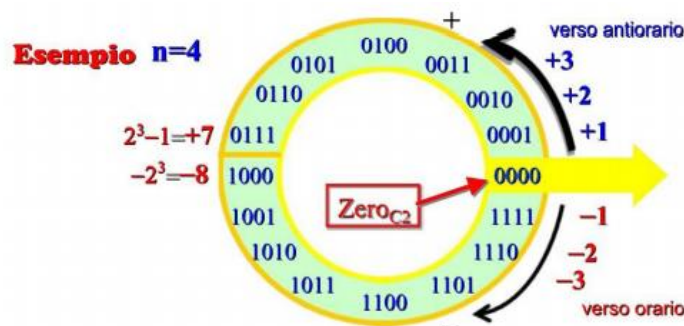
Le caratteristiche della rappresentazione per segno e modulo sono essenzialmente due: quella di considerare **due zeri** e quella di avere un **range simmetrico**. Gli altri due tipi di rappresentazione hanno un'unica rappresentazione di tutti i valori interi mediante sequenze consecutive di n bit nell'aritmetica modulo 2^n quindi il range non sarà simmetrico. Ciò che cambia tra le due rappresentazioni è solo la corrispondenza tra la sequenza di N bit di $\{0, 1, \dots, 2^n - 1\}$ e interi $\{i_{min}, i_{min} + 1, \dots, i_{max} - 1, i_{max}\}$.

Rappresentazione degli interi per segno e modulo

Essa viene usata per il campo mantissa di un numero reale floating-point. Per rappresentare interi ($\exists \mathbb{Z}$) simmetricamente negativi e positivi, la soluzione più semplice consiste nell'usare uno dei bit per il segno e gli altri per la rappresentazione in base a .

Rappresentazione degli interi per complemento a 2

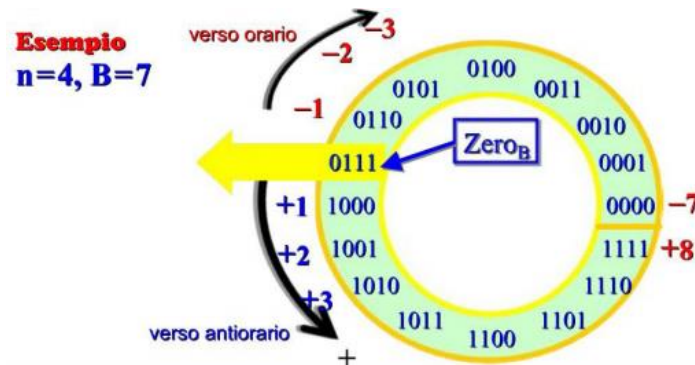
Essa viene usata per memorizzare i numeri interi con segno. In questa rappresentazione il valore 0 è associato alla sequenza di N zeri. I **numeri positivi** sono quelli che seguono lo 0 percorrendo la sequenza in verso antiorario. I **numeri negativi** sono quelli che precedono lo 0 percorrendo la sequenza in verso orario. Il range di valori rappresentabili è: $[-2^{n-1}, 2^{n-1} - 1]$, per cui **non è simmetrico**.



Rappresentazione per eccesso B (biased) degli interi

Essa viene usata per il campo esponente di un numero reale floating-point. Il valore 0 è rappresentato da bias B che è uguale a $B = 2^{n-1}$. Inoltre, la sequenza di N zeri rappresenta il valore $-B$. I **numeri positivi** sono quelli che seguono lo 0 percorrendo la sequenza in verso antiorario. I

numeri negativi sono quelli che precedono lo 0 percorrendo la sequenza in verso orario. Il range di valori rappresentabili: $[-(2^{n-1}), 2^{n-1}]$ che **non è simmetrico**.



Approfondimento sulle rappresentazioni per complemento a 2 e biased

Determinare la rappresentazione per complemento a 2 di un numero

La rappresentazione per complemento a 2 (r_{c2}) su n bit di un intero (rappresentabile) $k \in \mathbb{Z}$ si esprime con la formula: $r_{c2}(k) = [2^n + k]_{\text{mod}.2^n}$

Per intero $+k \geq 0$ coincide col numero stesso: $r_{c2}(k) = [2^n + (+k)]_{\text{mod}.2^n} = k$

Invece per intero $-k < 0$ è data da: $r_{c2}(-k) = [2^n + (-k)]_{\text{mod}.2^n} = 2^n - k$

In termini informatici invece, la rappresentazione per complemento a 2 di un numero negativo si può calcolare mediante gli operatori bitwise. Quello che occorre fare è sommare 1 al complemento del numero stesso:

- Si considera la rappresentazione binaria del numero
- Si complementa il numero positivo bit a bit
- Si addiziona 1
- Si taglia il risultato modulo 2^N

Operazioni aritmetiche in complemento a 2

La rappresentazione in complemento a 2 è la più usata nelle elaborazioni numeriche digitali per la facilità con la quale questa permette di eseguire le operazioni aritmetiche fondamentali. In un complemento a 2, l'addizione algebrica può essere sostituita da una semplice addizione in aritmetica modulare.

Rappresentazione per complemento alla base

La rappresentazione per complemento a 2 fa riferimento alla base 2, ma in realtà questo tipo di rappresentazione può essere introdotto qualunque sia la base del sistema di numerazione: si parla di **rappresentazione per complemento alla base**. Questo tipo di rappresentazione si basa sulle **cifre complementari**. Due cifre di un sistema in base β si dicono complementari tra loro quando la loro somma è $\beta - 1$. Per ottenere la rappresentazione per complemento alla base quando la base è -2 o +2, si possono usare gli operatori bitwise.

1. Si considera il valore assoluto del numero scritto su n bit
2. Si effettua il complemento a 1
3. Si addiziona 1
4. Si taglia il risultato modulo 2^n

Operazioni aritmetiche in rappresentazione B-biased

La rappresentazione biased di un numero k è ottenuta sommando a quel numero il valore $2^{N-1} - 1$, cioè $r_b(k) = k + Bias = k + 2^{N-1} - 1$. In questo tipo di operazione occorre sommare 2 numeri: h e k . Sono percorribili 2 strade:

- 1) Mi trovo prima la rappresentazione B-biased di h
 - a. Poi mi trovo la rappresentazione B-biased k
 - b. Infine, effettuo la somma tra h e k ; questa risulterà essere uguale a $h + k + 2B \rightarrow h + k + 2^N - 2$

Ma 2^N è congruo a 0, quindi si può eliminare somma: $h + k - 2$.

- 2) Effettuo prima la somma tra h e k , poi mi trovo la rappresentazione B-biased del risultato; questa risulterà $h + k + 2^{N-1} - 1$.

Correzione poiché non coincidono: $r_B(h \pm k) = r_B^{(h)} + r_B(\pm k) + \text{correzione} \rightarrow \text{correzione} = 2^{n-1} - 1$

Range degli interi in C

Char rappresenta un intero su $n = 8$ bit, il suo range è $[-128, +127]$.

Short int rappresenta un intero su $n = 16$ bit, il suo range è $[-32768, +32767]$.

Long int rappresenta un intero su $n = 32$ bit, il suo range è $[-2147483648, +2147483647]$.

Overflow un intero in C

Se consideriamo una variabile di tipo `char` il range è compreso tra -128 e +127. È abbastanza evidente che, essendo 128 superiore di una unità rispetto al massimo rappresentabile con 8 bit in complemento a 2 (+127), qualora si tenti di avanzare oltre il suddetto massimo si arriverà nella zona dei numeri negativi incontrando per prima -128.

Capitolo 3 – Tipo numerico reale

Per rappresentare i numeri reali viene usata la **notazione scientifica** che è la notazione più compatta in quanto vengono rappresentate solo le **cifre significative**. Una cifra è significativa quando, se eliminata, altera il valore originario. Una cifra **non è significativa** quando, se eliminata, non altera il valore originario. La notazione scientifica fa uso di **due oggetti** per rappresentare un numero reale (quando prefissata una base β):

- Mantissa m , costituita dalle cifre significative
- Esponente p , indica l'ordine di grandezza del numero, cioè indica gli zeri significativi.

Nella notazione scientifica un numero reale x si rappresenta tramite una mantissa m , contenente le cifre significative, ed un esponente p , indicazione degli zeri significativi, tali che $x = m * \beta^p$, dove β è la base del sistema di numerazione.

Normalizzazione della mantissa

Per uniformare la rappresentazione scientifica di un valore, si opera prima la **normalizzazione** della mantissa da cui discende il valore dell'esponente. Esistono fondamentalmente due **schemi di normalizzazione**. Nel **primo metodo** si fa sì che la prima cifra della mantissa sia diversa da 0; nel **secondo metodo** si fa sì che la prima cifra della mantissa sia uno 0, mentre la prima cifra diversa da 0 sia quella dopo il punto decimale. Il metodo adoperato nei moderni sistemi di elaborazione è il primo. Naturalmente i numeri normalizzati che richiedono la nostra attenzione sono quelli espressi nel sistema di numerazione binario. Facendo riferimento al sistema di numerazione binario accade che: nel **primo metodo** la prima cifra della mantissa dei numeri binari normalizzati è sempre 1, mentre nel **secondo metodo** la prima cifra significativa è sempre 0.

Questa conclusione fornisce la possibilità di evitare di memorizzare esplicitamente tale cifra significativa dando origine alla cosiddetta rappresentazione a bit implicito, che permette di guadagnare un bit. Ovviamente si tratta di un'eliminazione formale, ciò vuol dire che bisognerà tener conto della cifra "nascosta" nelle operazioni.

Il sistema aritmetico reale floating-point (concetti teorici)

Il **sistema aritmetico floating-point** denota l'insieme dei numeri reali rappresentati in un computer e le operazioni definite su di essi. Esso si indica con la **F** e dipende da alcuni parametri $(\beta, t, E_{\min}, E_{\max})$:

- β è la base del sistema di numerazione
- t è il numero di cifre β per la mantissa
- $E_{\min} < E_{\max}$, per quanto riguarda la limitazione per il campo esponente.

Formati dell'IEEE Standard 754

L'IEEE prevede due formati di cui uno opzionale che viene raramente implementato. Il formato basic prevede due tipi: il tipo **single** (a singola precisione, in C float) che occupa 32 bit e il tipo **double**

(doppia precisione, in C double) che occupa 64 bit. Il formato opzionale **extended double** occupa 80 bit ma non è sempre messo a disposizione dai compilatori.

Campi del formato BASIC

Nell'IEEE 754 un numero in formato basic è rappresentato in memoria attraverso 3 campi: **segno**, **esponente** e **mantissa**. Nel S.A. Std. IEEE std. $F(2, t, E_{\min}, E_{\max})$ un numero del formato Basic è rappresentato in memoria come (in ordine): segno (s), esponente (e), mantissa (m). Dove:

- **s** denota il segno della mantissa
- L'esponente **e** è rappresentato come "intero biased"
- La mantissa **m** è rappresentata, con s per segno e modulo, su t bit a bit implicito l^+ (precisione = t+1) ed è generata con lo schema del round to nearest;

Pertanto, il suo valore è: $x = (-1)^s [l.m] * 2^{e-Bias}$

Il **formato basic** dello standard definisce anche il numero di bit da assegnare a ciascun campo:

- Singola precisione [32 bit] [suddivisi] [1+8+23]
- Doppia precisione [64 bit] [suddivisi] [1+11+52]
- Extended [80 bit] [suddivisi] [1+15+64]

N.B. Quest'ultimo non prevede il bit implicito per la mantissa.

Aumentando i bit dell'esponente, si aumenta l'intervallo di rappresentabilità, cioè allontanano le situazioni di overflow ed underflow. Aumentando i bit della mantissa, si aumenta la densità dei numeri.

Rappresentazione sull'asse reale dei numeri floating-point

Vista la finitezza del sistema aritmetico floating-point vi saranno inevitabili fenomeni di **overflow ed underflow**. Inoltre, ci sarà una **distribuzione non uniforme** dei numeri rappresentabili. Visto che i numeri floating-point sono discreti esiste un certo "gap" (un vuoto) tra un valore floating-point e il successivo. Se si tenta di rappresentare un valore che è maggiore del massimo rappresentabile allora si verifica un **OVERFLOW**, mentre se si tenta di rappresentare un valore che è minore del minimo rappresentabile allora si verifica un **UNDERFLOW**.

Oggetti del Sistema Aritmetico Floating-Point

Oggetto	Caratterizzazione		l
	esponente e	mantissa m	
Numeri normalizzati valore = $(-1)^s [l.m] \times 2^{e-Bias}$	$E_{\min} < e < E_{\max}$	$m \geq 0$	1
Infinito con segno	$e = E_{\max}$	$m = 0$	-
NaN (Not A Number)	$e = E_{\max}$	$m \neq 0$	-
Zero con segno	$e = E_{\min}$	$m = 0$	-
Numeri denormalizzati valore = $(-1)^s [l.m] \times 2^{e-Bias+1}$	$e = E_{\min}$	$m \neq 0$	0

Gli oggetti rappresentabili non sono solo numeri.

Il **primo oggetto sono i numeri normalizzati**: sono i numeri reali rappresentati dal S.A. floating-point. Sono caratterizzati dall'avere una **mantissa non nulla** a prescindere dal segno, dal bit implicito =1 e dall'esponente strettamente compreso tra $E_{\min} < e < E_{\max}$.

Il **secondo oggetto è l'infinito con segno**: questo è caratterizzato dall'avere il campo esponente massimo e la mantissa nulla. Serve per individuare una situazione eccezionale come l'overflow di reale.

Il **terzo oggetto è il NaN (Not A Number)**: è caratterizzato dall'avere il campo esponente massimo e il campo mantissa non nullo. Questo è il risultato di output fornito dalla macchina quando l'operazione eseguita non è valida oppure quando il numero che si vuole rappresentare non è valido. In pratica è il NULL.

Il **quarto oggetto è lo Zero con segno**: esistono due zeri visto che la mantissa è rappresentata con segno e modulo. È caratterizzato dall'avere l'esponente minimo e la mantissa nulla.

Il **quinto oggetto sono i numeri denormalizzati**: servono per infittire il sistema aritmetico in prossimità dello 0. Questi numeri sono caratterizzati dall'avere l'esponente minimo, mantissa non nulla e il bit implicito = 0.

Il valore di un numero denormalizzato ha una formula leggermente diversa dai numeri normalizzati. Ciò che cambia è l'ordine di grandezza che si ottiene con $e - \text{Bias} + 1$.

Sistema Aritmetico Standard IEEE 754 (concetti pratici)

Estrazione di alcuni bit dal contenuto di una variabile

Codice

```
#include <stdio.h>
#define bias 127
short estrae_esp1(long);
void main() {
    union sp {
        float x;
        long n;
    } f;
    short xesp;
    scanf("%f", &f.x);
    xesp = estrae_esp1(f.n);
    printf("esp = %d", xesp);
}
short estrae_esp1(long n) {
    //estrae l'esponente
    long int xesp, mask;
    mask = 0x7f800000;
    xesp = n & mask;
    xesp = xesp >> 23;
    xesp = xesp - bias;
```

```

    return (short)xesp;
}

```

Visualizzazione esadecimale di variabili reali

Codice

```

#include <stdio.h>
#define MAX_LEN 64
void estrae_64_bit(short, char[], short[]);
void mostra_sp(long);
void mostra_dp(long []);
void main() {
    short k, bit[MAX_LEN];
    union sp {
        float fa;
        long la;
        char C[4];
    } a;
    union dp {
        double db;
        long lb[2];
        char C[8];
    } b;
    scanf("%le", &b.db); a.fa = (float)b.db;
    printf("\n\nfloat= %e", a.fa); mostra_sp(a.la);
    estrae_64_bit(sizeof(a.fa), a.C, bit);
    printf("\tbit = ");
    for (k=31; k>=0; k--)
        (k==31 | k==23) ? printf("%1d", bit[k]) : printf("%1d", bit[k]);
    printf("\n\ndouble= %e,", b.db); mostra_db(b.lb);
    estrae_64_bit(sizeof(b.db), b.C, bit);
    printf("\tbit = ");
    for (k=63; k>=0; k--)
        (k==63 | k==52) ? printf("%1d", bit[k]) : printf("%1d", bit[k]);
}
void mostra_sp(long n) {
    printf("float esadecimale = %08x", n);
}
void mostra_dp (long n[]) {
    printf("double esadecimale = %08x %08x", n[1], n[0]);
}
void estrae_64_bit(short len, char ch[], short bit[MAX_LEN]) {
    //...
}

```

Visto che aumenta il campo esponente da 8 a 11 bit, quindi il range aumenta da $[-127, 128]$ a $[-1023, +1024]$, allora aumenta l'intervallo di rappresentabilità. Visto che aumenta il campo mantissa da 23 a 52 bit, allora aumenta la densità dei numeri.

Numeri normalizzati	$E_{\min} < e < E_{\max}$	$m \geq 0$
---------------------	---------------------------	------------

+1	s.p. (32bit)	3f800000	
		0 011 1111 1	000 0000 0000 0000 0000 0000
	d.p. (64bit)	3ff00000 00000000	
		0 011 1111 1111	0000 0000 0000 0000 0000

-1	s.p. (32bit)	bf800000	
		1 011 1111 1	000 0000 0000 0000 0000 0000
	d.p. (64bit)	bff00000 00000000	
		1 011 1111 1111	0000 0000 0000 0000 0000

Estrazione dei campi segno, esponente e mantissa da un float

```
#include <stdio.h>
#include <stdlib.h>
#include <float.h>
#include <math.h>

void estrae_bit(long, char[32]);

void main() {
    char i, bit[32];
    union basic_single {
        float fa;
        long la;
    } a;
    a.f = ... //definisce float di cui visualizzare i bit
    estrae_bit(a.la, bit);
    printf("\tfloat = %e\n", a.f);
    printf("\tlong hex = %08x\n", a.la);
    puts("bit corrispondenti\tsegno esponente mantissa");
    printf("\t\t\t%1d", bit[0]);
    for (i=1; i<=8; i++) printf("%1d", bit[i]);
    printf(" ");
    for (i=9; i<32; i++) printf("%1d", bit[i]);
    puts("\n");
```



```

}
void estrae_bit(long reg, char B[32] {
    /* rappresentazione binaria del long reg nell'array B bit + significative <- bit meno
    significative.
    short i;
    for (i=31; i>=0; i--) {
        B[i]=(char)(1 & reg);
        reg=reg>>1;
    }
}

```

a.fa = FLT_MAX;

float = 3.402823e+038
long hex = 7f7fffff

bit corrispondenti segno esponente mantissa
0 11111110 111111111111111111111111

**FLT_MAX, FLT_MIN:
variabili predefinite**

massimo numero normalizzato rappresentabile

In questo caso è possibile dire che ci si trova dinanzi al massimo numero rappresentabile. Si può notare che il numero in questione è chiaramente normalizzato visto che l'esponente non è formato da tutti i bit 1. Dunque, l'esponente è quello massimo di un numero ancora rappresentabile senza scatenare fenomeni di overflow, è positivo e con una mantissa che, trattandosi di numero normalizzato, ha bit implicito 1 e che quindi è anch'essa la massima possibile.

a.fa = FLT_MIN;

float = 1.175494e-038
long hex = 00800000

bit corrispondenti segno esponente mantissa
0 00000001 000000000000000000000000

minimo numero normalizzato rappresentabile

Viene dato in input il valore minimo reale rappresentabile in singola precisione. Osservando l'esponente notiamo subito che composto da tutti i bit 0 tranne uno, quindi il numero è chiaramente normalizzato. Osservando la mantissa notiamo che contiene tutti i bit 0, quindi visto che si tratta di un numero normalizzato, la mantissa non è nulla ma contiene il bit implicito.

a.fa = FLT_MIN/pow(2,23);

float = 1.401298e-045
long hex = 00000001

bit corrispondenti segno esponente mantissa
0 00000000 000000000000000000000001

minimo numero denormalizzato

a.fa = FLT_MIN/pow(2,24);

bit corrispondenti segno esponente mantissa
0 00000000 000000000000000000000000

underflow

Tipo Re
(pr. Rizzordi)

Si vuole rappresentare il più piccolo numero floating-point denormalizzato, cioè quella che è effettivamente la soglia di underflow. Questa formula è stata ottenuta partendo dal FLT_MIN la cui mantissa prevede solo il bit implicito, e si è sposato quest'1 di tanti posti fino a fargli assumere la posizione del bit meno significativo della mantissa. Possiamo dire che non è possibile rappresentare un numero più piccolo di questo valore perché verrebbe prodotto un underflow e quindi il valore

che troveremo sarebbe 0. Questa situazione è possibile osservarle nel 2,24 dove viene prodotto un underflow.

ESEMPIO 3: Zero con segno (bit implicito=0)

+0	s.p.	00000000
	(32bit)	0 000 0000 0 000 0000 0000 0000 0000 0000
	d.p.	00000000 00000000
	(64bit)	0 000 0000 0000 0000 0000 0000 0000 0000
-0	s.p.	80000000 (*)
	(32bit)	1 000 0000 0 000 0000 0000 0000 0000 0000
	d.p.	80000000 00000000 (**)
	(64bit)	1 000 0000 0000 0000 0000 0000 0000 0000

(*) per esempio **a . fa = -DBL_MIN**

(**) per esempio **b . db = -DBL_MIN/DBL_MAX**

Ci dimostra che è possibile rappresentare lo zero col segno negativo. Lo zero negativo si può generare forzando il verificarsi di un underflow:

- Singola precisione a.fa = -DBL_MIN
- Doppia precisione b.db = -DBL_MIN/DBL_MAX

ESEMPIO 4: Infinito con segno (affine mode)

Infinito con segno		e = E _{max}		m = 0	
+INF	s.p.	7f800000 (+1. #INF00e+000)			
	(32bit)	0	111 1111 1	000 0000 0000 0000 0000 0000	
	d.p.	7ff00000 00000000 (+1. #INF00e+000)			
	(64bit)	0	111 1111 1111	0000 0000 0000 0000 0000	
-INF	s.p.	ff800000 (-1. #INF00e+000)			
	(32bit)	1	111 1111 1	000 0000 0000 0000 0000 0000	
	d.p.	fff00000 00000000 (-1. #INF00e+000)			
	(64bit)	1	111 1111 1111	0000 0000 0000 0000 0000	

per es. a . fa = -FLT_MAX/FLT_MIN
b . db = -DBL_MAX/DBL_MIN

a float = -1. #INF
b double = -1. #INF


L'infinito negativo si può generare forzando il verificarsi di un overflow:

- Singola precisione a.fa = -FLT_MAX/FLT_MIN
- Doppia precisione b.db = -DBL_MAX/DBL_MIN

NaN (Not A Number)		$e = E_{\max}$	$m > 0$
--------------------	--	----------------	---------

+NaN	s.p.	7fc00000	
	(32bit)	0 111 1111 1	100 0000 0000 0000 0000 0000
	d.p.	7ff80000 00000000	
	(64bit)	0 111 1111 1111	1000 0000 0000 0000 0000

per es. a. fa=+FLT_MAX/FLT_MIN-FLT_MAX/FLT_MIN
 b. db=+DBL_MAX/DBL_MIN-DBL_MAX/DBL_MIN



forma indeterminata $+\infty - \infty$

a float = 1.#QNAN0e+000
b double = 1.#QNAN0e+000

-NaN	s.p.	ffc00000	
	(32bit)	1 111 1111 1	100 0000 0000 0000 0000 0000
	d.p.	fff80000 00000000	
	(64bit)	1 111 1111 1111	1000 0000 0000 0000 0000

Qui ci sono le rappresentazioni dei NaN che sono caratterizzati dall'avere il campo esponente pari al valore massimo consentito e la mantissa non nulla. Il NaN può essere rappresentato sia col segno positivo che col segno negativo. Uno dei modi per generare un +NaN è quella di provocare la forma indeterminata $+\infty - \infty$. Mentre uno dei modi per generare un -NaN è quello di provocare un'operazione non valida.

ESEMPIO 6a: Numeri denormalizzati (bit implicit=0) in singola precisione

Numeri denormalizzati	$e = E_{\min}$	$m > 0$
-----------------------	----------------	---------

Per i denormalizzati il valore dell'esponente (costante) è dato da: $e - Bias + 1$ (-126 in single)

FLT_MIN/2 = $5.8...e-39_{10} = 00400000_{16}$		
0	000 0000 0	100 0000 0000 0000 0000 0000
FLT_MIN/4 = $2.9...e-39_{10} = 00200000_{16}$		
0	000 0000 0	010 0000 0000 0000 0000 0000
FLT_MIN/8 = $1.4...e-39_{10} = 00100000_{16}$		
0	000 0000 0	001 0000 0000 0000 0000 0000

Questa è la rappresentazione in memoria dei numeri denormalizzati che sono caratterizzati dall'aver il campo esponente pari al minimo ma mantissa diversa da zero. Un numero denormalizzato si ottiene per esempio dividendo FLT_MIN / 2 perché "flt_min" è una costante che indica il valore minimo normalizzato e suddividendolo per 2 otteniamo un numero denormalizzato. La rappresentazione binaria ha il campo esponente nullo mentre la mantissa ha un bit diverso da 0.

$FLT_{MIN}/2^{23}$ rappresenta la soglia di underflow.

Schemi di Rounding

Abbiamo già detto precedentemente che l'insieme dei numeri rappresentabili è finito e discreto. Se consideriamo un numero x reale che cade nell'intervallo di rappresentabilità ma che non corrisponde ad uno dei numeri floating-point, **come possiamo associargli un corrispondente floating-point?** Qualora un numero non sia precisamente rappresentabile con la notazione adoperata, lo standard prevede delle tecniche per associare comunque, a questo numero una sua rappresentazione in memoria. Si tratta banalmente di un problema di "approssimazione" che può essere risolto in modi diversi.

Ad ogni **numero reale rappresentabile** viene associato il suo rappresentante floating-point mediante uno schema di rounding. Il S.A. Standard IEEE prevede 4 schemi di rounding:

- (RN) round to nearest [approssimazione al più vicino]
- (RZ) round toward 0 [corrisponde al troncamento]
- (RM) round toward $-\infty$ [arrotondamento verso meno infinito]
- (RP) round toward $+\infty$ [arrotondamento verso più infinito]

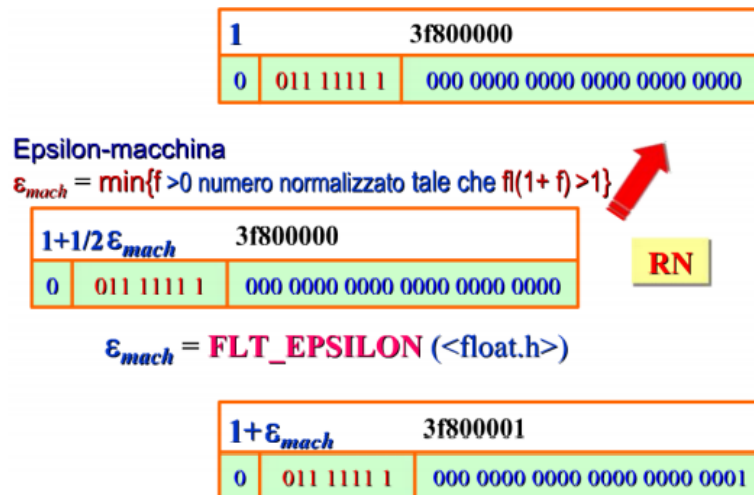
È possibile verificare che queste quattro tecniche danno luogo, in generale, a risultati diversi; per cui è impossibile adoperare una strategia in sostituzione di un'altra in queste non sono equivalenti. Lo schema **RN (Round to nearest)** è l'**arrotondamento** in base al quale il valore del primo bit della mantissa di x da eliminare [il $(t + 2)^{simo}$] influenza la mantissa di $fl(x)$: se questo bit è 1 allora si aggiunge 1 al bit meno significativo della mantissa di $fl(x)$.

In pratica, questo è uno schema che serve per ridurre la mantissa da un numero infinito di bit ad un numero finito di bit pari a $t+1$. Per realizzare questa operazione va ad esaminare il valore del primo bit da escludere, cioè il $t+2$ esimo:

- se il suo valore è 1, allora aggiunge 1 al bit meno significativo della mantissa
- se il suo valore è 0, allora non aggiunge niente (si comporta quindi come il troncamento)

Per evitare errori sistematici, nel caso in cui la mantissa di x sia equidistante dalle mantisse di due numeri floating-point consecutivi, il round to nearest la approssima con la mantissa che tra le due è pari.

Per **numero pari** intendiamo un numero che termina con bit 0. Per **numero dispari** intendiamo un numero che termina con bit 1. **Epsilon** individua il più piccolo reale rappresentabile tale che sommato ad 1 dia risultato maggiore di 1 stesso.



Errori di Roundoff

L'errore, detto **errore di roundoff**, è provocato dalla finitezza della mantissa. La finitezza dell'esponente provoca solo overflow o underflow. L'errore di roundoff si classifica in:

- **roundoff statico**, che dipende dalla rappresentazione in memoria dei numeri reali
- **roundoff dinamico**, che dipende dall'esecuzione delle operazioni aritmetiche sui reali.

Misure di errore

Per ottenere informazioni aggiuntive sul grado di correttezza dell'approssimazione di un reale solitamente si considerano due **misure di errore**: errore assoluto ed errore relativo. Se x indica il valore esatto e \tilde{x} una sua approssimazione ($\tilde{x} = fl(\tilde{x})$), per misurare l'accuratezza di \tilde{x} rispetto a x si usano:

- Errore assoluto $E_A(\tilde{x}) = |x - \tilde{x}|$
- Errore relativo $E_R(\tilde{x}) = \left| \frac{x - \tilde{x}}{x} \right|$ per $x \neq 0$

L'errore assoluto non è altro che lo scarto in valore assoluto tra l'approssimazione e il valore esatto. **L'errore relativo**, definito solo per i valori esatti non nulli, è una frazione che al numeratore contiene l'errore assoluto e al denominatore il valore assoluto del numero esatto. Calcolare l'**errore assoluto** ci fornisce il numero di cifre decimali corrette; l'**errore relativo**, indica il numero di cifre significative corrette.

Supponiamo che $X = 12.34567$ che si può scrivere come $0.1234567 * 10^2$ (notazione scientifica). Osservando la notazione scientifica, si può notare che:

- L'esponente ci dà l'ordine di grandezza
- La mantissa ci dà le cifre significative

Osservando la notazione decimale, si può notare che:

- Le cifre 12 sono le cifre intere
- Quelle a destra del punto frazionario sono le cifre decimali

Ci sono 2 proprietà della matematica che collegano un valore esatto e la sua approssimazione:

1. **Proprietà 1:** se \tilde{x} è un'approssimazione di x corretta a p cifre decimali, allora si ha $|x - \tilde{x}| < 10^{-p}$
2. **Proprietà 2:** se \tilde{x} è un'approssimazione di x corretta a p cifre significative, allora si ha $\frac{|x - \tilde{x}|}{|x|} < 10^{-p+1}$

Anche se le due proprietà precedenti valgono per una sola implicazione, nella pratica si suppone l'equivalenza, nel senso che dall'ordine di grandezza degli errori si hanno informazioni sulle cifre (decimali o significative) corrette di un'approssimazione rispetto al corrispondente valore esatto.

Codice

```
//Calcola gli errori di rappresentazione del tipo float
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
void main() {
    double double_x, rel_err, ass_err;
    float float_x; char i;
    for (i=-4; i<5; i=i+2) {
        double_x = 4*atan(1.0)*pow(10, i);
        float_x = (float) double_x;
        ass_err = fabs(double_x - float_x); //fabs dà il valore assoluto di un numero
        rel_err = ass_err / fabs(double_x);
        printf("double = %22.16e\n", double_x);
        printf("single = %22.16e\n", float_x);
        printf("errore assoluto = %22.16e\n", ass_err);
        printf("errore relativo = %22.16e\n", rel_err);
        puts("\n\n");
    }
}
```

Precisione decimale equivalente

La precisione è il numero di cifre β significative rappresentate (per i numeri reali si tratta della mantissa). La rappresentazione (binaria) floating-point a bit implicito su t bit per la mantissa è

equivalente alla precisione (binaria) = $t+1$. La precisione decimale equivalente (o mantissa):

$$-\log_{10} \left(\frac{|x-f(x)|}{|x|} \right) \geq -\log_{10} \left(\frac{\epsilon_{match}}{2} \right)$$

Accuratezza statica

L'errore introdotto nel passare da un numero reale a precisione infinita al suo rappresentante floating-point prende il nome di **accuratezza statica**. Naturalmente l'accuratezza statica dipende dalla precisione del S.A. floating-point e dallo schema di rounding utilizzato. Solitamente per ridurre gli errori di Roundoff si agisce sugli schemi di rounding, scegliendo tra questi quello che si ritiene il migliore. In linea generale è possibile affermare che il Round to Nearest è senz'altro il migliore tra gli schemi di rounding adoperabili in quanto fornisce solitamente l'errore assoluto minore, di conseguenza è possibile ritenerlo il più affidabile.

U.L.P. – Unit in the Last Place

Dalla matematica è noto che, nell'insieme dei numeri reali, se si somma ad un qualsiasi numero reale A , una quantità positiva, il valore di A aumenta. [Proprietà: $\forall a \in \mathbb{R}, \forall \epsilon > 0 \rightarrow a + \epsilon > a$]

Nel sistema floating-point questa proprietà non è sempre verificata. Esistono infatti quantità che non sortiscono cambiamenti se sommate o che in altri termini, non forniscono alcun contributo.

$$[\forall a \in \mathbb{F}(\beta, t, E_{min}, E_{max}), \exists \epsilon > 0 : a(+) \epsilon = a]$$

Per ogni numero A del S.A. floating-point F esiste un valore di ϵ positivo tale che l'addizione floating-point tra ϵ ed A restituisce A . Questo valore prende il nome di u.l.p. (unit in the last place) ed è definito come il **minimo ϵ positivo appartenente** al S.A. floating-point tale che $(A+\epsilon) > A$. [$ulp(a) = \min\{0 < \epsilon \in \mathbb{F}(\beta, t, E_{min}, E_{max}) : a(+) \epsilon > a\}$ *u.l.p. = unit in the last place*]

Importante: differenza tra epsilon (ϵ) e soglia di underflow. L' ϵ è quel più piccolo floating-point tale che $A + \epsilon > A$. La soglia di underflow è quel più piccolo floating-point oltre il quale c'è solo lo 0.

L'ULP di 1 prende il nome di **epsilon machina**.

Interpretazione geometrica dell'epsilon macchina

L'**epsilon macchina** è la metà dell'ampiezza dell'intervallo; tutto ciò è dovuto alla proprietà del Round to Nearest che approssima il valore a quello più prossimo. In definitiva possiamo dire che l'ulp di X è dato dall'epsilon macchina, cioè il valore ottenuto in corrispondenza di 1, moltiplicato per l'ordine di grandezza di X (2^e).

Risultato di massima accuratezza

Nel S.A. standard, se X è un numero reale rappresentabile e $FL(X)$ indica il corrispondente numero floating-point, si dice che $FL(X)$ è un risultato di massima accuratezza se si può scrivere $FL(X)$ come il prodotto tra X (valore esatto) e $1+\delta$ (errore relativo di $FL(X)$) dove l'errore relativo risulta essere **minore o uguale di $\frac{1}{2}$ epsilon macchina**. [Dove $\sigma = \frac{|x-FL(x)|}{|x|}$, σ errore relativo]

Quando si verifica questa disuguaglianza si parla di risultato di **massima accuratezza**, cioè l'errore relativo in un risultato di massima accuratezza è al più $\frac{1}{2}$ dell'epsilon macchina.

Il S.A. IEEE 754, mediante il bit implicito (rappresentazione della mantissa a bit implicito) e lo schema del Round to Nearest, garantisce la massima accuratezza nella rappresentazione in memoria dei numeri reali, cioè garantisce la **massima accuratezza statica**.

Accuratezza dinamica: si dimostra che nei registri dell'unità aritmetica per assicurare risultati di massima accuratezza sono sufficienti:

- 1 bit per rappresentare il bit implicito (I)
- 2 guard-bit (g) //sfruttati per ridurre l'Errore di Roundoff
- 1 sticky-bit //bit che vale 1 se per esso transita almeno un bit=1

L'ULP di 1, detto epsilon macchina, è definito come il più piccolo floating-point che sommato ad 1, fornisce un risultato maggiore di 1. Il linguaggio C è dotato di due variabili predefinite, FLT_epsilon e DBL_epsilon, che forniscono il valore dell'epsilon macchina per il tipo a singola precisione (float) e per quello a doppia precisione (double).

Solitamente le operazioni che coinvolgono quantità inferiori all'epsilon macchina (l'ulp di 1) sono evitate, in quanto si tradurrebbero solo in una perdita di tempo. Tale eventualità è descritta dal cosiddetto **criterio di arresto naturale**.

Quando un algoritmo contiene un modulo per calcolare una somma di molti addendi (prodotti scalari, formule di quadratura, ecc.), in assenza di algoritmi specifici, conviene eseguire la computazione intermedia ad una precisione maggiore.

Bisogna evitare quando possibile la cancellazione nelle somme algebriche.

Nella somma iterativa conviene usare il **criterio di arresto naturale** per evitare somme inutili di addendi non significativi rispetto a S.

Capitolo 4 – Stringhe

Differenze tra tipo carattere e stringa di caratteri in C

Prima di concentrare la nostra attenzione sul tipo **stringa di caratteri** nel linguaggio C, conviene fare un breve richiamo su cosa si intende per **carattere** e cosa si intende per **stringa di caratteri**. La sintassi del linguaggio prevede due notazioni diverse per i caratteri e per la stringa di caratteri. Infatti:

- 'a' indica una **costante carattere**. È un intero memorizzato su 1 byte contenente il valore del codice ASCII corrispondente al carattere.
- "a" indica una **costante stringa**. È una sequenza di (zero o più) caratteri memorizzata come array di caratteri la cui fine è individuata dal carattere di fine stringa '\0'.

Function che restituisce il nome del mese

```
#include <stdio.h>
void nome_mese(int n, char *);
void main() {
    int n; char mese_di_n[30];
    printf("numero mese = "); scanf("%d", &n);
    nome_mese(n, mese_di_n);
    printf("mese corrispondente = %s\n", mese_di_n);
}
void nome_mese(int n, char mese[]) {
    char mesi[][30] = {"Numero di mese non corretto", "Gennaio", "Febbraio", "Marzo",
    "Aprile", "Maggio", "Giugno", "Luglio", "Agosto", "Settembre", "Ottobre", "Novembre",
    "Dicembre"};
    short j, riga;
    if (n<1 || n>12)
        riga = 0;
    else
        riga = n;
    for (j=0; j<30; j++)
        mese[j] = mesi[riga][j];
}
```

I/O di caratteri e di stringhe

Getchar e Putchar

Per l'input da tastiera si usa la funzione **getchar** mentre per l'output si usa la funzione **putchar**. In particolare:

- Per l'input è predisposta la funzione **int getchar(int)**
- Per l'output è predisposta la funzione **int putchar(int)**

Si noti che, nonostante occorra operare su dei caratteri, i prototipi delle due funzioni lavorano di fatto su dati di tipo intero.

Codice

```
#include <stdio.h>
void main() {
    char buffer[61]; int c, j;
    j=0;
    do {
        c = getchar();
        buffer[j] = (char)c;
        j++;
    } while (c != '\n');
    buffer[j-1]='\0'; //toglie '\n'
    printf("la stringa %s e' lunga %d caratteri\n", buffer, strlen(buffer));
}
```

Nella libreria string.h esistono function di sistema per l'input e l'output di intere stringhe: rispettivamente la **gets()** e la **puts()**.

Codice

```
#include <stdio.h>
#include <string.h>
int main() {
    char stringa[10], c32=32;
    puts("immetti stringa ...");
    fflush(stdin); //ripulisce il buffer di input
    gets(stringa); //stesso di scanf("%s", stringa);
    puts("strlen = "); printf("%d\n", strlen(stringa));
    puts("la stringa e' ..."); printf("%s\n", stringa);
    stringa[0] = stringa[0]^c32; //tramite bitwise, scrive la maiuscola
    puts("la stringa e' ... ");
    printf("%s\n", stringa);
}
```

Gestione delle stringhe mediante puntatori: allocazione dinamica

È possibile gestire le stringhe tramite puntatori mediante allocazione dinamica. Quando si usa la funzione **malloc()**, la memoria allocata viene sottratta dall'**heap**, cioè una delle quattro sezioni in cui si suddivide la memoria occupata da un programma, creato appositamente per consentire l'allocazione dinamica della memoria. Se l'allocazione è andata a buon fine il puntatore restituito da **malloc()** punta all'area effettivamente ottenuta; se, invece, l'allocazione non è riuscita, la **malloc()** restituisce un puntatore nullo (**NULL**).

Codice

```
#include <stdio.h>
#include <string.h>
void main() {
    char *p_string; int stringlen;
    printf("\n lunghezza stringa = "); scanf("%d" stringlen);
    p_string = (char *)malloc(stringlen+1); //alloca un blocco di memoria adeguato
    printf("\n immetti stringa = ");
    fflush(stdin);
    gets(p_string); //stesso di scanf("%s", p_string));
    printf("\n strlen(p_string) = %d\n", strlen(p_string));
    printf("\n stringa immessa = %s\n", p_string);
    free(p_string); //libera l'area di memoria allocata precedentemente con malloc
}
```

Function della libreria <string.h>

I prototipi delle principali funzioni per la manipolazione di stringhe sono contenuti nella libreria **string**, il cui file di intestazione (**string.h**) va ovviamente incluso nel sorgente principale.

- **Strlen**(ps), restituisce la lunghezza (senza contare il carattere \0) di *ps
- **Strcpy**(pt, ps), copia *ps in *pt compreso il carattere \0
- **Strcat**(s1, s2), concatena ad *s1 la stringa *s2
- **Strcmp**(s1, s2), confronta *s1 e *s2, restituisce un valore < 0 se *s1 < *s2
- **Strchr**(pt, pc), restituisce un puntatore alla prima occorrenza del carattere *pc in *pt
- **Strstr**(pt, ps), restituisce un puntatore alla prima occorrenza della stringa *ps in *pt

Capitolo 5 – Allocazione dinamica in C

L'**allocazione statica** della memoria permette di riservare uno spazio di memoria fisso (definito a priori) per i dati del programma. Con questa allocazione si definisce un quantitativo fisso della memoria che viene riservato al programma prima che questo venga eseguito e viene liberato nel momento in cui l'esecuzione ha termine.

L'**allocazione dinamica** della memoria dà la possibilità di allocare e deallocare spazio di memoria durante l'esecuzione del programma. Questa allocazione permette di occupare memoria a seconda delle necessità.

Funzioni C per l'allocazione dinamica

Le funzioni C per l'allocazione dinamica della memoria hanno prototipi definiti nella libreria **stdlib** il cui file di intestazione **<stdlib.h>**.

malloc(...)	Alloca blocchi di memoria per un oggetto	Le prime due funzioni permettono l'allocazione dinamica, con lievi differenze, di blocchi di memoria ad un dato.
calloc(...)	Alloca blocchi di memoria per un array di oggetti (elementi inizializzati a 0)	
realloc(...)	Rialloca blocchi di memoria allocati prima con malloc(...) o calloc(...)	La funzione realloc() posiziona un blocco di memoria precedentemente allocato all'interno di una nuova area di memoria di dimensione solitamente diversa da quella originale.
free(...)	Dealloca (libera) blocchi di memoria	La funzione free() libera lo spazio allocato con le funzioni malloc() o calloc() rendendolo nuovamente disponibile per altri utilizzi.

Esempio di utilizzo realloc

```
float *b; int n = 10;
b = (float *)calloc(n, sizeof(float)); //alloca un array di n component reali
b = (float *)realloc(b, (2*n)*sizeof(float)); //raddoppia il numero delle componenti
```

Allocazione dinamica di matrici in C

Richiami sul tipo matrice in C

type A [n][m]

Alloca uno spazio fisso di memoria ad un array bidimensionale del tipo specificato. L'idea che comunemente si associa ad una dichiarazione di questo tipo è quella di una "tabella di dati", nella quale individuiamo elementi contraddistinti da un indice di riga e un indice di colonna. Non esistono

“colonne” o “righe” di dati, ma l’ordine con il quale gli elementi delle matrici statiche sono memorizzati dipende dal linguaggio di programmazione in uso: il **C organizza i dati per righe, quindi una matrice statica in C è allocata per righe.**

Una matrice è allocata in locazioni consecutive. In particolare, se andiamo ad esplorare il contenuto della memoria, troveremo che gli elementi della prima riga della matrice precedono quelli della seconda, che a loro volta precedono quelli della terza, e così via. Ovviamente, lo stesso elemento di una matrice, in memoria, occuperà posizione diversa che si abbia un’organizzazione per righe o per colonne.

Allocazione in memoria di matrici dinamiche

Sfruttando l’allocazione dinamica è possibile in C:

- Dichiarare matrici non statiche e quindi allocare memoria solo quando necessario
- Superare la limitazione del C per quanto concerne la memorizzazione per righe, disponendo che una matrice sia organizzata per colonne

`type *pa;`

Questo comporta che il riferimento $A[i][j]$ che si riferisce ad una tabella bidimensionale, va tradotto su array monodimensionale, e quindi si devono passare due indici i, j al singolo indice k che indica la sua posizione all’interno del “nastro” che rappresenta l’array nella memoria. L’indice k varia a seconda che la matrice sia stata allocata per righe o colonne.

Matrice memorizzata per righe

All’elemento di posto $A[i,j]$ si fa corrispondere il contenuto della locazione puntata da $*(pa+i*n+j)$, dove:

- ***pa** è il puntatore all’indirizzo base dell’array
- **n** è il numero delle colonne della matrice e corrisponde al numero di elementi di ogni riga
- **i** è il numero di righe che precedono la riga dove si trova l’elemento che ci interessa
- **j** è il numero di colonne che precedono la colonna dove si trova l’elemento che ci interessa
- **i*n** indica il numero di componenti che sicuramente precedono l’elemento che ci interessa
- ***pa+i*n** punta alla prima componente della matrice che si trova sulla stessa riga dell’elemento che ci interessa, **j** indica il numero di posti di cui bisogna spostarsi su questa riga per ottenere l’elemento che ci interessa

In pratica, $i*n$ consente di posizionarsi sul primo elemento della riga in cui si trova l’elemento che ci interessa e j è un “offset” che ci consente di spostarci su questa riga per ottenere l’elemento che ci interessa.

Matrice memorizzata per colonne

All’elemento di posto $A[i,j]$ si fa corrispondere il contenuto della locazione puntata da $*(pa + j*m + i)$, dove:

- **m** è il numero delle righe della matrice e corrisponde al numero di elementi colonna
- **j*m** indica il numero di componenti che sicuramente precedono l'elemento che ci interessa
- ***pa+j*m** punta alla prima riga della matrice che si trova nella stessa colonna dell'elemento che ci interessa, **i** indica il numero di righe di cui bisogna spostarsi su questa colonna per ottenere l'elemento che ci interessa.

In pratica, **j*m** consente di posizionarci sul primo elemento della colonna in cui si trova l'elemento che ci interessa e **i** è un "offset" che ci consente di spostarci lungo la colonna per ottenere l'elemento che ci interessa.

Esempio C: aggiunta di una riga in una matrice dinamica

Per fare ciò, bisogna utilizzare le funzioni:

- MEMMOVE (move block of memory)

```
void *memmove(void *destination, const void *source, size_t num);
```

Copia i valori dei "num" byte della locazione puntata da "source" ad un blocco di memoria puntato da "destination". La copia avviene come si stesse usando un buffer intermedio, per cui garantisce la sovrascrittura dei dati se il blocco "destination" coincide con quello "sorgente".

- MEMCPY (copy block of memory)

```
void *memcpy(void *destination, const void *source, size_t num);
```

Copia i valori dei "num" byte dalla locazione puntata "source" direttamente nel blocco di memoria puntato da "destination". Quindi, a differenza di MEMMOVE, non dà alcuna garanzia per la sovrascrittura dei dati.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define M 4
#define N 8
void main() {
    int a[M][N], *pa, *row;
    short i, j, irow;
    row = (int *)calloc(N, sizeof(int)); //azzerà *row
    printf("\nnumero riga da inserire tra 1 e %d =", M);
    scanf("%d", &irow);
    irow--;
    srand((unsigned) time(NULL)); //inizializza il seed del generatore di numeri casuali
    printf("RAND_MAX=%d\n", RAND_MAX);
    printf("\nmatrice statica A:\n");
    for (i = 0; i < M; i++) {
        for (j = 0; j < N; j++) {
```

```

        a[i][j] = rand()*100/RAND_MAX;
        printf("%d\t", a[i][j]);
    }
    printf("\n");
}
pa = malloc(M*N*sizeof(int));
printf("\nmatrice dinamica *pa prima:\n");
for (i=0; i<M; i++) {
    for (j=0; j<N; j++) {
        *(pa+i*N+j) = a[i][j];
        printf("%d\t", *(pa+i*N+j));
    }
    printf("\n");
}
pa=realloc(pa, (M+1)*N*sizeof(int)); //aumenta le dimensioni
memmove(pa+(irow+1)*N, pa+irow*N, (M-irow)*N*sizeof(int)); /* sposta la sottomatrice
inferiore*/
memcpy(pa+irow*N, row, N*sizeof(int)); //copia la riga
printf("\nmatrice dinamica *pa dopo:\n");
for (i=0; i<=M; i++) {
    for (j=0; j<N; j++) {
        printf("%d\t", *(pa+i*N+j));
        printf("\n"); }
}
}

```

Esempio C: eliminazione di una riga in una matrice dinamica

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define M 6
#define N 8
void main() {
    int *pa, *row;
    short i, j, irow;
    printf("\nnumero riga da eliminare tra 1 e %d =", M);
    scanf("%d", &irow);
    irow--;
    pa = malloc(M*N*sizeof(int));
    srand((unsigned) time(NULL));
    printf("\nmatrice dinamica *pa prima:\n");
    for (i=0; i<M; i++) {
        for (j=0; j<N; j++) {
            *(pa+i*N+j) = rand()*100/RAND_MAX;

```

```

        printf("%d\t", *(pa+i*N+j));
    }
    printf("\n");
}
memmove(pa+irow*N, pa+(irow+1), (M-irow)*N*sizeof(int)); /* sposta la sottomatrice
inferiore*/
pa=realloc(pa, (M-1)*N*sizeof(int)); //diminuisce le dimensioni
printf("\nmatrice dinamica *pa dopo:\n");
for (i=0; i<=M; i++) {
    for (j=0; j<N; j++) {
        printf("%d\t", *(pa+i*N+j));
        printf("\n");
    }
}
}

```

Le matrici come parametri dei sottoprogrammi

In C una matrice è memorizzata per righe e gli indici partono da 0. Il passaggio dei parametri normalmente avviene per valore, ma quando il parametro è un array allora il passaggio è per riferimento perché viene passato l'indirizzo base dell'array.

LDA (Leading Dimension Array)

È pratica comune in programmazione, quando si adoperano matrici, sovradimensionarle, in modo tale da poterne utilizzare diverse "porzioni" in base alle necessità.

1	2	3	0
4	5	6	0
0	0	0	0
0	0	0	0

Si supponga ora di avere una matrice come quella in figura: quadrata, di 4 righe per 4 colonne e di cui siamo interessati solo alla porzione composta dalle prime 3 colonne e 2 righe. Ipotizzando di passare ad una function che stampa il contenuto della sottomatrice interessata le sue dimensioni, in modo da produrre la visualizzazione sperata, produrremmo un output inevitabilmente errato. Ciò è dovuto al fatto che, come noto, il C adoperava una particolare "mappa di memorizzazione" per individuare gli elementi di un array multidimensionale in memoria.

Nel caso di matrici a 2 dimensioni $A[i][j]$ equivale a $*(\&A[0][0] + N_COL * i + j)$ dove:

- " $\&A[0][0]$ " è l'indirizzo base della matrice (pa)
- " N_COL " è il numero di colonne effettivamente allocate della matrice
- " i " è l'indice di riga
- " j " è l'indice di colonna

Se si vuole accedere all'elemento di posto [1,0]: $*(\&A[0][0] + 3 * 1 + 0)$. Ossia all'indirizzo base della matrice occorrerebbe sommare il valore 3 per raggiungere, in memoria, l'elemento $A[1][0]$.

È un errore passare al parametro "N_COL" (ossia il numero di colonne della sottomatrice) il valore 3; in realtà la dimensione da passare è quella originale della matrice completa. Quando si deve svolgere una funzione del genere, bisogna tenere in conto il totale delle colonne originale della matrice: cioè, il **Leading Dimension Array**.

Versione del programma con array statico

```
#include <stdio.h>
#include <stdlib.h>
void matrice(short, short, short, long *);
void main() {
    long A[4][4]={1, 2, 3, 0, 4, 5, 6}, (*pA)[4][4];
    short i, j, m=2, n=3, LDA=4;
    pA = &A;
    for (i=0; i<m; i++)
        for (j=0; j<n; j++)
            printf("( *pA)[%d][%d]=%d,\t", i, j, (*pA)[i][j]);
    matrice(LDA, m, n, pA);
}
void matrice(short lda, short mm, short nn, long *pM) {
    short h, k;
    for (h=0; h < mm; h++)
        for (k=0; k<nn; k++)
            printf("( *pM+%d*lda+%d)=%d,\t", h, k, *(pM+h*lda+k));
}
```

Dichiarando l'array statico pA con i puntatori risolviamo la limitazione della function a semplici array bidimensionali.

Versione del programma con array dinamico

```
#include <stdio.h>
#include <stdlib.h>
void matrice(short, short, long *);
void main() {
    long *pA;
    short i, j, m=2, n=3;
    pA = malloc(m*n*sizeof(long)); //oppure calloc(...)
    for (i=0; i<m; i++)
        for (j=0; j<n; j++) {
            *(pA+i*n+j) = i+j+1;
```



```
        printf("(pA)[%d][%d]=%d,\t", i, j, *(pA+i*n+j));
    }
    matrice(m, n, pA);
}
void matrice(short mm, short nn, long *pM) {
    short h, k;
    for (h=0; h < mm; h++)
        for (k=0; k<nn; k++)
            printf("(pM+%d*nn+%d)=%d,\t", h, k, *(pM+h*nn+k));
}
```

Capitolo 7 – Gestione dei file sequenziali in C

File in C

In C esistono due tipi di file:

- **file binario**, che deve essere dichiarato esplicitamente
- **file testo** [default], è un insieme di righe di caratteri dove ogni riga contiene massimo 255 caratteri e deve terminare con un carattere speciale (ad es. “\n”)

Attenzione al nome (completo di path) di un file C! Nel programma C è segnato come: `char *nomefile="c:\\dati\\elenco.txt"`; dalla tastiera si scrive: `"c:\\dati\\elenco.txt"`.

Nel linguaggio C, la dichiarazione di un file avviene con l’apertura. L’apertura viene eseguita con la funzione `fopen()`. [`FILE *fp = (file pointer)`]. Esso è un puntatore ad una struttura contenente informazioni sul file e serve per indirizzare tutte le operazioni sul file. [`fp=fopen(nome_file, modo)`]. Dove `nome_file` è un `char*`, mentre i modi possono essere: “r” read, “w” write, “a” append, “r+” read & write, “w+” read & write, “a+” read & append.

Una volta aperto un file sarà possibile scambiare informazioni tra il file e il programma. Quando il file non deve essere più utilizzato va chiuso mediante la funzione `fclose()`.

Apertura e chiusura di un file

In C, le operazioni di apertura e chiusura di un file sono realizzate mediante delle funzioni contenute nella libreria `stdio.h`. L’apertura viene realizzata mediante la funzione `fopen()`: `FILE *fopen (char *nomefile, char *modalita);`

Dove:

- “nomefile” è una stringa di caratteri indicante il nome del file da aprire
- “modalità” indica il modo in cui il file deve essere aperto

Se si verifica un errore in apertura del file, la `fopen()` restituisce un puntatore nullo. Per tale motivo è consigliato, prima di accedere ad un file, assicurarsi che la chiamata a `fopen()` sia stata eseguita con successo.

Esempio

```
/*file di caratteri: copia i caratteri del testo nel file specificato in ingresso */
#include <stdio.h>
#include <string.h>
void main() {
    char *testo = "Nel mezzo del cammin di nostra vita\n";
    int i; char *cp, nomefile[13];
    FILE *fp; //dichiara fp puntatore a file
    printf("nome del file (max 8 chars)=");
    scanf("%s", &nomefile);
    strcat(nomefile, ".txt");
    printf("nomefile=%s\n", nomefile);
    fp = fopen(nomefile, "w"); //apre il file associando fp al file (in scrittura)
```

```

cp = testo;
while (*cp != '\0') {
    putc(*cp, fp); //scrive un carattere sul file puntato da fp
    cp++;
}
fclose(fp); //chiude il file
}

```

File binari in C e funzioni di I/O

Un file binario è un file che contiene delle informazioni così come appaiono in memoria (senza conversione da formato interno a esterno). In pratica esso è del tutto analogo ad un file di testo, privato però della traduzione in caratteri del relativo contenuto in memoria. Ciò significa che nei file binari sono memorizzati i veri bit che compongono l'informazione registrata. Le operazioni di I/O su un file binario non prevedono alcuna operazione di codifica. L'apertura di un file binario in C è realizzata al solito invocando la funzione `fopen()`. Come secondo argomento va specificata l'opzione "b" per indicare che si tratta di un file binario. Ad esempio, alla "w" per l'apertura in scrittura di un file di testo si aggiunge una "b" per ottenere "wb", cioè l'apertura in scrittura di un file questa volta binario.

```
file *fp; char nome_file[20]; fp=fopen(nome_file, "rb") //apre in lettura (r) il file binario (b)
```

I/O sui file binari

Le operazioni di lettura e scrittura di un file binario sono solitamente realizzate tramite le funzioni `fread` e `fwrite`. Tali operazioni avvengono su interi blocchi di dati in formato interno.

- Input: `type_i fread(void *buffer, type_i size, type_i count, FILE *fp)`
- Output: `type_i fwrite(void *buffer, type_i size, type_i count, FILE *fp)`

La variabile **buffer** è un puntatore all'area di memoria interessata dal trasferimento di informazioni. Stessa cosa vale per **fp** che è il puntatore al file interessato dalle operazioni di I/O. Le variabili **size** e **count** indicano rispettivamente il numero massimo di voci trasferite e il numero di byte di ciascuna voce. Una chiamata ad una delle due funzioni con gli argomenti visti sopra produce il seguente effetto: sono trasferite fino a **count** voci, ciascun di **size** bytes, tra l'unità specificata da **fp** e l'area di memoria puntata da **buffer**. Il puntatore sul file avanza del numero di byte relativo.

Funzioni di I/O per file di testo

Il C prevede diverse funzioni per la gestione dell'I/O associato a file di testo. Queste funzioni sono divise in:

- **I/O formattato**, che consentono la lettura o la stampa secondo i codici di formato previsti
- **I/O non formattato**, che prelevano o restituiscono in output semplicemente il contenuto dei file così come registrato su di esso

Input		Output	
non formattato	formattato	non formattato	formattato
getc(...)		putc(...)	
gets(...)	fscanf(...)	puts(...)	fprintf(...)
fgets(...)		fputs(...)	

fscanf(FILE *, "formato", variabili)
fprintf(FILE *, "formato", variabili)

I/O singolo carattere (come int)

```
int getc(FILE *)
int getchar(void)
int putc(intero, FILE *)
int putchar(intero)
```

intero: qualsiasi tipo intero, ma solo gli 8 bit meno significativi saranno trasferiti

Unità standard di I/O
(stdin, stdout)

numero caratteri
da leggere

I/O stringa di caratteri

```
char *gets(char *)
char *fgets(char *, int, FILE *)
int puts(char *)
int fputs(char *, FILE *)
```

La **getc()** e la **putc()** sono analoghe alla **getchar()** e alla **putchar()** con l'aggiunta di un parametro che individua il file oggetto del trasferimento.

La **fputs()** e la **fgets()** sono analoghe alla **puts()** e alla **gets()** con l'aggiunta del riferimento al puntatore del file; inoltre la **fgets()** prevede un parametro intero che stabilisce il numero di caratteri da leggere.

Esempio

```
/*legge i caratteri dal file specificato in ingresso e li visualizza sullo schermo */
#include <stdio.h>
#include <string.h>
void main() {
    char ch, *cp, nomefile[13];
    FILE *fp;
    printf("nome del file (max 8 chars)=");
    scanf("%s", &nomefile);
    strcat(nomefile, ".txt");
    fp = fopen(nomefile, "r"); //apre il file in lettura
    while (ch != EOF) {
        ch = getc(fp);
        putchar(ch);
    }
    fclose(fp);
}
```

Fine di un file di testo e binario

La fine di un file di testo può essere individuata confrontando il carattere letto con EOF oppure tramite la function C [**int** feof(**FILE** *)].

Essa restituisce 0 (falso) se non è stata raggiunta la fine del file; un valore diverso da 0 (vero) altrimenti.

La fine di un file binario può essere individuata solo tramite la function C. [**int** feof(**FILE** *)]

Capitolo 8 – Strutture dati dinamiche lineari

Richiami sulle strutture dati statiche

Tipo di dato strutturato

Un dato strutturato è un insieme di informazioni logicamente collegate da un unico nome. Quando si dichiara un tipo di dato strutturato, molteplici sono le caratteristiche:

- Se il numero di componenti è fisso (statico) oppure è variabile (dinamico)
- Il tipo delle componenti
- Le modalità di accesso alle componenti
- L'eventuale possibilità di inserimento / eliminazione delle componenti
- L'eventuale ordinamento (quindi il collegamento logico) delle componenti

Tipo di dato primitivo e tipo di dato derivato

I linguaggi di programmazione mettono a disposizione alcuni tipi di dato, detti **tipi di dato primitivi**. I tipi di dato definitivi dall'utente prendono il nome di **tipi di dato derivati**. I tipi di dato primitivi in un linguaggio di programmazione sono "gestiti" nel linguaggio stesso, cioè sono possibili operazioni quali:

- Accesso all'informazione
- Modifica
- Inserimento / eliminazione

Nei tipi di dato derivati, il programmatore deve esprimere il nuovo tipo di dato tramite quelli primitivi e realizzare le procedure per le operazioni sugli oggetti del nuovo tipo.

Richiami sul tipo array

Il tipo di **dato array** stabilisce che:

- Il numero delle componenti è fisso (tipo di dato statico)
- Le componenti devono essere tutte dello **stesso tipo**
- L'accesso alle componenti è **diretto** tramite indici (legati alla posizione delle componenti nella struttura)

Richiami sul tipo record

Un altro tipo di dato comune a molti linguaggi di programmazione è il **tipo strutturato record**. Il tipo record nasce come "estensione" del tipo array, in quanto le sue componenti possono essere anche di tipo diverso. Infatti, la dichiarazione di un tipo record stabilisce che:

- Il numero di componenti è fisso (statico)
- Le componenti (chiamate campi) possono anche essere di tipo diverso
- L'accesso (diretto) alle componenti viene effettuato tramite il nome del campo

La struttura dati che descrive il tipo record in C è la **struct**.

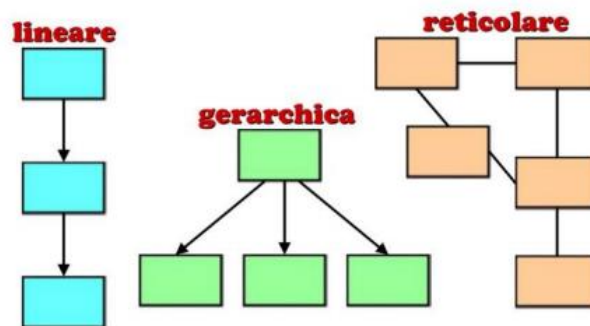
Generalità sulle strutture dati dinamiche

Classificazione delle strutture dati dinamiche

Le **strutture dinamiche** non hanno un numero fisso di componenti come le strutture statiche, ma durante l'elaborazione il numero di componenti può variare. Le strutture dati dinamiche sono basate sull'uso di dati di tipo **puntatore** e sull'**allocazione dinamica della memoria**. Gli elementi possono essere allocati (e deallocati) man mano che servono, collegati tra loro in diversi modi, e questi collegamenti possono a loro volta mutare durante l'esecuzione del programma. Lo spazio di memoria necessario per allocare i puntatori, e le operazioni necessarie alla loro manutenzione costituiscono il costo aggiuntivo delle strutture dati dinamiche.

In assenza dei puntatori, è anche possibile costruire strutture dati dinamiche utilizzando gli array, rinunciando però alla flessibilità nell'uso della memoria: viene allocato un array di dimensioni sufficienti a contenere tutti gli elementi che si pensa di dover gestire, e al posto dei puntatori si usano indici nell'array. Le strutture dinamiche sono classificate in:

- **Lineari**, in cui l'accesso alle informazioni è fatto secondo l'ordine degli elementi
- **Gerarchiche**, in cui l'accesso alle informazioni è sempre dall'alto verso il basso
- **Reticolari**, se l'accesso alle informazioni dipende dai collegamenti esistenti



Strutture dinamiche lineari

Nelle strutture dinamiche lineari le informazioni sono collegate da link secondo un preciso ordine. L'elemento **corrente** è sempre fiancheggiato a sinistra dall'elemento predecessore e a destra da quello **successore**. Il link rende esplicito l'**ordinamento** (logico) delle informazioni, svincolandole dalla posizione relativa in memoria (ordinamento fisico).

Inoltre, le strutture dinamiche lineari sono scindibili in: strutture lineari **aperte** e **chiuse**. Le **strutture dinamiche lineari aperte** sono assimilabili, per ordine, ai semplici array, in quanto gli elementi si susseguono linearmente uno dopo l'altro da un inizio ad una fine. Nelle **strutture dinamiche lineari chiuse** invece, non esiste una vera e propria fine, in quanto l'elemento più esterno si ricollega a quello più interno tramite un nuovo link costituendo in un certo senso, una struttura dati **circolare**.

Operazioni sulle strutture dinamiche lineari

Eliminazione elemento "centrale"

Supponendo di voler eliminare un elemento "centrale" l'unica operazione da compiere è quella di modificare il link del suo predecessore in modo tale che questo punti direttamente al successore.

Eliminazione elemento in “testa”

Per eliminare l’elemento in testa della struttura lineare occorre semplicemente modificare il puntatore relativo facendolo in un certo senso “avanzare” perché punti all’elemento successivo.

Eliminazione elemento in “coda”

Per eliminare l’elemento in coda della struttura occorre sostanzialmente eliminare l’ultimo link, ossia quello che realizza il collegamento tra il penultimo elemento e quello da cancellare.

Inserimento elemento “centrale”

Per inserire un elemento dopo quello corrente bisogna:

- Inserire il link che va dall’elemento nuovo al successore
- Modificare il link dell’elemento centrale affinché punti al nuovo elemento

Inserimento elemento in “testa”

Per inserire un elemento in testa alla struttura bisogna:

- Inserire il link che va dall’elemento nuovo alla testa della struttura
- Modificare il link alla testa affinché punti al nuovo elemento

Inserimento elemento in “coda”

Infine, per inserire un elemento in coda ad una struttura dinamica lineare, occorre semplicemente aggiungere il link che connette l’ex ultimo dato (il dato finale precedente) con il nuovo.

struttura lineare	array	lista	coda	pila
dinamica	NO	SI	SI	SI
ordinamento	fisico	SI (per accesso)	SI (per arrivo)	
accesso	diretto	sequenziale	ai 2 estremi	ad 1 estremo
inserimento	-	ovunque	alla fine	solo ad
eliminazione	-		all'inizio	1 estremo
First In First Out			F.I.F.O.	
Last In First Out			L.I.F.O.	

Principali strutture dinamiche lineari: pila e coda**Struttura LIFO: la pila (stack)**

La pila è una **struttura lineare aperta** in cui l’accesso alle componenti per l’inserimento e l’eliminazione avvengono solo ad un estremo della struttura (detta testa o head della pila). Le operazioni di inserimento e di estrazione sono solitamente indicate col termine di **push** e **pop**. La

pila è una struttura **L.I.F.O. (Last In First Out)** perché l'ultimo elemento inserito è il primo ad essere eliminato. Solitamente la pila è implementata facendo uso di liste o array.

Si può realizzare una pila in due modi: **statico** se si usa un array e si stabilisce una dimensione massima di riempimento dello stack; **dinamico** se si usa una lista lineare.

Esempio struttura pila con array statico

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_STACK_SIZE 100
void invert_array(char [], short);
void push_s(char, char[], short *);
void pop_s(char *, char [], short *);
void main() {
    char a[]={ 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'L' };
    short len_a = 10, i;
    puts("array prima");
    for (i=0; i < len_a; i++)
        printf("a[%d]=%c\n", i, a[i]);
    invert_array(a, len_a);
    puts("array dopo");
    for (i=0; i<len_a; i++)
        printf("a[%d]=%c\n", i, a[i]);
}
void invert_array(char a[], short len_a) {
    char temp[MAX_STACK_SIZE];
    head = -1; //indica stack vuoto
    for (i=0; i<len_a; i++) push_s(a[i], temp, &head);
    for (i=0; i<len_a; i++) pop_s(a+i, temp, &head);
}
void push_s (char elem, char p_stack[], short *head) {
    *(p_stack+ (++*head))=elem; //si deve controllare il riempimento dello stack
}
void pop_s(char *elem, char p_stack[], short *head) {
    *elem=*(p_stack+(*head)--); //attenzione allo svuotamento dello stack
}
```

Esempio struttura pila con array dinamico

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_STACK_SIZE 100
```



```

void invert_array(char [], short);
void push_s(char, char[], short *);
void pop_s(char *, char [], short *);
void main() {
    char a[]={ 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'L' };
    short len_a = 10, i;
    puts("array prima");
    for (i=0; i < len_a; i++)
        printf("a[%d]=%c\n", i, a[i]);
    invert_array(a, len_a);
    puts("array dopo");
    for (i=0; i<len_a; i++)
        printf("a[%d]=%c\n", i, a[i]);
}
void invert_array(char a[], short len_a) {
    char *p_temp; short i, head;
    head = -1; //indica stack vuoto
    p_temp = calloc(len_a, sizeof a[1]);
    for (i=0; i<len_a; i++) push_s(a[i], temp, &head);
    for (i=0; i<len_a; i++) pop_s(a+i, temp, &head);
    free(p_temp);
}
void push_s (char elem, char p_stack[], short *head) {
    *(p_stack+ (++*head))=elem; //si deve controllare il riempimento dello stack
}
void pop_s(char *elem, char p_stack[], short *head) {
    *elem=*(p_stack+(*head)--); //attenzione allo svuotamento dello stack
}

```

Struttura FIFO: la coda (queue)

La coda è una struttura lineare aperta in cui l'accesso alle componenti avviene solo ai due estremi:

- L'eliminazione (dequeue) avviene solo all'inizio della struttura
- L'inserimento (enqueue) avviene solo alla fine

La coda è una struttura **F.I.F.O. (First In First Out)** perché il primo elemento inserito è anche il primo ad essere eliminato. Solitamente la coda è realizzata mediante uso di array o di liste. La coda tende a slittare verso la fine dell'array.

Codice funzioni

```

void Enqueue(char Vet_Coda[], char Elemento, short *Fondo) {
    if (*Fondo<LEN_QUEUE) Vet_Coda[(*Fondo)++]=Elemento;
    else printf("\n*----- NON INSERISCO NULLA, FONDO FINITO! -----*\n");
}

```

```
void Dequeue(short *Testa) {  
    if (*Testa < LEN_QUEUE) (*Testa)++;  
}
```

Evitare lo slittamento dell'array: compattamento

Una prima semplice soluzione al problema dello slittamento di una coda verso la fine dell'array statico usato per simularla è quello di, per ogni eliminazione, riportare le componenti verso il top della coda, **shiftandole**. Questa soluzione è però da escludersi, in quanto, lo spostamento degli elementi di un array, soprattutto per size elevati, comporta un notevole dispendio di tempo.

Evitare lo slittamento dell'array: array "circolare"

Una soluzione da preferirsi rispetto a quella precedente è rappresentata dall'uso di un array "circolare". Per **array "circolare"** si intende un normale array in cui, la gestione delle componenti durante le operazioni di inserimento ed eliminazione dalla coda, è "assistita" dall'aritmetica modulo N.

Principali strutture dinamiche lineari: lista

Struttura sequenziale: la lista lineare (linked list)

Una lista è un **insieme di nodi** collegati linearmente. I nodi sono dei record che contengono almeno due campi:

- Uno per le **informazioni** (che può essere diviso in sottocampi)
- Uno per il **puntatore all'elemento successivo** della lista

La lista, a differenza della pila e della coda, è capace di mantenere un ordinamento logico fra le informazioni. All'inizio, è necessario avere un puntatore esterno che punta al primo nodo della struttura tramite il suo campo puntatore; dal primo nodo si passa al secondo; e così via, fino all'ultimo nodo il cui campo puntatore deve avere un valore convenzionale (es: indirizzo NULL) che indica che non ci sono più nodi dopo. Conoscendo un nodo interno alla lista, è possibile accedere ai nodi successivi, ma non a quelli precedenti. Ciò fa della lista, una struttura dati **ordinata**.

N.B. Le informazioni sono ordinate in senso logico ma non in senso fisico in quanto i nodi possono essere allocati in memoria in una posizione qualsiasi.

Inoltre, la lista è una struttura lineare **aperta** o **chiusa** in cui l'inserimento e l'eliminazione delle componenti possono avvenire in una qualunque posizione. Per implementare una lista in C, viene fatto solitamente uso di struct di almeno due campi: un campo informazione ed un campo link.

Operazioni su lista lineare: visita

La possibilità di esplorare il contenuto dei singoli nodi è facilmente realizzabile in una lista lineare. Visitare una lista significa accedere ordinatamente a tutti i nodi della struttura. Per realizzare questa

operazione occorre una variabile temporanea puntatore che punterà a tutti i nodi della struttura ordinatamente uno alla volta (ad esempio: la variabile *pt*).

Per iniziare la visita, la variabile puntatore deve avere lo stesso valore di testa per puntare al primo nodo della struttura. Per passare al nodo successivo dovrà assumere il valore contenuto nel campo puntatore di tale nodo. Passaggio dal nodo corrente al successivo (**avanzamento**): ad ogni iterazione, la variabile puntatore assume l'indirizzo contenuto nel campo *pt* del nodo in modo tale da puntare al nodo successivo.

```
void visualizza(struct PERSONA *head) {
    /* Visita della lista */
    do {
        printf(" Nome: %2s % \n", head->info.nome);
        head=head->p_next;
    } while(head!=NULL);
}
```

Operazioni su lista lineare: eliminazione

Eliminazione elemento corrente

Per eliminare l'elemento corrente occorre modificare il link del suo predecessore affinché punti direttamente al suo successore.

```
void elim_nodo(struct PERSONA *punt) {
    struct PERSONA *Libera; Libera=(punt)->p_next; //Libero dalla memoria successivamente
    quel nodo
    punt->p_next = (punt->p_next)->p_next;
    free(Libera); //Libera dalla memoria quel nodo eliminato
}
```

Eliminazione elemento in testa

Per eliminare l'elemento in testa occorre modificare il link che punta al primo record della lista.

```
void elim_testa(struct PERSONA **head) {
    struct PERSONA *Libera; Libera=*head; //Libero dalla memoria successivamente quel nodo
    *head = (*head)->p_next;
    free(Libera); //Libera dalla memoria quel nodo eliminato
}
```

Eliminazione elemento in coda

Per eliminare l'elemento in coda occorre distruggere il link che punta alla fine della lista.

Operazioni su lista lineare: inserimento

Inserimento di un elemento dopo quello corrente

- 1) **Inserire il link** che va dall'elemento nuovo al successore
- 2) **Modificare il link** dell'elemento corrente affinché punti al nuovo elemento

```
void insl_nodo(INFO_FIELD Dati, struct PERSONA **punt) {
    struct PERSONA *ptr; //Puntatore al NUOVO NODO
    /* Crea nodo e inserisci dati */
    ptr=(struct PERSONA *)calloc(1,sizeof (struct PERSONA)); //Richiediamo di allocare un nodo
    di grandezza persona
    ptr->info=Dati; //Inserisci dati; (*ptr).info
    /* Aggancia il new nodo al successivo di punt */
    ptr->p_next= (*punt)->p_next;
    /* Aggancia il nodo considerato a punt al nuovo nodo */
    (*punt)->p_next = ptr;
    /* punt ora lo faremo puntare al nuovo nodo, perché' sarà il corrente (facoltativo) */
    *punt=ptr;
}
```

Inserimento di un elemento in testa

- 1) **Inserire il link** che va dall'elemento nuovo alla testa della struttura
- 2) **Modificare il link** alla testa affinché punti al nuovo elemento

```
void insl_testa (INFO_FIELD Dati, struct PERSONA **head) {
    struct PERSONA *ptr; //Puntatore al NUOVO NODO
    /* Crea nodo e inserisci dati */
    ptr=(struct PERSONA *)calloc(1,sizeof (struct PERSONA)); //Richiediamo di allocare un nodo
    di grandezza persona
    ptr->info=Dati; //Inserisci dati; (*ptr).info
    /* Aggancia il new nodo al nodo in testa (me lo dice head dove sta)*/
    ptr->p_next = *head; //Head contiene l'indirizzo del nodo in testa
    /* Aggancia testa al new nodo */
    *head=ptr;
}
```

Inserimento di un elemento in coda

Per inserire un elemento in coda, bisogna **aggiungere il link** che punta alla fine della struttura al nuovo elemento.

Implementazione C di una lista lineare: fondamenti

ADT (Abstract Data Type)

Per l'implementazione delle principali strutture dinamiche, quali la lista, la pila e la coda, il linguaggio C mette a disposizione le struct, gli array e i puntatori.

- La **lista lineare** utilizza una struct per i singoli elementi della struttura (nodi) e i puntatori per la gestione dei link tra le informazioni.
- La **pila** può essere realizzata mediante array o mediante una lista lineare.
- La **coda** può essere realizzata mediante array, ma si presta meglio ad essere realizzata mediante una lista.

La lista è un tipo di dato derivato che si basa sul meccanismo della **struttura autoriferente** che è una struttura che presenta al suo interno un **puntatore autoriferente**, cioè un puntatore allo stesso tipo della struttura. In pratica, il tipo struct permette di introdurre il tipo derivato **lista lineare** definendo un campo della struct come puntatore allo stesso tipo di struct.

```
struct PERSONA {  
    char nome[20];  
    struct PERSONA *p_next;  
};
```

Struttura autoriferente statica

Il **tipo astratto record** (corrispondente in C al tipo struct) consente di introdurre come tipo derivato il **tipo astratto lista lineare** se si definisce un campo del record come **puntatore allo stesso tipo di record**. (NON SERVE A NULLA!!!)

```
struct PERSONA {  
    char nome[20];  
    struct PERSONA *p_next;  
} el_1, el_2, el_3;
```

Struttura autoriferente dinamica

In questo caso è prevista la dichiarazione di due puntatori a struttura, ad esempio sono ***head** e ***punt**. Il primo punta alla testa della lista, il secondo punta al nodo corrente. I collegamenti tra i nodi sono realizzati allocando dinamicamente spazio per la struct corrente (puntata da punt).

```
struct PERSONA {  
    char nome[20]; short eta;  
    struct PERSONA *p_next;  
} *head, *punt;
```

Gli operatori di struttura “.” e “->”, le parentesi tonde (e) per le chiamate a funzioni e le parentesi quadre [e] per gli indici di array hanno proprietà massima sugli altri operatori.

Esempio

```
#include <stdio.h>  
#include <string.h>  
void main() {  
    struct PERSONA {
```

```

    char nome[20];
    short eta;
    struct PERSONA *p_next;
} *head, *punt;
head = calloc(1, sizeof(struct PERSONA));
strcpy(head->nome, "Bianchi Roberto");
head->eta=22;
head->p_next = calloc(1, sizeof(struct PERSONA));
strcpy((head->p_next)->nome, "Rossi Maurizio");
(head->p_next)->eta=25;
(head->p_next)->p_next=calloc(1, sizeof(struct PERSONA));
strcpy(((head->p_next)->p_next)->p_next->nome, "Verdi Gianluca");
((head->p_next)->p_next)->eta=18;
((head->p_next)->p_next)->p_next=NULL;
punt = head;
while (punt->p_next != NULL) {
    printf("nome=%s, \tp_next=%d\n", punt->nome, punt->p_next);
    punt = punt->p_next;
}
printf("nome=%s, \tp_next=%d\n", punt->nome, punt->p_next);
}

```

Particolari organizzazioni dei dati per una lista lineare

Nodo sentinella

Per evitare di avere due function per l'inserimento e due function per l'eliminazione è possibile descrivere un'unica sequenza di operazioni sia per gli inserimenti/eliminazioni effettuati sulla testa che per quelli fatti sul nodo corrente. Occorre semplicemente aggiungere un nodo fittizio che funga da testa ma che punti alla testa reale della lista: un **nodo sentinella**. Ogni modifica sulla testa comporterà in questo modo la sola alterazione del campo puntatore del nodo sentinella.

Lista lineare generica

Il secondo approfondimento sulle liste lineari è la costruzione di funzioni di manipolazione a carattere generale. In pratica, è possibile scrivere funzioni di manipolazione che non facciano riferimento ad una particolare struttura dati (es: `struct PERSONA`) ma che possano essere utilizzate per qualsiasi problema. Per realizzare questa soluzione è necessario eseguire **operazioni di casting** e far uso di **puntatori a void**.

Esempio

```

typedef struct {
    char nome[20];
    short eta;
} INFO_FIELD;

```

```

void main () {
    struct PERSONA {
        INFO_FIELD info;
        struct PERSONA *p_next;
    };
    struct PERSONA *head, *punt;
    char *name[] = {"Bianchi Roberto", ...};
    short age[] = {22, 25, 18, ...};
}

```

Funzione per la creazione di una lista lineare generica:

//chiamata nel main(): head = (struct PERSONA *) creaLista();

```

void *creaLista(){
    char *testa;
    testa = NULL;
    return testa;
}

```

Funzione di inserimento in testa in una lista generica:

//inserisce dato in testa alla lista

//chiamata: insL_testa(len_info, p_nuovodato, &head);

```

void insL_testa(short len_info, INFO_FIELD *p_dato, void **p_head) {
    struct lista {
        INFO_FIELD info;
        struct lista *p_next;
    } *ptr;
    ptr = calloc(1, sizeof(struct lista));
    memcpy(ptr->info, p_dato, len_info); memcpy(&(ptr->info), p_dato, len_info)
    ptr->p_next=(struct lista *)*p_head;
    (struct lista *)*p_head=ptr;
}

```

Funzione di inserimento in mezzo in una lista generica

//inserisce dopo nodo corrente

//chiamata: insL_nodo(len_info, p_nuovodato, &punt);

```

void insL_nodo(short len_info, INFO_FIELD *p_dato, void **p_punt) {
    struct lista {
        INFO_FIELD info;
        struct lista *p_next;
    } *ptr;
    ptr = calloc(1, sizeof(struct lista));
    memcpy(ptr, p_dato, len_info);
    ptr->p_next=((struct lista *)*p_punt)->p_next;
    (struct lista *)*p_punt=ptr;
}

```

Funzione di eliminazione in testa in una lista generica

```
//elimina nodo in testa alla lista
//chiamata: eliL_testa(&head);
void eliL_testa(void **p_head) {
    struct lista {
        INFO_FIELD info;
        struct lista *p_next;
    } *ptr;
    ptr=((struct lista *)*p_head)->p_next;
    free((struct lista *)*p_head);
    *p_head=ptr;
}
```

Funzione di eliminazione in mezzo in una lista generica

```
//elimina nodo successore
//chiamata: eliL_nodo(&prec); (dove prec punta al nodo che precede quello da eliminare)
void eliL_nodo(void **p_punt) {
    struct lista {
        INFO_FIELD info;
        struct lista *p_next;
    } *ptr;
    ptr=((struct lista *)*p_punt)->p_next;
    ((struct lista *)*p_punt)->p_next = ptr->p_next;
    free(ptr);
}
```

Applicazione delle liste ad altre strutture dati**Pila e coda mediante la struttura dati lista lineare**

Le strutture **pila** e **coda** sono agevolmente descrivibili mediante liste lineari, tenendo presente le restrizioni tipiche di queste modalità di organizzazione dei dati relativamente a inserimento ed eliminazione. Ricordiamo che per la pila l'inserimento e l'eliminazione avvengono solo ad un estremo. Per questo motivo, se noi usiamo una lista lineare per rappresentare una pila, l'estremo più comodo è la testa della lista. È necessario far uso di un solo puntatore, TOP, che punti alla testa della lista. Nel caso della coda, invece, sono necessari due puntatori: TOP, che punta alla testa della lista (dove viene eseguita l'eliminazione) e BOTTOM che punta all'ultimo nodo della lista (dove viene inserito l'inserimento).

Lista circolare e lista bidirezionale

Lista circolare (catena): nella lista circolare non esiste una "fine" in quanto l'ultimo nodo viene fatto puntare al primo. È assimilabile quindi ad un anello. Il puntatore esterno, testa, serve anche per stabilire quando sono stati visitati tutti i nodi (siccome non c'è una fine).

Lista bidirezionale (simmetrica): nella lista bidirezionale i nodi contengono due campi puntatori: uno che punta al nodo successivo e l'altro che punta a quello precedente. È come se avessimo due liste che condividono le stesse informazioni ma in ordine inverso. Sono necessari due puntatori testa che puntano agli estremi opposti della struttura. Questo tipo di struttura consente agevolmente di spostarsi in avanti o indietro.

Liste multiple: rappresentazione di matrici sparse

Una **matrice sparsa** è una matrice di grandi dimensioni che ha un'alta percentuale di zeri al suo interno. Il problema delle matrici sparse è legato alla memorizzazione: a causa dell'alta percentuale di zeri presenti non conviene memorizzarle come array2D. Infatti, la memorizzazione degli zeri richiederebbe tantissimi bytes. Per la memorizzazione delle matrici sparse risulta proficuo utilizzare algoritmi adatti allo scopo e strutture dati che tengono conto della natura sparsa della matrice.

I dati sparsi sono, per loro natura, facilmente comprimibili, e la loro compressione comporta quasi sempre un utilizzo significativamente inferiore di memoria. Una delle strutture più adatte alla memorizzazione delle matrici sparse è la **lista multipla**, o **multilista**: si tratta di una lista concatenata costituita da nodi che hanno due o più link ad altri nodi.

Nelle liste multiple vengono utilizzati tre tipi di nodi:

- Il primo tipo, cioè il nodo di accesso primario alla struttura
- Il secondo tipo
- Il terzo tipo, che contiene gli elementi veri e propri della matrice

Per accedere per righe ad un elemento:

- 1) Si avanza sulla lista dei puntatori alle righe fino a trovare quella cui appartiene l'elemento cercato
- 2) Si scorre la lista della riga cercando l'indice di colonna dell'elemento voluto
- 3) Se l'indice di colonna è presente allora si è trovato l'elemento cercato, altrimenti l'elemento cercato è zero.

Capitolo 9 – Strutture dati dinamiche gerarchiche

Generalità sulla struttura dati albero

L'albero è una struttura dati gerarchica. Una struttura gerarchica permette di accedere alle informazioni solo partendo dal nodo che si trova più in alto per poi scendere nei livelli inferiori. Un albero si compone di due tipi di sottostrutture fondamentali:

- il **nodo**, che in genere contiene informazioni
- l'**arco**, che stabilisce un collegamento gerarchico tra due nodi

Definizioni

Un nodo può avere un solo arco entrante, ma può avere più archi uscenti.

Il **nodo padre** è un nodo dal quale esce almeno un arco che lo collega ad un altro nodo (**nodo figlio**). Ovviamente, un nodo può essere padre e figlio contemporaneamente.

La **radice** è quell'unico nodo privo di arco entrante (padre).

La **foglia** è un nodo che non presenta archi uscenti (figli). In ogni albero finito, cioè con un numero finito di nodi, si trova almeno un nodo foglia.

Grado e sottoalbero di un nodo

Solitamente si usa anche associare un grado per ciascun nodo di un albero in base al numero di nodi figli di cui esso è padre. Naturalmente i nodi foglie hanno sempre un grado pari a zero.

È interessante notare come un singolo albero definisca una moltitudine di molteplici **sottoalberi**. Questo aspetto garantisce una facile descrizione ricorsiva per gli algoritmi che lavorano su strutture dati di questo tipo.

Alberi binari

Un **albero binario** è una struttura dati formata da nodi collegati tra loro da archi. La caratteristica fondamentale dell'albero binario è che si possono avere **al più due figli per ogni nodo**: ciascuno dei due figli individua un sottoalbero, per cui si avranno sottoalbero sinistro e sottoalbero destro.

N.B. Nell'albero binario ciascuno dei due figli può mancare.

Gli alberi binari possono classificarsi in:

- **alberi binari completi**; sono quegli alberi in cui ogni nodo, escluse le foglie, ha esattamente due figli
- **alberi binari quasi completi**; sono quegli alberi in cui ogni nodo dispone di due figli eccetto che per i nodi dell'ultimo livello; in quest'ultimo livello le foglie devono trovarsi sulla sinistra

Quando l'albero non è completo c'è la possibilità di completarlo totalmente o parzialmente aggiungendo dei nodi fittizi, detti **nodi esterni** che servono solo per l'applicazione in alcuni algoritmi.

Algoritmi di visita di un albero binario

Ne sono tre perché in un albero binario ci sono tre entità fondamentali: la **radice**, il **sottoalbero sinistro** e il **sottoalbero destro**. *N.B. Ciascuno dei tre algoritmi termina quando si tenta di effettuare un'operazione di pop a stack vuoto.*

Visita preorder

Nella visita in preordine, se l'albero non è vuoto:

- si analizza la radice dell'albero
- si visita in preordine il sottoalbero sinistro
- si visita in preordine il sottoalbero destro

Visita inorder

Nella visita in ordine, se l'albero non è vuoto:

- si visita in ordine il sottoalbero sinistro
- si analizza la radice dell'albero
- si visita in ordine il sottoalbero destro

Visita postorder

Nella visita in postordine, se l'albero non è vuoto:

- si visita in postordine il sottoalbero sinistro
- si visita in postordine il sottoalbero destro
- si analizza la radice dell'albero

[Clicca qui per vedere gli algoritmi ricorsivi per la visita di un albero binario](#)

Parsing

Il **parsing** o **analisi sintattica** è il processo atto ad analizzare uno stream continuo in input in modo da determinare la sua struttura grammaticale grazie ad una data grammatica formale.

Un **parser** è un programma che esegue questo compito. Di solito i parser non sono scritti a mano ma generati attraverso dei generatori di parser.

Una tipica operazione di parsing è condotta dal compilatore del linguaggio C per il riconoscimento della grammatica con cui, le operazioni da eseguire, sono descritte.



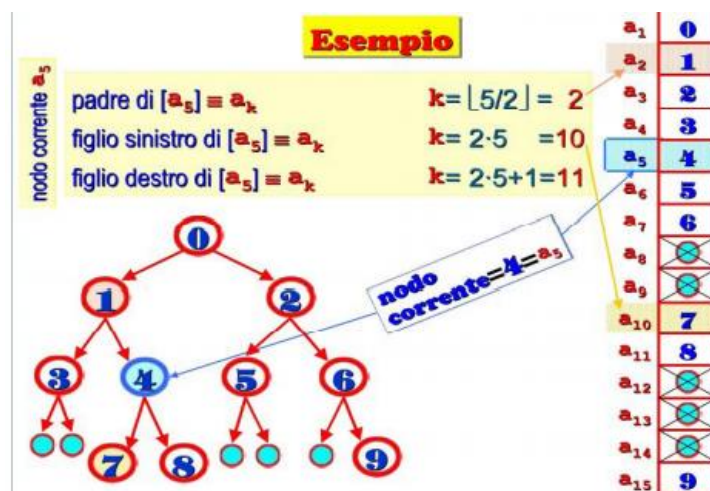
Strutture dati per la rappresentazione di un albero binario

Esso non è un tipo di dato primitivo, quindi dobbiamo sfruttare i dati primitivi che il C mette a disposizione. È possibile scegliere tra due strategie: rappresentare un albero binario con un semplice **array** monodimensionale oppure ricorrere all'uso delle **liste multiple**.

Rappresentazione di un albero binario mediante array

Un albero binario completo con $n = 2^P - 1$ nodi è rappresentato tramite un array $a = (a_1, a_2, \dots, a_n)$ e per un qualsiasi nodo a_i si ha:

- Se $i \neq 1 \rightarrow \text{padre}(a_i) = a_k$ dove $k = i/2$; se $i=1$ allora a_i è radice e non ha padre.
- Se $2i \leq n \rightarrow \text{figlio_sinistro}(a_i) = a_k$ dove $k = 2i$; se $2i > n$ allora a_i non ha figlio sinistro.
- Se $2i + 1 \leq n \rightarrow \text{figlio_destro}(a_i) = a_k$ dove $k = 2i + 1$; se $2i+1 > n$ allora a_i non ha figlio destro.

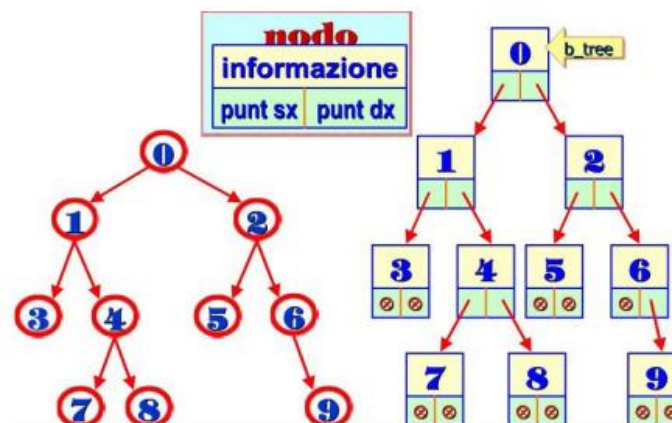


Rappresentazione di un albero binario mediante lista multipla

Un albero binario può essere memorizzato attraverso l'uso delle liste multiple. Le liste portano dei vantaggi nella memorizzazione, infatti:

- permettono, tramite la modifica dei puntatori, di evitare eventuali shifting
- non è necessario completare l'albero binario mediante l'inserimento di nodi fittizi
- il doppio puntatore facilita l'identificazione del figlio sinistro e del figlio destro

Sono più adatte a rappresentare gli alberi binari.



Algoritmi di visita (con stack esplicito) di un albero binario

A prescindere dal tipo di rappresentazione adoperata per l'albero binario, l'algoritmo di visita deve necessariamente far uso di uno stack, dove conservare **l'informazione sull'ordine dei nodi di cui non si è completata la visita dei sottoalberi**. Lo stack **va gestito esplicitamente** nell'algoritmo di visita iterativo mentre **va gestito implicitamente** nell'algoritmo di visita ricorsivo.

Esempio con stack implicito

```
preorder_ricors(struct btree *pt_nodo) {
    visita(pt_nodo->info);
    if foglia(pt_nodo)
        return;
    else {
        preorder_ricors(pt_nodo->pt_sx);
        preorder_ricors(pt_nodo->pt_dx); }
}
```

Alberi tramati

Gli **alberi tramati** sono quegli alberi in cui è possibile risalire da ogni nodo al suo nodo padre grazie agli **archi di ritorno**. Gli archi di ritorno consentono la "risalita" al nodo padre in modo semplice. La struttura di un nodo contiene in campi:

- **informazione**, che può essere diviso in sottocampi
- **grado del nodo**, cioè il numero di figli
- **puntatori ai figli**, è un array di puntatori, ognuno dei quali punta ad un figlio del nodo corrente
- **puntatore al padre**, un altro puntatore che punta al padre del nodo corrente

Esempio

```
struct {
    INFOFIELD info;
    short grado;
    struct NODO *pt_figli[2];
    struct NODO *pt_back;
} NODO;
struct NODO *p_Radice;
```

Alberi binari di ricerca (BST)

Un **albero binario di ricerca** in contesto informatico è un albero binario in cui i valori dei figli di un nodo sono ordinati, usualmente avendo valori minori di quelli del nodo di partenza nei figli a sinistra e valori più grandi nei figli a destra.

È un particolare albero in cui la visita **inorder** (in ordine simmetrico) consente di accedere alle informazioni in ordine crescente di chiave (key(nodo));

Proprietà di un BST (Binary Search Tree): se y è un qualsiasi nodo dell'albero si ha:

- \forall nodo x del sottoalbero sinistro di y $key(x) \leq key(y)$
- \forall nodo z del sottoalbero destro di y $key(y) \leq key(z)$

In un albero binario di ricerca con chiavi tutte diverse, si ha:

- La chiave di **valore minimo** occupa il nodo foglia del sottoalbero più a sinistra (cioè la prima foglia che si incontra con la visita inorder);
- La chiave di **valore massimo** occupa il nodo foglia del sottoalbero più a destra (cioè l'ultima foglia che si incontra con la visita inorder);

Costruzione di un albero binario di ricerca (BST o ABR)

La costruzione di un albero binario di ricerca viene effettuata in questo modo: si considera un flusso di dati, ad esempio un array; per ogni dato in arrivo, lo si confronta con la radice per poi farlo scendere nel sottoalbero opportuno fino ad assegnargli la posizione definitiva. Tale posizione viene determinata confrontando il valore della chiave (il dato inserito) con quello del nodo corrente, stabilendo dunque se si può trattare di un suo figlio destro o sinistro.

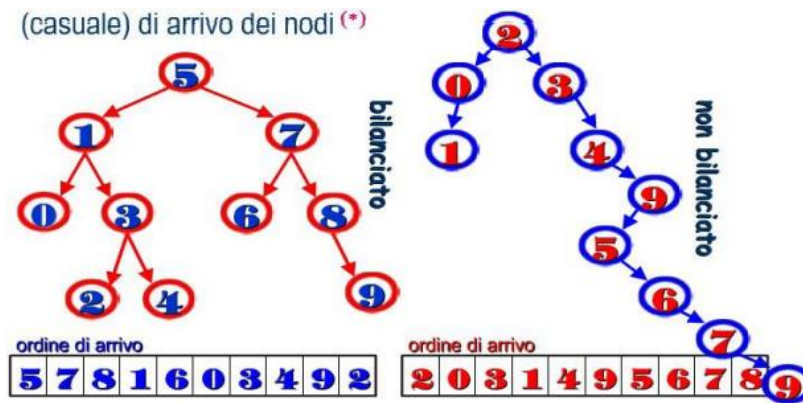
```
char function b_search_ric(KeyType key_dato, b_tree **p_btree) {
    if (*p_btree) == NULL
        return 0; //falso, non trovato!
    else {
        if key_dato == (*p_btree)->key
            return 1; //vero, trovato!
        else {
            if key_dato < (*p_btree)->key {
                *p_btree = (*p_btree)->pt_sx;
                return b_search_ric(key_dato, p_btree); //continua nel sottoalbero
sinistro}

            else {
                *p_btree = (*p_btree)->pt_dx;
                return b_search_ric(key_dato, p_btree); //continua nel sottoalbero
destro}
            }
        }
    }
}
```

Bilanciamento

Il **bilanciamento di un albero binario dipende fortemente dall'ordine di arrivo dei nodi.**

Un albero binario ordinato risulta molto efficiente nella ricerca solo se questo si presenta bilanciato. Nel caso di un albero sbilanciato invece, il costo dell'algoritmo di ricerca binaria degrada ad una complessità dell'ordine di n , ossia lineare.



Struttura dati Heap

Un **heap** è una struttura dati utilizzata in informatica, più precisamente è un albero binario completo o quasi completo usato principalmente per la memorizzazione di collezioni di dati, dette dizionari.

Questi tipi di alberi hanno la seguente proprietà: se x è un qualsiasi nodo dell'heap (ad esclusione della radice) si ha $key(x) \leq key(padre(x))$. Ne consegue che in un heap, la chiave di valore massimo è memorizzata nella radice (che non ha padre).

- Nel **max-heap** la chiave del nodo padre è sempre maggiore o uguale alla chiave dei nodi figli. Di conseguenza in un max-heap il nodo con la chiave più grande sarà sempre nella radice.
- Nel **min-heap** è il contrario, la chiave del nodo padre è sempre minore o uguale a quella dei nodi figli.

Memorizzazione di un Heap

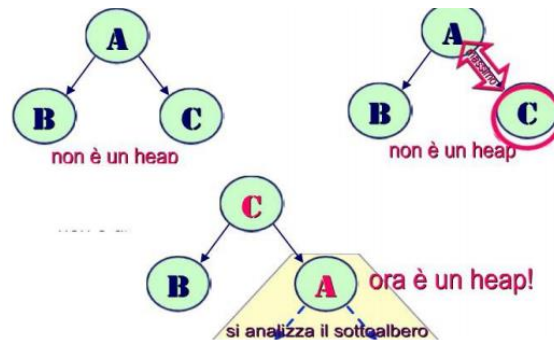
Poiché l'heap di fatto è un albero binario la sua implementazione riprende le tecniche già viste in precedenza di indicizzazione su array statici o di costruzione con liste multiple.

Nel caso di **memorizzazione con array statici**, poiché l'heap deve essere un albero binario almeno quasi completo ne consegue che non vi saranno sprechi di memoria nella sua implementazione. Infatti, la mancanza di alcune foglie determina l'eliminazione di componenti che nell'array, si sarebbero trovate in fondo e che dunque non vanno colmate con nodi fittizi.

Traduzione di un array in un heap – costruzione top-down

L'operazione base che consente di ripristinare la proprietà heap sui nodi di un albero binario quasi completo è chiamata **heapify** e coinvolge un nodo e i suoi figli:

Tra i figli del nodo, si determina il figlio con chiave massima, successivamente si confronta il figlio di chiave massima col padre, ed eventualmente si scambiano i due nodi se non verificano la proprietà heap. Se è avvenuto lo scambio, si scende poi nel relativo sottoalbero per ripristinare la proprietà heap.



Per ogni inserimento si effettuano le seguenti operazioni:

- si aggiunge il nuovo elemento nella prima componente disponibile dell'array
- si confronta l'elemento appena aggiunto col padre:
 - se sono in ordine corretto (figlio < padre), il processo termina
 - se non sono nell'ordine corretto (figlio > padre), i due nodi vengono scambiati e si ritorna al passo precedente

Costruzione bottom-up di un heap

Si vuole ora risolvere un problema leggermente diverso dal precedente: partendo da un albero binario almeno quasi completo, si vuole tradurlo in un heap. La soluzione più semplice consiste nell'usare la procedura **heapify** in maniera bottom-up, cioè dal basso verso l'alto (dalle foglie verso la radice). È possibile applicare la procedura heapify a partire dal penultimo livello dell'albero binario, più precisamente dal padre dell'ultima foglia.

Per sapere dove e fino a quando applicare la procedura heapify occorre semplicemente arretrare sull'array che rappresenta l'albero binario, considerando di volta in volta un nodo figlio e tramite i metodi di indicizzazione già visti, il suo nodo padre. Il meccanismo si ripete fino a quando l'array è stato trattato per intero, ossia fino al raggiungimento della radice.

Algoritmo di visita di un albero binario mediante array

```
#include <stdio.h>
#include <stdlib.h>
typedef struct {
    int info;
    int flag; //indica se il nodo contiene o no l'informazione
} BTreeNode;
void buildBTree(int N, BTreeNode Btree[]) {
    Btree[1].info = 0;
    Btree[1].flag = 1;
    Btree[2].info = 1;
    Btree[2].flag = 1;
    Btree[3].info = 2;
    Btree[3].flag = 1;
    Btree[4].info = 3;
    Btree[4].flag = 1;
```



```
Btree[5].info = 4;
Btree[5].flag = 1;
Btree[6].info = 5;
Btree[6].flag = 1;
Btree[7].info = 6;
Btree[7].flag = 1;
Btree[8].info = 0;
Btree[8].flag = 0;
Btree[9].info = 0;
Btree[9].flag = 0;
Btree[10].info = 7;
Btree[10].flag = 1;
Btree[11].info = 8;
Btree[11].flag = 1;
Btree[12].info = 0;
Btree[12].flag = 0;
Btree[13].info = 0;
Btree[13].flag = 0;
Btree[14].info = 0;
Btree[14].flag = 0;
Btree[15].info = 9;
Btree[15].flag = 1;

}

//ritorna k se esiste il padre
int padre(int i) {
    if (i <= 1)
        return -1; //nodo radice

    int k; //indice del nodo padre
    k = i/2;
    return k;
}

//N: indice ultima componente array
int figlioSX(int N, int i) {
    int k = 2*i;
    if (k >= N)
        return -1; //non ha il figlio sinistro

    return k;
}

int figlioDX(int N, int i) {
    int k = 2*i + 1;
    if (k >= N)
        return -1; //non ha il figlio destro
```

```

    return k;
}

void preOrderVisit(int N, Btnode Btree[], int i) {
    if (!Btree[1].flag)
        return;
    printf("\t%d\n", Btree[i].info);
    int k = figlioSX(N, i);
    if (k != -1)
        preOrderVisit(N, Btree, k);
    else
        return;
    k = figlioDX(N, i);
    if (k != -1)
        preOrderVisit(N, Btree, k);
    else
        return;
}

int main () {
    int N = 16; //numero dei nodi + 1
    Btnode Btree[16];
    buildBTree(N, Btree);

    preOrderVisit(N, Btree, 1);

    return 0;
}

```

Algoritmo per trasformare un array in un max heap

```

void maxHeapify(itemType a[], int n, int idx) {
    //trova il massimo tra un nodo ed i suoi nodi figli
    int largest = idx;
    int left = idx*2 + 1;
    int right = idx*2 + 2;
    if (left < n && a[left] > a[largest])
        largest = left;
    if (right < n && a[right] > a[largest])
        largest = right;

    //Effettua lo scambio se necessario
    if (largest != idx) {
        swap(&a[largest], &a[idx]);
        maxHeapify(a, n, largest);
    }
}

//CHIAMATA: for (int i = n/2-1; i >= 0; --i) maxHeapify(a, n, i);

```

Capitolo 10 – Strutture dati dinamiche reticolari

Struttura dati grafo

Il grafo è una struttura reticolare. Come l'albero, esso è un insieme di **nodi** (o vertici) e **archi** (o lati). I grafi si distinguono in:

- Grafo non orientato, un grafo che presenta archi che possono essere percorsi in entrambi i sensi. Gli archi non sono unidirezionali come nell'albero, dunque è possibile spostarsi lungo gli archi in entrambe le direzioni. In pratica è possibile spostarsi da un nodo A ad uno B e viceversa, sempre lungo lo stesso arco.
- Grafo orientato, un grafo che presenta archi a cui è assegnato un verso di percorrenza. Gli archi sono unidirezionali, quindi è possibile spostarsi solo lungo una direzione.

Cammino

Per **cammino** si intende una **lista di vertici connessi da archi che va da X ad Y**. Oltre al cammino generico si usa distinguere un **cammino semplice** (in cui non è possibile percorrere due volte lo stesso nodo) e **cammino minimo** (un cammino semplice composto dal minimo numero di nodi ed archi possibile).

Definizioni

Ciclo Euleriano: ciclo che attraversa ogni arco esattamente una volta.

Ciclo Hamiltoniano: ciclo che visita ogni nodo (tranne il primo) esattamente una volta.

Rappresentazione in memoria di un grafo

I grafi possono essere memorizzati in memoria in due modi:

- tramite matrice di adiacenze
- tramite lista di adiacenze

Matrice di adiacenza

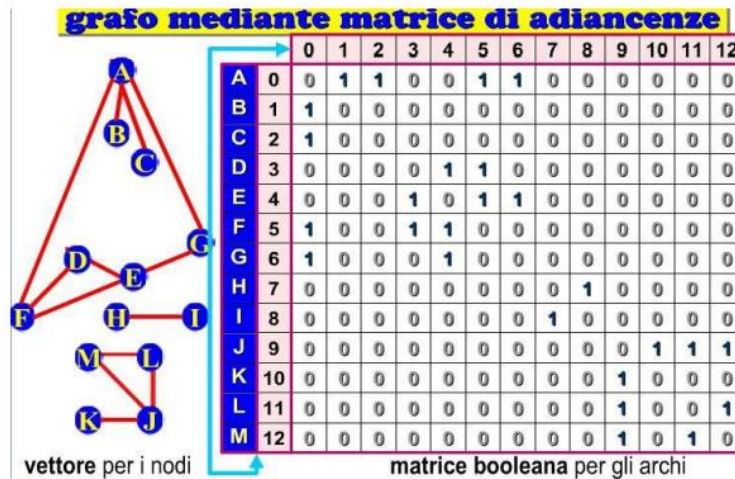
Se si usa la matrice di adiacenze per rappresentare un **grafo non orientato** equivale a produrre una potenziale matrice sparsa (molti zeri) contenente le informazioni circa i collegamenti tra i nodi. In pratica, gli "1" indicano che c'è un collegamento tra i due nodi coinvolti.

La matrice:

- è **booleana** (binaria) perché presenta solo bit 0 o 1; per questo motivo è inutile usare, ad esempio, il tipo char ma conviene usare le notazioni compatte e poi per estrarre si usano gli operatori bitwise
- è **quadrata** perché ha lo stesso numero di righe e colonne; il suo size è N^2 , ossia $M=(N \times N)$ dove N è il numero dei nodi del grafo



- è **simmetrica** perché il suo triangolo superiore e quello inferiore coincidono, poiché il numero di righe e colonne è legato ai nodi
- presenta sulla **diagonale principale tutti 0** perché il grafo non ha cappi (nodi collegati con sé stessi)



Costruzione della matrice di adiacenza di un grafo non orientato

Per costruire una matrice di adiacenza come quella riportata sopra è necessario che per ogni nodo vengano riportati quelli cui questo è collegato. Cioè, per ogni nodo, è necessario elencare i nodi adiacenti. Il numero dei nodi N individua la dimensione della matrice di adiacenze in quanto questa ha dimensioni $M(N \times N)$, cioè è quadrata.

Poiché il grafo è non orientato, la matrice delle adiacenze sarà simmetrica. Poiché è simmetrica, si può pensare di apportare una miglioria significativa memorizzando solo una delle due metà della matrice.

Costruzione della matrice di adiacenza di un grafo orientato

Per un **grafo orientato** la costruzione della matrice delle adiacenze ovviamente si complica.

È necessario, infatti, che oltre alla semplice informazione riguardante l'esistenza di un collegamento tra due nodi, sussista il dato riguardante il verso del collegamento stesso. Ancora una volta si produrrà una matrice di size N^2 , ma questa volta **non simmetrica**. L'input consisterà nell'insieme dei nodi raggiungibili da quello inteso come punto di partenza.

Matrice di adiacenza come lista multipla

In quanto potenziale matrice sparsa, la matrice di adiacenza può naturalmente essere implementata mediante liste multiple ([click per vedere come](#)). Poiché però si tratta di una matrice contenente dati di tipo binario (0 ed 1) oltre ad evitare la rappresentazione degli zeri si può anche tralasciare la registrazione del campo informazione del nodo con un dato non nullo in quanto è già noto a priori che quest'ultimo potrà contenere solo 1. Quindi, si sa a priori, che tutti i nodi presenti hanno valore 1, perciò è inutile memorizzare tale valore esplicitamente.

Lista di adiacenze

I nodi di un grafo possono essere elencati mediante liste di adiacenze, in cui ogni nodo punta alla serie di nodi cui è collegato. Abbiamo un array di struct (con campo informazione e campo puntatore, i puntatori puntano a delle liste) contenente tutti i nodi e per ogni nodo c'è un puntatore che punta ai suoi nodi adiacenti.

Evitare la ridondanza

In alcuni casi, può capitare che uno stesso nodo è memorizzato più volte secondo il numero dei nodi cui è collegato. Si può pensare di sostituire ai "duplicati" i puntatori gli originali, evitando in questo modo di memorizzare le stesse informazioni più volte.

Algoritmi di visita di una struttura reticolare

Un albero generico è un albero che non ha limitazioni inerenti al numero dei figli che un nodo può avere. La visita in ordine anticipato di un albero generico è simile all'algoritmo di visita preorder di un albero binario, dove viene visitata prima la radice, poi il sottoalbero di sinistra ed infine quello di destra.

Algoritmo DFS (Depth-First-Search) per la visita di un grafo

L'algoritmo DFS è utilizzato per la visita dei grafi e si basa sull'idea di visitare un nodo V, per poi visitare (ricorsivamente) ogni nodo U adiacente a V e non ancora visitato.

Se il grafo è connesso verranno visitati tutti i nodi. Se il grafo non è connesso viene visitato il sottografo connesso che ha come nodo il nodo da cui parte la visita, cioè viene visitato il sottografo connesso a cui appartiene V.

Versione iterativa dell'algoritmo DFS

L'algoritmo iterativo della visita in profondità di un grafo viene realizzato utilizzando esplicitamente una pila che contiene i nodi da elaborare (*stack esplicito*).

Questo algoritmo è capace di tradurre un qualsiasi grafo in un albero, chiamato albero di ricerca depth first che non ha tutti gli archi del grafo: sono presenti soltanto quelli che vengono attraversati una sola volta, a differenza degli originali che vengono attraversati due volte. Per ciascun nodo si utilizza un flag per indicare se è stato già visitato.

Nel primo passo si inseriscono nella pila la sorgente (specificata in input) e i nodi adiacenti (si pushano con un ciclo for partendo dalla sorgente). Dal passo successivo si cicla fino a quando la pila non è vuota e:

- si estrae un nodo dalla pila (operazione di POP)
- si visita se non è stato ancora visitato
- si inseriscono i nodi adiacenti nella pila

N.B. L'ordine di inserimento dei nodi nella pila si ripercuote sull'ordine di visita dei nodi. La **complessità di spazio** è pari al valore assoluto di $O(V)$, quella di **tempo** al valore assoluto di $O(V+E)$.

Miglioramenti dell'algoritmo DFS

Si può introdurre un contatore di vertici visitati in modo tale che raggiunto il size del grafo, l'esecuzione dell'algoritmo termini senza rischio di errore ed avendo di fatto, visitato tutti i nodi.

```
typedef struct {
    char nome;
} INFO_FIELD;
typedef struct grafo_labirinto {
    INFO_FIELD info;
    short Adiacenze[MAX_ADIAENZE];
    short n_nodi_adiac;
    short visitato;
} VERTICE_LABIRINTO;

void DFS_visita_labirinto(VERTICE_LABIRINTO Labirinto[], short inizio_lab) {
    /* n_cont conta i nodi visitati */
    short i_testa=-1, Stack[LEN_STACK], n_cnt=1, i, n_adiacenze=0;
    short i_estratto; /* Quando estraggo l'indice dalla pila, lo metto qui */
    /* VISITA LA RADICE DA CUI COMINCIA (inizio_lab) e PUSHO le adiacenze */
    printf("[%c] ", Labirinto[inizio_lab].info.nome); //Visita radice
    Labirinto[inizio_lab].visitato=1;
    //Inserisci le adiacenze nello stack per n nodi adiacenti ad esso
    for(i=0; i<Labirinto[inizio_lab].n_nodi_adiac; i++)
        push(Stack, Labirinto[inizio_lab].Adiacenze[i], &i_testa);
    while(i_testa!=-1 && n_cnt<MAX_INCROCI) {
        /* ESTRAI NODO ADIACENTE */
        pop(Stack, &i_estratto, &i_testa);
        /* SE IL NODO NON È GIA' STATO VISITATO, VISITALO */
        if (Labirinto[i_estratto].visitato==0) {
            //Visita nodo
            printf("[%c] ", Labirinto[i_estratto].info.nome);
            Labirinto[i_estratto].visitato=1;
            n_cnt++; /* Ad ogni estrazione, conto */

            //INSERISCI ADIACENZE NELLO STACK per n nodi adiacenti al nodo estratto
            n_adiacenze = Labirinto[i_estratto].n_nodi_adiac;
            for (i=0; i<n_adiacenze; i++)
                push(Stack, Labirinto[i_estratto].Adiacenze[i], &i_testa);
        }
    }
}

/* Push: Inserire in testa dell'array, appunto, come lo stack. */
void push(short Vet_Stack[], short Elemento, short *Testa) {
```

```

if(*Testa<LEN_STACK) //Se non ho superato il massimo dello stack
{
    // *Testa=*Testa+1 -> Vet_Stack=Elemento
    Vet_Stack[++(*Testa)]=Elemento;
}
else {
    puts("\n*----- NON INSERISCO NULLA, STACK PIENO! -----*\n");
    exit(1);
}
}

/* Pop: Eliminare dalla testa dell'array. */
void pop(short Vet_Stack[], short *i_estratto, short *Testa) {
    if(*Testa>-1) //Se c'è qualcosa
    {
        *i_estratto = Vet_Stack[*Testa];
        /* Decrementando l'indice dell'ultimo elemento inserito, appunto, non verrà più
        "visto" l'ultimo elemento inserito, nascondendolo*/
        (*Testa)--;
        //altrimenti "Stack vuoto" viene gestito nel main
    } else {
        puts("---- ERRORE Stack vuoto!! ----");
        exit(1);
    }
}

```

Algoritmo BFS (Breadth-First-Search, visita in ampiezza)

L'**algoritmo BFS** è uno degli algoritmi di visita per i grafi. Esso visita il grafo in ampiezza.

L'idea è quella di partire da un nodo, detto nodo sorgente, per poi esplorare tutti i nodi del grafo, sia quelli raggiungibili dal nodo sorgente che quelli non raggiungibili, utilizzando una struttura dati esterna, la coda. Per ciascun nodo si utilizza:

- un **flag** per indicare se è stato già visitato
 - bianco se non è stato visitato
 - grigio se è stato inserito nella coda
 - nero se è stato visitato e i suoi nodi adiacenti sono stati inseriti nella coda
- un **numero** (livello) che indica il numero di archi da attraversare per raggiungere quel nodo a partire dalla sorgente. Il livello è pari al livello del nodo adiacente +1.

Il tempo di esecuzione totale di questo algoritmo è $O(V+E)$ dove:

- V è l'insieme dei vertici del grafo
- E è il l'insieme degli archi che collegano i vertici

Nel primo passo si inserisce nella coda il nodo sorgente. Dal passo successivo:

- si estrae un nodo dalla coda e lo si visita
- si inseriscono nella coda i suoi nodi adiacenti non ancora visitati (bianchi)
- si inseriscono nella coda i nodi adiacenti (quindi diventano grigi)
- si visita il nodo successivo (quindi, si colora di nero)
- si imposta il livello dei nodi adiacenti bianchi (aggiungendo 1 al livello del nodo corrente)

Confronto algoritmi BFS-DFS

La differenza principale tra l'algoritmo BFS e DFS sta naturalmente nell'ordine di visita dei nodi del grafo cui sono applicati. Infatti, l'algoritmo BFS (in ampiezza) si allarga "a ventaglio", partendo da un nodo sorgente per poi analizzare quelli che gli sono vicini, mentre l'algoritmo DFS (in profondità) corrisponde alla visita in ordine anticipato dell'albero depth-first ben diverso dall'albero BFS in quanto solitamente meno ramificato e molto meno distribuito.

Capitolo 11 – Tecniche di programmazione ricorsiva

Ricorsione

Un algoritmo si dice **ricorsivo** se richiama sé stesso direttamente o indirettamente. L'algoritmo richiama sé stesso generando una sequenza di chiamate che ha termine al verificarsi di una condizione particolare che viene chiamata **condizione di terminazione**, che in genere si ha con particolari valori di input.

Struttura generale di una procedura ricorsiva

La stesura di una procedura ricorsiva deve rispettare alcuni requisiti.

Chiamata ricorsiva – deve essere presente una chiamata a sé stessa.

Caso banale – deve essere presente un caso base, o condizione di terminazione, che eviti l'entrata della procedura in un loop infinito. Il caso banale è alternativo alla chiamata ricorsiva e, solitamente, risulta essere un sottoproblema del problema che si sta risolvendo.

Tipi di ricorsione: classificazione

Esistono vari tipi di ricorsione.

Una prima classificazione suddivide la ricorsione in:

- ricorsione **diretta**, in cui c'è una procedura che al suo interno prevede una chiamata a sé stessa
- ricorsione **indiretta**, che avviene indirettamente mediante un'altra procedura, cioè quando nell'algoritmo una funzione ne richiama un'altra che a sua volta richiama la prima

Una seconda classificazione suddivide la **ricorsione diretta** in:

- ricorsione **lineare**, che si ha quando vi è solo una chiamata ricorsiva all'interno della funzione
- ricorsione **binaria**, che si ha quando le chiamate ricorsive all'interno della funzione sono due
- ricorsione **non lineare**, che si ha quando la chiamata ricorsiva si trova all'interno di un ciclo ripetitivo. In questo caso si potrebbe non sapere il numero di volte che la chiamata ricorsiva viene attivata.

Tipi di ricorsione: ricorsione lineare

La ricorsione lineare prevede che nel corpo della procedura compaia una sola chiamata ricorsiva. Esempi di ricorsione lineare possono essere il calcolo del fattoriale e il calcolo del MCD.

Esempio

```
long recurs_MCD(long m, long n) {  
    if (n==0) return m;  
    else return recurs_MCD(n, m%n);  
}
```

```
long recurs_fact(short n) {  
    if (n <= 1) return 1;  
    else return n*recurs_fact(n-1);  
}
```

Tipi di ricorsione: ricorsione binaria

La ricorsione binaria prevede che nel corpo della procedura compaiano due chiamate ricorsive. In questo tipo di ricorsione, il problema viene suddiviso in due sottoproblemi aventi ciascuno una propria chiamata ricorsiva, e poi la chiamata ricorsiva si applica a uno solo di questi o ad entrambi i sottoproblemi. Esempi di ricorsione binaria sono i numeri di Fibonacci oppure il calcolo del massimo in un array.

Esempi

```
int recurs_Fibo(int n) {
    if (n <= 1) return 1;
    else return recurs_Fibo(n-1) + recurs_Fibo(n-2);
}

int recurs_massimo(char v[], short inizio, short fine) {
    short mez; char t1, t2;
    if (inizio == fine)
        return v[fine];
    else {
        mez = (inizio+fine)/2;
        t1 = recurs_massimo(v, inizio, mez);
        t2 = recurs_massimo(v, mez+1, fine);
        if (t1 < t2)
            return t2;
        else
            return t1;
    }
}
```

Tipi di ricorsione: ricorsione non lineare

La ricorsione non lineare prevede che la chiamata ricorsiva si trovi all'interno di un ciclo ripetitivo. Un esempio di ricorsione non lineare è il calcolo di tutte le disposizioni con ripetizione dei primi n numeri naturali presi "r" alla volta.

Esempio

```
#include <stdio.h>
#include <stdlib.h>
void sample(int, int, int, int[]);
void main() {
    int n, r, column[20];
    n=3; r=2;
    sample(n, r, 1, column);
}

void sample(int n, int r, int k, int column[]) {
    int j;
    column[k]=0;
    while (column[k] < n) {
```

```

        column[k] = column[k]+1;
        if (k < r)
            sample(n, r, k+1, column);
        else {
            for (j=1; j<=r; j++)
                printf("\t%d", column[j]);
                puts("");
            }
        }
    }
}

```

Tipi di ricorsione: ricorsione indiretta

La ricorsione indiretta o di mutua ricorsione prevede che una funzione richiami un'altra funzione che a sua volta richiama la prima. Un esempio è quello del calcolo del fattoriale.

Esempio

```

long recurs_fact(short n) {
    long nfatt;
    if (n <= 1)
        nfatt = 1;
    else
        nfatt = prod_fact(n);
    return nfatt;
}

long prod_fact(short m) {
    return m*recurs_fact(m-1);
}

```

Analisi di funzioni ricorsive

Gestione della ricorsione da parte del compilatore

Quando si utilizza la programmazione ricorsiva, è necessario utilizzare uno stack in cui vengono memorizzate tutte le autoattivazioni generate dalle chiamate ricorsive. In particolare, è il compilatore che gestisce queste autoattivazioni utilizzando uno stack che, viene prima completamente riempito, poi, una volta raggiunto il caso base, ogni processo può essere concluso e lo stack viene liberato.

La ricorsione semplifica la leggibilità dell'algoritmo, ma non è sempre conveniente per la complessità di spazio e di tempo.

Profondità di ricorsione

Un parametro importante per poter valutare l'occupazione di spazio e la complessità di tempo di un algoritmo ricorsivo è la **profondità di ricorsione** che è definita come il massimo numero di chiamate ricorsive effettivamente eseguite.

La profondità di ricorsione moltiplicata per il numero di variabili usate dal sottoprogramma (variabili interne e parametri formali) quantifica l'occupazione di memoria dello stack di dati temporanei della procedura ricorsiva.

Analisi della profondità di ricorsione

Con **complessità computazionale** si intende sia la complessità di tempo che la complessità di spazio.

Ricorsione lineare

Nel caso della **ricorsione lineare**, la profondità di ricorsione è **dell'ordine di n** , cioè **$O(n)$** . Ciò dipende dal fatto che, per una dimensione computazionale di n , verranno effettuate proprio n chiamate ricorsive in quanto una ricorsione lineare prevede una singola chiamata all'interno del sottoprogramma.

Ricorsione binaria

Nel caso del calcolo **dell' n -esimo numero di Fibonacci** la profondità di ricorsione risulta esponenziale. Questo perché ogni chiamata a sua volta genera due chiamate.

Nel caso del calcolo del massimo in un array la profondità di ricorsione risulta lineare.

Entrambi seguono l'approccio classico della ricorsione binaria, cioè suddividono il problema iniziale in due sottoproblemi per poi risolverli indipendentemente, ma:

- nel caso di Fibonacci, i due sottoproblemi non sono indipendenti perché Fibonacci è una formula iterativa (cioè il valore attuale dipende dai precedenti), per questo motivo la ricorsione non è adatta alla risoluzione di questo problema
- nel caso della ricerca del massimo, i due sottoproblemi sono indipendenti ed ognuno di essi ha una dimensione dimezzata rispetto a quella di partenza, quindi è come se si risolvesse un problema a complessità lineare

Altri esempi di algoritmi ricorsivi

Algoritmo di Horner

L'algoritmo di Horner permette di valutare un polinomio di grado n in una fissata ascissa. Questo algoritmo prevede di mettere in evidenza la x tra tutti i termini escluso il valore del termine noto (quindi a_n rimane fuori). In questo modo è possibile valutare il polinomio andando a sommare il termine noto a x moltiplicato il valore di un polinomio di grado $n-1$ (ricorsivamente).

La versione iterativa dell'algoritmo calcola il valore del polinomio valutando la formula dalle parentesi più interne a quelle più esterne mentre quella ricorsiva opera dalle parentesi più esterne a quelle più interne (ogni livello di parentesi è una chiamata alla funzione).

Codice Horner iterativo

/* HORNER ITERATIVO: effettuando un ciclo for per 'grado' volte e partendo dal coefficiente di grado più basso, la funzione calcola il valore del polinomio in un punto prefissato $P(x) = c[N] + x(c[N-1] + x(((\dots c[3] + x(c[2] + x(c[1] + c[0]*x))))))$ */

```
double alg_di_Horner(double c[], double x, short grado)
{
    short i=grado;
    long double ris=c[0];
    for(i=1; i<=grado; i++)
        ris = c[i]+x*ris;
    return ris;
}
```

Codice Horner ricorsivo

/* HORNER RICORSIVO: Seguendo la formula ricorrente, ricaviamo che effettuando le autochiamate con $\text{coef}[\text{grado}] + x(\text{horner del grado precedente})$, inizierà a restituire $x[0]$ che si combinerà il risultato e così via, rispecchiando la forma di sotto: e' bottom up. Se ho x^3 , avro' ad esempio 4 coefficienti (il termine noto) $P(x) = c[N] + x(c[N-1] + x(((\dots c[3] + x(c[2] + x(c[1] + c[0]*x))))))$ */

```
double Horner_ricorsivo(double c[], double x, short grado)
{
    if(grado==0) return c[0];
    return c[grado]+x*Horner_ricorsivo(c,x, grado-1);
}
```

Costruzione ricorsiva di una lista lineare

```
#include <stdio.h>
#include <stdlib.h>
typedef char DATA;
struct linked_list {
    DATA d;
    struct linked_list *next;
};
typedef struct linked_list ELEMENT;
typedef ELEMENT *LINK;

LINK array_to_list(DATA []);
void main() {
    DATA c_arr[] = "questa e' una prova!";
    LINK head_list, p_list;
    head_list = array_to_list(c_arr);
    p_list = head_list;
    while (p_list != NULL) {
        putchar(p_list->d);
    }
}
```

```

        p_list=p_list->next;
    }
    puts("");
}
LINK array_to_list(DATA s[]) {
    LINK head;
    if (s[0] == '\0')
        return NULL;
    else {
        head = malloc(sizeof(ELEMENT));
        head->d = s[0];
        head->next = array_to_list(s+1);
        return head;
    }
}

```

Visita ricorsiva di un albero binario

Codice Inorder: ricorsivo

```

visita_inorder (nodo *T) {
    if (T != NULL) {
        visita_inorder(T -> SX);
        visita(T);
        visita_inorder(T -> DX);
    }
}

```

Codice Preorder: ricorsivo

```

visita_preorder (nodo *T) {
    if (T != NULL) {
        visita(T);
        visita_preorder(T -> SX);
        visita_preorder(T -> DX);
    }
}

```

Codice Postorder: ricorsivo

```

visita_postorder (nodo *T) {
    if (T != NULL) {
        visita_postorder(T -> SX);
        visita_postorder(T -> DX);
        visita(T);
    }
}

```

Ricerca ricorsiva su un albero binario di ricerca

Organizzazione dei dati

```
typedef char DATA;  
struct b_tree {  
    DATA d;  
    struct b_tree *figlio_sx;  
    struct b_tree *figlio_dx;  
};  
typedef struct b_tree ELEMENT;  
typedef ELEMENT *LINK;
```

Algoritmo

```
LINK btree_search(DATA elem, LINK radice) {  
    LINK ptr;  
    if (radice == NULL)  
        return NULL;  
    else {  
        if (elem == radice->d)  
            return radice;  
        else if (elem < radice->d)  
            ptr = btree_search(elem, radice->figlio_sx);  
        else  
            ptr = btree_search(elem, radice->figlio_dx);  
    }  
}
```

Capitolo 12 – Algoritmi di ordinamento a complessità quadratica

Un **algoritmo di ordinamento** è un algoritmo che viene usato per elencare gli elementi di un insieme secondo una sequenza stabilita da una relazione di ordine, in modo che ogni elemento sia minore (o maggiore) di quello che lo segue.

Il parametro più importante è l'**efficienza** di spazio e di tempo in funzione del tipo di dati a cui questi algoritmi andranno applicati. Spesso bisogna anche esaminare il caso peggiore e il caso migliore. Nel caso degli algoritmi di ordinamento, il caso migliore sarebbe quello di dati ordinati, il caso peggiore quello di dati completamente disordinati (cioè ordinati al contrario).

Classificazione degli algoritmi di ordinamento

Una prima distinzione tra gli algoritmi di ordinamento viene fatta relativamente a dove è memorizzato l'oggetto da ordinare. Si hanno quindi:

- **metodi di ordinamento interni**, se i dati risiedono in memoria centrale
- **metodi di ordinamento esterni**, se i dati risiedono su memoria di massa e in memoria centrale

Le principali strutture dati per gli algoritmi di ordinamento sono gli **array** e le **liste lineari**.

Una seconda distinzione tra gli algoritmi di ordinamento viene fatta relativamente al tipo di scambio effettuato nell'algoritmo. Infatti, negli algoritmi di ordinamento, l'operazione di base è quella di scambio delle informazioni. Esistono algoritmi:

- che effettuano **scambi reali**, quando i dati sono fisicamente scambiati
- che effettuano **scambi virtuali**, quando i dati non sono fisicamente scambiati, ma sono scambiati i puntatori per evitare di perdere tempo se ci sono troppi dati

Nel caso di scambi virtuali, si accede ai dati utilizzando un array ausiliario di puntatori. Gli scambi avverranno su questo array di puntatori e non sui dati, che rimarranno memorizzati nella stessa posizione occupata precedentemente.

Una terza distinzione tra gli algoritmi di ordinamento viene fatta relativamente alla proprietà di stabilità. Esistono algoritmi di ordinamento:

- **stabili**, se l'ordinamento delle chiavi uguali viene preservato
- **instabili**, se l'ordinamento delle chiavi uguali non viene preservato

Selection Sort

L'**algoritmo per selezione (selection sort)** è un algoritmo di ordinamento che opera in-place.

Nell'algoritmo di selezione per minimo, l'idea è quella di suddividere il vettore in due porzioni:

- una sequenza ordinata
- una sequenza disordinata

Ad ogni passo si ricerca il valore minimo dell'array e lo si sposta nella sua posizione definitiva nella sequenza ordinata attraverso uno scambio. Ad ogni iterazione, la sequenza ordinata cresce mentre

la sequenza disordinata si riduce. L'algoritmo termina quando il vettore disordinato si è svuotato, cioè dopo $n-1$ passi (l'indice i va da 0 a $n-1$).

Nell'algoritmo di selezione per massimo, l'idea è la medesima dell'algoritmo precedente. La differenza sostanziale è che ad ogni passo si ricerca il massimo del vettore, e una volta trovato, lo si sposta nella sua posizione definitiva nella sequenza ordinata. La sequenza ordinata, in questo algoritmo, si troverà sulla destra, al contrario di quanto succedeva prima (quindi i va da $n-1$ a 0).

L'algoritmo è di tipo non adattivo, ossia il suo tempo di esecuzione non dipende dall'input ma dalla dimensione dell'array.

La procedura è quella di inizializzare un puntatore i che va da 1 a n (dove n è la lunghezza dell'array), per poi:

- 1) cercare il più piccolo/più grande elemento dell'array
- 2) scambiare l'elemento più piccolo/più grande con l'elemento alla posizione i
- 3) incrementare/decrementare l'indice i

L'algoritmo si ripete fino alla fine dell'array.

Analisi della complessità

La **complessità di spazio** di questo algoritmo è $O(n)$ in quanto l'algoritmo opera in-place ed n è la dimensione dell'array da ordinare.

Per la **complessità di tempo** si può tener conto del numero di confronti e del numero di scambi. Il **numero di confronti** effettuati ha complessità $O(\frac{1}{2}n^2)$, infatti al primo passo vengono effettuati $n-1$ confronti, al secondo passo $n-2$, al terzo passo $n-3$ e così via, quindi la complessità è quadratica. Il **numero di scambi** ha complessità al più $O(n)$ in quanto ad ogni passo viene eseguito al più uno scambio, quindi la complessità è lineare.

Codice

```
typedef struct Persona {
    char Nome[MAX_NAME];
    short Eta;
} PERSONA;

void selection_sort_Ite(PERSONA Vet[], short N) {
    short i, i_Min;
    for(i=0; i<N-1; i++) {
        i_Min = Min(&Vet[i], N-i);
        Scambia(&Vet[i], &Vet[i_Min+i]);
    }
}
```

```

/* Trova il minimo nel vettore */
short Min(PERSONA Vet[], short N) {
    short i=0, i_min;
    i_min = i;
    for(i=1; i<N; i++)
        if(strcmp(Vet[i].Nome, Vet[i_min].Nome)<0 )i_min=i;
    return i_min;
}

/* Scambia 2 valori passati per indirizzo */
void Scambia(PERSONA *a, PERSONA *b)
{
    PERSONA tmp;
    tmp=*a;
    *a=*b;
    *b=tmp;
}

/* L'idea è identica all'iterativo, solo che utilizziamo una variabile 'inizio' che parte da N e con le
autoattivazioni la facciamo diventare 0. Ora a ritroso, quando devono ritornare le chiamate, simula
un ciclo for da inizio=0 fino ad N-1 (controlla la chiamata) */
void selection_sort_Ric(PERSONA Vet[], short N, short inizio) {
    short i_Min;
    /* CASOBASE */
    if(inizio<0) return;
    /* AUTOATTIVAZIONE */
    selection_sort_Ric(Vet, N, inizio-1);
    i_Min = Min(&Vet[inizio], N-inizio);
    Scambia(&Vet[inizio], &Vet[i_Min+inizio]);
}

```

Bubble Sort

Anche l'**algoritmo di ordinamento a bolla (bubble sort)** è un algoritmo che opera in-place.

Questo algoritmo è un algoritmo iterativo, ovvero basato sulla ripetizione di un procedimento fondamentale. Anche qui si suddivide l'array in due porzioni: una sequenza ordinata ed una sequenza disordinata.

L'idea di base è quella di confrontare due elementi adiacenti del vettore disordinato e, se risultano non ordinati, scambiarli di posizione. In pratica, una volta deciso il verso di percorrenza (si può partire dall'inizio dell'array e spostarci in avanti oppure partire dal fondo e spostarci all'indietro), si confrontano gli elementi dell'array a due a due. Questa operazione viene ripetuta fino ad arrivare alla fine dell'array. A questo punto, l'ultimo elemento della porzione disordinata conterrà l'elemento massimo.

Questo meccanismo viene ripetuto più volte (n-1 volte) ed in ogni passo verrà trovato il massimo della porzione disordinata dell'array. Quindi alla fine di ogni passo, l'elemento massimo della porzione disordinata dell'array sarà aggiunto alla porzione ordinata dell'array.

Considerazioni sul bubble sort

Per ogni confronto, se i due elementi confrontati non sono ordinati, essi vengono scambiati. Durante ogni iterazione, almeno un valore viene spostato rapidamente fino a raggiungere la sua collocazione definitiva; in particolare, alla prima iterazione il numero più grande raggiunge l'ultima posizione dell'array, alla seconda iterazione il numero più grande tra quelli disordinati raggiunge la penultima posizione dell'array. Ne conseguono due considerazioni:

1. se i numeri sono in tutto n , dopo $n-1$ iterazioni si avrà la garanzia che l'array sia ordinato
2. all' i -esima iterazione, le ultime $i-1$ celle dell'array ospitano i loro valori definitivi

Ovviamente, ogni iterazione, oltre a portare il numero più grande in fondo all'array, può contribuire anche ad un riordinamento parziale degli altri valori. Per questo motivo, può accadere che l'array sia ordinato prima che sia raggiunta la $n-1$ -esima iterazione.

Analisi della complessità

La **complessità di spazio** di questo algoritmo è $O(n)$ in quanto l'algoritmo opera in-place ed n è la dimensione dell'array da ordinare.

Per la **complessità di tempo** si può tener conto del numero di confronti e del numero di scambi. Il **numero di confronti** effettuati è lo stesso del selection sort, quindi la complessità risulta $O(\frac{1}{2}n^2)$. Il **numero di scambi** risulta maggiore rispetto al selection sort. Infatti, in ogni iterazione, possono essere eseguiti al più $\frac{1}{2}n^2$ scambi, per cui la complessità risulta al più $O(\frac{1}{2}n^2)$. Il numero di scambi dipende dalla proprietà dell'array di essere più o meno ordinato inizialmente.

Codice

/ bubble_sort_lte: Confronta a coppia, facendo "risalire" i valori piccoli. Se c'è uno scambio, si salva l'eventuale ultimo indice in cui è avvenuto lo scambio. Se c'è stato lo scambio, ricicla di nuovo fermandosi però all'ultimo indice salvato, per risparmiare molto tempo */*

```
void bubble_sort_lte(PERSONA V[], short N) {
    short Scambiato, i, ultimo_scambio;
    //Se non avviene scambio, è ordinato
    while(Scambiato) {
        Scambiato=0;
        for(i=0; i<N-1; i++) {
            //Confronto a 2 a 2
            if(strcmp(V[i].Nome, V[i+1].Nome)>0) {
                Scambia(&V[i], &V[i+1]);
                Scambiato = 1;
                ultimo_scambio = i+1; /* Salva eventuale ultimo indice */
            }
        }
        N = ultimo_scambio;
    }
}
```

```

}
/* bubble_sort_Ric: è meno efficiente, non salva l'indice dell'ultimo scambio. Se avviene uno
scambio, richiama sé stesso passando N-1, perché l'ultimo elemento sarà il più grande. */
void bubble_sort_Ric(PERSONA V[], short N) {
    short Scambiato=0, i;
    for(i=0; i<N-1; i++) {
        if(strcmp(V[i].Nome, V[i+1].Nome)>0) {
            Scambia(&V[i], &V[i+1]);
            Scambiato = 1;
        }
    }
    if (Scambiato==0) return; //CASO BASE
    bubble_sort_Ric(V, N--); //AUTOATTIVAZIONE
    return;
}

```

Insertion Sort

Anche l'insertion sort è un algoritmo che opera in-place. Non è altro che un'evoluzione del bubble sort.

L'idea di base è quella di confrontare due elementi adiacenti, però a differenza del bubble sort, i confronti riguardano una sottoporzione dell'array che va sempre più aumentando, e quando essa aumenta e si devono ricominciare i confronti, invece di cominciare dalle prime componenti inizia dalle ultime.

Anche questo algoritmo, quindi, suddivide l'array in due porzioni: una ordinata (quella di sinistra) e una disordinata (quella di destra), che, rispettivamente, crescono e decrescono di dimensione ad ogni passo.

L'algoritmo utilizza due indici:

- uno punta all'elemento da ordinare, cioè al primo elemento della porzione disordinata
- l'altro punta all'elemento immediatamente precedente

Se l'elemento puntato dall'indice j è maggiore di quello a cui punta il primo indice, il secondo indice indietreggia. L'azione si ripete fino a quando si trova nel punto in cui il valore del primo indice deve essere inserito. L'algoritmo così tende a spostare man mano gli elementi maggiori verso destra.

Analisi della complessità

La **complessità di spazio** di questo algoritmo è $O(n)$ in quanto l'algoritmo opera in-place ed n è la dimensione dell'array da ordinare. Per la **complessità di tempo** si può tener conto del numero di confronti e del numero di scambi. Il numero di confronti e il numero di scambi è lo stesso del bubble sort. **Numero di confronti:** complessità $O(\frac{1}{2}n^2)$. **Numero di scambi:** complessità al più $O(\frac{1}{2}n^2)$.

Questo algoritmo risulta, nel caso migliore e nel caso peggiore, uguale all'algoritmo bubble sort. Nel caso medio, invece, risulta 2 volte più veloce del bubble sort e un 40% più veloce del selection sort.

Miglioria dell'insertion sort

Questo algoritmo può essere ulteriormente migliorato evitando l'operazione di scambio riducendola ad assegnazioni, utilizzando un'area di lavoro aggiuntiva per memorizzare temporaneamente le informazioni. In questo modo l'algoritmo esegue un'assegnazione invece che uno scambio, e quindi il tempo si riduce ancora di più, al costo di un nodo aggiuntivo di informazioni.

Codice

```
void insertion_sort(PERSONA V[],short N) {
    PERSONA el_da_ins;
    short i, j;
    for(i=1; i<N; i++) {
        el_da_ins = V[i];
        j=i-1;
        while(j>=0 && (strcmp(V[j].Nome, el_da_ins.Nome)>0)) {
            V[j+1] = V[j];
            j--;
        }
        V[++j]=el_da_ins;
    }
}
```

//oppure in char (come fatto in PROG 1)

```
void insertion_sort(char array[],short N) {
    char el_da_ins;
    short i, j;
    for(i=1; i<N; i++) {
        el_da_ins = array[i];
        j=i-1;
        while(j>=0 && array[j]>el_da_ins) {
            array[j+1] = array[j];
            j--;
        }
        array[++j]=el_da_ins;
    }
}
```

Capitolo 13 – Algoritmi di ordinamento della classe “Divide et Impera”

L'approccio **divide et impera** afferma che: partendo dall'istanza da risolvere, si genera una sequenza di istanze via via più semplici del problema, fino all'istanza che non è più suddivisibile e che ha soluzione banale. Quindi, a ogni passo, la soluzione dell'istanza da risolvere viene espressa in termini delle soluzioni delle istanze più semplici in cui essa è decomposta.

Merge Sort

Il primo algoritmo di ordinamento della classe “Divide et Impera” è il Merge Sort.

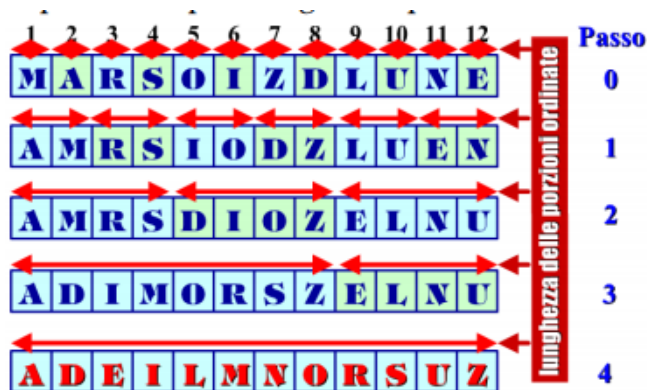
Richiamo: merge di array ordinati

Abbiamo due array ordinati di input ed un array di output. L'array di output viene costruito in questo modo: di volta in volta vengono confrontate due chiavi degli array di input e la chiave minima viene inserita nell'array di output. Il puntatore avanza sull'array che ha fornito la chiave da inserire nell'array di output.

Quando si verifica che uno dei due array è finito, i nodi rimanenti dell'altro array vengono inseriti nell'array di output, che avrà una dimensione pari alla somma delle dimensioni dei due array di input (non è in-place).

Idea dell'algoritmo

Il Merge Sort è un algoritmo efficiente che non opera in-place. L'idea è quella di ordinare un array applicando il merge (la fusione) su sottovettori ordinati. In pratica, l'algoritmo consta di vari passi e ad ogni passo viene applicato il merge su sottovettori di dimensione via via crescente.



Passo 0: merge applicato su porzioni di un solo elemento.

Passo 1: merge applicato su porzioni di due elementi.

Passo 2: merge applicato su porzioni di quattro elementi.

Passo 3: merge applicato su porzioni di otto elementi.

Analisi della complessità

La **complessità di spazio** di questo algoritmo è $O(n)+O(n)$ in quanto l'algoritmo non opera in-place ma ha bisogno, a causa dell'algoritmo di merge, di un'area di lavoro aggiuntiva.

La **complessità di tempo** di questo algoritmo è $O(n \log_2 n)$. Questa complessità viene raggiunta solo se il numero di componenti del vettore è potenza di 2. Se non lo è, le prestazioni sono peggiori.

Codice

```

/* merge_sort: Questa versione ricorsiva, divide il nostro vettore in tanti sottovettori. Se Inizio<Fine
significa che l'algoritmo di merge comincia quando considera minimo un sottovettore di due
elementi. (Inizio = 0, Fine = N-1)*/
void merge_sort(short V[], short Inizio, short Fine) {
    short Mez;
    /* Se ho un vettore con N > 1 */
    if (Inizio<Fine) {
        Mez=(Inizio+Fine)/2;
        merge_sort(V, Inizio, Mez); /* Considera la metà di sinistra */
        merge_sort(V, Mez+1, Fine); /* Considera la metà di destra */
        merge(V, Inizio, Mez, Fine); /* Di quel sottovettore, fai un merge considerando metà
sotto vettore */
    }
    return;
}

/* merge: Dato il sotto vettore, fa la fusione tra il sottovettore compreso tra [inizio, Mez] e [Mez+1,
Fine] e il risultato va in un array in più Aux. Finita la fusione, copia Aux nell'array, a partire da inizio
fino a fine */
void merge(short V[], short Inizio, short Mez, short Fine) {
    /* i=Inizio sottovettore di sinistra Mez=Fine sottovettore di sinistra
j=Inizio sottovettore di destra Fine=Fine sottovettore di Destra
k=Indice del vettore ausiliare y=Lo sfrutteremo per scorrere aux e copiare in V[] */
    short i=Inizio, j=Mez+1, k=0, *Aux, y;
    //Alloco giusto i valori che andranno in aux (Fine-Inizio+1)
    Aux=malloc(sizeof(short)*(Fine-Inizio+1));
    /* _____ MERGE _____ */
    // <= e non < Perché' devo considerare anche Mez - Fine, che sarebbero N-1
    while(i<=Mez && j<=Fine) {
        if(V[i]<V[j])
            Aux[k++]=V[i++];
        else
            Aux[k++]=V[j++];
    }
    while(i<=Mez)
        Aux[k++]=V[i++];
    while(j<=Fine)
        Aux[k++]=V[j++];
    //Copia la porzione di array costruito A partire da inizio
    for(y=0; y<k; y++) //o <=Fine
        V[Inizio++]=Aux[y];
}

```

Quick Sort

Il secondo algoritmo di ordinamento della classe “Divide et Impera” è il Quick Sort. Esso, a differenza del Merge Sort, risulta essere un algoritmo che opera in-place.

L’idea è quella di partizionare, ad ogni passo, il vettore in due sottovettori separando questi ultimi da un elemento, l’elemento pivot (o partizionatore). Ad ogni passaggio, si confrontano col pivot gli altri elementi del vettore e si posizionano i minori alla sua sinistra ed i maggiori alla sua destra.

La descrizione formale dell’algoritmo può scandirsi come segue. Ad ogni passo, dopo aver scelto il pivot:

- si scorre il vettore da sinistra verso destra fino a trovare un elemento $a[i] > p$
- si scorre il vettore da destra verso sinistra fino a trovare un elemento $a[j] < p$

Se vengono trovati sia $a[i]$ che $a[j]$ allora si scambiano, altrimenti si scambia l’elemento trovato tra i due con il pivot.

Analisi della complessità

La **complessità di spazio** di questo algoritmo è $O(n)$ in quanto l’algoritmo opera in-place.

La **complessità di tempo** di questo algoritmo nel caso migliore è $O(n \log_2 n)$ mentre nel caso peggiore è $O(n^2)$, cioè diventa quadratico. La complessità di tempo varia in funzione del criterio con cui si sceglie il pivot ad ogni passo.

Il Quick Sort va utilizzato quando i dati sono potenza di 2 e quando il vettore è mediamente disordinato, perché se il vettore è già ordinato, il Quick Sort fornisce la peggiore prestazione e non va usato.

Scelta del partizionatore (pivot)

La complessità di tempo di questo algoritmo dipende dal criterio con cui viene scelto, ad ogni passo, il pivot. La situazione ideale sarebbe quella di scegliere l’elemento mediano del vettore, ma questa situazione non può essere “prevista”, soprattutto in caso di ordinamenti di migliaia o milioni di valori.

Normalmente, se si conosce la distribuzione di probabilità delle chiavi, si sfrutta questa conoscenza per la scelta del partizionatore. Se invece non si conosce la distribuzione di probabilità dei dati, si procede scegliendo in modo random il partizionatore, cioè si sceglie a caso una delle componenti del vettore. Nel nostro caso, scegliamo come pivot l’elemento dell’array all’estrema destra.

Codice

```
/* quick_sort: Se Inizio<Fine (quindi se si hanno almeno due elementi), calcolo il perno con
partiziona. Il pivot sarà il valore per cui i suoi valori alla destra sono maggiori e viceversa, quindi
andiamo ad ordinare da INIZIO a PERNO-1 (PORZIONE DI SINISTRA) e da PERNO+1 a FINE (PORZIONE
DI DESTRA) */
void quick_sort(short V[], short Inizio, short Fine) {
```



```

int pivot;
if (Inizio<Fine) {
    pivot = Partiziona(V, Inizio, Fine); //Trova il partizionatore
    quick_sort(V, Inizio, pivot-1); //Ordina la porzione a sinistra di partizionatore
    quick_sort(V, pivot+1, Fine); //Ordina la porzione a destra di partizionatore
}
}
/* PARTIZIONA: IL CUORE DEL PROGRAMMA.
CASO MIGLIORE  $O(n \log n)$ 
CASO PEGGIORE  $O(n^2)$ 
Funzionamento: Ha il compito di ordinare gli elementi in base al  $x = V[PIVOT]$  e disporre l'elemento
PIVOT nella sua reale posizione ordinata. Lo scopo è quello di ottenere la porzione alla sua destra
con valori maggiori o uguali di  $x$  e alla sua porzione di sinistra valori minori o uguali a  $x$ . */
short Partiziona(short V[], short Inizio, short Fine) {
    short Pivot=Fine; //Pivot sistematico
    short i=Inizio, j=Fine, x=V[Pivot]; /* j parte da fine e non fine-1!
/* Fin quando i e j non si incontrano */
    while(i<j) {
        /* Mentre  $V[i] \leq$  al valore Pivot, da sinistra a destra, va bene e avanza i; mentre  $V[j]$ 
 $\geq$  al valore Pivot, da destra a sinistra, va bene e arretra j */
        while(V[i] <= x && i < j)
            i++;
        while(V[j] >= x && i < j)
            j--;
        /* se vengono trovati sia  $a[i]$  che  $a[j]$  allora si scambiano */
        if(i < j) Scambia(&V[i], &V[j]);
    }
    /* Scambia il pivot nella sua posizione reale */
    Scambia(&V[Pivot], &V[j]);
    /* Ritorna il PARTIZIONATORE */
    return j;
}

```

Heap Sort

Il terzo algoritmo di ordinamento della classe “Divide et Impera” è l’Heap Sort. Esso, come il Quick Sort, risulta essere un algoritmo che opera in-place. Esso sfrutta la struttura dati heap per l’ordinamento, tipicamente, di un array di elementi.

Richiami sulla struttura dati Heap

L’heap non è altro che un albero binario quasi completo, in cui ogni nodo gode della proprietà heap, cioè ogni nodo ha un valore non inferiore rispetto a quello di tutti i nodi dei suoi sottoalberi.

Formalmente, la proprietà dell'heap è la seguente: se X è un qualsiasi nodo dell'heap (ad esclusione della radice) si ha $key(X) \leq key(padre(X))$. Da ciò ne consegue che il massimo dei valori memorizzati nell'heap si troverà nella radice.

L'operazione fondamentale che viene eseguita su una struttura dati heap è l'operazione heapify. Questa operazione non fa altro che calcolare il massimo tra i due figli di un nodo. Una volta fatto questo, confronta il nodo massimo tra i figli con la radice, e, infine, se necessario, scambia il nodo figlio col nodo padre in modo tale da ripristinare la proprietà dell'heap.

Idea sull'algoritmo

L'algoritmo può suddividersi in due fasi:

- nella prima viene costruito un heap a partire dai dati originali (cioè i dati dell'albero binario)
- nella seconda si riorganizza l'heap in modo tale da ordinare l'array

L'idea dell'algoritmo è quella di considerare l'array da ordinare come un albero binario quasi completo, dove la radice è la prima componente, il figlio sinistro il secondo elemento, il figlio destro il terzo elemento, e così via. In questo modo, l'albero binario non è un heap perché non verifica la proprietà dell'heap.

L'algoritmo di Heap Sort eseguirà una ricorsione:

- Heapify bottom-up fin quando si arriva alla radice
- Scambio tra la radice ($i = 1$) e l'ultimo elemento dell'array
- Modifica della dimensione n alla dimensione $n - 1$ [$n = n - 1$]

Ad ogni ripetizione si decrementa la dimensione dell'Heap perché così facendo otteniamo (dal basso verso l'alto) l'ordinamento dell'array. La **complessità di spazio** è $O(n)$, in quanto l'algoritmo opera in place. La **complessità di tempo** è $O(n \log_2 n)$.

Codice

```
/* heap-sort: Tenendo in considerazione "costruisci heap", dopo che ho ricavato l'heap, prendo il
primo elemento, ossia la RADICE che per definizione è il valore piu' grande di tutti e lo metto
all'ultimo posto, decrementando n. Quando rieffettuerò la costruzione dell'heap, la precedente
radice messa all'ultima posizione dell'array non sarà piu' considerata (per il n--) e si troverà già
nella sua posizione finale ordinata.*/
```

```
void heap_sort(short V[], short n) {
    short tmp;
    /* Finche' non ho un solo elemento (n>1 e non 0 perché l'heap o l'albero binario parte da
1)*/
    while(n>1) {
        costruisci_heap(V, n);
        /* Scambia la radice(valore massimo) e mettila nella posizione piu' bassa */
        tmp = V[n];
        V[n]=V[1];
        V[1]=tmp;
    }
}
```

```

        /* Ricostruisci l'heap, non considerando l'ultimo elemento messo in ordine */
        n--;
    }
}

/* costruisci heap: In questo caso, "i" punta ai padri. Heapify opera mediante i figli del padre[i]. Il
ciclo principale termina quando i=0 perché la radice deve essere considerata per l'heapify dato
che avrà dei figli. NB 'i' SCORRE I PADRI. */
void costruisci_heap(short heap[], short n) {
    /* Partiamo dal padre dell'ultima foglia perché le foglie per definizione già sono heap; i
    scorrerà i vari padri di cui controlleremo i relativi figli */
    short i=n/2;
    short esito, esito_foglia, j=0;
    /* Fin quando non si arriva oltre la radice */
    while(i>=1) {
        /* effettua un heapify tra heap[i] (il padre) e i figli ( heap[sx] e heap[dx] ) */
        esito = heapify(heap, i, n);
        j=esito; /* contengono l'indice di dove avviene lo scambio. Tale indice ci serve per
        scendere nel relativo sottoalbero perché avendo effettuato lo scambio, dobbiamo
        verificare se abbiamo violato nel sottoalbero le proprietà dell'heap. SE NON È AVVENUTO
        LO SCAMBIO, VIENE SEGNALATO DA ESITO CON -1 */
        /* Quindi se avviene uno scambio, effettuiamo le stesse operazioni nel sottoalbero
        in cui era avvenuto lo scambio. Cicliamo fin quando non abbiamo una foglia o non avviene
        uno scambio */
        while(esito>=0 && is_foglia(heap, j, n)) {
            esito = heapify(heap, j, n);
            j=esito;
        }
        i--; //passa ai padri successivi
    }
}

/* is_foglia: controlla semplicemente se quel nodo è una foglia */
short is_foglia(short heap[], short i, short n) {
    if(2*i>n && (2*i)+1>n)
        return 0; //non è foglia
    /* Altrimenti è una foglia se */
    return 1;
}

/* heapify: controllo se ci sono figli per padre (heap[i]) e inoltre mi trovo il più grande tra il padre e
i figli. */
short heapify(short heap[], short i, short n) {
    short tmp, scambio=0, figlio_sx=2*i, figlio_dx=(2*i)+1, maggiore;
    /* Controlla se sx non superi n e se il figlio sinistro sia maggiore del padre. Se sì, conservati
    l'indice maggiore, ossia il figlio, altrimenti se il padre è più grande, conserveremo l'indice del
    padre in maggiore*/
    if((figlio_sx<=n) && (heap[figlio_sx]>heap[i]))

```

```
        maggiore=figlio_sx;
    else
        maggiore=i;
    /* Ora confrontiamo il maggiore trovato con il figlio destro. Se la condizione è valida
    avremo un nuovo maggiore, altrimenti rimane invariato */
    if((figlio_dx<=n) && (heap[figlio_dx]>heap[maggiore]))
        maggiore=figlio_dx;
    /* Se il padre è piu' grande di uno dei 2 figli (quindi maggiore non e' l'indice padre) */
    if (maggiore != i) {
        /* Scambia i 2 nodi e ritorna l'indice di maggiore, ossia quello che ora contiene un
        valore minore di maggiore. tale ci serve per scendere nel relativo sottoalbero */
        scambia(&heap[maggiore], &heap[i]);
        return maggiore;
    }
    /* Il padre è un heap effettuando nessuno scambio */
    return -1;
}
```

C++

Operatori logici e bitwise

Variabili logiche in C++

In C++ esiste il tipo predefinito `bool` (tipo booleano o logico) con valori `{false, true}` corrispondenti ai valori interi `{0, 1}` e gli stessi operatori del C (`and`, `or`, `not`).

Per visualizzare "true" e "false" in C++ invece di 1 e 0, possiamo utilizzare uno dei format flag presenti nella libreria standard: `boolalpha`.

Esempio:

```
#include <iostream>
using namespace std;
int main()
{
    int x1=10, x2=20, m=2;
    bool b1, b2;
    b1 = x1 == x2; //false
    b2 = x1 < x2; //true
    cout << "b1 = " << boolalpha << b1 << "\n";
    cout << "b2 = " << boolalpha << b2 << "\n";
    bool b3 = true;
    int x3 = false + 5*m - b3;
    cout << x3 << boolalpha << "\n";
    return 0;
}
```

L'output sarà:

```
b1 = false
b2 = true
9
```

Il modo migliore per gestire bit in C++ è usare la classe template `bitset<N>`, dove `N` è noto al tempo della compilazione. La classe emula un array di bit, ma ottimizza l'allocazione di spazio: ogni elemento occupa solo un bit.

Esempio:

```
#include <iostream>
#include <bitset>
using namespace std;
int main()
{
    unsigned short N = 125;
    bitset<8> Bits (N); //oppure Bits = N;
    cout << "Bits = " << Bits << endl;
    bitset<8> Bits2 = 0x0F;
    cout << Bits << " & 00001111 = " << (Bits2 & Bits) << endl;

    cout << Bits << " << 4 = " << (Bits << 4) << endl;
    //uguale a Bits <=< 4 oppure Bits=Bits << 4
    return 0;
}
```

L'output sarà:

```
Bits = 01111101
01111101 & 00001111 = 00001101
01111101 << 4 = 11010000
```

Si può accedere ad ogni singolo bit di un `bitset<N>` mediante `bitset::reference`, come in un array. L'indice 0 corrisponde al bit meno significativo (più a destra), l'indice N-1 corrisponde al bit più significativo (più a sinistra).

Esempio:

```
#include <iostream>
#include <bitset>
#define N 4
using namespace std;
int main()
{
    bitset<N> Bits;
    Bits[1] = 1; //0010
    Bits[2] = Bits[1]; //0110
    cout << "Bits: " << Bits << '\n';
    int l = (int)Bits.to_ulong();
    cout << "l=Bits: " << l << endl;
    return 0;
}
```

L'output sarà:
 Bits = 0110
 l=Bits: 6

Stringhe

Per visualizzare un carattere a partire da una sottostringa si può usare `.at`

```
cout << "" << mystring[5] << endl;
```

Per confrontare le stringhe, al posto di utilizzare `strcmp` e operatori simili, bisogna semplicemente utilizzare gli operatori classici di confronto (`<`, `>`, `==`).

Tra le varie operazioni abbiamo:

- **insert**, dove si specifica la posizione di inserimento e la stringa da inserire [Esempio: `str1.insert(2, str2)`]
- **erase**, dove si specifica la posizione iniziale e il numero di caratteri [Esempio: `str1.erase(2, 3)`]
- **replace** [Esempio: `str1.replace(1, 2, str2);`]
- **find** [Esempio: `int i = str1.find("EF");`]
- **assign**, copia in `str2` la porzione di `str1` che inizia alla posizione `i+str2.length` e ha lunghezza `str1.length` caratteri [Esempio: `str2.assign(str1, i+str2.length(), str1.length())`]

La posizione dopo la fine della stringa è una costante chiamata `npos` (cioè -1).

Per l'input di stringhe utilizziamo `getline`, della libreria `std`.

Allocazione dinamica della memoria

In C i parametri vengono passati per riferimento (indirizzo). [Quindi con &]

Variabili puntatore

<code>Int *p1;</code>	puntatore a int
<code>Char **p2;</code>	puntatore a char*
<code>Float *p3[5]</code>	array di 5 float*
<code>Int (*fp)(char*);</code>	puntatore a funzione con parametro char* e valore di ritorno int
<code>Int *f(char*);</code>	funzione con parametro char* e valore di ritorno int*

In C **NULL** è la macro per indicare il puntatore nullo (un puntatore che non punta a niente). In C++ **nullptr** è la costante per indicare il puntatore nullo.

Variabili reference

Il C++, come il C, prevede la dichiarazione di:

- variabili mediante nome: **int v;**
- variabili puntatore: **int* pt; pt = &v;**

In aggiunta il C++ prevede anche la dichiarazione di **variabili reference**: **int& r = v;**

Da questo momento in poi **r** e **v** rappresentano lo stesso valore. L'istruzione **r++** equivale a **v++**, mentre **pt++** incrementa di **8 byte** l'indirizzo di memoria.

Un **reference**, quando dichiarato, deve puntare ad una variabile già dichiarata; quindi, la dichiarazione ne prevede anche l'inizializzazione. L'indirizzo cui punta una variabile **reference** non può essere cambiato.

Restrizioni:

- Non può esserci un **reference** a una variabile reference
- Non si può creare un array di **reference**
- Non si può creare un puntatore a un **reference**, cioè l'operatore indirizzo (&) non è applicabile ad un reference.
- I **reference** non sono consentiti per i campi di bit (in una struct).

Allocazione dinamica in C++

L'allocazione dinamica in C++ è gestita mediante gli operatori new e delete:

- **new** per allocare l'area dinamica (nella memoria heap);

- **delete** per la sua deallocazione.

Gli operatori **new** e **delete** sono come le funzioni **malloc()** e **free()** del C. Non esiste in C++ qualcosa come **realloc()**. L'alternativa è usare la classe **vector**.

Esempio: alloca dinamicamente un'area per un intero int (puntata da pt) e vi assegna il valore 7

```
int* pt = new int;
cout << " *pt = " << *pt << endl;
*pt = 7; //oppure si poteva fare int* pt = new int (7);
cout << " *pt = " << *pt << endl;
delete pt;
```

Esempio: alloca dinamicamente un'area per 100 interi (puntata da pt). Se l'allocazione non avviene, new restituisce nullptr.

```
int * pt = new int[100];
if (pt == nullptr)
    //errore...
delete [] pt;
```

Un **dangling pointer** è un puntatore (non nullo) che punta a dati non più validi, cioè punta ad un'area di memoria deallocata (dopo aver fatto delete, il puntatore e il suo contenuto non sono modificati, ma l'area di memoria viene deallocata).

La classe vector

I vector sono contenitori sequenziali per rappresentare array che possono cambiare size durante l'esecuzione, a differenza degli array allocati dinamicamente con new.

Operazioni sui vector:

- Accedere a qualunque elemento
- Aggiungere o eliminare elementi dovunque

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector<int> v = {7, 5, 16, 8};
    v.pop_back(); //elimina l'ultimo elemento
    v.push_back(25); //inserisce 25 in coda
    v.push_back(13); //inserisce 13 in coda
    for (auto &n : v) //range-based for loop
        cout << n << endl;
    vector<int> v2;
    v2 = v; //copia v in v2
    return 0;
}
```


Classi e oggetti

Alla base della programmazione orientata agli oggetti (Object Oriented Programming) c'è l'oggetto. Oggetti che hanno le stesse caratteristiche sono descritti da una stessa classe.

Un oggetto è caratterizzato da un suo stato, determinato dai valori di alcuni attributi, proprietà (i campi di una classe) e da certi comportamenti, azioni (i metodi di una classe). La programmazione ad oggetti descrive le interazioni tra oggetti.

Esempio

Di solito le variabili che descrivono lo stato sono locali e non sono visibili al di fuori dell'oggetto (data hiding).

Il comportamento è controllato da funzioni che agiscono sulle variabili di stato e si interfacciano con l'esterno (encapsulation)

In C++ la classe costituisce la base per la programmazione orientata agli oggetti (Object Oriented Programming). La classe è usata per definire la natura di un oggetto ed è l'unità base di incapsulamento del C++. Una classe ha un proprio nome e contiene due tipi di membri: campi e metodi.

Dichiarazione di classe

```
class class_name {  
    //private data (campi) and functions (metodi)  
    access_specifier:  
        //data and functions  
    access_specifier:  
        //data and functions  
} object_list;
```

Access specifier

- **public**, se si tratta di funzioni e dati accessibili da altre parti del programma
- **private**, se si tratta di funzioni e dati accessibili solo all'interno della classe
- **protected**, quando si tratta di funzioni e dati accessibili dalla classe e dalle sue sottoclassi

La differenza tra classi e struct è che per default, tutti i membri di una classe sono privati mentre quelli di una struct sono pubblici.

I membri sono accessibili mediante il punto "." oppure "->"

Esempio: (objectID.memberID) oppure (objectID.classID::memberID)

Access specifier (modificatori di accesso)

access specifier	public	protected	private
stessa classe	SI	SI	SI
classe derivata	SI	SI	NO
altre classi	SI	NO	NO

Esempio

Si vuole gestire un array di studenti memorizzati con i seguenti campi: matricola (10 **char**), cognome e nome (50 **char**), voto esame (**unsigned int**). Si andranno inoltre ad effettuare operazioni come l'inserimento dei dati delle studente e la visualizzazione degli stessi.

File studente.hpp (specifica di classe)

```
class Studente {
    char matr[11]; //10+terminatore
    char nome[51]; //50+terminatore
    unsigned int voto;

    public:
        void inserisci_dati(const char *M, const char *N, unsigned int V);
        void visualizza_dati();
};
```

File Studente.cpp (implementazione della classe)

```
#include <iostream>
#include <cstring>
#include "Studente.hpp"
using namespace std;
void Studente::inserisci_dati(const char *M, const char *N, unsigned int V) {
    strcpy(matr, M);
    strcpy(nome, N);
    voto = V;
}
void Studente::visualizza_dati() {
    cout << "matr.: " << matr << "\tnome:" << nome << "\tvoto: " << voto << endl;
}
```

File esempio1.cpp

```
#include "Studente.hpp"
#define NumStud 3

int main() {
    Studente E [NumStud];
```

```
E[0].inserisci_dati("0124001233", "Bianchi Aldo", 26);
E[1].inserisci_dati("0124001343", "Rossi Maria", 28);
E[2].inserisci_dati("0124001345", "Verdi Marco", 24);
```

```
for (int i=0; i < NumStud; i++)
    E[i].visualizza_dati();

return 0;
}
```

Output

```
matr.: 0124001233  nome:Bianchi Aldo  voto:26
matr.:0124001343  nome:Rossi Maria  voto:28
matr.:0124001345  nome:Verdi Marco  voto:24
```

Costruttori e distruttori

In una classe un costruttore è un metodo con lo stesso nome della classe che viene chiamato ogni volta che viene istanziato un oggetto della classe.

Un costruttore può essere usato per inizializzare i dati della classe. Per garantire che la memoria allocata dal costruttore sia deallocata c'è il distruttore, che ha sempre lo stesso nome della classe ma preceduto dal carattere "~" (tilde).

Il costruttore e il distruttore non hanno valore di ritorno (neanche **void**). Il distruttore non si può invocare da programma, è invocato dal compilatore quando viene deallocato lo spazio di memoria assegnato all'oggetto.

Costruttore default

Se il programmatore non prevede alcun costruttore, il compilatore comunque crea il costruttore default. Se il programmatore definisce un costruttore particolare, allora, se richiesto, va definito anche il costruttore default.

Esempio costruttore con parametro

```
#include <iostream>
using namespace std;
class myclass {
public:
    int who;
    myclass(int id);
    ~myclass();
} glob_ob1(1), glob_ob2(2);

int main() {
    myclass local_ob0; //darà un errore: no
                       //matching function for call to
                       //'myclass::myclass()'
    myclass local_ob1(3);
}
```

Esempio costruttore default

```
#include <iostream>
using namespace std;
class myclass {
public:
    int who;
    myclass() {};
    myclass(int id);
    ~myclass();
} glob_ob1(1), glob_ob2(2);

int main() {
    myclass local_ob0; //no error
    myclass local_ob1(3);
    cout << "This will not be first line displayed. \n";
    myclass local_ob2(4);
    return 0;
}
```

Costruttore di copia

In una classe il costruttore di copia crea un oggetto a partire da un altro oggetto della classe (i valori delle variabili membro vengono copiati). La sintassi dei costruttori di copia è la seguente:

```
class Myclass {
    ...
    Myclass(const Myclass &object);
    ...
}
```

Function overloading

Function overloading indica la possibilità di usare lo stesso nome per due o più funzioni nella stessa classe. Ciò è possibile solo quando il compilatore è in grado di stabilire quale funzione chiamare e quindi le ridefinizioni di una funzione devono usare:

- tipi differenti di parametri;
- numero diverso di parametri.

```
int f(int i);
double f(double i); //tipi differenti di parametri
int f(int i, int j); //numero diverso di parametri
```

Ereditarietà

Una classe può essere derivata da un'altra classe: si instaura una gerarchia tra la superclasse (o classe base) e la sottoclasse (o classe derivata), che aggiunge dettagli. La sintassi è la seguente:

```
class nomeClasseDerivata : access nomeClasseBase {
    //corpo della classe derivata
};
```

tipo di ereditarietà	membri della classe base	membri ereditati in classe derivata
public	public protected private	public protected inaccessibile
protected	public protected private	protected protected inaccessibile
private	public protected private	private private inaccessibile

Costruttori, distruttori, metodi virtuali non sono ereditati in una classe derivata. Se necessari, questi vanno creati.

Upcasting

Un puntatore ad una classe base può anche essere utilizzato come puntatore ad un oggetto di una qualunque classe derivata da tale classe base. **Puntando ad un oggetto della classe derivata con un puntatore alla classe base si può accedere solo ai membri della classe base che sono stati ereditati dalla classe derivata.**

Non vale il viceversa: un puntatore ad una classe derivata non può puntare ad un oggetto della classe base.

Esempio

File classi

```
class Base {
    protected:
        int i;
    public:
        void set_i(int a) {i=a;}
        int get_i() {return i;}
};
class Derived : public Base {
    int j;
    public:
        void set_j(int a) {i=a;}
        int get_j() {return i;}
};
```

File main

```
#include <iostream>
#include "es5.hpp"
using namespace std;
int main() {
    Base *bp;
    Derived d;
    bp = &d;

    //OK
    bp -> set_i(10);
    cout << bp -> get_i() << endl;

    //ERRORE!
    bp -> set_j(1);
    cout << bp -> get_j() << endl;

    return 0;
}
```

La correzione è il seguente cast:

```
((Derived *)bp) -> set_j(1);
cout << ((Derived *)bp) -> get_j() << endl;
```

Virtual functions e polymorphism

Una **funzione virtuale** è una funzione membro dichiarata in una classe base e ridefinita in una classe derivata. Le funzioni virtuali sono lo strumento principale del polimorfismo ed implementano la filosofia "una sola interfaccia, più metodi".

File classi

```
#include <iostream>
```

```
using namespace std;
```

```
class base {
    public:
        virtual void vfunc() {
            cout << "metodo vfunc() della classe base\n";
        }
};

class derived1 : public base {
    public:
        void vfunc() {
            cout << "metodo vfunc() della classe derived1\n";
        }
};

class derived2 : public base {
    public:
        void vfunc() {
            cout << "metodo vfunc() della classe derived2\n";
        }
};
```

File main

```
int main() {
    base *p, b; //p è un puntatore ad un oggetto della classe base
    derived1 d1;
    derived2 d2;

    p = &b;
    p -> vfunc();

    p = &d1; //p punta ad un oggetto della classe derivata
    p -> vfunc();

    p = &d2;
    p -> vfunc();
}
```

Output

```
metodo vfunc() della classe base
metodo vfunc() della classe derived1
metodo vfunc() della classe derived2
```

Overload vs override

Function overloading (*compile time polymorphism*) indica la possibilità di usare lo stesso nome per due o più funzioni nella stessa classe. Ciò è possibile solo quando la "**signature**" (tipo e numero dei parametri) delle funzioni è diversa.

Esempio

```
int f(int i);
double f(double i);
int f(int i, int j);

int main() {
    int h; double hh;
    h = f(10);
    h = f(10, 20);
    hh = f(12.34);
    return 0;
}
```

Function overriding (*run-time polymorphism*) indica la possibilità per una classe derivata di ridefinire una funzione della superclasse con la stessa "**signature**", cioè eguale tipo dei parametri e del valore di ritorno.

Esempio

```
int main() {
    base *p, b;
    derived1 d1;
    derived2 d2;

    p = &b;
    p -> vfunc();

    p = &d1;
    p -> vfunc();

    p = &d2;
    p -> vfunc();
}
```

I costruttori possono essere "overloaded", ma non possono essere "overriding".

Function overload vs function override

- Ereditarietà: l'overriding di funzioni si verifica solo quando una classe eredita da un'altra classe, mentre l'overloading può avvenire anche senza ereditarietà.

- Function Signature: le funzioni "sovraccaricate" ("overloaded") devono differire in "signature", cioè o nel numero dei parametri oppure nel tipo dei parametri; invece, nell' "overriding" la "signature" delle funzioni deve essere la stessa.
- "Scope" delle funzioni: le funzioni "overridden" hanno "scope" (ambiti di visibilità) differenti, mentre le funzioni "overloaded" hanno lo stesso "scope".
- Comportamento delle funzioni: l'overriding è necessario quando una funzione della classe derivata deve avere qualche funzionalità aggiuntiva oppure diversa da quella della classe base. L'overloading è usato per avere funzioni con lo stesso nome che si comportano in modo diverso in dipendenza dei parametri passati ad esse.

Pure virtual function e abstract class

Una **funzione virtuale pura** è una funzione virtuale che non ha una definizione nella classe base, la quale viene detta classe base astratta. Una classe è **astratta** se contiene almeno una funzione virtuale pura.

La sintassi per dichiarare una funzione virtuale pura è:

```
virtual type func-name(parameter-list) = 0;
```

In tal caso ciascuna classe derivata deve prevedere la sua propria definizione altrimenti verrà considerata anch'essa classe astratta. Se la classe derivata non riesce a sovrascrivere la funzione virtuale pura, si verificherà un errore in fase di compilazione. Non si può istanziare un oggetto di una classe astratta.

Esempio

Classi

```
class number {
    protected:
        int val;
    public:
        void setval(int i) {val = i;}
        virtual void show() = 0;
};

class hextype : public number {
    public:
        void show() {
            cout << "hex.: "; << hex << val << "\n";
        }
};

class dectype : public number {
    public:
        void show() {
            cout << "dex.: " << val << "\n";
        }
};

class octtype : public number {
    public:
```



```
void show() {  
    cout << "oct.: " << oct << val << "\n";  
}  
};
```

Main

```
int main() {  
    dectype d;  
    hextype h;  
    octtype o;  
  
    d.setval(20);  
    d.show();  
  
    h.setval(20);  
    h.show();  
  
    o.setval(20);  
    o.show();  
  
    return 0;  
}
```

```
Output  
dec.: 20  
hex.: 14  
oct.: 24
```

File e Stream

Il C++ supporta due File System completi: uno ereditato dal C e l'altro (proprio del C++) orientato agli oggetti.

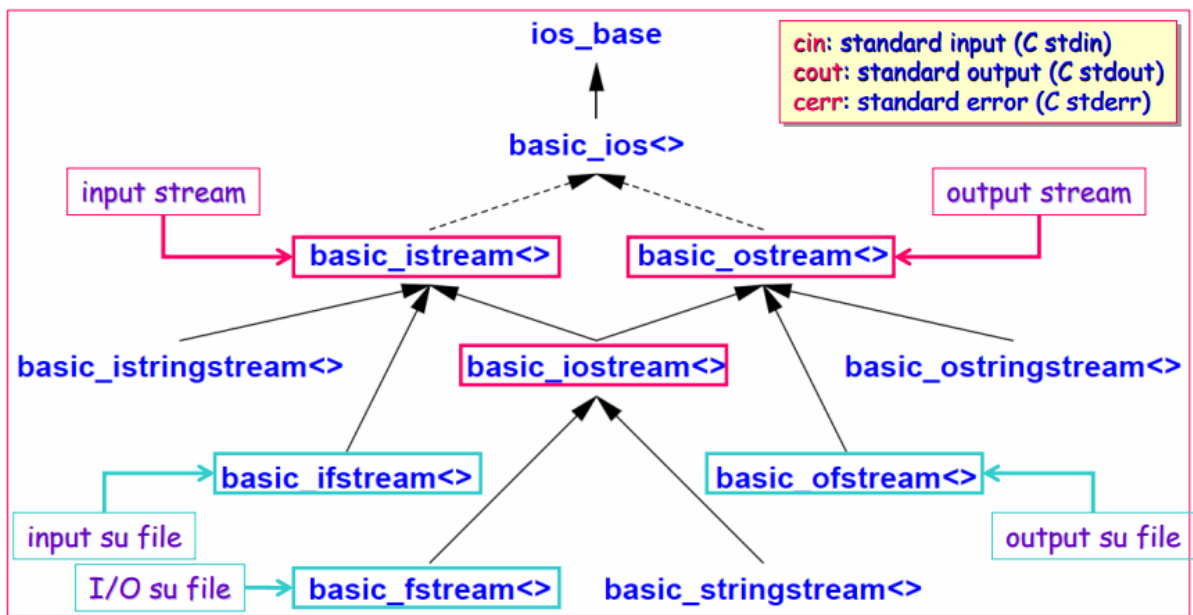
Il termine **stream** indica un generico flusso logico di dati (astrazione), sotto forma di byte, indipendente dal device coinvolto nel trasferimento (terminale, disk drive, tape drive, ...). È compito del File System gestire i device associati a stream.

Text stream: una sequenza di caratteri

Binary stream: una sequenza di byte sui quali non avviene alcuna conversione in caratteri.

Il termine **file** è invece legato al device (come, ad esempio, un disk file); gli viene associato uno stream mediante l'operazione di apertura, mentre l'operazione di chiusura rimuove tale collegamento. In funzione del device il file può supportare un indicatore di posizione (per i disk drive sì, per una stampante no!).

Gerarchia delle classi I/O Stream



Un **istream** può essere associato ad un input device (esempio: tastiera), un file o una stringa.

Un **ostream** può essere associato ad un output device (finestra di testo), un file o una stringa.

I/O formattato

adjustfield	basefield	boolalpha	dec
fixed	floatfield	hex	internal
left	oct	right	scientific
showbase	showpoint	showpos	skipws
unitbuf	uppercase		

basefield: oct, dec, hex

floatfield: scientific, fixed

adjustfield: left, right, internal

Funzioni membro

streamsize width(**streamsize** w);

ampiezza del campo

streamsize precision(**streamsize** p);

numero di cifre da visualizzare

char fill(**char** ch);

riempimento del campo con un carattere

Manipolatori

Manipulator	Purpose	Input/Output
boolalpha	Turns on boolalpha flag	Input/Output
dec	Turns on dec flag	Input/Output
endl	Output a newline character and flush the stream	Output
ends	Output a null	Output
fixed	Turns on fixed flag	Output
flush	Flush a stream	Output
hex	Turns on hex flag	Input/Output
internal	Turns on internal flag	Output
left	Turn on left flag	Output
noboolalpha	Turns off boolalpha flag	Input/Output
noshowbase	Turns off showbase flag	Output
noshowpoint	Turns off showpoint flag	Output
noshowpos	Turns off showpos flag	Output

Per usare i manipolatori che accettano un parametro, bisogna includere `<iomanip>`.

File in C++

Prima di aprire un file bisogna creare uno stream. Esistono 3 tipi di stream:

Input: `ifstream` in;

Output: `ofstream` out;

Input/Output: `fstream` io;

In C++ si apre un file collegandolo ad uno stream. La funzione `open()` è un metodo della classe `ifstream`, `ofstream` oppure `fstream`. I suoi prototipi sono:

```
void ifstream::open(const char * filename, ios::openmode mode = ios::in);  
void ofstream::open(const char *filename, ios::openmode mode = ios::out);  
void fstream::open(const char *filename, ios::openmode mode = ios::in | ios::out);
```

Altri openmode

ios::app	append
ios::ate	seek end of file
ios::binary	file binario
ios::in	per leggere
ios::out	per scrivere
ios::trunc	sovrascrive file esistente

Esempi: sono equivalenti

```
ofstream out; out.open("test", ios::out);  
ofstream out; out.open("test");  
ofstream out("test");
```

Le classi ifstream, ofstream, fstream hanno costruttori che automaticamente aprono il file nella modalità default.

Esempio: scrittura di un file di caratteri

```
#include <iostream>  
#include <fstream>  
using namespace std;  
int main() {  
    //crea lo stream di output e gli associa il file  
    ofstream out("mystream.txt");  
  
    //controlla se il file è stato aperto  
    if (!out) {  
        cerr << "Errore in apertura file\n";  
        exit(1);  
    }  
  
    //scrive il file  
    out << "Nel mezzo del cammin di nostra vita\n";  
    out << "mi ritrovai per una selva oscura\n";  
    out << "che la diritta via era smarrita!\n";  
  
    //chiude il file
```

```
    out.close();  
    return 0;  
}
```

Esempio: lettura di un file di caratteri

```
#include <iostream>  
#include <fstream>  
#include <string>  
using namespace std;  
int main() {  
    //crea lo stream di input e gli associa il file  
    ifstream in("mystream.txt");  
  
    //controlla se il file è stato aperto  
    if (!in) {  
        cerr << "Errore in apertura file\n";  
        exit(1);  
    }  
  
    //legge dal file finché non è finito  
    string str;  
    while (in) {  
        getline(in, str);  
        cout << str << endl;  
    }  
  
    //chiude il file  
    in.close();  
    return 0;  
}
```

Template

In C++ è possibile definire delle funzioni o classi generiche che abbiano come parametro il tipo di dato, potendo usare lo stesso codice per tipi differenti. In tal modo il C++ fornisce un metodo per creare un polimorfismo parametrico. Una funzione o classe generica si chiama **funzione o classe template**.

Esempio

```
#include <iostream>  
#include <vector>  
using namespace std;  
int main() {  
    vector<int> v = {7, 5, 16, 8};  
    v.pop_back();  
}
```

```

    v.push_back(25);
    v.push_back(13);
    for (auto n : v)
        cout << n << endl;
    vector<int> v2;
    v2 = v;

    return 0;
}

```

La classe vector è un template

Sintassi e uso dei Template

Funzione generica

```

template<class Ttype> ret_type func_name(parameter list)
oppure
template<typename Ttype> ret_type func_name(parameter list) {
    //corpo della funzione
}

```

Classe generica

```

template<class Ttype> class class_name
oppure
template<typename Ttype> class class_name {
    //definizioni di classe
}

```

Per istanziare un oggetto di classe generica:

```
class_name <type> ob;
```

Il compilatore automaticamente genera il codice corretto per il tipo di dato usato nella chiamata alla funzione (**overloading automatico**).

La libreria Standard del C++ (STL - Standard Template Library), progettata dallo stesso inventore del C++, mette a disposizione numerose **classi template**, alcune delle quali per strutture dati di base, e molte funzioni generiche che implementano algoritmi fondamentali.

In particolare, per le strutture dati dinamiche lineari ci sono (fra altre) le classi template (ADT sta per Abstract Data Type):

- **stack** per la ADT pila;
- **queue** per la ADT coda;
- **forward_list** per la ADT lista lineare;
- **list** per la ADT bidirezionale.

Namespace e Using

Si è visto che per "implementare" un metodo al di fuori della sua classe bisogna usare:

`nome_classe::nome_metodo`

Il simbolo `::` è chiamato **scope resolution operator**. Esso informa il compilatore che tale metodo appartiene alla classe specificata, in questo caso `nome_classe` (quindi è nello scope di `nome_classe`).

Il **namespace** è una regione dichiarativa il cui obiettivo è "localizzare" i nomi degli identificatori evitando collisioni di nome. Gli elementi dichiarati in un namespace sono separati dagli elementi dichiarati in un altro namespace.

La sintassi per creare un namespace è:

```
namespace nome {  
    //dichiarazioni  
}
```

L'uso di **using** ha due forme generali:

```
using namespace nome;  
using nome::member;
```

La prima si riferisce al namespace che si vuole usare e serve per semplificare il riferimento agli elementi più frequentemente usati del namespace.

La seconda fa riferimento ad un singolo membro del namespace.

Esempio

```
#include <iostream>  
using std::cout; //lo usiamo solo per cout  
int main() {  
    cout << "sizeof(int) = " << sizeof(int) << std::endl;  
    return 0;  
}
```

Overload di operatori

In una classe si possono definire degli operatori (+, *, >, ...) che sostituiscono quelli default del C++. Una funzione operatore, membro di una classe, è definito come:

`return_type class_name::operator#(arg_list)`

Il simbolo `#` indica l'operatore che si vuole ridefinire.

Member Initializer List

La lista di inizializzazione dei membri è una caratteristica propria dei costruttori di classe. Essa appare sempre tra la lista di argomenti del costruttore e il suo corpo ed è preceduta dai ":".

Esempio

```
class Complex {
    private:
        double Re;
        double Im;
    public:
        Complex(double, double);
        //altro
};

Complex::Complex(double a, double b) : Re(a), Im(b) {
    //che equivale al costruttore con parametri
}
```

Essa va usata per:

- Inizializzare i dati membro costanti non statici

```
#include <iostream>
using namespace std;
class Test {
    const int t;
    public:
        Test(int i) : t(t) {}
        int getT() {return t;}
};

int main () {
    Test t1(10);
    cout << t1.getT() << endl;
    return 0;
}
```

Output
10

- Inizializzare i dati membro reference

```
#include <iostream>
using namespace std;
class Test {
    int &t; //reference
    public:
        Test(int &t) : t(t) {}
        int getT() {return t;}
};

int main() {
    int x = 20;
    Test t1(x);
}
```



```

    cout << t1.getT() << endl;
    x = 30;
    cout << t1.getT() << endl;
    return 0;
}

```

Output
20
30

- Inizializzare gli oggetti membro che non hanno un costruttore default

```

#include <iostream>
using namespace std;
class A {
    int i;
    public:
        A(int); //costruttore con parametro ci A
};
A::A(int arg) {
    i = arg;
    cout << "costruttore con parametro di A: i=" << i << endl;
}
class B {
    A a; //l'oggetto a è membro della classe B
    public:
        B(int); //costruttore con parametro di B
};
B::B(int x) : a(x) {
    cout << "costruttore con parametro di B" << endl;
}
int main() {
    B obj(10);
    return 0;
}

```

Output
costruttore con parametro di A: i = 10
costruttore con parametro di B

- Inizializzare i dati membro della classe base

```

#include <iostream>
using namespace std;
class A {
    int i;
    public:
        A(int); //costruttore con parametro ci A
};
A::A(int arg) {
    i = arg;
    cout << "costruttore con parametro di A: i=" << i << endl;
}
class B : public A {
    public:
        B(int); //costruttore con parametro di B
};
B::B(int x) : A(x) {
    //chiama il costruttore di A
}

```

```

        cout << "costruttore con parametro di B" << endl;
    }
    int main() {
        B obj(10);
        return 0;
    }

```

Output

```

costruttore con parametro di A: i = 10
costruttore con parametro di B

```

- Quando il nome del parametro del costruttore è lo stesso del dato membro

```

#include <iostream>
using namespace std;
class A {
    int i;
public:
    A(int ); //costruttore con parametro di A
    int getI() {return i;}
};
A::A(int i) : i(i) {
    // in alternativa A::A(int i) {this->i = i;}
}
int main() {
    A a(15);
    cout << a.getI() << endl;
}

```

Output

```

15

```